Minimal Suffix and Rotation of a Substring in Optimal Time*

Tomasz Kociumaka

Institute of Informatics, University of Warsaw, ul. Stefana Banacha 2, 02-097 Warsaw, Poland
kociumaka@mimuw.edu.pl

Abstract

For a text of length n given in advance, the substring minimal suffix queries ask to determine the lexicographically minimal non-empty suffix of a substring specified by the location of its occurrence in the text. We develop a data structure answering such queries optimally: in constant time after linear-time preprocessing. This improves upon the results of Babenko et al. (CPM 2014), whose trade-off solution is characterized by $\Theta(n \log n)$ product of these time complexities. Next, we extend our queries to support concatenations of $\mathcal{O}(1)$ substrings, for which the construction and query time is preserved. We apply these generalized queries to compute lexicographically minimal and maximal rotations of a given substring in constant time after linear-time preprocessing.

Our data structures mainly rely on properties of Lyndon words and Lyndon factorizations. We combine them with further algorithmic and combinatorial tools, such as fusion trees and the notion of order isomorphism of strings.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases minimal suffix, minimal rotation, Lyndon factorization, substring canonization, substring queries

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.28

1 Introduction

Lyndon words, as well as the inherently linked concepts of the lexicographically minimal suffix and the lexicographically minimal rotation of a string, are one of the most successful concepts of combinatorics of words. Introduced by Lyndon [26] 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 [13], maximal repetitions [6], and the shortest common superstring problem [28].

The central combinatorial property of Lyndon words, proved by Chen et al. [8], states that every string can be uniquely decomposed into a non-increasing sequence of Lyndon words. Duval [14] devised a simple algorithm computing the Lyndon factorization in linear time and constant space. He also observed that the same algorithm can be used to determine the lexicographically minimal and maximal suffix, as well as the lexicographically minimal and maximal rotation of a given string.

The first two algorithms are actually on-line procedures: in linear time they allow computing the minimal and maximal suffix of every prefix of a given string. For rotations

* This work is supported by Polish budget funds for science in 2013-2017 as a research project under the 'Diamond Grant' program.

such a procedure was later introduced by Apostolico and Crochemore [3]. Both these solutions lead to the optimal, quadratic-time algorithms computing the minimal and maximal suffixes and rotations for all substring of a given string. Our main results are the data-structure versions of these problems: we preprocess a given text T to answer the following queries:

▶ **Problem** (Minimal Suffix Queries). *Given a substring $v = T[\ell..r]$ of T, report the lexicographically smallest non-empty suffix of v (represented by its length).*

▶ **Problem** (Minimal Rotation Queries). *Given a substring $v = T[\ell..r]$ of T, report the lexicographically smallest rotation of v (represented by the number of positions to shift).*

For both problems we obtain optimal solutions with linear construction time and constant query time. For Minimal Suffix Queries this improves upon the results of Babenko et al. [4], who developed a trade-off solution, which for a text of length n has $\Theta(n \log n)$ product of preprocessing and query time. We are not aware of any results for Minimal Rotation Queries except for a data structure only testing cyclic equivalence of two subwords [24]. It allows constant-time queries after randomized preprocessing running in expected linear time.

An optimal solution for the Maximal Suffix Queries was already obtained in [4], while the Maximal Rotation Queries are equivalent to Minimal Rotation Queries subject to alphabet reversal. Hence, we do not focus on the maximization variants of our problems.

Using an auxiliary result devised to handle Minimal Rotation Queries, we also develop a data structure answering in $\mathcal{O}(k^2)$ time the following generalized queries:

▶ **Problem** (Generalized Minimal Suffix Queries). *Given a sequence of substrings $v_1, \ldots, v_k$ ($v_i = T[\ell_i..r_i]$), report the lexicographically smallest non-empty suffix of their concatenation $v_1 v_2 \ldots v_k$ (represented by its length).*

All our algorithms are deterministic procedures for the standard word RAM model with machine words of size $W = \Omega(\log n)$ [17]. The alphabet is assumed to be $\Sigma = \{0, \ldots, \sigma - 1\}$ where $\sigma = n^{\mathcal{O}(1)}$, so that all letters of the input text T can be sorted in linear time.

Applications The last factor of the Lyndon factorization of a string is its minimal suffix. As noted in [4], this can be used to reduce computing the factorization $v = v_1^{p_1} \cdots v_m^{p_m}$ of a substring $v = T[\ell..r]$ to $\mathcal{O}(m)$ Minimal Suffix Queries in T. Hence, our data structure determines the factorization in the optimal $\mathcal{O}(m)$ time. If v is a concatenation of k substrings, this increases to $\mathcal{O}(k^2 m)$ time (which we did not attempt to optimize in this paper).

The primary use of Minimal Rotation Queries is *canonization* of substrings, i.e., classifying them according to cyclic equivalence (conjugacy); see [3]. As a proof-of-concept application of this natural tool, we propose counting distinct substring with a given exponent.

Related work Our work falls in a class of substring queries: data structure problems solving basic stringology problems for substrings of a preprocessed text. This line of research, implicitly initiated by substring equality and longest common prefix queries (using suffix trees and suffix arrays; see [10]), now includes several problems related to compression [9, 22, 24, 5], pattern matching [24], and the range longest common prefix problem [1, 2]. Closest to ours is a result by Babenko et al. [5], which after $\mathcal{O}(n\sqrt{\log n})$-expected-time preprocessing allows determining the k-th smallest suffix of a given substring, as well as finding the lexicographic rank of one substring among suffixes of another substring, both in logarithmic time.

Outline of the paper In Section 2 we recall standard definitions and two well-known data structures. Next, in Section 3, we study combinatorics of minimal suffixes, using in particular a notion of *significant suffixes*, introduced by I et al. [19, 20] to compute Lyndon factorizations of grammar-compressed strings. Section 4 is devoted to answering Minimal Suffix Queries. We use *fusion trees* by Pătraşcu and Thorup [29] 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* [25, 23] to reduce the number of precomputed values. In Section 5 we repeat the same steps for Generalized Minimal Suffix Queries. We conclude with Section 6, where we briefly discuss the applications.

2 Preliminaries

We consider strings over an alphabet $\Sigma = \{0, \ldots, \sigma - 1\}$ with the natural order $\prec$. The empty string is denoted as ε. By Σ^* (Σ^+) we denote the set of all (resp. non-empty) finite strings over Σ. We also define Σ^∞ as the set of infinite strings over Σ. We extend the order $\prec$ on Σ in the standard way to the *lexicographic* order on $\Sigma^* \cup \Sigma^\infty$.

Let $w = w[1] \ldots w[n]$ be a string in Σ^*. We call n the *length* of w and denote it by $|w|$. For $1 \le i \le j \le n$, a string $u = w[i] \ldots w[j]$ is called a *substring* of w. By $w[i..j]$ we denote the occurrence of u at position i, called a *fragment* of w. A fragment of w other than the whole 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 n is called a *suffix* of w. We use abbreviated notation $w[..j]$ and $w[i..]$ for a prefix $w[1..j]$ and a suffix $w[i..n]$ of w, respectively. A *border* of w is a substring of w which occurs both as a prefix and as a suffix of w. An integer p, $1 \le p \le |w|$, is a *period* of w if $w[i] = w[i+p]$ for $1 \le i \le n - p$. If w has period p, we also say that is has *exponent* $\frac{|w|}{p}$. Note that p is a period of w if and only if w has a border of length $|w| - p$.

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.

Augmented suffix array The *suffix array* [27] of a text T of length n is a permutation SA of $\{1, \ldots, n\}$ defining the lexicographic order on suffixes $T[i..n]$: $T[SA[i]..n] \prec T[SA[j]..n]$ if and only if $i < j$. For a string T, both SA and its inverse permutation ISA take $\mathcal{O}(n)$ space and can be computed in $\mathcal{O}(n)$ time; see e.g. [10]. Typically, one also builds the LCP table and extends it with a data structure for range minimum queries [18, 7], so that the longest common prefix of any two suffixes of T can be determined efficiently.

Similarly to [4], we also construct these components for the reversed text T^R. Additionally, we preprocess the ISA table to answer range minimum and maximum queries. The resulting data structure, which we call the *augmented suffix array* of T, lets us perform many queries.

▶ **Theorem 1** (Augmented suffix array; see Fact 3 and Lemma 4 in [4]). *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 allows answering the following queries in $\mathcal{O}(1)$ time given fragments x, y of T:*

1. *determine if $x \prec y$, $x = y$, or $x \succ y$,*
2. *compute the longest common prefix $\mathrm{lcp}(x, y)$ and the longest common suffix $\mathrm{lcs}(x, y)$,*
3. *compute $\mathrm{lcp}(x^\infty, y)$ and determine if $x^\infty \prec y$, $x^\infty = y$, or $x^\infty \succ y$.*

Moreover, given indices i, j, it can compute in $\mathcal{O}(1)$ time the minimal *and the* maximal *suffix among $\{T[k..n] : i \le k \le j\}$.*

Fusion trees Consider a set $\mathcal{A}$ of W-bit integers (recall that W is the machine word size). *Rank* queries given a W-bit integer x return $\mathrm{rank}_{\mathcal{A}}(x)$ defined as $|\{y \in \mathcal{A} : y < x\}|$. Similarly, *select* queries given an integer r, $0 \le r < |\mathcal{A}|$, return $\mathrm{select}_{\mathcal{A}}(r)$, the r-th smallest element in $\mathcal{A}$, i.e., $x \in \mathcal{A}$ such that $\mathrm{rank}_{\mathcal{A}}(x) = r$. These queries can be used to determine the *predecessor* and the *successor* of a W-bit integer x, i.e., $\mathrm{pred}_{\mathcal{A}}(x) = \max\{y \in \mathcal{A} : y < x\}$ and $\mathrm{succ}_{\mathcal{A}}(x) = \min\{y \in \mathcal{A} : y \ge x\}$. We answer these queries with dynamic fusion trees by Pătraşcu and Thorup [29]. We only use these trees in a static setting, but the original static fusion trees by Fredman and Willard [15] do not have an efficient construction procedure.

▶ **Theorem 2** (Fusion trees [29, 15]). *There exists a data structure of size $\mathcal{O}(|\mathcal{A}|)$ which answers* $\mathrm{rank}_{\mathcal{A}}$, $\mathrm{select}_{\mathcal{A}}$, $\mathrm{pred}_{\mathcal{A}}$, *and* $\mathrm{succ}_{\mathcal{A}}$ *queries in $\mathcal{O}(1 + \log_W |\mathcal{A}|)$ time. Moreover, it can be constructed in $\mathcal{O}(|\mathcal{A}| + |\mathcal{A}| \log_W |\mathcal{A}|)$ time.*

3 Combinatorics of minimal suffixes and Lyndon words

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 Σ and extend it to the *reverse lexicographic* order, and an auxiliary symbol $\$ \notin \Sigma$. We extend the order $\prec$ on Σ so that $\mathtt{c} \prec \$$ (and thus $\$ \prec^R \mathtt{c}$) for every $\mathtt{c} \in \Sigma$. We define $\bar{\Sigma} = \Sigma \cup \{\$\}$, but unless otherwise stated, we still assume that the strings considered belong to Σ^*.

▶ **Observation 3.** *If $u, v \in \Sigma^*$, then $u\$ \prec v$ if and only if $v \prec^R u$.*

We use MinSuf^R and MaxSuf^R to denote the minimal (resp. maximal) suffix with respect to $\prec^R$. The following observation relates the notions we introduced:

▶ **Observation 4.**
1. $\mathrm{MaxSuf}(v, \varepsilon) = \mathrm{MaxSuf}(v)$ *for every* $v \in \bar{\Sigma}^*$,
2. $\mathrm{MinSuf}(vw) = \min(\mathrm{MinSuf}(v, w), \mathrm{MinSuf}(w))$ *for every* $v \in \bar{\Sigma}^*$ *and* $w \in \bar{\Sigma}^+$,
3. $\mathrm{MinSuf}(vc) = \mathrm{MinSuf}(v, c)$ *for every* $v \in \bar{\Sigma}^*$ *and* $c \in \bar{\Sigma}$,
4. $\mathrm{MinSuf}(v, w\$) = \mathrm{MaxSuf}^R(v, w)\$$ *for every* $v, w \in \Sigma^*$,
5. $\mathrm{MinSuf}(v\$) = \mathrm{MaxSuf}^R(v)\$$ *for every* $v \in \Sigma^*$.

A property seemingly similar to 5. is false for every $v \in \Sigma^+$: $\$ = \mathrm{MinSuf}^R(v\$) \neq \mathrm{MaxSuf}(v)\$$.

A notion deeply related to minimal and maximal suffixes is that of a Lyndon word [26, 8]. A string $w \in \Sigma^+$ is called a *Lyndon word* if $\mathrm{MinSuf}(w) = w$. Note that such w does not have proper borders, since a border would be a non-empty suffix smaller than w. A *Lyndon factorization* of a string $u \in \bar{\Sigma}^*$ is a representation $u = u_1^{p_1} \dots u_m^{p_m}$, where u_i are Lyndon words such that $u_1 \succ \dots \succ u_m$. Every non-empty word has a unique Lyndon factorization [8], which can be computed in linear time and constant space [14].

3.1 Significant suffixes

Below we recall a notion of *significant suffixes*, introduced by I et al. [19, 20] 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 [20].

▶ **Definition 5** (see [19, 20]). A suffix s of a string $v \in \Sigma^*$ is a *significant suffix* of v if $sw = \mathrm{MinSuf}(v, w)$ for some $w \in \bar{\Sigma}^*$.

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 $\lambda \le i \le m$. Observe that $s_\lambda \succ \ldots \succ s_m \succ s_{m+1} = \varepsilon$, since v_i is a prefix of s_i. 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 $\Lambda(w) = \{s_\lambda, \dots, s_m, s_{m+1}\}$, $X(w) = \{x_\lambda^\infty, \dots, x_m^\infty\}$, and $X'(w) = \{x_\lambda^{p_\lambda}, \dots, x_m^{p_m}\}$. The observation below lists several immediate properties of the introduced strings:

▶ **Observation 6.** *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 $|s_i| - |s_{i+1}|$, and (c) $|s_i| > 2|s_{i+1}|$. In particular, $|\Lambda(v)| = \mathcal{O}(\log |v|)$.*

The following lemma shows that $\Lambda(v)$ is equal to the set of significant suffixes of v. (Significant suffixes are actually defined in [20] as $\Lambda(v)$ and only later proved to satisfy our Definition 5.) 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 7** (I et al. [20], 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 $r = \mathrm{rank}_{X(v)}(w)$.

We conclude this section with a precise characterization of $\Lambda(uv)$ for $|u| \le |v|$ in terms of $\Lambda(v)$ and $\mathrm{MaxSuf}^R(u, v)$. This is another key ingredient of our data structure, in particular letting us efficiently compute significant suffixes of a given fragment of T. The proof is deferred to the full version due to space constraints.

▶ **Lemma 8.** *Let $u, v \in \Sigma^+$ be strings such that $|u| \le |v|$. Also, 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*

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

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

4 Answering 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 4.3. and the notion of significant suffixes. Efficient computation of these suffixes, also used in the construction of further versions of our data structure, is based on Lemma 8, which yields a recursive procedure. The only "new" suffix needed at each step is determined using the following result. It can be seen as a cleaner formulation of Lemma 14 in [4].

▶ **Lemma 9.** *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 $\mathrm{MaxSuf}^R(u, v)$ in $\mathcal{O}(1)$ time.*

▶ **Lemma 10.** *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 $|v'| = \lceil \frac{1}{2}|v| \rceil$. We recursively generate $\Lambda(v')$ and use Lemma 9 to compute $s = \mathrm{MaxSuf}^R(u, v')$. Then, we apply the characterization of Lemma 8 to determine $\Lambda(v) = \Lambda(uv')$, using the augmented suffix array (Theorem 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 calls) 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. ◀

▶ **Corollary 11.** Minimal Suffix Queries *can be answered in $\mathcal{O}(\log|v|)$ time using the augmented suffix array of T.*

Proof. Recall that Observation 4.3. yields $\mathrm{MinSuf}(v) = \mathrm{MinSuf}(v[1..m-1], v[m])$ where $m = |v|$. Consequently, $\mathrm{MinSuf}(v) = sv[m]$ for some $s \in \Lambda(v[1..m-1])$. We apply Lemma 10 to compute $\Lambda(v[1..m-1])$ and determine the answer among $\mathcal{O}(\log|v|)$ candidates using lexicographic comparison of fragments, provided by the augmented suffix array (Theorem 1). ◀

4.1 $\mathcal{O}(\log^*|v|)$-time Minimal Suffix Queries

An alternative $\mathcal{O}(\log|v|)$-time algorithm could be developed based just on the second part of Lemma 8: 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 Lemma 8 and Observation 4.3. Here, the first candidate $\mathrm{MaxSuf}^R(u, v')$ is determined via Lemma 9, while the second one using 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 *basic* fragments, i.e., 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 [4], along with a smooth trade-off between their preprocessing and query times.

Our $\mathcal{O}(\log^*|v|)$-time query algorithm combines recursion with preprocessing for certain *distinguished* fragments. More precisely, we say that $v = T[\ell..r]$ 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 12.** *Given a fragment v such that $|v| > f(|v|)$, we can in constant time decompose $v = uv'v''$ such 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 $r' \ge r - 2^{q'} \ge \ell$. 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 $|u| \le |v'|$. Moreover, $|v'| \le |v|$ assures that $f(|v'|) \mid f(|v|)$, so $f(|v'|) \mid r'$, and therefore v' is indeed distinguished. ◀

Observation 4.2. implies $\mathrm{MinSuf}(v) \in \{\mathrm{MinSuf}(uv', v''), \mathrm{MinSuf}(v'')\}$ and Lemma 8 further yields $\mathrm{MinSuf}(v) \in \{\mathrm{MaxSuf}^R(u, v')v'', \mathrm{MinSuf}(v', v''), \mathrm{MinSuf}(v'')\}$, i.e., leaves us with three candidates for $\mathrm{MinSuf}(v)$. Our query algorithm obtains $\mathrm{MaxSuf}^R(u, v')$ using Lemma 9,

computes $\mathrm{MinSuf}(v'')$ recursively, and determines $\mathrm{MinSuf}(v', v'')$ through the characterization of Lemma 7. The latter step is performed using the following component based on a fusion tree, which we build for all distinguished fragments.

▶ **Lemma 13.** *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 $\mathrm{MinSuf}(v, T[r+1..r'])$. Moreover, this data structure can be constructed in $\mathcal{O}(\log|v|)$ time using the augmented suffix array of T.*

Proof. By Lemma 7, we have $\mathrm{MinSuf}(v, w) = s_{m+1-\mathrm{rank}_{X(v)}(w)}w$, so in order to determine $\mathrm{MinSuf}(v, T[r+1..r'])$, it suffices to store $\Lambda(v)$ and efficiently compute $\mathrm{rank}_{X(v)}(w)$ given $w = T[r+1..r']$. We shall reduce these rank queries to rank queries in an integer set $R(v)$.

▶ **Claim.** *Denote $X(v) = \{x^\infty_\lambda, \ldots, x^\infty_m\}$ and let*

$$R(v) = \{r + \mathrm{lcp}(T[r+1..], x^\infty_j) : x^\infty_j \in X(w) \wedge x^\infty_j \prec T[r+1..]\}.$$

For every index r', $r < r' \le n$, we have $\mathrm{rank}_{X(v)}(T[r+1..r']) = \mathrm{rank}_{R(v)}(r')$.

Proof. We shall prove that for each j, $\lambda \le j \le m$, we have

$$x^\infty_j \prec T[r+1..r'] \iff \big(r + \mathrm{lcp}(T[r+1..], x^\infty_j) < r' \ \wedge\ x^\infty_j \prec T[r+1..]\big).$$

First, if $x^\infty_j \succ T[r+1..]$, then clearly $x^\infty_j \succ T[r+1..r']$ and both sides of the equivalence are false. Therefore, we may assume $x^\infty_j \prec T[r+1..]$. Observe that in this case $d := \mathrm{lcp}(T[r+1..], x^\infty_j)$ is strictly less than $n - r$, and $T[r+1..r+d] \prec x^\infty_j \prec T[r+1..r+d+1]$. Hence, $x^\infty_j \prec T[r+1..r']$ if and only if $r + d < r'$, as claimed. ◀

We apply Theorem 2 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 6.

The construction algorithm uses Lemma 10 to compute $\Lambda(v) = \{s_\lambda, \ldots, s_{m+1}\}$. Next, for each j, $\lambda \le j \le m$, we need to determine $\mathrm{lcp}(T[r+1..], x^\infty_j)$. This is the same as $\mathrm{lcp}(T[r+1..], (x^{p_j}_j)^\infty)$ and, by Observation 6, $x^{p_j}_j$ can be retrieved as 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.

▶ **Theorem 14.** *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 (Theorem 1) and the components of Lemma 13 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 $\mathcal{O}(\sum_q \frac{nq}{q^{\omega(1)}}) = \mathcal{O}(\sum_q \frac{n}{q^{\omega(1)}}) = \mathcal{O}(n)$.

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 11. If $|v| > 2^q$, we apply Fact 12 to determine a decomposition $v = uv'v''$, which gives us three candidates for $\mathrm{MinSuf}(v)$. As already described, $\mathrm{MinSuf}(v'')$ is computed recursively, $\mathrm{MinSuf}(v', v'')$ using Lemma 13, and $\mathrm{MaxSuf}^R(u, v')v''$ using Lemma 9. The latter two 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, $f(f(x)) \leq \log x$ 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 $\mathcal{O}(\log^* |v|)$. ◀

4.2 $\mathcal{O}(1)$-time Minimal Suffix Queries

The $\mathcal{O}(\log^* |v|)$ time complexity of the query algorithm of Theorem 14 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 $f(f(n)) = o(\frac{\log n}{\log \log n})$. In this section we show that the minimal suffixes of such short fragments can precomputed in a certain sense, and thus after reaching $\tau = f(f(n))$ we do not need to perform further recursive calls.

For constant alphabets, we could actually store all the answers for all $\mathcal{O}(\sigma^\tau) = n^{o(1)}$ strings of length up to τ. Nevertheless, in general all letters of T, and consequently all fragments of T, could even be distinct. 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* [25, 23], denoted as $x \approx y$, if $|x| = |y|$ and for every two positions i, j $(1 \leq i, j \leq |x|)$ 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., $x[i..j] \prec x[i'..j'] \iff y[i..j] \prec y[i'..j']$. Consequently, order-isomorphic strings cannot be distinguished using Minimal Suffix Queries or Generalized Minimal Suffix Queries.

Moreover, observe that every string of length m is order-isomorphic to a string over an alphabet $\{1, \ldots, m\}$. 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 15.** *For every fixed integer $m = W^{\mathcal{O}(1)}$, 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 can be evaluated in $\mathcal{O}(m)$ time.*

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 among these letters. We allocate $\lceil \log m \rceil$ bits per character and prepend such a representation with $\lceil \log m \rceil$ bits encoding $|w|$. This way $\mathrm{oid}(w)$ is a sequence of $(|w| + 1) \lceil \log m \rceil = \mathcal{O}(m \log m)$ bits. Using Theorem 2 to build the fusion tree, we obtain an $\mathcal{O}(m)$-time evaluation algorithm. ◀

To answer queries for short fragments of T, we define overlapping *blocks* of length $m = 2\tau$: for $0 \leq i \leq \frac{n}{\tau}$ we create a block $T_i = T[1 + i\tau .. \min(n, (i+2)\tau)]$. For each block we apply Fact 15 to compute the identifier $\mathrm{oid}(T_i)$. 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, $|v| \leq \tau$, to a 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 16.** *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 can answer 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 17.** *For every text T of length n there exists a data structure of size $\mathcal{O}(n)$ which can be constructed in $\mathcal{O}(n)$ time and answers* Minimal Suffix Queries *in $\mathcal{O}(1)$ time.*

5 Answering Generalized Minimal Suffix Queries: Overview

In this section we sketch our solution for Generalized Minimal Suffix Queries, focusing on the differences compared to the data structure developed in Section 4. As in Section 4, we proceed in three steps gradually improving the query time; we start, however, with some terminology.

We define a *k-fragment* of a text T as a concatenation $T[\ell_1..r_1]\cdots T[\ell_k..r_k]$ of k fragments of the text 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, 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 can be implemented in $\mathcal{O}(k)$ time and referring to $w[\ell..r]$ in our algorithms, we assume that such an operation is performed.

Basic queries regarding k-fragments easily reduce to their counterparts for 1-fragments:

▶ **Observation 18.** *The augmented suffix array can answer queries 1., 2., and 3. in $\mathcal{O}(k)$ time if x and y are k-fragments of T.*

Generalized Minimal Suffix Queries can be reduced to the following auxiliary queries:

▶ **Problem** (Auxiliary Minimal Suffix Queries)**.** *Given a fragment v of T and a k-fragment w of T, compute* $\mathrm{MinSuf}(v, w)$ *(represented as a $(k+1)$-fragment of T).*

▶ **Lemma 19.** *For every text T, the minimal suffix of a k-fragment v can be determined by k* Auxiliary Minimal Suffix Queries *(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 4.2., $\mathrm{MinSuf}(v) = \mathrm{MinSuf}(v_k)$ or for some i, $1 \le i < k$, we have $\mathrm{MinSuf}(v) = \mathrm{MinSuf}(v_i, v_{i+1} \cdots v_k)$. 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 4.3. lets 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 smallest among them using $k-1$ comparisons, each performed in $\mathcal{O}(k)$ time; see Theorem 1 and Observation 18. ◀

Below we focus on the auxiliary queries only. Answering them in $\mathcal{O}(k \log |v|)$ time is easy: We apply Lemma 10 to determine $\Lambda(v)$, and then we compute the smallest string among $\{sw : s \in \Lambda(v)\}$. These strings are $(k+1)$-fragments of T and thus a single comparison takes $\mathcal{O}(k)$ time using the augmented suffix array.

5.1 $\mathcal{O}(k \log^* |v|)$-time Auxiliary Minimal Suffix Queries

Our solution is based on that in Section 4.1. The only big challenge is to generalize Lemma 13: preprocess v to compute $\mathrm{MinSuf}(v, w)$ for an arbitrary k-fragment w in $\mathcal{O}(k)$ time. We still apply Lemma 7, but this time we actually determine $\mathrm{rank}_{X'(v)}(w)$, which differs from $\mathrm{rank}_{X(v)}(w)$ by at most one (and therefore leaves us with two candidates for $\mathrm{rank}_{X(v)}(w)$).

This is because in general we are able to preprocess a family A of fragments of T to determine $\mathrm{rank}_A(w)$ given a k-fragment w of T. Our solution is based on the compressed trie

of fragments in A, accompanied with several fusion trees to allow efficient navigation. For $|A| \leq W^{\mathcal{O}(1)}$ it takes $\mathcal{O}(|A|^2)$ time to construct and determines ranks in the optimal time $\mathcal{O}(k)$. By our choice of distinguished fragments v of length 2^q, building this component for all sets $X'(v)$ takes $\mathcal{O}(\frac{nq^2}{q^{\omega(1)}}) = \mathcal{O}(\frac{n}{q^{\omega(1)}})$ time, which is $\mathcal{O}(n)$ in total (over all values of q).

5.2 $\mathcal{O}(k)$-time Auxiliary Minimal Suffix Queries

To achieve the optimal query time, we again focus on $|v| \leq \tau$ with $\tau = o(\frac{\log n}{\log \log n})$. Computing $\mathrm{MinSuf}(v, w)$, we need to handle k-fragments w of arbitrary length, which might be scattered around the text T (not just in a block T_i containing v), so the task is much more difficult than in Section 4.2. Our approach is to replace w with a similar k'-fragment w' of $T_i\$$, such that $k' \leq k+1$ and $\mathrm{rank}_{X'(v)}(w) = \mathrm{rank}_{X'(v)}(w')$. This is achieved again using fusion trees.

As already noted, a fixed value of $\mathrm{rank}_{X'(v)}(w)$ gives two candidates for $\mathrm{rank}_{X(v)}(w)$, i.e., for $\mathrm{MinSuf}(v, w)$. Simultaneously $\mathrm{rank}_{X'(v)}(w')$ depends only on the relative order of letters of $T_i\$$. Hence, for each distinct $\mathrm{oid}(T_i)$ and for each fragment v of $T_i\$$, we construct $\Lambda(v)$ and a data structure able to efficiently rank k-fragments of $T_i\$$ in $X'(v)$. This component is built using the general tool for ranking k-fragments in a collection of fragments, which we mentioned in Section 5.1. This ultimately leads to the strongest result of this paper:

▶ **Theorem 20.** *For every text T of length n there exists a data structure of size $\mathcal{O}(n)$ which can be constructed in $\mathcal{O}(n)$ time and answers* Generalized Minimal Suffix Queries *in $\mathcal{O}(k^2)$ time.*

6 Applications

As already noted in [4], Minimal Suffix Queries can be used to compute Lyndon factorization. For fragments of T, and in general $k = \mathcal{O}(1)$, we obtain an optimal solution:

▶ **Corollary 21.** *For every text T of length n there exists a data structure of size $\mathcal{O}(n)$ which given a k-fragment v of T determines the Lyndon factorization $v = v_1^{q_1} \dots v_m^{q_m}$ in $\mathcal{O}(k^2 m)$ time. The data structure takes $\mathcal{O}(n)$ time to construct.*

Our main motivation of 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; see [10]:

▶ **Observation 22.** *The minimal cyclic rotation of v is the prefix of $\mathrm{MinSuf}(v, v)$ of length $|v|$.*

▶ **Theorem 23.** *For every text T of length n there exists a data structure of size $\mathcal{O}(n)$ which given a k-fragment v of T determines the lexicographically smallest cyclic rotation of v in $\mathcal{O}(k^2)$ time. The data structure takes $\mathcal{O}(n)$ time to construct.*

Using Minimal Rotation Queries, we can compute the Karp-Rabin fingerprint [21] of the minimal rotations of a given fragment v of T (or in general, of a k-fragment). This can be interpreted as computing fingerprints up to cyclic equivalence, i.e., evaluating a function h such that $h(\ell, r) = h(\ell', r')$ if and only if $T[\ell..r]$ and $T[\ell'..r']$ are cyclically equivalent.

Consequently, we are able, for example, to count distinct substrings of T with a given exponent $1 + 1/\alpha$. They occur within runs or α-gapped repeats, which can be generated in time $\mathcal{O}(n\alpha)$ [6, 12, 16] and classified using Minimal Rotation Queries according to the cyclic equivalence class of their period. For a fixed equivalence class the set of substrings generated by a single repeat can be represented as a cyclic interval, and the cardinality of a union of intervals is simple to determine; see also [11], where this approach was used to count and list squares and, in general, substrings with a given exponent 2 or more.

Acknowledgements I would like to thank the remaining co-authors of [4], collaboration with whom on earlier results about minimal and maximal suffixes sparked some of my ideas used in this paper. Special acknowledgments to Paweł Gawrychowski and Tatiana Starikovskaya for numerous discussions on this subject.

References

1 Amihood Amir, Alberto Apostolico, Gad M. Landau, Avivit Levy, Moshe Lewenstein, and Ely Porat. Range LCP. *J. Comput. Syst. Sci.*, 80(7):1245–1253, 2014. doi:10.1016/j.jcss.2014.02.010.

2 Amihood Amir, Moshe Lewenstein, and Sharma V. Thankachan. Range LCP queries revisited. In Costas S. Iliopoulos, Simon J. Puglisi, and Emine 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.

3 Alberto Apostolico and Maxime Crochemore. Optimal canonization of all substrings of a string. *Inf. Comput.*, 95(1):76–95, 1991. doi:10.1016/0890-5401(91)90016-U.

4 Maxim Babenko, Paweł Gawrychowski, Tomasz Kociumaka, Ignat Kolesnichenko, and Tatiana Starikovskaya. Computing minimal and maximal suffixes of a substring. *Theor. Comput. Sci.*, 2015. In press. doi:10.1016/j.tcs.2015.08.023.

5 Maxim Babenko, Paweł Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In Piotr Indyk, editor, *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 572–591. SIAM, 2015. doi:10.1137/1.9781611973730.39.

6 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem, 2015. arXiv:1406.0263v7.

7 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *Latin American Symposium on Theoretical Informatics, LATIN 2000*, volume 1776 of *LNCS*, pages 88–94. Springer Berlin Heidelberg, 2000. doi:10.1007/10719839_9.

8 Kuo Tsai Chen, Ralph Hartzler Fox, and Roger Conant Lyndon. Free differential calculus, IV. The quotient groups of the lower central series. *Ann. Math.*, 68(1):81–95, 1958. doi:10.2307/1970044.

9 Graham 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.

10 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, New York, NY, USA, 2007.

11 Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Extracting powers and periods in a word from its runs structure. *Theor. Comput. Sci.*, 521:29–41, 2014. doi:10.1016/j.tcs.2013.11.018.

12 Maxime Crochemore, Roman Kolpakov, and Gregory Kucherov. Optimal bounds for computing α-gapped repeats. In Adrian-Horia Dediu, Jan Janousek, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications, LATA 2016*, volume 9618 of *LNCS*, pages 245–255. Springer, 2016. doi:10.1007/978-3-319-30000-9_19.

13 Maxime Crochemore and Dominique Perrin. Two-way string-matching. *J. ACM*, 38(3):650–674, July 1991. doi:10.1145/116825.116845.

14 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983. doi:10.1016/0196-6774(83)90017-2.

15 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

16 Paweł Gawrychowski, Tomohiro I, Shunsuke Inenaga, Dominik Köppl, and Florin Manea. Efficiently finding all maximal α-gapped repeats. In Nicolas Ollinger and Heribert Vollmer, editors, *Symposium on Theoretical Aspects of Computer Science, STACS 2016*, volume 47 of *LIPIcs*, pages 39:1–39:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.STACS.2016.39.

17 Torben Hagerup. Sorting and searching on the word RAM. In Michel Morvan, Christoph Meinel, and Daniel Krob, editors, *Symposium on Theoretical Aspects of Computer Science, STACS 1998*, volume 1373 of *LNCS*, pages 366–398. Springer, Berlin Heidelberg, 1998. doi:10.1007/BFb0028575.

18 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.

19 Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Efficient Lyndon factorization of grammar compressed text. In Johannes Fischer and Peter Sanders, editors, *Combinatorial Pattern Matching, CPM 2013*, volume 7922 of *LNCS*, pages 153–164. Springer, 2013. doi:10.1007/978-3-642-38905-4.

20 Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. In Oren Kurland, Moshe Lewenstein, and Ely Porat, editors, *String Processing and Information Retrieval, SPIRE 2013*, volume 8214 of *LNCS*, pages 174–185. Springer International Publishing Switzerland, 2013. doi:10.1007/978-3-319-02432-5_21.

21 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.

22 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theor. Comput. Sci.*, 525:45–54, 2014. doi:10.1016/j.tcs.2013.10.010.

23 Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theor. Comput. Sci.*, 525:68–79, 2014. doi:10.1016/j.tcs.2013.10.006.

24 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In Piotr Indyk, editor, *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.

25 Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear time algorithm for consecutive permutation pattern matching. *Inf. Process. Lett.*, 113(12):430–433, 2013. doi:10.1016/j.ipl.2013.03.015.

26 Roger Conant Lyndon. On Burnside's problem. *T. Am. Math. Soc.*, 77(2):202–215, 1954. doi:10.1090/S0002-9947-1954-0064049-X.

27 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.

28 Marcin Mucha. Lyndon words and short superstrings. In Sanjeev Khanna, editor, *24th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013*, pages 958–972. SIAM, 2013. doi:10.1137/1.9781611973105.

29 Mihai Pătraşcu and Mikkel 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.