University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Tomasz Kociumaka

Efficient Data Structures for Internal Queries in Texts

PhD dissertation

Supervisor

prof. dr hab. Wojciech Rytter
Institute of Informatics
University of Warsaw

October 2018

Author's declaration:
I hereby declare that this dissertation is my own work.

October 26, 2018 . . . .

Tomasz Kociumaka

Supervisor's declaration:
The dissertation is ready to be reviewed

October 26, 2018 . . . .

prof. dr hab. Wojciech Rytter

Abstract

This thesis is devoted to *internal queries* in texts, which ask to solve classic text-processing problems for substrings of a given text. More precisely, the task is to preprocess a static string T of length n (called the *text*) and construct a data structure answering certain questions about the substrings of T. The substrings involved in each query are specified in constant space by their occurrences in T, called *fragments* of T, identified by the start and the end positions. Components for internal queries often become parts of more complex data structures, and they are used in many algorithms for text processing.

Longest Common Extension Queries, asking for the length of the longest common prefix of two substrings of the text T, are by far the most popular internal queries. They are used for checking if two fragments match (represent the same string) and for lexicographic comparison of substrings. Due to an optimal solution in the standard setting of texts over polynomially-bounded integer alphabets, with $\mathcal{O}(1)$-time queries, $\mathcal{O}(n)$ size, and $\mathcal{O}(n)$ construction time, they have found numerous applications across stringology. In this dissertation, we provide the first optimal data structure for smaller alphabets of size $\sigma \ll n$: it handles queries in $\mathcal{O}(1)$ time, takes $\mathcal{O}(n/\log_\sigma n)$ space, and admits an $\mathcal{O}(n/\log_\sigma n)$-time construction (from the *packed* representation of T with $\Theta(\log_\sigma n)$ characters in each machine word).

We then go back to alphabets of size σ polynomial in n and focus on more complex internal queries. Our first data structure supports Internal Pattern Matching Queries, which ask for the occurrences of one substring x within another substring y. After $\mathcal{O}(n)$-time preprocessing of the text T, it answers these queries in time proportional to the quotient $|y|/|x|$ of substrings' lengths, which is required due to the information content of the output. We also use this data structure for Period Queries, asking for the periods of a given substring. Here, our logarithmic query time is also optimal by a similar information-theoretic argument.

Further data structures are designed for Minimal Suffix and Minimal Rotation Queries, asking to compute the lexicographically smallest non-empty suffix and cyclic rotation of a given substring, respectively. They are answered in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time preprocessing. We also consider a more general problem of simulating the suffix array of a given substring (Substring Suffix Selection Queries, asking for the kth lexicographically smallest suffix of a substring) and its inverse suffix array (Substring Suffix Rank Queries, asking for the lexicographic rank of a substring's suffix). Our data structure supports these queries in $\mathcal{O}(\log n)$ time, takes $\mathcal{O}(n)$ space, and can be constructed in $\mathcal{O}(n\sqrt{\log n})$ time.

The tools developed in this dissertation additionally yield improved results for several kinds of Substring Compression Queries, which ask for the compressed representation of a given substring obtained using a specific method; we consider schemes based on the Lempel–Ziv parsing and the Burrows–Wheeler transform.

Our results combine text-processing tools with combinatorics on words and state-of-the-art general-purpose data structures. The key technical contribution is a novel locally consistent symmetry-breaking scheme, formalized in terms of *synchronizing functions*, which is central to our solutions for Longest Common Extension Queries and Internal Pattern Matching Queries.

2012 ACM Subject Classification: Theory of computation → Pattern matching

Keywords: Data structures, Longest common extension, Longest common prefix, Local consistency, Minimal Suffix, Period, Substring compression, Substring queries

Streszczenie

Rozprawa dotyczy *zapytań wewnętrznych* w tekstach, czyli rozwiązywania klasycznych problemów algorytmiki tekstów dla podsłów danego słowa. Formalnie, zadanie polega na wstępnym przetworzeniu statycznego słowa T długości n (*tekstu*), tak aby skonstruować strukturę danych odpowiadającą na pewne pytania dotyczące jego podsłów. Podsłowa te są identyfikowane w stałej pamięci za pomocą wystąpień w T, zwanych *fragmentami* tekstu i reprezentowanych przez pozycje początkową oraz końcową. Komponenty dla zapytań wewnętrznych są używane w wielu algorytmach tekstowych i wchodzą w skład bardziej rozbudowanych struktur danych.

Zapytania o długość najdłuższego wspólnego prefiksu dwóch podsłów tekstu T stanowią zdecydowanie najpowszechniejszy problem tego rodzaju. Używa się ich między innymi do sprawdzania, czy dwa fragmenty pasują do siebie (reprezentują to samo słowo), i do porównywania podsłów w porządku leksykograficznym. Mnogość zastosowań wynika z istnienia optymalnego rozwiązania w standardowym modelu tekstów (nad alfabetem złożonym z liczb naturalnych wielkości wielomianowej ze względu na n), czyli struktury danych wielkości $\mathcal{O}(n)$, która odpowiada na zapytania w czasie $\mathcal{O}(1)$ i posiada procedurę konstrukcji w czasie $\mathcal{O}(n)$. W rozprawie przedstawiamy pierwszą optymalną strukturę danych dla małych alfabetów rozmiaru $\sigma \ll n$: strukturę wielkości $\mathcal{O}(n/\log_\sigma n)$ charakteryzującą się czasem zapytania $\mathcal{O}(1)$ i czasem konstrukcji $\mathcal{O}(n/\log_\sigma n)$ (na podstawie *spakowanej* reprezentacji tekstu, gdzie każda komórka pamięci zawiera $\Theta(\log_\sigma n)$ symboli).

W dalszej części rozprawy wracamy do tekstów nad alfabetem wielkości wielomianowej ze względu na n i skupiamy się na bardziej złożonych zapytaniach wewnętrznych. Nasza pierwsza struktura danych rozwiązuje problem wewnętrznego wyszukiwania wzorca, czyli zapytań o wystąpienia jednego podsłowa x w obrębie drugiego podsłowa y. Po przetworzeniu tekstu T w czasie $\mathcal{O}(n)$, odpowiada ona na takie zapytania w czasie proporcjonalnym do ilorazu $|y|/|x|$ długości podsłów, który jest wymagany ze względu na pesymistyczną złożoność informacji zawartej w wyniku. Tej samej struktury danych używamy także do zapytań o wszystkie okresy wskazanego podsłowa; uzyskany tutaj czas logarytmiczny również jest optymalny.

Kolejne struktury danych stworzone zostały do obsługi zapytań o najmniejszy leksykograficznie sufiks i najmniejszą leksykograficznie rotację cykliczną wskazanego podsłowa. Odpowiadamy na nie w czasie $\mathcal{O}(1)$ po przetworzeniu tekstu w czasie $\mathcal{O}(n)$. Rozważamy także bardziej ogólne pytania o k-ty leksykograficznie sufiks podsłowa (czyli o k-ty elementy tablicy sufiksowej podsłowa) oraz o leksykograficzną rangę wskazanego sufiksu podsłowa (czyli element odwrotnej tablicy sufiksowej). Nasza struktura danych przetwarza zapytania obydwu rodzajów w czasie $\mathcal{O}(\log n)$, zajmuje $\mathcal{O}(n)$ komórek pamięci i można ją zbudować w czasie $\mathcal{O}(n\sqrt{\log n})$.

Stworzone w rozprawie narzędzia dają także lepsze od znanych dotychczas rozwiązania problemu kompresji podsłów, czyli zapytań o reprezentację podsłowa według ustalonego algorytmu kompresji; rozważamy w tym kontekście metody oparte na parsowaniu Lempela-Ziva i transformacie Burrowsa-Wheelera.

Rezultaty rozprawy łączą narzędzia do przetwarzania tekstów z kombinatoryką słów i najnowszymi osiągnięciami w dziedzinie abstrakcyjnych struktur danych. Naszą kluczową technikę stanowi innowacyjna metoda lokalnie zgodnego łamania symetrii między pozycjami tekstu, która stoi za nowymi rozwiązaniami dla zapytań o długość najdłuższego wspólnego prefiksu i dla wewnętrznego wyszukiwania wzorca.

Tytuł rozprawy w języku polskim: Efektywne struktury danych dla zapytań wewnętrznych w tekstach

Słowa kluczowe: struktury danych, najdłuższy wspólny prefiks, lokalna zgodność, najmniejszy sufiks, okres, kompresja podsłów, zapytania o podsłowa

Acknowledgements

First, I wish to thank Wojciech Rytter, my advisor, for introducing me to text algorithms and guidance over the course of my undergraduate and graduate studies. I am also indebted to Jakub Radoszewski, who was the first to expose me to research problems and since then has become my closest collaborator. My further gratitude goes to Tomasz Waleń, with whom we have created a successful team at the University of Warsaw.

Special thanks must be made to Paweł Gawrychowski, to whom I owe some of the most interesting problems I have been working on, as well as several short research visits in Saarbrücken, Haifa, and Wrocław. I am also obliged to Tatiana Starikovskaya and Maxim Babenko, collaboration with whom on Minimal Suffix Queries inspired most of the directions I followed preparing this thesis. I particularly appreciate numerous conversations with Tatiana and her warm welcome during my short stays in Moscow.

Listed above are the coauthors of the papers that this dissertation is based on. However, a few other people have also made a significant impact on its contents. I have greatly benefited from Ely Porat, who has invited me to Bar-Ilan University and offered me an early access to his results on local consistency and LCE Queries [24]. I wish to thank Moshe Lewenstein for drawing my attention to Generalized LZ Substring Compression Queries, as well as Djamal Belazzougui, Travis Gagie, and Simon J. Puglisi, who suggested applications related to the Burrows–Wheeler transform. I am also grateful to Dominik Kempa for proofreading the previously unpublished Chapter 4 of this dissertation.

Due to my extensive research in stringology and a few episodes in various other areas of algorithm design, I had the pleasure to work with many people. Here, I wish to thank all my co-authors: Anna Adamaszek, Mai Alzamel, Maxim Babenko, Golnaz Badkobeh, Evangelos Bampas, Hideo Bannai, Carl Barton, Anudhyan Boral, Panagiotis Charalampopoulos, Raphaël Clifford, Maxime Crochemore, Marek Cygan, Jerzy Czyżowicz, Gabriele Fici, Johannes Fischer, Tomáš Flouri, Travis Gagie, Michał Gańczorz, Paweł Gawrychowski, Leszek Gąsieniec, Garance Gourdel, Szymon Grabowski, Fabrizio Grandoni, David Ilcinkas, Costas S. Iliopoulos, Shunsuke Inenaga, Artur Jeż, Adam Karczmarz, Ralf Klasing, Ignat Kolesnichenko, Dmitry Kosolobov, Marcin Kubica, Ritu Kundu, Alessio Langiu, Thierry Lecroq, Arnaud Lefebvre, Chang Liu, Jakub Łącki, Manal Mohamed, Jakub Pachocki, Dominik Pająk, Marcin Pilipczuk, Michał Pilipczuk, Solon P. Pissis, Ely Porat, Élise Prieur-Gaston, Simon J. Puglisi, Jakub Radoszewski, Wojciech Rytter, Piotr Sankowski, Arseny M. Shur, William F. Smyth, Tatiana Starikovskaya, Juliusz Straszyński, Shiho Sugimoto, Bartosz Szreder, Wojciech Tyczyński, Hjalte Wedel Vildhøj, Tomasz Waleń, Bartłomiej Wiśniewski, Michał Włodarczyk, and Wiktor Zuba.

Finally, I would like to express my deepest gratitude to my parents. Without their love and strong support, this thesis would not have materialized.

Funding: This work has been supported by Polish budget funds for science in 2013–2017 as a research project under the 'Diamond Grant' program.

Contents

Chapter 1

Introduction

Texts (strings, words)—finite sequences of symbols from some alphabet—are among the most fundamental data types. They appear as natural-language passages, biological sequences, computer programs, and machine-generated files. Furthermore, they are used in data serialization and as a representation of network traffic, trajectories, time series, etc. Consequently, computational tasks involving strings arise in many areas of applied computer science, including bioinformatics, data compression, data mining, intrusion detection, plagiarism detection, and search engines. The abundance of applications motivates theoretical and experimental research on algorithms and data structures for text processing; see [71, 48, 41] for textbooks covering the area.

The central problem of text processing is pattern matching, which asks to enumerate the occurrences of one given string (a pattern) in another given string (a text), i.e., to identify fragments of the text that match the pattern. A linear-time pattern matching algorithm is known since 1970 [111], but optimization of space consumption and practical performance led to multiple further developments [55]. Many scenarios require indexing for pattern matching, where the task is to construct a data structure for the text allowing for pattern-matching queries with arbitrary patterns. A suffix tree [142] is the classic index with linear construction time and query time proportional to the pattern length. Nevertheless, its relatively large size motivated the development of more space-efficient alternatives such as the suffix array [107] and compressed indexes [119]. In the classic setting, matching is defined in terms of equality of strings, but numerous relaxations of this condition have been studied; see e.g. [116] for a survey on approximate string matching.

Another major family of problems involves dictionary compression, a class of lossless data compression methods including Lempel–Ziv [145] algorithms applied in many standard formats (such as GIF, PNG, PDF, ZIP, gzip). The key tasks in this area are not limited to efficient compression and decompression algorithms, but also include processing compressed texts [102].

A further main research direction is to seek regular structures, such as repetitions or palindromes, that can appear in strings [137]. Such structures are primarily interesting from the combinatorial point of view. Some also naturally appear in application-specific contexts (e.g. tandem repeats in DNA sequences), while others in the solutions of seemingly unrelated text-processing problems.

In many modern applications, the size of the textual data is in gigabytes. For example, already a single human genome consists of over 3 billion characters (base pairs), and computational tasks often involve many genomes. This amount of data requires fast and space-efficient processing, which poses many algorithmic challenges. As a result, designing text algorithms typically involves optimizing factors separating the running time from what is necessary to read the input. Space consumption of algorithms and data structures is even more important due to a risk of exceeding the capacity of the main memory. Consequently, research aims at achieving linear running time and working space. Often the goal is even more ambitious: to overcome this natural barrier, for example with the use of bit-parallelism; see [117] for a textbook on compact data structures.

1.1 Internal Queries in Texts

A diverse collection of text processing problems requires many algorithmic techniques. Nevertheless, some auxiliary tasks appear throughout the whole area, resulting in a widespread use of the tools employed to solve them. A prime example is a problem of determining the length of the longest common prefix of any two suffixes of the input text. Among many other applications, these LONGEST COMMON EXTENSION (LCE) QUERIES, also known as LONGEST COMMON PREFIX (LCP) QUERIES, arise in approximate pattern matching [99], during the construction of text indexing data structures such as the suffix tree [53] or the suffix array [82], and in the problem of identifying maximal repetitions [14].

Note that the longest common prefix is a notion defined for arbitrary two strings, and a naive scan determines its length in the optimal linear time if these strings are given explicitly in the input. In the problem of answering LCE QUERIES, we restrict the possible inputs to the suffixes of a fixed text T, with each suffix identified by its starting position. This transformation makes the problem much more interesting and brings many applications because now we can determine the longest common prefix in constant time after linear-time preprocessing of the text T [99, 53]. A straightforward generalization supports queries concerning arbitrary substrings of T, each represented by its occurrence in T, which we call a *fragment* of T, identified by the start and end position. LCE QUERIES are also used for answering even more basic questions concerning the fragments of T: to decide whether two fragments match, i.e., if they are occurrences of the same substring, and to compare them lexicographically.

A similar transformation can be applied to other problems involving one or more strings; the resulting questions are called *internal queries* in this thesis. In other words, the task is to build a data structure for a given text T such that later we can solve instances of a certain problem with input strings given as fragments of T. In more practical applications, the text T can be interpreted as a corpus containing all the data gathered for a given study (such as the concatenation of genomes to be compared). On the other hand, when internal queries arise during the execution of algorithms, then T is typically the input to be processed.

Note that the setting of internal queries does not include problems like text indexing, where the pattern is explicitly provided at query time. This restricts the expressibility of internal queries but at the same time allows for better running times, which do not need to account for reading any strings. Another difference is in the typical usage scenario: Data structures for the indexing problems are primarily designed to be interactively queried by a user. In other words, they are meant to be constructed relatively rarely, but they need to be stored for prolonged periods of time. As a result, space usage (including the multiplicative constants) is heavily optimized, while the efficiency of construction procedures is of secondary importance. On the other hand, internal queries often arise during bulk processing of textual data. The relevant data structures are then built within the preprocessing phase of the enclosing algorithms, so the running times of the construction procedures are counted towards the overall processing time. In this setting, efficient construction is as significant as fast queries. Components with linear-time deterministic construction and constant-time queries (like the classic solution for LCE Queries) are particularly valued, because algorithms can use them essentially for free, i.e., with no negative effect on the overall complexity in the standard theoretical setting.

In this thesis, we develop data structures for several types of internal queries; some have appeared in the literature prior to our work, while others are internal versions of classic problems of text algorithms. Below, we discuss each query type, state relevant previous and our results, and cover their consequences.

We always denote the input text by T and its length by n. We also make a standard assumption that the characters of T are (or can be identified with) integers $\{0,\dots,\sigma-1\}$, where the alphabet size σ is bounded by a polynomial in n. Our results are designed for the standard word RAM model with machine words of size $\Omega(\log n)$. All the algorithms we develop are deterministic.

1.1.1 Longest Common Extension Queries

In retrospect, the origins of internal queries in texts can be traced back to the invention of Longest Common Extension Queries, originally introduced by Landau and Vishkin for approximate pattern matching with respect to edit distance [99] and Hamming distance [60, 98]. Recall that this auxiliary task is to build a data structure that handles the following queries concerning the text T:

Longest Common Extension Queries
Given two positions i, i' in the text T, compute $\mathrm{LCE}(i,i') = \mathrm{lcp}(T[i\mathinner{.\,.}], T[i'\mathinner{.\,.}])$, i.e., the length of the longest common prefix of the suffixes starting at positions i and i'.

Technically, this formulation does not let us classify LCE Queries as internal queries. However, if $x = T[i\mathinner{.\,.}j]$ and $y = T[i'\mathinner{.\,.}j']$ are arbitrary fragments of T, then their longest common prefix is of length $\mathrm{lcp}(x,y) = \min(\mathrm{LCE}(i,i'), |x|, |y|)$. Thus, LCE Queries admit the following equivalent formulation:

LCE Queries (formulated as internal queries)
Given two fragments x, y of the text T, compute $\mathrm{lcp}(x,y)$, i.e., the length of the longest common prefix of the underlying substrings of T.

The classic data structure for LCE Queries is based on the suffix tree [142] of the text T equipped with a component for lowest common ancestor queries [73]. It takes $\mathcal{O}(n)$ space, supports constant-time queries, and can be constructed in $\mathcal{O}(n)$ time from the suffix tree of T. Early suffix tree construction algorithms took $\mathcal{O}(n\log\sigma)$ time, but Farach [53] later designed an $\mathcal{O}(n)$-time algorithm for any $\sigma = n^{\mathcal{O}(1)}$. Nevertheless, modern implementations of LCE Queries achieve the same results based on simpler and more practical tools: the suffix array and the LCP table of T [107], with a component for range minimum queries [17] built on top of the latter table.

The $\mathcal{O}(n)$ construction time and $\mathcal{O}(1)$ query time are obviously optimal, and the size of $\mathcal{O}(n)$ machine words (which is $\mathcal{O}(n\log n)$ bits) is also necessary in general. Nevertheless, this bound is no longer tight if the alphabet size σ is much smaller than the text length n. More precisely, the text only takes $\mathcal{O}(n \log \sigma)$ bits, which is $o(n \log n)$ for $\sigma = n^{o(1)}$. Furthermore, such a *packed* representation of T can be allowed in the input, making way for $o(n)$-time construction.

Recent work of multiple research groups brings improvements to LCE QUERIES in this setting. Tanimura et al. [140] and Munro et al. [114] showed that constant-time queries can be implemented using data structures of $\mathcal{O}(n\sqrt{\log n}\log \sigma)$ and $\mathcal{O}(n\sqrt{\log n \log \sigma})$ bits, respectively. The latter result admits an $\mathcal{O}(n/\sqrt{\log_\sigma n})$-time construction from the packed representation of T. In yet another study, Birenzwige et al. [24] applied our local consistency techniques (discussed in Section 1.2.1) so that constant-time LCE QUERIES in the optimal space of $\mathcal{O}(n \log \sigma)$ bits can be deduced as a corollary. Nevertheless, the original implementation of these tools only yields an $\mathcal{O}(n)$-time randomized construction algorithm. In the thesis, we provide a deterministic construction optimized for the packed setting, which lets us derive our first main result:

Theorem 1.1.1. *For every text T of length n over an alphabet of size σ, there exists a data structure of $\mathcal{O}(n \log \sigma)$ bits (i.e., $\mathcal{O}(n/\log_\sigma n)$ machine words) which answers* LCE QUERIES *in $\mathcal{O}(1)$ time. It can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time given the packed representation of the text T.*

Related Work

Several other active streams of research concern LCE QUERIES in various settings. One of them trades query time for improved *additional space* on top of the text T, which is then assumed to be stored in a read-only memory (or, equivalently, to be given by an oracle providing random access to individual characters). Many papers covering this setting [20, 22, 23, 24, 65, 139] provide randomized and deterministic data structures with various trade-offs between additional space usage and query time, as well as time and space complexity of the construction algorithms. Moreover, there is a tight unconditional lower bound [94] for the case when the additional space is relatively large ($\Omega(n)$ bits).

An alternative direction [57, 129] is to reuse the $n\lceil \log \sigma \rceil$ bits representing the text T, storing instead a data structure which not only allows accessing T but also supports fast LCE QUERIES. In this setting, the additional space consumption can be reduced to as little as $\mathcal{O}(1)$ machine words [129].

Nevertheless, compression allows storing many real-life texts in much less than $n\lceil \log \sigma \rceil$ bits. This phenomenon has been exploited in the development of data structures for LCE QUERIES in compressed texts [19, 21, 74, 75, 121, 139].

Another line of work studies LCE QUERIES in texts over abstract alphabets beyond the polynomially-bounded integers. Depending on whether such an alphabet is linearly ordered, it can be reduced to $\{0, \ldots, n-1\}$ using $\mathcal{O}(n \log n)$ or $\mathcal{O}(n^2)$ comparisons. Such a reduction often becomes the bottleneck of algorithms using LCE QUERIES, and various ways to circumvent this issue have been considered [44, 66, 93].

LCE QUERIES have also been studied in dynamic settings, where queries are interleaved with updates to the text [2, 64, 110, 121].

1.1.2 Period Queries

One of the central notions of combinatorics on words is that of a *period* of a string. An integer p is a period of a length-m string w if $1 \le p \le m$ and there is a length-p string u

such that w is a prefix of u^k (the concatenation of k copies of u) for a sufficiently large integer k. Equivalently, w has a period p if and only if its prefix of length $m-p$ matches the suffix of length $m-p$; such a substring, which occurs both as a prefix and as a suffix of w, is called a *border* of w.

The set of all periods of a string is computed in $\mathcal{O}(m)$ time as a step of the Morris–Pratt pattern matching algorithm [111]. A later version of this procedure [85] lets one further observe that while a string may have up to m periods, the sorted sequence of these values can be cut into $\mathcal{O}(\log m)$ arithmetic progressions. A complete characterization of the possible families of periods [70] further shows that the size of such a representation ($\mathcal{O}(\log^2 m)$ bits) is asymptotically tight. Hence, we adopt it in the internal version of the problem of finding all periods of a string, formally specified below.

Period Queries
Given a fragment x of the text T, report all periods of x (represented by non-overlapping arithmetic progressions).

We have introduced Period Queries in [88], presenting two solutions. The first data structure takes $\mathcal{O}(n \log n)$ space and answers Period Queries in the optimal $\mathcal{O}(\log |x|)$ time after $\mathcal{O}(n \log n)$-time randomized construction. The other one is based on orthogonal range searching; its size is $\mathcal{O}(n+S_{rsucc}(n))$ and the query time is $\mathcal{O}(Q_{rsucc}(n)\cdot\log|x|)$, where $S_{rsucc}(n)$ and $Q_{rsucc}(n)$ are analogous values for data structures answering range successor queries; see Section 3.7 for a definition. The state-of-the-art trade-offs are $S_{rsucc}(n) = \mathcal{O}(n)$ and $Q_{rsucc}(n) = \mathcal{O}(\log^{\varepsilon} n)$ for every constant $\varepsilon > 0$ [120], $S_{rsucc}(n) = \mathcal{O}(n \log\log n)$ and $Q_{rsucc}(n) = \mathcal{O}(\log\log n)$ [144], as well as $S_{rsucc}(n) = \mathcal{O}(n^{1+\varepsilon})$ and $Q_{rsucc}(n) = \mathcal{O}(1)$ for every constant $\varepsilon > 0$ [46]. The first of these data structures can be constructed in time $C_{rsucc}(n) = \mathcal{O}(n\sqrt{\log n})$ [16], and the third one in time $C_{rsucc}(n) = \mathcal{O}(n^{1+\varepsilon})$ [46]. On the other hand, no efficient construction algorithm has been provided for the second trade-off.

In this thesis, we develop a data structure that is asymptotically optimal for texts over polynomially-bounded integer alphabets.

Theorem 1.1.2. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which answers* Period Queries *in $\mathcal{O}(\log |x|)$ time. It can be constructed in $\mathcal{O}(n)$ time.*

Our query algorithm is based on the intrinsic relation between borders and periods of a string. In fact, to answer each Period Query, it combines the results of the following Prefix-Suffix Queries, used with $x = y$ to determine the borders of the fragment x.

Prefix-Suffix Queries
Given fragments x and y of the text T and a positive integer d, report all suffixes of y of length between d and $2d-1$ that also occur as prefixes of x (represented as an arithmetic progression of their lengths).

In other words, we actually prove the following auxiliary result, which has already been applied in the dynamic longest common substring problem [5] and for computing the longest unbordered substring [87].

Theorem 1.1.3. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which answers* Prefix-Suffix Queries *in $\mathcal{O}(1)$ time. It can be constructed in $\mathcal{O}(n)$ time.*

In many scenarios, very long periods ($p = m - o(m)$ for a string of length m) are irrelevant. The remaining periods correspond to borders of length $\Theta(m)$ and thus can be retrieved with just a constant number of Prefix-Suffix Queries. This restricted version of Period Queries turned out to be useful for algorithms detecting gapped repeats and subrepetitions [91, 63].

The case of $p \le \frac{1}{2}m$ is especially important since fragments x with periods not exceeding $\frac{1}{2}|x|$, called *periodic* fragments, can be extended to *maximal repetitions* (also known as runs; see Section 1.2.2). Our solution to arbitrary Prefix-Suffix Queries actually relies on a specialized component covering this particular case. This component has already been applied for approximate period recovery [3], identifying two-dimensional maximal repetitions [7], and detecting one-variable patterns [95]. Moreover, Bannai et al. [14] presented its alternative optimal implementation. The underlying special case of Period Queries also generalizes Primitivity Queries (asking if a fragment x is *primitive*, i.e., whether it does not match u^k for any string u and integer $k \ge 2$), earlier considered by Crochemore et al. [45], who developed a data structure of size $\mathcal{O}(n + S_{rsucc}(n))$ with $\mathcal{O}(Q_{rsucc}(n))$-time query algorithm.

1.1.3 Internal Pattern Matching Queries

The main technical contribution behind our solution to Prefix-Suffix Queries is, in fact, a data structure for the internal version of the pattern matching problem, which asks for the occurrences of one substring within another substring:

Internal Pattern Matching (IPM) Queries
Given fragments x and y of the text T satisfying $|y| < 2|x|$, report the starting positions of fragments matching x and contained in y (represented as an arithmetic progression).

We impose a restriction $|y| < 2|x|$ on the fragments involved in a query so that the starting positions of the occurrences of x within y form a single arithmetic progression and thus can be represented in constant space; see e.g. [27, 127] for a proof of this folklore property. Note that if $|y| \ge 2|x|$, then one can ask $\mathcal{O}(|y|/|x|)$ IPM Queries (for the occurrences of x within fragments y' of length $2|x| - 1$ contained in y, with overlaps of at least $|x| - 1$ characters between the subsequent fragments y') and report $\mathcal{O}(|y|/|x|)$ arithmetic progressions on the output.

Theorem 1.1.4. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which answers* IPM Queries *in $\mathcal{O}(1)$ time. It can be constructed in $\mathcal{O}(n)$ time.*

While IPM Queries have not been studied before, Keller et al. [83] showed that the decision version of these queries can be answered in $\mathcal{O}(Q_{rsucc}(n))$ time using a data structure of size $\mathcal{O}(n + S_{rsucc}(n))$ that can be constructed in $\mathcal{O}(n + C_{rsucc}(n))$ time.[1] The aforementioned query time is valid for arbitrary lengths $|x|$ and $|y|$, so the efficiency of this data structure is incomparable to ours. Keller et al. [83] also introduced more general Bounded Longest Common Prefix Queries, defined as follows:

[1]Recall that $S_{rsucc}(n)$, $Q_{rsucc}(n)$, and $C_{rsucc}(n)$ have been introduced in Section 1.1.2 as the space, query time, and construction time needed to answer range successor queries; see Section 3.7 for details.

Bounded Longest Common Prefix Queries
Given two fragments x and y of the text T, find the longest prefix p of x which occurs in y.

Our result for IPM Queries can be combined with the techniques of [83] in a more efficient implementation of Bounded Longest Common Prefix Queries. Compared to the original version in [83], the resulting data structure, specified below, has a $\log\log|p|$ factor instead of a $\log|p|$ factor in the query time.

Theorem 1.1.5. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n+S_{rsucc}(n))$ which answers* Bounded Longest Common Prefix Queries *in time $\mathcal{O}(Q_{rsucc}(n)\log\log|p|)$. It can be constructed in $\mathcal{O}(n+C_{rsucc}(n))$ time.*

1.1.4 Substring Compression Queries

The original motivation behind Bounded Longest Common Prefix Queries is in answering Substring Compression Queries, i.e., internal queries asking for a compressed representation of a substring or for the (exact or approximate) size of this representation. This family of problems was introduced by Cormode and Muthukrishnan [39], and some of the results were later improved by Keller et al. [83]. Substring Compression Queries have a fairly direct motivation: Consider a server holding a long repetitive text T and clients asking for substrings of T (e.g., chunks that should be displayed). A limited capacity of the communication channel justifies compressing these substrings.

Both the aforementioned papers apply the classic LZ77 compression scheme [145], and among other problems, they consider internal queries for the LZ factorization of a given fragment x and for the generalized factorization of one fragment x in the context of another fragment y. The latter is defined as the part representing x in the LZ factorization of a string $y\#x$, where # is a special sentinel symbol not present in the text. Bounded Longest Common Prefix Queries naturally appear in the solution to the following queries:

Generalized LZ Substring Compression Queries
Given two fragments x and y of the text T, compute the generalized LZ factorization of x with respect to y.

Consequently, the improved results for Bounded Longest Common Prefix Queries immediately yield a solution to Generalized LZ Substring Compression Queries with $\mathcal{O}(C\cdot Q_{rsucc}\log\log\frac{|x|}{C})$ query time, compared to the previously known $\mathcal{O}(C\cdot Q_{rsucc}\log\frac{|x|}{C})$-time queries of [83]; here, C is the number of phrases in the reported factorization. We also observe that other variants of LZ77, such as the non-overlapping (non-self-referential) factorization of x or the factorization of x relative to y (into substrings of y), can be constructed in $\mathcal{O}(C\cdot Q_{rsucc}\log\log\frac{|x|}{C})$ time; see Section 2.3 for formal definitions and sample factorizations. For comparison, the basic LZ Substring Compression Queries are answered in $\mathcal{O}(C\cdot Q_{rsucc})$ time [83].

While the original works on substring compression focus on LZ77, Cormode and Muthukrishnan [39] left other compression schemes for further research. In particular, they mention methods based on the Burrows–Wheeler transform [29] in this context. We make a major step in this direction and study substring compression queries with respect to the simplest of the schemes based on the BWT: the one where the transformed string is run-length encoded (see Section 2.3 for definitions).

> BWT+RLE Substring Compression Queries
> Given a fragment x of the text T, compute the run-length encoding RLE(BWT(x)) of the Burrows–Wheeler transform of the underlying substring.

Our data structure answers these queries with a logarithmic-time overhead.

Theorem 1.1.6. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which answers* BWT+RLE Substring Compression Queries *in $\mathcal{O}(C \cdot \log |x|)$ time, where C is the size of the run-length encoded Burrows–Wheeler transform of x. The data structure can be constructed in $\mathcal{O}(n\sqrt{\log n})$ time.*

1.1.5 Substring Suffix Rank and Selection Queries

The primary use for the data structure behind BWT+RLE Substring Compression Queries is actually emulating the suffix array [107] and the inverse suffix array of a substring. Retrieving the kth entry in such a suffix array can be formulated as the following internal queries:

> Substring Suffix Selection Queries
> Given a fragment x of the text T and an integer k, find the kth lexicographically smallest suffix of x.

Recall that the $(k+1)$th character of the Burrows–Wheeler transform BWT(x) precedes the kth lexicographically smallest suffix of x. Hence, the Substring Suffix Selection Queries also provide random access to BWT(x).

The problem of emulating the inverse suffix array is to compute the rank of a particular suffix of x among all the suffixes of x, i.e., to count the suffixes lexicographically smaller than or equal to the given one. Our Substring Suffix Rank Queries, defined below, are slightly more general as they take arbitrary fragments of T in the input.

> Substring Suffix Rank Queries
> Given fragments x and y of the text T, find the lexicographic rank of y among the suffixes of x.

Our data structure for Substring Suffix Selection Queries and Substring Suffix Rank Queries combines characteristic features of suffix trees [142] and wavelet trees [68], and thus we call it a *wavelet suffix tree.* Its implementation also relies on Internal Pattern Matching Queries.

Theorem 1.1.7. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which answers* Substring Suffix Selection Queries *and* Substring Suffix Rank Queries *in $\mathcal{O}(\log |x|)$ time. It can be constructed in $\mathcal{O}(n\sqrt{\log n})$ time.*

While the logarithmic query time is not necessarily optimal, a simple reduction to orthogonal range queries (range counting and selection queries; see Section 3.7) proves that it cannot be improved by more than an $\mathcal{O}(\log \log n)$ factor, even at the cost of slightly increasing the data structure size.

Proposition 1.1.8. *A data structure of size $\mathcal{O}(n \log^{\mathcal{O}(1)} n)$ must take $\Omega(\frac{\log n}{\log\log n})$ time in the worst case for both* Substring Suffix Selection Queries *and* Substring Suffix Rank Queries.

1.1.6 Minimal Suffix and Rotation Queries

For $k = 1$ and $k = |x|$, Substring Suffix Selection Queries reduce to computing the lexicographically smallest and the lexicographically largest suffixes of a given substring.

Minimal (Maximal) Suffix Queries
Given a fragment x of the text T, report the lexicographically smallest (largest) non-empty suffix of x.

A classic algorithm by Duval [52] computes both the minimal and the maximal suffix of a string in linear time using constant additional space. A simple extension of this procedure also determines these suffixes for all prefixes of the input string. Consequently, the answers to Minimal and Maximal Suffix Queries can be precomputed in quadratic time.

The first data structures for these internal queries are due to Babenko et al. [13], who showed how to answer Minimal Suffix Queries in $\mathcal{O}(\log^{1+\varepsilon} n)$ time and Maximal Suffix Queries in $\mathcal{O}(\log n)$ time, both using $\mathcal{O}(n)$ space. A later work [11] improved the query times to constant, preserving the linear size of the data structures. The one for Maximal Suffix Queries also admits an $\mathcal{O}(n)$-time construction algorithm, which makes it optimal for polynomially-bounded integer alphabets. On the other hand, the component for constant-time Minimal Suffix Queries is constructed in $\mathcal{O}(n \log n)$ time, which might be improved to $\mathcal{O}(n\frac{\log n}{\tau})$ at the cost of increasing the query time to $\mathcal{O}(\tau)$ (for $1 \le \tau \le \log n$).

In this thesis, we develop the first data structure for Minimal Suffix Queries to achieve both $\mathcal{O}(n)$ construction time and $\mathcal{O}(1)$ query time.

Theorem 1.1.9. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which answers* Minimal Suffix Queries *in $\mathcal{O}(1)$ time. It can be constructed in $\mathcal{O}(n)$ time.*

Our techniques give an optimal implementation of Maximal Suffix Queries too, which lets us reproduce the results of Babenko et al. [11]. More importantly, they are also useful in answering related queries for the lexicographically smallest or largest cyclic rotation of a substring. The two versions of these queries are symmetric (up reversing of the order of the alphabet), so we focus on the following Minimal Rotation Queries:

Minimal Rotation Queries
Given a fragment x of the text T, report the lexicographically smallest cyclic rotation of x.

The first linear-time algorithms computing the smallest cyclic rotation of a given string are due to Booth [26] and Shiloach [136], whereas Duval [52] later provided a constant-space implementation of such a procedure. However, unlike for Minimal Suffix Queries, the first linear-time algorithm determining the answers to Minimal Rotation Queries for all prefixes of a given text is by Apostolico and Crochemore [10].

Our data structure is the first one for Minimal Rotation Queries, but it is already optimal for texts over polynomially-bounded integer alphabets.

Theorem 1.1.10. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which answers* Minimal Rotation Queries *in $\mathcal{O}(1)$ time. It can be constructed in $\mathcal{O}(n)$ time.*

Both Minimal Suffix Queries and Minimal Rotation Queries have a few natural applications. The former can be used for computing the Lyndon factorization [35] of an arbitrary substring. With Theorem 1.1.9, we achieve query time proportional to the number of distinct Lyndon words in the factorization. The primary application of Minimal Rotation Queries, on the other hand, is canonization of substrings, i.e., classifying them according to cyclic equivalence (conjugacy). In particular, Theorem 1.1.10 yields matching results for the following Cyclic Equivalence Queries:

Cyclic Equivalence Queries
Given two fragment x and y of the text T, decide whether x and y are cyclically equivalent.

For technical reasons, our data structures also handle Generalized Minimal Suffix Queries and Generalized Minimal Rotation Queries: rather then just substrings of the text T, they support concatenations of such substrings as well. The query time for a concatenation of k substrings (each represented by an occurrence in T) increases to $\mathcal{O}(k^2)$. Even though we have not optimized this value, Generalized Minimal Rotation Queries have already been applied in the context of finding repetitions in partial words (strings with don't cares) [33].

1.1.7 Related Queries

Internal queries are not the only problems in the literature involving fragments of a static text. For example, the Interval Longest Common Prefix Queries of [39, 83] ask to find the maximum value $\mathrm{LCE}(i, j)$ for a given index i and an index j from a given interval $\{\ell, \ldots, r\}$. Similarly, the Range Longest Common Prefix Queries [1, 4, 8, 123] ask to maximize $\mathrm{LCE}(i, j)$ across distinct positions i, j both contained in a given interval. A closely related internal query would be to find the longest repeat in a substring, i.e., the longest string with at least two occurrences within the given fragment x of the text.

A few further problems cannot be classified as internal queries since their formulation involves the entire family of all fragments of the text T. This includes Substring Hashing Queries [54, 62, 67], which ask to evaluate a function h such that $h(x) = h(x')$ holds for fragments x, x' if and only if the two fragments match. On the other hand, Fragmented Pattern Matching Queries [6, 67] ask whether the concatenation of given substrings (each represented by its occurrence in T) is itself a substring of T.

In many indexing problems, the text T is logically partitioned into several documents. While most queries considered in this setting take an additional string in the input, in Cross-Document Pattern Matching Queries [92] such a string is specified by its occurrence as a fragment of one of the considered documents.

1.2 Our Techniques

Beyond the results stated above, technical advancements are also an outcome of our research. Similarly to many modern works in text processing, this thesis relies on tools from combinatorics on words, classic string algorithms, and state-of-the-art general-purpose data structures. On top of that, we introduce some novel ideas and contribute to the development of several existing techniques. Below, we briefly introduce a few most important foundations of our work.

1.2.1 Local Consistency

Our main innovation and the key tool behind the solutions for LONGEST COMMON EXTENSION QUERIES and INTERNAL PATTERN MATCHING QUERIES is a novel implementation of local consistency—the idea to make symmetry-breaking decisions involving a position i of the text T based on the characters at the nearby positions. As a result, positions i and i' are handled consistently provided that they appear in the same context. This way, we can guarantee that matching fragments of the text are represented in the same way.

The previous implementations of local consistency involve parsing the text. The first *locally consistent parsing*, by Sahinalp and Vishkin [133], is based on the deterministic coin tossing technique [38]. It has been successfully used for many problems such as parallel suffix tree construction, approximate pattern matching, and dynamic text indexing [133, 134, 132]. A closely related *edit-sensitive parsing* can be used to approximate edit distance with moves [40] and compressibility with respect to the LZ77 compression scheme [39]. Mehlhorn et al. [110] proposed an alternative randomized construction of a locally consistent parsing. Much more recently, Jeż invented *recompression* [80]: a technique which results in a simpler and more efficient parsing scheme. Besides bringing new applications, such as solving word equations [80] and pattern matching in grammar-compressed texts [79], recompression also allowed for improving and extending several results originally relying on earlier locally consistent parsing schemes; see e.g. [64, 61].

Unfortunately, the structure of all these schemes is not suitable for constant-time implementation of internal queries. This is because extracting any useful information about a fragment requires time proportional to the height of the parsing, which is typically logarithmic. Moreover, the context size at a given level of the parsing is expressed in terms of the number of phrases, whose lengths may vary significantly between regions of the text. To overcome these limitations, rather than using a locally consistent parsing, we define *synchronizing functions*, which select $\mathcal{O}(n/\tau)$ fragments of a given length τ based on a context of a size $\mathcal{O}(\tau)$. Uniform context size and constant evaluation time are the main distinctive features of this implementation.

A related technique is already known to practitioners working on sequence similarity: Synchronizing functions resemble *winnowing* schemes introduced by Schleimer et al. [135] and rediscovered as *minimizers* by Roberts et al. [130]. These tools have been applied for plagiarism detection [135, 138, 30], network forensics [128], and bioinformatics [130, 143, 51, 101]. Growing usage in the latter domain has also inspired a line of research providing rigorous analysis and performance improvements for uniformly random texts [108, 109, 122]. Nevertheless, none of these results is applicable in the worst-case scenario studied in theory, mainly because highly repetitive fragments cannot be ignored in our setting.

1.2.2 Maximal Repetitions (Runs)

Repetitive fragments require special treatment in many text-processing techniques. This includes applications of local consistency because symmetry-breaking choices are often infeasible within periodic regions of the text. Locally consistent parsing schemes automatically find repetitive fragments and collapse them using run-length encoding. On the other hand, when building synchronizing functions, we need an external tool to detect periodic regions of the text. Then, our implementation of LCE QUERIES seamlessly handles these substrings (as opposed to the earlier data structure of Birenzwige et al. [24]). On the other hand, our approach to IPM QUERIES fails if the pattern x has a small period. Consequently, we develop a specialized component for this special case (answering so-called INTERNAL PERIODIC PATTERN MATCHING QUERIES).

To handle periodic fragments, we exploit the fact that the structure of these fragments can be encoded by *maximal repetitions* (also known as runs) [106, 90]. This notion has been intensely studied over the past two decades, both in the algorithmic and combinatorial context, and the underlying line of research recently resulted in breakthrough structural and quantitative results [14]. Our main new technical contribution in this area is an auxiliary component allowing to efficiently extend an arbitrary periodic fragment to the maximal repetition with the same period. Nevertheless, our usage of maximal repetitions also relies on an earlier paper [45], which provides a convenient algorithmic interface for extracting relevant information from maximal repetitions.

1.2.3 Lyndon Words

Lyndon words are a further notion of combinatorics on words central to some results of this dissertation. Introduced by Lyndon [104] in the context of Lie algebras, they are widely used in algebra and combinatorics. They also have surprising algorithmic applications, including ones related to constant-space pattern matching [47], maximal repetitions [14], and the shortest common superstring problem [112]. The key combinatorial property of Lyndon words, proved by Chen et al. [35], states that every string can be uniquely decomposed into a non-increasing sequence of Lyndon words. Our approach to MINIMAL SUFFIX QUERIES and MINIMAL ROTATION QUERIES is centered around the notion of *significant suffixes* of this factorization, introduced by I et al. [76] to compute the Lyndon factorizations of grammar-compressed texts. While the Lyndon factorization is known to be crucially related to the lexicographically largest and smallest suffixes of a string (see [52], for example), ours is the first data structure for MINIMAL SUFFIX QUERIES relying on this connection.

1.2.4 Orthogonal Range Searching

A routine approach to many fragment-related queries is based on orthogonal range searching; see [100] for a survey covering many applications of this technique. Orthogonal range searching involves a collection of n points in a d-dimensional space (typically $d = 2$ in text processing), which has to be preprocessed subject to queries concerning points contained in a given (hyper)rectangle. A textbook solution to these computational-geometry problems is based on range trees [18]. Despite many significant improvements [34, 31, 16] and ongoing attention from leading researchers, even usage of the 2-dimensional range emptiness queries results in overheads in the asymptotic query time or data structure space. Moreover, the state-of-the-art construction time is no better than $\Theta(n\sqrt{\log n})$.

Thus, the main challenge in designing efficient data structures for some internal queries is to avoid orthogonal range searching. In the case of IPM QUERIES and PREFIX-SUFFIX QUERIES, this strategy let us obtain optimal solutions (for large alphabets). Other internal queries are more challenging, often at least as difficult as some range queries; we then apply IPM Queries just to make the overhead as small as possible. Sometimes (for Bounded Longest Common Prefix Queries), we can use data structures for orthogonal range searching as black boxes. In other cases (for Substring Suffix Selection Queries and Substring Suffix Rank Queries), we need to adjust a modern data structure for range queries (a wavelet tree [68]) to our purposes. Due to this connection, the research reported in this thesis also resulted in a faster wavelet tree construction algorithm, originally applied for range selection [12] and later for range successor queries [16].

1.2.5 Fusion Trees

Orthogonal range searching is not the only family of abstract problems that naturally appear in the context of internal queries in texts. Our procedures for Minimal Suffix Queries and Minimal Rotation Queries also crucially rely on predecessor search in sets of $\mathcal{O}(\log n)$ integers. In the word RAM model, these auxiliary queries can be answered in $\mathcal{O}(1)$ time using fusion trees [59] of Fredman and Willard. The original implementation of these trees does not provide an efficient construction procedure, so we actually apply much more recent dynamic fusion trees [126] by Pătraşcu and Thorup. Our usage of fusion trees also extends to IPM Queries for short patterns of length $|x| = \mathcal{O}(\log n)$. The specialized implementation for this case is needed to compensate for the overheads arising in the construction of deterministic dictionaries [131] and synchronizing functions, both central to answering IPM Queries for longer patterns.

1.3 Organization of the Thesis

The technical part of this dissertation starts with two preliminary chapters, where we recall standard notions and results related to text algorithms (Chapter 2) and abstract data structures (Chapter 3). Next, in Chapter 4, we study Longest Common Extension Queries in texts over small integer alphabets. Chapter 5 is devoted to handling periodic fragments of the input text. The auxiliary components developed there are then used in Chapter 6, where we show how to answer Internal Pattern Matching Queries. Chapter 7 covers applications of IPM Queries to further internal queries, including Period Queries and subroutines later used within wavelet suffix trees, developed in Chapter 8 to answer Substring Suffix Selection Queries and Substring Suffix Rank Queries. Our data structures for Minimal Suffix and Minimal Rotation Queries are described in Chapter 9. We conclude with Chapter 10, where we discuss possible directions for further research concerning internal queries.

1.4 Concise Summary of Our Main Contributions

Our primary results are data structures answering the following internal queries:

- LCE Queries in texts over small alphabets (see Theorem 1.1.1 and Chapter 4);
- Period Queries (see Theorem 1.1.2 and Chapter 7);
- Prefix-Suffix Queries (see Theorem 1.1.3 and Chapter 7);
- Internal Pattern Matching Queries (see Theorem 1.1.4 and Chapter 6);
- Bounded Longest Common Prefix Queries (see Theorem 1.1.5 and Chapter 7);
- LZ Substring Compression Queries (see Section 1.1.4 and Chapter 7);

- BWT+RLE Substring Compression Queries (see Theorem 1.1.6 and Chapter 8);
- Substring Suffix Selection Queries and Substring Suffix Rank Queries (see Theorem 1.1.7 and Chapter 8);
- Minimal Suffix Queries (see Theorem 1.1.9 and Chapter 9);
- Minimal Rotation Queries (see Theorem 1.1.10 and Chapter 9).

Some of the underlying techniques are also of independent interest beyond internal queries:

- synchronizing functions, a novel implementation of local consistency suitable for efficient processing of static texts (see Section 1.2.1 and Chapter 4);
- new applications of maximal repetitions (runs), including a component for extending periodic fragments to maximal repetitions (see Section 1.2.2 and Chapter 5).

1.5 Papers Covered in the Thesis

Most of the results included in this dissertation come from the following conference papers:

- *Internal pattern matching queries in a text and applications* [89], a joint work with Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń, published in the *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA 2015). This paper comprises results for Internal Pattern Matching Queries, Period Queries, and LZ Substring Compression Queries. Our data structure answering Longest Common Extension Queries is a new application of the underlying techniques. Moreover, deterministic construction algorithms are provided in the thesis instead of earlier randomized ones.
- *Wavelet trees meet suffix trees* [12], a joint work with Maxim Babenko, Paweł Gawrychowski, and Tatiana Starikovskaya, published in the *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA 2015). This paper includes results for Substring Suffix Selection Queries, Substring Suffix Rank Queries, and BWT+RLE Substring Compression Queries.
- *Minimal suffix and rotation of a substring in optimal time* [86], published in the *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching* (CPM 2016), distinguished with the *Alberto Apostolico Best Paper Award.* This paper comprises results for Minimal Suffix Queries and Minimal Rotation Queries.

Chapter 2

Preliminaries I: Strings

We consider *strings* over an *alphabet* Σ, i.e., finite sequences of *characters* from the set Σ. Throughout the thesis, we assume that Σ is identified with a range[1] $[\![0, \sigma - 1]\!]$ of non-negative integers. In the algorithmic results, we further assume that the alphabet size σ does not exceed $n^{\mathcal{O}(1)}$, where n is the length of the input string, called the *text* and typically denoted as T. Such an alphabet, called a (polynomially-bounded) *integer alphabet*, is currently standard in the literature.

In this chapter, we recall some basic concepts and results of text algorithms. This includes fundamental notions from combinatorics on words and standard data structures for text processing. A more detailed exposition can be found in the classic textbooks [103, 71, 48, 41]; they do not include some of the most recent developments, though.

2.1 Basic Combinatorics on Words

The set of all strings over Σ is denoted by Σ^*, the empty string is ε, and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ is the set of non-empty strings over Σ. The *concatenation* of two strings $u, v \in \Sigma^*$ is denoted $u \cdot v$ or uv.

Let us fix a string $w \in \Sigma^*$ of *length* $|w| = n$. For $1 \le i \le n$, we refer to the ith character as $w[i]$ (we use 1-based indexing). We identify strings of length 1 with the underlying characters, which lets us write $w = w[1] \cdots w[n]$. The string $w^R = w[n] \cdots w[1]$ is called the *reverse* of w.

For a fixed string w, we say that an integer i such that $1 \le i \le |w|$ is a *position in w*. A string u is called a *substring* (or a *factor*) of w if there are two positions $i \le j$ in w such that $u = w[i] \cdots w[j]$. In this case, we say that u *occurs* in w at position i, and we denote by $w[i\,..\,j]$ the *occurrence* of u at position i.

We call $w[i\,..\,j]$ a *fragment* of w. Formally, the fragment $w[i\,..\,j]$ is a pair of positions i, j in w such that $i \le j$: the *start position* i and the *end position* j. If $w[i\,..\,j]$ is an occurrence of a string u, then we write $u \cong w[i\,..\,j]$ and say that $w[i\,..\,j]$ *matches* u. Similarly, if $w[i\,..\,j]$ and $w[i'\,..\,j']$ are occurrences of the same string, we denote this by $w[i\,..\,j] \cong w[i'\,..\,j']$, and we say that these fragments match. On the other hand, the equality of fragments $w[i\,..\,j] = w[i'\,..\,j']$ is reserved for occasions when $w[i\,..\,j]$ is the same fragment as $w[i'\,..\,j']$ (i.e., $i = i'$ and $j = j'$). We assume that a fragment x of w, $x = w[i\,..\,j]$, inherits some notions from the underlying substring: the *length* $|x| = j - i + 1$, the characters $x[i']$ for $1 \le i' \le |x|$, defined as $w[i' + i - 1]$, and the *subfragments* $x[i'\,..\,j']$

[1] For any integers i, j with $i \le j$, we denote the range (the integer interval) $\{i, \ldots, j\}$ by $[\![i, j]\!]$.

for $1 \le i' \le j' \le |x|$, defined as the fragments $w[i'+i-1\mathinner{.\,.} j'+i-1]$.

A fragment $w[i\mathinner{.\,.} j]$ also has a natural interpretation as a *range* $[\![i,j]\!]$ of positions in w. This lets us consider *disjoint* or *intersecting* (*overlapping*) fragments and define the containment relation ($\subseteq$) on fragments. Moreover, for positions $i \le j < k$, fragments $w[i\mathinner{.\,.} j]$ and $w[j+1\mathinner{.\,.} k]$ are called *consecutive* and $w[i\mathinner{.\,.} k] = w[i\mathinner{.\,.} j]w[j+1\mathinner{.\,.} k]$ is assumed to be their *concatenation.* If fragments $w[i\mathinner{.\,.} j]$ and $w[i'\mathinner{.\,.} j']$ intersect, we define their *intersection* $w[\max(i,i')\mathinner{.\,.}\min(j,j')]$ by $w[i\mathinner{.\,.} j] \cap w[i'\mathinner{.\,.} j']$. Similarly, if $w[i\mathinner{.\,.} j]$ and $w[i'\mathinner{.\,.} j']$ are overlapping or consecutive, then their *union* $w[i\mathinner{.\,.} j] \cup w[i'\mathinner{.\,.} j']$ is $w[\min(i,i')\mathinner{.\,.}\max(j,j')]$.

A fragment x of w of length $|x| < |w|$ is called a *proper* fragment of w. A fragment starting at position 1 is called a *prefix* of w and a fragment ending at position $|w|$ is called a *suffix* of w. We sometimes use abbreviated notation $w[\mathinner{.\,.} j]$ and $w[i\mathinner{.\,.}]$ for a prefix $w[1\mathinner{.\,.} j]$ and a suffix $w[i\mathinner{.\,.} |w|]$ of w, respectively. We extend the notions of a prefix and a suffix to the underlying substrings. A string u is a *common prefix* of two strings v, w if it is a prefix of both v and w; *common suffixes* are defined symmetrically. The length of the longest common prefix of two strings v, w is denoted by $\mathrm{lcp}(v,w)$, and the length of their longest common suffix is denoted by $\mathrm{lcs}(v,w)$.

We denote by $\prec$ the natural order on Σ and extend this order in the standard way to the *lexicographic* order on Σ^*: a string x is lexicographically smaller than y (denoted $x \prec y$) if either x is a proper prefix of y, or $\mathrm{lcp}(x,y) < \min(|x|,|y|)$ and $x[\mathrm{lcp}(x,y)+1] \prec y[\mathrm{lcp}(x,y)+1]$.

We introduce two distinct sentinel symbols $\#, \$ \notin \Sigma$ and extend the order $\prec$ to $\bar{\Sigma} = \Sigma \cup \{\#, \$\}$ so that $\# \prec c \prec \$$ for every $c \in \Sigma$. In the algorithmic results, we assume that Σ is the input alphabet, while $\bar{\Sigma}$ is sometimes internally used by our solutions.

2.2 Periodic Structures in Strings

An integer p, $1 \le p \le |w|$, is a *period* of a string $w \in \Sigma^+$ if $w[i] = w[i+p]$ for $1 \le i \le |w| - p$. The shortest period of w is denoted as $\mathrm{per}(w)$; the *exponent* of w is then defined as $\exp(w) = \frac{|w|}{\mathrm{per}(w)}$. We call w *periodic* if $\exp(w) \ge 2$ and *highly periodic* if $\exp(w) \ge 3$. A *border* of a string w is a substring of w which occurs both as a prefix and as a suffix of w. Note that p is a period of w if and only if w has a border of length $|w| - p$. Periods of a string w satisfy the periodicity lemma, one of the classic results in combinatorics on words.

Lemma 2.2.1 (Periodicity Lemma [105, 56])**.** *Let w be a string with periods p and q. If $p + q - \gcd(p,q) \le |w|$, then $\gcd(p,q)$ is also a period of w.*

For a string $w \in \Sigma^*$ and an integer $k \in \mathbb{Z}_{\ge 0}$, we denote the concatenation of k copies of w by w^k. In the case of $k \ge 2$, the string w^k is called a *power* of w (with *root* w); w^2 is called a *square* and w^3 is a *cube.* A non-empty string $u \in \Sigma^+$ is *primitive* if it is not a power, i.e., $u \ne w^k$ for every integer $k \ge 2$ and every root w. Primitive strings enjoy a synchronizing property, which is an easy consequence of Lemma 2.2.1.

Lemma 2.2.2 (see [41, Lemma 1.11])**.** *A non-empty string u is primitive if and only if it occurs exactly twice in u^2 (as a prefix and as a suffix).*

We say that a string w' is a *rotation* (cyclic shift, conjugate) of a string w if there exists a decomposition $w = uv$ such that $w' = vu$. Here, w' is the left rotation of w by $|u|$

characters and the right rotation of w by $|v|$ characters. By Lemma 2.2.2, a string $w \in \Sigma^+$ is primitive if and only if it has $|w|$ distinct cyclic rotations. A string $w \in \Sigma^+$ is called a *Lyndon word* [104, 35] if w is primitive and $w \preceq w'$ for every cyclic rotation w' of w.

2.2.1 Maximal Repetitions (Runs)

A *run* (maximal repetition) [106, 90] in a string w is a periodic fragment $\gamma = w[i\,..\,j]$ which can be extended neither to the left nor to the right without increasing the shortest period $p = \operatorname{per}(\gamma)$, that is, $w[i-1] \neq w[i+p-1]$ and $w[j-p+1] \neq w[j+1]$ provided that the respective positions exist. The algorithms considered in this thesis always store runs together with their periods so that $\operatorname{per}(\gamma)$ can be retrieved in constant time. We denote the set of all runs in a string w by $\mathcal{R}(w)$.

Example 2.2.3. A string $w =$ `baababaababb` has three runs with period 1: $w[2\,..\,3] \cong$ `aa`, $w[7\,..\,8] \cong$ `aa`, and $w[11\,..\,12] \cong$ `bb`; two runs with period 2: $w[3\,..\,7] \cong$ `ababa` and $w[8\,..\,11] \cong$ `abab`; one run with period 3: $w[5\,..\,10] \cong$ `abaaba`; and one run with period 5: $w[1\,..\,11] \cong$ `baababaabab`.

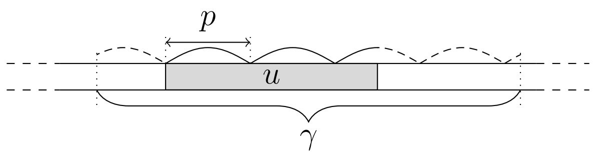


Figure 2.1: An illustration of a run γ extending a fragment u. We have $\mathsf{run}(u) = \gamma$ and $\operatorname{per}(u) = \operatorname{per}(\gamma) = p \le \frac{1}{2}|u|$.

We say that a run γ *extends* a fragment x if x is contained in γ ($x \subseteq \gamma$) and $\operatorname{per}(x) = \operatorname{per}(\gamma)$; see Figure 2.1. Note that every periodic fragment can be extended to a run with the same period. Moreover, the following easy consequence of Lemma 2.2.1 implies that this extension is unique:

Fact 2.2.4. *Let $\gamma \neq \gamma'$ be overlapping runs in a string w. If $p = \operatorname{per}(\gamma)$ and $p' = \operatorname{per}(\gamma')$, then $|\gamma \cap \gamma'| < p + p' - \gcd(p, p')$.*

Proof. For a proof by contradiction, suppose that $|\gamma \cap \gamma'| \ge p + p' - \gcd(p, p')$. By Lemma 2.2.1, this means that $\gcd(p, p')$ is a period of the intersection $\gamma \cap \gamma'$, which we denote $w[\ell\,..\,r]$. Since $\gamma \neq \gamma'$, one of these runs must contain position $\ell - 1$ or $r + 1$. Due to symmetry, we may assume without loss of generality that γ contains position $\ell - 1$.

Observe that positions $\ell+p-1$ and $\ell+p'-1$ are located within $\gamma\cap\gamma'$, so $w[\ell+p-1] = w[\ell+p'-1]$. On the other hand, $w[\ell-1] = w[\ell+p-1]$ (because γ contains position $\ell-1$) and $w[\ell-1] \neq w[\ell+p'-1]$ (by maximality of γ'). Thus, $w[\ell-1] = w[\ell+p-1] = w[\ell+p'-1] \neq w[\ell-1]$, which is a contradiction that concludes the proof. $\square$

We denote the unique run extending x by $\mathsf{run}(x)$. If x is not periodic, we leave $\mathsf{run}(x)$ undefined, which we denote as $\mathsf{run}(x) = \perp$.

Kolpakov and Kucherov [90] proved $\mathcal{O}(n)$ upper bounds on the number of runs in a text of length n and on the sum of their exponents. They also designed an $\mathcal{O}(n)$-time algorithm for computing runs (for texts over constant-sized alphabets). A number of follow-up papers improved the constants in these combinatorial bounds. Also, the linear-time algorithm has been extended to texts over integer alphabets [42]. In a recent breakthrough paper, Bannai et al. [14] proved that $|\mathcal{R}(w)| < |w|$ and that $\sum_{\gamma\in\mathcal{R}(w)} \exp(\gamma) < 3|w|$. They also gave a significantly simpler linear-time algorithm for computing runs, which is based on the structure of Lyndon words in the text. Nevertheless, our results rely on the asymptotic bounds only:

Proposition 2.2.5 ([90, 42, 14])**.** *Given a text T of length n, the set $\mathcal{R}(T)$ of all runs in T (together with their periods) can be computed in $\mathcal{O}(n)$ time. Moreover, the number of runs $\mathcal{R}(T)$ and the sum of their exponents $\sum_{\gamma\in\mathcal{R}(T)} \exp(\gamma)$ are both $\mathcal{O}(n)$.*

2.3 String Compression

Some of the problems considered in this thesis ask to compute compressed representations of strings. We work with two compression methods: the Lempel–Ziv LZ77 algorithm [145] and the run-length encoding of the Burrows–Wheeler transform [29].

2.3.1 LZ77 Compression

Consider a string $w \in \Sigma^*$. We say that a fragment $w[\ell\mathinner{.\,.} r]$ has a *previous occurrence* (or is a *previous fragment*) if $w[\ell\mathinner{.\,.} r] \cong w[\ell'\mathinner{.\,.} r']$ for some positions $\ell' < \ell$ and $r' < r$. The fragment $w[\ell\mathinner{.\,.} r]$ has a *non-overlapping previous occurrence* (or is a *non-overlapping previous fragment*) if additionally $r' < \ell$.

The Lempel–Ziv factorization $\mathrm{LZ}(w)$ is a factorization $w = f_1 \cdots f_k$ into fragments (called *phrases*) such that each phrase f_i is the longest previous fragment starting at position $1 + |f_1 \cdots f_{i-1}|$, or a single letter if there is no such previous fragment. The non-overlapping Lempel–Ziv factorization $\mathrm{LZ}_N(w)$ is defined in an analogous way, allowing for non-overlapping previous fragments only. Both factorizations (and several closely related variants) are useful for compression because a previous fragment can be represented using a reference to the previous occurrence (e.g., the positions of its endpoints).

Strings $w \in \Sigma^*$ are sometimes compressed with respect to a *context string* (or *dictionary string*) $v \in \Sigma^*$. Essentially, there are two ways to define the factorization $\mathrm{LZ}(w \mid v)$ of w with respect to v. In the *relative LZ factorization* [146, 97] $\mathrm{LZ}_R(w \mid v)$, each phrase is the longest fragment of w which starts at the given position and occurs in v (or a single letter if there is no such fragment). An alternative approach is to allow both substrings of v and previous fragments of w as phrases. This results in the *generalized LZ factorization*, denoted $\mathrm{LZ}_G(w \mid v)$; see [39, 83]. Equivalently, $\mathrm{LZ}_G(w \mid v)$ can be defined as the suffix of $\mathrm{LZ}(v\#w)$ corresponding to w. The previous fragments in the non-overlapping generalized LZ factorization $\mathrm{LZ}_{NG}(w \mid v)$ must be non-overlapping.

Example 2.3.1. Let $w = \texttt{aaaabaabaaaa}$ and $v = \texttt{baabab}$. We have

$$\begin{aligned}
\mathrm{LZ}(w) &= \texttt{a}\cdot\texttt{aaa}\cdot\texttt{b}\cdot\texttt{aabaa}\cdot\texttt{aa},\\
\mathrm{LZ}_N(w) &= \texttt{a}\cdot\texttt{a}\cdot\texttt{aa}\cdot\texttt{b}\cdot\texttt{aab}\cdot\texttt{aaaa},\\
\mathrm{LZ}_R(w \mid v) &= \texttt{aa}\cdot\texttt{aaba}\cdot\texttt{aba}\cdot\texttt{aa}\cdot\texttt{a},\\
\mathrm{LZ}_G(w \mid v) &= \texttt{aa}\cdot\texttt{aaba}\cdot\texttt{abaa}\cdot\texttt{aa},\\
\mathrm{LZ}_{GN}(w \mid v) &= \texttt{aa}\cdot\texttt{aaba}\cdot\texttt{aba}\cdot\texttt{aaa}.
\end{aligned}$$

2.3.2 BWT+RLE Compression

The *Burrows–Wheeler transform* [29] (BWT) of a string w is a string $\mathrm{BWT}(w) = b_1 b_2 \cdots b_{|w|+1}$, where b_k is the character preceding the kth lexicographically smallest suffix of $w\#$ (assuming that $\#$ precedes $w[1]$). Defined this way, $\mathrm{BWT}(w)$ uniquely determines the original string w. Moreover, if w is repetitive, then $\mathrm{BWT}(w)$ tends to contain long segments of equal characters, i.e., long *runs* with period 1 (see Section 2.2.1). This, combined with *run-length encoding* (RLE), allows compressing strings efficiently. The *run-length encoding* of a string is obtained by replacing each run with period 1 by a pair consisting of the character that forms the run and the length of the run.

Example 2.3.2. The Burrows–Wheeler transform of a string $w = \texttt{bacaca}$ is $\mathrm{BWT}(w) = \texttt{accb\#aa}$, whose run-length encoding is $\mathrm{RLE}(\mathrm{BWT}(w)) = \texttt{a}^1\texttt{c}^2\texttt{b}^1\#^1\texttt{a}^2$.

2.4 Infinite Strings

Occasionally, we also work with the family Σ^∞ of *infinite strings* (indexed by positive integers). The notion of concatenation uv extends to $u \in \Sigma^*$ and $v \in \Sigma^\infty$, resulting in $uv \in \Sigma^\infty$. Also, the notions of substrings, fragments, suffixes, prefixes (including the longest common prefix of two strings), as well as the lexicographic order naturally extend to infinite strings. For a word $w \in \Sigma^+$, we also introduce the infinite power $w^\infty \in \Sigma^\infty$, i.e., the concatenation of infinitely many copies of w. More formally, this is the unique infinite string which has w^k as a prefix for every $k \in \mathbb{Z}_{\geq 0}$; see Figure 2.2.

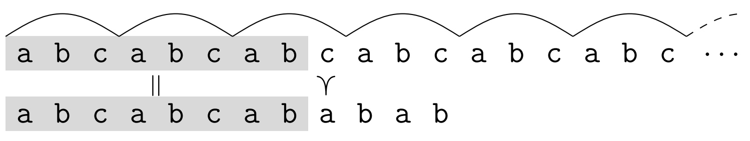


Figure 2.2: Lexicographic comparison of a finite and an infinite string. We have $\mathrm{lcp}((\texttt{abc})^\infty, \texttt{abcabcababab}) = 8$ and $(\texttt{abc})^\infty \succ \texttt{abcabcababab}$.

2.5 Suffix Arrays, LCE Queries, and Applications

The *suffix array* [107] of a text T of length n is a permutation SA of $\{1, \ldots, n\}$ defining the lexicographic order on the suffixes of T: $T[SA[i]\,..] \prec T[SA[j]\,..]$ if and only if $i < j$. The suffix array takes $\mathcal{O}(n)$ space and can be constructed in $\mathcal{O}(n)$ time [82].

One of the most important applications of the suffix array is in answering LCE QUERIES, first introduced by Landau and Vishkin in the context of approximate pattern matching [99]. Given two positions i, j in T, these queries ask for $\mathrm{LCE}(i, j) = \mathrm{lcp}(T[i\,..], T[j\,..])$. The standard solution is to build the suffix array SA, its inverse ISA (defined so that $SA[ISA[i]] = i$), the LCP table (storing $LCP[i] = \mathrm{LCE}(SA[i-1], SA[i])$ for $2 \leq i \leq n$), and a data structure for range minimum queries (see Section 3.7) built on the LCP table; see [82, 53, 99].

Proposition 2.5.1 (Answering LCE QUERIES)**.** *Given a text T of length n, one can construct in $\mathcal{O}(n)$ time a data structure of size $\mathcal{O}(n)$ answering* LCE QUERIES *in $\mathcal{O}(1)$ time.*

The LCE Queries can be used to answer further questions concerning fragments of the input text T.

Fact 2.5.2 (Applications of LCE Queries)**.** *Assume that we have access to a text T equipped with a data structure answering* LCE Queries *in constant time. Given fragments x, y of T, the following queries can be answered in $\mathcal{O}(1)$ time:*

(a) compute the length of the longest common prefix $\mathrm{lcp}(x, y)$*,*

(b) decide whether $x \prec y$, $x \cong y$, or $x \succ y$,

(c) compute $\mathrm{lcp}(x^\infty, y)$ *and decide if $x^\infty \prec y$ or $x^\infty \succ y$.*

Proof. Let $x = T[i_x \mathinner{.\,.} j_x]$ and $y = T[i_y \mathinner{.\,.} j_y]$.
(a) We have $\mathrm{lcp}(x, y) = \min(\mathrm{LCE}(i_x, i_y), |x|, |y|)$.
(b) Let $\ell = \mathrm{lcp}(x, y)$. If $\ell = \min(|x|, |y|)$, then the lexicographic order of x and y coincides with the order of $|x|$ and $|y|$. Otherwise, it coincides with the order of $x[\ell + 1]$ and $y[\ell + 1]$.
(c) If $\mathrm{lcp}(x, y) < |x|$, i.e., x is not a prefix of y, then $\mathrm{lcp}(x^\infty, y) = \mathrm{lcp}(x, y)$ and the order between x^∞ and y is the same as between x and y. If $x \cong y$ on the other hand, then clearly $\mathrm{lcp}(x^\infty, y) = |x|$ and $x^\infty \succ y$. Otherwise, consider a fragment $y' = T[i_y + |x| \mathinner{.\,.} j_y]$. A simple inductive proof shows that $\mathrm{lcp}(x^\infty, y) = |x| + \mathrm{lcp}(x^\infty, y') = |x| + \mathrm{lcp}(y, y')$ and that the order between x^∞ and y is the same as between y and y'. Consequently, the query can be answered in constant time in each case. □

2.6 Tries, Compressed Tries, and Suffix Trees

A *trie* is a rooted tree whose nodes correspond to prefixes of strings in a given (finite) family of strings $A \subseteq \Sigma^*$. If ν is a node, then the corresponding prefix v is called the *value* of the node. The node with value v is called the *locus* of v.

The parent-child relation in the trie is defined so that the root is the locus of ε, while the parent μ of a node ν is the locus of the value of ν with the last character removed. This character is the *label* of the edge from μ and ν. In general, if μ is an ancestor of ν, then the label of the path from μ to ν is the concatenation of edge labels on the path.

A node is *branching* if it has at least two children and *terminal* if its value belongs to A. A *compacted trie* is obtained from the underlying trie by dissolving all nodes except the root, branching nodes, and terminal nodes. In other words, we compress paths of vertices with single children, and thus the number of remaining nodes becomes bounded by $2|A|$. In general, we refer to all preserved nodes of the trie as *explicit* (since they are stored explicitly) and to the dissolved ones as *implicit*. An implicit node ν can be represented as a pair (μ, d), where μ is the lowest explicit descendant of ν, and d is the distance (in the uncompacted trie) from ν to μ. The pair (μ, d) is called the *locus* of the value of ν in the compacted trie, and μ is called its *explicit locus*. Edges of a compacted trie correspond to paths in the underlying trie and thus their labels are strings in Σ^+. Typically, these labels are stored as references to fragments of the strings in A.

The *suffix trie* of a text $T \in \Sigma^*$ is the trie of all suffixes of T. Consequently, there is a bijection between substrings of T and nodes of the suffix trie. The *suffix tree* of T [142], denoted $\mathcal{T}_{\mathsf{suf}}(T)$, is the compacted suffix trie of T. For a text T of length n, it takes $\mathcal{O}(n)$ space and can be constructed in $\mathcal{O}(n)$ time either directly [53] or from the suffix array of T; see [48, 41].

Chapter 3

Preliminaries II: Data Structures

In Chapter 2, we introduced classic combinatorial and algorithmic tools for text processing. The results of this thesis also rely on several abstract data structures, which we recall below. All these data structures are *static*—they are constructed for some input data (which cannot be modified) and their task is to answer queries about this data.

We start, in Section 3.1, with a brief description of the word RAM model of computation used throughout the thesis. Section 3.2 presents the standard *packed* representation of strings over small alphabet in the word RAM model. Next, in Section 3.3, we define fundamental abstract queries: rank, selection, predecessor, and successor. In the following sections, we recall data structures answering these queries in various settings. Fusion trees (Section 3.4) are efficient for small integer sets, while data structures based on bitmasks (Section 3.5) can be used for relatively small universes. Rank and selection on bitmasks can be generalized to similar queries on sequences over a larger alphabet. In Section 3.6, we discuss the standard tool for this setting: a wavelet tree. We give a relatively detailed description of wavelet trees since the main data structure of Chapter 8 is based on them. We conclude by recalling results for several types of range queries in Section 3.7.

3.1 Word RAM Model

Throughout the thesis, we assume the standard word RAM model of computation, which nowadays is the default choice for sequential internal-memory algorithms and data structures. Below, we briefly introduce the model; we refer to [72] for a full exposition.

In the word RAM model, the memory is composed of M cells which hold *machine words*, i.e., W-bit integers (from $[\![0, 2^W - 1]\!]$). Cells are addressed by consecutive integers $[\![0, M-1]\!]$, which must fit in machine words (this requires $M \le 2^W$). We always assume that the memory is large enough to fit the input data. Consequently, $W \ge \log N$, where N is the input size.[1] The model supports constant-time random access to the memory (reads and writes of the cell at an arbitrary address), as well as constant-time bit-wise and arithmetic operations on W-bit integers (including multiplication). A non-trivial consequence of these assumptions in that the positions of the most significant bit and the least significant bit can be retrieved in $\mathcal{O}(1)$ time:

Proposition 3.1.1 (Fredman and Willard [59]). *There is a constant-time word RAM algorithm which, given a W-bit integer $x \neq 0$, returns $\lfloor \log x \rfloor$, i.e., the index of the most significant bit in x which is set to one.*

[1]The log function denotes the binary logarithm unless a different base appears in the subscript.

Corollary 3.1.2. *There is a constant-time word RAM algorithm which, given a W-bit integer $x \neq 0$, returns the index of the least significant bit of x which is set to one.*

Proof. It suffices to observe that the least significant bit of x can be retrieved as the most significant bit of $x \oplus (x-1)$, where $\oplus$ denotes the bit-wise 'xor'. □

Memory cells (machine words) are the default unit while measuring the space complexity of algorithms and the size of data structures. Single bits can be used as an alternative unit (if mentioned explicitly). Lower bounds for the time and space complexity of data structures in the word RAM model are usually proved for a more powerful *cell-probe model*, whose memory organization is the same, but only memory access operations are counted towards the running time (which means that computation on individual machine words is free).

3.2 Packed Representation of Strings

Strings are typically represented in the word RAM model as arrays, with each character occupying a single memory cell. Nevertheless, this representation is wasteful for small alphabets; see e.g. [85]. If the alphabet Σ is of size σ, then a single character can be represented using $\lceil \log \sigma \rceil$ bits, which might be much less than W. Consequently, one may store a text $T \in \Sigma^n$ using a sequence of $n \lceil \log \sigma \rceil$ bits occupying $\left\lceil \frac{n \lceil \log \sigma \rceil}{W} \right\rceil$ consecutive memory cells. In the *packed representation* of T, we assume that the first character corresponds to the $\lceil \log \sigma \rceil$ least significant bits of the first cell and so on. Constant-time operations available in the word RAM model let us efficiently retrieve packed representations of substrings and perform some basic operations.

Proposition 3.2.1. *Let T be a text over an alphabet of size σ, stored in the packed representation. The packed representation of T of any length-ℓ fragment of T can be retrieved in $\mathcal{O}(\lceil \frac{\ell \log \sigma}{W} \rceil)$ time. Moreover, the length of the longest common prefix of two length-ℓ fragments can be computed in the same time.*

Proof. The bit sequence corresponding to any fragment of length ℓ is contained in the concatenation of at most $1 + \left\lceil \frac{\ell \lceil \log \sigma \rceil}{W} \right\rceil$ memory cells of the packed representation of T. Its location can be determined in $\mathcal{O}(1)$ time, and the resulting sequence can be aligned using $\mathcal{O}(\lceil \frac{\ell \log \sigma}{W} \rceil)$ bit-wise shift operations, as well as $\mathcal{O}(1)$ bit-wise 'and' operations to mask out the adjacent characters. This results in a packed representation of the length-ℓ fragment of T. In order to compute the length of the longest common prefix of two such fragments, we 'xor' the packed representations and find the position p of the least significant bit in the resulting sequence, repeatedly applying Corollary 3.1.2. The resulting length is $\left\lfloor \frac{p-1}{\lceil \log \sigma \rceil} \right\rfloor$ assuming 1-based indexing of positions. □

A particularly important case is that of binary strings, which we sometimes call *bitmasks*. A bitmask of length n can be stored in just $\left\lceil \frac{n}{W} \right\rceil$ memory cells.

3.3 Basic Queries

Let U be an arbitrary universe and let $\prec$ be a linear order on U. Consider a finite multiset $\mathcal{A} \subseteq U$. *Rank* queries, given an element $x \in U$, return *the rank of x in $\mathcal{A}$*, which is defined as the number (i.e., the total multiplicity) of elements $y \in \mathcal{A}$ not exceeding x:

$$\operatorname{rank}_{\mathcal{A}}(x) = |\{y \in \mathcal{A} : y \preceq x\}|.$$

Similarly, *selection* queries, given an integer $k \in [\![1, |\mathcal{A}|]\!]$, return the kth smallest element in $\mathcal{A}$ (1-based). Formally, $\operatorname{select}_{\mathcal{A}}(k)$ can be defined as the smallest $x \in U$ such that $\operatorname{rank}_{\mathcal{A}}(x) \geq k$. For $k \notin [\![1, |\mathcal{A}|]\!]$, we leave $\operatorname{select}_{\mathcal{A}}(k)$ undefined, i.e., $\operatorname{select}_{\mathcal{A}}(k) = \perp$.

In general, rank and select queries can be used to determine the *predecessor* and the *successor* of $x \in U$ in $\mathcal{A}$, i.e.,

$$\operatorname{pred}_{\mathcal{A}}(x) = \max\{y \in \mathcal{A} : y \preceq x\}$$

and

$$\operatorname{succ}_{\mathcal{A}}(x) = \min\{y \in \mathcal{A} : y \succ x\}.$$

Indeed, $\operatorname{pred}_{\mathcal{A}}(x) = \operatorname{select}_{\mathcal{A}}(\operatorname{rank}_{\mathcal{A}}(x))$ and $\operatorname{succ}_{\mathcal{A}}(x) = \operatorname{select}_{\mathcal{A}}(\operatorname{rank}_{\mathcal{A}}(x) + 1)$. In some natural settings, predecessor and successor queries are actually strictly easier than rank and selection; see e.g. Section 3.7.

3.4 Fusion Trees

In the word RAM model, the most natural universe U consists of W-bit integers (recall that W is the machine word size). In this framework, dynamic fusion trees by Pătraşcu and Thorup [126] provide a very efficient solution even in the static setting.

Theorem 3.4.1 (Fusion trees [126, 59])**.** *For every set $\mathcal{A}$ of W-bit integers, there exists a data structure of size $\mathcal{O}(|\mathcal{A}|)$ which answers* $\operatorname{rank}_{\mathcal{A}}$, $\operatorname{select}_{\mathcal{A}}$, $\operatorname{pred}_{\mathcal{A}}$, *and* $\operatorname{succ}_{\mathcal{A}}$ *queries in $\mathcal{O}(1 + \log_W |\mathcal{A}|)$ time. It can be constructed in $\mathcal{O}(|\mathcal{A}|)$ time if $\mathcal{A}$ is sorted in the input.*

Proof. The original static fusion trees by Fredman and Willard [59] match the claimed size and query time bound, but their construction procedure is not sufficiently fast. Pătraşcu and Thorup [126] dynamized fusion trees, with updates implemented in $\mathcal{O}(1 + \log_W |\mathcal{A}|)$ time, which immediately yields an $\mathcal{O}(|\mathcal{A}|(1 + \log_W |\mathcal{A}|))$-time construction. We observe that this running time can be easily improved to $\mathcal{O}(|\mathcal{A}|)$ in the static setting.

The main technical contribution of [126] is that the queries and updates are supported in $\mathcal{O}(1)$ time if $|\mathcal{A}| \leq W^{1/4}$; such components are called *dynamic fusion nodes.* The fusion tree is implemented in [126] as a B-tree [15] with $B = \Theta(W^{1/4})$ and keys (elements of $\mathcal{A}$) stored in the leaves. Each internal node is a dynamic fusion node storing the smallest elements of its children's subtrees. The total size of dynamic fusion nodes is $\mathcal{O}(|\mathcal{A}|)$, so the whole fusion tree can be constructed in $\mathcal{O}(|\mathcal{A}|)$ time provided that $\mathcal{A}$ is already sorted. Moreover, the $\operatorname{rank}_{\mathcal{A}}$ operation is easy to implement in $\mathcal{O}(1 + \log_W |\mathcal{A}|)$-time by traversing a root-to-leaf path, with rank queries on fusion nodes applied to retrieve the child to proceed to. To answer $\operatorname{select}_{\mathcal{A}}$ in the static setting, we can simply store $\mathcal{A}$ in an array. □

In some applications, we need to trade the linear size for faster queries.

Corollary 3.4.2. *For every set $\mathcal{A} \subseteq [\![1, n]\!]$ and parameter $\tau \in [\![1, n]\!]$, there exists a data structure of size $\mathcal{O}(|\mathcal{A}| + \frac{n}{\tau})$ which answers* $\operatorname{rank}_{\mathcal{A}}$, $\operatorname{select}_{\mathcal{A}}$, $\operatorname{pred}_{\mathcal{A}}$, *and* $\operatorname{succ}_{\mathcal{A}}$ *queries in $\mathcal{O}(1 + \log_W \tau)$ time. It can be constructed in $\mathcal{O}(|\mathcal{A}| + \frac{n}{\tau})$ time if $\mathcal{A}$ is sorted in the input.*

Proof. We partition the universe $[\![1,n]\!]$ into blocks $B_i = [\![1+(i-1)\tau, i\tau]\!]$. For each block, we store the fusion tree of $\mathcal{A}_i = \mathcal{A} \cap B_i$, as well as the values

$$r_i := \operatorname{rank}_{\mathcal{A}}((i-1)\tau) = |\mathcal{A}_1| + \cdots + |\mathcal{A}_{i-1}|.$$

This way, for each $x \in B_i$, we have

$$\operatorname{rank}_{\mathcal{A}}(x) = r_i + \operatorname{rank}_{\mathcal{A}_i}(x).$$

Consequently, $\operatorname{rank}_{\mathcal{A}}$ queries can be answered in $\mathcal{O}(\log_W \tau)$ time. For $\mathcal{O}(1)$-time $\operatorname{select}_{\mathcal{A}}$ queries, we simply store $\mathcal{A}$ in a sorted array. Each block requires $\mathcal{O}(|\mathcal{A}_i|+1)$ space and construction time, which is $\mathcal{O}(|\mathcal{A}| + \frac{n}{\tau})$ in total. □

3.4.1 Evaluating Functions Given by Step Representations

In Chapters 4 to 6, we repeatedly need to evaluate piecewise constant functions on integer domains. Below, we describe this simple application of Theorem 3.4.1 and Corollary 3.4.2.

Suppose we have a function f defined on $[\![1,n]\!]$ and that we are to support random access to values $f(x)$ for $x \in [\![1,n]\!]$. In general, the best we can do is to store f in an array of size n. If the values of f fit in $\mathcal{O}(1)$ machine words each, this takes $\mathcal{O}(n)$ space and $\mathcal{O}(1)$ evaluation time. However, we are often going to work with functions for which the values $f(x)$ and $f(x+1)$ are usually equal. For such functions, we identify *steps*, which are maximal integer intervals $I \subseteq [\![1,n]\!]$ such that f is constant on I. Formally, the step representation $\mathsf{Step}(f)$ of f is defined as a sequence $([\![\ell_j, r_j]\!], v_j)_{j=1}^{m}$ which satisfies the following conditions:

- $f(x) = v_j$ for $x \in [\![\ell_j, r_j]\!]$,
- $\ell_1 = 1$ and $r_m = n$,
- $\ell_{j+1} = r_j + 1$ for $1 \le j < m$,
- $v_j \ne v_{j+1}$ for $1 \le j < m$.

The value m is called the *size* of the step representation, denoted $|\mathsf{Step}(f)|$.

Observe that we have $x \in [\![\ell_j, r_j]\!]$ for $j = \operatorname{rank}_{\{\ell_i : 1 \le i \le m\}}(x)$. For this reason, Theorem 3.4.1 is useful to answer the evaluation queries, whose formal definition follows.

EVALUATION QUERIES
Input: The step representation $\mathsf{Step}(f)$ of a function $f : [\![1,n]\!] \to U$ whose values fit into $\mathcal{O}(1)$ machine words.
Queries: Given $x \in [\![1,n]\!]$, return the step $([\![\ell,r]\!], v) \in \mathsf{Step}(f)$ such that $x \in [\![\ell,r]\!]$ (in particular, $f(x) = v$).

Lemma 3.4.3. *For every function $f : [\![1,n]\!] \to U$ and trade-off parameter $\tau \in [\![1,n]\!]$, there is a data structure of size $\mathcal{O}(|\mathsf{Step}(f)| + \frac{n}{\tau})$ that answers evaluation queries in time $\mathcal{O}(\log_W \min(\tau, |\mathsf{Step}(f)|))$. It can be constructed from $\mathsf{Step}(f)$ in $\mathcal{O}(|\mathsf{Step}(f)| + \frac{n}{\tau})$ time.*

Proof. Let $\mathsf{Step}(f) = ([\![\ell_j, r_j]\!], v_j)_{j=1}^{m}$. We apply Theorem 3.4.1 to construct a data structure for rank queries in $\mathcal{L} = \{\ell_j : 1 \le j \le m\}$. Moreover, we store the $\mathsf{Step}(f)$ in a sorted array of size m. As we have already observed, the index of the step to be reported is $\operatorname{rank}_{\mathcal{L}}(i)$, so these components let us handle evaluation queries in $\mathcal{O}(\log_W m) = \mathcal{O}(\log_W |\mathsf{Step}(f)|)$ time after $\mathcal{O}(|\mathsf{Step}(f)|)$-time preprocessing.

If $\tau < |\mathsf{Step}(f)|$, we use Corollary 3.4.2 instead of Theorem 3.4.1 so that the query time becomes $\mathcal{O}(\log_W \tau)$ at the price of extra $\mathcal{O}(\frac{n}{\tau})$ terms in the size and construction time. □

For $\tau = W$ (or, in general, $\tau = W^{\mathcal{O}(1)}$), Lemma 3.4.3 yields constant query time.

Corollary 3.4.4. *For any function $f : [\![1, n]\!] \to U$, there exists a data structure of size $\mathcal{O}(|\mathsf{Step}(f)| + \frac{n}{W})$ that answers evaluation queries in $\mathcal{O}(1)$ time. It can be constructed in $\mathcal{O}(|\mathsf{Step}(f)| + \frac{n}{W})$ time given $\mathsf{Step}(f)$.*

3.5 Rank and Selection Queries on Bitmasks

In text processing, rank and selection queries are very often used for sets given as bitmasks. More precisely, a set $\mathcal{B} \subseteq [\![1, N]\!]$ can be represented using a bitmask $B[1\mathinner{.\,.} N]$ such that $B[i] = \mathbf{1}$ if and only if $i \in \mathcal{B}$. Following this convention, we define $\mathrm{rank}_B = \mathrm{rank}_{\mathcal{B}}$ and $\mathrm{select}_B = \mathrm{select}_{\mathcal{B}}$. In other words, $\mathrm{rank}_B(i)$ is the number of set bits ($\mathbf{1}$'s) in $B[1\mathinner{.\,.} i]$ and $\mathrm{select}_B(i)$ returns the position of the ith set bit (the ith $\mathbf{1}$) in B. Often, one also defines $\mathrm{rank}_{B,\mathbf{0}}$ and $\mathrm{select}_{B,\mathbf{0}}$ which operate on $\mathbf{0}$'s instead of $\mathbf{1}$'s. In this setting, $\mathrm{rank}_{B,\mathbf{1}}$ and $\mathrm{select}_{B,\mathbf{1}}$ are used to denote the standard versions dealing with $\mathbf{1}$'s. Also, B in the subscript is sometimes omitted if the bitmask is clear from the context.

Jacobson [77], Munro [113], and Clark [36] proved that one can build an additional structure of $o(N)$ bits so that rank_B and select_B queries can be answered in constant time. However, an efficient construction procedure was missing for many years. We have provided such an algorithm [12, Lemma 2.1] in parallel with Munro et al. [115, Theorem 5].

Proposition 3.5.1 ([12, 115])**.** *Given a bitmask $B[1\mathinner{.\,.} N]$ packed in $\left\lceil \frac{N}{W} \right\rceil$ machine words and a parameter $\tau \in [\![1, W]\!]$, we can extend it in $\mathcal{O}(\lceil \frac{N}{\tau} \rceil)$ time with a constant-time rank/select data structure occupying $\mathcal{O}(\lceil \frac{N \log \tau}{W\tau} \rceil)$ additional space, assuming $o(2^\tau)$ preprocessing time and bits of space shared by all instances of the data structure.*

3.6 Wavelet Trees

Recall that for a bitmask B and a symbol $c \in \{\mathbf{0}, \mathbf{1}\}$ we defined the $\mathrm{rank}_{B,c}$ and $\mathrm{select}_{B,c}$ functions based on the set of positions where c occurs in B. These notions naturally generalize to strings over a larger alphabet $\Sigma = [\![0, \sigma - 1]\!]$.

A standard tool for answering these queries in $\mathcal{O}(\log \sigma)$ time is the wavelet tree of a string. Invented by Grossi, Gupta, and Vitter [68] for space-efficient text indexing, it is an important data structure with a vast number of applications far beyond stringology (see [118] for a survey). In Section 3.7, we shall see that answering orthogonal range queries is among these applications. The data structure we develop in Chapter 8, called *wavelet suffix tree*, is based on wavelet trees of a certain shape.

In Section 3.6.1, we recall a formal definition of wavelet trees in the standard (binary, perfectly balanced) version. Next, in Section 3.6.2, we discuss efficient construction algorithms. We conclude by introducing wavelet trees of arbitrary shape in Section 3.6.3.

3.6.1 Wavelet Tree Definition

Consider a string s of length n over an alphabet $\Sigma = [\![0, \sigma - 1]\!]$. We assume that σ is a power of two (otherwise, we artificially extend Σ). The wavelet tree of s is defined as follows. First, we create the root node r and construct its bitmask B_r of length n. To build the bitmask, we think of every character $s[i]$ as of a binary number consisting of exactly $\log \sigma$ bits and put the most significant bit of $s[i]$ in $B_r[i]$. Then, we partition s into two subsequences s_0 and s_1 by scanning through s and appending the character $s[i]$ with the most significant bit removed to either s_0 or s_1, depending on whether the removed bit of $s[i]$ was **0** or **1**, respectively. Finally, we recursively define the wavelet trees for s_0 and s_1, which are strings over the alphabet $[\![0, \sigma/2 - 1]\!]$, and attach these trees to the root. We stop when the alphabet is unary. The final result is a perfect binary tree on σ leaves with a bitmask associated to every non-leaf node; see Figure 3.1 for an example.

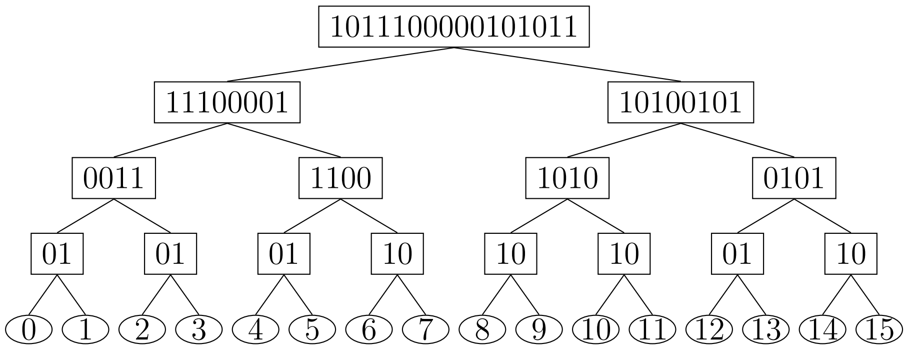


Figure 3.1: The wavelet tree for a string 12 7 11 15 9 6 4 0 1 2 10 3 13 5 8 14 = 1100_2 0111_2 1011_2 1111_2 1001_2 0110_2 0100_2 0000_2 0001_2 0010_2 1010_2 0011_2 1101_2 0101_2 1000_2 1110_2. The leaves are labeled with the corresponding characters $c \in \Sigma$.

Assuming that the edges are labeled by **0** or **1** depending on whether they go to the left or to the right, respectively, we can define the *label* of a node to be the concatenation of the labels of the edges on the path from the root to this node. This way, leaf labels are the binary representations of the characters in $[\![0, \sigma - 1]\!]$.

In virtually all applications, each bitmask B_r is augmented with a structure supporting constant-time rank and selection queries; see Proposition 3.5.1.

The bitmasks and their corresponding rank/selection structures are stored one after another, each starting at a new machine word. The total space occupied by the bitmasks alone is $\mathcal{O}(n \log \sigma)$ bits because there are $\log \sigma$ levels and the lengths of the bitmasks for all nodes at one level sum up to n. A rank/select structure built for a bitmask $B[1\,..\,N]$ takes $o(N)$ bits, so the space taken by all of them is $o(n \log \sigma)$ bits. Additionally, we might lose one machine word per node because of the word alignment, which sums up to $\mathcal{O}(\sigma)$. For efficient navigation, we number the nodes in a heap-like fashion and, using $\mathcal{O}(\sigma)$ space, store for every node the offset where its bitmasks and the corresponding rank/select structures begin. Thus, the total size of a wavelet tree is $\mathcal{O}(\sigma + n/\log_\sigma n)$ machine words, which is $\mathcal{O}(n/\log_\sigma n)$ for $\sigma = \mathcal{O}(n)$.

3.6.2 Binary Wavelet Tree Construction

Despite the importance of wavelet trees, until very recently there has not been much research concerning the efficient construction of this data structure. For a string s of length n, one can derive a construction algorithm with running time $\mathcal{O}(n \log \sigma)$ directly from the definition. Apart from this, two works [141, 37] present construction algorithms in the setting where only limited extra space is allowed. The running times of these procedures are higher than that of the naive algorithm.

One of our results (not included in the thesis) [12, Theorem 2.1] is a novel deterministic algorithm for constructing a wavelet tree in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time. In a parallel independent work, Munro et al. [115, Theorem 1] obtained essentially the same result.

Theorem 3.6.1 ([12, 115]). *Given the packed representation of a string s of length n over $\Sigma = [\![0, \sigma - 1]\!]$ for $\sigma = \mathcal{O}(n)$, we can construct its wavelet tree in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time.*

3.6.3 Arbitrarily-Shaped Wavelet Trees

Standard wavelet trees form a perfect binary tree with σ nodes, but different shapes have been introduced for several applications. Among others, this includes wavelet trees for Huffman encoding [58] and wavelet tries [69] that have the shape of a trie for a given set of strings.

In this setting, apart from the string s, we are given an arbitrary full binary tree $\mathcal{T}_{\mathsf{shp}}$ (i.e., a rooted tree whose nodes have 0 or 2 children) on σ leaves, together with a bijective mapping between the leaves and the characters in Σ. Then, while defining the bitmasks B_v, we do not remove the most significant bit of each character, and instead of partitioning the values based on this bit, we make a decision based on whether the leaf corresponding to the character lies in the left or in the right subtree of v. Both construction algorithms behind Theorem 3.6.1 generalize to such arbitrarily-shaped wavelet trees. The running time bound is preserved provided that the height of $\mathcal{T}_{\mathsf{shp}}$ is $\mathcal{O}(\log \sigma)$; see [12, Theorem 2.2] and [115, Theorem 2].

Theorem 3.6.2 ([12, 115]). *Let s be a string of length n over $\Sigma = [\![0, \sigma - 1]\!]$ for $\sigma = \mathcal{O}(n)$, and let $\mathcal{T}_{\mathsf{shp}}$ be a full binary tree of height $\mathcal{O}(\log \sigma)$ with σ leaves, each assigned a distinct character in Σ. The $\mathcal{T}_{\mathsf{shp}}$-shaped wavelet tree of s can be constructed in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time given $\mathcal{T}_{\mathsf{shp}}$ and the packed representation of s.*

3.7 Range Queries

An important setting for the basic queries defined in Section 3.3 is when the multiset $\mathcal{A}$ is defined as $A[R] := \{A[i] : i \in R\}$, where $A[1\,..\,n]$ is an array of integers from a universe U and $R = [\![\ell, r]\!]$ is a *range* contained in $[\![1, n]\!]$. Then, we obtain the following queries:

Range Rank Queries (Range Counting Queries)
Given a range R and an element $x \in U$, compute $\mathrm{rank}_{A[R]}(x)$.

Range Selection Queries
Given a range R and a value $k \in [\![1, |R|]\!]$, compute $\mathrm{select}_{A[R]}(k)$
(and an index $j \in R$ such that $A[j] = \mathrm{select}_{A[R]}(k)$).

Range Predecessor Queries
Given a range R and an element $x \in U$, compute $\mathrm{pred}_{A[R]}(x)$
(and an index $j \in R$ such that $A[j] = \mathrm{pred}_{A[R]}(x)$, if any).

Range Successor Queries (Range Next Value Queries)
Given a range R and an element $x \in U$, compute $\mathrm{succ}_{A[R]}(x)$
(and an index $j \in R$ such that $A[j] = \mathrm{succ}_{A[R]}(x)$, if any).

In the literature, range queries are sometimes defined in a slightly different way, with a set X of n points in a 2-dimensional space instead of the array A, which could be represented by points $(i, A[i])$. In this setting, range counting queries, for example, ask to count points in a given orthogonal region bounded from three sides (which is sufficient for counting points in an arbitrary orthogonal rectangle). Range predecessor and successor queries admit a similar interpretation. This is why these queries are often called (2-dimensional) *orthogonal range queries.* While the geometric interpretation formally makes the queries more general, a simple reduction (replacing the coordinates with their ranks) allows going back to the more restricted variant at a negligible cost and the state-of-the-art implementations exploit this phenomenon. Thus, we stick to the array setting, which is also more natural for applications contained in this thesis.

Range counting queries have been widely studied and the optimal query time is known.

Proposition 3.7.1 (JáJá et al. [78])**.** *There is a data structure of size $\mathcal{O}(n)$, which answers range counting queries in $\mathcal{O}(\frac{\log n}{\log\log n})$ time.*

Proposition 3.7.2 (Pătraşcu [125, 124])**.** *In the cell-probe model with W-bit cells, a static data structure of size $c \cdot n$ must take $\Omega(\frac{\log n}{\log c + \log W})$ time for range counting queries. The lower bounds already holds if $U = [\![1, n]\!]$ and the array A forms a permutation.*

More recently, Chan and Pătraşcu [32] designed an $\mathcal{O}(n\sqrt{\log n})$-time construction algorithm for a data structure satisfying Proposition 3.7.1. Recall that the wavelet tree for the array A (with elements replaced by their ranks to reduce the alphabet size to $\sigma \le n$) can be constructed in the same time. This is not a coincidence: such a wavelet tree can answer range counting queries in $\mathcal{O}(\log n)$ time, and its variant, which involves nodes of arity $\log^{\varepsilon} n$, achieves the optimal $\mathcal{O}(\log n/\log\log n)$ time. Moreover, one can see deep structural similarities between wavelet tree construction [12, 115] and the construction algorithm by Chan and Pătraşcu [32].

Results for range selection are analogous, but they came with a delay of a few years.

Proposition 3.7.3 (Brodal et al. [28])**.** *There is a data structure of size $\mathcal{O}(n)$, which answers range selection queries in $\mathcal{O}(\frac{\log n}{\log\log n})$ time.*

Proposition 3.7.4 (Jørgensen and Larsen [81])**.** *In the cell-probe model with W-bit cells, a static data structure of size $c \cdot n$ must take $\Omega(\frac{\log n}{\log c + \log W})$ time for range selection queries. The lower bounds already holds if $U = [\![1, n]\!]$ and A is a permutation.*

Chan and Pătraşcu [32] also gave an $\mathcal{O}(n\sqrt{\log n})$ time construction algorithm, but the query time of their data structure is $\mathcal{O}(\log n)$. An improvement came only after the applicability of wavelet trees became evident: we showed in [12] that multiary wavelet trees can be extended to support range selection in $\mathcal{O}(\log n/\log\log n)$ time.

Range successor and predecessor queries have been introduced more recently, motivated by several applications in text processing. The $\Omega(\log n/\log\log n)$ lower bound for the query time of $\mathcal{O}(n \log^{\mathcal{O}(1)} n)$-space data structures does not hold for these queries. The following three trade-offs describe the current state of the art.

Proposition 3.7.5 (Nekrich and Navarro [120], Zhou [144], Crochemore et al. [46])**.** *For the functions S_{rsucc} and Q_{rsucc} specified below, there is a data structure of size $S_{rsucc}(n)$ answering range predecessor and successor queries in $Q_{rsucc}(n)$ time:*

(a) $S_{rsucc}(n) = \mathcal{O}(n)$ and $Q_{rsucc}(n) = \mathcal{O}(\log^{\varepsilon} n)$ (for every constant $\varepsilon > 0$),

(b) $S_{rsucc}(n) = \mathcal{O}(n \log\log n)$ *and* $Q_{rsucc}(n) = \mathcal{O}(\log\log n)$*,*
(c) $S_{rsucc}(n) = \mathcal{O}(n^{1+\varepsilon})$ *and* $Q_{rsucc}(n) = \mathcal{O}(1)$ *(for every constant* $\varepsilon > 0$*).*

In a recent paper, Belazzougui and Puglisi [16] designed an efficient construction algorithm for a data structure (based on multiary wavelet trees) satisfying (a). Its running time is $C_{rsucc}(n) = \mathcal{O}(n\sqrt{\log n})$. The data structure for (c) can be constructed in $C_{rsucc}(n) = \mathcal{O}(n^{1+\varepsilon})$ time [46], while [144] does not give any construction algorithm for (b).

We conclude by recalling a famous special case of both range selection queries and range successor queries.

Range Minimum Queries (RMQ)
Given a range R and a value $k \in [\![1, |R|]\!]$, compute $\min\{A[R]\}$
(and an index $j \in R$ such that $A[j] = \min\{A[R]\}$).

Proposition 3.7.6 (Harel and Tarjan [73], Bender et al. [17])**.** *There is a data structure of size* $\mathcal{O}(n)$ *that answers range minimum queries in* $\mathcal{O}(1)$ *time. It can be constructed in* $\mathcal{O}(n)$ *time.*

Chapter 4

LCE Queries for Small Alphabets

In this chapter, we develop new data structures for the classic LCE Queries:

Longest Common Extension Queries
Given two positions i, i' in the text T, compute $\mathrm{LCE}(i,i') = \mathrm{lcp}(T[i\,..], T[i'\,..])$, i.e., the length of the longest common prefix of the suffixes starting at positions i and i'.

We already introduced these queries in Section 2.5 along with a standard $\mathcal{O}(n)$-size data structure answering LCE Queries in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time preprocessing. The $\mathcal{O}(n)$ size and construction time is optimal for texts over (polynomially-bounded) integer alphabets. However, if the alphabet size σ is significantly smaller than n, then the input text can be encoded in $\mathcal{O}(n \log \sigma)$ bits so that it fits within $\mathcal{O}(n/\log_\sigma n)$ machine words. Such a compact encoding, called the *packed representation*, is specified in Section 3.2.

Our main contribution in this chapter is a data structure of size $\mathcal{O}(n/\log_\sigma n)$ which answers LCE Queries in $\mathcal{O}(1)$ time and can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time from the packed representation of T; this is an optimal solution for the standard machine word size $W = \Theta(\log n)$. In our data structure, we apply techniques originating from LCE Queries in the read-only random access model, where the characters of the text T are available through an oracle and the text size is not counted towards the space complexity. As a side result, we present a data structure of size $\mathcal{O}(\frac{n}{\tau})$ (for a trade-off parameter $\tau \in [\![1, \lceil \frac{n}{2} \rceil]\!]$) which answers LCE Queries in $\mathcal{O}(\lceil \frac{\tau \log \sigma}{W} \rceil)$ time assuming oracle access to the packed representation of T. Its construction has not been optimized though: it requires $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ working space.

4.1 Overview

We follow an approach similar to that of many data structures answering LCE Queries in the read-only random access model [20, 22, 23, 24, 65, 139]. The high-level idea is to spend some space and preprocessing time so that computing $\mathrm{LCE}(i,i')$ is easier if both i and i' belong to a certain set $S \subseteq [\![1, n]\!]$ of *selected positions*. Given arbitrary positions i, i', one can start a naive scan checking if $T[i+\delta] = T[i'+\delta]$ for consecutive integers $\delta \ge 0$. If $\mathrm{LCE}(i,i')$ is small, then this procedure terminates quickly. Otherwise, we hope to reach a shift δ such that both $i+\delta \in S$ and $i'+\delta \in S$ so that we can apply the component for selected positions to calculate $\mathrm{LCE}(i,i') = \delta + \mathrm{LCE}(i+\delta, i'+\delta)$. In most solutions, this component allows extracting the precise value $\mathrm{LCE}(i+\delta, i'+\delta)$ or an additive approximation, in which case another naive scan is needed to determine the final answer. More sophisticated implementations involve multiple layers of selected positions.

The number of the selected positions is often subject to a trade-off between the space consumption and the query time. The positions are usually selected based on the text length n and a trade-off parameter τ only, even though the nearby selected positions $i+\delta, i'+\delta \in S$ are useful only if $\mathrm{LCE}(i,i')$ is relatively large. This observation has not been exploited prior to the recent work of Birenzwige et al. [24], who select positions based on *local consistency* [133, 132], a concept that has been successfully used for LCE QUERIES in dynamic [2, 64, 121] and grammar-compressed strings [74, 140].

In this setting, the decision to select a position $p \in [\![1,n]\!]$ depends only on the context $T[p-\Delta\,..\,p+\Delta]$ for some threshold parameter Δ rather than on the integer p. This way, essentially the same positions are selected within two matching fragments $T[i\,..\,j] \cong T[i'\,..\,j']$ (any differences must be within the boundary regions of length Δ). Consequently, if the set S is sufficiently dense, then we are guaranteed to find positions $i+\delta, i'+\delta \in S$ relatively early while computing $\mathrm{LCE}(i,i')$. Unfortunately, this method fails for very highly periodic regions of T. For example, if $T = \mathsf{a}^n$, then we would need to choose all positions $\Delta+1,\ldots,n-\Delta$ or none of them. We cannot select $\Theta(n)$ positions, so the query algorithm in [24] is adapted so that it skips periodic regions without selected positions.

Our data structure is based on the same ideas as the one in [24], but the deterministic construction algorithms require a novel approach. Moreover, we provide an alternative formalization which results in a query procedure seamlessly handling periodic fragments.

The central notion of our approach is that of a *τ-synchronizing function* (for a trade-off parameter $\tau \in [\![1, \lceil \frac{n}{2} \rceil]\!]$), introduced in Section 4.2. In the language of selected positions introduced above, it *consistently* assigns selected positions to fragments of length $2\tau-1$ (so that matching fragments obtain analogous positions). In our interpretation, the selected positions represent *synchronizing fragments* of length τ starting there and are associated with integer identifiers of the underlying substrings. A single synchronizing fragment is usually assigned to multiple subsequent fragments of length $2\tau-1$, so we use the *step representations* (of Section 3.4.1) to store synchronizing functions.

In Section 4.3, we show how a synchronizing function (with the associated identifiers) can be used to efficiently approximate $\mathrm{LCE}(i,i')$ in a certain sense. Our approach incurs a constant-factor overhead in space consumption (compared to the size of the step representation) and does not require access to the text T. The central technical contribution of this chapter is presented in Section 4.4, where we describe two deterministic algorithms constructing τ-synchronizing functions with step representations of size $\mathcal{O}(\frac{n}{\tau})$. The first procedure takes $\mathcal{O}(n)$ time for any given $\tau \in [\![1, \lceil \frac{n}{2} \rceil]\!]$, while the second one runs in $\mathcal{O}(n/\tau)$ time for a specific value $\tau = \Theta(\log_\sigma n)$. We conclude with Section 4.5, where all the components are combined to develop the announced data structures for LCE QUERIES.

4.2 Fundamental Concepts

Throughout this chapter, we fix a text T of length n. We denote by F the family of (non-empty) fragments of T and for $x \in \mathsf{F}$, we distinguish the subset $\mathsf{F}[x] \subseteq \mathsf{F}$ of all fragments contained in x. For a fixed length $m \in [\![1,n]\!]$, we also identify subsets $\mathsf{F}_m \subseteq \mathsf{F}$ and $\mathsf{F}_m[x] \subseteq \mathsf{F}[x]$ which consist of length-m fragments. We also introduce a family

$$\mathsf{N} = \{x \in \mathsf{F} : \mathrm{per}(x) > \tfrac{1}{3}|x|\}$$

of *non-highly-periodic* fragments of T and its subfamilies N_m, $\mathsf{N}[x]$, and $\mathsf{N}_m[x]$ defined for any $m \in [\![1, n]\!]$ and $x \in \mathsf{F}$.

Observe that for every m there is a natural bijection mapping $[\![1, n-m+1]\!]$ to F_m. We denote it by F_m so that $F_m(i) := T[i \mathinner{.\,.} i+m-1]$ for $i \in [\![1, n-m+1]\!]$.

The single most important concept for our data structure is that of a τ-synchronizing function; its intuitive interpretation is provided in the overview above. Note that there is much freedom on when to leave the value of the synchronizing function undefined, and our specific choice below is somewhat arbitrary.

Definition 4.2.1. *Consider a text T of length n and a parameter $\tau \in [\![1, \lceil \frac{n}{2} \rceil]\!]$. We say that a function $\mathsf{sync} : \mathsf{F}_{2\tau-1} \to \mathsf{F}_\tau \cup \{\bot\}$ is a τ-*synchronizing function *if it satisfies the following conditions for each fragment $x \in \mathsf{F}_{2\tau-1}$:*

(a) *if $\mathsf{sync}(x) = \bot$, then $\mathsf{N}_\tau[x] = \emptyset$;*
(b) *if $\mathsf{sync}(x) \neq \bot$, then $\mathsf{sync}(x) \in \mathsf{N}_\tau[x]$;*
(c) *if $\mathsf{sync}(x) = x[i \mathinner{.\,.} i+\tau-1]$ for some position i, then $\mathsf{sync}(x') = x'[i \mathinner{.\,.} i+\tau-1]$ for each fragment $x' \in \mathsf{F}_{2\tau-1}$ matching x (satisfying $x \cong x'$).*

Fragments in the image $\mathsf{sync}(\mathsf{F}_{2\tau-1})$ are called the synchronizing fragments *(of sync).*

The τ-synchronizing functions sync used in our data structure often assign the same value to subsequent fragments $F_{2\tau-1}(i)$ and $F_{2\tau-1}(i+1)$. Thus, we store sync using the *step representation* of $\mathsf{sync} \circ F_{2\tau-1}$, as defined in Section 3.4.1. We slightly abuse notation and write $\mathsf{Step}(\mathsf{sync})$ instead of $\mathsf{Step}(\mathsf{sync} \circ F_{2\tau-1})$.

We also introduce a partition $\mathcal{P}_m = \mathsf{F}_m/{\cong}$ so that $x, x' \in \mathsf{F}_m$ belong to the same equivalence class if and only if they match. This way, each length-m substring of T corresponds to a class $P \in \mathcal{P}_m$ of its occurrences. Our algorithms often represent the partition $\mathcal{P}_m$ using an *m-identifier function* $\mathsf{id} : \mathsf{F}_m \to [\![1, n^{\mathcal{O}(1)}]\!]$ such that $\mathsf{id}(x) = \mathsf{id}(x')$ if and only if x and x' match.

4.3 Approximate LCE with Synchronizing Functions

The key intuition behind Definition 4.2.1 is that a synchronizing function must consistently handle matching fragments. Formally, we say that fragments $x, x' \in \mathsf{F}_{2\tau-1}$ are *consistent with respect to* sync, denoted $x \sim_{\mathsf{sync}} x'$, if $\mathsf{sync}(x) = \mathsf{sync}(x') = \bot$ or $\mathsf{sync}(x) = x[i \mathinner{.\,.} i+\tau-1] \cong x'[i \mathinner{.\,.} i+\tau-1] = \mathsf{sync}(x')$ for some position i.

Fact 4.3.1. *If fragments $x, x' \in \mathsf{F}_{2\tau-1}$ match (i.e., $x \cong x'$), then they are consistent with respect to any τ-synchronizing function sync (i.e., $x \sim_{\mathsf{sync}} x'$).*

Proof. If $\mathsf{sync}(x) = \bot$, then $\mathsf{N}_\tau[x] = \emptyset$ by Definition 4.2.1(a), which implies $\mathsf{N}_\tau[x'] = \emptyset$ due to the natural bijection between $\mathsf{F}[x]$ and $\mathsf{F}[x']$. Consequently, $\mathsf{sync}(x') = \bot$ by Definition 4.2.1(b).

Otherwise, $\mathsf{sync}(x) = x[i \mathinner{.\,.} i+\tau-1]$ for some position i, so $\mathsf{sync}(x') = x'[i \mathinner{.\,.} i+\tau-1]$ by Definition 4.2.1(c). Moreover, this yields $\mathsf{sync}(x) \cong \mathsf{sync}(x')$ due to $x \cong x'$. □

The inverse implication does not hold in general, but consistency can be used to approximate the values $\mathrm{LCE}(i, i')$. For this, we define $\mathrm{LCCE}_{\mathsf{sync}}(i, i')$ as the largest integer Δ such that $F_{2\tau-1}(i+\delta) \sim_{\mathsf{sync}} F_{2\tau-1}(i'+\delta)$ for $0 \le \delta < \Delta$.

Lemma 4.3.2. *If two positions i, i' in T satisfy* $\mathrm{LCE}(i, i') \geq \tau - 1$, *then*

$$\mathrm{LCCE}_{\mathsf{sync}}(i, i') + (\tau - 1) \leq \mathrm{LCE}(i, i') \leq \mathrm{LCCE}_{\mathsf{sync}}(i, i') + 2(\tau - 1)$$

holds for every τ-synchronizing function sync.

Proof. To prove the upper bound on $\mathrm{LCE}(i, i')$, let us take $\delta \in [\![0, \mathrm{LCE}(i, i') - (2\tau - 1)]\!]$ and consider fragments $x = F_{2\tau-1}(i + \delta)$ and $x' = F_{2\tau-1}(i' + \delta)$. Note that $x \cong x'$, so $x \sim_{\mathsf{sync}} x'$ due to Fact 4.3.1. Thus, $\mathrm{LCCE}_{\mathsf{sync}}(i, i') \geq \mathrm{LCE}(i, i') - 2(\tau - 1)$, as claimed.

For the other inequality, we inductively prove that $\mathrm{LCE}(i + \delta, i' + \delta) \geq \tau$ holds for every $\delta \in [\![0, \mathrm{LCCE}_{\mathsf{sync}}(i, i') - 1]\!]$. The assumption (for $\delta = 0$) or the inductive hypothesis (for $\delta > 0$) yields $\mathrm{LCE}(i + \delta, i' + \delta) \geq \tau - 1$. Moreover, the fragments $x = F_{2\tau-1}(i + \delta)$ and $x' = F_{2\tau-1}(i' + \delta)$ are consistent with respect to sync by definition of δ. This can be due to one of the two reasons.

The first possibility is that $\mathsf{sync}(x) = \mathsf{sync}(x') = \bot$. Definition 4.2.1(a) implies $\mathsf{N}_\tau[x] = \mathsf{N}_\tau[x'] = \emptyset$, so $p = \mathrm{per}(x[1\,..\,\tau])$ and $p' = \mathrm{per}(x'[1\,..\,\tau])$ are both at most $\frac{\tau}{3}$. Applying the Periodicity Lemma (Lemma 2.2.1), we conclude that $\gcd(p, p')$ is a period of the common prefix $x[1\,..\,\tau - 1] \cong x'[1\,..\,\tau - 1]$. Thus, $x[\tau] = x[\tau - p] = x[\tau - p'] = x'[\tau - p'] = x'[\tau]$, i.e., $\mathrm{lcp}(x, x') \geq \tau$.

The other possibility is that $\mathsf{sync}(x) = x[i\,..\,i + \tau - 1] \cong x'[i\,..\,i + \tau - 1] = \mathsf{sync}(x')$ for some position $i \in [\![1, \tau]\!]$. In particular, this yields $i \leq \tau \leq i + \tau - 1$, so $\mathsf{sync}(x) \cong \mathsf{sync}(x')$ implies $x[\tau] = x'[\tau]$ and $\mathrm{lcp}(x, x') \geq \tau$.

Consequently, $\mathrm{LCE}(i + \delta, i' + \delta) \geq \tau$ for each $\delta \in [\![0, \mathrm{LCCE}_{\mathsf{sync}}(i, i') - 1]\!]$, which means that $\mathrm{LCE}(i, i') \geq \mathrm{LCCE}_{\mathsf{sync}}(i, i') + \tau - 1$ holds as claimed. □

In the light of Lemma 4.3.2, the value $\mathrm{LCCE}_{\mathsf{sync}}(i, i')$ for $i, i' \in [\![1, n]\!]$ can be used to approximate $\mathrm{LCE}(i, i')$. The following fact lets us utilize the step representation $\mathsf{Step}(\mathsf{sync})$ in the computation of $\mathrm{LCCE}_{\mathsf{sync}}(i, i')$.

Fact 4.3.3. *Let* sync *be a τ-synchronizing function, and let $x, y, x', y' \in \mathsf{F}_{2\tau-1}$ be fragments starting at positions i, $i+1$, i', and $i'+1$, respectively. If $x \sim_{\mathsf{sync}} x'$ and* $\mathsf{sync}(x) = \mathsf{sync}(y)$, *then* $\mathsf{sync}(x') = \mathsf{sync}(y')$ *is equivalent to $y \sim_{\mathsf{sync}} y'$.*

Proof. First, suppose that $\mathsf{sync}(x) = \mathsf{sync}(x') = \bot$. We have $\mathsf{sync}(y) = \mathsf{sync}(x) = \bot$, so $y \sim_{\mathsf{sync}} y'$ is equivalent to $\mathsf{sync}(y') = \bot$, i.e., to $\mathsf{sync}(y') = \mathsf{sync}(x')$.

The other case is that $\mathsf{sync}(x) = x[j\,..\,j + \tau - 1] \cong x'[j\,..\,j + \tau - 1] = \mathsf{sync}(x')$ for some position j. The equality $\mathsf{sync}(y) = \mathsf{sync}(x)$ yields $\mathsf{sync}(y) = y[j - 1\,..\,j + \tau - 2]$, so $y \sim_{\mathsf{sync}} y'$ is equivalent to $\mathsf{sync}(y') = y'[j - 1\,..\,j + \tau - 2]$, i.e., to $\mathsf{sync}(y') = \mathsf{sync}(x')$. □

Due to the property formulated above, computing $\mathrm{LCCE}_{\mathsf{sync}}(i, i')$ based on $\mathsf{Step}(\mathsf{sync})$ turns out to be closely related to answering LCE QUERIES in run-length encoded strings. We apply this intuition to design the following component.

Lemma 4.3.4. *If $|\mathsf{Step}(\mathsf{sync})| \leq M$ for a τ-synchronizing function* sync *and an integer M, then there exists a data structure of size $\mathcal{O}(M)$ which computes* $\mathrm{LCCE}_{\mathsf{sync}}(i, i')$ *for any positions i, i' of T in $\mathcal{O}(\log_W \frac{n}{M})$ time. For any $\varepsilon > 0$, one can build it in $\mathcal{O}(M + n^\varepsilon)$ time given* $\mathsf{Step}(\mathsf{sync})$, *the value M, and a τ-identifier* $\mathsf{id}(y)$ *for each synchronizing fragment y.*

Proof. Let $\mathsf{Step}(\mathsf{sync}) = ([\![\ell_j, \ell_{j+1} - 1]\!], y_j)_{j=1}^{m}$ be the step representation of $\mathsf{sync} \circ F_{2\tau-1}$. Below, we reduce the queries to computation of $\mathrm{LCCE}_{\mathsf{sync}}(\ell_j, \ell_{j'})$ for $1 \leq j, j' \leq m$. Thus, for $j \in [\![1, m]\!]$ we define $x_j = F_{2\tau-1}(\ell_j)$ and δ_j so that $\delta_j = \bot$ if $y_j = \bot$ and

$y_j = x_j[\delta_j \mathinner{.\,.} \delta_j + \tau - 1]$ otherwise. Observe that $F_{2\tau-1}(\ell_j) \sim_{\mathsf{sync}} F_{2\tau-1}(\ell_{j'})$ if and only if $\delta_j = \delta_{j'}$ and $\mathsf{id}(y_j) = \mathsf{id}(y_{j'})$ (assuming $\mathsf{id}(\bot) = \bot$). Moreover, repeated application of Fact 4.3.3 yields $\mathrm{LCCE}_{\mathsf{sync}}(\ell_j, \ell_{j'}) = \ell_{j+1} - \ell_j + \mathrm{LCCE}_{\mathsf{sync}}(\ell_{j+1}, \ell_{j'+1})$ if additionally $\ell_{j+1} - \ell_j = \ell_{j'+1} - \ell_{j'}$.

Consequently, our data structure consists of the following two components of size $\mathcal{O}(M)$ each:

- a string V of length m such that $V[j] = (\delta_j, \mathsf{id}(y_j), \ell_{j+1} - \ell_j)$ equipped with the component of Proposition 2.5.1 for LCE Queries, and
- the set $\mathcal{L} = \{\ell_j : 1 \le j \le m\}$ equipped with the component of Corollary 3.4.2 for $\mathrm{rank}_{\mathcal{L}}$ queries, constructed for a trade-off parameter $\frac{n}{M}$.

The string V is easy to build in $\mathcal{O}(m)$ time because $\mathsf{id}(y)$ is available for each synchronizing fragment y. Sorting the characters of V takes $\mathcal{O}(M + n^{\varepsilon})$ time because each character consists of three integers bounded by $\mathcal{O}(n)$. After this alphabet reduction, Proposition 2.5.1 guarantees $\mathcal{O}(m)$ construction time. The data structure of Corollary 3.4.2 is built in $\mathcal{O}(m + M) = \mathcal{O}(M)$ time. This completes the description of the construction algorithm.

Now, suppose that we are to handle a query asking for $\mathrm{LCCE}_{\mathsf{sync}}(i, i')$. We assume that $i, i' \in [\![1, n - 2(\tau - 1)]\!]$ (otherwise, we immediately return 0). Our first goal is to compute $\mathsf{sync}(x)$ for the fragment $x = F_{2\tau-1}(i)$. For this, we retrieve $j = \mathrm{rank}_{\mathcal{L}}(i)$ and observe that $\mathsf{sync}(x) = \bot$ if $\delta_j = \bot$ and $\mathsf{sync}(x) = \mathsf{sync}(x_j) = x_j[\delta_j \mathinner{.\,.} \delta_j + \tau - 1] = F_\tau(\ell_j + \delta_j - 1)$ otherwise. We repeat the same process for $x' = F_{2\tau-1}(i')$ so that we can test whether $x \sim_{\mathsf{sync}} x'$. If the fragments x and x' are not consistent with respect to sync, we must return 0, while in the other case we may use Fact 4.3.3. Its repeated application yields

$$\mathrm{LCCE}_{\mathsf{sync}}(i, i') = \begin{cases} \min(\ell_{j+1} - i, \ell_{j'+1} - i) & \text{if } \ell_{j+1} - i \ne \ell_{j'+1} - i', \\ \ell_{j+1} - i + \mathrm{LCCE}_{\mathsf{sync}}(\ell_{j+1}, \ell_{j'+1}) & \text{otherwise.} \end{cases}$$

Further processing is needed in the second case only. We make an LCE Query in V, asking for the length d of the longest common prefix of $V[j+1 \mathinner{.\,.} m]$ and $V[j'+1 \mathinner{.\,.} m]$ and conclude that

$$\mathrm{LCCE}_{\mathsf{sync}}(i, i') = \ell_{j+1} - i + \mathrm{LCCE}_{\mathsf{sync}}(\ell_{j+1}, \ell_{j'+1}) = \ell_{j+d+1} - i + \mathrm{LCCE}_{\mathsf{sync}}(\ell_{j+d+1}, \ell_{j'+d+1}).$$

To compute the latter $\mathrm{LCCE}_{\mathsf{sync}}$ value, we recursively run the query algorithm. Observe that $V[j+d+1] \ne V[j'+d+1]$ implies $F_{2\tau-1}(\ell_{j+d+1}) \not\sim_{\mathsf{sync}} F_{2\tau-1}(\ell_{j'+d+1})$ or $\ell_{j+d+2} - \ell_{j+d+1} \ne \ell_{j'+d+2} - \ell_{j'+d+1}$, so no further recursive call will be made.

The total query time is constant except for the $\mathrm{rank}_{\mathcal{L}}$ queries, which take $\mathcal{O}(\log_W \frac{n}{M})$ time due to Corollary 3.4.2. □

4.4 Synchronizing Function Construction

In this section, we design algorithms that construct τ-synchronizing functions with step representations of size $\mathcal{O}(\frac{n}{\tau})$. Our procedures rely on a single generic scheme, applying a carefully chosen τ-identifier function to derive a τ-synchronizing function.

Construction 4.4.1. *Let* id *be a τ-identifier function, and let $\tau \in [\![1, \lceil \frac{n}{2} \rceil]\!]$. For a fragment $x \in \mathsf{F}_{2\tau-1}$, we define* $\mathsf{sync}(x) = \bot$ *if* $\mathsf{N}_\tau[x] = \emptyset$. *Otherwise,* $\mathsf{sync}(x)$ *is the leftmost fragment $y \in \mathsf{N}_\tau[x]$ which minimizes* $\mathsf{id}(y)$.

Construction 4.4.1 clearly yields a well-defined function sync. Below, we prove that it is a τ-synchronizing function.

Fact 4.4.2. *The function* sync *defined with Construction 4.4.1 satisfies Definition 4.2.1.*

Proof. Conditions (a) and (b) are clearly satisfied. As for condition (c), we observe that the natural bijection between $\mathsf{F}_\tau[x]$ and $\mathsf{F}_\tau[x']$ for matching fragments $x, x' \in \mathsf{F}_{2\tau-1}$ preserves the τ-identifiers: $\mathsf{id}(x[i\,..\,i+\tau-1]) = \mathsf{id}(x'[i\,..\,i+\tau-1])$ for $i \in [\![1,\tau]\!]$. □

The main challenge in building a τ-synchronizing function sync with Construction 4.4.1 is to choose an appropriate τ-identifier id function so that $\mathsf{Step}(\mathsf{sync})$ is small. As we show in Section 4.4.1, choosing id uniformly at random leads to satisfactory results if no length-τ fragment is highly periodic. We aim at deterministic algorithms, so we also design an $\mathcal{O}(n)$-time deterministic construction procedure resulting in $|\mathsf{Step}(\mathsf{sync})| = \mathcal{O}(\frac{n}{\tau})$. We study the structure of highly periodic length-τ fragments in Section 4.4.2 in order to drop the assumption $\mathsf{N}_\tau = \mathsf{F}_\tau$ in the subsequent Section 4.4.3, where we adapt both our constructions so that they work for arbitrary strings. We conclude in Section 4.4.4 with an $\mathcal{O}(n/\log_\sigma n)$-time procedure which constructs the step representation of a $\left\lfloor \frac{1}{6}\log_\sigma n\right\rfloor$-synchronizing function given the packed representation of T.

4.4.1 Construction for Texts with $\mathsf{N}_\tau = \mathsf{F}_\tau$

The key feature of non-highly-periodic strings is that their occurrences cannot overlap too much. To formalize this property, we say that a set $A \subseteq \mathsf{F}$ is *d-sparse* if the starting positions i, i' of any distinct $x, x' \in A$ satisfy $|i - i'| > d$.

Fact 4.4.3. *An equivalence class* $P \in \mathcal{P}_\tau$ *is* $\frac{1}{3}\tau$*-sparse if* $P \subseteq \mathsf{N}_\tau$*.*

Proof. Suppose that $F_\tau(i), F_\tau(i') \in P$ for positions i, i' such that $i < i' \le i + \frac{1}{3}\tau$. We have $T[i\,..\,i+\tau-1] \cong T[i'\,..\,i'+\tau-1]$, so $\operatorname{per}(T[i\,..\,i'+\tau-1]) \le i' - i \le \frac{1}{3}\tau$. In particular, this yields $\operatorname{per}(T[i\,..\,i+\tau-1]) \le \frac{1}{3}\tau$, so $F_\tau(i) \notin \mathsf{N}_\tau$, i.e., $P \not\subseteq \mathsf{N}_\tau$. □

As announced, we start with an existential proof based on the probabilistic method.

Lemma 4.4.4. *Let T be a text of length n and let $\tau \in [\![1, \lceil \frac{n}{2} \rceil]\!]$. If $\mathsf{N}_\tau = \mathsf{F}_\tau$, then there exists a τ-identifier function* id *such that Construction 4.4.1 results in* $|\mathsf{Step}(\mathsf{sync})| \le \frac{6n}{\tau}$*.*

Proof. We shall prove that $\mathbb{E}[|\mathsf{Step}(\mathsf{sync})|] \le \frac{6n}{\tau}$ if id is uniformly random. Formally, we construct a uniformly random bijection π mapping $\mathcal{P}_\tau$ to $[\![1, |\mathcal{P}_\tau|]\!]$ and define $\mathsf{id}(x) = \pi(P)$ for $x \in P$.

Observe that for each fragment $x \in \mathsf{F}_{2\tau-1}$, we have $|\mathsf{N}_\tau[x]| = \tau$. Moreover, Fact 4.4.3 guarantees that $|\mathsf{N}_\tau[x] \cap P| \le 3$ for each class $P \in \mathcal{P}_\tau$. Thus, fragments in $\mathsf{N}_\tau[x]$ belong to at least $\frac{\tau}{3}$ distinct classes. Each of these classes has the same probability of having the smallest identifier, so $\mathbb{P}[\mathsf{sync}(x) = y] \le \frac{3}{\tau}$ for any $y \in \mathsf{N}_\tau[x]$. Next, consider $x = F_{2\tau-1}(i)$ and $x' = F_{2\tau-1}(i+1)$. If $\mathsf{sync}(x)$ and $\mathsf{sync}(x')$ both belong to $\mathsf{N}_\tau[x] \cap \mathsf{N}_\tau[x']$, then $\mathsf{sync}(x) = \mathsf{sync}(x')$ by Construction 4.4.1. Consequently,

$$\mathbb{P}[\mathsf{sync}(x) \ne \mathsf{sync}(x')] \le \mathbb{P}[\mathsf{sync}(x) = x[1\,..\,\tau]] + \mathbb{P}[\mathsf{sync}(x') = x'[\tau\,..\,2\tau-1]] \le \tfrac{3}{\tau} + \tfrac{3}{\tau} = \tfrac{6}{\tau}.$$

By linearity of expectation, we conclude that

$$\mathbb{E}[|\mathsf{Step}(\mathsf{sync})|] = 1 + \sum_{i=1}^{n-(2\tau-1)} \mathbb{P}[\mathsf{sync}(F_{2\tau-1}(i)) \ne \mathsf{sync}(F_{2\tau-1}(i+1))] \le 1 + \tfrac{6(n-(2\tau-1))}{\tau} \le \tfrac{6n}{\tau}.$$

In particular, we must have $|\mathsf{Step}(\mathsf{sync})| \le \frac{6n}{\tau}$ for some τ-synchronizing function id. □

Next, we provide an efficient deterministic construction. The idea is to assign the consecutive positive integers, one at a time, to classes $P \in \mathcal{P}_\tau$. Our choice is guided by a scoring function carefully designed to keep $|\mathsf{Step}(\mathsf{sync})|$ low.

Lemma 4.4.5. *Given a text T of length n and a parameter $\tau \in [\![1, \lceil \frac{n}{2} \rceil]\!]$ such that $\mathsf{N}_\tau = \mathsf{F}_\tau$, in $\mathcal{O}(n)$ time one can construct a τ-identifier function* id *and a τ-synchronizing function* sync *(defined with Construction 4.4.1 based on* id*) such that $|\mathsf{Step}(\mathsf{sync})| \le \frac{9n}{\tau}$.*

Proof. First, we build the partition $\mathcal{P}_\tau$ and sort the fragments in each class according to the left-to-right order. A simple $\mathcal{O}(n)$-time implementation is based on the suffix array and the LCP table of T (see Section 2.5): We cut the suffix array before any position i with $LCP[i] < \tau$ and remove the positions i with $SA[i] > n - \tau + 1$. For each of the remaining regions $SA[\ell \mathinner{.\,.} r]$, $\{F_\tau(SA[i]) : i \in [\![\ell, r]\!]\}$ belongs to $\mathcal{P}_\tau$. Hence, we initialize a new object for this class $P \in \mathcal{P}_\tau$ and store a pointer to this object at each fragment $F_\tau(SA[i]) \in P$. We then iterate over all length-τ fragments in the left to right order, and we append each fragment to the list constructed for the class it belongs to.

Next, we iteratively construct the functions id and sync. Initially, each value $\mathsf{id}(y)$ and $\mathsf{sync}(x)$ is undefined ($\perp$). In the jth iteration, we choose a partition class $P_j \in \mathcal{P}_\tau$ and process $y \in P_j$ from left to right: we assign $\mathsf{id}(y) = j$ and set $\mathsf{sync}(x) = y$ for all $x \in \mathsf{F}_{2\tau-1}$ such that $\mathsf{sync}(x) = \perp$ and $y \in \mathsf{F}_\tau[x]$. Due to the order of identifiers assigned and fragments processed, this results in a function sync compatible with Construction 4.4.1.

To define the scoring function, we distinguish *active blocks*, which are inclusion-wise maximal fragments $z \in \mathsf{F}$ such that $|z| \ge 2\tau - 1$ and $\mathsf{id}(y) = \perp$ for each $y \in \mathsf{F}_\tau[z]$; note that $\mathsf{sync}(x) = \perp$ holds for $x \in \mathsf{F}_{2\tau-1}$ if and only if x is contained in an active block. For each active block z, we assign *scores* to *active fragments* $y \in \mathsf{F}_\tau[z]$. The score is -1 for the leftmost and the rightmost $\left\lfloor \frac{1}{3}\tau \right\rfloor$ fragments in $\mathsf{F}_\tau[z]$, and $+2$ for the remaining fragments in $\mathsf{F}_\tau[z]$. Note that $|\mathsf{F}_\tau[z]| \ge \tau$, so the total score is non-negative.

We explicitly maintain the aggregate score of active fragments from each partition class $P \in \mathcal{P}_\tau$, and a collection $\mathcal{P}^+_\tau \subseteq \mathcal{P}_\tau$ of unprocessed classes with non-negative aggregate scores. The class P_j to be processed in the jth iteration is chosen arbitrarily from $\mathcal{P}^+_\tau$; such a class exists because the total score is non-negative and the already processed classes do not contain active fragments.

Having chosen P_j, we need to update the maintained data. For every $y \in P_j$, we assign $\mathsf{id}(y) = j$ and, if $y \in P_j$ is active, we set $\mathsf{sync}(x) = y$ for each $x \in \mathsf{F}_{2\tau-1}$ such that $\mathsf{sync}(x) = \perp$ and $y \in \mathsf{F}_\tau[x]$. As a result, some fragments may cease to be active and the score of some active fragments may change. Nevertheless, this happens only if y is active and the affected fragments overlap y. Thus, $\mathcal{O}(\tau)$ time per active fragment $y \in P_j$ is sufficient to process these changes and amend the aggregate scores of classes $P \in \mathcal{P}_\tau$, possibly moving these classes into $\mathcal{P}^+_\tau$ or out of this collection.

To analyze the running time, we define sets $A_j \subseteq P_j$ of fragments which were active prior to processing P_j and $A^+_j \subseteq A_j$ of active fragments which had score $+2$ at that time. Note that $|A_j| \le 3|A^+_j|$ because the aggregate score of P_j was non-negative. The running time of the jth iteration is therefore $\mathcal{O}(|P_j| + \tau|A_j|) = \mathcal{O}(|P_j| + \tau|A^+_j|)$, and the number of steps introduced in sync is at most $|A_j| \le 3|A^+_j|$. Thus, the overall number of steps is $|\mathsf{Step}(\mathsf{sync})| \le 3|A^+|$ and the total running time is $\mathcal{O}(n + \tau|A^+|)$, where $A^+ = \bigcup_j A^+_j$.

Consequently, our final goal is to bound the size of A^+. We shall prove that this set is $\frac{1}{3}\tau$-sparse. Consider two distinct fragments $y, y' \in A^+$ such that $y \in A^+_j$ and $y' \in A^+_{j'}$. Due to Fact 4.4.3, A^+_j is $\frac{1}{3}\tau$-sparse. Thus, we may assume without loss of generality that $j' < j$. Prior to processing P_j, the fragment y' was not active and y had score $+2$. Hence, there must have been at least $\lfloor\frac{1}{3}\tau\rfloor$ active fragments with score -1 in between, so the starting positions of y and y' are at distance at least $1+\lfloor\frac{1}{3}\tau\rfloor > \frac{1}{3}\tau$. Consequently, $|A^+| \le \frac{3n}{\tau}$, so $|\mathsf{Step}(\mathsf{sync})| \le \frac{9n}{\tau}$ and the overall running time of the construction procedure is $\mathcal{O}(n)$. □

4.4.2 Structure of Highly Periodic Fragments

In this section, we study the structure of the set N_τ of the non-highly-periodic length-τ fragments of T so that the approach from Section 4.4.1 can be generalized to arbitrary texts. The probabilistic argument in the proof of Lemma 4.4.4 relies on a large number of possibilities for $\mathsf{sync}(x)$ that we had due to $\mathsf{N}_\tau[x] = \mathsf{F}_\tau[x]$ for each $x \in \mathsf{F}_{2\tau-1}$. However, even $|\mathsf{N}_\tau[x]| = 1$ is possible in general. To handle fragments with $\mathsf{N}_\tau[x] \ne \mathsf{F}_\tau[x]$, we define

$$\mathsf{B}_\tau = \{y \in \mathsf{N}_\tau : \mathrm{per}(y[1\mathinner{.\,.}\tau-1]) \le \tfrac{1}{3}\tau \text{ or } \mathrm{per}(y[2\mathinner{.\,.}\tau]) \le \tfrac{1}{3}\tau\}$$

for $\tau \in [\![2,n]\!]$. Moreover, we set $\mathsf{B}_1 = \emptyset$ and $\mathsf{B}_\tau[x] = \mathsf{F}_\tau[x] \cap \mathsf{B}_\tau$ for every fragment $x \in \mathsf{F}$. Intuitively, B_τ forms a boundary which separates N_τ from its complement $\mathsf{F}_\tau \setminus \mathsf{N}_\tau$, as formalized in the fact below. However, it also contains some additional fragments included to make sure that our choice is consistent, i.e., that $y \in \mathsf{B}_\tau$ is equivalent to $y' \in \mathsf{B}_\tau$ if the two fragments match.

Fact 4.4.6. *If $\mathsf{N}_\tau[x] \ne \emptyset$ and $\mathsf{N}_\tau[x] \ne \mathsf{F}_\tau[x]$ for a fragment $x \in \mathsf{F}$, then $\mathsf{B}_\tau[x] \ne \emptyset$.*

Proof. We proceed by induction on $|x|$. The first non-trivial case is $|x| = \tau+1$, when $|\mathsf{F}_\tau[x]| = 2$. Let $\mathsf{N}_\tau[x] = \{y\}$, $\mathsf{F}_\tau[x] = \{y, y'\}$, and $z = x[2\mathinner{.\,.}\tau]$. Note that $z = y \cap y'$, so $\mathrm{per}(z) \le \mathrm{per}(y') \le \frac{1}{3}\tau$ yields $y \in \mathsf{B}_\tau[x]$.

For the inductive step, it suffices to note that if $\mathsf{N}_\tau[x] \ne \emptyset$ and $\mathsf{N}_\tau[x] \ne \mathsf{F}_\tau[x]$ for a fragment x of length $|x| > \tau+1$, then the length-$(|x|-1)$ prefix or suffix of x satisfies an analogous condition. □

The sets N_τ and B_τ can be characterized using the notion of *runs* (maximal repetitions) defined in Section 2.2. For this, we introduce a set $\mathcal{R}_\tau$ of *τ-runs*:

$$\mathcal{R}_\tau = \{\gamma \in \mathcal{R}(T) : |\gamma| \ge \tau - 1 \text{ and } \mathrm{per}(\gamma) \le \tfrac{1}{3}\tau\}.$$

Lemma 4.4.7. *Let T be a text of length n and let $\tau \in [\![1,n]\!]$. For each $y \in \mathsf{F}_\tau$:*

(a) *If $y \notin \mathsf{N}_\tau$, then y is contained in a unique τ-run $\gamma \in \mathcal{R}_\tau$; otherwise, there is no such τ-run γ.*

(b) *If $y \in \mathsf{B}_\tau$, then there exists a unique τ-run $\gamma \in \mathcal{R}_\tau$ such that $|y \cap \gamma| = \tau - 1$; otherwise, there is no such τ-run γ.*

Moreover, $|\mathcal{R}_\tau| \le \frac{3n}{\tau}$ and $|\mathsf{B}_\tau| \le \frac{6n}{\tau}$.

Proof. We assume $\tau \in [\![3,n]\!]$; otherwise, $\mathsf{N}_\tau = \mathsf{F}_\tau$ and $\mathcal{R}_\tau = \emptyset = \mathsf{B}_\tau$, so all the claims are trivial. First, we observe that $|\gamma \cap \gamma'| < \frac{2}{3}\tau - 1$ for any distinct $\gamma, \gamma' \in \mathcal{R}_\tau$ due to Fact 2.2.4. Since both τ-runs are of length at least $\tau-1$, they must start at least $\tau - 1 - |\gamma\cap\gamma'| > \frac{1}{3}\tau$ positions apart, i.e., $|\mathcal{R}_\tau| \le \frac{3n}{\tau}$.

(a) Recall that $y \notin \mathsf{N}_\tau$ means that y is highly periodic, so $\gamma = \mathsf{run}(y)$ satisfies $\mathrm{per}(\gamma) = \mathrm{per}(y) \le \frac{1}{3}|y| = \frac{1}{3}\tau$ and $|\gamma| \ge |y| > \tau - 1$. Thus, y is contained in $\mathsf{run}(y) \in \mathcal{R}_\tau$. On the other hand, if $y \subseteq \gamma$ for a τ-run γ, then $\mathrm{per}(y) \le \mathrm{per}(\gamma) \le \frac{1}{3}\tau$, so $y \notin \mathsf{N}_\tau$. Moreover, we must have $\gamma = \mathsf{run}(y)$ due to $|\gamma \cap \mathsf{run}(y)| \ge \tau \ge \frac{2}{3}\tau - 1$ and Fact 2.2.4.

(b) If $y \in \mathsf{B}_\tau$, then a fragment $z \in \mathsf{F}_{\tau-1}(y)$ (the prefix or the suffix of y) satisfies $\mathrm{per}(z) \le \frac{1}{3}\tau$ and $|z| = \tau - 1 \ge \frac{2}{3}\tau$, so z is periodic and $\gamma = \mathsf{run}(z) \in \mathcal{R}_\tau$. Moreover, $y \not\subseteq \gamma$ due to $y \in \mathsf{N}_\tau$. Thus, $y \cap \gamma = z$, so $|y \cap \gamma| = |z| = \tau - 1$. For the converse implication, suppose that $|y \cap \gamma| = \tau - 1$ for a τ-run $\gamma \in \mathcal{R}_\tau$. In this case, we must have $y[2\,.\,.\,\tau-1] \subseteq \gamma$. Moreover, if a τ-run γ' contains $y[2\,.\,.\,\tau-1]$, then $|\gamma \cap \gamma'| \ge \tau - 2 \ge \frac{2}{3}\tau - 1$, so $\gamma = \gamma'$ by Fact 2.2.4. Consequently, the run γ satisfying (b) is unique and, due to $|y \cap \gamma| = \tau - 1$ and $\mathrm{per}(y \cap \gamma) \le \frac{1}{3}\tau$, we also conclude that $y \in \mathsf{B}_\tau$.

Finally, we note that the characterization (b) yields $|\mathsf{B}_\tau| \le 2|\mathcal{R}_\tau| \le \frac{6n}{\tau}$. □

4.4.3 Construction for Arbitrary Texts

The combinatorial structure described in Section 4.4.2 lets us adopt the constructions of Lemmas 4.4.4 and 4.4.5 to arbitrary texts. The crucial trick is to assign the smallest identifiers to B_τ. In other words, we use the fragments of B_τ as synchronizing fragments whenever possible. A probabilistic construction explains why this is very helpful.

Lemma 4.4.8. *Let T be a text of length n. For each $\tau \in [\![1, \lceil \frac{n}{2} \rceil]\!]$, there exists a τ-identifier function* id *such that Construction 4.4.1 results in* $|\mathsf{Step}(\mathsf{sync})| = \mathcal{O}(\frac{n}{\tau})$.

Proof. Again, we take a random bijection $\pi : \mathcal{P}_\tau \to [\![1, |\mathcal{P}_\tau|]\!]$ and set $\mathsf{id}(y) = \pi(P)$ if $y \in P$. However, this time we draw π uniformly at random among bijections such that if $P \subseteq \mathsf{B}_\tau$ and $P' \cap \mathsf{B}_\tau = \emptyset$ for classes $P, P' \in \mathcal{P}_\tau$, then $\pi(P) < \pi(P')$. (Note that each class in $\mathcal{P}_\tau$ is either contained in B_τ or is disjoint with this set.)

We shall prove that $\mathbb{P}[\mathsf{sync}(x) = y] \le \frac{3}{\tau}$ if $y \in \mathsf{N}_\tau[x] \setminus \mathsf{B}_\tau[x]$. The constraint on π guarantees that $\mathsf{B}_\tau[x] = \emptyset$ if $\mathsf{sync}(x) \in \mathsf{N}_\tau[x] \setminus \mathsf{B}_\tau[x]$. Obviously, we must also have $\mathsf{N}_\tau[x] \ne \emptyset$ in this case, so Fact 4.4.6 yields $\mathsf{N}_\tau[x] = \mathsf{F}_\tau[x]$. Consequently, as in the proof of Lemma 4.4.4, by Fact 4.4.3, $\mathsf{N}_\tau[x]$ contains members of at least $\frac{\tau}{3}$ classes $P \in \mathcal{P}_\tau$. These classes are disjoint with B_τ, so they have the same probability of getting the smallest identifier.

As a result, if $x, x' \in \mathsf{F}_{2\tau-1}$ start at the consecutive positions i and $i+1$, then

$$\mathbb{P}[\mathsf{sync}(x) \ne \mathsf{sync}(x')] \le \mathbb{P}[\mathsf{sync}(x) = x[1\,.\,.\,\tau]] + \mathbb{P}[\mathsf{sync}(x') = x'[\tau\,.\,.\,2\tau-1]] \le \tfrac{6}{\tau}$$

unless $F_\tau(i) = x[1\,.\,.\,i] \in \mathsf{B}_\tau$ or $F_\tau(i+\tau) = x'[\tau\,.\,.\,2\tau-1] \in \mathsf{B}_\tau$. Consequently, due to Lemma 4.4.7:

$$\mathbb{E}[|\mathsf{Step}(\mathsf{sync})|] = 1 + \sum_{i=1}^{n-(2\tau-1)} \mathbb{P}[\mathsf{sync}(F_{2\tau-1}(i)) \ne \mathsf{sync}(F_{2\tau-1}(i+1))] \le$$
$$\le 1 + 2|\mathsf{B}_\tau| + \tfrac{6(n-(2\tau-1))}{\tau} \le \tfrac{18n}{\tau}.$$

In particular, $|\mathsf{Step}(\mathsf{sync})| \le \frac{18n}{\tau} = \mathcal{O}(\frac{n}{\tau})$ for some τ-identifier function id. □

Our adaptation of the deterministic construction is based on similar arguments and on the characterization of Lemma 4.4.7 to efficiently retrieve N_τ and B_τ.

Lemma 4.4.9. *Given a text T of length n and a parameter $\tau \in [\![1, \lceil \frac{n}{2} \rceil]\!]$, in $\mathcal{O}(n)$ time one can construct a τ-identifier function* id *and a τ-synchronizing function* sync *(defined with Construction 4.4.1 based on* id*) such that* $|\mathsf{Step}(\mathsf{sync})| = \mathcal{O}(\frac{n}{\tau})$.

Proof. We proceed as in the proof of Lemma 4.4.5, with some extra care for fragments $x \in \mathsf{F}_{2\tau-1}$ such that $\mathsf{F}_\tau[x] \neq \mathsf{N}_\tau[x]$.

The construction of the partition $\mathcal{P}_\tau$ still takes $\mathcal{O}(n)$ time. During the preprocessing phase, we additionally build the sets N_τ and B_τ. It is easy to come up with a linear-time implementation using Proposition 2.2.5 and Lemma 4.4.7: We apply Proposition 2.2.5 to build $\mathcal{R}(T)$ in $\mathcal{O}(n)$ time, and we filter out runs γ with $|\gamma| < \tau - 1$ or $\operatorname{per}(\gamma) > \frac{1}{3}\tau$ to obtain $\mathcal{R}_\tau$. For each τ-run $\gamma = T[\ell \mathinner{.\,.} r] \in \mathcal{R}_\tau$, as instructed by Lemma 4.4.7, we mark that:

- $F_\tau(\ell - 1)$ belongs to B_τ if $\ell > 1$,
- $F_\tau(r - \tau + 2)$ belongs to B_τ if $r < n$,
- fragments $F_\tau(\ell), \ldots, F_\tau(r - \tau + 1)$ do not belong to N_τ.

Lemma 4.4.7 also implies that no fragment is marked twice and that the unmarked fragments belong to $\mathsf{N}_\tau \setminus \mathsf{B}_\tau$. Hence, we retrieve N_τ and B_τ in $\mathcal{O}(n)$ time.

After the preprocessing, we gradually construct id and sync, handling one partition class $P \in \mathcal{P}_\tau$ at a time. This procedure has three phases now: we start with classes contained in B_τ, then we process the remaining classes contained in N_τ. At this point, the synchronizing function sync is already in its final form, but for completeness of the function id, in the third phase, we assign identifiers to highly periodic length-τ fragments.

Classes $P \subseteq \mathsf{B}_\tau$, which obtain the initial identifiers, are processed in an arbitrary order. Each $y \in \mathsf{B}_\tau$ is processed in $\mathcal{O}(\tau)$ time so the whole first phase takes $\mathcal{O}(|\mathsf{B}_\tau|\tau) = \mathcal{O}(n)$ time due to Lemma 4.4.7. Moreover, $\mathsf{Step}(\mathsf{sync})$ contains at most $|\mathsf{B}_\tau| \leq \frac{6n}{\tau}$ steps with value in B_τ.

In the second phase, we process the remaining classes $P \subseteq \mathsf{N}_\tau$ as in the proof of Lemma 4.4.5. We only need to make sure that it is always possible to choose such a class with a non-negative aggregate score. For this, let us analyze an active block z. After the first phase, we have $\mathsf{id}(y) \neq \perp$ for $y \in \mathsf{B}_\tau$, so $\mathsf{B}_\tau[z] = \emptyset$. Due to Fact 4.4.6, this yields $\mathsf{N}_\tau[z] = \emptyset$ or $\mathsf{N}_\tau[z] = \mathsf{F}_\tau[z]$. In other words, highly periodic active fragments form separate active blocks which we do not need to be bothered with because the total score in each active block is non-negative. Thus, we can indeed reuse the original algorithm. To analyze the running time, we reintroduce the set A^+ containing, for each iteration j, active fragments $y \in P_j$ which had score $+2$ prior to processing P_j. Since $A^+ \subseteq \mathsf{N}_\tau$, this set is still $\frac{1}{3}\tau$-sparse, and consequently the running time of the second phase is $\mathcal{O}(n + \tau|A^+|) = \mathcal{O}(n)$. Moreover, $\mathsf{Step}(\mathsf{sync})$ contains at most $3|A^+| \leq \frac{9n}{\tau}$ steps with value in $\mathsf{N}_\tau \setminus \mathsf{B}_\tau$.

In the third phase, we just assign the remaining identifiers to highly periodic fragments of length τ, which takes $\mathcal{O}(n)$ time. The synchronizing function sync is not modified anymore, but we need to account for the steps with value $\perp$, which correspond to active blocks after the second phase. Such blocks start at least τ positions apart (because they do not share active fragments), so the number of steps with value $\perp$ is at most $\frac{n}{\tau}$.

Summing up, the overall running time is $\mathcal{O}(n)$ and the total number of steps is $|\mathsf{Step}(\mathsf{sync})| \leq \frac{6n}{\tau} + \frac{9n}{\tau} + \frac{n}{\tau} \leq \frac{16n}{\tau}$. □

4.4.4 Faster Implementation for $\tau = \Theta(\log_\sigma n)$

Below, we provide a more efficient construction procedure for a specific value $\tau = \lfloor \frac{1}{6} \log_\sigma n \rfloor$.

Lemma 4.4.10. *Given the packed representation of a text T of length n over an alphabet of size σ, in $\mathcal{O}(n/\log_\sigma n)$ time one can construct* $\mathsf{Step}(\mathsf{sync})$ *for a τ-synchronizing function*

sync *defined with Construction 4.4.1 based on a τ-identifier function* id *for* $\tau = \lfloor \frac{1}{6}\log_\sigma n\rfloor$. *The identifier* $\mathsf{id}(y)$ *is reported along with each synchronizing fragment* y.

Proof. We simulate the algorithm described in the proof of Lemma 4.4.9, exploiting local consistency of the underlying approach. More specifically, we observe that the way this procedure handles a fragment $y = F_\tau(i)$ depends only on the classes of the nearby fragments $F_\tau(j)$ with $|j-i| < \tau$. In particular, these classes determine the score of y during the algorithm and, if y becomes a synchronizing fragment, the relative location of the step of $\mathsf{Step}(\mathsf{sync})$ with value y.

Motivated by this observation, we partition F_τ into $\lceil \frac{n}{\tau}\rceil$ *blocks* so that the kth block contains fragments $F_\tau(i)$ with $\lceil \frac{i}{\tau}\rceil = k$. We also define the *context* of the kth block as a string $T[1+(k-2)\tau]\cdots T[(k+2)\tau]$, assuming that $T[i] = \#$ if i is not a position of T, and we say that two blocks are *equivalent* if they share the same context.

Based on the initial observation, we note that if two blocks are equivalent, then the corresponding fragments $y, y' \in \mathsf{F}_\tau$ (with the same relative position within each block) are processed in the same way by the procedure of Lemma 4.4.9. This essentially means that it suffices to process just one *representative block* in each equivalence class.

Proposition 3.2.1 lets us retrieve each context in $\mathcal{O}(\lceil \frac{\tau\log\sigma}{W}\rceil) = \mathcal{O}(1)$ time. Consequently, it takes $\mathcal{O}(n/\log_\sigma n)$ time to partition the blocks into equivalence classes and to construct a family $\mathcal{B}$ of representative blocks. Furthermore, our choice of τ guarantees that $|\mathcal{B}| = \mathcal{O}(1+\sigma^{4\tau}) = \mathcal{O}(n^{2/3})$. Similarly, the class $P \in \mathcal{P}_\tau$ of a fragment $y \in \mathsf{F}_\tau$ is determined by the underlying substring, so $|\mathcal{P}_\tau| = \mathcal{O}(\sigma^\tau) = \mathcal{O}(n^{1/6})$ and the substring can also be retrieved in $\mathcal{O}(1)$ time. Consequently, the procedure in the proof of Lemma 4.4.9 has $\mathcal{O}(n^{1/6})$ iterations. If we spend $\Theta(\tau^{\mathcal{O}(1)})$ time for each representative block at each iteration, we still obtain the overall running time $o(\frac{n}{\log n})$. This generous margin allows for a relatively straightforward approach.

Our implementation maintains classes $P \in \mathcal{P}_\tau$ indexed by the underlying substrings. For each class, we store the value $\mathsf{id}(y)$ assigned to the occurrences $y \in P$, a list of occurrences $y \in P$ contained in the representative blocks, and information whether $P \subseteq \mathsf{N}_\tau$ and whether $P \subseteq \mathsf{B}_\tau$. To initialize these components (with the values $\mathsf{id}(y)$ set to $\perp$ at first), we scan all representative blocks, spending $\mathcal{O}(\tau^{\mathcal{O}(1)})$ time per block, which results in $\mathcal{O}(|\mathcal{B}|\tau^{\mathcal{O}(1)}) = o(\frac{n}{\log n})$ time in total.

In the first phase, we simply assign initial positive integers to classes $P \subseteq \mathsf{B}_\tau$. Second-phase iterations are more complicated because they involve computing scores. To determine the score of a particular class $P \in \mathcal{P}_\tau$, we iterate over all occurrences $y = F_\tau(i) \in P$ contained in representative blocks. We retrieve the class of each fragment $F_\tau(j)$ with $|j-i| < \tau$ in order to compute the score of y. We add this score, multiplied by the number of equivalent blocks, to the aggregate score of P. Having computed the score of each class, we take an arbitrary class P_j with a non-negative score (and no value assigned yet), and we assign the subsequent value j to this class. As announced above, the running time of a single iteration is $\mathcal{O}(|\mathcal{B}|\tau^{\mathcal{O}(1)})$ since we spend $\mathcal{O}(\tau^{\mathcal{O}(1)})$ time for each fragment $y \in \mathsf{F}_\tau$ contained in a representative block.

In the post-processing, we compute $\mathsf{Step}(\mathsf{sync})$ restricted to values from the representative blocks (with $\mathsf{id}(y)$ stored for each synchronizing fragment y). To achieve this goal, for every $y = F_\tau(i)$ contained in a representative block, we retrieve the classes of the nearby fragments $F_\tau(j)$ (with $|j-i| < \tau$) to check whether y is a synchronizing fragment and, if so, to determine its step in $\mathsf{Step}(\mathsf{sync})$. This takes $\mathcal{O}(\tau^{\mathcal{O}(1)})$ time per fragment y, which is $\mathcal{O}(|\mathcal{B}|\tau^{\mathcal{O}(1)}) = o(\frac{n}{\log n})$ in total.

Finally, we build $\mathsf{Step}(\mathsf{sync})$: For each block, we copy the representation from the corresponding representative block (shifting the indices accordingly). Next, we concatenate the step representations, inserting steps with value $\perp$ to fill the gaps.

The running time of this final phase is $\mathcal{O}(|\mathsf{Step}(\mathsf{sync})| + \frac{n}{\tau})$; this is $\mathcal{O}(\frac{n}{\tau})$ because we simulated the construction in the proof of Lemma 4.4.9. □

4.5 Data Structure

We are now ready to present the data structures for LCE QUERIES in texts over small alphabets. We assume that the packed representation of the text is available. Nevertheless, our construction algorithm for arbitrary τ uses $\Theta(n)$ time and $\Theta(n)$ working space.

Theorem 4.5.1. *Let T be a text of length n over an alphabet of size σ, packed into $\mathcal{O}(\lceil\frac{n\log\sigma}{W}\rceil)$ machine words stored in a read-only random-access memory. For every parameter $\tau \in [\![1, \lceil\frac{n}{2}\rceil]\!]$, there is a data structure of size $\mathcal{O}(\frac{n}{\tau})$ which answers* LCE QUERIES *in $\mathcal{O}(\lceil\frac{\tau\log\sigma}{W}\rceil)$ time. It can be constructed in $\mathcal{O}(n)$ time.*

Proof. We use Lemma 4.4.9 to construct a τ-identifier function id and a τ-synchronizing function sync with $|\mathsf{Step}(\mathsf{sync})| = \mathcal{O}(\frac{n}{\tau})$. We compute $\mathsf{Step}(\mathsf{sync})$, associate each synchronizing fragment y with its identifier $\mathsf{id}(y)$, and discard the remaining τ-identifiers. Finally, we plug $\mathsf{Step}(\mathsf{sync})$ to Lemma 4.3.4, which results in a component answering $\mathrm{LCCE}_{\mathsf{sync}}$ queries in $\mathcal{O}(\log_W \tau)$ time. This way, we obtain an $\mathcal{O}(n)$-time construction algorithm of a data structure taking $\mathcal{O}(\frac{n}{\tau})$ space.

The remaining challenge is to implement $\mathrm{LCE}(i, i')$ queries. First, we compute $\min(\mathrm{LCE}(i, i'), 2\tau-1)$ using Proposition 3.2.1, which takes $\mathcal{O}(\lceil\frac{\tau\log\sigma}{W}\rceil)$ time. If $\mathrm{LCE}(i, i') < 2\tau - 1$, this already gives us the sought value $\mathrm{LCE}(i, i')$. Otherwise, we determine $\ell := \mathrm{LCCE}_{\mathsf{sync}}(i, i') + (\tau - 1)$ using Lemma 4.3.4. Based on Lemma 4.4.9, we conclude that

$$\mathrm{LCE}(i, i') = \ell + \min(\mathrm{LCE}(i + \ell, i' + \ell), \tau - 1).$$

The latter LCE value is determined using Proposition 3.2.1 again. The overall query time is $\mathcal{O}(\lceil\frac{\tau\log\sigma}{W}\rceil + \log_W \tau) = \mathcal{O}(\lceil\frac{\tau\log\sigma}{W}\rceil + \lceil\frac{\tau}{W}\rceil) = \mathcal{O}(\lceil\frac{\tau\log\sigma}{W}\rceil)$. □

For $\tau = \left\lfloor\frac{1}{6}\log_\sigma n\right\rfloor$, we achieve optimal construction time based on Lemma 4.4.10.

Theorem 1.1.1. *For every text T of length n over an alphabet of size σ, there exists a data structure of $\mathcal{O}(n\log\sigma)$ bits (i.e., $\mathcal{O}(n/\log_\sigma n)$ machine words) which answers* LCE QUERIES *in $\mathcal{O}(1)$ time. It can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time given the packed representation of the text T.*

Proof. We keep the packed representation of T so that it becomes a part of the data structure. We also plug it to Lemma 4.4.10, which results in $\mathsf{Step}(\mathsf{sync})$ including identifiers associated with synchronizing positions. Finally, we use this representation of sync to construct the component (of Lemma 4.3.4) for $\mathrm{LCCE}_{\mathsf{sync}}$ queries.

The procedure of Lemma 4.4.10 runs in $\mathcal{O}(n/\log_\sigma n)$ time and results in $|\mathsf{Step}(\mathsf{sync})| = \mathcal{O}(n/\log_\sigma n)$. Consequently, the running time in Lemma 4.3.4 is also $\mathcal{O}(n/\log_\sigma n)$.

Queries are processed in $\mathcal{O}(\lceil\frac{\log_\sigma n\cdot\log\sigma}{W}\rceil) = \mathcal{O}(\lceil\frac{\log n}{W}\rceil) = \mathcal{O}(1)$ time as in the proof of Theorem 4.5.1. □

Chapter 5

Queries Concerning Periodic Fragments

In this chapter, we develop data structures for two types of queries concerning periodic fragments of the input text: Periodic Extension Queries and Internal Periodic Pattern Matching Queries, both defined below. The latter is a special case of Internal Pattern Matching Queries discussed in Chapter 6.

Periodic Extension Queries
Given a fragment x of the text T, compute the run $\mathsf{run}(x)$ extending x. (Recall that $\mathsf{run}(x) = \perp$ if x is not periodic.)

Internal Periodic Pattern Matching (IPPM) Queries
Given a periodic fragment x and a fragment y of the text T with $|y| < 2|x|$, report the starting positions of fragments matching x and contained in y (represented as an arithmetic progression).

Both our data structures handle queries in $\mathcal{O}(1)$ time and admit $\mathcal{O}(n)$-time construction algorithms, which makes them optimal for integer alphabets. The solutions are based on similar tools: the structure of *runs* (maximal repetitions) in the text (introduced in Section 2.2.1) and efficient representation of piecewise-constant functions (see Section 3.4.1).

5.1 Periodic Extension Queries

A combinatorial fact allowing for an efficient implementation of Periodic Extension Queries is, briefly speaking, that at most two runs induce periodic fragments of similar lengths starting at the same position. Its precise formulation, along with a proof based on the *Three Squares Lemma*, is stated in Section 5.1.1. Motivated by this property, in Section 5.1.2 we organize the fragments of T into $\mathcal{O}(\log n)$ layers. Although a single run γ may induce periodic fragments in several layers, the number of such layers is $\mathcal{O}(\exp(\gamma))$, so due Proposition 2.2.5 we can afford to spend $\mathcal{O}(1)$ space on γ for each of these layers. Each layer is handled by a separate component, which stores the *step representations* of a certain function, equipped with a tool of Corollary 3.4.4 for $\mathcal{O}(1)$-time evaluation.

5.1.1 Runs Extending Fragments of Similar Lengths

Fragments x satisfying $\mathsf{run}(x) = \gamma$ admit an elegant characterization:

Observation 5.1.1. *Consider a text T and a run $\gamma \in \mathcal{R}(T)$. A fragment x of T satisfies $\mathsf{run}(x) = \gamma$ if and only if x is contained in γ and $|x| \ge 2\,\mathrm{per}(\gamma)$.*

Consequently, for a fixed starting position ℓ and a run $\gamma \in \mathcal{R}(T)$, the lengths of the fragments $T[\ell\mathinner{.\,.} r]$ with $\mathsf{run}(T[\ell\mathinner{.\,.} r]) = \gamma$ form an interval. Moreover, if the interval is non-empty, then $T[\ell\mathinner{.\,.}\ell+2\,\mathrm{per}(\gamma)-1]$ is a primitively rooted square. The number of such squares (and thus the number of runs extending a periodic fragment starting at position ℓ) is bounded by $\mathcal{O}(\log n)$ due to the following classic result:

Lemma 5.1.2 (Three Squares Lemma [49, 43])**.** *Let v_1, v_2, v_3 be strings such that v_1^2 is a proper prefix of v_2^2, v_2^2 is a proper prefix of v_3^2, and v_1 is primitive. Then $|v_1| + |v_2| \le |v_3|$.*

Our data structure is based on the following consequence of Lemma 5.1.2:

Corollary 5.1.3. *Let x_1, x_2, x_3 be periodic fragments of the text T, all starting at the same position ℓ. If $\lfloor \log |x_1| \rfloor = \lfloor \log |x_2| \rfloor = \lfloor \log |x_3| \rfloor$, then $\mathsf{run}(u_1)$, $\mathsf{run}(u_2)$, $\mathsf{run}(u_3)$ cannot be all distinct.*

Proof. Let k be the common value of $\lfloor \log |x_i| \rfloor$. For a proof by contradiction, suppose that the runs $\gamma_i = \mathsf{run}(u_i)$ are pairwise distinct, and let $p_i = \mathrm{per}(\gamma_i)$ be their periods. Without loss generality, assume that $p_1 \le p_2 \le p_3$.

Note that $p_i \le \frac{1}{2}|x_i| < 2^k$ and $|\gamma_i \cap \gamma_j| \ge 2^k$ for every $i, j \in \{0, 1, 2\}$. Thus, Fact 2.2.4 implies $p_1 < p_2 < p_3$. Consequently, strings $v_i \cong T[\ell\mathinner{.\,.}\ell + p_i - 1]$ satisfy the assumptions of Lemma 5.1.2, which yields $p_1 + p_2 \le p_3 \le 2^k$. However, Fact 2.2.4 further implies $p_1 + p_2 > p_1 + p_2 - \gcd(p_1, p_2) > |\gamma_1 \cap \gamma_2| \ge 2^k$, a contradiction. $\square$

5.1.2 Data Structure

With the combinatorial tools at hand, we are ready to describe our data structure.

Theorem 5.1.4. *Using a data structure of size $\mathcal{O}(n)$ which can be constructed in $\mathcal{O}(n)$ time, one can answer* Periodic Extension Queries *in $\mathcal{O}(1)$ time.*

Proof. For each $k \in [\![0, \lfloor \log n \rfloor]\!]$, let us define a function $R_k : [\![1, n]\!] \to 2^{\mathcal{R}(T)}$ which assigns to each position i the set of runs γ such that $\gamma = \mathsf{run}(x)$ for a fragment x which starts at position i and satisfies $\lfloor \log |x| \rfloor = k$. We shall store the step representation $\mathsf{Step}(R_k)$ equipped with a component of Corollary 3.4.4 for $\mathcal{O}(1)$-time evaluation.

Answering a query for $x = T[\ell\mathinner{.\,.} r]$ is simple since $\mathsf{run}(x) \in R_{\lfloor \log |x| \rfloor}(\ell)$ or $\mathsf{run}(x) = \perp$. Thus, we compute $k = \lfloor \log |x| \rfloor$ (applying Proposition 3.1.1) and make an evaluation query to retrieve $R_k(\ell)$. Corollary 5.1.3 guarantees that $|R_k(\ell)| \le 2$, so we use Observation 5.1.1 for each $\gamma \in R_k(\ell)$ to verify whether $\gamma = \mathsf{run}(x)$. If none of these checks succeeds, we conclude that $\mathsf{run}(x) = \perp$.

Designing an $\mathcal{O}(n)$-time construction algorithm is slightly more demanding; the key challenge is to build the step representations $\mathsf{Step}(R_k)$. For this, let us analyze when a run $\gamma = T[\ell\mathinner{.\,.} r] \in \mathcal{R}(T)$ belongs to $R_k(i)$. The characterization of Observation 5.1.1 provides an answer: $\gamma \in R_k(i)$ if and only if $\mathrm{per}(\gamma) < 2^k$ and $i \in [\![\ell, r - \max(2\,\mathrm{per}(\gamma), 2^k) + 1]\!]$. Moreover, we note the set of runs satisfying this condition for some position i is $\mathcal{R}_k(T) = \{\gamma \in \mathcal{R}(T) : \mathrm{per}(\gamma) < 2^k \le |\gamma|\}$. For each run $\gamma \in \mathcal{R}_k(T)$, we shall prepare two *events*:

$\mathsf{add}_k(\ell,\gamma)$ and $\mathsf{remove}_k(r-\max(2\operatorname{per}(\gamma),2^k)+2,\gamma)$. Now, in order to transform $R_k(i-1)$ into $R_k(i)$, it suffices to process the events $\mathsf{remove}_k(i,\gamma)$ and $\mathsf{add}_k(i,\gamma)$.

This approach is implemented as follows. First, we apply Proposition 2.2.5 to build the set $\mathcal{R}(T)$ of all runs and we process each run $\gamma = T[\ell\mathinner{.\,.} r] \in \mathcal{R}(T)$ to construct the appropriate events. For this, we compute an interval $[\![\lfloor\log(\operatorname{per}(\gamma))\rfloor+1, \lfloor\log|\gamma|\rfloor]\!]$ consisting of indices k such that $\gamma \in \mathcal{R}_k(T)$. For each such k, we create two events: $\mathsf{add}_k(\ell,\gamma)$ and $\mathsf{remove}_k(r-\max(2\operatorname{per}(\gamma),2^k)+2,\gamma)$. Next, we sort the events by the position and group them by the layer k. To construct $\mathsf{Step}(R_k)$, we scan the sorted list of events for layer k, maintaining the set $R_k(i)$, initialized as $\emptyset$. If there are no events at position i, we are guaranteed that $R_k(i) = R_k(i-1)$, so we do not need to do anything. Otherwise, we process the events to transform $R_k(i-1)$ into $R_k(i)$ and start a new step in $\mathsf{Step}(R_k)$. In the final phase, we equip the step representations $\mathsf{Step}(R_k)$ with the components of Corollary 3.4.4.

Let us bound the running time of the procedure above. Due to Proposition 2.2.5, constructing $\mathcal{R}(T)$ takes $\mathcal{O}(n)$ time. The number of events created for a single run γ is

$$|\{k : \gamma \in \mathcal{R}_k(T)\}| = \lfloor\log|\gamma|\rfloor - \lfloor\log\operatorname{per}(\gamma)\rfloor \le 1 + \log\tfrac{|\gamma|}{\operatorname{per}(\gamma)} = 1 + \log\exp(\gamma) \le \exp(\gamma),$$

where the last inequality follows from $\exp(\gamma) \ge 2$. The total number of events is $\mathcal{O}(n)$ because Proposition 2.2.5 states that $\sum_{\gamma\in\mathcal{R}(T)}\exp(\gamma) = \mathcal{O}(n)$. Sorting the events also takes $\mathcal{O}(n)$ time since the keys are positive integers bounded by n. As we generate $\mathsf{Step}(R_k)$, the maintained set is of constant size due to Corollary 5.1.3, so each event is processed in $\mathcal{O}(1)$ time. Finally, applying Corollary 3.4.4 takes $\mathcal{O}(|\mathsf{Step}(R_k)| + \frac{n}{W})$ space and construction time per layer, which is $\mathcal{O}(n)$ in total. Thus, our construction procedure takes $\mathcal{O}(n)$ time and the resulting data structure is of size $\mathcal{O}(n)$. □

5.2 Internal Periodic Pattern Matching Queries

Our approach to IPPM QUERIES mainly relies on the structure of runs in the text (see Section 2.2.1) and on the following trivial observation in particular.

Observation 5.2.1. *If x and x' are matching fragments of the text T, then the runs $\gamma = \mathsf{run}(x)$ and $\gamma' = \mathsf{run}(x')$, extending x and x' respectively, have equal periods.*

We use PERIODIC EXTENSION QUERIES of Section 5.1 to compute $\gamma = \mathsf{run}(x)$ and derive the period $\operatorname{per}(\gamma)$. Next, we look for runs γ' which may extend the fragments x' matching x and contained within y. Due to Observation 5.2.1 and the assumption $|y| < 2|x|$, it suffices to find all runs of period $\operatorname{per}(\gamma)$ which contain the middle position of y (formally defined as the position of T corresponding to $y[\lceil\frac{1}{2}|y|\rceil]$). In Section 5.2.1, we show that there are at most two such runs and develop a component for RUN FINDING QUERIES, specified below, allowing us to identify them efficiently. Its implementation relies on the tools of Section 3.4.1 (evaluation of functions with small step representations).

RUN FINDING QUERIES
Given a position i of the text T and an integer p, find all runs $\gamma \in \mathcal{R}(T)$ of period p containing position i.

Next, we look for the occurrences of x contained in each of the candidate runs γ'. We use techniques based on *Lyndon roots* and *compatibility* of runs, recalled in Section 5.2.2, originating from a paper by Crochemore et al. [45]. We may ignore runs γ' incompatible with γ, while for the compatible runs γ', it is easy to find all occurrences of x in $y \cap \gamma'$ We obtain at most two arithmetic progressions representing the occurrences of x in y, but the following folklore result guarantees that they can be merged into a single progression.

Fact 5.2.2 (Breslauer and Galil [27], Plandowski and Rytter [127])**.** *Let x, y be strings satisfying $|y| \le 2|x|$. The set of starting positions of the occurrences of x in y forms a single arithmetic progression.*

5.2.1 Run Finding Queries

Before we proceed to an implementation of Run Finding Queries, let us prove that the query output consists of at most two runs.

Fact 5.2.3. *Each position i of the text T lies within at most two runs of the same period p.*

Proof. Suppose that there are three distinct runs γ_1, γ_2, and γ_3 with period p, all containing position i. Observe that each of these runs contains positions $i-p$ or $i+p$, and thus $T[i\,..\,i+p]$ or $T[i-p\,..\,i]$ lies in the intersection of some two of them. However, by Fact 2.2.4, such an intersection may contain at most $p-1$ positions, a contradiction. □

Lemma 5.2.4. *For every text T, there exist a data structure that answers* Run Finding Queries *in $\mathcal{O}(1)$ time, takes $\mathcal{O}(n)$ space, and can be constructed in $\mathcal{O}(n)$ time.*

Proof. For each period p, let $\mathcal{R}^p(T) = \{\gamma \in \mathcal{R}(T) : \mathrm{per}(\gamma) = p\}$. Let us define a function $R^p : [\![1, n]\!] \to 2^{\mathcal{R}^p(T)}$ such that $R^p(i)$ consists of runs containing position i. Observe that $R^p(i) \ne R^p(i+1)$ only if a run $\gamma \in \mathcal{R}^p(T)$ ends at position i or starts at position $i+1$. Hence, the size of the step representation of R^p is bounded by $1 + 2|\mathcal{R}^p(T)|$, which is $\mathcal{O}(n)$ in total. Moreover, it is easy to compute the step representations $\mathsf{Step}(\mathcal{R}^p)$ in $\mathcal{O}(n)$ time.

Nevertheless, the total size of *evaluators* $\mathcal{E}(R^p)$ of Corollary 3.4.4 would be by far too large: $\Theta(\frac{n^2}{W})$. As a workaround, we observe that it suffices to evaluate R^p at positions i divisible by p. Indeed, every run $\gamma \in R^p(i)$ satisfies $|\gamma| \ge 2p$, so $R^p(i) \subseteq R^p(p\lfloor\frac{i}{p}\rfloor) \cup R^p(p\lceil\frac{i}{p}\rceil)$. Due to Fact 5.2.3, the right-hand side consists of at most 4 runs, and we can afford to check which of them contain position i.

Thus, we define $\tilde{R}^p(j) = R^p(p\cdot j)$ and note that the step representation of $\tilde{R}^p$ is easy to construct from the step representation of R^p (and the size cannot increase). The total size of evaluators $\mathcal{E}(\tilde{R}^p)$ can be expressed as follows using *harmonic numbers* $H_n = \sum_{i=1}^{n}\frac{1}{n} \le 1 + \ln n$:

$$\sum_{p=1}^{n} \mathcal{O}\left(1 + \tfrac{n}{pW} + |\mathcal{R}^p(T)|\right) = \mathcal{O}\left(n + \tfrac{nH_n}{W} + n\right) = \mathcal{O}(n).$$

Corollary 3.4.4 also yields $\mathcal{O}(n)$ total construction time of these evaluators. □

5.2.2 Compatibility of Strings

Recall that a string which is both primitive and lexicographically minimal in the class of its cyclic rotations is called a Lyndon word; see Section 2.2. Let u be a string with the shortest period $\mathrm{per}(u) = p$. The *Lyndon root* λ of u is the Lyndon word that is a cyclic rotation of the prefix $u[1]\cdots u[p]$, i.e., the minimal cyclic rotation of that prefix. We say that two strings are *compatible* if they have the same Lyndon root.

A string u with Lyndon root λ can be uniquely represented as $\lambda'\lambda^k\lambda''$, where λ' is a proper suffix of λ, λ'' is a proper prefix of λ, and $k \in \mathbb{Z}_{\geq 0}$ is a non-negative integer. The *Lyndon signature* of u is defined as $(|\lambda'|, k, |\lambda''|)$. Note that the Lyndon signature uniquely determines u within its compatibility class. This representation is very convenient for pattern matching if the text is compatible with the pattern.

Lemma 5.2.5. *Let x and y be compatible strings. The set of positions where x occurs in y is an arithmetic progression that can be computed in $\mathcal{O}(1)$ time given the Lyndon signatures of x and y.*

Proof. Let λ be the common Lyndon root of x and y and let their Lyndon signatures be (p, k, s) and (p', k', s') respectively. Lemma 2.2.2 (synchronization property) implies that λ occurs in y only at positions i such that $i \equiv p' + 1 \pmod{|\lambda|}$. Consequently, x occurs in y only at positions i such that $i \equiv p' - p + 1 \pmod{|\lambda|}$. Clearly, x occurs in y at all such positions i within the interval $[\![1, |y| - |x| + 1]\!]$. Therefore, it is a matter of simple calculations to compute the arithmetic progression of these positions. □

Crochemore et al. [45] already showed how to efficiently compute Lyndon signatures of the runs of a given text.

Fact 5.2.6 (Crochemore et al. [45])**.** *Given a text T of length n, in $\mathcal{O}(n)$ time one can compute Lyndon signatures of all runs in T and partition the runs into compatibility classes.*

Next, we note that the Lyndon signature of a run γ determines the Lyndon signatures of periodic fragments x induced by γ, i.e., satisfying $\gamma = \mathsf{run}(x)$.

Observation 5.2.7. *Let w be a periodic string of period p. If u is a fragment of w and $|u| \geq 2p$, then u is compatible with w. Moreover, given the Lyndon signature of w, one can compute the Lyndon signature of u in constant time.*

5.2.3 Implementation of IPPM Queries

In this section, we describe our data structure for IPPM Queries. It consists of:

1. the set of runs $\mathcal{R}(T)$, each run accompanied with its period, Lyndon signature and an identifier of the compatibility class;
2. the data structure of Theorem 5.1.4 for Periodic Extension Queries; and
3. the data structure of Lemma 5.2.4 for Run Finding Queries.

Note that Proposition 2.2.5 and Fact 5.2.6 guarantee that the first component takes $\mathcal{O}(n)$ space and can be constructed in $\mathcal{O}(n)$ time. For the latter two components, Theorem 5.1.4 and Lemma 5.2.4 yield the same bounds.

Next, we show that this data structure can handle IPPM Queries in $\mathcal{O}(1)$ time. The query algorithm consists of the following steps, introduced at the beginning of Section 5.2:

(A) Compute the run $\gamma = \mathsf{run}(x)$ (raise an error if $\mathsf{run}(x) = \perp$).
(B) Find all runs γ' with $\mathrm{per}(\gamma') = \mathrm{per}(\gamma)$ containing the middle position of y, defined as $T[\lfloor \frac{\ell+r}{2} \rfloor]$ for $y = T[\ell \mathinner{.\,.} r]$.
(C) Filter out runs γ' incompatible with γ.
(D) For each of the compatible runs γ', compute an arithmetic progression representing the occurrences of x in $y \cap \gamma'$.
(E) Combine the resulting occurrences of x in y into a single arithmetic progression.

Before implementing these steps, let us justify the correctness of the algorithm. Clearly, a fragment matching x (and contained in y) starts at each of the reported positions. It remains to prove that no occurrence x' is missed. By Observation 5.2.1, $\gamma' = \mathsf{run}(x')$ is a run of period $\mathrm{per}(\gamma)$. Note that x' cannot end earlier than at position $\ell + |x| - 1$ and it cannot start later than at position $r - |x| + 1$. Due to $|y| < 2|x|$, we have

$$r - |x| + 1 \le r - \left\lceil \tfrac{|y|+1}{2} \right\rceil + 1 = \left\lfloor \tfrac{\ell+r}{2} \right\rfloor \le \left\lceil \tfrac{\ell+r}{2} \right\rceil = \ell + \left\lceil \tfrac{|y|+1}{2} \right\rceil - 1 \le \ell + |x| - 1,$$

so x' must contain the middle position of y. Therefore, $\gamma' = \mathsf{run}(x')$ is among the runs found in step (B). By Observation 5.2.7, γ is compatible with x and γ' is compatible with x'. Since x and x' match, γ and γ' must be compatible. Hence, γ' is considered in step (D) and the starting position of x' is reported in step (E).

We conclude with the implementation of the subsequent steps of the query algorithm. In step (A), we simply evaluate $\gamma = \mathsf{run}(x)$ using a Periodic Extension Query of Theorem 5.1.4. In step (B), we retrieve $p = \mathrm{per}(\gamma)$ and use a Run Finding Query to identify all runs γ' with $\mathrm{per}(\gamma') = p$ containing the middle position of y. Fact 5.2.3 guarantees that we obtain at most two runs γ'. We use the partition of runs into compatibility classes to implement step (C). For the remaining (compatible) runs γ', we apply Lemma 5.2.5 to find the occurrences of x in $y \cap \gamma'$. There is nothing to do if $|x| > |y \cap \gamma'|$. Otherwise, $|y \cap \gamma'| \ge |x| \ge 2\,\mathrm{per}(\gamma) = 2\,\mathrm{per}(\gamma')$, so Observation 5.2.7 lets us retrieve the Lyndon signatures of both x and $y \cap \gamma'$. We are left with at most two arithmetic progressions (one for each compatible run γ'). As discussed above, their union represents all occurrences of x in y. By Fact 5.2.2, this set must form a single arithmetic progression. If the progressions are stored by (at most) three elements—the last one and the first two—it is easy to compute the union in constant time. This concludes the proof of the following result:

Theorem 5.2.8. *There exists a data structure of size $\mathcal{O}(n)$ which can be constructed in $\mathcal{O}(n)$ time and answers* IPPM Queries *in $\mathcal{O}(1)$ time. The query algorithm reports an error whenever the query pattern x is not periodic.*

Chapter 6

Internal Pattern Matching Queries

This chapter is devoted to our solution for INTERNAL PATTERN MATCHING QUERIES, formally defined as follows:

> INTERNAL PATTERN MATCHING (IPM) QUERIES
> Given fragments x and y of the text T satisfying $|y| < 2|x|$, report the starting positions of fragments matching x and contained in y (represented as an arithmetic progression).

The description of our data structure is organized as follows. We outline the main ideas in Section 6.1; in particular, this is where we introduce our central combinatorial tool—the notion of a *representative assignment* assigning *representative fragments* $\mathsf{repr}(x)$ to fragments x of the text T. Section 6.2 is devoted to constructing a representative assignment. We start with a simple construction resulting in $\mathcal{O}(n \log n)$ representative fragments. Next, using the synchronizing functions of Chapter 4, we reduce the number of representative fragments to $\mathcal{O}(n)$ and obtain an $\mathcal{O}(n \log n)$-time construction algorithm of the underlying assignment. We then improve the construction time to $\mathcal{O}(n)$ by restricting the representative assignment to fragments of length $|x| = \Omega(W)$ only. (Recall that $W = \Omega(\log n)$ is the machine word size.) At the same time, the number of representative fragments is reduced to $\mathcal{O}(\frac{n \log W}{W})$. Consequently, in the final Section 6.3, we develop two components: an auxiliary one for IPM QUERIES queries with short patterns (using fusion trees; see Section 3.4) and the main one handling long patterns based on the representative assignment.

6.1 Overview

Throughout the chapter, we fix a text T of length n. In order to support IPM QUERIES in T, we use a classic idea of pattern matching by sampling in a novel way. To every fragment x of the text T, we assign a *representative* (a sample) $\mathsf{repr}(x)$, which is a fragment of T contained in x. We make sure that this assignment is *consistent*, i.e., if x and x' match, then the location of $\mathsf{repr}(x')$ relative to x' must be the same as the location of $\mathsf{repr}(x)$ relative to x; see Figure 6.1. This property is crucial for answering IPM QUERIES: if a fragment x' contained in y matches x, then we are guaranteed that $\mathsf{repr}(x')$ and $\mathsf{repr}(x)$ also match. Applying this fact, our query algorithm locates the *representative fragments* matching $\mathsf{repr}(x)$ and contained in y, and it checks which of them can be extended to occurrences of x; see Figure 6.2.

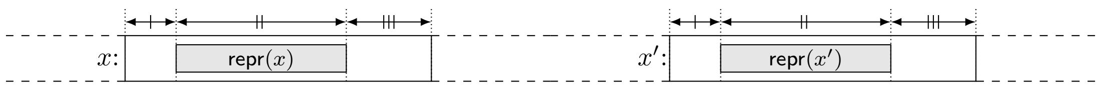


Figure 6.1: If x and x' match, then $\mathsf{repr}(x)$ and $\mathsf{repr}(x')$ must be chosen consistently: lengths marked in the same way must be equal. In particular, $\mathsf{repr}(x)$ and $\mathsf{repr}(x')$ also match.

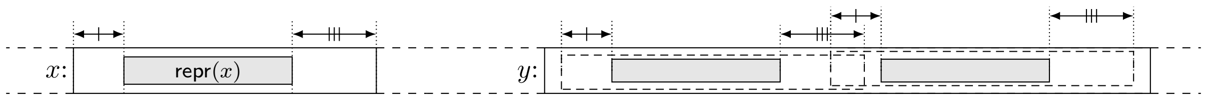


Figure 6.2: The main idea of the query algorithm for fragments x and y. We find the representative fragments matching $\mathsf{repr}(x)$ and contained in y (depicted as gray rectangles). If there is an occurrence x' of x contained in y, then $\mathsf{repr}(x')$ is one of these representative fragments. Thus, x' must be one of the fragments marked with dashed rectangles.

In order to achieve constant query time with this approach, we need to guarantee that $\mathsf{repr}(x)$ has $\mathcal{O}(1)$ representative occurrences in y. For this, we need to impose some conditions on the representative fragments. First, we require that $|\mathsf{repr}(x)| = \Theta(|x|)$. Since $|y| < 2|x|$, this way we make sure that the number of non-overlapping occurrences of $\mathsf{repr}(x)$ in y is constant. However, there might still be many occurrences with large overlaps. To exclude this possibility, we prohibit representative fragments with very small periods and rely on the following fact for the remaining substrings of T.

Fact 6.1.1 (Sparsity of occurrences)**.** *The set of positions where a substring u occurs in a text T is* $\mathrm{per}(u)$*-sparse, i.e., every two distinct positions i, i' satisfy* $|i - i'| > \mathrm{per}(u)$.

Proof. If u occurs in T at positions i and $i + d$ such that $i < i + d \le i + |u|$, then $u[j + d] = u[j]$ for every $j \in [\![1, |u| - d]\!]$, i.e., d is a period of u. □

Unfortunately, the restriction $\mathrm{per}(\mathsf{repr}(x)) = \Theta(|x|)$ makes assigning a representative $\mathsf{repr}(x)$ unfeasible for some fragments x. Nevertheless, this may happen only if $\mathrm{per}(x) = o(|x|)$, and thus only answering IPM Queries with periodic patterns becomes problematic. This special case is covered by Internal Periodic Pattern Matching Queries considered in Section 5.2, so we build the specialized component of Theorem 5.2.8 to deal with periodic patterns.

Consequently, we define repr only for a subset G of the family F of fragments of the text T. The co-domain R of repr is another subset of F and its members are called the *representative fragments*. The number $|\mathsf{R}|$ of the representative fragments is the main parameter governing the size of our data structure.

Definition 6.1.2. *A* representative assignment for G *is a function* $\mathsf{repr} : \mathsf{G} \to \mathsf{R}$ *mapping* $\mathsf{G} \subseteq \mathsf{F}$ *to* $\mathsf{R} \subseteq \mathsf{F}$ *such that for each* $x \in \mathsf{G}$*:*

(a) the fragment $\mathsf{repr}(x)$ *is contained in x and its period satisfies* $\mathrm{per}(\mathsf{repr}(x)) = \Theta(|x|)$*,*

(b) if $x' \in \mathsf{F}$ *matches x, then* $x' \in \mathsf{G}$ *and the location of* $\mathsf{repr}(x')$ *relative to x' is the same as the location of* $\mathsf{repr}(x)$ *relative to x, i.e.,* $\mathsf{repr}(x) = x[i\mathinner{.\,.} j]$ *and* $\mathsf{repr}(x') = x'[i\mathinner{.\,.} j]$ *for some positions* $1 \le i \le j \le |x|$.

To implement IPM Queries with patterns $x \in \mathsf{G}$, the underlying representative assignment $\mathsf{repr} : \mathsf{G} \to \mathsf{R}$ also needs to satisfy certain algorithmic properties listed below.

Definition 6.1.3. *We say that a representative assignment* $\mathsf{repr} : \mathsf{G} \to \mathsf{R}$ *admits* an efficient implementation *if it can be stored in* $\mathcal{O}(n)$ *space so that:*

(1) the representative $\mathsf{repr}(x)$ *of a given fragment* $x \in \mathsf{G}$ *can be computed in* $\mathcal{O}(1)$ *time;*

(2) the representative fragments can be listed in $\mathcal{O}(|\mathsf{R}|)$ *time.*

6.2 Representative Assignment Construction

Our goal in this section is to build an efficient representative assignment, i.e., a function which satisfies both the combinatorial requirements of Definition 6.1.2 and the algorithmic properties specified in Definition 6.1.3. We start in Section 6.2.1 with a simple representative assignments for the family $\mathsf{N} = \{y \in \mathsf{F} : \operatorname{per}(y) > \frac{1}{3}|y|\}$ of non-highly-periodic fragments of T. The representative fragments of this assignment are *basic fragments*, whose length is a power of 2. The construction makes use of the characterization of N provided in Section 4.4.2.

To achieve $\mathcal{O}(n)$ representative fragments, in Section 6.2.2 we introduce a representative assignment for N based on $\mathcal{O}(\log n)$ synchronizing functions of Section 4.2 and their construction specified in Section 4.4. Lemma 4.4.9 yields an $\mathcal{O}(n)$-time construction algorithm for a single synchronizing function, so it takes $\mathcal{O}(n \log n)$ time to build this representative assignment. In the final Section 6.2.3, we provide an $\mathcal{O}(n)$-time construction of a similar representative assignment restricted to fragments of length $|x| \geq W$.

6.2.1 $\mathcal{O}(n \log n)$ Representative Fragments in $\mathcal{O}(n)$ Time

Let us first analyze simple ways to obtain a representative assignment for N. Arguably, the most trivial construction sets $\mathsf{repr}(x) = x$ for each $x \in \mathsf{N}$. The correctness is obvious, but it is also clear that the number of representative fragments might be $\Theta(n^2)$.

An easy way to dramatically reduce the number of representative fragments is to partition N into several *layers* so that fragments in a single layer have representatives of the same length. The number of layers must be $\Omega(\log n)$ because Definition 6.1.2 requires $|\mathsf{repr}(x)| = \Theta(|x|)$. For example, we can impose a condition that $|\mathsf{repr}(x)| = 2^{\lfloor \log |x| \rfloor}$ for each $x \in \mathsf{N}$. As a result, only the $\mathcal{O}(n \log n)$ *basic fragments* (fragments whose length is a power of 2) might be representative fragments. A naive choice of $\mathsf{repr}(x)$ would then be the longest prefix of x which is a basic fragment, but such a prefix might have a very small period even if x does not have one. Thus, we need to be more careful; for example, as $\mathsf{repr}(x)$ we may set the leftmost fragment of length $2^{\lfloor \log |x| \rfloor}$ which is contained in x and belongs to N. Below, we formally define the resulting representative assignment and provide an efficient implementation, partially based on the characterization of N in Section 4.4.2.

Formal Definition

Recall the subsets F_m, $\mathsf{F}[x]$, and $\mathsf{F}_m[x]$ of F (defined in Section 4.2) that consist of, respectively, fragments of a given length $m \in [\![1, n]\!]$, fragments contained in a given fragment $x \in \mathsf{F}$, and fragments satisfying both conditions. We also use analogous notions for subsets of the set N of non-highly-periodic fragments of T.

Construction 6.2.1. *For a fragment* $x \in \mathsf{N}$*, we define* $\mathsf{repr}(x)$ *as the leftmost fragment* $y \in \mathsf{N}_{2^k}[x]$ *with* $k = \lfloor \log |x| \rfloor$*.*

To prove that the value $\mathsf{repr}(x)$ is well defined for every $x \in \mathsf{N}$, we apply the characterization of Lemma 4.4.7 to derive the following result:

Fact 6.2.2. *If a fragment $x \in \mathsf{F}$ satisfies $|x| \geq \tau$ and $\operatorname{per}(x) > \frac{1}{3}\tau$ for an integer $\tau \in [\![1, n]\!]$, then $\mathsf{N}_\tau[x] \neq \emptyset$.*

Proof. If the prefix $x[1\mathinner{.\,.}\tau]$ belongs to N_τ, the claim holds trivially. Otherwise, by Lemma 4.4.7(a), there exists a τ-run $\gamma \in \mathcal{R}_\tau$ containing that prefix. Note that $\operatorname{per}(\gamma) \leq \frac{1}{3}\tau < \operatorname{per}(x)$, so x is not contained in γ. Thus, there exists a fragment $y \in \mathsf{F}_\tau[x]$ such that $|y \cap \gamma| = \tau - 1$. By Lemma 4.4.7(b), we have $y \in \mathsf{B}_\tau$, which yields $y \in \mathsf{N}_\tau[x]$. □

Now, it is easy to prove that the constructed assignment satisfies Definition 6.1.2.

Observation 6.2.3. *The function $\mathsf{repr} : \mathsf{N} \to \mathsf{R}$ defined with Construction 6.2.1, where R is the set of all $\mathcal{O}(n \log n)$ basic fragments of T, is a representative assignment.*

Efficient Implementation

Our next goal is to provide an efficient implementation for the representative assignment defined with Construction 6.2.1. To meet condition (1) of Definition 6.1.3, we build step representations of functions mapping a position $i \in [\![1, n-2^k+1]\!]$ to 1 if $T[i\mathinner{.\,.}i+2^k-1] \in \mathsf{N}_{2^k}$ and to 0 otherwise. Formally, such a function is the composition of the function F_{2^k} (defined in Section 4.2, mapping $i \in [\![1, n-2^k+1]\!]$ to $T[i\mathinner{.\,.}i+2^k-1]$) with the characteristic function of N_{2^k}. However, we slightly abuse notation and denote its step representation as $\mathsf{Step}(\mathsf{N}_{2^k})$. Below, we show that these step representations can be constructed efficiently across all $k \in [\![0, \lfloor \log n \rfloor]\!]$. We rely on Lemma 4.4.7, which characterizes N_τ using the family $\mathcal{R}_\tau$ of τ-runs (see Section 4.4.2 for definition), so as an intermediate step we also construct these families for each $\tau = 2^k$.

Fact 6.2.4. *Given a text T of length n,*

- *the families $\mathcal{R}_{2^k}$ of 2^k-runs (in the left-to-right order), and*
- *the step representations $\mathsf{Step}(\mathsf{N}_{2^k})$*

can be computed for all $k \in [\![0, \lfloor \log n \rfloor]\!]$ in $\mathcal{O}(n)$ total time.

Proof. To construct the sets $\mathcal{R}_{2^k}$, we first determine the set $\mathcal{R}(T)$ of all runs in T using Proposition 2.2.5, and we order the runs by their starting positions. Next, we iterate over $\gamma \in \mathcal{R}(T)$ and add γ to the appropriate sets $\mathcal{R}_{2^k}$ (for $\log(3\operatorname{per}(\gamma)) \leq k \leq \log(|\gamma|+1)$; the integer boundaries of this interval can be computed using Proposition 3.1.1). We have $|\mathcal{R}_{2^k}| \leq \frac{3n}{2^k}$ by Lemma 4.4.7, so the total size of the produced sets and the total running time are both $\mathcal{O}(n)$.

By the characterization of Lemma 4.4.7, there is a bijection between 2^k-runs of length at least 2^k and value-0 steps in $\mathsf{Step}(\mathsf{N}_{2^k})$: each such run $T[i\mathinner{.\,.}j]$ corresponds to a step $([\![i, j-2^k+1]\!], 0) \in \mathsf{Step}(\mathsf{N}_{2^k})$. Hence, $\mathsf{Step}(\mathsf{N}_{2^k})$ can be built from $\mathcal{R}_{2^k}$ in time $\mathcal{O}(\frac{n}{2^k})$. □

We are now ready to provide an efficient implementation of the representative assignment of Construction 6.2.1.

Proposition 6.2.5. *Given text T of length n, a representative assignment $\mathsf{repr} : \mathsf{N} \to \mathsf{R}$ with an efficient implementation and $|\mathsf{R}| = \mathcal{O}(n \log n)$ can be constructed in $\mathcal{O}(n)$ time.*

Proof. At the preprocessing phase, we build the step representations $\mathsf{Step}(N_{2^k})$ (Fact 6.2.4) and equip them with the components for evaluation queries (Corollary 3.4.4). The resulting construction time is $\mathcal{O}\Big(n + \sum_{k=0}^{\lfloor \log n \rfloor} \left(\frac{n}{2^k} + \frac{n}{W}\right)\Big) = \mathcal{O}(n)$ and this is also a bound on the data structure size.

Our family R of representative fragments consists of all basic fragments of T, so it is trivial to report it in $\mathcal{O}(|R|) = \mathcal{O}(n \log n)$ time. In order to determine the representative $\mathsf{repr}(x)$ of $x = T[\ell \mathinner{.\,.} r]$, we compute $k = \lfloor \log |x| \rfloor$ (using Proposition 3.1.1) and make an evaluation query asking for the step $(\llbracket \ell', r' \rrbracket, v) \in \mathsf{Step}(N_{2^k})$ containing ℓ. If $v = 1$, then $F_{2^k}(\ell) \in \mathsf{N}_{2^k}$, and we report this fragment as $\mathsf{repr}(x)$. Otherwise, we return $\mathsf{repr}(x) = F_{2^k}(r'+1)$; due to Fact 6.2.2, $x \in \mathsf{N}$ guarantees that $\mathsf{repr}(x)$ is indeed contained in x. □

6.2.2 $\mathcal{O}(n)$ Representative Fragments in $\mathcal{O}(n \log n)$ Time

Our next goal is to reduce the number of representative fragments to $\mathcal{O}(n)$. For this, we pursue a stronger goal of reducing the number of representative fragments of length 2^k to $\mathcal{O}(\frac{n}{2^k})$. In order to let more fragments share the same representative, we alter the partition of N into layers. The kth layer, with representatives of length 2^k, is going to be $\mathsf{L}_k = \{x \in \mathsf{N} : k = \lfloor \log(|x|+1) - 1 \rfloor\}$. This way, $|\mathsf{F}_{2^k}[x]| \ge 2^k$ for each $x \in \mathsf{L}_k$ (although $|\mathsf{N}_{2^k}[x]|$ could be much smaller). To simplify notation, for $m \in \llbracket 1, n \rrbracket$ we set $\mathsf{k}(m) = \lfloor \log(m+1) - 1 \rfloor$ to be the index of the layer containing fragments of length m.

The problem of consistently choosing representative fragments for L_k is similar to the one of constructing a 2^k-synchronizing function (see Section 4.2). In fact, we use the following black-box reduction to build a representative assignment.

Construction 6.2.6. *For each $k \in \llbracket 0, \mathsf{k}(n) \rrbracket$, let sync_k be a 2^k-synchronizing function. For every $x \in \mathsf{L}_k$, we define $\mathsf{repr}(x) = \mathsf{sync}_k(y)$, where y is the leftmost fragment in $\mathsf{F}_{2^{k+1}-1}[x]$ with $\mathsf{sync}_k(y) \ne \bot$.*

It is not hard to prove that Construction 6.2.6 indeed gives a representative assignment.

Lemma 6.2.7. *The function $\mathsf{repr} : \mathsf{N} \to \mathsf{R}$ defined with Construction 6.2.6, based on the 2^k-synchronizing functions sync_k with $k \in \llbracket 0, \mathsf{k}(n) \rrbracket$, is a representative assignment. Here, R consists of all the synchronizing fragments of the functions sync_k.*

Proof. First, we shall prove that repr is well defined. If $x \in \mathsf{L}_k$, then $|x| \ge 2^{k+1} - 1 > 2^k$ and $\mathrm{per}(x) > \frac{1}{3}|x| > \frac{1}{3} \cdot 2^k$, so $\mathsf{N}_{2^k}[x] \ne \emptyset$ by Fact 6.2.2. In particular, $\mathsf{N}_{2^k}[y] \ne \emptyset$ for some $y \in \mathsf{F}_{2^{k+1}-1}[x]$, and thus $\mathsf{sync}_k(y) \ne \bot$ by Definition 4.2.1(a). This means that $\mathsf{repr}(x)$ is indeed well defined and that we have $\mathsf{repr}(x) = \mathsf{sync}_k(y)$ for some $y \in \mathsf{F}_{2^{k+1}-1}[x]$. Hence, $\mathsf{repr}(x) \in \mathsf{N}_{2^k}[y] \subseteq \mathsf{N}_{2^k}[x]$ due to Definition 4.2.1(b). Moreover, $\mathrm{per}(\mathsf{repr}(x)) > \frac{1}{3} \cdot 2^k = \frac{1}{12} \cdot 2^{k+2} > \frac{1}{12}|x|$, which implies condition (a) of Definition 6.1.2.

As for condition (b), note that $x \cong x'$ induces a natural bijection between $\mathsf{F}[x]$ and $\mathsf{F}[x']$ which pairs up matching fragments. Consequently, consistency follows from Fact 4.3.1. □

We conclude with an efficient implementation of a representative assignment based on Construction 6.2.6, with Lemma 4.4.9 applied to build the synchronizing functions. Recall that $\mathsf{Step}(\mathsf{sync}_k)$ is an abbreviated notion for the step representation of $\mathsf{sync}_k \circ F_{2^{k+1}-1}$.

Proposition 6.2.8. *Given a text T of length n, a representative assignment $\mathsf{repr} : \mathsf{N} \to \mathsf{R}$ with an efficient implementation and $|\mathsf{R}| = \mathcal{O}(n)$ can be constructed in $\mathcal{O}(n \log n)$ time.*

Proof. We decompose N into layers L_k with $k \in [\![0, \mathsf{k}(n)]\!]$. For each layer, the representative assignment repr is based on Construction 6.2.6 with the underlying synchronizing function sync_k constructed using Lemma 4.4.9 in $\mathcal{O}(n)$ time.

For an efficient implementation, we build the *evaluators* of Corollary 3.4.4 for the 2^k-synchronizing functions sync_k. The size of a single evaluator is $\mathcal{O}(\frac{n}{W} + |\mathsf{Step}(\mathsf{sync}_k)|) = \mathcal{O}(\frac{n}{W} + \frac{n}{2^k})$, which is $\mathcal{O}(\frac{n \log n}{W} + n) = \mathcal{O}(n)$ in total. The overall construction time is $\mathcal{O}(n \log n)$ due to Lemma 4.4.9.

To list the representative fragments in $\mathcal{O}(|\mathsf{R}|)$ time, we simply report z for each step $([\![\ell, r]\!], z) \in \mathsf{Step}(\mathsf{sync}_k)$ with $z \neq \bot$. The queries for $\mathsf{repr}(x)$ with $x = T[\ell \mathinner{.\,.} r]$ are answered as follows. First, we determine the layer L_k containing x, i.e., we compute $k = \mathsf{k}(|x|)$. Next, we retrieve the step $([\![\ell', r']\!], z) \in \mathsf{Step}(\mathsf{sync}_k)$ with $\ell \in [\![\ell', r']\!]$. If $z \neq \bot$, we report $\mathsf{repr}(x) = z$. Otherwise, we determine the subsequent step (starting at position $r' + 1$) and report its value as $\mathsf{repr}(x)$. The overall query time is constant. □

6.2.3 $\mathcal{O}(n)$-Time Construction for Long Fragments Only

Our final goal is to construct a representative assignment in $\mathcal{O}(n)$ time and to reduce the number of representative fragments to $o(n)$. The latter is impossible if we allow very short fragments in the domain G of repr, so we set $\mathsf{G} = \mathsf{N}_{\ge W}$ defined as $\{x \in \mathsf{N} : |x| \ge W\}$.

The bottleneck of the construction algorithm behind Proposition 6.2.8 is the repeated use of Lemma 4.4.9 to construct the 2^k-synchronizing functions sync_k with step representations of size $\mathcal{O}(\frac{n}{2^k})$. To overcome this issue, we provide an alternative scheme allowing for faster construction of synchronizing functions with larger step representations. The key idea is to transform a τ-synchronizing function sync into a τ'-synchronizing function sync' with $\tau' \ge \tau$. We rely on the sets $\mathsf{N}_{\tau'}$ and $\mathsf{B}_{\tau'}$, the latter defined in Section 4.4.2.

Construction 6.2.9. *Let* sync *be a* τ*-synchronizing function and let* $\tau' \in [\![\tau, \lceil\frac{n}{2}\rceil]\!]$*. We define a function* $\mathsf{sync}' : \mathsf{F}_{2\tau'-1} \to \mathsf{F}_{\tau'} \cup \{\bot\}$ *so that for each* $x \in \mathsf{F}_{2\tau'-1}$*:*

- *If* $\mathsf{N}_{\tau'}[x] = \emptyset$*, then* $\mathsf{sync}'(x) = \bot$*.*
- *If* $\mathsf{B}_{\tau'}[x] \neq \emptyset$*, then* $\mathsf{sync}'(x)$ *is the leftmost fragment of* $\mathsf{B}_{\tau'}[x]$*.*
- *Otherwise,* $\mathsf{sync}'(x) = F_{\tau'}(p)$*, where* $F_\tau(p) = \mathsf{sync}(y)$ *and* $y \in \mathsf{F}_{2\tau-1}[x]$ *is the leftmost fragment with* $\mathsf{sync}(y) \neq \bot$*.*

Below, we prove that Construction 6.2.9 indeed results in a τ'-synchronizing function. Note that at first it is not necessarily clear that $\mathsf{sync}'(x) \subseteq x$ holds in the third case.

Fact 6.2.10. *Construction 6.2.9 defines a* τ'*-synchronizing function* sync'*.*

Proof. Consider a fragment $x \in \mathsf{F}_{2\tau'-1}$. If $\mathsf{sync}'(x) = \bot$, then $\mathsf{N}_{\tau'}[x] = \emptyset$, so condition (a) of Definition 4.2.1 is clearly satisfied. Moreover, if $\mathsf{B}_{\tau'}[x] \neq \emptyset$, then $\mathsf{sync}'(x) \in \mathsf{B}_{\tau'}[x] \subseteq \mathsf{N}_{\tau'}[x]$. Otherwise, $\mathsf{N}_{\tau'}[x] = \mathsf{F}_{\tau'}[x]$ holds by Fact 4.4.6. In particular $z := x[1 \mathinner{.\,.} \tau'] \in \mathsf{N}_{\tau'}[x]$. By Fact 6.2.2, this yields $\mathsf{N}_\tau[z] \neq \emptyset$, so there is a fragment $y' \in \mathsf{F}_{2\tau-1}[x]$ such that $\mathsf{N}_\tau[z \cap y'] \neq \emptyset$. The latter condition yields $\mathsf{sync}(y') \neq \bot$ due to Definition 4.2.1(b). As a result, the fragment y in the definition of $\mathsf{sync}'(x)$ also satisfies $|z \cap y| \ge \tau$. Consequently, $F_\tau(p) = \mathsf{sync}(y)$ has a non-empty overlap with z and therefore $F_{\tau'}(p)$ is contained within x. Hence, $\mathsf{sync}'(x) \in \mathsf{N}_{\tau'}[x]$ holds also in the third case. This concludes the proof that sync' satisfies Definition 4.2.1(b).

Finally, we note that $\mathsf{sync}'(x)$ satisfies condition (c) of Definition 4.2.1 because whenever $x' \cong x$, the natural bijection between $\mathsf{F}[x]$ and $\mathsf{F}'[x]$ pairs up matching fragments, and these fragments are treated consistently by sync as well as by N and B. □

Next, we show how to efficiently implement Construction 6.2.9.

Lemma 6.2.11. *Given the step representation* $\mathsf{Step}(\mathsf{sync})$ *of a* τ*-synchronizing function, an integer* $\tau' \in [\![\tau, \lceil \frac{n}{2} \rceil]\!]$*, and the family* $\mathcal{R}_{\tau'}$ *of* τ'*-runs (in the left-to-right order), in* $\mathcal{O}(\frac{n}{\tau'} + |\mathsf{Step}(\mathsf{sync})|)$ *time we can construct the step representation* $\mathsf{Step}(\mathsf{sync}')$ *of a* τ'*-synchronizing function* sync' *defined with Construction 6.2.9.*

Proof. Steps arising from each of the three cases are constructed separately. For the first two cases, we rely on Lemma 4.4.7, which characterizes $\mathsf{N}_{\tau'}$ and $\mathsf{B}_{\tau'}$ in terms of $\mathcal{R}_{\tau'}$.

In particular, we observe that every τ'-run $\gamma = T[i\,..\,j]$ with $|\gamma| \ge 2\tau' - 1$ gives rise to a step $([\![i, j - 2\tau' + 2]\!], \perp)$; these are the only steps of sync' with value $\perp$.

Next, we use Lemma 4.4.7(b) to generate $\mathsf{B}_{\tau'}$ from $\mathcal{R}_{\tau'}$ (again, in the left-to-right order). Each $z = F_{\tau'}(p)$ belongs to $\mathsf{B}_{\tau'}[F_{2\tau'-1}(i)]$ for $i \in [\![p - \tau' + 1, p]\!]$. This gives rise to a step $([\![p - \tau' + 1, p]\!], z)$. Before inserting this step to $\mathsf{Step}(\mathsf{sync}')$, we trim it so that it does not overlap steps created for the already processed fragments of $\mathsf{B}_{\tau'}$ (i.e., $F_{\tau'}(p') \in \mathsf{B}_{\tau'}$ with $p' < p$) and to make sure that it is contained in the domain $[\![1, n - 2\tau' + 2]\!]$ of $\mathsf{F}_{2\tau'-1}$.

The gaps between the already created steps of sync' need to be filled based on sync according to the third case of Construction 6.2.9. For this, we first remove the $\perp$-value steps of sync and fill each of the created gaps by extending the step to the right of the gap. The resulting function maps a position i to the value $\mathsf{sync}(y)$ of the leftmost fragment $y = F_{2\tau-1}(j)$ with $j \ge i$ and $\mathsf{sync}(y) \ne \perp$. Next, we replace any value $F_\tau(p)$ with $F_{\tau'}(p)$, removing the trailing steps for which this is impossible due to $p > n - \tau' + 1$. Finally, we fill all the gaps of sync' by copying the corresponding parts of this auxiliary step representation.

Observe that the first two phases of the algorithm take $\mathcal{O}(\frac{n}{\tau'})$ time because this is an upper bound on $\mathcal{R}_{\tau'}$ (see Lemma 4.4.7), while the construction of the auxiliary step representation takes $\mathcal{O}(|\mathsf{Step}(\mathsf{sync})|)$ time. Hence, $|\mathsf{Step}(\mathsf{sync}')| = \mathcal{O}(\frac{n}{\tau'} + |\mathsf{Step}(\mathsf{sync})|)$ and this is also an upper bound on the overall running time. □

Finally, we use Lemma 6.2.11 to build an efficient representative assignment for the family $\mathsf{N}_{\ge W}$ of non-highly-periodic fragments of length at least W.

Proposition 6.2.12. *Given text* T *of length* n*, a representative assignment* $\mathsf{repr} : \mathsf{N}_{\ge W} \to \mathsf{R}$ *with an efficient implementation and* $|\mathsf{R}| = \mathcal{O}(\frac{n \log W}{W})$ *can be constructed in* $\mathcal{O}(n)$ *time.*

Proof. We decompose $\mathsf{N}_{\ge W}$ into layers L_k for $k \in [\![\mathsf{k}(W), \mathsf{k}(n)]\!]$ (the first of these layers does not need to be fully contained within $\mathsf{N}_{\ge W}$). For each later L_k, we construct a 2^k-synchronizing function sync_k and define repr based on Construction 6.2.6. We build $\mathsf{sync}_{\mathsf{k}(W)}$ using Lemma 4.4.9 so that its step representation is of size $\mathcal{O}(\frac{n}{W})$. Based on $\mathsf{sync}_{\mathsf{k}(W)}$, we build 2^k-synchronizing functions sync_k for subsequent layers L_k with $\mathsf{k}(W) < k < \mathsf{k}(W^2)$ using Lemma 6.2.11. The total size of their step representations is $\mathcal{O}(\frac{n \log W}{W})$. For $k = \mathsf{k}(W^2)$, we use Lemma 4.4.9 again so that in this case $|\mathsf{Step}(\mathsf{sync}_{\mathsf{k}(W^2)})| = \mathcal{O}(\frac{n}{W^2})$. For the remaining layers L_k (with $\mathsf{k}(W^2) < k \le \mathsf{k}(n)$), we apply Lemma 6.2.11 plugging $\mathsf{sync}_{\mathsf{k}(W^2)}$ to obtain 2^k-synchronizing functions sync_k. The total size of their step representations is $\mathcal{O}(\frac{n \log n}{W^2}) = \mathcal{O}(\frac{n}{W})$.

Across all layers, the total size of step representations is $\mathcal{O}(\frac{n \log W}{W})$, as claimed. The construction algorithm needs $\mathcal{O}(n)$ time both to use Lemma 4.4.9 twice and to generate the families $\mathcal{R}_{2^k}$ of 2^k-runs for each layer L_k (see Fact 6.2.4). On the top of that, applications of Lemma 6.2.11 take $\mathcal{O}(\frac{n \log W}{W}) = o(n)$ time in total.

Once the synchronizing functions are constructed, the efficient implementation of the resulting representative assignment repr is exactly as in the proof of Proposition 6.2.8. □

6.3 Implementation of the Data Structure

In this section, we give a complete description of the data structure for IPM Queries. As mentioned in Section 6.2.1, periodic patterns are supported using Theorem 5.2.8. We also need a specialized component for short patterns (of length $|x| < W$), which is described in Section 6.3.1. In Section 6.3.2, we present the main part of the data structure which handles long non-periodic patterns using the representative assignment of Proposition 6.2.12.

6.3.1 Short Patterns

Before we present our data structure for IPM Queries with patterns x of length $|x| < W$, let us consider a special case of a very short text of length $n = \mathcal{O}(W)$. In the general case, we then partition the text T into overlapping blocks of length $\mathcal{O}(W)$.

Lemma 6.3.1. *Let T be a text of length $n = \mathcal{O}(W)$ over an alphabet consisting of W-bit integers. In $\mathcal{O}(n)$ time, one can construct a data structure of size $\mathcal{O}(n)$ which, given a fragment x of T, in $\mathcal{O}(1)$ time computes a bitmask representing the starting positions of the fragments of T matching x.*

Proof. We specify the data structure using the suffix array $SA[1\mathinner{.\,.} n]$, the inverse suffix array $ISA[1\mathinner{.\,.} n]$, and the LCP table $LCP[2\mathinner{.\,.} n]$ of the text T; see Section 2.5 for definitions. It consists of the following components:

- the inverse suffix array ISA of the text T;
- for each value $k \in [\![1, n+1]\!]$:
  - a bitmask $I_k[1\mathinner{.\,.} n]$ such that $I_k[i] = 1$ if and only if $ISA[i] < k$,
  - a bitmask $L_k[1\mathinner{.\,.} n+1]$ such that $L_k[i] = 1$ if and only if $i \in \{1, n+1\}$ or $LCP[i] < k$.

Clearly, the inverse suffix array takes $\mathcal{O}(n)$ space and each of the $2n+2$ bitmasks can be stored in $\mathcal{O}(\lceil \frac{n}{W} \rceil) = \mathcal{O}(1)$ machine words. Hence, the size of the data structure is $\mathcal{O}(n)$.

Let us proceed to a description of the query algorithm for $x = T[i\mathinner{.\,.} j]$. First, we retrieve the predecessor ℓ and the successor r of $ISA[i]$ in $L_{|x|}$. To compute ℓ, we mask out bits at positions larger than $ISA[i]$ and retrieve the most significant bit using Proposition 3.1.1. Similarly, to determine r, we mask out bits at positions $ISA[i]$ or smaller and compute the least significant bit using Corollary 3.1.2. Now, $[\![\ell, r-1]\!]$ is the maximal interval containing $ISA[i]$ such that $LCP[p] \ge |x|$ for $p \in [\![\ell+1, r-1]\!]$. Consequently, a fragment matching x starts at position i' of T if and only if $ISA[i'] \in [\![\ell, r-1]\!]$. The latter condition can be expressed as $I_r[i'] = 1$ and $I_\ell[i'] = 0$, so the resulting bitmask is the bit-wise 'and' of I_r and the complement of I_ℓ. This concludes the implementation of an $\mathcal{O}(1)$-time query algorithm.

It remains to develop an $\mathcal{O}(n)$-time construction procedure. As noted in Section 2.5, the suffix array, the inverse suffix array, and the LCP table can be constructed in $\mathcal{O}(n)$ time. To compute the bitmasks I_k, we observe that I_1 consists of zeros only, and I_{k+1} can be obtained from I_k by setting to one the bit at index $i = SA[k]$ (so that $ISA[i] = k$). Thus, these bitmasks can be computed in $\mathcal{O}(1)$ time each. The construction of bitmasks L_k is similar: to obtain L_{k+1} from L_k, we need to set to one every bit at an index i such that $LCP[i] = k$. The list of such indices i for each k can be determined by sorting the pairs $(LCP[i], i)$ for $i \in [\![2, n]\!]$. The overall running time is clearly $\mathcal{O}(n)$. □

Corollary 6.3.2. *There is a data structure of size $\mathcal{O}(n)$ which can answer* IPM QUERIES *in $\mathcal{O}(1)$ time for patterns x of length $|x| < W$. It can be constructed in $\mathcal{O}(n)$ time.*

Proof. Our data structure contains the suffix array $SA[1\mathinner{.\,.} n]$ of the text T, the inverse suffix array $ISA[1\mathinner{.\,.} n]$, and the component for LCE QUERIES in T; see Proposition 2.5.1. Moreover, we introduce $m = \lceil \frac{n}{2W} \rceil$ *blocks*—fragments $T_1, \ldots, T_m$ of length $4W$ and starting every $2W$ positions, i.e., $T_b = T[2W(b-1)+1\mathinner{.\,.} 2W(b+3)]$ (the last two blocks are trimmed to fit within T). For each block T_b, we build the data structure of Lemma 6.3.1 and a fusion tree (of Theorem 3.4.1) storing values $ISA[j]$ for the leftmost $3W$ positions j contained in the block (or $|T_b|$ positions if the block is shorter than $3W$). These components clearly take $\mathcal{O}(n)$ space and can be constructed in $\mathcal{O}(n)$ time in total.

Hence, the main challenge is to provide an $\mathcal{O}(1)$-time query algorithm. Suppose that we are given a query with a pattern x and a text y, starting at positions i_x and i_y, respectively. Observe that the rightmost block containing y is T_b for $b = \lceil \frac{i_y}{2W} \rceil$. Moreover, y starts within the first $2W$ positions of T_b, so the sought occurrences of x must start within the first $2W + |y| - |x| < 2W + |x| < 3W$ positions of T_b. If a fragment matching x starts at any position $p \in [\![1, 3W]\!]$ of T_b, we could reduce the query in T to a query in T_b. Otherwise, we are guaranteed that no occurrence of x is contained in y.

To take this approach, we use the fusion tree for the bth block to retrieve the predecessor ℓ and the successor r of $ISA[i_x]$ (among the values $ISA[j]$ for the leftmost $3W$ positions j contained in the block). Observe that if x occurs at one of these positions, then it also occurs at position $SA[\ell]$ or $SA[r]$. We verify this by checking if $\mathrm{LCE}(i_x, SA[\ell]) \geq |x|$ and $\mathrm{LCE}(i_x, SA[r]) \geq |x|$, respectively. If x occurs at neither of the two positions, we report that it has no occurrence in y. Otherwise, we have a fragment x' contained in T_b and matching x. We apply the component of Lemma 6.3.1 to retrieve a bitmask of positions where the fragments matching x' (equivalently, the fragments matching x) start in T_b. We mask out positions which do not correspond to occurrences contained in y and use Proposition 3.1.1 and Corollary 3.1.2 to retrieve positions of the first, the second, and the last occurrence of x in y. These at most three positions let us derive the arithmetic sequence representing all the occurrences. Each step of the query algorithm takes $\mathcal{O}(1)$ time. □

6.3.2 Long Patterns

In this section, we present the main component responsible for IPM QUERIES with patterns $x \in \mathsf{N}_{\geq W}$. Our implementation relies on an arbitrary representative assignment $\mathsf{repr} : \mathsf{N}_{\geq W} \to \mathsf{R}$ with an efficient implementation. However, we provide the complexity analysis based on the bounds obtained in Proposition 6.2.12 for a specific construction.

As outlined in Section 6.1, to search for the occurrences of x in y, we first find the *representative occurrences* of $\mathsf{repr}(x)$ contained in y, i.e., representative fragments matching $\mathsf{repr}(x)$ and contained in y. This step is implemented using auxiliary RESTRICTED IPM QUERIES specified below. Next, we apply LCE QUERIES (see Section 2.5) to check which of these representative occurrences can be extended to fragments matching x and contained in y; see also Figure 6.2.

RESTRICTED IPM QUERIES
Input: A text T and a family $\mathsf{R} \subseteq \mathsf{F}$ of fragments of T.
Queries: Given a fragment $x \in \mathsf{R}$ and a fragment $y \in \mathsf{F}$, report all fragments $x' \in \mathsf{R}$ contained in y and matching x.

Due to the sparsity of occurrences (Fact 6.1.1), it is relatively easy to implement Restricted IPM Queries in $\mathcal{O}(|y|/\operatorname{per}(x))$ time using deterministic dictionaries [131].

Lemma 6.3.3. *For a text T of length n and a family $\mathsf{R} \subseteq \mathsf{F}$ of fragments of T, there exists a data structure of size $\mathcal{O}(|\mathsf{R}|)$ that answers* Restricted IPM Queries *in $\mathcal{O}(|y|/\operatorname{per}(x))$ time. It can be constructed in $\mathcal{O}(n + |\mathsf{R}| \log^2 \log |\mathsf{R}|)$ time.*

Proof. Given the family R, we construct an identifier function id such that $\mathsf{id}(x) = \mathsf{id}(x')$ if and only if the two fragments $x, x' \in \mathsf{R}$ match. For this, we order the fragments $x = T[\ell\,..\,r] \in \mathsf{R}$ by the length $|x|$ and the lexicographic rank $ISA[\ell]$ of the suffix $T[\ell\,..]$ among the suffixes of T (see Section 2.5). Matching fragments $x \in \mathsf{R}$ appear consecutively in this order, so we use LCE Queries (see Proposition 2.5.1) to determine the boundaries between the equivalence classes. Finally, we store the id function in a static dictionary [131] mapping each fragment $x = T[\ell\,..\,r] \in \mathsf{R}$ (represented by the positions ℓ and r) to the identifier $\mathsf{id}(x)$. The overall running time of this phase is $\mathcal{O}(n + |\mathsf{R}| \log^2 \log |\mathsf{R}|)$.

Next, consider a class of matching fragments $x \in \mathsf{R}$ with a common identifier $i = \mathsf{id}(x)$ and length $m = |x|$. We partition the text T into blocks of length $2m - 1$ with overlaps of length $m - 1$ (the last block can be shorter). The resulting family of blocks is denoted by $\mathsf{Y}(i)$. We precompute the answers to Restricted IPM Queries with $\mathsf{id}(x) = i$ and $y \in \mathsf{Y}(i)$, and we store non-empty answers in another static dictionary [131]. Note that $\{\mathsf{F}_m[y] : y \in \mathsf{Y}(i)\}$ forms a partition of F_m, so each $x \in \mathsf{R}$ appears in exactly one precomputed answer. Consequently, the running time of this phase is $\mathcal{O}(|\mathsf{R}| \log^2 \log |\mathsf{R}|)$ in total across all classes.

To answer a query, we first compute $i = \mathsf{id}(x)$ and $m = |x|$. Next, we use simple arithmetics to obtain $\mathcal{O}(|y|/|x|)$ blocks $y' \in \mathsf{Y}(i)$ such that $\mathsf{F}_m[y]$ is contained in the union of $\mathsf{F}_m[y']$ across these blocks. We also take the union of the corresponding precomputed answers to obtain a collection of fragments $x' \in \mathsf{R}$ matching x and contained in one of the blocks y'. By Fact 6.1.1, there are $\mathcal{O}(|y|/\operatorname{per}(x))$ such fragments x', so we can filter and report those contained in y, spending $\mathcal{O}(1)$ time on each candidate x'. □

We conclude with a full description of the data structure supporting IPM Queries.

Theorem 1.1.4. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which answers* IPM Queries *in $\mathcal{O}(1)$ time. It can be constructed in $\mathcal{O}(n)$ time.*

Proof. The main component of our data structure is an efficient implementation of the representative assignment $\mathsf{repr} : \mathsf{N}_{\geq W} \to \mathsf{R}$ constructed using Proposition 6.2.12. Additionally, we include the components for Restricted IPM Queries (Lemma 6.3.3), for LCE Queries (Proposition 2.5.1), for IPPM Queries (Theorem 5.2.8), and for IPM Queries with short patterns (Corollary 6.3.2).

The efficient representative assignment takes $\mathcal{O}(n)$ space due to Definition 6.1.3, and Proposition 6.2.12 provides an $\mathcal{O}(n)$-time construction algorithm. Furthermore, Proposition 6.2.12 guarantees that the family R of representative fragments satisfies $|\mathsf{R}| = \mathcal{O}(\frac{n \log W}{W})$ and can be enumerated in $\mathcal{O}(|\mathsf{R}|)$ time. Consequently, the component for Restricted IPM Queries takes $\mathcal{O}(n + \frac{n \log W \log^2 \log n}{W}) = \mathcal{O}(n + \frac{n \log^3 \log n}{\log n}) = \mathcal{O}(n)$ time to construct. The remaining three components are also built in $\mathcal{O}(n)$ time each.

The query algorithm for fragments x and y works as follows. If $|x| < W$, we simply forward the query to the component of Corollary 6.3.2. For $|x| \geq W$, we first try using the component of Theorem 5.2.8. It successfully answers the query if x is periodic. Otherwise, it reports an error, and we are guaranteed that $x \in \mathsf{N}_{\geq W}$. By condition (1) of Definition 6.1.3,

$\mathsf{repr}(x)$ can therefore be retrieved in $\mathcal{O}(1)$ time. Next, we query the component of Lemma 6.3.3 to find all representative fragments matching $\mathsf{repr}(x)$ and contained in y. Due to Definition 6.1.2(a) and $|y| < 2|x|$, this takes $\mathcal{O}(|y|/\operatorname{per}(\mathsf{repr}(x))) = \mathcal{O}(1)$ time. The consistency of the representative assignment (Definition 6.1.2(b)) guarantees that for every fragment x' contained y and equal to x, the representative $\mathsf{repr}(x')$ is one of the identified representative fragments. Thus, we have a constant number of positions where x may occur in y. We verify them using LCE QUERIES in $\mathcal{O}(1)$ time each. Since x is not periodic in the current case, by Fact 6.1.1 it has at most two occurrences in y. Consequently, their starting positions trivially form an arithmetic progression. □

Chapter 7

Applications of IPM Queries

In this short chapter, we present immediate applications of our data structure for Internal Pattern Matching Queries. This includes answering Period Queries (Section 7.2) and variants of LZ Substring Compression Queries (Section 7.3). Before that, we introduce a few auxiliary tools which are useful for processing the output of IPM Queries not only here but also in Chapter 8.

7.1 Periodic Progressions

We say that a sequence $\mathbf{p}$ of positions $p_0 < p_1 < \cdots < p_{k-1}$ in a string w is a *periodic progression* of length k (in w) if $w[p_0\mathinner{.\,.} p_1 - 1] \cong \cdots \cong w[p_{k-2}\mathinner{.\,.} p_{k-1} - 1]$. If $k \geq 2$, we call the string $v \cong w[p_i\mathinner{.\,.} p_{i+1} - 1]$ the *(string) period* of $\mathbf{p}$, while its length $p_{i+1} - p_i$ is the *difference* of $\mathbf{p}$. Periodic progressions $\mathbf{p}, \mathbf{p}'$ are called *non-overlapping* if the last term of $\mathbf{p}$ is smaller than the first term of $\mathbf{p}'$ or vice versa, the last term of $\mathbf{p}'$ is smaller than the first term of $\mathbf{p}$. Note that every periodic progression is an arithmetic progression and consequently it can be represented by three integers, e.g., the terms p_0, p_1, and p_{k-1} (with p_1 omitted if $k = 1$, i.e., if $p_{k-1} = p_0$). Periodic progressions appear in our work because of the following characterization (see also Fact 5.2.2):

Observation 7.1.1. *Let x, y be fragments of a string w satisfying $|y| \leq 2|x|$. The positions where x occurs in y form a periodic progression in w.*

All our applications of IPM Queries rely on the structure of the values $\mathrm{LCE}(p_i, q)$ for a periodic progression $(p_i)_{i=0}^{k-1}$. Below, we formally state this combinatorial property (in a slightly more general form) and group its immediate algorithmic applications in a single black-box lemma. Let us start with a simple combinatorial result.

Fact 7.1.2. *For strings $u, v \in \Sigma^*$ and $\rho \in \Sigma^+$, let $d_u = \mathrm{lcp}(\rho^\infty, u)$ and $d_v = \mathrm{lcp}(\rho^\infty, v)$.*
(a) If $d_u > d_v$, then $\mathrm{lcp}(u, v) = d_v$ and $u \prec v \Leftrightarrow \rho^\infty \prec v$.
(b) If $d_u = d_v$, then $\mathrm{lcp}(u, v) \geq d_u = d_v$.
(c) If $d_u < d_v$, then $\mathrm{lcp}(u, v) = d_u$ and $u \prec v \Leftrightarrow u \prec \rho^\infty$.

Proof. Let $d = \min(d_u, d_v)$. Note that $u[1\mathinner{.\,.} d] \cong (\rho^\infty)[1\mathinner{.\,.} d] \cong v[1\mathinner{.\,.} d]$, so $\mathrm{lcp}(u, v) \geq d$. If $d = d_u < d_v$, then $v[d + 1] = (\rho^\infty)[d + 1] \neq u[d + 1]$, so $\mathrm{lcp}(u, v) = d$. If $|u| = d$, then u is a common prefix v and ρ^∞. Otherwise, $u \prec v$ is equivalent to $u[d + 1] \prec v[d + 1]$, $u[d + 1] \prec (\rho^\infty)[d + 1]$, and $u \prec \rho^\infty$. The case of $d = d_v < d_u$ is symmetric. □

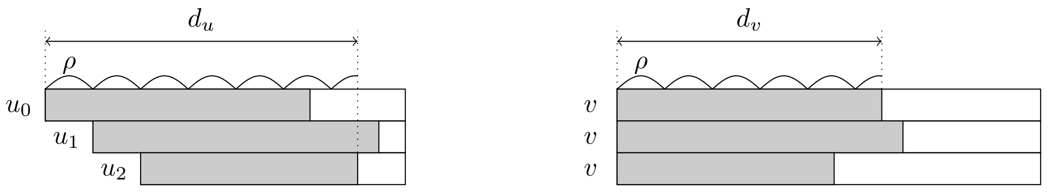


Figure 7.1: An illustration of notions used in Corollary 7.1.3 and Lemma 7.1.4. Shaded rectangles represent the common prefixes of u_i and v. In this case, $\frac{d_u - d_v}{|\rho|} = 1$.

Corollary 7.1.3. *Let $\mathbf{p} = (p_i)_{i=0}^{k-1}$ be a periodic progression of length $k \ge 2$ in a string w and, for $i \in [\![0, k-1]\!]$, let u_i be substrings of w with occurrences starting at positions p_i and ending at a common position. Also, let v be an arbitrary string.*

Denote $d_u = \mathrm{lcp}(\rho^\infty, u_0)$ and $d_v = \mathrm{lcp}(\rho^\infty, v)$, where ρ is the string period of $\mathbf{p}$. Then we have $\rho^\infty \prec u_0 \prec \cdots \prec u_{k-1}$ or $\rho^\infty \succ u_0 \succ \cdots \succ u_{k-1}$. Moreover,

(a) if $i < \frac{d_u - d_v}{|\rho|}$, then $\mathrm{lcp}(u_i, v) = d_v$ and $u_i \prec v \Leftrightarrow \rho^\infty \prec v$;

(b) if $i = \frac{d_u - d_v}{|\rho|}$, then $\mathrm{lcp}(u_i, v) \ge d_u - i|\rho| = d_v$;

(c) if $i > \frac{d_u - d_v}{|\rho|}$, then $\mathrm{lcp}(u_i, v) = d_u - i|\rho|$ and $u_i \prec v \Leftrightarrow u_0 \prec \rho^\infty$.

Proof. Observe that $u_0 = \rho^i u_i$, so $\mathrm{lcp}(\rho^\infty, u_i) = \mathrm{lcp}(\rho^\infty, u_0) - i|\rho| = d_u - i|\rho|$.

We start with the first claim. We shall inductively prove that $\rho^\infty \prec u_0 \prec \cdots \prec u_i$ or $\rho^\infty \succ u_0 \succ \cdots \succ u_i$ (for $i = 0, \ldots, k-1$). This is trivial for $i = 0$ due to $u_0 \ne \rho^\infty$. For the proof of the inductive step, we apply Fact 7.1.2 with $(u, v) := (u_i, u_{i+1})$. We have $\mathrm{lcp}(\rho^\infty, u_i) > \mathrm{lcp}(\rho^\infty, u_{i+1})$, so $u_i \prec u_{i+1}$ is equivalent to $\rho^\infty \prec u_{i+1} = \rho \cdot u_i$ and therefore to $\rho^\infty \prec u_i$. Thus, $\rho^\infty \prec u_0 \prec \cdots \prec u_i$ yields $u_i \prec u_{i+1}$ and $\rho^\infty \succ u_0 \succ \cdots \succ u_i$ yields $u_i \succ u_{i+1}$, which completes the inductive proof.

Since $d_v = d_u - \frac{d_u - d_v}{|\rho|} \cdot |\rho|$, the claims in cases (a)–(c) immediately follow from Fact 7.1.2 applied for $(u, v) := (u_i, v)$. □

Although each of our applications uses Corollary 7.1.3 for a slightly different purpose, the overall scheme is the same each time. Consequently, we group the application-specific queries in a single algorithmic lemma.

Lemma 7.1.4. *Suppose that we are given a text T equipped with a data structure answering* LCE Queries *in constant time. Given a fragment v of T and a collection of fragments $u_i = T[p_i \mathinner{.\,.} r]$ represented with a periodic progression $\mathbf{p} = (p_i)_{i=0}^{k-1}$ and a common end position r, the following queries can be answered in constant time:*

(a) Report all indices i such that u_i occurs as a prefix of v.

(b) Report all indices i such that $u_i \prec v$ (or all indices i such that $u_i \succ v$).

(c) Report all indices i such that $\mathrm{lcp}(u_i, v)$ is maximized (along with the lcp *value).*

The results are represented as subintervals of $[\![0, k-1]\!]$.

Proof. There is nothing to do for $k = 0$; for $k = 1$, Fact 2.5.2 lets us easily check if u_0 satisfies the required conditions. Thus, we shall assume $k \ge 2$. In this case, we retrieve an occurrence $T[p_0 \mathinner{.\,.} p_1 - 1]$ of the string period ρ and apply Fact 2.5.2(c) to determine d_u and d_v (as defined in Corollary 7.1.3). We also compute $i_t = \frac{d_u - d_v}{|\rho|}$.

(a) We shall report i such that $\mathrm{lcp}(u_i, v) = |u_i|$. For $i < i_t$, we have $\mathrm{lcp}(u_i, v) = d_v < d_u - i|\rho| \le |u_0| - i|\rho| = |u_i|$, so these indices are never reported. If i_t is a valid index, we

compute $\mathrm{lcp}(u_{i_t}, v)$ using Fact 2.5.2(a) and this index may need to be reported. For $i > i_t$, we have $\mathrm{lcp}(u_i, v) = d_u - i|\rho|$ and $|u_i| = |u_0| - i|\rho|$, so we either report all these indices (if $d_u = |u_0|$), or none of them (otherwise).
(b) In this query, we apply Fact 2.5.2(c) to retrieve the order between ρ^∞ and u_0, and between ρ^∞ and v. For $i < i_t$, we either report all the indices or none (depending on whether $\rho^\infty \prec v$). For $i > i_t$, we also either report all the indices or none (depending on whether $u_0 \prec \rho^\infty$). If i_t is a valid index, we use Fact 2.5.2(b) to manually check if the index needs to be reported; note that this is the only case where $u_i \cong v$ is possible. Recall that the sequence $(u_i)_{i=0}^{k-1}$ is monotone, so the result is always a single interval.
(c) Note that $\mathrm{lcp}(u_i, v) \le d_v$ unless $i = i_t$. Hence, if i_t is a valid index and $\mathrm{lcp}(u_{i_t}, v) > d_v$, then the only index maximizing $\mathrm{lcp}(u_i, v)$ is i_t. Otherwise, we have $\mathrm{lcp}(u_i, v) = d_v$ if and only if $i \le i_t$. Hence, provided that $i_t \ge 0$, we report all indices i satisfying $i \le i_t$. Finally, if $i_t < 0$, then we have $\mathrm{lcp}(u_i, v) = d_u - i|\rho|$ for every i, and the only maximum is d_u attained at $i = 0$. □

7.2 Period Queries

In this section, we show the solution for Prefix-Suffix Queries and Period Queries using IPM Queries. Let us recall the definition of the former.

Prefix-Suffix Queries
Given fragments x and y of the text T and a positive integer d, report all suffixes of y of length between d and $2d-1$ that also occur as prefixes of x (represented as an arithmetic progression of their lengths).

We assume that $|x|, |y| \ge d$; otherwise, there are no suffixes to report. Let x' be the prefix of x of length d and y' be the suffix of y of length $\min(2d-1, |y|)$. Suppose that a suffix z of y occurs as a prefix x. If $|z| \ge d$, then z must start with a fragment matching x'. Moreover, if $|z| \le 2d-1$, then z is a suffix of y', so this yields an occurrence of x' in y'. We find all such occurrences with a single IPM Query and then use Lemma 7.1.4 to find out which of them can be extended to the sought suffixes z of y.

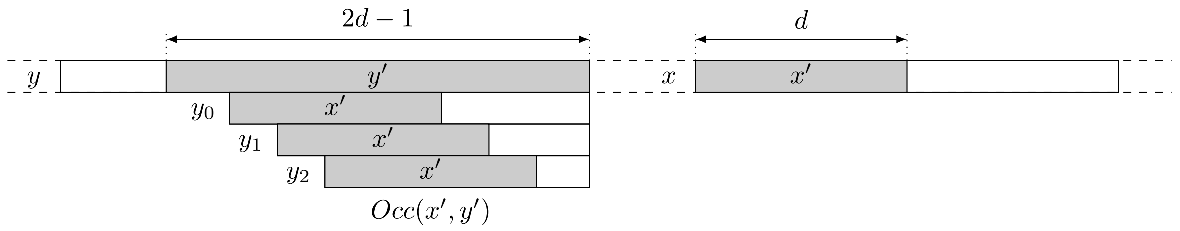


Figure 7.2: The notions used in the algorithms answering Prefix-Suffix Queries and Bounded Longest Common Prefix Queries.

More formally, denote by $Occ(x', y')$ the set of starting positions of fragments matching x' and contained in y'. By Observation 7.1.1, $Occ(x', y')$ forms a periodic sequence of positions in T. Let y_i be the suffix of y starting with the ith occurrence of x'; see Figure 7.2. We need to check which of the fragments y_i occur as prefixes of x. This is possible using Lemma 7.1.4(a), which lets us find all indices i such that y_i is a prefix of x. The result is an integer interval of indices, which can be transformed into an arithmetic sequence of lengths $|y_i|$. Consequently, the data structure of Theorem 1.1.4 (which already contains the component of Proposition 2.5.1 for LCE Queries) can answer Prefix-Suffix Queries in $\mathcal{O}(1)$ time. Hence, we obtain the following results.

Theorem 1.1.3. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which answers* Prefix-Suffix Queries *in $\mathcal{O}(1)$ time. It can be constructed in $\mathcal{O}(n)$ time.*

Period Queries
Given a fragment x of the text T, report all periods of x (represented by non-overlapping arithmetic progressions).

Theorem 1.1.2. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which answers* Period Queries *in $\mathcal{O}(\log|x|)$ time. It can be constructed in $\mathcal{O}(n)$ time.*

Proof. Period Queries can be answered using the data structure for Prefix-Suffix Queries. To compute all periods of x, we use Prefix-Suffix Queries to find all borders of x of length between 2^k and $2^{k+1}-1$ for each $k \in [\![0, \lfloor\log|x|\rfloor]\!]$. The lengths of borders can be easily transformed to periods since x has period p if and only if it has a border of length $|x|-p$. □

7.3 LZ Substring Compression Queries

In this section, we consider LZ Substring Compression Queries and its variants based on several types of LZ factorizations introduced in Section 2.3.1.

(Non-Overlapping) LZ Substring Compression Queries
Given a fragment x of the text T, compute the (non-overlapping) LZ factorization of x, i.e., $LZ(x)$ ($LZ_N(x)$, respectively).

Relative LZ Substring Compression Queries
Given two fragments x and y of the text T, compute the relative LZ factorization of x with respect to y, i.e., $LZ_R(x|y)$.

Generalized (Non-Overlapping) LZ Substring Compression Queries
Given two fragments x and y of the text T, compute the generalized (non-overlapping) LZ factorization of x with respect to y, i.e., $LZ_G(x|y)$ ($LZ_{GN}(x|y)$, respectively).

Our query algorithms heavily rely on the results of Keller et al. [83] for LZ Substring Compression Queries and Generalized LZ Substring Compression Queries. The main improvement is a more efficient solution for the following auxiliary problem:

Bounded Longest Common Prefix Queries
Given two fragments x and y of the text T, find the longest prefix p of x which occurs in y.

The other, easier auxiliary problem defined in [83] is used as a black box.

INTERVAL LONGEST COMMON PREFIX QUERIES
Given a fragment x of the text T and an interval $[\![\ell, r]\!]$ of positions in T, find the longest prefix p of x which occurs in T at some position within $[\![\ell, r]\!]$.

The data structure for INTERVAL LONGEST COMMON PREFIX QUERIES is based on range successor queries, defined in Section 3.7. Several trade-offs are available for these queries (they are listed in Proposition 3.7.5), so we state the complexity in an abstract form. This convention gets propagated to further results in this section.

Lemma 7.3.1 (Keller et al. [83]). *For a text T of length n, there exists a data structure of size $\mathcal{O}(n + S_{rsucc}(n))$ that answers* INTERVAL LONGEST COMMON PREFIX QUERIES *in $\mathcal{O}(Q_{rsucc}(n))$ time. The data structure can be constructed in $\mathcal{O}(n + C_{rsucc}(n))$ time.*

As observed in [83], the decision version of IPM QUERIES easily reduces to INTERVAL LONGEST COMMON PREFIX QUERIES. For $x = T[\ell_x \mathinner{.\,.} r_x]$ and $y = T[\ell_y \mathinner{.\,.} r_y]$, it suffices to check if the longest prefix of x occurring at some position in $[\![\ell_y, r_y - r_x + \ell_x]\!]$ is x itself.

Corollary 7.3.2 (Keller et al. [83]). *For a text T of length n, there exists a data structure of size $\mathcal{O}(n+S_{rsucc}(n))$ that, given fragments x, y of the text T, can decide in $\mathcal{O}(Q_{rsucc}(n))$ time whether x occurs in y. The data structure can be constructed in $\mathcal{O}(n + C_{rsucc}(n))$ time.*

We proceed with our solution for BOUNDED LONGEST COMMON PREFIX QUERIES. Let $x = T[\ell_x \mathinner{.\,.} r_x]$ and $y = T[\ell_y \mathinner{.\,.} r_y]$. First, we search for the largest k such that the prefix of x of length 2^k (i.e., $T[\ell_x \mathinner{.\,.} \ell_x + 2^k - 1]$) occurs in y. We use a variant of the binary search involving exponential search (also called galloping search), which requires $\mathcal{O}(\log K)$ steps, where K is the optimal value of k. At each step, for a fixed k we need to decide if $T[\ell_x \mathinner{.\,.} \ell_x + 2^k - 1]$ occurs in y. This can be done in $\mathcal{O}(Q_{rsucc}(n))$ time using Corollary 7.3.2. At this point, we have an integer K such that the optimal prefix p has length $|p| \in [\![2^K, 2^{K+1} - 1]\!]$. The running time is $\mathcal{O}(Q_{rsucc}(n) \log K) = \mathcal{O}(Q_{rsucc}(n) \log\log |p|)$ so far.

Let p' be the prefix obtained from an INTERVAL LONGEST COMMON PREFIX QUERY for x and $[\![\ell_y, r_y - 2^{K+1} + 1]\!]$. Note that $T[\ell_x \mathinner{.\,.} \ell_x + 2^{K+1} - 1]$ does not occur in y, so $|p'| < 2^{K+1}$ and thus the occurrence of p' starting in $[\![\ell_y, r_y - 2^{K+1} + 1]\!]$ lies within y. Consequently, $|p| \geq |p'|$; moreover, if p occurs at a position within $[\![\ell_y, r_y - 2^{K+1} + 1]\!]$, then $p = p'$.

The other possibility is that p only occurs near the end of y, i.e., within the suffix of y of length $2^{K+1} - 1$, which we denote as y'. We use a similar approach as for PREFIX-SUFFIX QUERIES with $d = 2^K$ to detect p in this case. We define x' as the prefix of x of length 2^K. Note that an occurrence of p must start with an occurrence of x', so we find all occurrences of x' in y'. If there are no such occurrences, we conclude that $p = p'$. Otherwise, we define y_i as the suffix of y starting with the ith occurrence of x; see Figure 7.2. Next, we apply Lemma 7.1.4(c) to compute $\max_i \operatorname{lcp}(y_i, x)$. By the discussion above, this must be the length of the longest prefix of x which occurs in y'. We compare its length to $|p'|$ and choose the final answer p as the longer of the two candidates.

Thus, the data structure for IPM QUERIES, accompanied by the components of Lemma 7.3.1, Corollary 7.3.2, and Proposition 2.5.1, yields the following result:

Theorem 1.1.5. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n + S_{rsucc}(n))$ which answers* BOUNDED LONGEST COMMON PREFIX QUERIES *in time $\mathcal{O}(Q_{rsucc}(n) \log\log |p|)$. It can be constructed in $\mathcal{O}(n + C_{rsucc}(n))$ time.*

Finally, we generalize the approach of [83] to support multiple types of LZ Substring Compression Queries using Theorem 1.1.5 to improve the running time.

Theorem 7.3.3. *For every text T of length n, there is a data structure of size $\mathcal{O}(n + S_{rsucc}(n))$ that answers:*

(a) Non-Overlapping LZ Substring Compression Queries*,*

(b) Relative LZ Substring Compression Queries*,*

(c) Generalized LZ Substring Compression Queries*, and*

(d) Generalized Non-Overlapping LZ Substring Compression Queries*,*

each in $\mathcal{O}\Big(F \cdot Q_{rsucc}(n) \log\log \frac{|x|}{F}\Big)$ time, where F is the number of phrases reported. The data structure can be constructed in $\mathcal{O}(n + C_{rsucc}(n))$ time.

Proof. Let $x = T[\ell_x \mathinner{.\,.} r_x]$ and suppose that we have already factorized $x' = T[\ell_x \mathinner{.\,.} m-1]$, i.e., the next phrase needs to be a prefix of $x'' = T[m \mathinner{.\,.} r_x]$. Depending on the factorization type (see Section 2.3.1 for definitions), it is chosen among the longest prefix of x'' that is a previous fragment of x (i.e., has an occurrence starting within $[\![\ell_x, m-1]\!]$), the longest prefix of x'' that is a non-overlapping previous fragment of x (i.e., occurs in x'), or the longest prefix of x'' that occurs in y. Clearly, the first case reduces to an Interval Longest Common Prefix Query, while the latter two—to Bounded Longest Common Prefix Queries. For each factorization type, we compute the relevant candidates and choose the longest one as the phrase; if there are no valid candidates, the next phrase is a single letter, i.e., $T[m \mathinner{.\,.} m]$.

Thus, regardless of the factorization type, we report each phrase f_i of the factorization $x = f_1 \cdots f_F$ in $\mathcal{O}(Q_{rsucc}(n) \log\log |f_i|)$ time. This way, the total running time is $\mathcal{O}\Big(\sum_{i=1}^{F} Q_{rsucc}(n) \log\log |f_i|\Big)$, which is $\mathcal{O}\Big(F \cdot Q_{rsucc}(n) \log\log \frac{|x|}{F}\Big)$ due to Jensen's equality applied to the concave $\log\log$ function. □

Note that in the case of ordinary LZ Substring Compression Queries the approach presented in Theorem 7.3.3 would result in $\mathcal{O}(F \cdot Q_{rsucc}(n))$ query time because only Interval Longest Common Prefix Queries would be used. In fact, this is exactly the algorithm for LZ Substring Compression Queries provided in [83].

Hence, despite our improvements, there is still an overhead for using variants of the LZ factorization other than the standard one. Nevertheless, the overhead disappears if we use the state-of-the-art $\mathcal{O}(n)$-size data structure for range successor queries. This is because the $\mathcal{O}(\log^{\varepsilon} n)$ time complexity lets us hide $\log^{o(1)} n$ factors by choosing a slightly different ε. Formally, Theorem 7.3.3 and Proposition 3.7.5 yield the following result:

Corollary 7.3.4. *For every text T of length n and constant $\varepsilon > 0$, there is a data structure of size $\mathcal{O}(n)$ that answers* Bounded Longest Common Prefix Queries *in $\mathcal{O}(\log^{\varepsilon} n)$ time and* LZ Substring Compression Queries *(for all five factorization types defined in Section 2.3.1) in $\mathcal{O}(\log^{\varepsilon} n)$ time per phrase reported. Moreover, the data structure can be constructed in $\mathcal{O}(n\sqrt{\log n})$ time.*

Chapter 8

Wavelet Suffix Trees

In this chapter, we introduce wavelet suffix trees, a data structure combining features of wavelet trees and suffix trees (see Sections 2.6 and 3.6, respectively). The wavelet suffix tree of a text T of length n is similar to the wavelet tree built for the suffix array of T (see Section 2.5). However, its shape resembles the suffix tree of T to the extent possible for a tree of height $\mathcal{O}(\log n)$. Wavelet suffix trees are equipped with additional components for text processing (including the data structure for Internal Pattern Matching Queries developed in Chapter 6) so that they become a powerful tool. With $\mathcal{O}(n)$ size and $\mathcal{O}(n\sqrt{\log n})$ construction time, they allow answering the following queries in $\mathcal{O}(\log n)$ time:

Substring Suffix Selection Queries
Given a fragment x of the text T and an integer k, find the kth lexicographically smallest suffix of x.

Substring Suffix Rank Queries
Given fragments x and y of the text T, find the lexicographic rank of y among the suffixes of x.

Moreover, wavelet suffix trees allow for efficient substring compression with respect to the BWT+RLE scheme defined in Section 2.3.2. The running time of the following queries is $\mathcal{O}(|\operatorname{RLE}(\operatorname{BWT}(x))|\log n)$ time, i.e., $\mathcal{O}(\log n)$ per run in the run-length encoding.

BWT+RLE Substring Compression Queries
Given a fragment x of the text T, compute the run-length encoding $\operatorname{RLE}(\operatorname{BWT}(x))$ of the Burrows–Wheeler transform of the underlying substring.

Section 8.1 provides a high-level description of the wavelet suffix trees. It forms an interface between the query algorithms (Section 8.4) and the more technical content: full description of the data structure (Section 8.2) and its construction algorithm (Section 8.3). Consequently, Sections 8.2 and 8.4 can be read separately. The latter additionally contains simple $\Omega(\log n/\log\log n)$ cell-probe lower bounds for Substring Suffix Selection Queries and Substring Suffix Rank Queries valid for all data structures of size $\mathcal{O}(n\log^{\mathcal{O}(1)} n)$. We conclude with Section 8.5, where we present a generic transformation of the data structure which allows replacing the dependence on n with a dependence on $|x|$ in the running times of the query algorithms.

8.1 Overview

The *wavelet suffix tree* of a text T of length n is a full binary tree of logarithmic height. Each of its $n+1$ leaves corresponds to a non-empty suffix of $T\#$. The lexicographic order of suffixes is preserved as the left-to-right order of leaves.

Each node ν of the wavelet suffix tree stores two bitmasks. Bits of the first bitmask correspond to suffixes in the subtree of ν sorted by their starting positions, and bits of the second bitmask correspond to these suffixes sorted first according to the preceding character and then according to the starting position. The ith bit of either bitmask is set to **0** if the ith suffix belongs to the left subtree of ν and set to **1** otherwise. Like in the standard wavelet trees, on top of the bitmasks we maintain a component for rank and selection queries (see Proposition 3.5.1). Figure 8.1 provides a sample wavelet suffix tree with both bitmasks depicted inside each node.

Each edge e of the wavelet suffix tree is associated with a sorted list $L(e)$ containing substrings of T. The wavelet suffix tree enjoys an important *lexicographic property*: Suppose that we traverse the tree depth-first, and when going *down* an edge e, we write out the contents of $L(e)$, whereas when visiting a leaf, we output the corresponding suffix of $T\#$. Then, we obtain the lexicographically sorted list of all non-empty substrings of

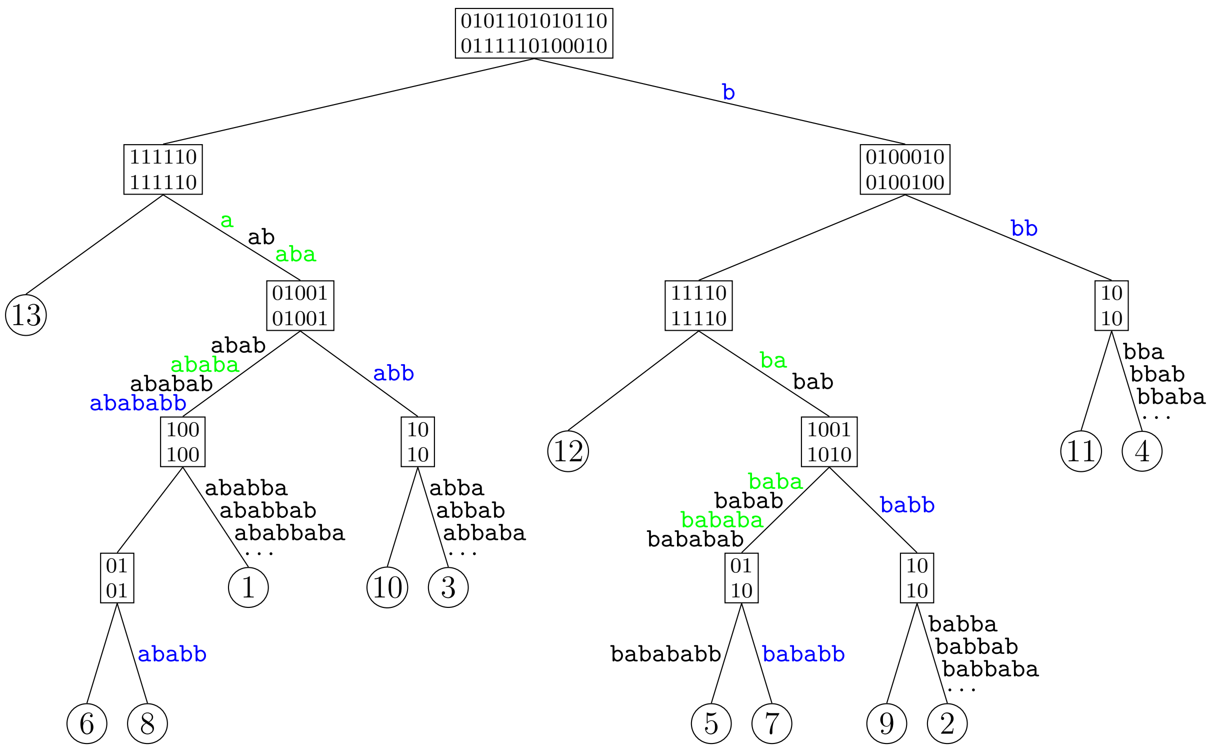


Figure 8.1: The wavelet suffix tree of a text $T =$ `ababbabababb`. Leaves corresponding to $T[i\,..]\#$ are labeled with i. Elements of the lists $L(e)$ are listed next to e, with ellipses denoting further substrings up to the suffix of T. Suffixes of $x =$ `bababa` are marked green, and of $x =$ `abababb`—blue. Note that the substrings occurring at position i do not need to lie above the leaf i (see $T[1\,..\,5] =$ `ababb`), and the substrings above the leaf i do not need to be prefixes of $T[i\,..]$ (see $T[10\,..]\# =$ `abb#` and `aba`).

$T\#$ (without repetitions).[1] This, in particular, implies that the substrings in $L(e)$ are consecutive prefixes of the longest substring in $L(e)$ and that for each substring y of T there is exactly one edge e such that the $y \in L(e)$.

In the query algorithms, we actually work with $L_x(e)$, containing the suffixes of x among the elements of $L(e)$. For each edge e, starting positions of these suffixes form $\mathcal{O}(1)$ non-overlapping periodic progressions (defined in Section 7.1), and consequently the list $L_x(e)$ admits a constant-space representation. Nevertheless, we do not store the lists explicitly, but instead generate some of them on the fly. This is one of the auxiliary operations, each of which is supported by the wavelet suffix tree in constant time.

(1) For a fragment $x = T[i \mathinner{.\,.} j]$ and an edge e, output the list $L_x(e)$ represented as $\mathcal{O}(1)$ non-overlapping periodic progressions (of the starting positions of the reported suffixes of x).
(2) Compute the number of suffixes of $x = T[i \mathinner{.\,.} j]$ in the left/right subtree of a node (given along with the segment of its first bitmask corresponding to suffixes of $T\#$ that start at a position in $[\![i, j]\!]$).
(3) Compute the number of suffixes of $x = T[i \mathinner{.\,.} j]$ that are preceded by a character c and lie in the left/right subtree of a node (given along with the segment of its second bitmask corresponding to the suffixes of $T\#$ that start at a position in $[\![i, j]\!]$ and are preceded by c).
(4) For a fragment x and an edge e, compute the run-length encoding of the sequence of characters preceding the suffixes in $L_x(e)$.

In order to implement these operations efficiently, we assume that the wavelet suffix tree contains two external components built for the text T. These are the data structure for Longest Common Extension Queries (see Proposition 2.5.1), and our data structure for Internal Pattern Matching Queries implemented in Chapter 6 (see Theorem 1.1.4).

8.2 Full Description of Wavelet Suffix Trees

We start the description with Section 8.2.1, where we introduce *string intervals*, a notion central to the definition of wavelet suffix trees. We also show that LCE Queries (see Proposition 2.5.1) let us efficiently deal with string intervals. Then, in Section 8.2.2, we give a precise definition of wavelet suffix trees and prove its several combinatorial consequences. We conclude with Section 8.2.3, where we implement the auxiliary operations listed in Section 8.1. These procedures heavily rely on the IPM Queries of Chapter 6.

8.2.1 String Intervals

To define wavelet suffix trees, we often need to compare substrings of $T\#$ restricted to a certain number of leading characters. If instead of x and y, we compare their counterparts trimmed to the first ℓ characters, i.e., $x[1 \mathinner{.\,.} \min(\ell, |x|)]$ and $y[1 \mathinner{.\,.} \min(\ell, |y|)]$, we use ℓ in the subscript of the operator, e.g., $x \prec_\ell y$ or $x \preceq_\ell y$.

For a pair of strings s, t and a positive integer ℓ, we define *string intervals* $[s, t]_\ell = \{z \in \bar{\Sigma}^* : s \preceq_\ell z \preceq_\ell t\}$ and $(s, t)_\ell = \{z \in \bar{\Sigma}^* : s \prec_\ell z \prec_\ell t\}$. Intervals $[s, t)_\ell$ and $(s, t]_\ell$ are defined analogously. The strings s and t are called the *endpoints* of these intervals.

[1] A similar property holds for suffix trees if we define $L(e)$ so that it contains the labels of all implicit nodes on e and the label of the lower explicit endpoint of e.

Recall that in Section 7.1 we introduced periodic progressions and, in Lemma 7.1.4, showed that LCE QUERIES can be applied for processing such progressions. In Section 8.2.3, we shall apply these results using the following lemma as an interface:

Lemma 8.2.1. *Suppose that we are given a text T equipped with a data structure answering* LCE QUERIES *in constant time. Given a periodic progression $p_0 < \cdots < p_{k-1}$ in T, a position $j \geq p_{k-1}$, and a string interval I whose endpoints are given as fragments of T, we can report in $\mathcal{O}(1)$ time, as a single periodic progression, all positions p_i such that $T[p_i \mathinner{..} j] \in I$.*

Proof. Let $u_i = T[p_i \mathinner{..} j]$. We shall compute the subinterval of $[\![0, k-1]\!]$ consisting of the indices i such that $u_i \in I$. The indices are consecutive because the sequence $(u_i)_{i=0}^{k-1}$ is monotone (as proved in Corollary 7.1.3).

Let s and t be the endpoints of I. First, we shall compute a subinterval of positions i such that $u_i \in (s, t)$. For this, we apply Lemma 7.1.4(b) twice, asking for $u_i \succ s$ and $u_i \prec t$, and we intersect the obtained intervals. Next, we find the indices i such that $\operatorname{lcp}(u_i, s) \geq \ell$ and the indices i such that $\operatorname{lcp}(u_i, t) \geq \ell$. For this, we apply Lemma 7.1.4(c) for $v = s[1 \mathinner{..} \max(|s|, \ell)]$ and $v = t[1 \mathinner{..} \max(|t|, \ell)]$, respectively. If the resulting lcp values are smaller than $|v|$, we replace the obtained interval with an empty one. Finally, depending on the type of the string interval I, we add or subtract the obtained interval from the subinterval of $[\![0, k-1]\!]$ corresponding to (s, t).

Once we know the indices i such that $u_i \in I$, we simply retrieve the periodic progression of the starting positions p_i of these fragments u_i. □

8.2.2 Definition of Wavelet Suffix Trees

Let T be a text of length n. To define the wavelet suffix tree of T, we start from an auxiliary tree $\mathcal{T}_{\mathsf{aux}}$ of height $\mathcal{O}(\log n)$ with $\mathcal{O}(n \log n)$ nodes. Its leaves represent non-empty suffixes of $T\#$, and the left-to-right order of leaves corresponds to the lexicographic order on the suffixes. Internal nodes of $\mathcal{T}_{\mathsf{aux}}$ represent all substrings of T whose length is a power of two, with an exception of the root, which represents the empty word. Edges in $\mathcal{T}_{\mathsf{aux}}$ are defined so that a node representing v is an ancestor of a node representing v' if and only if v is a prefix of v'. To each non-root node ν, we assign the *level* $\ell(\nu) := 2|v|$, where v is the substring that ν represents. For the root r, we set $\ell(r) := 1$; see Figure 8.2 for a sample tree $\mathcal{T}_{\mathsf{aux}}$ with levels assigned to nodes.

For a node ν, we define $\mathcal{S}(\nu)$ to be the set of suffixes of $T\#$ that are represented by ν or its descendants. Note that $\mathcal{S}(\nu)$ is a singleton if ν is a leaf. The following observation characterizes the levels and the sets $\mathcal{S}(\nu)$.

Observation 8.2.2. *For every node ν other than the root:*

(a) $\ell(parent(\nu)) \leq \ell(\nu)$,

(b) if $y \in \mathcal{S}(\nu)$ and y' is a suffix of $T\#$ such that $\operatorname{lcp}(y, y') \geq \ell(parent(\nu))$, then $y' \in \mathcal{S}(\nu)$,

(c) if $y, y' \in \mathcal{S}(\nu)$, then $\operatorname{lcp}(y, y') \geq \left\lfloor \frac{1}{2}\ell(\nu) \right\rfloor$.

Next, we modify $\mathcal{T}_{\mathsf{aux}}$ to obtain a binary tree of $\mathcal{O}(n)$ nodes. In order to reduce the number of nodes, we dissolve all internal nodes with exactly one child, i.e., while there is a non-root node ν with exactly one child ν', we set $parent(\nu') := parent(\nu)$ and remove ν. To make the tree binary, for each node ν with $k > 2$ children, we remove the edges

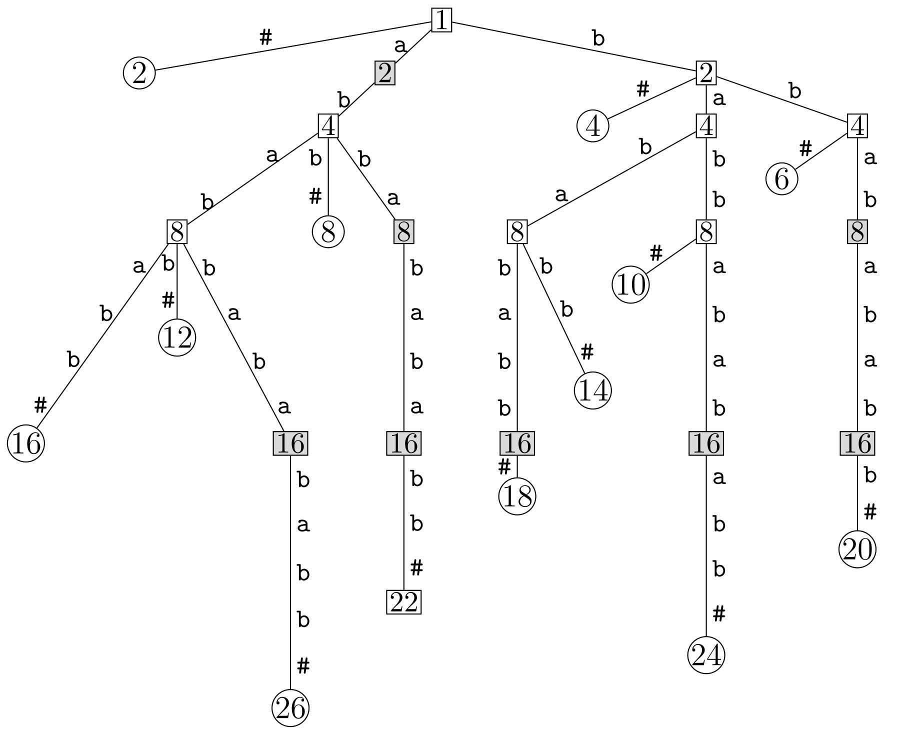


Figure 8.2: An auxiliary tree $\mathcal{T}_{\mathsf{aux}}$ introduced to define the wavelet suffix tree of $T =$ `ababbabababb`. Levels are written inside the nodes. The gray nodes are dissolved during the construction of the wavelet suffix tree.

between ν and its children, and instead we put a *replacement tree*: a full binary tree with k leaves whose root is ν, and whose leaves are the k children of ν (preserving the left-to-right order). We choose the replacement trees so that the resulting tree still has height $\mathcal{O}(\log n)$. In Section 8.3.1, we provide a constructive proof that such a choice is possible. This procedure introduces new nodes (inner nodes of the replacement trees); their levels are inherited from the parents.

The obtained tree is the wavelet suffix tree of T; see Figure 8.3 for an example. Observe that, as claimed in Section 8.1, it is a full binary tree of logarithmic height whose leaves correspond to non-empty suffixes of $T\#$. Moreover, it is easy to see that this tree still satisfies Observation 8.2.2.

As described in Section 8.1, each node ν (except for the leaves) stores two bitmasks. In either bitmask, each bit corresponds to a suffix $y \in \mathcal{S}(\nu)$, and it is equal to $\mathbf{0}$ if $y \in \mathcal{S}(lchild(\nu))$ and to $\mathbf{1}$ if $y \in \mathcal{S}(rchild(\nu))$, where $lchild(\nu)$ and $rchild(\nu)$ denote the children of ν. In the first bitmask, the suffixes $y = T[j\,..]\#$ are ordered by the starting position j, and in the second bitmask—by pairs $(T[j-1], j)$ (assuming $T[0] = \#$). Both bitmasks are equipped with data structures for rank and selection queries; see Proposition 3.5.1.

Additionally, each node and each edge of the wavelet suffix tree is associated with a string interval whose endpoints are suffixes of $T\#$. Namely, for an arbitrary node ν, we define $I(\nu) = [\min \mathcal{S}(\nu), \max \mathcal{S}(\nu)]_{\ell(\nu)}$. Additionally, if ν is not a leaf, we set $I(\nu, lchild(\nu)) = [\min \mathcal{S}(\nu), y]_{\ell(\nu)}$ and $I(\nu, rchild(\nu)) = (y, \max \mathcal{S}(\nu)]_{\ell(\nu)}$, where $y = \max \mathcal{S}(lchild(\nu))$ is the suffix corresponding to the rightmost leaf in the left subtree of ν; see also Figure 8.3. For each node ν, we store the starting positions of $\min \mathcal{S}(\nu)$ and $\max \mathcal{S}(\nu)$ in order to efficiently retrieve a representation of $I(\nu)$ and $I(e)$ for incident edges e. The following

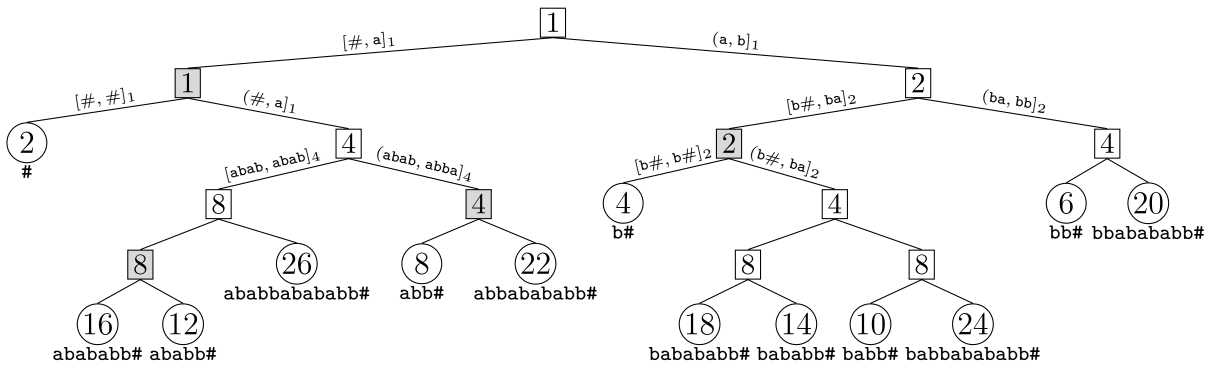


Figure 8.3: The wavelet suffix tree of a text T = ababbabababb (see also Figures 8.1 and 8.2). Levels are written inside the nodes. The gray nodes have been introduced as inner nodes of replacement trees. The corresponding suffix is written down below each leaf. Selected edges e are depicted with the corresponding intervals $I(e)$; their endpoints are trimmed for visual clarity.

lemma characterizes the intervals.

Lemma 8.2.3. *For every node ν of the wavelet suffix tree, we have:*

(a) If ν is not a leaf, then $I(\nu)$ is a disjoint union of $I(\nu, lchild(\nu))$ and $I(\nu, rchild(\nu))$.

(b) If y is a suffix of $T\#$, then $y \in I(\nu)$ if and only if $y \in \mathcal{S}(\nu)$.

(c) If ν is not the root, then $I(\nu) \subseteq I(parent(\nu), \nu)$.

Proof. (a) This claim is a trivial consequence of the definitions.

(b) Clearly, $y \in \mathcal{S}(\nu)$ if and only if $y \in [\min \mathcal{S}(\nu), \max \mathcal{S}(\nu)]$. Therefore, it suffices to show that if $\mathrm{lcp}(y, y') \geq \ell(\nu)$ for $y' = \min \mathcal{S}(\nu)$ or $y' = \max \mathcal{S}(\nu)$, then $y \in \mathcal{S}(\nu)$. This is, however, a consequence of points (a) and (b) of Observation 8.2.2.

(c) Let $\ell_p = \ell(parent(\nu))$. If $\nu = lchild(parent(\nu))$, then $\mathcal{S}(\nu) \subseteq \mathcal{S}(parent(\nu))$ and, by Observation 8.2.2(a), $\ell(\nu) \geq \ell_p$, which implies the statement.

Therefore, assume that $\nu = rchild(parent(\nu))$, and let ν' be the left sibling of ν. Note that $I(parent(\nu), \nu) = (\max \mathcal{S}(\nu'), \max \mathcal{S}(\nu)]_{\ell_p}$ and $I(\nu) \subseteq [\min \mathcal{S}(\nu), \max \mathcal{S}(\nu)]_{\ell_p}$, since $\ell(\nu) \geq \ell_p$ by Observation 8.2.2(a). Consequently, it suffices to prove that $\max \mathcal{S}(\nu') \prec_{\ell_p} \min \mathcal{S}(\nu)$. This is, however, a consequence of Observation 8.2.2(b) for $y = \min \mathcal{S}(\nu)$ and $y' = \max \mathcal{S}(\nu')$, and the fact that the left-to-right order of leaves coincides with the lexicographic order of the corresponding suffixes of $T\#$. □

For each edge $e = (parent(\nu), \nu)$ of the wavelet suffix tree, we define $L(e)$ to be the sorted list of those substrings of T which belong to $I(e) \setminus I(\nu)$.

Recall that the wavelet suffix tree shall enjoy the *lexicographic property*: if we traverse the tree, and when going down an edge e, we write out the contents of $L(e)$, whereas when visiting a leaf, we output the corresponding suffix of $T\#$, we shall obtain the lexicographically sorted list of all non-empty substrings of $T\#$. This is proved in the following series of claims.

Lemma 8.2.4. *Let $e = (parent(\nu), \nu)$ for a node ν. Substrings in $L(e)$ are smaller than any string in $I(\nu)$.*

Proof. We use a shorthand ℓ_p for $\ell(parent(\nu))$. Let $y = \max \mathcal{S}(\nu)$ be the rightmost suffix in the subtree of ν. Consider a substring $s \in L(e)$ and its occurrence $T[k \mathinner{.\,.} j] \cong s$; also let $t = T[k \mathinner{.\,.}]\#$.

We first prove that $s \preceq y$. Note that $I(e) = [x, y]_{\ell_p}$ or $I(e) = (x, y]_{\ell_p}$ for some string x. We have $s \in L(e) \subseteq I(e)$, and thus $s \preceq_{\ell_p} y$. If $\mathrm{lcp}(s, y) < \ell_p$, this already implies that $s \preceq y$. Thus, let us assume that $\mathrm{lcp}(s, y) \geq \ell_p$. The suffix t has s as a prefix, so this also means that $\mathrm{lcp}(t, y) \geq \ell_p$. By Observation 8.2.2(b), we conclude that $t \in \mathcal{S}(\nu)$, so $t \preceq y$. Therefore, $s \preceq t \preceq y$, as claimed.

Finally, note that $y \in \mathcal{S}(\nu) \subseteq I(\nu)$, $s \notin I(\nu)$, and $I(\nu)$ is a string interval. Consequently, $s \preceq y$ implies that s is lexicographically smaller than any string in $I(\nu)$. □

Lemma 8.2.5. *The wavelet suffix tree satisfies the lexicographic property.*

Proof. Note that for the root r we have $I(r) = [\#, c]_1$, where c is the largest character present in T. Thus, $I(r)$ contains all non-empty substrings of $T\#$, and it suffices to show that if we traverse the subtree of r, writing out the contents of $L(e)$ when going down an edge e and the corresponding suffix when visiting a leaf, we obtain a sorted list of substrings of $T\#$ contained in $I(r)$. We will prove that this property actually holds not just for r but for all nodes ν of the wavelet suffix tree.

This is clear if ν is a leaf since $I(\nu)$ consists of the corresponding suffix of $T\#$ only. Next, if we have already proved the hypothesis for ν, then prepending the output with the contents of $L(parent(\nu), \nu)$, by Lemmas 8.2.4 and 8.2.3(c), we obtain a sorted list of substrings of $T\#$ contained in $I(parent(\nu), \nu)$. Applying this property for both children of a non-leaf node ν', we conclude that if the hypothesis holds for children of ν' then, by Lemma 8.2.3(a), it also holds for ν'. □

Corollary 8.2.6. *For each edge e of the wavelet suffix tree, the list $L(e)$ contains consecutive prefixes of the largest element of $L(e)$.*

Proof. Note that if $x \prec y$ are substrings of T such that x is not a prefix of y, then x can be extended to a suffix x' of $T\#$ such that $x \prec x' \prec y$. However, $L(e)$ does not contain any suffix of $T\#$. By Lemma 8.2.5, $L(e)$ contains lexicographically consecutive substrings of $T\#$, so x and y cannot be both present in $L(e)$. Consequently, each element of $L(e)$ is a prefix of $\max L(e)$. Similarly, since $L(e)$ contains lexicographically consecutive substrings of $T\#$, it must comprise all prefixes of $\max L(e)$ no shorter than $\min L(e)$. □

8.2.3 Implementation of Auxiliary Queries

Recall that for a given fragment x and an edge e of the wavelet suffix tree, the list $L_x(e)$ consists of the suffixes of x present in $L(e)$. The wavelet suffix tree shall handle the following four types of queries, answering each in constant time:

(1) For a fragment $x = T[i \mathinner{.\,.} j]$ and an edge e, output the list $L_x(e)$ represented as $\mathcal{O}(1)$ non-overlapping periodic progressions (of the starting positions of the reported suffixes of x);

(2) Compute the number of suffixes of $x = T[i \mathinner{.\,.} j]$ in the left/right subtree of a node (given along with the segment of its first bitmask corresponding to suffixes of $T\#$ that start at a position in $[\![i, j]\!]$);

(3) Compute the number of suffixes $x = T[i \mathinner{.\,.} j]$ that are preceded by a character c and lie in the left/right subtree of a node (given along with the segment of its second

bitmask corresponding to the suffixes of $T\#$ that start at a position in $[\![i, j]\!]$ and are preceded by c);

(4) For a fragment x and an edge e, compute the run-length encoding of the sequence of characters preceding the suffixes in $L_x(e)$.

We start with an auxiliary lemma applied in the solutions to all four query types.

Lemma 8.2.7. *Let ν be a node of the wavelet suffix tree of a text T. The following operations can be implemented in constant time.*

(1) Given a fragment x of T, $|x| < \ell(\nu)$, return, as a single periodic progression of starting positions, all suffixes s of x such that $s \in I(\nu)$.

(2) Given a range of positions $[\![i, j]\!]$, $j-i < \ell(\nu)$, return all positions $k \in [\![i, j]\!]$ such that $T[k\,..]\# \in I(\nu)$, represented as at most two non-overlapping periodic progressions.

Proof. Let p be the longest common prefix of all strings in $I(\nu)$; by Observation 8.2.2(c), we have $|p| \geq \lfloor \frac{1}{2}\ell(\nu)\rfloor$. Moreover, we can use $\text{lcp}(\min \mathcal{S}(\nu), \max \mathcal{S}(\nu))$ to determine in $\mathcal{O}(1)$ time a fragment of T matching p.

(1) Assume $x = T[i\,..\,j]$. We make IPM Queries (Theorem 1.1.4) to find all occurrences of p within x. By Observation 7.1.1, their starting positions can be represented as a single periodic progression due to $|x| \leq \ell(\nu) - 1 \leq 2\lfloor \frac{1}{2}\ell(\nu)\rfloor \leq 2|p|$. Then, using Lemma 8.2.1, we filter positions k for which $T[k\,..\,j] \in I(\nu)$.

(2) Let $x = T[i\,..\,j+|p|-1]$ (or $x = T[i\,..]\#$ if $j+|p|-1 > |T|$). We make IPM Queries to find all occurrences of p within x. By Observation 7.1.1, their starting positions can represented as two non-overlapping periodic progression due to $|x| \leq \ell(\nu) + |p| - 1 \leq 2\lfloor \frac{1}{2}\ell(\nu)\rfloor + |p| \leq 3|p|$. Like previously, using Lemma 8.2.1, we filter positions k for which $T[k\,..]\# \in I(\nu)$. □

We are now ready to implement the auxiliary queries.

Lemma 8.2.8. *The wavelet suffix tree allows answering queries (1) in constant time. In more details, for any edge $e = (parent(\nu), \nu)$ and fragment x of T, the starting positions of suffixes in $L_x(e)$ form at most three non-overlapping periodic progressions which can be reported in $\mathcal{O}(1)$ time.*

Proof. First, we consider short suffixes. We use Lemma 8.2.7(1) to find all suffixes s of x of length $|s| < \ell(parent(\nu))$ such that $s \in I(parent(\nu))$. Lemma 8.2.3 yields $L(e) \subseteq I(parent(\nu))$, so we apply Lemma 8.2.1 to filter all suffixes belonging to $L(e)$, i.e., to $I(e) \setminus I(\nu)$. By Lemma 8.2.4, we obtain at most one periodic progression.

Now, it suffices to generate suffixes s of length $|s| \geq \ell(parent(\nu))$ that belong to $L(e)$. Suppose that $s = T[k\,..\,j]$. If $s \in I(e)$, then equivalently $T[k\,..]\# \in I(e)$, since s is a long enough prefix of $T[k\,..]\#$ to determine whether the latter belongs to $I(e)$. Consequently, by Lemma 8.2.3, $T[k\,..]\# \in I(\nu)$. This implies $|s| < \ell(\nu)$ (otherwise we would have $s \in I(\nu)$), i.e., $k \in [\![j - \ell(\nu) + 2, j - \ell(parent(\nu)) + 1]\!]$. We apply Lemma 8.2.7(2) to compute all positions k in this range for which $T[k\,..]\# \in I(\nu)$. Then, using Lemma 8.2.1, we filter positions k such that $T[k\,..\,j] \in I(e) \setminus I(\nu)$. By Lemma 8.2.4, this cannot increase the number of periodic progressions, so we end up with at most two non-overlapping periodic progressions from the case of $|s| \geq \ell(parent(\nu))$ and at most three in total. □

Lemma 8.2.9. *The wavelet suffix tree allows answering queries (2) in constant time.*

Proof. Let ν be the given node and ν' be its right/left child (depending on the variant of the query). First, we use Lemma 8.2.7(1) to find all suffixes s of x of length $|s| < \ell(\nu)$ such that $s \in I(\nu)$. Then, we apply Lemma 8.2.3 to filter suffixes $s \in I(\nu, \nu')$, i.e., such that s lies in the appropriate subtree of ν. We add the number of these suffixes to the result.

Thus, it remains to count suffixes of length at least $\ell(\nu)$. Suppose that $s = T[k\,..\,j]$ is a suffix of x such that $|s| \geq \ell(\nu)$ and $s \in I(\nu, \nu')$. Then $T[k\,..]\# \in I(\nu, \nu')$, and the number of suffixes $T[k\,..]\# \in I(\nu, \nu')$ such that $k \in [\![i, j]\!]$ is simply the number of **1**'s or **0**'s in the given segment of the first bitmask in ν, which we can compute in constant time using a rank query. Observe, however, that we have also counted positions k such that $|T[k\,..\,j]| < \ell(\nu)$ and $T[k\,..]\# \in I(\nu, \nu')$; we need to subtract the number of these positions. For this, we use Lemma 8.2.7(2) to retrieve the positions $k \in [\![j - \ell(\nu) + 2, j]\!]$ such that $T[k\,..]\# \in I(\nu)$ and Lemma 8.2.3 to filter those with $T[k\,..]\# \in I(\nu, \nu')$. We determine the total size of the obtained periodic progressions and subtract it from the final result. □

Lemma 8.2.10. *The wavelet suffix tree allows answering queries (3) and (4) in constant time.*

Proof. Observe that every periodic progression $p_0, \ldots, p_{k-1}$ satisfies $T[p_1 - 1] = \ldots = T[p_{k-1} - 1]$. While $T[p_0 - 1]$ may be a different character, it is still straightforward to determine in $\mathcal{O}(1)$ time which positions of such a progression are preceded by a given character c.

Answering queries (3) is analogous to answering queries (2); we just use the second bitmask at the given node and consider only positions preceded by c instead of counting the sizes of the whole periodic progressions.

To answer queries (4), we first retrieve $L_x(e)$ using Lemma 8.2.8. By Corollary 8.2.6, the suffixes in $L_x(e)$ are prefixes of one another, so the lexicographic order on these suffixes coincides with the order of ascending lengths. Consequently, the run-length encoding of the piece corresponding to $L_x(e)$ has at most six phrases (runs) and can be easily constructed in $\mathcal{O}(1)$ time from the at most three periodic progressions representing $L_x(e)$. □

8.3 Construction of Wavelet Suffix Trees

The actual construction algorithm is presented in Section 8.3.2. Before, in Section 8.3.1, we introduce several auxiliary tools for abstract weighted trees.

8.3.1 Toolbox for Weighted Trees

Let $\mathcal{T}$ be a rooted ordered tree with positive integer weights on edges, n leaves, and no inner nodes of degree one. We say that $L_1, \ldots, L_{n-1}$ is an *LCA sequence* of $\mathcal{T}$, if L_i is the (weighted) depth of the lowest common ancestor of the ith and $(i+1)$th leaves. The following fact is usually applied to construct the suffix tree of a string from the suffix array and the LCP table; see e.g. [48].

Fact 8.3.1. *Given a sequence $(L_i)_{i=1}^{n-1}$ of non-negative integers, one can construct in $\mathcal{O}(n)$ time a tree whose LCA sequence is $(L_i)_{i=1}^{n-1}$.*

The LCA sequence suffices to detect if a tree is binary.

Observation 8.3.2. *A rooted tree is a binary tree if and only if its LCA sequence $(L_i)_{i=1}^{n-1}$ satisfies the following property for every $i < j$: if $L_i = L_j$, then there exists k, $i < k < j$, such that $L_k < L_i$.*

The trees constructed by the following lemma can be seen as a variant of the weight-balanced trees, whose existence for arbitrary weights was by proved Blum and Mehlhorn [25].

Lemma 8.3.3. *Given a sequence $w_1, \ldots, w_n$ of positive integers, one can construct in $\mathcal{O}(n)$ time a binary tree $\mathcal{T}$ with n leaves such that the depth of the ith leaf is $\mathcal{O}(1 + \log \frac{W}{w_i})$, where $W = \sum_{j=1}^{n} w_j$.*

Proof. Define $W_i = \sum_{j=1}^{i} w_j$ for $i \in [\![0, n]\!]$. Let p_i be the 0-based position of the most significant bit where the binary representations of W_{i-1} and W_i differ, and let $P = \max_{i=1}^{n} p_i$. Observe that $P = \lfloor \log W \rfloor$ and $p_i \geq \lfloor \log w_i \rfloor$. Using Fact 8.3.1, we construct a tree $\mathcal{T}$ with $n+1$ leaves whose LCA sequence is $L_i = P - p_i$ for $i \in [\![1, n]\!]$. Note that this sequence satisfies the condition of Observation 8.3.2, and thus the tree $\mathcal{T}$ is binary.

Next, we insert an extra leaf between the two children of every node to make the tree ternary. The ith of these leaves is inserted at (weighted) depth $1 + L_i = \mathcal{O}(1 + \log \frac{W}{w_i})$, which is also an upper bound for its unweighted depth. Next, we remove the original leaves. This way, we get a tree satisfying the lemma, except for the fact that inner nodes may have between one and three children rather than exactly two.

In order to resolve this issue, we remove (dissolve) all inner nodes with exactly one child, and for each node ν with three children μ_1, μ_2, μ_3, we introduce a new node ν', setting μ_1, μ_2 as the children of ν' and ν', μ_3 as the children of ν. This way, we get a full binary tree, and the depth of every node may increase at most twice. In particular, the depth of the ith leaf stays within $\mathcal{O}(1 + \log \frac{W}{w_i})$. □

Let $\mathcal{T}$ be an ordered rooted tree and let ν be a node of $\mathcal{T}$ which is neither the root nor a leaf. Also, let μ be the parent of ν. We say that $\mathcal{T}'$ is obtained from $\mathcal{T}$ by *contracting* the edge (μ, ν) if ν is removed and the children of ν replace ν at its original location in the list of the children of μ. If $\mathcal{T}'$ is obtained from $\mathcal{T}$ by a sequence of edge contractions, we say that $\mathcal{T}'$ is a *contraction* of $\mathcal{T}$. Note that contraction alters neither the pre-order nor the post-order of the preserved nodes, which implies that the ancestor-descendant relation also remains unchanged for these nodes.

Corollary 8.3.4. *Let $\mathcal{T}$ be an ordered rooted tree of height h, which has n leaves and no inner node with exactly one child. Then, in $\mathcal{O}(n)$ time one can construct a full binary ordered rooted tree $\mathcal{T}'$ of height $\mathcal{O}(h + \log n)$ such that $\mathcal{T}$ is a contraction of $\mathcal{T}'$ and $\mathcal{T}'$ has $\mathcal{O}(n)$ nodes.*

Proof. For each node ν of $\mathcal{T}$ with three or more children, we replace the star-shaped tree joining it with its children $\mu_1, \ldots, \mu_k$ with an appropriate replacement tree: Let $W(\nu)$ be the number of leaves in the subtree of ν, and let $W(\mu_i)$ be the number of leaves in the subtrees of μ_i for $1 \leq i \leq k$. We use Lemma 8.3.3 for $w_i = W(\mu_i)$ to construct the replacement tree. Consequently, a node ν with depth d in $\mathcal{T}$ has depth $\mathcal{O}(d + \log \frac{n}{W(\nu)})$ in $\mathcal{T}'$, as one can prove with an easy top-down induction. In particular, the resulting tree has the claimed height $\mathcal{O}(h + \log n)$. □

8.3.2 Construction Algorithm

In this section, we show how to construct the wavelet suffix tree of a text T of length n in $\mathcal{O}(n\sqrt{\log n})$ time. The construction algorithm has two phases: first, it builds the *shape* of the wavelet suffix tree following the description in Section 8.2.2, and then it uses the results of Section 3.6 to obtain the bitmasks. Prior to that, we construct the component of Proposition 2.5.1 for LCE QUERIES in $T\#$, which includes the suffix array and the LCP table for $T\#$. We also build the data structure of Theorem 1.1.4 for IPM QUERIES.

Recall that in the definition of the wavelet suffix tree, we have started with a tree of size $\mathcal{O}(n \log n)$. We cannot afford that in an $o(n \log n)$-time construction. Thus, we construct the tree $\mathcal{T}_{\mathsf{aux}}$ already without the inner nodes having exactly one child. Observe that this tree is closely related to the suffix tree $\mathcal{T}_{\mathsf{suf}}(T\#)$. The only difference is that if the longest common prefix of two consecutive suffixes is d, their root-to-leaf paths diverge at weighted depth $\lceil \log(d+1) \rceil$ instead of d. Guided by this property, we use Fact 8.3.1 for $L_i = \lceil \log(LCP[i]+1) \rceil$ (rather than $L_i = LCP[i]$ which we would use for the suffix tree). This way, an inner node ν at weighted depth j represents a substring of length 2^{j-1}. The level $\ell(\nu)$ of such an inner node ν is set to 2^j. If ν is the leaf representing a suffix s of $T\#$, we set $\ell(\nu) = 2|s|$. After this operation, the tree $\mathcal{T}_{\mathsf{aux}}$ may have inner nodes of large degree, so we use Corollary 8.3.4 to obtain a binary tree $\mathcal{T}'_{\mathsf{aux}}$ such that $\mathcal{T}_{\mathsf{aux}}$ is its contraction. We set this binary tree as the shape of the wavelet suffix tree. Since $\mathcal{T}_{\mathsf{aux}}$ has height $\mathcal{O}(\log n)$, so does $\mathcal{T}'_{\mathsf{aux}}$.

To construct the bitmasks, we apply Theorem 3.6.2 for the shape $\mathcal{T}'_{\mathsf{aux}}$, with the leaf representing $T[i\..]\#$ assigned to i. The sequence s for the first bitmask satisfies $s[i] = i$ for each position i. For the second bitmask, we sort all positions i with respect to $(T[i-1], i)$ and take the resulting sequence of the second coordinates as s.

This way, we complete the proof of the main theorem concerning wavelet suffix trees.

Theorem 8.3.5. *The wavelet suffix tree of a text T of length n occupies $\mathcal{O}(n)$ space and can be constructed in $\mathcal{O}(n\sqrt{\log n})$ time.*

8.4 Applications

In this section, we show how to implement SUBSTRING SUFFIX RANK QUERIES, SUBSTRING SUFFIX SELECTION QUERIES, and BWT+RLE SUBSTRING COMPRESSION QUERIES using the wavelet suffix tree of the text T.

8.4.1 Substring Suffix Rank and Selection

Recall that in the substring suffix rank problem, we are asked to find the lexicographic rank of a fragment y among the suffixes of another fragment x. The substring suffix selection problem, in contrast, is to find the kth lexicographically smallest suffix of x for a given an integer k and a fragment x of T.

Theorem 8.4.1. *The wavelet suffix tree can answer* SUBSTRING SUFFIX RANK QUERIES *in $\mathcal{O}(\log n)$ time.*

Proof. Using a binary search on the leaves of the wavelet suffix tree of T, we locate the lexicographically smallest suffix t of $T\#$ such that $t \succ y$. Let π denote the path from the root to the leaf corresponding to t. Due to the lexicographic property, the rank of y

among the suffixes of x is equal to the sum of two values. The first one is the number of suffixes of x in the left subtrees hanging from the path π, whereas the second summand is the number of suffixes not exceeding y in the lists $L_x(e)$ for $e \in \pi$.

To compute those two numbers, we traverse π maintaining a segment $[\![\ell, r]\!]$ of the first bitmask corresponding to the suffixes of $T\#$ starting within x. When we descend to the left child, we set $[\![\ell, r]\!] := [\![\mathrm{rank}_0(\ell-1)+1, \mathrm{rank}_0(r)]\!]$, while for the right child, we set $[\![\ell, r]\!] := [\![\mathrm{rank}_1(\ell-1)+1, \mathrm{rank}_1(r)]\!]$. In the latter case, we pass $[\![\ell, r]\!]$ to type (2) queries, which let us count the suffixes of x in the left subtree hanging from π in the current node. This way, we compute the first summand.

For the second number, we use type (1) queries to generate all lists $L_x(e)$ for $e \in \pi$. For each list, we use Lemma 7.1.4(b) to determine the number of elements not exceeding y, and we add this value to the final result.

The described procedure takes $\mathcal{O}(\log n)$ time since type (1) and (2) queries, as well as LCE Queries, are implemented in $\mathcal{O}(1)$ time. □

Theorem 8.4.2. *The wavelet suffix tree can answer* Substring Suffix Selection Queries *in $\mathcal{O}(\log n)$ time.*

Proof. The algorithm traverses a path in the wavelet suffix tree of T. It maintains a segment $[\![\ell, r]\!]$ of the first bitmask corresponding to the suffixes of T starting within $x = T[i\mathinner{.\,.} j]$, and a variable k' counting the suffixes of x represented in the left subtrees hanging from the path or on the edges of the path. The algorithm starts at the root initializing $[\![\ell, r]\!]$ with $[\![i, j]\!]$ and k' with 0.

At each node ν, it first decides to which child of ν to proceed. For this, it performs a type (2) query to determine k'', the number of suffixes of x in the left subtree of ν. If $k' + k'' \geq k$, it chooses to go to the left child, otherwise to the right one; in the latter case, it also updates $k' := k' + k''$. The algorithm additionally updates the segment $[\![\ell, r]\!]$ using the rank queries on the first bitmask at ν.

Let ν' be the child of ν that the algorithm has chosen to proceed to. Before reaching ν', the algorithm performs a type (1) query to compute $L_x(\nu, \nu')$. If k' summed with the size of this list is at least k, then the algorithm terminates, returning the $(k-k')$th element of the list (which is easy to retrieve from the representation as $\mathcal{O}(1)$ periodic progressions). Otherwise, it sets $k' := k' + |L_x(\nu, \nu')|$ so that k' satisfies the definition for the newly extended path from the root to ν'.

The correctness of the algorithm follows from the lexicographic property, which implies that at the beginning of each step, the sought suffix of x is the $(k-k')$th smallest suffix of x represented in the subtree of ν. In particular, the procedure always terminates before reaching a leaf. The running time of the algorithm is $\mathcal{O}(\log n)$ due to $\mathcal{O}(1)$-time implementations of type (1) and (2) queries. □

A simple reduction proves that the query time in Theorems 8.4.1 and 8.4.2 is nearly optimal: in the word RAM model with machine words of size $W = \Theta(\log n)$, any data structure of size $\mathcal{O}(n \log^{\mathcal{O}(1)} n)$ must have query time $\Omega(\frac{\log n}{\log\log n})$.

Proposition 8.4.3. *In the cell-probe model with W-bit cells, a static data structure of size $c \cdot n$ must take $\Omega(\frac{\log n}{\log c + \log W})$ time both for* Substring Suffix Rank Queries *and* Substring Suffix Selection Queries.

Proof. We shall prove that Substring Suffix Rank Queries and Substring Suffix Selection Queries are at least as difficult as range rank and selection queries (defined in Section 3.7), respectively. Propositions 3.7.2 and 3.7.4 show that these range queries require $\Omega(\frac{\log n}{\log c+\log W})$ time already if the input array A is a permutation of $[\![1,n]\!]$. Such an array A can be interpreted as a text T over $\Sigma = [\![1,n]\!]$ (with $T[i] = A[i]$). Since A is a permutation, the order of non-empty fragments of T is specified by their first characters and (as a secondary criterion) by their lengths. In particular, for a range $R = [\![\ell,r]\!] \subseteq [\![1,n]\!]$ and an integer $k \in [\![1,|R|]\!]$, if $A[j] = \mathrm{select}_{A[R]}(k)$, then $T[j\,..\,r]$ is the kth lexicographically smallest suffix of $T[\ell\,..\,r]$. Similarly, the $\mathrm{rank}_{A[R]}(c)$ of a value $c \in [\![1,n]\!]$ occurring in T at some position i equals the rank of $T[i\,..\,i]$ among the suffixes of $T[\ell\,..\,r]$ or one plus the rank of $T[i\,..\,i]$ among the suffixes of $T[\ell\,..\,r]$ (the latter holds if and only if $\ell \le i < r$). Consequently, in order to support range rank and selection on a permutation A, it suffices to make a single Substring Suffix Rank Query and Substring Suffix Selection Query, respectively. In the former case, we also need to store the inverse permutation A^{-1}, which takes negligible $\mathcal{O}(n)$ space. □

8.4.2 BWT+RLE Substring Compression

Wavelet suffix trees can also be used to compute the run-length encoding of the Burrows–Wheeler transform of a substring; see Section 2.3.2 for a definition. Consider a fragment $x = T[i\,..\,j]$ and, for $1 \le k \le |x|$, let $T[i_k\,..\,j]$ be the kth lexicographically smallest suffix of x. Observe that $\mathrm{BWT}(x)$ is a string $b_1 b_2 \cdots b_{|x|+1}$ where $b_1 = T[j]$ and, for $2 \le k \le |x|+1$, we have $b_k = T[i_{k-1}-1]$ if $i_{k-1} > i$ and $b_k = \#$ if $i_{k-1} = i$.

Our algorithm initially generates a string almost equal to $\mathrm{BWT}(x)$ which instead of # contains $T[i-1]$. However, we know that # should occur at the position equal to one plus the rank of x among all the suffixes of x. Consequently, a single Substring Suffix Rank Query suffices to find the position which needs to be corrected.

Recall that the wavelet suffix tree satisfies the lexicographic property. Consequently, if we traverse the tree and write out the characters preceding the suffixes in the lists $L_x(e)$, we obtain $\mathrm{BWT}(x)$ (without the first symbol b_0). Our algorithm simulates such a traversal. Assume that the last character appended to $\mathrm{BWT}(x)$ is c, and the algorithm is to move down an edge $e = (\nu, \nu')$. Before deciding to do so, it checks whether all the suffixes of x in the appropriate (left or right) subtree of ν are preceded with c. For this, it performs type (2) and (3) queries, and if both results are equal to the same value q, it simply appends c^q to $\mathrm{BWT}(x)$ and decides not to proceed to ν'. In order to make these queries possible, for each node on the path from the root to ν, the algorithm maintains segments corresponding to $[\![i,j]\!]$ in the first bitmasks, and to $(c, [\![i,j]\!])$ in the second bitmasks. These segments are updated using rank queries on the bitmasks while moving down the tree.

Before the algorithm continues at ν', if it decides to do so, suffixes in $L_x(e)$ need to be handled. We perform a type (4) query to compute the characters preceding these suffixes, and append the result to $\mathrm{BWT}(x)$. This, however, may result in c no longer being the last symbol appended to $\mathrm{BWT}(x)$. If so, the algorithm updates the segments of the second bitmask for all nodes on the path from the root to ν'. We assume that the root stores all positions i sorted by $(T[i-1], i)$, which lets us binary search for both endpoints of the segment at the root. For the subsequent nodes on the path, the rank queries on the second bitmasks are applied. Overall, this update takes $\mathcal{O}(\log n)$ time and it is necessary at most once per run of $\mathrm{BWT}(x)$.

Now, let us estimate the number of edges visited. Observe that if we go down an edge, then the last character of $\mathrm{BWT}(x)$ changes at least once before we go up this edge. Thus, all the edges traversed down between such character changes form a path. The length of any path is $\mathcal{O}(\log n)$, and consequently the total number of visited edges is $\mathcal{O}(r \log n)$, where r is the number of runs.

Theorem 8.4.4. *The wavelet suffix tree can compute the run-length encoding of the Burrows–Wheeler transform of a fragment x in $\mathcal{O}(|\operatorname{RLE}(\operatorname{BWT}(x))| \log n)$ time, where $|\operatorname{RLE}(\operatorname{BWT}(x))|$ is the number of runs in the Burrows–Wheeler transform.*

8.5 Speeding up Queries

Finally, we note that building wavelet suffix trees for several fragments of T, we can make the query time adaptive to the length of the query fragment x. In other words, we can replace the $\mathcal{O}(\log n)$ factor in the running times by $\mathcal{O}(\log |x|)$.

Theorem 8.5.1. *Using a data structure of size $\mathcal{O}(n)$, which can be constructed in $\mathcal{O}(n\sqrt{\log n})$ time,* Substring Suffix Rank Queries *and* Substring Suffix Selection Queries *can be answered in $\mathcal{O}(\log |x|)$ time. The run-length encoding $\operatorname{RLE}(\operatorname{BWT}(x))$ of the BWT of a fragment x can be found in $\mathcal{O}(|\operatorname{RLE}(\operatorname{BWT}(x))| \log |x|)$ time.*

Proof. We build wavelet suffix trees for some fragments of each length $n_k = \lfloor n^{2^{-k}} \rfloor$ with $k \in [\![0, \lfloor \log \log n \rfloor]\!]$. For such length n_k, we choose every $\lfloor \frac{1}{2} n_k \rfloor$th fragment, starting from the prefix and, additionally, we choose the suffix. Auxiliary data structures of Proposition 2.5.1 and 1.1.4 are built for T only.

We have $n_k = \left\lfloor \sqrt{n_{k-1}} \right\rfloor$, so $n_{k-1} \le (n_k + 1)^2$ and thus every fragment x of T lies within a fragment v, $|v| \le 2(|x| + 1)^2$, for which we store the wavelet suffix tree. For each m, $1 \le m \le n$, we store the length n_k such that $2m \le n_k \le 2(m + 1)^2$. This reduces finding an appropriate fragment v to simple arithmetics. Using the wavelet suffix tree for v instead of the tree for the whole text T gives the announced query times. The only thing we must be careful about is that the input for the Substring Suffix Rank Queries also consists of a fragment y, which does not need to be contained in v. However, looking at the query algorithm, it is easy to see that y is only accessed through the LCE Queries of Proposition 2.5.1.

It remains to analyze the space usage and construction time. Observe that the wavelet suffix tree of a fragment v is simply a binary tree with two bitmasks at each node and with some pointers to the positions of the string T. In particular, it does not contain any characters of T and, if all pointers are stored as relative values, it can be stored using $\mathcal{O}(|v| \log |v|)$ bits, i.e., $\mathcal{O}(|v| \frac{\log |v|}{\log n})$ machine words. For each n_k, the total length of selected fragments is $\mathcal{O}(n)$, and thus the space usage is $\mathcal{O}(n \frac{\log n_k}{\log n}) = \mathcal{O}(n 2^{-k})$ machine words, which sums up to $\mathcal{O}(n)$ across all lengths n_k. The construction time is $\mathcal{O}(|v| \sqrt{\log |v|})$ for every fragment v (including alphabet renumbering), and this sums up to $\mathcal{O}(n \sqrt{2^{-k} \log n})$ for each length n_k, which is $\mathcal{O}(n \sqrt{\log n})$ in total. $\square$

Chapter 9

Minimal Suffix and Rotation Queries

The main result of this chapter is an optimal data structure for MINIMAL SUFFIX QUERIES, formally defined as follows:

> MINIMAL SUFFIX QUERIES
> Given a fragment v of the text T, report the lexicographically smallest non-empty suffix of v.

In Section 9.1, we study combinatorics of minimal suffixes. Our main tool there is the notion of *significant suffixes*, introduced by I et al. [76] to compute Lyndon factorizations of grammar-compressed strings. Section 9.2 is devoted to an efficient computation of the set of significant suffixes of a given fragment of T. Next, in Section 9.3, we develop our data structure for MINIMAL SUFFIX QUERIES. We use *fusion trees* by Pătraşcu and Thorup [126] to improve the query time from logarithmic to $\mathcal{O}(\log^* |v|)$, and then, by preprocessing shorts strings, we achieve constant query time. That final step uses a notion of *order-isomorphism* [96, 84] to reduce the number of precomputed values. Next, in Section 9.4, we repeat the same steps for the following more general queries, answered in $\mathcal{O}(k^2)$ time.

> GENERALIZED MINIMAL SUFFIX QUERIES
> Given a sequence of fragments $v_1, \ldots, v_k$ of the text T, report the lexicographically smallest non-empty suffix of their concatenation $v_1 v_2 \cdots v_k$ (represented by its length).

Our main motivation for GENERALIZED MINIMAL SUFFIX QUERIES is to efficiently handle MINIMAL ROTATION QUERIES, defined below. The simple reduction is given in Section 9.5 along with a brief discussion of other applications.

> MINIMAL ROTATION QUERIES
> Given a fragment v of the text T, report the lexicographically smallest rotation of v (represented by the number of positions to shift).

9.1 Combinatorics of Minimal and Maximal Suffixes

For a non-empty string v, the *minimal suffix* $\mathrm{MinSuf}(v)$ is the lexicographically smallest non-empty suffix s of v. Similarly, for an arbitrary string v the *maximal suffix* $\mathrm{MaxSuf}(v)$ is the lexicographically largest suffix s of v. We extend these notions as follows: for a pair of strings v, w we define $\mathrm{MinSuf}(v, w)$ and $\mathrm{MaxSuf}(v, w)$ as the lexicographically smallest (resp. largest) string sw such that s is a (possibly empty) suffix of v.

In order to relate minimal and maximal suffixes, we introduce the *reverse* order $\prec^R$ on the extended alphabet $\bar{\Sigma} = \Sigma \cup \{\$, \#\}$ (defined in Section 2.1), and we extend it to the *reverse lexicographic* order on $\bar{\Sigma}^*$. Observe that we have $\$ \prec^R c \prec^R \#$ for every $c \in \Sigma$. This lets us relate $\prec$ and $\prec^R$.

Observation 9.1.1. *The following conditions are equivalent for $u, v \in \Sigma^*$:*

(a) $u \prec^R v$,

(b) $u\$ \prec^R v\$$,

(c) $u\$ \succ v\$$.

We use MinSuf^R and MaxSuf^R to denote the minimal and the maximal suffixes with respect to the reverse order $\prec^R$.

Example 9.1.2. Consider a string $v = \mathtt{abaabaa}$. We have $\mathrm{MaxSuf}(\mathtt{abaabaa}) = \mathtt{baabaa}$, $\mathrm{MinSuf}(\mathtt{abaabaa}) = \mathtt{a}$, $\mathrm{MaxSuf}^R(\mathtt{abaabaa}) = \mathtt{aabaa}$, and $\mathrm{MinSuf}^R(\mathtt{abaabaa}) = \mathtt{baa}$.

The following observation relates the notions we introduced:

Observation 9.1.3. *(a)* $\mathrm{MaxSuf}(v, \varepsilon) = \mathrm{MaxSuf}(v)$ *for every* $v \in \bar{\Sigma}^*$,

(b) $\mathrm{MinSuf}(vw) = \min(\mathrm{MinSuf}(v, w), \mathrm{MinSuf}(w))$ *for every* $v \in \bar{\Sigma}^*$ *and* $w \in \bar{\Sigma}^+$,

(c) $\mathrm{MinSuf}(vc) = \mathrm{MinSuf}(v, c)$ *for every* $v \in \bar{\Sigma}^*$ *and* $c \in \bar{\Sigma}$,

(d) $\mathrm{MinSuf}(v, w\$) = \mathrm{MaxSuf}^R(v, w)\$$ *for every* $v, w \in \Sigma^*$,

(e) $\mathrm{MinSuf}(v\$) = \mathrm{MaxSuf}^R(v)\$$ *for every* $v \in \Sigma^*$.

Remark 9.1.4. A property seemingly similar to (e) is false: for every $v \in \Sigma^+$, we have $\$ = \mathrm{MinSuf}^R(v\$) \neq \mathrm{MaxSuf}(v)\$$.

Recall that Lyndon words have been defined in Section 2.1 as primitive strings which are minimal in their conjugacy class. Equivalently, a string w is a Lyndon word if and only $w = \mathrm{MinSuf}(w)$; see [103]. Note that a Lyndon word w does not have proper borders since a border would be a non-empty suffix smaller than w. A *Lyndon factorization* of a string $u \in \Sigma^*$ is a representation $u = u_1^{p_1} \dots u_m^{p_m}$, where u_i are Lyndon words such that $u_1 \succ \dots \succ u_m$ and $p_1, \dots, p_m \in \mathbb{Z}_{>0}$ are integer exponents. Every string has a unique Lyndon factorization [35], which can be computed in linear time and constant auxiliary space [52]. The following result characterizes the Lyndon factorization of the concatenation of two strings (see also Figure 9.1):

Lemma 9.1.5 ([9, 50]). *Let $u = u_1^{p_1} \cdots u_m^{p_m}$ and $v = v_1^{q_1} \cdots v_\ell^{q_\ell}$ be Lyndon factorizations. Then the Lyndon factorization of uv is $uv = u_1^{p_1} \cdots u_c^{p_c} z^k v_{d+1}^{q_{d+1}} \cdots v_\ell^{q_\ell}$ for integers c, d, k and a Lyndon word z such that $0 \le c < m$, $0 \le d \le \ell$, and $z^k = u_{c+1}^{p_{c+1}} \cdots u_m^{p_m} v_1^{q_1} \cdots v_d^{q_d}$.*

Next, we prove another simple yet useful property of Lyndon words:

Fact 9.1.6. *Let $v, w \in \Sigma^+$ be strings such that w is a Lyndon word. If $v \prec w$, then $v^\infty \prec w$.*

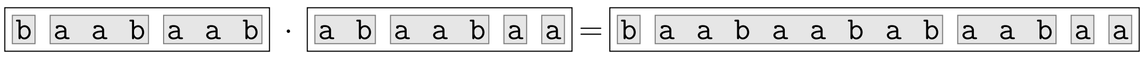


Figure 9.1: Lyndon factorizations of $\mathtt{baabaab}$, $\mathtt{abaabaa}$, and their concatenation.

Proof. For a proof by contradiction suppose that $v \prec w \prec v^\infty$. Let $w = v^k s$, where the integer exponent k is largest possible, i.e., v is not a prefix of s. Due to $w = v^k s \prec v^\infty$, we have $s \prec v^\infty$. On the other hand, w is a Lyndon word, so $w \preceq s$. Consequently, $v \prec w \preceq s \prec v^\infty$, which means that v is a prefix of s and contradicts its definition. □

9.1.1 Significant Suffixes

Below, we recall a notion of *significant suffixes*, introduced by I et al. [76] in order to compute Lyndon factorizations of grammar-compressed strings. Then, we state combinatorial properties of significant suffixes; some of them are novel and some were proved in [76].

Definition 9.1.7 (see [76]). *A suffix s of a string $v \in \Sigma^*$ is a* significant suffix *of v if $sw = \mathrm{MinSuf}(v, w)$ for some $w \in \Sigma^*$.*

Example 9.1.8. For $v = \mathtt{abaabaa}$, we have $\Lambda(v) = \{\mathtt{aabaa}, \mathtt{aa}, \varepsilon\}$. Witness strings w are $\mathtt{bb}$, $\mathtt{b}$, and $\mathtt{a}$, respectively. Note that $\mathtt{a} = \mathrm{MinSuf}(v) \notin \Lambda(v)$ because $\mathtt{a}w \succ \min(w, \mathtt{aa}w)$.

Let $v = v_1^{p_1} \dots v_m^{p_m}$ be the Lyndon factorization of a string $v \in \Sigma^+$. For $1 \le j \le m$, we denote $s_j = v_j^{p_j} \cdots v_m^{p_m}$; moreover, we assume $s_{m+1} = \varepsilon$. Let λ be the smallest index such that s_{i+1} is a prefix of v_i for $\lambda \le i \le m$; see Figure 9.2. Observe that

$$s_\lambda \succ v_\lambda \succ s_{\lambda+1} \succ \cdots \succ s_m \succeq v_m \succ s_{m+1} = \varepsilon;$$

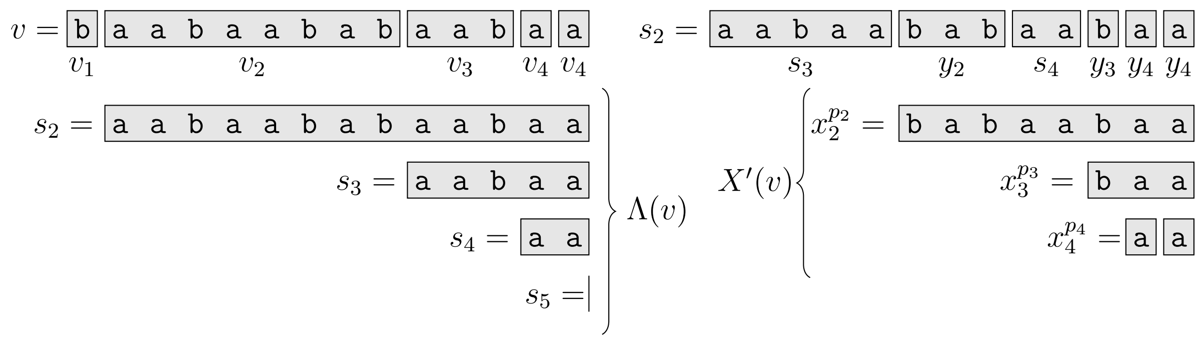
in fact, each string in this sequence is a prefix of the previous one. We define y_i so that $v_i = s_{i+1} y_i$, and we set $x_i = y_i s_{i+1}$. Note that

$$s_i = v_i^{p_i} s_{i+1} = (s_{i+1} y_i)^{p_i} s_{i+1} = s_{i+1} (y_i s_{i+1})^{p_i} = s_{i+1} x_i^{p_i}.$$

We also denote

$$\begin{aligned} \Lambda(v) &= \{s_\lambda, \dots, s_m, s_{m+1}\}, \\ X(v) &= \{x_\lambda^\infty, \dots, x_m^\infty\}, \\ X'(v) &= \{x_\lambda^{p_\lambda}, \dots, x_m^{p_m}\}; \end{aligned}$$

see Figure 9.2 for examples. The observation below lists several immediate properties of the introduced strings:



$v =$ b a a b a a b a b a a b a a
v_1 v_2 v_3 v_4 v_4
$s_2 =$ a a b a a b a b a a b a a
$s_3 =$ a a b a a
$s_4 =$ a a
$s_5 =$
$\Lambda(v)$
$s_2 =$ a a b a a b a b a a b a a
s_3 y_2 s_4 y_3 y_4 y_4
$X'(v)$
$x_2^{p_2} =$ b a b a a b a a
$x_3^{p_3} =$ b a a
$x_4^{p_4} =$ a a

Figure 9.2: Illustration of the sets $\Lambda(v)$ and $X'(v)$ for $v = \mathtt{baabaababaabaa}$. We have $m = 4$ and $\lambda = 2$.

Observation 9.1.9. *For each i, $\lambda \le i \le m$:*
(a) $x_i^\infty \succ x_i^{p_i} \succeq x_i \succeq y_i$,
(b) $x_i^{p_i}$ *is a suffix of v of length* $|s_i| - |s_{i+1}|$,
(c) $|s_i| > 2|s_{i+1}|$.
Consequently, $|\Lambda(v)| = \mathcal{O}(\log |v|)$.

The following lemma shows that $\Lambda(v)$ is equal to the set of significant suffixes of v. The significant suffixes are actually defined in [76] as $\Lambda(v)$ and only later proved to be characterized by the condition stated in Definition 9.1.7. In fact, the lemma is much deeper; in particular, the formula for $\mathrm{MaxSuf}(v, w)$ is one of the key ingredients of our efficient algorithms answering Minimal Suffix Queries.

Lemma 9.1.10 (I et al. [76], Lemmas 12–14)**.** *For a string $v \in \Sigma^+$, let s_i, λ, x_i, and y_i be defined as above. Then*

$$x_\lambda^\infty \succ x_\lambda^{p_\lambda} \succeq y_\lambda \succ x_{\lambda+1}^\infty \succ x_{\lambda+1}^{p_{\lambda+1}} \succeq y_{\lambda+1} \succ \ldots \succ x_m^\infty \succ x_m^{p_m} \succeq y_m.$$

Moreover, for every string $w \in \bar{\Sigma}^*$, we have

$$\mathrm{MinSuf}(v, w) = \begin{cases} s_\lambda w & \text{if } w \succ x_\lambda^\infty, \\ s_i w & \text{if } x_{i-1}^\infty \succ w \succ x_i^\infty \text{ for } \lambda < i \le m, \\ s_{m+1} w & \text{if } x_m^\infty \succ w. \end{cases}$$

In other words, $\mathrm{MinSuf}(v, w) = s_{m+1-r} w$, *where* $r = \mathrm{rank}_{X(v)}(w)$.

Example 9.1.11. For a string $v = \texttt{baabaababaabaa}$ (see Figure 9.2), we have:

$$\mathrm{MinSuf}(v, w) = \begin{cases} \texttt{aabaababaabaa} \cdot w & \text{if } (\texttt{babaabaa})^\infty \prec w, \\ \texttt{aabaa} \cdot w & \text{if } (\texttt{baa})^\infty \prec w \prec (\texttt{babaabaa})^\infty, \\ \texttt{aa} \cdot w & \text{if } \texttt{a}^\infty \prec w \prec (\texttt{baa})^\infty, \\ w & \text{if } w \prec \texttt{a}^\infty. \end{cases}$$

Lemma 9.1.10 yields several properties of the set $\Lambda(v)$ of significant suffixes.

Corollary 9.1.12. *Consider a string $v \in \Sigma^+$.*
(a) The longest suffix in $\Lambda(v)$ is $\mathrm{MaxSuf}^R(v)$ *and* $\Lambda(v) = \Lambda(\mathrm{MaxSuf}^R(v))$.
(b) If a decomposition $v = uv'$ satisfies $|u| \le |v'| + 1$, then

$$\Lambda(v) \subseteq \Lambda(v') \cup \{\mathrm{MaxSuf}^R(v)\} \subseteq \Lambda(v') \cup \{\mathrm{MaxSuf}^R(u, v')\}.$$

Consequently, $\mathrm{MinSuf}(v, w) \in \{\mathrm{MaxSuf}^R(u, v')w, \mathrm{MinSuf}(v', w)\}$ *for every $w \in \bar{\Sigma}^*$.*

Proof. Throughout the proof, we use the notation introduced just above Observation 9.1.9.

To prove (a), observe that $x_\lambda \in \Sigma^+$, so $x_\lambda^\infty \prec \$$. Consequently, Lemma 9.1.10 states that $s_\lambda\$ = \mathrm{MinSuf}(v, \$)$. However, we have $\mathrm{MinSuf}(v, \$) = \mathrm{MaxSuf}^R(v)\$$ by Observation 9.1.3(e), and thus $s_\lambda = \mathrm{MaxSuf}^R(v)$. Uniqueness of the Lyndon factorization implies that $u_\lambda^{p_\lambda} \cdots u_m^{p_m}$ is the Lyndon factorization of s_λ, and hence $\Lambda(v) = \Lambda(s_\lambda)$ by definition of $\Lambda(\cdot)$.

For a proof of the first inclusion in (b), we shall show that for $i \ge \lambda + 1$ the string s_i is a significant suffix of v'. The suffix $s_{m+1} = \varepsilon$ is clearly a significant suffix of

v', so we assume $\lambda < i \le m$. Note that, by Observation 9.1.9, s_i is a suffix of v' since $2|s_i| + 1 \le |s_{i-1}| \le |s_\lambda| \le |v| \le 2|v'| + 1$. By Lemma 9.1.10, one can choose $w \in \bar{\Sigma}^*$ (setting $x_{i-1}^\infty \succ w \succ x_i^\infty$) so that $s_i w = \mathrm{MinSuf}(v, w)$. However, this also implies $s_i w = \mathrm{MinSuf}(v', w)$ because all suffixes of v' are suffixes of v. Consequently, s_i is a significant suffix of v', as claimed.

Note that $\mathrm{MaxSuf}^R(v) = \mathrm{MaxSuf}^R(uv') \in \{\mathrm{MaxSuf}^R(u, v'), \mathrm{MaxSuf}^R(v')\}$ by Observation 9.1.3. Moreover, part (a) yields $\mathrm{MaxSuf}^R(v') \in \Lambda(v')$, so $\mathrm{MaxSuf}^R(v) \in \Lambda(v') \cup \{\mathrm{MaxSuf}^R(u, v')\}$. This proves the second inclusion in (b). The final claim follows from $\Lambda(v) \subseteq \Lambda(v') \cup \{\mathrm{MaxSuf}^R(u, v')\}$ due to the definition of significant suffixes. □

Next, we provide a precise characterization of $\Lambda(uv)$ for $|u| \le |v|$ in terms of $\Lambda(v)$ and $\mathrm{MaxSuf}^R(u, v)$; see Table 9.1. This is another key ingredient of our data structure. In Section 9.2, we use it to efficiently compute significant suffixes of a given fragment of T.

Lemma 9.1.13. *Let $u, v \in \Sigma^+$ be strings such that $|u| \le |v|$. Moreover, let $\Lambda(v) = \{s_\lambda, \dots, s_{m+1}\}$, $s' = \mathrm{MaxSuf}^R(u, v)$, and let s_i be the longest suffix in $\Lambda(v)$ which is a prefix of s'. Then*

$$\Lambda(uv) = \begin{cases} \{s_\lambda, \dots, s_{m+1}\} & \text{if } s' \preceq^R s_\lambda \text{ (i.e., if } s_\lambda \preceq s' \text{ and } i \ne \lambda), \\ \{s', s_{i+1}, \dots, s_{m+1}\} & \text{if } s' \succ^R s_\lambda,\ i \le m, \text{ and } |s_i| - |s_{i+1}| \text{ is a period of } s', \\ \{s', s_i, s_{i+1}, \dots, s_{m+1}\} & \text{otherwise.} \end{cases}$$

Proof. Observation 9.1.3 yields $\mathrm{MaxSuf}^R(uv) \in \{\mathrm{MaxSuf}^R(u, v), \mathrm{MaxSuf}^R(v)\}$, which is equivalent to $\mathrm{MaxSuf}^R(uv) \in \{s', s_\lambda\}$ by Corollary 9.1.12(a). Consequently, if $s' \preceq^R s_\lambda$, then $\mathrm{MaxSuf}^R(uv) = s_\lambda$ and Corollary 9.1.12(a) implies $\Lambda(uv) = \Lambda(s_\lambda) = \Lambda(v)$, as claimed.

Thus, we may assume that $s' \succ^R s_\lambda$ and in particular that $s' = \mathrm{MaxSuf}^R(uv)$. Let s_j be the longest suffix in $\Lambda(uv) \cap \Lambda(v)$ $(\lambda \le j \le m + 1)$. By Corollary 9.1.12(b), $\Lambda(uv) \subseteq \{s'\} \cup \{s_j, s_{j+1}, \dots, s_{m+1}\}$. Lemma 9.1.5 and the definition the $\Lambda(\cdot)$ set in terms of the Lyndon factorization yield that the inclusion above is actually an equality. Moreover, the definition also implies that s_j is a prefix of s', and thus $j \ge i$. If $i = m + 1$, this already proves our statement, so in the remainder of the proof we assume $i \le m$.

First, let us suppose that $j \ge i + 1$. We shall prove that $j = i + 1$ and $|s_i| - |s_{i+1}|$ is a period of s'. Let u' be a string such that $s' = u' s_j$. Note that $v_i^{p_i} \dots v_{j-1}^{p_{j-1}}$ is a border of u' since s_i is a border of s'. Moreover, by definition of $\Lambda(uv)$, the string u' must be a power of a Lyndon word. Lyndon words do not have proper borders, so any border of u' must be a power of the same Lyndon word. Thus, u' and $v_i^{p_i} \dots v_{j-1}^{p_{j-1}}$ are powers of the same

u	v	s'	$s' \preceq^R s_\lambda$	i	$\Lambda(uv)$
abaabaa	abaabaa	aa abaabaa	no	3	$\{$aaabaabaa, aa, $\varepsilon\}$
babaaba	abaabaa	aaba abaabaa	no	2	$\{$aabaabaabaa, aa, $\varepsilon\}$
baabaab	abaabaa	aabaab abaabaa	no	2	$\{$aabaababaabaa, aabaa, aa, $\varepsilon\}$
ababaab	abaabaa	aab abaabaa	yes	3	$\{$aabaa, aa, $\varepsilon\}$

Table 9.1: Example applications of Lemma 9.1.13 for $v =$ abaabaa and various strings u. We have $v =$ ab $\cdot$ aab $\cdot$ a $\cdot$ a, $m = 4$, $\lambda = 2$, and $\Lambda(v) = \{s_2, s_3, s_4\} = \{$aabaa, aa, $\varepsilon\}$.

Lyndon word, which must be $v_i = v_{j-1}$ by the uniqueness of the Lyndon factorization. What is more, as s_{i+1} is a prefix of v_i, we conclude that $|v_i|$ is a period of $s' = u's_{i+1}$. Therefore, $|s_i| - |s_{i+1}| = p_i|v_i|$ is also a period of s'.

It remains to prove that $j = i$ implies that $|s_i| - |s_{i+1}|$ is not a period of s'. Suppose that $i = j$, i.e., $s_i \in \Lambda(uv)$. By definition of $\Lambda(uv)$, we have $s' = (v')^{p'} s_i$ for a Lyndon word v' which has s_i as a prefix. Since v' cannot have a proper border, the shortest period of s' is at least $|v'| \geq |s_i| > |s_i| - |s_{i+1}|$. Thus, $|s_i| - |s_{i+1}|$ indeed is not a period of s'. $\square$

We conclude with two combinatorial lemmas which are both useful in determining $\mathrm{MaxSuf}^R(u, v)$ for $|u| \leq |v|$. The first of them is also applied later in Section 9.4.

Lemma 9.1.14. *Let $v \in \Sigma^+$ and $w, w' \in \bar{\Sigma}^+$ be strings such that $w \prec w'$ and the longest common prefix of w and w' is not a proper substring of v. Also, let $\Lambda(v) = \{s_\lambda, \dots, s_{m-1}\}$. If $\mathrm{MinSuf}(v, w) = s_i w$, then $\mathrm{MinSuf}(v, w') \in \{s_{i-1}w', s_i w'\}$.*

Proof. Due to the characterization in Lemma 9.1.10, we may equivalently prove that $\mathrm{rank}_{X(v)}(w')$ is $\mathrm{rank}_{X(v)}(w)$ or $\mathrm{rank}_{X(v)}(w) + 1$. Clearly, $\mathrm{rank}_{X(v)}(w) \leq \mathrm{rank}_{X(v)}(w')$, so it suffices to show that $\mathrm{rank}_{X(v)}(w') \leq \mathrm{rank}_{X(v)}(w) + 1$. This is clear if $|X(v)| = 1$, so we assume $|X(v)| > 1$. This assumption in particular yields that $X'(v)$ consists of proper substrings of v, and thus $\mathrm{rank}_{X'(v)}(w) = \mathrm{rank}_{X'(v)}(w')$ by the condition on the longest common prefix of w and w'. However, the inequality in Lemma 9.1.10 implies

$$\mathrm{rank}_{X(v)}(w') \leq \mathrm{rank}_{X'(v)}(w') = \mathrm{rank}_{X'(v)}(w) \leq \mathrm{rank}_{X(v)}(w) + 1.$$

This concludes the proof. $\square$

Lemma 9.1.15. *Let $v \in \Sigma^+$, $v = v_1^{p_1} \cdots v_m^{p_m}$ be the Lyndon factorization of v, and let $\Lambda(v) = \{s_\lambda, \dots, s_{m+1}\}$. If $\mathrm{MinSuf}(v, w) = s_i w$ for some $w \in \bar{\Sigma}^*$ and $\lambda < i \leq m+1$, then $v_{i-1}s_i w \preceq sw$ for every suffix s of v satisfying $|s| > |s_i|$.*

Proof. Let s' be a non-empty string such that $s = s's_i$. First, suppose that $|s| < |v_{i-1}s_i|$. In this case, s' is a proper suffix of the Lyndon word v_{i-1}; thus, $s' \succ v_{i-1}$ and, moreover, $sw \succ s' \succ v_{i-1}s_i w$. Consequently, we may assume that $|s| \geq |v_{i-1}s_i|$.

Let $w' = v_{i-1}s_i w$ and let v' be a string such that $v = v'v_{i-1}s_i$. Observe that it suffices to prove that $\mathrm{MinSuf}(v', w') = w'$, which implies that $sw \succeq w' = v_{i-1}s_i w$ for $|s| \geq |v_{i-1}s_i|$. If $v' = \varepsilon$, then there is nothing to prove, so we shall assume $|v'| > 0$. Note that we have the Lyndon factorization $v' = v_1^{p_1} \cdots v_{i-1}^{p_{i-1}-1}$ with $i > 2$ or $p_{i-1} > 1$. By Lemma 9.1.10, $\mathrm{MinSuf}(v, w) = s_i w$ implies $w \prec x_{i-1}^\infty$, whereas $\mathrm{MinSuf}(v', w') = w'$ is equivalent to $w' \prec v_{i-1}^\infty$ (if $p_{i-1} > 1$) or $w' \prec v_{i-2}^\infty$ (if $p_{i-1} = 1$). We have

$$w' = v_{i-1}s_i w \prec v_{i-1}s_i x_{i-1}^\infty = v_{i-1}s_i(y_{i-1}s_i)^\infty = v_{i-1}(s_i y_{i-1})^\infty = v_{i-1}v_{i-1}^\infty = v_{i-1}^\infty.$$

If $p_{i-1} > 1$, this already concludes the proof, and thus we may assume that $p_{i-1} = 1$. By definition of the Lyndon factorization, we have $v_{i-2} \succ v_{i-1}$, and by Fact 9.1.6, this implies $v_{i-2} \succ v_{i-1}^\infty$. Hence, $w' \prec v_{i-1}^\infty \prec v_{i-2} \prec v_{i-2}^\infty$, which concludes the proof. $\square$

9.2 Computing Significant Suffixes

In this section, we develop a data structure computing $\Lambda(v)$ for a given fragment v of the input text T. We call it the *augmented suffix array* of T because it only consists of fairly standard components built on top of the suffix array of T and its reverse T^R:

- the data structures for LCE QUERIES in T and T^R (see Proposition 2.5.1), and
- the inverse suffix array ISA of T with a component for range minimum queries (see Section 3.7).

The same tools have already been applied in [11]. As stated below, the augmented suffix array lets us efficiently support many basic queries, some of which are listed in Fact 2.5.2.

Proposition 9.2.1 (Augmented suffix array)**.** *The augmented suffix array of a text T of length n takes $\mathcal{O}(n)$ space, can be constructed in $\mathcal{O}(n)$ time, and answers in $\mathcal{O}(1)$ time the following queries for fragments x, y of T or fragments x, y of T^R:*

(a) compute the length of the longest common prefix $\operatorname{lcp}(x, y)$;
(b) determine if $x \prec y$, $x \cong y$, or $x \succ y$ (as well as whether $x \prec^R y$, $x \cong y$, or $x \succ^R y$);
(c) compute $\operatorname{lcp}(x^\infty, y)$ and determine if $x^\infty \prec y$ or $x^\infty \succ y$ (as well as whether $x^\infty \prec^R y$ or $x^\infty \succ^R y$).

Moreover, given indices $i, j \in [\![1, n]\!]$, it can compute in $\mathcal{O}(1)$ time the lexicographically smallest suffix among $\{T[k\,..] : k \in [\![i, j]\!]\}$.

Proof. The time and space complexity of the components for LCE QUERIES is specified by Proposition 2.5.1. The inverse suffix array ISA and a component for constant-time range minimum queries are also constructed in $\mathcal{O}(n)$; see Proposition 3.7.6.

Queries (a)–(c) can be answered efficiently due to Fact 2.5.2; note that the reverse lexicographic order, just like the usual lexicographic order, is determined by the characters following the longest common prefix (or the lack of these characters). For the final query type, recall that $ISA[k]$ is the rank of $T[k\,..]$ among the suffixes of T (with respect to the lexicographic order). Hence, the minimum suffix among $\{T[k\,..] : k \in [\![i, j]\!]\}$ and the minimum value in $\{ISA[k] : k \in [\![i, j]\!]\}$ are attained for the same index k. □

Efficient computation of significant suffixes is based on Lemma 9.1.13, which yields a recursive procedure. The only "new" suffix needed at each step is determined using the following result, which can be seen as a cleaner formulation of Lemma 14 in [11].

Lemma 9.2.2. *Let $u = T[\ell\,..\,r]$ and $v = T[r+1\,..\,r']$ be fragments of T such that $|u| \le |v|$. Using the augmented suffix array of T, we can compute $\operatorname{MaxSuf}^R(u, v)$ in $\mathcal{O}(1)$ time.*

Proof. Let $sv = \operatorname{MaxSuf}^R(u, v)$. Note that $sv\$ = \operatorname{MinSuf}(u, v\$)$ by Observation 9.1.3(d). Let us focus on determining the latter value. The augmented suffix array lets us compute an index $k \in [\![\ell, r]\!]$ which minimizes $T[k\,..]$. Equivalently, we have $T[k\,..] = \operatorname{MinSuf}(u, T[r+1\,..])$. Consequently, $T[k\,..\,r] \in \Lambda(u)$, i.e., $T[k\,..\,r] = s_i$ for some $i \in [\![\lambda, m+1]\!]$, where $\{s_\lambda, \dots, s_{m+1}\} = \Lambda(u)$. Note that $|v| = \operatorname{lcp}(v\$, T[r+1\,..])$ and v is not a proper substring of u because $|u| \le |v|$. Hence, by Lemma 9.1.14, we have $s \in \{s_{i-1}, s_i\}$ (if $i = \lambda$, then $s = s_i$).

Thus, we shall generate a suffix of $T[\ell\,..\,r]$ which matches s_{i-1} if $i > \lambda$, and return the better of the two candidates for $\operatorname{MinSuf}(u, v\$)$. If $k = \ell$, we must have $i = \lambda$ and there is nothing to do. Hence, let us assume $k > \ell$. By Lemma 9.1.15, if we compute an index $k' \in [\![\ell, k-1]\!]$ which minimizes $T[k'\,..]$, we shall have $T[k'\,..\,k-1] \cong v_{i-1}$ provided that $i > \lambda$. Now, the exponent p_{i-1} can be generated as the largest integer such that $v_{i-1}^{p_{i-1}}$ is a suffix of $T[\ell\,..\,k-1]$. In other words, $p_{i-1} = \left\lfloor \frac{1}{k-k'} \operatorname{lcp}((T[k'\,..\,k-1]^R)^\infty, T[\ell\,..\,k-1]^R) \right\rfloor$ can be computed using the augmented suffix array. Our candidates for sv are $T[k\,..\,r']$ and $T[k - p_{i-1}(k-k')\,..\,r']$. We use the augmented suffix array to find the larger of the two with respect to $\prec^R$. □

Lemma 9.2.3. *Given a fragment v of T, we can compute $\Lambda(v)$ in $\mathcal{O}(\log |v|)$ time using the augmented suffix array of T.*

Proof. If $|v| = 1$, we return $\Lambda(v) = \{v, \varepsilon\}$. Otherwise, we decompose $v = uv'$ so that $|v'| = \left\lceil \frac{1}{2}|v| \right\rceil$. We recursively generate $\Lambda(v')$ and use Lemma 9.2.2 to compute $s = \mathrm{MaxSuf}^R(u, v')$. Then, we apply the characterization of Lemma 9.1.13 to determine $\Lambda(v) = \Lambda(uv')$, using the augmented suffix array (Proposition 9.2.1) to lexicographically compare fragments of T.

We store the lengths of the significant suffixes in an ordered list. This way, we can implement a single phase (excluding the recursive call) in time proportional to $\mathcal{O}(1)$ plus the number of suffixes removed from $\Lambda(v')$ to obtain $\Lambda(v)$. Since this is amortized constant time, the total running time becomes $\mathcal{O}(\log |v|)$ as announced. □

9.3 Minimal Suffix Queries

In this section, we present our data structure for Minimal Suffix Queries. We proceed in three steps improving the query time from $\mathcal{O}(\log |v|)$ via $\mathcal{O}(\log^* |v|)$ to $\mathcal{O}(1)$. The first solution is an immediate application of Observation 9.1.3(c), the notion of significant suffixes, and the procedure computing these suffixes (Lemma 9.2.3).

Corollary 9.3.1. Minimal Suffix Queries *can be answered in $\mathcal{O}(\log |v|)$ time using the augmented suffix array of T.*

Proof. Recall that Observation 9.1.3(c) yields $\mathrm{MinSuf}(v) = \mathrm{MinSuf}(v[1\mathinner{.\,.} m-1], v[m])$, where $m = |v|$. Consequently, $\mathrm{MinSuf}(v) = s \cdot v[m]$ for some $s \in \Lambda(v[1\mathinner{.\,.} m-1])$. We apply Lemma 9.2.3 to compute $\Lambda(v[1\mathinner{.\,.} m-1])$ and determine the answer among the $\mathcal{O}(\log |v|)$ candidates using the lexicographic comparison of fragments, which is supported by the augmented suffix array (Proposition 9.2.1). □

An alternative $\mathcal{O}(\log |v|)$-time algorithm can be stated as a recursive procedure: decompose $v = uv'$ so that $|v'| > |u|$ and return $\min(\mathrm{MaxSuf}^R(u, v'), \mathrm{MinSuf}(v'))$. The result is $\mathrm{MinSuf}(v)$ due to Corollary 9.1.12(b) and Observation 9.1.3(c). Here, the first candidate $\mathrm{MaxSuf}^R(u, v')$ is determined using Lemma 9.2.2, while the second one is obtained from a recursive call. A way to improve query time to $\mathcal{O}(1)$ at the price of $\mathcal{O}(n \log n)$-time preprocessing is to precompute the answers for the *basic fragments*, i.e., for fragments whose length is a power of two. Then, in order to determine $\mathrm{MinSuf}(v)$, we perform just a single step of the aforementioned procedure, making sure that v' is a basic fragment. Both these ideas are actually present in [11], along with a smooth trade-off between their preprocessing and query times.

9.3.1 $\mathcal{O}(\log^* |v|)$-Time Minimal Suffix Queries

Our $\mathcal{O}(\log^* |v|)$-time query algorithm combines recursion with preprocessing for certain *distinguished* basic fragments. More precisely, we say that $v = T[\ell\mathinner{.\,.} r]$ is distinguished if both $|v| = 2^q$ and $f(2^q) \mid r$ for some positive integer q, where $f(x) = 2^{\lfloor \log\log x \rfloor^2}$. Note that the number of distinguished fragments of length 2^q is at most $\frac{n}{2^{\lfloor \log q \rfloor^2}} = \mathcal{O}(\frac{n}{q^{\omega(1)}})$.

The query algorithm is based on the following decomposition ($x > f(x)$ for $x > 2^{16}$):

Fact 9.3.2. *Given a fragment v such that $|v| > f(|v|)$, we can in constant time decompose $v = uv'v''$ so that $1 \le |v''| \le f(|v|)$, v' is distinguished, and $|u| \le |v'|$.*

Proof. Let $v = T[\ell\,..\,r]$, $q = \lfloor \log |v| \rfloor$ and $q' = \lfloor \log q \rfloor^2$. We determine r' as the largest integer strictly smaller than r divisible by $2^{q'} = f(|v|)$. By the assumption that $|v| > 2^{q'}$, we conclude that $r > r' \geq r - 2^{q'} \geq \ell$. We define $v'' = T[r'+1\,..\,r]$ and partition $T[\ell\,..\,r'] = uv'$ so that $|v'|$ is the largest possible power of two. This guarantees $|u| \leq |v'|$. Moreover, $|v'| \leq |v|$ assures that $f(|v'|) \mid f(|v|)$, so $f(|v'|) \mid r'$, and therefore v' is indeed distinguished. □

Observation 9.1.3(b) implies $\text{MinSuf}(v) \in \{\text{MinSuf}(uv', v''), \text{MinSuf}(v'')\}$, and Corollary 9.1.12(b) further yields $\text{MinSuf}(v) \in \{\text{MaxSuf}^R(u, v')v'', \text{MinSuf}(v', v''), \text{MinSuf}(v'')\}$. Thus, there are three candidates for $\text{MinSuf}(v)$. Our query algorithm uses Lemma 9.2.2 to obtain $\text{MaxSuf}^R(u, v')v''$ and computes $\text{MinSuf}(v'')$ recursively. It determines the remaining candidate, $\text{MinSuf}(v', v'')$, through the characterization of Lemma 9.1.10. This is performed using the following component, which is based on a fusion tree and built for all distinguished fragments.

Lemma 9.3.3. *Let $v = T[\ell\,..\,r]$ be a fragment of T. There exists a data structure of size $\mathcal{O}(\log |v|)$ which answers the following queries in $\mathcal{O}(1)$ time: given a position $r' > r$ compute* $\text{MinSuf}(v, T[r+1\,..\,r'])$. *Moreover, this data structure can be constructed in $\mathcal{O}(\log |v|)$ time using the augmented suffix array of T.*

Proof. By Lemma 9.1.10, we have $\text{MinSuf}(v, w) = s_{m+1-\text{rank}_{X(v)}(w)} w$, so in order to determine $\text{MinSuf}(v, T[r+1\,..\,r'])$, it suffices to store $\Lambda(v)$ and efficiently compute $\text{rank}_{X(v)}(w)$ given $w = T[r+1\,..\,r']$. We shall reduce these rank queries to rank queries in an integer multiset $R(v)$; see Figure 9.3.

Claim 9.3.4. Denote $X(v) = \{x_\lambda^\infty, \ldots, x_m^\infty\}$ and define a multiset

$$R(v) = \{r + 1 + \text{lcp}(x_j^\infty, T[r+1\,..]) : x_j^\infty \in X(w) \text{ and } x_j^\infty \prec T[r+1\,..]\}.$$

For every index r', $r < r' \leq n$, we have $\text{rank}_{X(v)}(T[r+1\,..\,r']) = \text{rank}_{R(v)}(r')$.

Proof. We shall prove that for each j, $\lambda \leq j \leq m$, we have

$$x_j^\infty \preceq T[r+1\,..\,r'] \iff \Big(r + 1 + \text{lcp}(x_j^\infty, T[r+1\,..]) \leq r' \;\wedge\; x_j^\infty \prec T[r+1\,..]\Big).$$

First, if $x_j^\infty \succ T[r+1\,..]$, then clearly $x_j^\infty \succ T[r+1\,..\,r']$ and both sides of the equivalence are false. Therefore, we may assume $x_j^\infty \prec T[r+1\,..]$. Observe that in this case $d := \text{lcp}(T[r+1\,..], x_j^\infty) < n-r$ (i.e., $T[r+1\,..]$ is not a prefix of x_j^∞), and $T[r+1\,..\,r+d] \prec x_j^\infty \prec T[r+1\,..\,r+d+1]$. Hence, $x_j^\infty \prec T[r+1\,..\,r']$ if and only if $r+1+d \leq r'$, as claimed. □

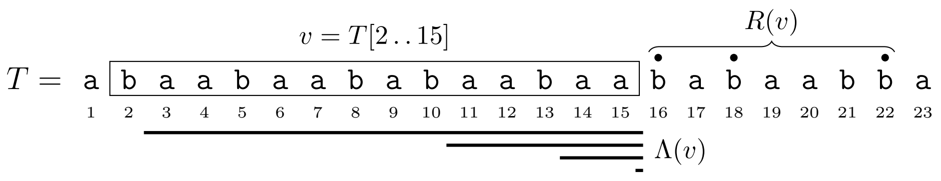


Figure 9.3: Illustration of Claim 9.3.4 for a fragment $v = T[2\,..\,15]$ matching the string considered in Figure 9.2 and Example 9.1.11. For example, we have $\text{MinSuf}(v, T[16\,..\,16]) \cong T[14\,..\,16]$ because $\text{rank}_{R(v)}(16) = 1$, $\text{MinSuf}(v, T[16\,..\,20]) \cong T[11\,..\,20]$ because $\text{rank}_{R(v)}(20) = 2$, and $\text{MinSuf}(v, T[16\,..\,22]) \cong T[3\,..\,22]$ because $\text{rank}_{R(v)}(22) = 3$.

We apply Theorem 3.4.1 to build a fusion tree for $R(v)$ so that the ranks are can be obtained in $\mathcal{O}(1+\frac{\log|R(v)|}{\log W})$ time, which is $\mathcal{O}(1+\frac{\log\log|v|}{\log\log n})=\mathcal{O}(1)$ by Observation 9.1.9.

The construction algorithm uses Lemma 9.2.3 to compute $\Lambda(v)=\{s_\lambda,\ldots,s_{m+1}\}$. Next, for each j, $\lambda\le j\le m$, we need to determine $\operatorname{lcp}(T[r+1\..],x_j^\infty)$. This is the same as $\operatorname{lcp}(T[r+1\..],(x_j^{p_j})^\infty)$ and, by Observation 9.1.9, $x_j^{p_j}$ matches the suffix of v of length $|s_i|-|s_{i+1}|$. Hence, the augmented suffix array can be used to compute these longest common prefixes and therefore to construct $R(v)$ in $\mathcal{O}(|\Lambda(v)|)=\mathcal{O}(\log|v|)$ time. □

With this central component, we are ready to give a full description of our data structure with $\mathcal{O}(\log^*|v|)$-time queries.

Theorem 9.3.5. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which answers* Minimal Suffix Queries *in $\mathcal{O}(\log^*|v|)$ time and can be constructed in $\mathcal{O}(n)$ time.*

Proof. Our data structure consists of the augmented suffix array (Proposition 9.2.1) and the components of Lemma 9.3.3 for all distinguished fragments of T. Each such fragment of length 2^q contributes $\mathcal{O}(q)$ to the space consumption and to the construction time, which in total over all lengths sums up to $\mathcal{O}(\sum_q q\frac{n}{q^{\omega(1)}})=\mathcal{O}(\sum_q \frac{n}{q^{\omega(1)}})=\mathcal{O}(n)$.

Let us proceed to the query algorithm. Assume we are to compute the minimal suffix of a fragment v. If $|v|\le f(|v|)$ (i.e., if $|v|\le 2^{16}$), we use the logarithmic-time query algorithm given in Corollary 9.3.1. If $|v|>2^q$, we apply Fact 9.3.2 to determine a decomposition $v=uv'v''$, which gives us three candidates for $\operatorname{MinSuf}(v)$. As already described, $\operatorname{MinSuf}(v'')$ is computed recursively, $\operatorname{MinSuf}(v',v'')$ using Lemma 9.3.3, and $\operatorname{MaxSuf}^R(u,v')v''$ using Lemma 9.2.2. The latter two procedures both support constant-time queries, so the overall time complexity is proportional to the depth of the recursion. We have $|v''|\le f(|v|)<|v|$, so it terminates. Moreover,

$$f(f(x))=2^{\lfloor\log(\log f(x))\rfloor^2}\le 2^{(\log(\log\log x)^2)^2}=2^{4(\log\log\log x)^2}=2^{o(\log\log x)}=o(\log x).$$

Thus, $f(f(x))\le\log x$ unless $x=\mathcal{O}(1)$. Consequently, unless $|v|=\mathcal{O}(1)$, when the algorithm clearly needs constant time, the length of the queried fragment is in two steps reduced from $|v|$ to at most $\log|v|$. This concludes the proof that the query time is $\mathcal{O}(\log^*|v|)$. □

9.3.2 $\mathcal{O}(1)$-Time Minimal Suffix Queries

The $\mathcal{O}(\log^*|v|)$ time complexity of the query algorithm of Theorem 9.3.5 is only due to the recursion, which in a single step reduces the length of the queried fragment from $|v|$ to $f(|v|)$, where $f(x)=2^{\lfloor\log\log x\rfloor^2}$. Since $f(f(x))=2^{o(\log\log x)}$, after just two steps the fragment length does not exceed $f(f(n))=2^{o(\log\log n)}=(\log n)^{o(1)}=o(\frac{\log n}{\log\log n})$. In this section, we show that the minimal suffixes of such short fragments can be precomputed in a certain sense, and thus after reaching $\tau=f(f(n))$ we do not need to perform further recursive calls.

For alphabets of constant size σ, we could actually store all the answers for the $\mathcal{O}(\sigma^\tau)=n^{o(1)}$ strings of length up to τ. Nevertheless, in general all letters of T, and consequently all fragments of T, could even be distinct strings. However, the answers to Minimal Suffix Queries actually depend only on the relative order between letters, which is captured by order-isomorphism.

Two strings x and y are called *order-isomorphic* [96, 84], denoted as $x \approx y$, if $|x| = |y|$ and for every two positions i, j ($1 \le i, j \le |x|$) we have $x[i] \prec x[j] \iff y[i] \prec y[j]$. Note that the equivalence extends to arbitrary corresponding fragments of x and y, i.e., $x[i\mathinner{.\,.} j] \prec x[i'\mathinner{.\,.} j'] \iff y[i\mathinner{.\,.} j] \prec y[i'\mathinner{.\,.} j']$. Consequently, order-isomorphic strings cannot be distinguished using Minimal Suffix Queries.

Example 9.3.6. Strings $x = \texttt{bcbecbcf}$ and $y = \texttt{abadbabg}$ are order-isomorphic. Consequently, $\mathrm{MinSuf}(x[2\mathinner{.\,.}7]) = \mathrm{MinSuf}(\texttt{cbecbc}) = \texttt{bc} \cong x[6\mathinner{.\,.}7]$ implies $\mathrm{MinSuf}(y[2\mathinner{.\,.}7]) \cong y[6\mathinner{.\,.}7]$. We have $\mathrm{oid}(x) = \mathrm{oid}(y) = (1\,000\,001\,000\,010\,001\,000\,001\,011)_2$ for $m = 8$.

Moreover, observe that every string of length m is order-isomorphic to a string over an alphabet $[\![0, m-1]\!]$. Consequently, order-isomorphism partitions strings of length up to m into $\mathcal{O}(m^m)$ equivalence classes. The following fact lets us compute canonical representations of strings whose length is bounded by $m = W^{\mathcal{O}(1)}$.

Fact 9.3.7. *For every fixed integer m, there exists a function* oid *mapping each string w of length up to m to a non-negative integer* $\mathrm{oid}(w)$ *with $\mathcal{O}(m \log m)$ bits, so that $w \approx w' \iff \mathrm{oid}(w) = \mathrm{oid}(w')$. Moreover, the function* oid *can be evaluated in $\mathcal{O}(m)$ time if $m = W^{\mathcal{O}(1)}$.*

Proof. To compute $\mathrm{oid}(w)$, we first build a fusion tree storing all (distinct) letters which occur in w. Next, we replace each character of w with its rank (minus one) among these letters. We allocate $\lceil \log m \rceil$ bits per character and prepend the representation with an extra **1**. This way, $\mathrm{oid}(w)$ is a sequence of $1 + |w| \lceil \log m \rceil = \mathcal{O}(m \log m)$ bits. Using Theorem 3.4.1 to build the fusion tree, we obtain an $\mathcal{O}(m)$-time evaluation algorithm. $\square$

To answer queries for short fragments of T, we define overlapping *blocks* of length $m = 2\tau$: for $0 \le i < \frac{n}{\tau}$, we create a block $T_i = T[1 + i\tau \mathinner{.\,.} \min(n, (i+2)\tau)]$. For each block, we apply Fact 9.3.7 to compute the identifier $\mathrm{oid}(T_i)$ of the underlying string. The total length of the blocks is bounded $2n$, so this takes $\mathcal{O}(n)$ time. The identifiers use $\mathcal{O}(\frac{n}{\tau}\tau \log \tau) = O(n \log \tau)$ bits of space.

Moreover, for each distinct identifier $\mathrm{oid}(T_i)$, we store the answers to all the Minimal Suffix Queries in T_i. This takes $\mathcal{O}(\log m)$ bits per answer and $\mathcal{O}(2^{\mathcal{O}(m \log m)} m^2 \log m) = 2^{\mathcal{O}(\tau \log \tau)}$ in total. Since $\tau = o(\frac{\log n}{\log \log n})$, this is $n^{o(1)}$. The preprocessing time is also $n^{o(1)}$.

It is a matter of simple arithmetics to extend a given fragment v of T, $|v| \le \tau$, to an enclosing block T_i. We use the precomputed answers stored for $\mathrm{oid}(T_i)$ to determine the minimal suffix of v. We only need to translate the indices within T_i to indices within T before returning the answer. Below, we state our results for short and arbitrary fragments, respectively:

Theorem 9.3.8. *For every text T of length n and every parameter $\tau = o(\frac{\log n}{\log \log n})$, there exists a data structure of size $\mathcal{O}(\frac{n \log \tau}{\log n})$ which answers in $\mathcal{O}(1)$ time* Minimal Suffix Queries *for fragments of length not exceeding τ. Moreover, it can be constructed in $\mathcal{O}(n)$ time.*

Theorem 1.1.9. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which answers* Minimal Suffix Queries *in $\mathcal{O}(1)$ time. It can be constructed in $\mathcal{O}(n)$ time.*

9.4 Generalized Minimal Suffix Queries

In this section, we develop a data structure answering Generalized Minimal Suffix Queries. We start with preliminary definitions and then we describe the counterparts of the three data structures presented in Section 9.3. Their query times are $\mathcal{O}(k^2 \log |v|)$, $\mathcal{O}(k^2 \log^* |v|)$, and $\mathcal{O}(k^2)$, respectively, i.e., there is an $\mathcal{O}(k^2)$ overhead compared to Minimal Suffix Queries.

We define a *k-fragment* of the text T as a sequence $T[\ell_1 \mathinner{.\,.} r_1] \cdots T[\ell_k \mathinner{.\,.} r_k]$ of k fragments of T. Observe that a k-fragment can be stored in $\mathcal{O}(k)$ space as a sequence of pairs (ℓ_i, r_i). If a string w admits such a decomposition using k' ($k' \le k$) substrings of T, we call it a *k-substring* of T. Every k'-fragment (with $k' \le k$) whose value is equal to w is called an *occurrence* of w as a k-substring of T. Observe that a substring of a k-substring w of T is itself a k-substring of T. Moreover, given an occurrence of w, one can canonically assign each fragment of w to a k'-fragment of T ($k' \le k$). This subroutine can be implemented in $\mathcal{O}(k)$ time and referring to $w[\ell \mathinner{.\,.} r]$ in our algorithms, we assume that such an operation is performed behind the scenes.

The augmented suffix array can answer basic queries regarding k-fragments since they are easily reduced to the counterparts for fragments.

Observation 9.4.1. *The augmented suffix array answers queries (a)–(c) in $\mathcal{O}(k)$ time if x and y are k-fragments of T or k-fragments of T^R.*

We answer Generalized Minimal Suffix Queries using the following auxiliary queries:

> Auxiliary Minimal Suffix Queries
> Given a fragment v of T and a k-fragment w of T, compute $\mathrm{MinSuf}(v, w)$ (represented by its length).

Lemma 9.4.2. *For every text T, the minimal suffix of a k-fragment v can be determined by k* Auxiliary Minimal Suffix Queries *(asked for k'-fragments w with $k' < k$) and additional $\mathcal{O}(k^2)$-time processing using the augmented suffix array of T.*

Proof. Let $v = v_1 \cdots v_k$. By Observation 9.1.3(b), either $\mathrm{MinSuf}(v) = \mathrm{MinSuf}(v_k)$ or $\mathrm{MinSuf}(v) = \mathrm{MinSuf}(v_i, v_{i+1} \cdots v_k)$ for some $i \in [\![1, k]\!]$. Hence, we apply Auxiliary Minimal Suffix Queries to determine $\mathrm{MinSuf}(v_i, v_{i+1} \cdots v_k)$ for each $1 \le i < k$. Observation 9.1.3(c) lets us reduce computing $\mathrm{MinSuf}(v_k)$ to another auxiliary query. Having obtained k candidates for $\mathrm{MinSuf}(v)$, we use the augmented suffix array to return the lexicographically smallest of them using $k-1$ comparisons, each performed in $\mathcal{O}(k)$ time; see Proposition 9.2.1 and Observation 9.4.1. $\square$

Fact 9.4.3. Auxiliary Minimal Suffix Queries *can be answered in $\mathcal{O}(k \log |v|)$ time using the augmented suffix array of T.*

Proof. We apply Lemma 9.2.3 to determine $\Lambda(v)$, and then we compute the smallest string among $\{sw : s \in \Lambda(v)\}$. These strings occur as $(k+1)$-fragments of T and thus a single comparison takes $\mathcal{O}(k)$ time using the augmented suffix array. $\square$

Corollary 9.4.4. Generalized Minimal Suffix Queries *can be answered in $\mathcal{O}(k^2 \log |v|)$ time using the augmented suffix array of T.*

9.4.1 $\mathcal{O}(k \log^* |v|)$-Time Auxiliary Minimal Suffix Queries

Our data structure closely follows its counterpart described in Section 9.3.1. We define distinguished fragments in the same manner and provide a recursive algorithm based on Fact 9.3.2. However, instead of applying Lemma 9.3.3, for each distinguished fragment we build the following much stronger data structure. Its implementation is the main technical contribution of this section.

Lemma 9.4.5. *Let v be a fragment of T. There exists a data structure of size $\mathcal{O}(\log^2 |v|)$ which answers the following queries in $\mathcal{O}(k)$ time: determine* $\mathrm{MinSuf}(v, w)$ *for a given k-fragment w of T. The data structure can be constructed in $\mathcal{O}(\log^2 |v|)$ time, assuming that it has access to the augmented suffix array of T.*

Before proving Lemma 9.4.5, we describe how it is used for answering Auxiliary Minimal Suffix Queries. If $f(|v|) \geq |v|$ ($|v| \leq 2^{16}$), we use Fact 9.4.3 to compute $\mathrm{MinSuf}(v, w)$ in $\mathcal{O}(k \log |v|) = \mathcal{O}(k)$ time. Otherwise, we apply Fact 9.3.2 to decompose $v = uv'v''$ so that v' is distinguished, $|u| \leq |v'|$, and $|v''| \leq f(|v|)$, where $f(x) = 2^{\lfloor \log \log x \rfloor^2}$. The characterization of Observation 9.1.3 and Corollary 9.1.12(b) again gives three candidates for $\mathrm{MinSuf}(v, w)$: $\mathrm{MaxSuf}^R(u, v')v''w$, $\mathrm{MinSuf}(v', v''w)$, and $\mathrm{MinSuf}(v'', w)$. We determine the first using Lemma 9.2.2, the second using Lemma 9.4.5, and the third one is computed recursively. The application of Lemma 9.4.5 takes $\mathcal{O}(k + 1)$ time since $v''w$ is a $(k + 1)$-fragment of T. We return the best of the three candidates using the augmented suffix array to choose it in $\mathcal{O}(k)$ time. Since $f(f(x)) = o(\log x)$, the depth of the recursion is $\mathcal{O}(\log^* |v|)$. This concludes the proof of the following result:

Theorem 9.4.6. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which answers* Auxiliary Minimal Suffix Queries *in $\mathcal{O}(k \log^* |v|)$ time and* Generalized Minimal Suffix Queries *in $\mathcal{O}(k^2 \log^* |v|)$ time. It can be constructed in $\mathcal{O}(n)$ time.*

Rank queries in a collection of fragments

The crucial tool we use in the proof of Lemma 9.4.5 is a data structure constructed for a collection A of $W^{\mathcal{O}(1)}$ fragments of T to support $\mathrm{rank}_A(w)$ queries for arbitrary k-fragments w of T. It heavily relies on the compacted trie of the fragments in A (see Section 2.6 for the definition of compacted tries and related concepts).

Before we proceed with ranking a k-fragment in a collection of fragments, let us prove that fusion trees make it relatively easy to rank a suffix in a collection of $W^{\mathcal{O}(1)}$ suffixes.

Fact 9.4.7. *Let A be a set of $W^{\mathcal{O}(1)}$ suffixes of T. There exists a data structure of size $\mathcal{O}(|A|)$, which answers the following queries in $\mathcal{O}(1)$ time: given a suffix v of T, find a suffix $u \in A$ maximizing* $\mathrm{lcp}(u, v)$. *The data structure can be constructed in $\mathcal{O}(|A|)$ time, assuming that it has access to the augmented suffix array of T.*

Proof. Let $A = \{T[\ell_1 \mathinner{.\,.}], \ldots, T[\ell_m \mathinner{.\,.}]\}$. We build a fusion tree storing the ranks $\{ISA[\ell_i] : 1 \leq i \leq m\}$ and during a query for $v = T[\ell \mathinner{.\,.}]$, we determine the predecessor and the successor of $ISA[\ell]$. We use the suffix array (the SA table) to translate these integers into indices ℓ_{i_p} and ℓ_{i_s}. Since the order of the ranks $ISA[\ell_i]$ coincides with the lexicographic order of suffixes $T[\ell_i \mathinner{.\,.}]$, the suffixes $T[\ell_{i_p} \mathinner{.\,.}]$ and $T[\ell_{i_s} \mathinner{.\,.}]$ are the predecessor $\mathrm{pred}_A(v)$ and the successor $\mathrm{succ}_A(v)$, respectively. These are the two candidates for $u \in A$ maximizing $\mathrm{lcp}(u, v)$. We perform two Longest Common Extension Queries and return the candidate for which the obtained value is larger, breaking ties arbitrarily. □

Lemma 9.4.8. *Let A be a set of $W^{\mathcal{O}(1)}$ fragments of T. There exists a data structure of size $\mathcal{O}(|A|^2)$ which answers the following queries in $\mathcal{O}(k)$ time: determine the rank $\mathrm{rank}_A(v)$ of a given k-fragment v of T. The data structure can be constructed in $\mathcal{O}(|A|^2)$ time, assuming that it has access to the augmented suffix array of T.*

Proof. Let $A = \{T[\ell_1 \mathinner{.\,.} r_1], \ldots, T[\ell_m \mathinner{.\,.} r_m]\}$ and let $\mathcal{T}$ be the compacted trie of the underlying substrings of T. Note that $\mathcal{T}$ can be easily constructed in $\mathcal{O}(m \log m)$ time using the augmented suffix array. For each edge, we store a fragment of T representing its label, and for each terminal node, we store its rank in A. Moreover, for each explicit node ν of $\mathcal{T}$, we store pointers to the first and last (in pre-order) terminal nodes in its subtree as well as the following two components: a fusion tree containing the children of ν indexed by the first character of the corresponding edge label, and an instance of the data structure of Fact 9.4.7 for $\{T[\ell_i + d_\nu \mathinner{.\,.}] : \ell_i \in L_\nu\}$, where d_ν is the (weighted) depth of ν and L_ν contains ℓ_i whenever the locus of $T[\ell_i \mathinner{.\,.} r_i]$ is in the subtree of ν. Finally, for each ℓ_i, we store a fusion tree containing (pointers to) all explicit nodes of $\mathcal{T}$ which represent prefixes of $T[\ell_i \mathinner{.\,.}]$, indexed by their (weighted) node depths. All these components can be constructed in $\mathcal{O}(m^2)$ overall time, with Theorem 3.4.1 applied to build fusion trees.

Let us proceed to a description of the query algorithm. Let $v = v_1 \cdots v_k$ be the decomposition of the given k-fragment into fragments, and let $p_i = v_1 \cdots v_i$ for $0 \le i \le k$. We shall scan all v_i consecutively and after processing v_i, store a pointer to the (possibly implicit) node ν_i defined as the locus of the longest prefix of p_i present in $\mathcal{T}$. We start with $p_0 = \varepsilon$, whose locus is the root of $\mathcal{T}$. Therefore, it suffices to describe how to determine ν_i provided that we already know ν_{i-1}.

If ν_{i-1} is at a depth smaller than $|p_{i-1}|$, then there is nothing to do since $\nu_i = \nu_{i-1}$. Otherwise, we proceed as follows: Let ν be the nearest explicit descendant of ν_{i-1} ($\nu = \nu_{i-1}$ if already ν_{i-1} is explicit), and let u be a fragment of T representing the label from ν_{i-1} to ν. First, we check if u matches a proper prefix of v_i. If not, ν_i is on the same edge of $\mathcal{T}$ and its depth is $|p_{i-1}| + \mathrm{lcp}(u, v_i)$. Thus, we may assume that u matches a proper prefix of v_i. Let $v_i \cong u \cdot T[\ell \mathinner{.\,.} r]$. We make a query for the suffix $T[\ell \mathinner{.\,.}]$ to the data structure of Fact 9.4.7 built for ν. This lets us determine an index $\ell_j \in L_\nu$ such that $\mathrm{lcp}(T[\ell \mathinner{.\,.}], T[\ell_j + d_\nu \mathinner{.\,.}])$ is largest possible. This is also an index $\ell_j \in L_\nu$ which maximizes $D := \mathrm{lcp}(p_i, T[\ell_j \mathinner{.\,.}]) = d_\nu + \mathrm{lcp}(T[\ell \mathinner{.\,.} r], T[\ell_j + d_\nu \mathinner{.\,.}])$. Consequently, we observe that ν_i represents $T[\ell_j \mathinner{.\,.} \ell_j + D - 1]$. To obtain the locus of ν_i, we compute its depth D using the augmented suffix array and retrieve the nearest explicit descendant of ν_i from the fusion tree built for ℓ_j as the node whose depth is equal to D or is the successor of D.

After processing the whole k-fragment v, we are left with ν_k, which is the locus of the longest prefix p of v present in $\mathcal{T}$. First, suppose that $|p| < |v|$ and let $c = v[|p| + 1]$. Note that by definition of ν_k, this node does not have an outgoing edge labeled with c. If ν_k is a leaf (and, consequently, a terminal node), then $p = \mathrm{pred}_A(v)$ and we return the rank of ν_k. If ν_k is not a leaf, but it does not have any outgoing edge labeled with a character smaller then c, then the first terminal node of the subtree rooted at the leftmost child of ν_k represents the successor of v in A. We return its rank minus one as the rank of v. Otherwise (if ν_k has an outgoing edge specified above), we determine the edge going from ν_k to some node ν so that the edge label is smaller than c and largest possible. If ν_k is explicit, we use the appropriate fusion tree to determine ν. We observe that the predecessor of v in A is the last terminal node in the subtree of ν and thus we return the rank stored at that terminal node as the rank of v.

Thus, it remains to consider the case when $p = v$. If ν_k is a terminal node, we simply return its rank. Otherwise, the first terminal node in the subtree of ν_k is the successor of

v in A, and thus we return the rank of that node minus one. □

Proof of Lemma 9.4.5

Having developed the key component, we are ready to generalize Lemma 9.3.3, i.e., to prove Lemma 9.4.5. Its statement is repeated below for completeness.

Lemma 9.4.5. *Let v be a fragment of T. There exists a data structure of size $\mathcal{O}(\log^2 |v|)$ which answers the following queries in $\mathcal{O}(k)$ time: determine* $\mathrm{MinSuf}(v, w)$ *for a given k-fragment w of T. The data structure can be constructed in $\mathcal{O}(\log^2 |v|)$ time, assuming that it has access to the augmented suffix array of T.*

Proof. We use Lemma 9.2.3 to compute $\Lambda(v)$ in $\mathcal{O}(\log |v|)$ time. By Lemma 9.1.10, in order to find $\mathrm{MinSuf}(v, w)$, it suffices determine $\mathrm{rank}_{X(v)}(w)$. Moreover, by Lemma 9.1.10 and Observation 9.1.9, $\mathrm{rank}_{X(v)}(w) \in \{\mathrm{rank}_{X'(v)}(w), \mathrm{rank}_{X'(v)}(w) - 1\}$, where $X'(v) = \{x_\lambda^{p_\lambda}, \ldots, x_m^{p_m}\}$ can be determined in $\mathcal{O}(\log |v|)$ time from $\Lambda(v)$. We build the data structure of Lemma 9.4.8 for $A = X'(v)$ so that we can determine $\mathrm{rank}_{X'(v)}(w)$ in $\mathcal{O}(k)$ time. This leaves two possibilities for $\mathrm{rank}_{X(v)}(w)$, i.e., for $\mathrm{MinSuf}(v, w)$. We simply need to compare $s_i w$, $s_{i+1} w$ for these two candidates suffixes $s_i, s_{i+1} \in \Lambda(v)$. Using the augmented suffix array, this takes $\mathcal{O}(k)$ time. Consequently, the query algorithm takes $\mathcal{O}(k)$ time in total. In the preprocessing, we need to construct $\Lambda(v)$ and the data structure of Lemma 9.4.8 for $A = X'(v)$, which takes $\mathcal{O}(\log |v| + |\Lambda(v)|^2) = \mathcal{O}(\log^2 |v|)$ time. The space consumption is also $\mathcal{O}(\log^2 |v|)$. □

9.4.2 $\mathcal{O}(k)$-Time Auxiliary Minimal Suffix Queries

Like in Section 9.3.2, in order to improve the query time in the data structure of Theorem 9.4.6, we simply add a component responsible for computing $\mathrm{MinSuf}(v, w)$ for $|v| \le \tau$ where $\tau = f(f(n)) = o(\frac{\log n}{\log \log n})$.

Again, we partition T into $\lceil \frac{n}{\tau} \rceil$ overlapping blocks T_i of length $m = \mathcal{O}(\tau)$, so that the number of blocks is much larger than the number of order-isomorphism classes of strings of length at most m. Next, we precompute some data for each equivalence class and we reduce a query in T to a query in one of the blocks T_i.

While this approach was easy to apply for computing $\mathrm{MinSuf}(v)$ for a fragment v (with $|v| \le \tau$), it is much more difficult for $\mathrm{MinSuf}(v, w)$ for a fragment v ($|v| \le \tau$) and a k-fragment w. That is because w might be composed of fragments w_j starting in different blocks. As a workaround, we shall replace w by a similar (in a certain sense) k'-fragment of $T_i\$$ (with $k' \le k + 1$) where T_i is a block containing v.

For $0 \le i < \frac{n}{\tau}$, we define $T_i := T[i\tau + 1 \mathinner{.\,.} \min(n, (i + 3)\tau)]$. We determine $\mathrm{oid}(T_i\$)$ for each block using Fact 9.3.7. For a single block T_i per each identifier, we build the augmented suffix array of $T_i\$$, and for all fragments v of $T_i\$$, we construct the set $\Lambda(v)$ along with the data structure of Lemma 9.4.5. In total, this data takes $\mathcal{O}(2^{\mathcal{O}(m \log m)} m^{\mathcal{O}(1)}) = n^{o(1)}$ space and time to construct.

Now, suppose that we are to compute $\mathrm{MinSuf}(v, w)$ where $|v| \le \tau$ and w is a k-fragment of T. We determine the rightmost block T_i containing v. Next, we shall try to represent w as a k-fragment of T_i. We will either succeed or obtain a k'-fragment w' of T_i ($k' \le k$) and a character $c \in \Sigma$ such that $w'c$ matches a prefix of w but does not match any substring of v. In this case, Lemma 9.1.14 states that $\mathrm{MinSuf}(v, w'c)$ can be used to determine two candidates for $\mathrm{MinSuf}(v, w)$.

We decompose $w = w_1 \cdots w_k$ into fragments and process them one by one. Given a fragment w_j, we shall either find an equal fragment of T_i or determine a fragment w'_j of T_i and a character $c \in \Sigma$ such that $w'_j c$ matches a prefix of w_j but not a substring of v. Clearly, if we proceed to w_{j+1} in the first case and terminate in the second case, at the end we either successfully represent w or we find a k'-fragment $w' = w_1 \dots w_{k'-1} w'_{k'}$ satisfying the desired condition. Note that since v is a substring of $T[i\tau + 1 \mathinner{.\,.} \min(n, (i+2)\tau)]$, every substring of v must occur in T at one of the positions in $R_i = \{i\tau+1, \dots, \min(n, (i+2)\tau)\}$. Hence, for each block we build an instance of the data structure of Fact 9.4.7 for suffixes of T starting in R_i. Given w_j, this lets us determine a position $\ell \in R_i$ such that $d_j = \mathrm{lcp}(T[\ell \mathinner{.\,.}], w_j)$ is largest possible. If $d_j = |w_j|$ and $d_j \le \tau$, we have found w_j occurring as a substring of T_i. Otherwise, we set $w'_j \cong w_j[1 \mathinner{.\,.} \min(d_j, \tau)]$, which is a substring of T_i, and $c = w_j[|w'_j| + 1]$. Clearly, $w'_j c$ is a prefix of w_j, so we shall only prove that it is not a substring of v. If $d_j \ge \tau$, then simply $|w'_j c| > \tau \ge |v|$. Otherwise, by the choice of ℓ maximizing $d_j = \mathrm{lcp}(T[\ell \mathinner{.\,.}], w_j)$ among $\ell \in R_i$, the string $w'_j c$ cannot occur at any position in R_i and, in particular, it cannot be a substring of v.

If the described procedure succeeds in finding a k-fragment of T_i equal to w, we simply apply the data structure of Lemma 9.4.5 built for v to determine $\mathrm{MinSuf}(v, w)$ in $\mathcal{O}(k)$ time. Thus, we may assume that this is not the case and it returns a k'-fragment w' and a character c. As already mentioned, having computed $\mathrm{MinSuf}(v, w'c)$, we can determine $\mathrm{MinSuf}(v, w)$ just by comparing the two candidates with the augmented suffix array. If c occurs in T_i, then $w'c$ is a $(k'+1)$-fragment of T_i and we may use Lemma 9.4.5 to compute $\mathrm{MinSuf}(v, w'c)$. Otherwise, we replace c by its successor among letters occurring in $T_i\$$. The successor can be computed in constant time provided that for each block we store a fusion tree of all characters occurring in $T_i\$$ (mapping each character to an arbitrary position where is occurs). To see that replacing c by its successor c' does not change the answer, observe that Lemma 9.1.10 expresses $\mathrm{MinSuf}(v, w'c)$ in terms of $\mathrm{rank}_{X(v)}(w'c)$. The ranks $\mathrm{rank}_{X(v)}(w'c)$ and $\mathrm{rank}_{X(v)}(w'c')$ are equal because $X(v)$ consists of infinite strings composed of characters of v (which are automatically present in T_i), and the modification increases to c' the character c (not present in T_i) at the last position of $w'c$.

Theorem 9.4.9. *For every text T of length n and every parameter $\tau = o(\frac{\log n}{\log\log n})$, there exists a data structure of size $\mathcal{O}(n)$ which answers* Auxiliary Minimal Suffix Queries *in $\mathcal{O}(k)$ time if $|v| \le \tau$. The data structure can be constructed in $\mathcal{O}(n)$ time.*

This was the last missing ingredient necessary for the strongest result of this chapter.

Theorem 9.4.10. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which answers* Auxiliary Minimal Suffix Queries *in $\mathcal{O}(k)$ time and* Generalized Minimal Suffix Queries *in $\mathcal{O}(k^2)$ time. The data structure can be constructed in $\mathcal{O}(n)$ time.*

9.5 Applications

As already noted in [11], Minimal Suffix Queries can be used to compute the Lyndon factorization of a fragment. For fragments of T, and in general k-fragments of T with $k = \mathcal{O}(1)$, we obtain an optimal solution:

Corollary 9.5.1. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which determines the Lyndon factorization $v = v_1^{q_1} \dots v_m^{q_m}$ of a given k-fragment v of T in $\mathcal{O}(k^2 m)$ time. The data structure can be constructed in $\mathcal{O}(n)$ time.*

Our main motivation for introducing Generalized Minimal Suffix Queries, however, was to answer Minimal Rotation Queries, for which we obtain constant query time after linear-time preprocessing. This is achieved using the following observation:

Fact 9.5.2 (see [48]). *The lexicographically smallest cyclic rotation of v is the prefix of* $\mathrm{MinSuf}(v, v)$ *of length* $|v|$.

Proof. Note that the rotations of v are precisely the length-$|v|$ prefixes of strings sv, where s is a suffix of v. Consequently, the lexicographic order of rotations coincides with the lexicographic order of the strings sv; the characters following the length-$|v|$ prefixes only serve as a tie-breaker. □

Due to Fact 9.5.2, Minimal Rotation Queries trivially reduce to Auxiliary Minimal Suffix Queries.

Theorem 9.5.3. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which given determines the lexicographically smallest cyclic rotation of a given k-fragment v of the text T in $\mathcal{O}(k^2)$ time. The data structure can be constructed in $\mathcal{O}(n)$ time.*

The minimal rotation can be used as a canonical representation of a string in its cyclic equivalence class. Consequently, one can verify the cyclic equivalence of two fragments of T by first computing their minimal rotations, which can be represented as 2-fragments, and then by checking whether these 2-fragments match (using Observation 9.4.1). Thus, we can answer the following Cyclic Equivalence Queries in $\mathcal{O}(1)$ time.

Cyclic Equivalence Queries
Given two fragment x and y of the text T, decide whether x and y are cyclically equivalent.

The same approach generalizes to arbitrary k-fragments.

Corollary 9.5.4. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which given two k-fragments x and y of T, checks in $\mathcal{O}(k^2)$ time whether x and y are cyclically equivalent. The data structure can be constructed in $\mathcal{O}(n)$ time.*

Finally, we note that due to Observation 9.1.3(e), queries for maximal suffixes can be answered using the data structure of Theorem 9.4.10 constructed for the text $T\$$ with the alphabet order reversed.

Corollary 9.5.5. *For every text T of length n, there exists a data structure of size $\mathcal{O}(n)$ which answers* Auxiliary Maximal Suffix Queries *in $\mathcal{O}(k)$ time and* Generalized Maximal Suffix Queries *in $\mathcal{O}(k^2)$ time. The data structure can be constructed in $\mathcal{O}(n)$ time.*

Chapter 10

Final Remarks

In this dissertation, we have presented efficient data structures for several types of internal queries in texts. Many of our solutions are optimal in the standard setting of texts over polynomially-bounded integer alphabets. Nevertheless, tight complexity bounds for the remaining problems are yet to be settled; this happens to be the case whenever we apply orthogonal range queries. In particular, the wavelet suffix trees handle Substring Suffix Rank Queries and Substring Suffix Selection Queries in $\mathcal{O}(\log n)$ time compared to the lower bound of $\Omega(\log n/\log\log n)$ inherited from range rank and selection (tight for these range queries). A natural open question is whether this is an indication of the limitations of our tools or of the inherent difficulty of the internal queries studied. The gap is even bigger for Bounded Longest Common Prefix Queries and all kinds of Substring Compression Queries, for which no lower bounds are known, even conditioned on the hardness of range searching problems.

The optimality of $\mathcal{O}(\log n)$ query time for Period Queries, on the other hand, follows from the fact that we ask for a representation of all periods of the specified fragment. This argument is not valid for a related natural question of computing the shortest period only. Yet, we do not know how to improve the worst-case query time compared to the base variant. Perhaps one can prove some lower bound for this restricted version. The situation with Internal Pattern Matching Queries is somewhat similar: We obtain $\mathcal{O}(|y|/|x|)$ query time if we drop the assumption that $|y| < 2|x|$. This is optimal in the worst-case (due to an information-theoretic lower bound on the output size), but one may be interested in a restricted output (the leftmost occurrence only, for example) or an output-sensitive query time. It is not clear to what extent (if any) our techniques surpass orthogonal range searching in these scenarios.

Another research direction is to further investigate the applicability of our data structures. In particular, we have not yet explored the consequences of the most recent result contained in this thesis—the optimal solution for Longest Common Extension Queries in texts over small alphabets. A growing number of algorithms using Prefix-Suffix Queries [3, 5, 7, 63, 87, 91] also indicates a potential for new applications of the less established query types. However, conducting a systematic study is difficult because the existing work is not formulated in terms of these recently introduced problems. For the same reason, it is possible that an important and useful type of internal queries is still to be discovered.

The contribution of this dissertation is not only in the main theorems but also in a few novel techniques. In particular, our implementation of local consistency seems to be a powerful tool. It is likely to be useful for other problems on texts over small alphabets, including fundamental tasks such as the efficient construction of text indexing data structures (see [114] for recent developments). At the same time, further research should indicate whether our formalization in terms of synchronizing functions can be replaced by a more elegant one. A natural specific goal is to design a more versatile way of handling periodic fragments so that they could be captured automatically rather than being extracted from maximal repetitions. Currently, this drawback seems to hinder adaptability to some settings where locally consistent parsing schemes turned out to be very successful.

A wide variety of existing results for LCE Queries suggests many possible scenarios where other internal queries can be studied as well. The best-established ones involve grammar-compressed and dynamic texts. Other interesting questions concern optimality for small alphabets (which we have studied for LCE Queries only) and the existence of time-space trade-offs with limited additional memory (on top of the text stored in a random-access read-only memory).

Bibliography

[1] P. Abedin, A. Ganguly, W. Hon, Y. Nekrich, K. Sadakane, R. Shah, and S. V. Thankachan. A linear-space data structure for range-LCP queries in poly-logarithmic time. In L. Wang and D. Zhu, editors, *Computing and Combinatorics, COCOON 2018*, volume 10976 of *LNCS*, pages 615–625. Springer, 2018. DOI: 10.1007/978-3-319-94776-1_51 (cited on page 10).

[2] S. Alstrup, G. S. Brodal, and T. Rauhe. Pattern matching in dynamic texts. In D. B. Shmoys, editor, *11th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2000*, pages 819–828. ACM/SIAM, 2000. URL: http://dl.acm.org/citation.cfm?id=338219.338645 (cited on pages 4, 32).

[3] A. Amir, M. Amit, G. M. Landau, and D. Sokol. Period recovery of strings over the Hamming and edit distances. *Theoretical Computer Science*, 710:2–18, 2018. DOI: 10.1016/j.tcs.2017.10.026 (cited on pages 6, 99).

[4] A. Amir, A. Apostolico, G. M. Landau, A. Levy, M. Lewenstein, and E. Porat. Range LCP. *Journal of Computer and System Sciences*, 80(7):1245–1253, 2014. DOI: 10.1016/j.jcss.2014.02.010 (cited on page 10).

[5] A. Amir, P. Charalampopoulos, S. P. Pissis, and J. Radoszewski. Longest common factor made fully dynamic, 2018. arXiv: 1804.08731 (cited on pages 5, 99).

[6] A. Amir, G. M. Landau, M. Lewenstein, and D. Sokol. Dynamic text and static pattern matching. *ACM Transactions on Algorithms*, 3(2):19, 2007. DOI: 10.1145/1240233.1240242 (cited on page 10).

[7] A. Amir, G. M. Landau, S. Marcus, and D. Sokol. Two-dimensional maximal repetitions. In Y. Azar, H. Bast, and G. Herman, editors, *Algorithms, ESA 2018*, volume 112 of *LIPIcs*, 2:1–2:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018. DOI: 10.4230/LIPIcs.ESA.2018.2 (cited on pages 6, 99).

[8] A. Amir, M. Lewenstein, and S. V. Thankachan. Range LCP queries revisited. In C. S. Iliopoulos, S. J. Puglisi, and E. Yilmaz, editors, *String Processing and Information Retrieval, SPIRE 2015*, volume 9309 of *LNCS*, pages 350–361. Springer, 2015. DOI: 10.1007/978-3-319-23826-5_33 (cited on page 10).

[9] A. Apostolico and M. Crochemore. Fast parallel Lyndon factorization with applications. *Theory of Computing Systems*, 28(2):89–108, 1995. DOI: 10.1007/BF01191471 (cited on page 82).

[10] A. Apostolico and M. Crochemore. Optimal canonization of all substrings of a string. *Information and Computation*, 95(1):76–95, 1991. DOI: 10.1016/0890-5401(91)90016-U (cited on page 9).

[11] M. Babenko, P. Gawrychowski, T. Kociumaka, I. Kolesnichenko, and T. Starikovskaya. Computing minimal and maximal suffixes of a substring. *Theoretical Computer Science*, 638:112–121, 2016. DOI: `10.1016/j.tcs.2015.08.023` (cited on pages 9, 87, 88, 96).

[12] M. Babenko, P. Gawrychowski, T. Kociumaka, and T. Starikovskaya. Wavelet trees meet suffix trees. In P. Indyk, editor, *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 572–591. SIAM, 2015. DOI: `10.1137/1.9781611973730.39` (cited on pages 13, 14, 25–28).

[13] M. Babenko, I. Kolesnichenko, and T. Starikovskaya. On minimal and maximal suffixes of a substring. In J. Fischer and P. Sanders, editors, *Combinatorial Pattern Matching, CPM 2013*, volume 7922 of *LNCS*, pages 28–37. Springer, 2013. DOI: `10.1007/978-3-642-38905-4_5` (cited on page 9).

[14] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. The "runs" theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. DOI: `10.1137/15M1011032` (cited on pages 2, 6, 12, 18).

[15] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972. DOI: `10.1007/bf00288683` (cited on page 23).

[16] D. Belazzougui and S. J. Puglisi. Range predecessor and Lempel-Ziv parsing. In R. Krauthgamer, editor, *27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, pages 2053–2071. SIAM, 2016. DOI: `10.1137/1.9781611974331.ch143` (cited on pages 5, 12, 13, 29).

[17] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005. DOI: `10.1016/j.jalgor.2005.08.001` (cited on pages 3, 29).

[18] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979. DOI: `10.1016/0020-0190(79)90117-0` (cited on page 12).

[19] P. Bille, A. R. Christiansen, P. H. Cording, and I. L. Gørtz. Finger search in grammar-compressed strings. *Theory of Computing Systems*, 62(8):1715–1735, 2018. DOI: `10.1007/s00224-017-9839-9` (cited on page 4).

[20] P. Bille, J. Fischer, I. L. Gørtz, T. Kopelowitz, B. Sach, and H. W. Vildhøj. Sparse text indexing in small space. *ACM Transactions on Algorithms*, 12(3):39:1–39:19, 2016. DOI: `10.1145/2836166` (cited on pages 4, 31).

[21] P. Bille, I. L. Gørtz, P. H. Cording, B. Sach, H. W. Vildhøj, and S. Vind. Fingerprints in compressed strings. *Journal of Computer and System Sciences*, 86:171–180, 2017. DOI: `10.1016/j.jcss.2017.01.002` (cited on page 4).

[22] P. Bille, I. L. Gørtz, M. B. T. Knudsen, M. Lewenstein, and H. W. Vildhøj. Longest common extensions in sublinear space. In F. Cicalese, E. Porat, and U. Vaccaro, editors, *Combinatorial Pattern Matching, CPM 2015*, volume 9133 of *LNCS*, pages 65–76. Springer, 2015. DOI: `10.1007/978-3-319-19929-0_6` (cited on pages 4, 31).

[23] P. Bille, I. L. Gørtz, B. Sach, and H. W. Vildhøj. Time-space trade-offs for longest common extensions. *Journal of Discrete Algorithms*, 25:42–50, 2014. DOI: `10.1016/j.jda.2013.06.003` (cited on pages 4, 31).

[24] O. Birenzwige, S. Golan, and E. Porat. Locally consistent parsing for text indexing in small space. Unpublished manuscript, 2017 (cited on pages iii, 4, 12, 31, 32).

[25] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11:303–320, 1980. DOI: `10.1016/0304-3975(80)90018-3` (cited on page 76).

[26] K. S. Booth. Lexicographically least circular substrings. *Information Processing Letters*, 10(4–5):240–242, 1980. DOI: `10.1016/0020-0190(80)90149-0` (cited on page 9).

[27] D. Breslauer and Z. Galil. Finding all periods and initial palindromes of a string in parallel. *Algorithmica*, 14(4):355–366, 1995. DOI: `10.1007/BF01294132` (cited on pages 6, 46).

[28] G. S. Brodal, B. Gfeller, A. G. Jørgensen, and P. Sanders. Towards optimal range medians. *Theoretical Computer Science*, 412(24):2588–2601, 2011. DOI: `10.1016/j.tcs.2010.05.003` (cited on page 28).

[29] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report 124, Digital Equipment Corporation, Palo Alto, California, 1994. URL: `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf` (cited on pages 7, 18, 19).

[30] S. Butakov and V. Scherbinin. The toolbox for local and global plagiarism detection. *Computers & Education*, 52(4):781–788, 2009. DOI: `10.1016/j.compedu.2008.12.001` (cited on page 11).

[31] T. M. Chan, K. G. Larsen, and M. Pătraşcu. Orthogonal range searching on the RAM, revisited. In F. Hurtado and M. J. van Kreveld, editors, *27th ACM Symposium on Computational Geometry, SoCG 2011*, pages 1–10. ACM, 2011. DOI: `10.1145/1998196.1998198` (cited on page 12).

[32] T. M. Chan and M. Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In M. Charikar, editor, *21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*, pages 161–173. SIAM, 2010. DOI: `10.1137/1.9781611973075.15` (cited on page 28).

[33] P. Charalampopoulos, M. Crochemore, C. S. Iliopoulos, T. Kociumaka, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Waleń. Efficient enumeration of non-equivalent squares in partial words with few holes. *Journal of Combinatorial Optimization*, 2018. DOI: `10.1007/s10878-018-0300-z` (cited on page 10).

[34] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988. DOI: `10.1137/0217026` (cited on page 12).

[35] K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, IV. The quotient groups of the lower central series. *Annals of Mathematics*, 68(1):81–95, 1958. DOI: `10.2307/1970044` (cited on pages 10, 12, 17, 82).

[36] D. Clark. *Compact Pat trees*. PhD thesis, University of Waterloo, 1996. URL: `http://hdl.handle.net/10012/64` (cited on page 25).

[37] F. Claude, P. K. Nicholson, and D. Seco. Space efficient wavelet tree construction. In R. Grossi, F. Sebastiani, and F. Silvestri, editors, *String Processing and Information Retrieval, SPIRE 2011*, volume 7024 of *LNCS*, pages 185–196. Springer, 2011. DOI: `10.1007/978-3-642-24583-1_19` (cited on page 26).

[38] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986. DOI: `10.1016/S0019-9958(86)80023-7` (cited on page 11).

[39] G. Cormode and S. Muthukrishnan. Substring compression problems. In *16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005*, pages 321–330. SIAM, 2005. URL: `http://dl.acm.org/citation.cfm?id=1070432.1070478` (cited on pages 7, 10, 11, 18).

[40] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Transactions on Algorithms*, 3(1):2:1–2:19, 2007. DOI: `10.1145/1219944.1219947` (cited on page 11).

[41] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on strings*. Cambridge University Press, 2007. DOI: `10.1017/cbo9780511546853` (cited on pages 1, 15, 16, 20).

[42] M. Crochemore and L. Ilie. Computing Longest Previous Factor in linear time and applications. *Information Processing Letters*, 106(2):75–80, 2008. DOI: `10.1016/j.ipl.2007.10.006` (cited on page 18).

[43] M. Crochemore, L. Ilie, and W. Rytter. Repetitions in strings: Algorithms and combinatorics. *Theoretical Computer Science*, 410(50):5227–5235, 2009. DOI: `10.1016/j.tcs.2009.08.024` (cited on page 44).

[44] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, R. Kundu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Waleń. Near-optimal computation of runs over general alphabet via non-crossing LCE queries. In S. Inenaga, K. Sadakane, and T. Sakai, editors, *String Processing and Information Retrieval, SPIRE 2016*, volume 9954 of *LNCS*, pages 22–34, 2016. DOI: `10.1007/978-3-319-46049-9_3` (cited on page 4).

[45] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń. Extracting powers and periods in a word from its runs structure. *Theoretical Computer Science*, 521:29–41, 2014. DOI: `10.1016/j.tcs.2013.11.018` (cited on pages 6, 12, 46, 47).

[46] M. Crochemore, C. S. Iliopoulos, M. Kubica, M. S. Rahman, G. Tischler, and T. Waleń. Improved algorithms for the range next value problem and applications. *Theoretical Computer Science*, 434:23–34, 2012. DOI: `10.1016/j.tcs.2012.02.015` (cited on pages 5, 28, 29).

[47] M. Crochemore and D. Perrin. Two-way string-matching. *Journal of the ACM*, 38(3):650–674, 1991. DOI: `10.1145/116825.116845` (cited on page 12).

[48] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2003. DOI: `10.1142/4838` (cited on pages 1, 15, 20, 75, 97).

[49] M. Crochemore and W. Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13(5):405–425, 1995. DOI: `10.1007/BF01190846` (cited on page 44).

[50] J. W. Daykin, C. S. Iliopoulos, and W. F. Smyth. Parallel RAM algorithms for factorizing words. *Theoretical Computer Science*, 127(1):53–67, 1994. DOI: 10.1016/0304-3975(94)90100-7 (cited on page 82).

[51] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz. KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015. DOI: 10.1093/bioinformatics/btv022 (cited on page 11).

[52] J.-P. Duval. Factorizing words over an ordered alphabet. *Journal of Algorithms*, 4(4):363–381, 1983. DOI: 10.1016/0196-6774(83)90017-2 (cited on pages 9, 12, 82).

[53] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000. DOI: 10.1145/355541.355547 (cited on pages 2, 3, 19, 20).

[54] M. Farach-Colton and S. Muthukrishnan. Perfect hashing for strings: formalization and algorithms. In D. S. Hirschberg and E. W. Myers, editors, *Combinatorial Pattern Matching, CPM 1996*, volume 1075 of *LNCS*, pages 130–140. Springer, 1996. DOI: 10.1007/3-540-61258-0_11 (cited on page 10).

[55] S. Faro and T. Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Computing Surveys*, 45(2):13:1–13:42, 2013. DOI: 10.1145/2431211.2431212 (cited on page 1).

[56] N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. DOI: 10.2307/2034009 (cited on page 16).

[57] J. Fischer, T. I, and D. Köppl. Deterministic sparse suffix sorting on rewritable texts. In E. Kranakis, G. Navarro, and E. Chávez, editors, *Latin American Symposium on Theoretical Informatics, LATIN 2016*, volume 9644 of *LNCS*, pages 483–496. Springer, 2016. DOI: 10.1007/978-3-662-49529-2_36 (cited on page 4).

[58] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006. DOI: 10.1145/1198513.1198521 (cited on page 27).

[59] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993. DOI: 10.1016/0022-0000(93)90040-4 (cited on pages 13, 21, 23).

[60] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986. DOI: 10.1145/8307.8309 (cited on page 3).

[61] M. Gańczorz, P. Gawrychowski, A. Jeż, and T. Kociumaka. Edit distance with block operations. In Y. Azar, H. Bast, and G. Herman, editors, *Algorithms, ESA 2018*, volume 112 of *LIPIcs*, 33:1–33:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018. DOI: 10.4230/LIPIcs.ESA.2018.33 (cited on page 11).

[62] P. Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic. In C. Demetrescu and M. M. Halldórsson, editors, *Algorithms, ESA 2011*, volume 6942 of *LNCS*, pages 421–432. Springer, 2011. DOI: 10.1007/978-3-642-23719-5_36 (cited on page 10).

[63] P. Gawrychowski, T. I, S. Inenaga, D. Köppl, and F. Manea. Tighter bounds and optimal algorithms for all maximal α-gapped repeats and palindromes: finding all maximal α-gapped repeats and palindromes in optimal worst case time on integer alphabets. *Theory of Computing Systems*, 62(1):162–191, 2018. DOI: `10.1007/s00224-017-9794-5` (cited on pages 6, 99).

[64] P. Gawrychowski, A. Karczmarz, T. Kociumaka, J. Łącki, and P. Sankowski. Optimal dynamic strings. In A. Czumaj, editor, *29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1509–1528. SIAM, 2018. DOI: `10.1137/1.9781611975031.99` (cited on pages 4, 11, 32).

[65] P. Gawrychowski and T. Kociumaka. Sparse suffix tree construction in optimal time and space. In P. N. Klein, editor, *28th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017*, pages 425–439. SIAM, 2017. DOI: `10.1137/1.9781611974782.27` (cited on pages 4, 31).

[66] P. Gawrychowski, T. Kociumaka, W. Rytter, and T. Waleń. Faster longest common extension queries in strings over general alphabets. In R. Grossi and M. Lewenstein, editors, *Combinatorial Pattern Matching, CPM 2016*, volume 54 of *LIPIcs*, 5:1–5:13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. DOI: `10.4230/LIPIcs.CPM.2016.5` (cited on page 4).

[67] P. Gawrychowski, M. Lewenstein, and P. K. Nicholson. Weighted ancestors in suffix trees. In A. S. Schulz and D. Wagner, editors, *Algorithms, ESA 2014*, volume 8737 of *LNCS*, pages 455–466. Springer, 2014. DOI: `10.1007/978-3-662-44777-2_38` (cited on page 10).

[68] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *14th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2003*, pages 841–850. SIAM, 2003. URL: `http://dl.acm.org/citation.cfm?id=644108.644250` (cited on pages 8, 13, 25).

[69] R. Grossi and G. Ottaviano. The wavelet trie: maintaining an indexed sequence of strings in compressed space. In M. Benedikt, M. Krötzsch, and M. Lenzerini, editors, *31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012*, pages 203–214. ACM, 2012. DOI: `10.1145/2213556.2213586` (cited on page 27).

[70] L. J. Guibas and A. M. Odlyzko. Periods in strings. *Journal of Combinatorial Theory, Series A*, 30(1):19–42, 1981. DOI: `10.1016/0097-3165(81)90038-8` (cited on page 5).

[71] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. DOI: `10.1017/cbo9780511574931` (cited on pages 1, 15).

[72] T. Hagerup. Sorting and searching on the word RAM. In M. Morvan, C. Meinel, and D. Krob, editors, *Symposium on Theoretical Aspects of Computer Science, STACS 1998*, volume 1373 of *LNCS*, pages 366–398. Springer, 1998. DOI: `10.1007/BFb0028575` (cited on page 21).

[73] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984. DOI: `10.1137/0213024` (cited on pages 3, 29).

[74] T. I. Longest common extensions with recompression. In J. Kärkkäinen, J. Radoszewski, and W. Rytter, editors, *Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPIcs*, 18:1–18:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. DOI: `10.4230/LIPIcs.CPM.2017.18` (cited on pages 4, 32).

[75] T. I, W. Matsubara, K. Shimohira, S. Inenaga, H. Bannai, M. Takeda, K. Narisawa, and A. Shinohara. Detecting regularities on grammar-compressed strings. *Information and Computation*, 240:74–89, 2015. DOI: `10.1016/j.ic.2014.09.009` (cited on page 4).

[76] T. I, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. *Theoretical Computer Science*, 656:215–224, 2016. DOI: `10.1016/j.tcs.2016.03.005` (cited on pages 12, 81, 83, 84).

[77] G. Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, FOCS 1989*, pages 549–554. IEEE Computer Society, 1989. DOI: `10.1109/SFCS.1989.63533` (cited on page 25).

[78] J. JáJá, C. W. Mortensen, and Q. Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In R. Fleischer and G. Trippen, editors, *Algorithms and Computation, ISAAC 2004*, volume 3341 of *LNCS*, pages 558–568. Springer, 2004. DOI: `10.1007/978-3-540-30551-4_49` (cited on page 28).

[79] A. Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms*, 11(3):20:1–20:43, 2015. DOI: `10.1145/2631920` (cited on page 11).

[80] A. Jeż. Recompression: A simple and powerful technique for word equations. *Journal of the ACM*, 63(1):4:1–4:51, 2016. DOI: `10.1145/2743014` (cited on page 11).

[81] A. G. Jørgensen and K. G. Larsen. Range selection and median: Tight cell probe lower bounds and adaptive data structures. In D. Randall, editor, *22nd Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011*, pages 805–813. SIAM, 2011. DOI: `10.1137/1.9781611973082.63` (cited on page 28).

[82] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006. DOI: `10.1145/1217856.1217858` (cited on pages 2, 19).

[83] O. Keller, T. Kopelowitz, S. Landau Feibish, and M. Lewenstein. Generalized substring compression. *Theoretical Computer Science*, 525:42–54, 2014. DOI: `10.1016/j.tcs.2013.10.010` (cited on pages 6, 7, 10, 18, 64–66).

[84] J. Kim, P. Eades, R. Fleischer, S. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama. Order-preserving matching. *Theoretical Computer Science*, 525:68–79, 2014. DOI: `10.1016/j.tcs.2013.10.006` (cited on pages 81, 91).

[85] D. E. Knuth, J. H. Morris Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. DOI: `10.1137/0206024` (cited on pages 5, 22).

[86] T. Kociumaka. Minimal suffix and rotation of a substring in optimal time. In R. Grossi and M. Lewenstein, editors, *Combinatorial Pattern Matching, CPM 2016*, volume 54 of *LIPIcs*, 28:1–28:12. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. DOI: `10.4230/LIPIcs.CPM.2016.28` (cited on page 14).

[87] T. Kociumaka, R. Kundu, M. Mohamed, and S. P. Pissis. Longest unbordered factor in quasilinear time. In W.-L. Hsu, D.-T. Lee, and C.-S. Liao, editors, *Algorithms and Computation, ISAAC 2018*, LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018. arXiv: `1805.09924` (cited on pages 5, 99).

[88] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Waleń. Efficient data structures for the factor periodicity problem. In L. Calderón-Benavides, C. N. González-Caro, E. Chávez, and N. Ziviani, editors, *String Processing and Information Retrieval, SPIRE 2012*, volume 7608 of *LNCS*, pages 284–294. Springer, 2012. DOI: `10.1007/978-3-642-34109-0_30` (cited on page 5).

[89] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Waleń. Internal pattern matching queries in a text and applications. In P. Indyk, editor, *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551. SIAM, 2015. DOI: `10.1137/1.9781611973730.36` (cited on page 14).

[90] R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS 1999*, pages 596–604. IEEE Computer Society, 1999. DOI: `10.1109/SFFCS.1999.814634` (cited on pages 12, 17, 18).

[91] R. Kolpakov, M. Podolskiy, M. Posypkin, and N. Khrapov. Searching of gapped repeats and subrepetitions in a word. *Journal of Discrete Algorithms*, 46–47:1–15, 2017. DOI: `10.1016/j.jda.2017.10.004` (cited on pages 6, 99).

[92] T. Kopelowitz, G. Kucherov, Y. Nekrich, and T. Starikovskaya. Cross-document pattern matching. *Journal of Discrete Algorithms*, 24:40–47, 2014. DOI: `10.1016/j.jda.2013.05.002` (cited on page 10).

[93] D. Kosolobov. Computing runs on a general alphabet. *Information Processing Letters*, 116(3):241–244, 2016. DOI: `10.1016/j.ipl.2015.11.016` (cited on page 4).

[94] D. Kosolobov. Tight lower bounds for the longest common extension problem. *Information Processing Letters*, 125:26–29, 2017. DOI: `10.1016/j.ipl.2017.05.003` (cited on page 4).

[95] D. Kosolobov, F. Manea, and D. Nowotka. Detecting one-variable patterns. In G. Fici, M. Sciortino, and R. Venturini, editors, *String Processing and Information Retrieval, SPIRE 2017*, volume 10508 of *LNCS*, pages 254–270. Springer, 2017. DOI: `10.1007/978-3-319-67428-5_22` (cited on page 6).

[96] M. Kubica, T. Kulczyński, J. Radoszewski, W. Rytter, and T. Waleń. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 113(12):430–433, 2013. DOI: `10.1016/j.ipl.2013.03.015` (cited on pages 81, 91).

[97] S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In E. Chávez and S. Lonardi, editors, *String Processing and Information Retrieval, SPIRE 2010*, volume 6393 of *LNCS*, pages 201–206. Springer, 2010. DOI: `10.1007/978-3-642-16321-0_20` (cited on page 18).

[98] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989. DOI: `10.1016/0196-6774(89)90010-2` (cited on page 3).

[99] G. M. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988. DOI: 10.1016/0022-0000(88)90045-1 (cited on pages 2, 3, 19).

[100] M. Lewenstein. Orthogonal range searching for text indexing. In A. Brodnik, A. López-Ortiz, V. Raman, and A. Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *LNCS*, pages 267–302. Springer, 2013. DOI: 10.1007/978-3-642-40273-9_18 (cited on page 12).

[101] H. Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018. DOI: 10.1093/bioinformatics/bty191 (cited on page 11).

[102] M. Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012. DOI: 10.1515/gcc-2012-0016 (cited on page 1).

[103] M. Lothaire. *Combinatorics on Words*. Cambridge Mathematical Library. Cambridge University Press, second edition, 1997. DOI: 10.1017/CBO9780511566097 (cited on pages 15, 82).

[104] R. C. Lyndon. On Burnside's problem. *Transactions of the American Mathematical Society*, 77(2):202–215, 1954. DOI: 10.1090/S0002-9947-1954-0064049-X (cited on pages 12, 17).

[105] R. C. Lyndon and M.-P. Schützenberger. The equation $a^M = b^N c^P$ in a free group. *The Michigan Mathematical Journal*, 9(4):289–298, 1962. DOI: 10.1307/mmj/1028998766 (cited on page 16).

[106] M. G. Main. Detecting leftmost maximal periodicities. *Discrete Applied Mathematics*, 25(1–2):145–153, 1989. DOI: 10.1016/0166-218X(89)90051-6 (cited on pages 12, 17).

[107] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. DOI: 10.1137/0222058 (cited on pages 1, 3, 8, 19).

[108] G. Marçais, D. F. DeBlasio, and C. Kingsford. Asymptotically optimal minimizers schemes. *Bioinformatics*, 34(13):i13–i22, 2018. DOI: 10.1093/bioinformatics/bty258 (cited on page 11).

[109] G. Marçais, D. Pellow, D. Bork, Y. Orenstein, R. Shamir, and C. Kingsford. Improving the performance of minimizers and winnowing schemes. *Bioinformatics*, 33(14):i110–i117, 2017. DOI: 10.1093/bioinformatics/btx235 (cited on page 11).

[110] K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997. DOI: 10.1007/BF02522825 (cited on pages 4, 11).

[111] J. H. Morris Jr. and V. R. Pratt. A linear pattern-matching algorithm. Technical report 40, Department of Computer Science, University of California, Berkeley, 1970 (cited on pages 1, 5).

[112] M. Mucha. Lyndon words and short superstrings. In S. Khanna, editor, *24th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013*, pages 958–972. SIAM, 2013. DOI: 10.1137/1.9781611973105.69 (cited on page 12).

[113] J. I. Munro. Tables. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science, FSTTCS 1996*, volume 1180 of *LNCS*, pages 37–42. Springer, 1996. DOI: 10.1007/3-540-62034-6_35 (cited on page 25).

[114] J. I. Munro, G. Navarro, and Y. Nekrich. Text indexing and searching in sublinear time, 2017. arXiv: 1712.07431 (cited on pages 4, 100).

[115] J. I. Munro, Y. Nekrich, and J. S. Vitter. Fast construction of wavelet trees. *Theoretical Computer Science*, 638:91–97, 2016. DOI: 10.1016/j.tcs.2015.11.011 (cited on pages 25, 27, 28).

[116] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001. DOI: 10.1145/375360.375365 (cited on page 1).

[117] G. Navarro. *Compact Data Structures: A Practical Approach.* Cambridge University Press, 2016. DOI: 10.1017/cbo9781316588284 (cited on page 2).

[118] G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014. DOI: 10.1016/j.jda.2013.07.004 (cited on page 25).

[119] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007. DOI: 10.1145/1216370.1216372 (cited on page 1).

[120] Y. Nekrich and G. Navarro. Sorted range reporting. In F. V. Fomin and P. Kaski, editors, *Algorithm Theory, SWAT 2012*, volume 7357 of *LNCS*, pages 271–282. Springer, 2012. DOI: 10.1007/978-3-642-31155-0_24 (cited on pages 5, 28).

[121] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Fully dynamic data structure for LCE queries in compressed space. In P. Faliszewski, A. Muscholl, and R. Niedermeier, editors, *Mathematical Foundations of Computer Science, MFCS 2016*, volume 58 of *LIPIcs*, 72:1–72:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. DOI: 10.4230/LIPIcs.MFCS.2016.72 (cited on pages 4, 32).

[122] Y. Orenstein, D. Pellow, G. Marçais, R. Shamir, and C. Kingsford. Designing small universal k-mer hitting sets for improved analysis of high-throughput sequencing. *PLoS Computational Biology*, 13(10), 2017. DOI: 10.1371/journal.pcbi.1005777 (cited on page 11).

[123] M. Patil, R. Shah, and S. V. Thankachan. Faster range LCP queries. In O. Kurland, M. Lewenstein, and E. Porat, editors, *String Processing and Information Retrieval, SPIRE 2013*, volume 8214 of *LNCS*, pages 263–270. Springer, 2013. DOI: 10.1007/978-3-319-02432-5_29 (cited on page 10).

[124] M. Pătraşcu. Lower bounds for 2-dimensional range counting. In D. S. Johnson and U. Feige, editors, *39th Annual ACM Symposium on Theory of Computing, STOC 2007*, pages 40–46. ACM, 2007. DOI: 10.1145/1250790.1250797 (cited on page 28).

[125] M. Pătraşcu. Unifying the landscape of cell-probe lower bounds. *SIAM Journal on Computing*, 40(3):827–847, 2011. DOI: 10.1137/09075336X (cited on page 28).

[126] M. Pătraşcu and M. Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014*, pages 166–175. IEEE Computer Society, 2014. DOI: 10.1109/FOCS.2014.26 (cited on pages 13, 23, 81).

[127] W. Plandowski and W. Rytter. Application of Lempel-Ziv encodings to the solution of words equations. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *ICALP 1998*, volume 1443 of *LNCS*, pages 731–742. Springer, 1998. DOI: 10.1007/BFb0055097 (cited on pages 6, 46).

[128] M. Ponec, P. Giura, J. Wein, and H. Brönnimann. New payload attribution methods for network forensic investigations. *ACM Transactions on Information and System Security*, 13(2):15:1–15:32, 2010. DOI: 10.1145/1698750.1698755 (cited on page 11).

[129] N. Prezza. In-place sparse suffix sorting. In A. Czumaj, editor, *29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1496–1508. SIAM, 2018. DOI: 10.1137/1.9781611975031.98 (cited on page 4).

[130] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004. DOI: 10.1093/bioinformatics/bth408 (cited on page 11).

[131] M. Ružić. Constructing efficient dictionaries in close to sorting time. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *Automata, Languages and Programming, ICALP 2008, Part I*, volume 5125 of *LNCS*, pages 84–95. Springer, 2008. DOI: 10.1007/978-3-540-70575-8_8 (cited on pages 13, 58).

[132] S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *37th IEEE Annual Symposium on Foundations of Computer Science, FOCS 1996*, pages 320–328. IEEE Computer Society, 1996. DOI: 10.1109/SFCS.1996.548491 (cited on pages 11, 32).

[133] S. C. Sahinalp and U. Vishkin. On a parallel-algorithms method for string matching problems. In M. A. Bonuccelli, P. Crescenzi, and R. Petreschi, editors, *Algorithms and Complexity, CIAC 1994*, volume 778 of *LNCS*, pages 22–32. Springer, 1994. DOI: 10.1007/3-540-57811-0_3 (cited on pages 11, 32).

[134] S. C. Sahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. In F. T. Leighton and M. T. Goodrich, editors, *26th Annual ACM Symposium on Theory of Computing, STOC 1994*, pages 300–309. ACM, 1994. DOI: 10.1145/195058.195164 (cited on page 11).

[135] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In A. Y. Halevy, Z. G. Ives, and A. Doan, editors, *International Conference on Management of Data, SIGMOD 2003*, pages 76–85. ACM, 2003. DOI: 10.1145/872757.872770 (cited on page 11).

[136] Y. Shiloach. Fast canonization of circular strings. *Journal of Algorithms*, 2(2):107–121, 1981. DOI: 10.1016/0196-6774(81)90013-4 (cited on page 9).

[137] W. F. Smyth. Computing regularities in strings: A survey. *European Journal of Combinatorics*, 34(1):3–14, 2013. DOI: 10.1016/j.ejc.2012.07.010 (cited on page 1).

[138] D. Sorokina, J. Gehrke, S. Warner, and P. Ginsparg. Plagiarism detection in arXiv. In *6th IEEE International Conference on Data Mining, ICDM 2006*, pages 1070–1075. IEEE Computer Society, 2006. DOI: 10.1109/ICDM.2006.126 (cited on page 11).

[139] Y. Tanimura, T. I, H. Bannai, S. Inenaga, S. J. Puglisi, and M. Takeda. Deterministic sub-linear space LCE data structures with efficient construction. In R. Grossi and M. Lewenstein, editors, *Combinatorial Pattern Matching, CPM 2016*, volume 54 of *LIPIcs*, 1:1–1:10. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. DOI: `10.4230/LIPIcs.CPM.2016.1` (cited on pages 4, 31).

[140] Y. Tanimura, T. Nishimoto, H. Bannai, S. Inenaga, and M. Takeda. Small-space LCE data structure with constant-time queries. In K. G. Larsen, H. L. Bodlaender, and J. Raskin, editors, *Mathematical Foundations of Computer Science, MFCS 2017*, volume 83 of *LIPIcs*, 10:1–10:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. DOI: `10.4230/LIPIcs.MFCS.2017.10` (cited on pages 4, 32).

[141] G. Tischler. On wavelet tree construction. In R. Giancarlo and G. Manzini, editors, *Combinatorial Pattern Matching, CPM 2011*, volume 6661 of *LNCS*, pages 208–218. Springer, 2011. DOI: `10.1007/978-3-642-21458-5_19` (cited on page 26).

[142] P. Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, SWAT 1973*, pages 1–11. IEEE Computer Society, 1973. DOI: `10.1109/SWAT.1973.13` (cited on pages 1, 3, 8, 20).

[143] D. E. Wood and S. L. Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15(3):R46, 2014. DOI: `10.1186/gb-2014-15-3-r46` (cited on page 11).

[144] G. Zhou. Two-dimensional range successor in optimal time and almost linear space. *Information Processing Letters*, 116(2):171–174, 2016. DOI: `10.1016/j.ipl.2015.09.002` (cited on pages 5, 28, 29).

[145] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. DOI: `10.1109/TIT.1977.1055714` (cited on pages 1, 7, 18).

[146] J. Ziv and N. Merhav. A measure of relative entropy between individual sequences with application to universal classification. *IEEE Transactions on Information Theory*, 39(4):1270–1279, 1993. DOI: `10.1109/18.243444` (cited on page 18).