Constructing and Indexing the Bijective and Extended Burrows–Wheeler Transform *

Hideo Bannai Juha Kärkkäinen Dominik Köppl Marcin Piątkowski

Abstract

The Burrows–Wheeler transform (BWT) is a permutation whose applications are prevalent in data compression and text indexing. The *bijective BWT* is a bijective variant of it that has not yet been studied for text indexing applications. We fill this gap by proposing a self-index built on the bijective BWT. The self-index applies the backward search technique of the FM-index to find a pattern P with $\mathcal{O}(|P| \lg |P|)$ backward search steps. Additionally, we propose the first linear-time construction algorithm that is based on SAIS, improving the best known result of $\mathcal{O}(n \lg n / \lg \lg n)$ time to linear.

Keywords: Burrows–Wheeler transform, construction algorithm, FM index, Lyndon factorization

1 Introduction

The Burrows–Wheeler transform (BWT) [20] is a transformation permuting the characters of a given string T, defined as the last characters of all the cyclic rotations of T sorted in lexicographic order. The BWT is the main ingredient of compressed text indexes such as the FM-index [28, 29], the run-length FM-index [57], or the r-index [32]. It is used in mainstream bioinformatics tools such as SOAP2 [53], BWA [52], Bowtie 2 [51], MONI [19, 71], or SPUMONI 2 [2]. It also has many applications in data compression [1].

Despite that the BWT is used in the aforementioned self-indexes capable of restoring the original text, the BWT is not a bijection on the set of strings since it is neither injective nor surjective, e.g., $\mathsf{BWT}(\mathtt{ab}) = \mathsf{BWT}(\mathtt{ba}) = \mathtt{ba}$, and there is no string T such that the BWT of T is $\mathtt{ab}$. However, the BWT can be interpreted as a bijection between multisets of primitive, cyclic strings and strings, while preserving the frequencies of the symbols [59, Theorem 20]: the BWT of a multiset of primitive, cyclic strings is defined as the last characters of all the rotations of all the cyclic strings in the multiset sorted in a lexicographic-like order called the ω-order. This BWT variant is also called the *extended BWT* [59] or eBWT. For example, the extended BWT of the multiset $\{\mathtt{ab}, \mathtt{ab}, \mathtt{aba}\}$ is $\mathtt{babbaaa}$. The inverse of the eBWT, known as the *Gessel–Reutenauer transform* [33], has already been conceived more than ten years prior to the discovery of the eBWT. Given a string T, the Gessel–Reutenauer transform maps T to the multiset of strings whose eBWT is T. The eBWT can be turned into a bijection on the set of strings by introducing another bijection between multisets of primitive, cyclic strings and strings. Such a variant has been proposed as the bijective BWT [38, 50], or BBWT. The said bijection for BBWT is based on the Lyndon factorization [23] of T, which is a unique factorization of a string into a lexicographically non-increasing sequence of Lyndon words. Since Lyndon words are primitive, and any primitive string has a unique rotation that is Lyndon, the Lyndon factorization induces a bijection between strings of length n and multisets of primitive cyclic strings of total length n, which preserves the frequencies of the symbols. Gil and Scott [38] experimentally evaluated on the Calgary corpus that the output of the BBWT was for some datasets a little more compressible than the BWT. Unfortunately, not much is known about the compressibility of the BBWT, which has remained a puzzling open problem when writing this article. Nevertheless, the possibility of obtaining a new self-index based on the BBWT that might outperform BWT-based solutions with respect to space has sparked new questions such as whether we can compute the BBWT efficiently, and whether we can build a self-index upon it. We will address both questions in this article affirmatively.

*Parts of this work have already been presented at the 30th Annual Symposium on Combinatorial Pattern Matching [6] and at the 32nd Annual Symposium on Combinatorial Pattern Matching [7].

To make the connection between the BWT, the BBWT, and the eBWT clearer, we elaborate on the aforementioned example. There, ab is Lyndon, while aba has the rotation aab that is Lyndon. We can therefore identify the multiset $\{\mathtt{ab}, \mathtt{ab}, \mathtt{aba}\}$ with the string $\mathtt{ab} \cdot \mathtt{ab} \cdot \mathtt{aab}$ via the Lyndon factorization. Since the extended BWT of a set containing just one string S is the same as the BWT of S, we can regard the extended BWT as an extension of the BWT to a multiset of words, which is the BBWT of the Lyndon conjugates of the input strings concatenated in lexicographically decreasing order. While other generalizations to multisets of strings such as the BCR BWT [8] exist, only the eBWT does not require to introduce artificial delimiters to map a set of input strings to a single string.

For instance, given the set of strings $\mathcal{S} = \{\mathtt{a}, \mathtt{c}, \mathtt{bac}, \mathtt{adacb}, \mathtt{acbbcad}, \mathtt{bbc}\}$, the eBWT of $\mathcal{S}$ is given by abddbcccccbbbaaabcaa. We can obtain the same output, if we rotate each input string to its lexicographically smallest rotation a, c, acbad, adacb, acbbcad, and bbc, and sort these strings lexicographically to a, acbad, adacb, acbbcad, bbc, and c, concatenate them to a single string T and apply the BBWT on T.

In the following, we call the BWT *traditional* to ease the distinguishability of both transformations. The crucial operation for text indexes built upon the BWT is the backward search [62]: given a pattern P and the traditional BWT of T, the occurrences of P in a text T can be computed with $\mathcal{O}(|P|)$ backward search steps. In this light, one may ask whether it is possible to build similar index data structures by exchanging the traditional BWT with the bijective BWT.

1.1 Our Contributions

In this article, we answer affirmatively the above question: We show that searching a pattern P on the bijective BWT can be conducted with $\mathcal{O}(|P|\hat{p})$ backward search steps, where $\hat{p}$ is the number of distinct factors in the Lyndon factorization of the longest pre-Lyndon suffix of P. Since $\hat{p}$ is known to be in $\mathcal{O}(\lg |P|)$ [44], we can reduce the number of backward search steps to $\mathcal{O}(|P| \lg |P|)$.

Our results are based on combinatoric properties of Lyndon words and the bijective BWT. They may have applications in distributed implementations of the BWT index [45] or in practical database systems storing dynamic yet compressed data [14, 15].

Our second contribution is the first linear-time algorithm computing the BBWT in the word RAM model. The main idea is to adapt SAIS to compute the circular suffix array of the Lyndon factors. We obtain linear running time by exploiting some facts based on the nature of the Lyndon factorization.

Compared to the conference versions [6, 7], we could improve the space bounds of the indexing data structure in the light of novel advances for indexing the extended BWT. Further, we added a comparison on the number of character runs of the standard BWT and the BBWT on various datasets. The C++ implementation of the BBWT construction algorithm described in this paper is available at `https://github.com/mmpiatkowski/bbwt`.

1.2 Related Work

We separate work related to the research results of this article into the two categories construction and indexing.

1.2.1 Construction

In what follows, we review the traditional BWT construction via suffix arrays, and some algorithms computing the BBWT or the extended BWT. For the complexity analysis, we take a text T of length n whose characters are drawn from a polynomial bounded integer alphabet $[1..n^{\mathcal{O}(1)}]$. Let us start with the traditional BWT, which we can construct in linear time thanks to linear-time suffix array construction algorithms [47, 64]. That is because the traditional BWT, denoted by $\mathsf{BWT}[1..n]$, is determined by $\mathsf{BWT}[i] = T[\mathsf{SA}[i] - 1]$ for $\mathsf{SA}[i] > 1$ and $\mathsf{BWT}[i] = T[n]$ for $\mathsf{SA}[i] = 1$. This definition by the suffix array only holds if the text is terminated by an artificial character smaller than all other characters appearing in T (the famous dollar sign terminal), or T has been rotated such that it is strictly smaller than all its rotations (for that, T must be primitive, i.e., T cannot be the iterated concatenation of a same string). Considering the bijective BWT, Gil and Scott [38] postulated that it can be built in linear time, but did not give a construction algorithm. It is clear that the time is upper bounded by the total length of all conjugates [59, after Example 9], which is $\mathcal{O}(n^2)$. In the same paper, Mantaci et al. [59] also introduced the *extended* BWT. Later, Hon et al. [42] provided an algorithm building the extended

BWT in $\mathcal{O}(n \lg n)$ time. Later, in a larger number of authors [43], they refined the working space to $\mathcal{O}(n \lg \sigma)$, and gave the eBWT the alternative name *circular BWT*. Their idea is to construct the *circular suffix array* $\mathsf{SA}_\circ$ such that the i-th position of the extended BWT is given by $T[\mathsf{SA}_\circ[i] - 1]$, where T is the concatenation of all strings in $\mathcal{S}$. Bonomo et al. [13] presented an online algorithm building the extended BWT in $\mathcal{O}(n \lg n / \lg \lg n)$ time; here, online means that a construction algorithm updates the eBWT on reading a new input string; the input strings (of the multiset of strings to index) are expected to be lexicographically sorted and given by their Lyndon conjugates. An interesting aspect regarding the online construction is that the up so far only known online technique [65, 68] for computing the traditional BWT needs the text to be given in reversed order (starting with the last character). In [13, Sect. 6], they also gave a linear-time reduction from computing the extended BWT to computing the BBWT. Knowing that an irreducible word has exactly one conjugate being a Lyndon word, the reduction is done by exchanging each element of the set of irreducible strings $\mathcal{S}$ by the conjugate being a Lyndon word, and concatenating these Lyndon words after sorting them in descending order. Consequently, a linear-time BBWT construction algorithm can be used to compute the extended BWT in linear time. If only constant space is allowed, Köppl et al. [49] gave an algorithm running in time quadratic to the input string length.

The relationship between suffix array construction and Lyndon words is not new. In fact, there are approaches [5, 10, 55, 60, 66] that use the Lyndon factorization, Lyndon words, or the Lyndon array to compute the suffix array. Implicit in the suffix array construction algorithm of Olbrich et al. [66] is another construction algorithm for the BBWT, which exploits the Lyndon factorization boundaries for the suffix array computation, and seems to be able to compute the BBWT in linear time.

On the practical side, we are aware of the work of Branden Brown[1], Yuta Mori in his OpenBWT library[2], and of Neal Burns[3]. While the first is a naive but easily understandable implementation calling a general sorting algorithm on all conjugates to directly compute the BBWT, the second seems to be an adaptation of the suffix array – induced sorting (SAIS) algorithm [64] to induce the BBWT. The last one takes an already computed suffix array SA as input, and modifies SA such that reading the characters $T[\mathsf{SA}[i] - 1]$[4] gives the BBWT. For that, this algorithm shifts entries in SA to the right until they fit. Hence, the running time is based on the lengths of these shifts, which can be $\mathcal{O}(n^2)$, but seem to be negligible in practice for common texts.

With respect to the most recent progress on the computation of variations of the BWT, we have noted the following results. Giancarlo et al. [35] can compute the alternating Burrows–Wheeler transform [34] via a modification of the difference cover suffix sorting algorithm [47] with a linear time algorithm for finding the minimal cyclic rotation of a word with respect to the alternating lexicographical order. The alternating BWT can be briefly explained by the construction via the sorted rotations of an input string T. While the standard BWT is determined by reading the last character of each string in the list of all lexicographically sorted rotations of T, from start to end, we obtain the alternating BWT if this list is sorted lexicographically by the first character, then inverse-lexicographically by the second, again lexicographically for the third, and so on. Boucher et al. [17] proposed an adaptation of our algorithm for computing the eBWT, and showed that it is possible to perform the computation without rotating each input string to its Lyndon conjugate. Although their algorithm runs in the same asymptotic time complexity, they can omit this extra step that we would need to perform if we would use our BBWT-construction algorithm directly for the eBWT computation — for that it would be sufficient to concatenate all Lyndon conjugates in lexicographically descending order of the input strings. However, without the Lyndon rotations, they need to keep track for each text position they process, from which input string that position came. To that end, they define the *generalized conjugate array*, mimicking the cyclic suffix array, as an array of tuples of an input string index and a position inside that string. The authors additionally propose a practical variant that uses the prefix-free parsing technique [16].

Besides the eBWT, there are other BWT variants that support multiple input strings, notably the BCR BWT [8] and the BWT on the concatenated input separated by delimiters such as dollar signs. Both variants require, unlike the eBWT, that every string ends with a terminal symbol. More precisely, given t texts $T_1, \ldots, T_t$ over an alphabet Σ, it is required to append to each input text T_j a distinct

[1] `https://github.com/zephyrtronium/bwst`

[2] `https://web.archive.org/web/20170306035431/https://encode.ru/attachment.php?attachmentid=959&d=1249146089`

[3] `https://github.com/NealB/Bijective-BWT`

[4] Special care has to be taken when $\mathsf{SA}[i]$ is the start of a Lyndon factor.

terminal symbol $\$_j$ being smaller than any character appearing in Σ. If we order the texts by their lexicographic order and assign the symbols $\$_j$ the order $\$_1 < \ldots < \$_t$, then the BCR BWT and the eBWT of the set $\{T_1\$_1, \ldots, T_t\$_t\}$ are the same, for instance $\texttt{abddcc}\$_1\$_2\$_3\$_4\texttt{bccc}\$_5\texttt{cbbb}\$_6\texttt{baaaaa}$ for our running example $\{\texttt{a}\$_1, \texttt{acb}\$_2, \texttt{acbad}\$_3\texttt{acbbcad}\$_4, \texttt{bbc}\$_5, \texttt{c}\$_6\}$. Here, the ω-order coincides with the lexicographic ordering of the suffixes by the use of the distinct delimiters $\$_1, \ldots, \$_t$. With respect to algorithmic aspects, the construction of the BCR BWT differs to applying the BWT over the concatenation of the strings $T_1\$_1, \ldots, T_t\$_\texttt{t}$, meaning that we first linearize the input before application of the BWT. A similar approach to the concatenation of the strings was introduced by Boucher et al. [16], who made all dollar signs equal but appended another delimiter # being smaller than dollar. By doing so, the dollar signs separate the input strings, while # marks the end of the entire concatenated string. If we prepend # to the concatenated string instead of appending #, then the resulting string is Lyndon.

With respect to research on the BCR BWT, Díaz-Domínguez and Navarro [25] showed how to construct the BCR BWT from a grammar [24], which is based on the LMS-substrings of the SAIS algorithm. Finally, Cenzato and Lipták [21] conducted an empirical survey of the number of character runs in different BWT variations that are capable for presenting multiple strings.

1.2.2 Indexing

Besides the introduction of the circular suffix array, Hon et al. [42] proposed, in the same paper, an algorithm for circular pattern matching on the extended BWT. Circular pattern matching is the task to count all occurrences of a pattern P inside every string of $\mathcal{S}$, where the substrings $P[1..\ell]$ and $P[\ell+1..|P|]$ for a split position $\ell \in [1..|P|-1]$ occurring respectively as a suffix and a prefix of one string of $\mathcal{S}$ is also considered as an occurrence. In that sense, circular pattern matching does not only report substrings $T[i..i+|P|-1]$ with $T \in \mathcal{S}$ equal to P, but also occurrences of TT that contain the middle position.

Like our approach, this technique is based on the backward search. Applied on the bijective BWT, it allows us to perform circular pattern matching on the Lyndon factors ($\mathcal{S}$ becomes the set of Lyndon factors in this context). Boucher et al. [18] revisited this problem, and gave an adaptation of the r-index for the eBWT, using space linear in the number of runs of the eBWT, and locating queries in the same time bounds as the r-index. While their index works only for circular pattern matching, our index on the bijective BWT supports both types of queries,

- the circular matching when interpreting the Lyndon factors as independent input strings and
- the classic matching as performed by the standard BWT.

The former is an implicit consequence of [18], while the latter involves several technical proofs based on properties the Lyndon factorization.

With respect to different BWT variants, Giancarlo et al. [35] translated the backward search technique of the FM-index to the alternating BWT. They generalized their methods in a follow-up [36] for Burrows–Wheeler transforms whose orders are so-called *local orderings-based transformations*. This class contains the alternating BWT as well as the classic BWT. The authors proved that these BWT variations can support pattern matching taking time quadratic in the pattern length, and improved the time bounds to linear for a special subset of these orderings. For that special subset, including the alternating BWT, they additionally could prove that the pattern locating queries can be answered in words of space linear to the number of runs of the underlying BWT variant, which makes these BWT variants indexable in space linear to the number of character runs like the r-index for the standard BWT. The connection to the BBWT is that the BBWT as well as the eBWT apply the ω-order instead of the lexicographic order on all conjugates of the input. Their approach can be considered as orthogonal as we are unsure whether alternative orderings for the ω-order have already been investigated.

2 Preliminaries

Our computational model is the word RAM model with word size $\Omega(\lg n)$. Accessing a word costs $\mathcal{O}(1)$ time. In this article, we study strings on an *integer* alphabet $\Sigma = [1..\sigma]$ with size $\sigma = n^{\mathcal{O}(1)}$.

Table 1: Used symbols and variables in this article

symbol	meaning
T	input text
$F_1 \cdots F_f$	Lyndon factorization of T
$T_1^{\tau_1} \cdots T_t^{\tau_t}$	composed Lyndon factorization of T
R	reduced text, $R = T_1 \cdots T_t$

Strings. We call an element $T \in \Sigma^*$ a *string*. Its length is denoted by $|T|$. Given an integer $j \in [1..|T|]$, we access the j-th character of T with $T[j]$. Given a string $T \in \Sigma^*$, we denote with T^k that we concatenate k times the string T. The i-th *conjugate* $\mathsf{conj}_i(T)$ of a string $T[1..n]$ is defined as $T[i+1..n]T[1..i]$ for an integer $i \in [1..n]$.

A *bit vector* B is a string on the binary alphabet $\{0,1\}$. A *rank-support* data structure provides support for a *rank query* on B, i.e., retrieving the number of ones up to a queried position in B. There exist rank-support data structures that use $o(|B|)$ space on top of B, can be built in time linear to the length of B, and answer rank queries in constant time [46].

When T is represented by the concatenation of $X, Y, Z \in \Sigma^*$, i.e., $T = XYZ$, then X, Y, and Z are called a *prefix*, *substring*, and *suffix* of T, respectively. A prefix X, substring Y, or suffix Z is called *proper* if $X \neq T$, $Y \neq T$, or $Z \neq T$, respectively. A proper prefix X of T is called a *border* of T if it is also a suffix of T. T is called *border-free* if it has no border. For two integers i and j with $1 \leq i \leq j \leq |T|$, let $T[i..j]$ denote the substring of T that begins at position i and ends at position j in T. If $i > j$, then $T[i..j]$ is the empty string. In particular, the suffix starting at position j of T is denoted with $T[j..n]$. A *cyclic occurrence* of a string S in T is that S is a prefix of $\mathsf{conj}_i(T)$ for some $i \in [1..n]$. A string T is called *primitive* if it cannot be written as $T = S^e$ for a string $S \in \Sigma^+$ and an integer $e \geq 2$. The *root* of a string T denotes the primitive string S such that $T = S^e$ for an integer $e \geq 1$; in particular, T is primitive if $e = 1$.

Text Data Structures. Suppose we have given a text $T[1..n]$. If $[\mathsf{conj}_{\pi(1)}(T), \mathsf{conj}_{\pi(2)}(T), \ldots, \mathsf{conj}_{\pi(n)}(T)]$ is the lexicographically sorted list of all conjugates of T, then the *Burrows–Wheeler transform (BWT)* [20] of T is $\mathsf{BWT} = \mathsf{conj}_{\pi(1)}(T)[n] \cdot \mathsf{conj}_{\pi(2)}(T)[n] \cdots \mathsf{conj}_{\pi(n)}(T)[n]$.

If we assume that T is terminated with a delimiter smaller than all other characters appearing in T, the following data structures are well-defined. Let SA denote the *suffix array* [58] of T, which is a permutation of the integers in $[1..n]$. The entry $\mathsf{SA}[i]$ is the starting position of the i-th lexicographically smallest suffix such that $T[\mathsf{SA}[i]..] \prec T[\mathsf{SA}[i+1]..]$ for all integers i with $1 \leq i \leq n-1$. With SA, we can give an alternative definition of the BWT of T, which is given by $\mathsf{BWT}[i] = T[n]$ if $\mathsf{SA}[i] = 1$ and $\mathsf{BWT}[i] = T[\mathsf{SA}[i]-1]$ otherwise, for every i with $1 \leq i \leq n$.

Orders on Strings. We denote the *lexicographic order* with $\prec_{\text{lex}}$. Given two strings S and T, then $S \prec_{\text{lex}} T$ if S is a proper prefix of T or there exists an integer ℓ with $1 \leq \ell \leq \min(|S|, |T|)$ such that $S[1..\ell-1] = T[1..\ell-1]$ and $S[\ell] < T[\ell]$. We write $S \prec_\omega T$ if the infinite concatenation $S^\omega := SSS\cdots$ is lexicographically smaller than $T^\omega := TTT\cdots$. For instance, $\mathtt{ab} \prec_{\text{lex}} \mathtt{aba}$ but $\mathtt{aba} \prec_\omega \mathtt{ab}$. The relation $\prec_\omega$ induces an order on the set of *primitive* strings. Although not needed here, $\prec_\omega$ can be generalized to general strings by comparing the exponent if two strings have the same root [59, Definition 4], meaning $\mathtt{ab} \prec_\omega \mathtt{abab}$ since both have the root $\mathtt{ab}$, but the left has exponent one, while the right has exponent two. The relation $\prec_\omega$ is computable in time linear in the lengths of both strings by leveraging Fine and Wilf's Theorem [31].

Lemma 2.1 (Lemma 5 by Hon et al. [43])**.** For two primitive strings S and T with $\ell := \max(|S|, |T|)$, $S \prec_\omega T$ if and only if $S^\omega[1..2\ell] \prec_{\text{lex}} T^\omega[1..2\ell]$.

For a set of primitive strings $\mathcal{S} = \{T_1, \ldots, T_x\}$ of total length n, the *circular suffix array* [43] $\mathsf{SA}_\circ[1..n]$ of $\mathcal{S}$ is an integer array determined by the list L of the conjugates of all the strings in $\mathcal{S}$ sorted by the ω-order. If $L[r] = \mathsf{conj}_i(T_y)$, then $\mathsf{SA}_\circ[r] = \sum_{x=1}^{y-1} |T_x| + (i \mod |T_y|)$. To put this definition into words, we conceptionally consider the concatenation $T := T_1 \cdots T_x$, for which $\mathsf{SA}_\circ[r]$ maps to a position T, where

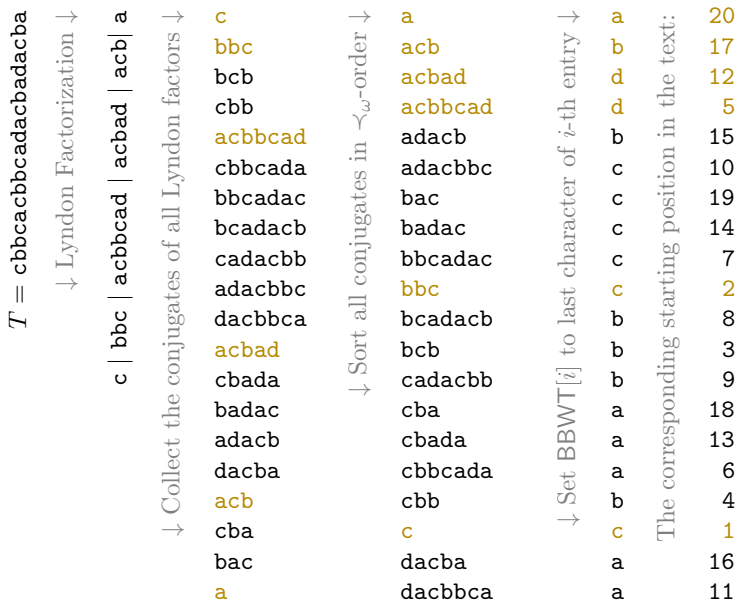

Figure 1: Constructing BBWT of $T =$ `cbbcacbbcadacbadacba`. The Lyndon factors are highlighted (■). Reading the characters of the penultimate column top-down yields BBWT. The last column shows in its i-th row the starting position of the i-th smallest conjugate of a Lyndon factor in the text. It is the circular suffix array studied later in Sect. 3.2.1. Note that `cbb` $\prec_{\text{lex}}$ `cbbcada`, but `cbbcada` $\prec_\omega$ `cbb`.

we can read the r-th smallest conjugate (among all conjugates of $\mathcal{S}$) if we stop reading at the end of an input string T_x and continue reading at its beginning.

Lyndon Words. Let us suppose we have given a primitive text $T[1..n]$. Then all its conjugates are distinct. We say that every conjugate of T (including itself) belongs to the *conjugate class* $\mathsf{conj}(T) := \{\mathsf{conj}_1(T), \ldots, \mathsf{conj}_n(T)\}$. Since T is primitive, its conjugate class contains *exactly* one conjugate that is lexicographically smaller than all other conjugates; this conjugate is called a *Lyndon word* [56]. Equivalently, a string T is said to be a Lyndon word if and only if $T \prec S$ for every proper suffix S of T. A consequence is that a Lyndon word is *border-free*. A *pre-Lyndon* word is a string that is a prefix of a Lyndon word.

The *Lyndon factorization* [23] of $T \in \Sigma^+$ is the unique factorization of T into a sequence of Lyndon words $F_1 \cdots F_f$, where (a) each $F_x \in \Sigma^+$ is a Lyndon word, and (b) $F_x \succeq_{\text{lex}} F_{x+1}$ for each $x \in [1..f)$.

Lemma 2.2 ([27, Algo. 2.1])**.** The Lyndon factorization of a string can be computed in linear time.

Each Lyndon word F_x for $x \in [1..f]$ is called a *Lyndon factor*. We denote the multiset of T's Lyndon factors by $\mathsf{LynF}(T) := \{F_1, \ldots, F_f\}$. There is a bijection between $\mathsf{LynF}(T)$ and T in such a sense that $\mathsf{LynF}(T)$ uniquely defines T. That is because we can restore T by

1. sorting the Lyndon factors of $\mathsf{LynF}(T)$ in lexicographically descending order, and
2. subsequently concatenating them.

The last factor F_f is special, as it has the following property:

Lemma 2.3 ([27, Prop. 1.9])**.** The last Lyndon factor of a string T is the smallest suffix of T.

We borrow from [44, Sect. 2.2] the notation $\mathsf{lfs}_T(j) := F_j \cdots F_f$ for the suffix of T starting with the j-th Lyndon factor. For what follows, we fix a string $T[1..n]$ over an alphabet Σ of size σ. We use the string $T :=$ `cbbcacbbcadacbadacba` as our running example. Its Lyndon factorization is `c`, `bbc`, `acbbcad`, `acbad`, `acb`, `a`.

We further define the *composed Lyndon factorization* of T, writing T as $F_1 \cdots F_f = T_1^{\tau_1} \cdots T_t^{\tau_t}$, where $T_1, \ldots, T_t$ is a sequence of *strictly* lexicographically decreasing Lyndon words and $\tau_x \geq 1$ denotes the

Table 2: Suffix array and BWT variants of the example string $T =$ `cbbcacbbcadacbadacba`. The Lyndon factorization $T = F_1 \cdots F_6$ of the text is symbolized by vertical bars. Note that the produced BWT does not coincide with the usual one defined by the lexicographically sorted rotations; it would coincide if we had appended an artificial delimiter smaller than all characters appearing in T. Nevertheless, we keep that *erroneous* BWT since the construction would work algorithmically in the same way as with the delimiter, and because we want to emphasize on the difference of the construction between BWT and BBWT. See Table 3 for an example with a correctly computed BWT.

	F_1	F_2			F_3							F_4					F_5			F_6
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	c	b	b	c	a	c	b	b	c	a	d	a	c	b	a	d	a	c	b	a
SA	20	17	12	5	15	10	19	14	2	7	3	8	4	9	18	13	1	6	16	11
BWT	b	d	d	c	b	c	c	c	c	c	b	b	b	b	a	a	a	a	a	a
$\mathsf{SA}_\circ$	20	17	12	5	15	10	19	14	7	2	8	3	9	18	13	6	4	1	16	11
BBWT	a	b	d	d	b	c	c	c	c	c	b	b	b	a	a	a	b	c	a	a

Table 3: Suffix array and BWT variants of the example string $T\$ =$ `cbbcacbbcadacbadacba`$ with a delimiter \$ smaller than all characters appearing in T. By definition of \$, it always creates a new Lyndon factor, which gets sorted on top of all cyclic rotations of all Lyndon factors such that $\mathsf{BBWT}[1] =$ \$. The rest of the BBWT entries are the same as in Table 2.

	F_1	F_2			F_3							F_4					F_5			F_6	F_7
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$T\$$	c	b	b	c	a	c	b	b	c	a	d	a	c	b	a	d	a	c	b	a	\$
SA	21	20	17	12	5	15	10	19	14	2	7	3	8	4	9	18	13	1	6	16	11
BWT	a	b	d	d	c	b	c	c	c	c	c	b	b	b	b	a	a	\$	a	a	a
BBWT	\$	a	b	d	d	b	c	c	c	c	c	b	b	b	a	a	a	b	c	a	a
$\mathsf{SA}_\circ$	21	20	17	12	5	15	10	19	14	7	2	8	3	9	18	13	6	4	1	16	11

multiplicity of the appearance of the factor T_x, for every $x \in [1..t]$. For each $x \in [1..t]$ there exists a $y \in [x..f]$ such that $T_x = F_y$. Let $R := T_1 \cdots T_t$ denote the text, in which all duplicate Lyndon factors are removed. Obviously, the Lyndon factorization of R is $T_1, \ldots, T_t$. Let $\mathsf{b}_R(T_x)$ and $\mathsf{e}_R(T_x)$ denote the starting and ending position of the x-th Lyndon factor in R, i.e., $R[\mathsf{b}_R(T_x)..\mathsf{e}_R(T_x)]$ is the x-th Lyndon factor T_x of R.

Bijective Burrows–Wheeler Transform. We denote the bijective BWT of T by BBWT, where $\mathsf{BBWT}[i]$ is the last character of the i-th string in the list storing the conjugates of all Lyndon factors $F_1, \ldots, F_f$ of T sorted with respect to $\prec_\omega$. Figure 1 shows the BBWT of our running example, and Table 2 presents the suffix array variants involved in the computation of the BBWT and the traditional BWT. The careful reader can observe that the BWT shown there does not coincide with BWT defined on the rotations-matrix, i.e., taking the last character of all cyclic rotations of T, after sorting them lexicographically. Doing so gives us the string `ddbcbcccccbbbbaaaaaa` as BWT, which differs from the BWT defined by $\mathsf{BWT}[i] = T[\mathsf{SA}[i] - 1]$. Both BWT definitions coincide, however, if we append a unique delimiter `$` to the text with `$` $< c$ for every $c \in \Sigma$. We omit `$` here since we emphasize on the BBWT, which does not need this delimiter character. Like the SA-based definition of the BWT, we can similarly define the BBWT in terms of the $\mathsf{SA}_\circ$ built on the set of Lyndon factors as input strings, where $\mathsf{BBWT}[i] = T[\mathsf{SA}_\circ[i] - 1]$ if $T[\mathsf{SA}_\circ[i] - 1]$ and $T[\mathsf{SA}_\circ[i]]$ belong to the same Lyndon factor, or $\mathsf{BBWT}[i] = c$, where c is the last character of the Lyndon factor that contains $\mathsf{SA}_\circ[i]$.

3 Constructing

We start with a brief review of the SAIS algorithm that constructs SA in linear time.

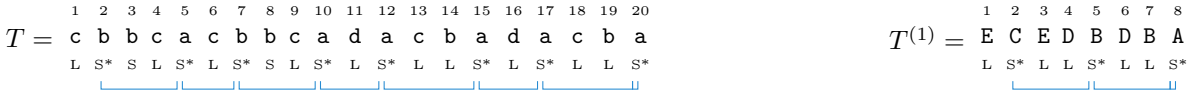

Figure 2: Splitting T and $T^{(1)}$ into LMS substrings. The rectangular brackets below the types represent the LMS substrings. $T^{(1)}$ is T after the replacement of its LMS substrings with their corresponding ranks defined in Sect. 3.2.3 and on the left of Fig. 3.

LMS Substring	Contents	Non-Terminal
$T[2..5]$	`bbca`	`E`
$T[5..7]$	`acb`	`C`
$T[7..10]$	`bbca`	`E`
$T[10..12]$	`ada`	`D`
$T[12..15]$	`acba`	`B`
$T[15..17]$	`ada`	`D`
$T[17..20]$	`acba`	`B`
$T[20..20]$	`a`	`A`

$\mathtt{S}^*$ Suffix	Contents
$T[20]$	`a`
$T[17..20]$	`acba`
$T[12..20]$	`acbadacba`
$T[5..20]$	`acbbcadacbadacba`
$T[15..20]$	`adacba`
$T[10..20]$	`adacbadacba`
$T[2..20]$	`bbcacbbcadacbadacba`
$T[7..20]$	`bbcadacbadacba`

Figure 3: Ranking of the LMS substrings and the $\mathtt{S}^*$ suffixes of our running example given in Sect. 3.2.3 and Fig. 2. *Left*: LMS substrings assigned with non-terminals reflecting their corresponding rank in $\prec_{\text{LMS}}$-order. *Right*: $\mathtt{S}^*$ suffixes of T sorted in $\prec_{\text{lex}}$-order. Note that $T[5..7] = \mathtt{acb} \prec_{\text{lex}} \mathtt{acba} = T[12..15] = T[17..20]$, but $\mathtt{acba} \prec_{\text{LMS}} \mathtt{acb}$.

3.1 Reviewing SAIS

Our idea is to adapt SAIS to compute $\mathsf{SA}_\circ$ instead of the suffix array. To explain this adaptation, we briefly review SAIS. First, SAIS assigns each suffix a type, which is either `L` or `S`:

- $T[i..|T|]$ is an `L` suffix if $T[i..|T|] \succ_{\text{lex}} T[i+1..|T|]$, or
- $T[i..|T|]$ is an `S` suffix otherwise, i.e., $T[i..|T|] \prec_{\text{lex}} T[i+1..|T|]$,

where we stipulate that $T[|T|]$ is always type `S`. Since it is not possible that $T[i..|T|] = T[i+1..|T|]$, SAIS assigns each suffix a type. An `S` suffix $T[i..|T|]$ is additionally an $\mathtt{S}^*$ suffix (also called LMS suffix in [64]) if $T[i-1..|T|]$ is an `L` suffix. The substring between two succeeding $\mathtt{S}^*$ suffixes is called an *LMS substring*. In other words, a substring $T[i..j]$ with $i < j$ is an LMS substring if and only if $T[i..|T|]$ and $T[j..|T|]$ are $\mathtt{S}^*$ suffixes and there is no $k \in [i+1..j-1]$ such that $T[k..|T|]$ is an $\mathtt{S}^*$ suffix. A border case is $T[|T|..|T|]$, which has to be the smallest suffix of T (and can be achieved by appending the artificial character `$` to T lexicographically smaller than all other characters appearing it T) such that $T[|T|..|T|]$ in an $\mathtt{S}^*$ suffix. We additionally treat $T[|T|..|T|]$ as an LMS substring. The types for the suffixes of our running example are given in Fig. 2. Regarding the defined types, we make no distinction between suffixes and their starting positions (e.g., the statements that (a) $T[i]$ is type `L` and (b) $T[i..|T|]$ is an `L` suffix are equivalent).

Next, Nong et al. [64, Def. 3.3] define a relation $\prec_{\text{LMS}}$ on substrings of T based on the lexicographic order and the types: Given two substrings S and U. Let i be the smallest integer such that (1) $S[i] < U[i]$ or (2) $S[i]$ is type `L` and $U[i]$ is type `S` or $\mathtt{S}^*$. If such an i exists, then we write $S \prec_{\text{LMS}} U$. For two LMS substrings S and U with $S \neq U$, either $S \prec_{\text{LMS}} U$ or $U \prec_{\text{LMS}} S$, even if S is a prefix of U (cf. the discussion below of Def. 3.3 in [64]). So $\prec_{\text{LMS}}$ is an order on the LMS substrings. The $\prec_{\text{LMS}}$-order is shown on the left side of Fig. 3 for the LMS substrings listed of the left side of Fig. 2. The crucial observation is that the $\prec_{\text{LMS}}$-order of the distinct LMS substrings coincides with the lexicographic order of the suffixes starting with these LMS substrings [64, Lemma 3.8]. See Fig. 6 for a juxtaposition of the different orders defined in this article.

Nong et al. [64, A3.4] compute the $\prec_{\text{LMS}}$-order of all LMS substrings with the induced sorting (which we describe below for the step of computing the rank of all suffixes). Figure 4 visualizes this computation on our running example. Hence, we can assign each LMS substring a rank based on the $\prec_{\text{LMS}}$-order. Next, we build a string $T^{(1)}$ of LMS substring ranks with $T^{(1)}[i]$ being the rank of the i-th LMS substring

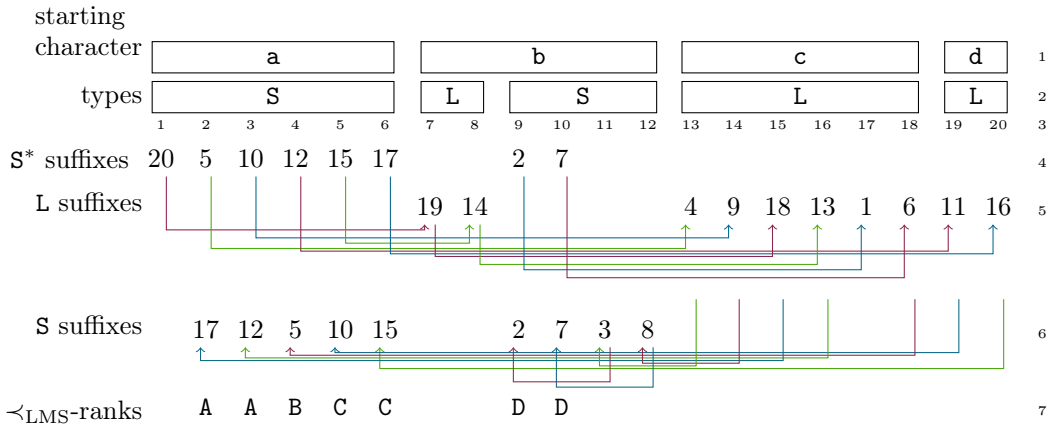

Figure 4: Inducing LMS substrings. Rows 1 and 2 show the partitioning of SA into buckets, first divided by the starting characters of the respective LMS substrings, and second by the types L and S. In Row 4, the S* suffixes are inserted into their respective S buckets. Here it is sufficient to only put the smallest S* suffix in the correct order among all other S* suffixes in the same bucket. This suffix is $T[20..20]$ in our example, stored at the suffix array entry 1. The S* (resp. L) suffixes induce the L (resp. S) suffixes in Row 5 (resp. Row 6). The last row assigns each S* suffix a meta-character representing its $\prec_{\text{LMS}}$-rank. We can compute two subsequent suffixes by character-wise comparison, spending $\mathcal{O}(|T|)$ time in total since the LMS substrings have a total length of $\mathcal{O}(|T|)$.

of T in text order.[5] See the right side of Fig. 2 for our running example. We recursively call SAIS on this text of ranks until the ranks of all LMS substrings are distinct. Given that we have computed $T^{(k)}$ and all characters of $T^{(k)}$ (i.e., the ranks of the respective LMS substrings) are distinct, then these ranks determine the order of the S* suffixes of $T^{(k-1)}$. That is because each S* suffix in $T^{(k-1)}$ starts with an LMS substring, and because all LMS substrings in $T^{(k-1)}$ are distinct, the lexicographical order of two S* suffixes are given by the order of the ranks of the LMS substrings they have as prefixes. We terminate the recursion for $T^{(k-1)}$ after having built the suffix array by inducing the ranks of the other suffixes from the S* suffixes, and move up the recursion level until returning to the original input string T while knowing the ranks of its S* suffixes. The order of the S* suffixes of our running example are given in Fig. 3 on the right side. Having the order of the S* suffixes, we allocate space for the suffix array, and divide the suffix array into buckets, grouping each suffix with the same starting character and same type (either L or S) into one bucket. Among all suffixes with the same starting character, the L suffixes precede the S suffixes [48, Corollary 3]. Putting S* suffixes in their respective buckets according to their order (smallest elements are the leftmost elements in the buckets), we can induce the L suffixes, as these precede either L or S* suffixes. For that, we scan SA from left to right, and take action only for suffix array entries that are not empty: When accessing the entry $\mathsf{SA}[k] = i$ with $i > 1$, write $i-1$ to the leftmost available slot of the L bucket with the character $T[i-1]$ if $T[i-1..|T|]$ is an L suffix. Finally, we can induce the $\prec_{\text{lex}}$-order of the S suffixes by scanning the suffix array from right to left: When accessing the entry $\mathsf{SA}[k] = i$, write $i-1$ to the rightmost available slot of the S type bucket with the character $T[i-1]$ if $T[i-1..|T|]$ is an S suffix. As an invariant, we always fill an L bucket and an S bucket from left to right and from right to left, respectively. So we can think of each L bucket and each S bucket as a list with an insertion operation at the end or at the beginning, respectively. We conduct these steps for our running example in Fig. 5.

In total, the induced sorting procedure takes $\mathcal{O}(|T|)$ time. The recursion step takes also $\mathcal{O}(|T|)$ time since there are at most $|T|/2$ LMS substrings (there are no two text positions $T[i]$ and $T[i+1]$ with type S* for $i \in [1..n-1]$). This gives $\mathcal{T}(n) = \mathcal{T}(n/2) + cn \in \mathcal{O}(n)$ total time, where $c > 0$ is a constant and $\mathcal{T}(n)$ denotes the time complexity for computing a suffix array of length n.

However, with SAIS we cannot obtain $\mathsf{SA}_\circ$ *ad hoc* since we need to (a) exchange $\prec_{\text{lex}}$ with $\prec_\omega$ and (b) make the algorithm run on a multiset of Lyndon factors instead of a single string. For the former (a)

[5] We can obtain $T^{(1)}$ by scanning T from left to right and replacing each LMS substring by its respective rank, but keep its last character in T if this character is the first character of the subsequent LMS substring. We further omit the first characters of T that are not part of an LMS substring (which must be of type L).

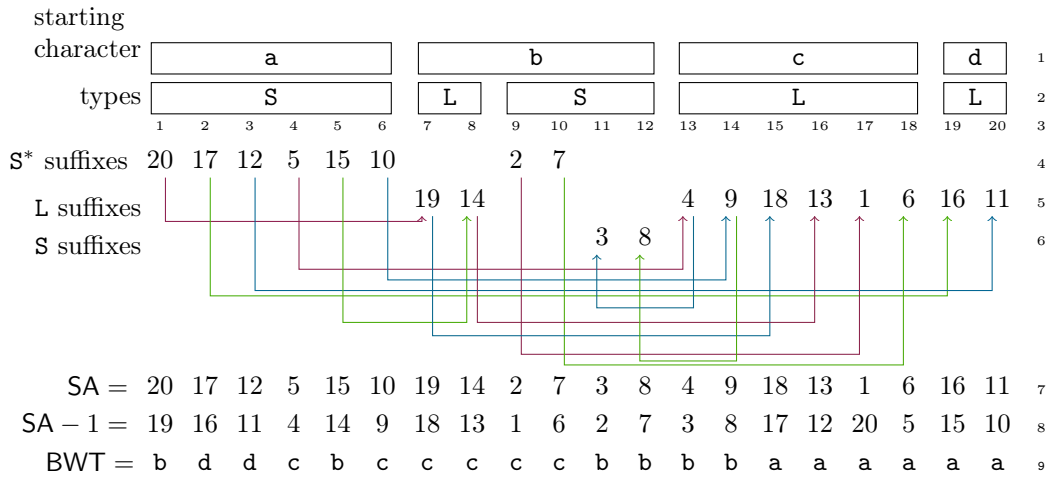

Figure 5: Inducing L and S suffixes from the $\prec_{\text{lex}}$-order of the S* suffixes given in Fig. 2. Rows 1 and 2 show the partitioning of SA into buckets, first divided by the starting characters of the respective suffixes, and second by the types L and S. Row 4 is SA after inserting the S* suffixes according to their $\prec_{\text{lex}}$-order rank obtained from the right of Fig. 3. The S* (resp. L) suffixes induce the L (resp. S) suffixes in Row 5 (resp. Row 6). Putting all together yields SA in Row 7. In the penultimate row $\mathsf{SA}-1$, each text position stored in SA is decremented by one, or set to n if this position was 1. The last row shows $T[(\mathsf{SA}-1)[i]] = \mathsf{BWT}[i]$ in its i-th column, which is the BWT of T. This BWT is not reversible since the input is not terminated with a unique character like $. To obtain the BWT of $T$$, we first write $T[\mathsf{SA}[1]] = T[20] =$ a to the output, and then BWT, but exchanging $\mathsf{BWT}[\mathsf{SA}^{-1}[1]] = \mathsf{BWT}[17] =$ a with $, i.e., abddcbccccbbbbaa$aaa.

U	V	$\prec_{\text{lex}}$	$\prec_{\omega}$	$\prec_{\text{LMS}}$
aba	aca	$<$	$<$	$<$
adc	adcb	$<$	$<$	$>$
acb	acba	$<$	$>$	$>$

Figure 6: Comparison of the three orders studied in this paper applied to LMS substrings. Assume that U and V are substrings of the text surrounded by a character d (i.e., $T = \ldots$ dUd$\ldots$dVd$\ldots$) such that the first and the last character of both U and V start with an S* suffix. We mark with the signs $<$ and $>$ whether U is smaller or respectively larger than V according to the corresponding order. The orders can differ only when one string is the prefix of another string, as this is the case in the last two rows. Finally, occurrences of U and V can be $\prec_{\text{LMS}}$-incomparable in different contexts such as $\ldots$dUa$\ldots$dVd$\ldots$, for instance.

we note that $\prec_{\text{lex}}$ and $\prec_\omega$ are interchangeable for Lyndon words [13, Thm. 8]. Since the SAIS algorithm sorts LMS substrings by $\prec_{\text{LMS}}$ that retains the lexicographical order of the LMS substrings, we can be sure that the Lyndon factors can be correctly sorted. To cope with (b), we remove duplicate Lyndon factors, and slightly modify the behavior of SAIS when it accesses positions prior to the first position of a Lyndon factor.

3.2 Our Adaptation

We want SAIS to sort Lyndon conjugates in $\prec_\omega$-order instead of suffixes in $\prec_{\text{lex}}$-order. For that, we first get rid of duplicate Lyndon factors to facilitate the analysis, and then subsequently introduce a slightly different notion to the types of suffixes and LMS substrings, which translates the suffix sorting problem into computing the BBWT.

3.2.1 Reduced String and Composed Lyndon Factorization

In a pre-computation step, we want to facilitate our analysis by removing all identical Lyndon factors from T yielding a reduced string R. We want to remove them to make conjugates unique; thus we can linearly order them. Consequently, the first step is to show that we can obtain the BBWT of T from the circular suffix array of R (which we will subsequently define).

Our aim is to compute the $\prec_\omega$-order of all conjugates of all Lyndon factors of R, which are given by the set $\mathcal{S} := \bigcup_{x=1}^{t} \mathsf{conj}(T_x)$. Like Hon et al. [43], we present this order in the *circular suffix array* $\mathsf{SA}_\circ$ of $\{T_1, \ldots, T_t\}$. For these input strings, $\mathsf{SA}_\circ$ is of length $|R|$ with $\mathsf{SA}_\circ[k] = i$ if $R[i..\mathsf{e}_R(T_x)]R[\mathsf{b}_R(T_x)..i-1]$ is the k-th smallest string in $\mathcal{S}$ with respect to $\prec_\omega$, where $i \in [\mathsf{b}_R(T_x)..\mathsf{e}_R(T_x)]$. The length of $\mathsf{SA}_\circ$ is $|R|$ since we can associate each text position $\mathsf{SA}_\circ[k]$ in R with a conjugate starting with $R[\mathsf{SA}_\circ[k]]$.

Having the circular suffix array $\mathsf{SA}_\circ$ of $\{T_1, \ldots, T_t\}$, we can compute the BBWT of T by reading $\mathsf{SA}_\circ[k]$ for $k \in [1..|R|]$ from left to right: Given $\mathsf{SA}_\circ[k] = i \in [\mathsf{b}_R(T_x)..\mathsf{e}_R(T_x)]$, we append $T[i^-]$ exactly τ_x times to BBWT, where i^- is $i-1$ or $\mathsf{e}_R(T_x)$ if $i = \mathsf{b}_R(T_x)$. (This is analogous to the definition of BWT where we set $\mathsf{BWT}[i] = T[n]$ for $\mathsf{SA}[i] = 1$, but here we wrap around each Lyndon factor.)

Algorithm 1: $\mathsf{BBWT}(T)$ – linear-time construction of BBWT. We have $\mathsf{SA}_\circ[i] \in [\mathsf{b}_R(T_k)..\mathsf{e}_R(T_k)]$ and $\mathsf{SA}_\circ[i]^- = \mathsf{SA}_\circ[i]-1$ for $\mathsf{b}_R(T_k) < \mathsf{SA}_\circ[i] \leq \mathsf{e}_R(T_k)$ and $\mathsf{SA}_\circ[i]^- = \mathsf{e}_R(T_k)$ for $\mathsf{SA}_\circ[i] = \mathsf{b}_R(T_k)$.

Input: string $T \in \Sigma^+$ over a finite alphabet Σ
Output: BBWT of T

```
1 T = T_1^{τ_1} ··· T_t^{τ_t} ;                          // Lyndon factorization of T
2 R = T_1 ··· T_t ;                                      // reduced version of T
3 SA_∘ ←Algorithm 2(R, T_1 ··· T_t) // compute circular suffix array of R with Algo. 2
4 B = ε ;                                  // build BBWT(T) in B based on SA_∘(R)
5 for i ← 1 to |R| do
6 ⌊ B = B + (R[SA_∘[i]^-])^{τ_k} ;                       // here + denotes concatenation
7 return B;
```

3.2.2 Translating Types to Inf-Suffixes

In what follows, we continue working with R defined in Sect. 3.2.1 instead of T. Let $R[i..]$ denote the infinite string $R[i..\mathsf{e}_R(T_x)]T_xT_x\cdots = \mathsf{conj}_k(T_x)\mathsf{conj}_k(T_x)\cdots$ with x such that $i \in [\mathsf{b}_R(T_x)..\mathsf{e}_R(T_x)]$ and $k = i - \mathsf{b}_R(T_x)$ or $k = n$ if $i = \mathsf{b}_R(T_x)$. We say that $R[i..]$ is an *inf-suffix*. As a shorthand, we also write $T_x[i..] = \mathsf{conj}_{i-1}(T_x)\mathsf{conj}_{i-1}(T_x)\cdots$ for the inf-suffix starting at $R[\mathsf{b}_R(T_x)+i-1]$. In particular, $T_x[|T_x|+1..] = T_x[1..] = T_xT_x\cdots$.

Like in SAIS, we distinguish between `L` and `S` inf-suffixes:

- $R[i..]$ is an `L` inf-suffix if $R[i..] \succ_{\text{lex}} R[i^+..]$, and
- $R[i..]$ is an `S` inf-suffix if $R[i..] \prec_{\text{lex}} R[i^+..]$,

Algorithm 2: Linear-time construction of circular suffix array $\mathsf{SA}_\circ$ of $R = T_1 \cdots T_t$

Input: string $R = T_1 \cdots T_t \in \Sigma^+$ together with its Lyndon factorization $T_1 \prec T_2 \prec \ldots \prec T_t$
Output: circular suffix array $\mathsf{SA}_\circ$ of R

assign each position in R type `S` or `L` ; // $\mathcal{O}(n)$ time
mark starting positions of LMS inf-suffixes of R ; // $\mathcal{O}(n)$ time
compute bucket sizes for R ;
// pre-sort all LMS inf-substrings
rank each LMS inf-substring of R preserving their lexicographical order ;
if *not all LMS inf-substrings are distinct* **then**
    create a string $R^{(1)}$ by exchanging each LMS inf-suffix in R by its rank ; // $|R^{(1)}| \leq |R|/2$
    // The recursive call for a string of at most half the length of the input
    $\mathsf{SA}_\circ^{(1)} \leftarrow$ output of Algorithm 2 for the input $R^{(1)}$;
    determine ranks of LMS inf-suffixes by $\mathsf{SA}_\circ^{(1)}$;
else
    rank of an LMS inf-suffix is the rank of its respective LMS inf-substring ;
allocate space for $\mathsf{SA}_\circ$ and create `S` and `L` type buckets for each character ;
insert each LMS inf-suffix of R at the beginning of its proper `S` bucket ;
// induce $\mathsf{SA}_\circ$ from R and the inf-suffixes
foreach *L-type inf-suffix v of R (scanning succeeding positions from left to right in $\mathsf{SA}_\circ$)* **do**
    insert v into the leftmost available slot of its proper `L` bucket ;
clear all S type inf-suffixes from $\mathsf{SA}_\circ$;
foreach *S-type inf-suffix v of R (scanning succeeding positions from right to left in $\mathsf{SA}_\circ$)* **do**
    insert v into the rightmost available slot of its proper `S` bucket ;
return $\mathsf{SA}_\circ$;

where i^+ is either $i+1$ or $\mathsf{b}_R(T_x)$ if $i = \mathsf{e}_R(T_x)$, and x is given such that $i \in [\mathsf{b}_R(T_x)..\mathsf{e}_R(T_x)]$. Finally, we introduce the $\mathtt{S}^*$ inf-suffixes as a counterpart to the $\mathtt{S}^*$ suffixes: If $R[i..]$ is an `S` inf-suffix, it is further an $\mathtt{S}^*$ inf-suffix if $R[i^-..]$ is an `L` inf-suffix with i^- being either $i-1$ or $\mathsf{e}_R(T_x)$ if $i = \mathsf{b}_R(T_x)$, and $x \in [1..t]$ chosen such that $i \in [\mathsf{b}_R(T_x)..\mathsf{e}_R(T_x)]$.

When speaking about types, we do not distinguish between an inf-suffix and its starting position in R. This definition assigns all positions of R a type except those belonging to a Lyndon factor of length one. We solve this by stipulating that all Lyndon factors of length one start with an $\mathtt{S}^*$ inf-suffix. However, in what follows, we temporarily omit all Lyndon factors of length one because we will later see that they can be placed at the beginning of their corresponding buckets in the circular suffix array. They nevertheless appear in the examples for completeness. To show that suffixes and inf-suffixes starting at the same position have the same type (except for some border-cases), the following lemma will be particularly useful:

Lemma 3.1 ([13, Lemma 7])**.** For $i, j \in [1..|T_x|]$ and $x \in [1..t]$, the following statements are equivalent:

1. $\mathrm{conj}_{i-1}(T_x) = T_x[i..|T_x|]T_x[1..i-1] \prec_{\mathrm{lex}} T_x[j..|T_x|]T_x[1..j-1] = \mathrm{conj}_{j-1}(T_x)$;
2. $\mathrm{conj}_{i-1}(T_x) \prec_\omega \mathrm{conj}_{j-1}(T_x)$, i.e., $T_x[i..] \prec_{\mathrm{lex}} T_x[j..]$;
3. $T_x[i..|T_x|] \prec_{\mathrm{lex}} T_x[j..|T_x|]$.

Proof. The statements follow directly from the properties of Lyndon words. □

Lemma 3.2. Omitting all Lyndon factors of length one from R, the types of all positions match the original SAIS types, except maybe $R[1]$ and $R[|R|..]$, where $R[1..]$ and $R[|R|..|R|]$ are always an $\mathtt{S}^*$ inf-suffix and an $\mathtt{S}^*$ suffix, respectively.

Proof. We show that inf-suffixes as well as suffixes starting with Lyndon factors have the same type $\mathtt{S}^*$:

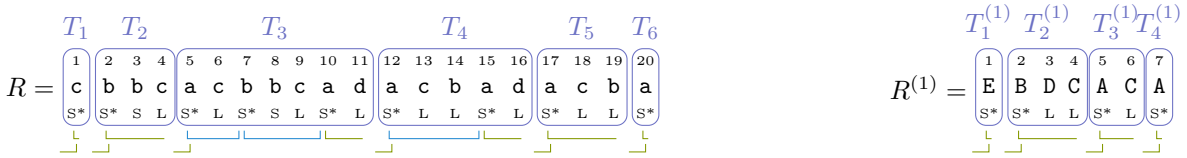

Figure 7: Splitting R and $R^{(1)}$ into LMS inf-substrings. The rectangular brackets below the types represent the LMS inf-substrings. Broken brackets denote that the corresponding LMS inf-substring ends with the first character of the Lyndon factor in which it is contained. They are colored in green (■); all other LMS inf-substrings are represented by brackets colored in blue (■). $R^{(1)}$ is R after the replacement of its LMS inf-substrings with their corresponding ranks defined in Sect. 3.2.3 and on the left of Fig. 8.

inf-suffixes. Assume that $R[\mathsf{b}_R(T_x)..]$ is an $\mathtt{L}$ inf-suffix for an $x \in [1..t]$. According to the definition of the Lyndon factorization, $R[\mathsf{b}_R(T_x)+1..] \prec_{\text{lex}} R[\mathsf{b}_R(T_x)..]$, i.e., $T_x[2..] \prec_{\text{lex}} T_x[1..]$, and with Lemma 3.1, $T_x[2..|T_x|] \prec_{\text{lex}} T_x$, contradicting that T_x is a Lyndon word. Finally, $R[\mathsf{b}_R(T_x)..]$ is an $\mathtt{S}^*$ inf-suffix because $T_x \prec_{\text{lex}} T_x[|T_x|]$ and hence $T_x[1..] \prec_{\text{lex}} T_x[|T_x|..]$, again with Lemma 3.1.

suffixes. Due to the Lyndon factorization, $R[\mathsf{b}_R(T_x)..|R|] \succ_{\text{lex}} R[\mathsf{b}_R(T_{x+1})..|R|]$ for $x \in [1..t-1]$. Hence, the suffix $R[\mathsf{e}_R(T_x)..|R|]$ starting at $R[\mathsf{e}_R(T_x)]$ has to be lexicographically larger than the suffix $R[\mathsf{e}_R(T_x)+1..|R|] = R[\mathsf{b}_R(T_{x+1})..|R|]$, otherwise we could extend the Lyndon factor T_x.

Consequently, $R[\mathsf{b}_R(T_x)..|R|]$ and $R[\mathsf{b}_R(T_x)..]$ are an $\mathtt{S}^*$ suffix and an $\mathtt{S}^*$ inf-suffix, respectively, and $R[\mathsf{e}_R(T_x)..|R|]$ and $R[\mathsf{e}_R(T_x)..]$ are an $\mathtt{L}$ suffix and an $\mathtt{L}$ inf-suffix.

The claim for all other positions ($\bigcup_{x=1}^{t-1}[\mathsf{b}_R(T_x)+1..\mathsf{e}_R(T_x)-1]$) follows by observing that $T_x[1..]$ is the $\prec_{\text{lex}}$-smallest inf-suffix among all inf-suffixes starting in T_x and $R[\mathsf{b}_R(T_{x+1})..|R|]$ is $\prec_{\text{lex}}$-smaller than all suffixes starting in $R[\mathsf{b}_R(T_x)..\mathsf{e}_R(T_x)]$ for $x \in [1..t-1]$. □

We excluded the suffix $R[\mathsf{b}_R(T_t)+1..|R|]$ in the claim of the lemma since we require for SAIS that the last character is always an $\mathtt{S}^*$ suffix, which is usually enforced by adding an artificial character at the end of the string that is lexicographically smaller than all other characters appearing in the text.

A corollary is that $R[i..|R|] \prec_{\text{lex}} R[i..]$ for $i \in [\mathsf{b}_R(T_x)..\mathsf{e}_R(T_x)]$ and $x \in [1..t-1]$ since $T_{x+1} \prec_{\text{lex}} T_x$.[6] Next, we define the equivalent to the LMS substrings for the inf-suffixes, which we call *LMS inf-suffixes*: For $1 \le i < j \le |T_x|+1$, the substring $(T_xT_x)[i..j]$ is called an *LMS inf-substring* if and only if $T_x[i..]$ and $T_x[j..]$ are $\mathtt{S}^*$ inf-suffixes and there is no $k \in [i+1..j-1]$ such that $T_x[k..]$ is an $\mathtt{S}^*$ inf-suffix. This definition differs from the original LMS substrings (omitting the last one $R[|R|..|R|]$ being a border case) only for the last LMS inf-substring of each Lyndon factor. Here, we append $T_x[1]$ instead of $T_{x+1}[1]$ to the suffix starting with the last type $\mathtt{S}^*$ position of T_x.

3.2.3 Example

The LMS inf-substrings of our running example $T :=$ `cbbcacbbcadacbadacba` with $R = T$ are given in Fig. 7. Their $\prec_{\text{LMS}}$-ranking is given on the left side of Fig. 8, where we associate each LMS inf-substring, except those consisting of a single character, with a non-terminal reflecting its rank. By replacing the LMS inf-substrings by their $\prec_{\text{LMS}}$-ranks in the text while discarding the single character Lyndon factors, we obtain the string $T^{(1)} :=$ `EBDCACA`, whose LMS inf-substrings are given on the right side of Fig. 7. Among these LMS inf-substrings, we only continue with `BDC` and `AC`. Since all LMS-inf substrings are distinct, their $\prec_{\text{LMS}}$-ranks determine the $\prec_\omega$-order of the $\mathtt{S}^*$ inf-suffixes as shown on the right side of Fig. 8. It is left to induce the $\mathtt{L}$ and $\mathtt{S}$ suffixes, which is done exactly as in the SAIS algorithm. We conduct these steps in Fig. 9, which finally lead us to $\mathsf{SA}_\circ$.

3.2.4 Correctness and Time Complexity

Let us recall that our task is to compute the $\prec_\omega$-order of the conjugates $\mathsf{conj}_{i_x-1}(T_x)$ for $i_x \in [1..|T_x|]$ of all Lyndon factors $T_1, \ldots, T_t$ of R. We will frequently use that $\mathsf{conj}_{i_x-1}(T_x) \prec_\omega \mathsf{conj}_{i_y-1}(T_y)$ is equivalent

[6] Consequently, for transforming SA into $\mathsf{SA}_\circ$, one only needs to shift values in SA to the right, as this is done by one of the implementations (`https://github.com/NealB/Bijective-BWT`) mentioned in the related work.

LMS Inf-Substring	Contents	Non-Terminal
$R[1]R[1]$	cc	-
$R[2..4]R[2]$	bbcb	E
$R[5..7]$	acb	B
$R[7..10]$	bbca	D
$R[10..11]R[5]$	ada	C
$R[12..15]$	acba	A
$R[15..16]R[12]$	ada	C
$R[17..19]R[17]$	acba	A
$R[20]R[20]$	aa	-

$\mathtt{S}^*$ Inf-Suffix	Contents
$R[20..]$	a...
$R[17..]$	acb...
$R[12..]$	acbad...
$R[5..]$	acbbcad...
$R[15..]$	adacb...
$R[10..]$	adacbbc...
$R[7..]$	bbcadac...
$R[2..]$	bbc...
$R[1..]$	c...

Figure 8: Ranking of the LMS inf-substrings and the $\mathtt{S}^*$ suffixes of our running example $T = R$ given in Sect. 3.2.3 and Fig. 7. *Left*: LMS inf-substrings assigned with non-terminals reflecting their corresponding rank in $\prec_{\text{LMS}}$-order. They have the same color as the respective rectangular brackets on the left of Fig. 7. The first and the last LMS substring do not receive a non-terminal since their lengths are one (remember that we omit Lyndon factors of length 1 in the recursive call). *Right*: $\mathtt{S}^*$ inf-suffixes of T sorted in $\prec_{\text{lex}}$-order, which corresponds to the $\prec_\omega$ of the conjugate starting with this inf-suffix. Compared with Fig. 3, the suffixes $R[2..20]$ and $R[7..20]$ in the $\prec_{\text{lex}}$-order are order differently than their respective inf-suffixes $R[2..]$ and $R[7..]$ in the $\prec_{\text{lex}}$-order.

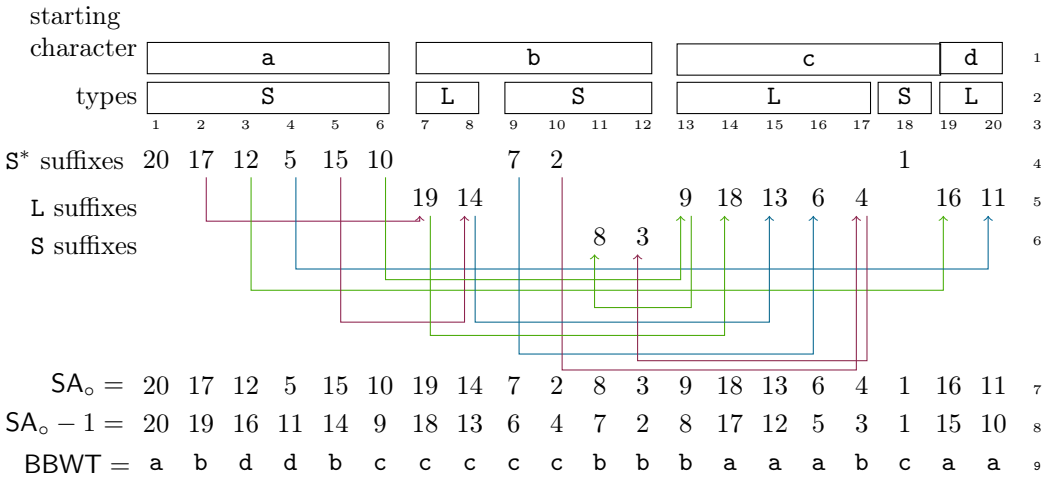

Figure 9: Inducing L and S inf-suffixes from the $\prec_{\text{lex}}$-order of the $\mathtt{S}^*$ inf-suffixes given in Fig. 7. Rows 1 and 2 show the partitioning of $\mathsf{SA}_\circ$ into buckets, first divided by the starting characters of the respective inf-suffixes, and second by the types L and S. Row 4 is $\mathsf{SA}_\circ$ after inserting the $\mathtt{S}^*$ inf-suffixes according to their $\prec_{\text{lex}}$-order rank obtained from the right of Fig. 8. The $\mathtt{S}^*$ (resp. L) inf-suffixes induce the L (resp. S) inf-suffixes in Row 5 (resp. Row 6). Putting all together yields $\mathsf{SA}_\circ$ in Row 7. In the penultimate row $\mathsf{SA}_\circ - 1$, each text position stored in $\mathsf{SA}_\circ$ is decremented by one, wrapping around a Lyndon factor if necessary (for instance, $(\mathsf{SA}_\circ - 1)[2] = 19 = \mathsf{e}_R(T_5)$ since $\mathsf{SA}_\circ[2] = 17 = \mathsf{b}_R(T_5)$). The last row shows $R[(\mathsf{SA}_\circ - 1)[i]]$ in its i-th column, which is the BBWT of R as given in Fig. 1.

to $T_x[i_x..] \prec_{\text{lex}} T_y[i_y..]$ for $i_x \in [1..|T_x|]$ and $i_y \in [1..|T_y|]$. We start with showing that the $\prec_{\text{LMS}}$-ranks of the LMS inf-substrings determine the $\prec_{\text{lex}}$-order of the $\mathtt{S}^*$ inf-suffixes[7], whenever the LMS inf-suffixes are all distinct.

Lemma 3.3. Let S_x and S_y be two LMS inf-substrings that are prefixes of $T_x[i_x..]$ and $T_y[i_y..]$, respectively, for $i_x \in [1..|T_x|]$ and $i_y \in [1..|T_y|]$. If $S_x \prec_{\text{LMS}} S_y$ then $T_x[i_x..] \prec_{\text{lex}} T_y[i_y..]$.

Proof. Given $S_x \prec_{\text{LMS}} S_y$, there is a position i such that (a) $S_x[i] < S_y[i]$ or (b) $S_x[i]$ is type $\mathtt{L}$ and $S_y[i]$ is type $\mathtt{S}$; let i be the smallest such position. In the latter case (b), there is a position $j > i$ such that $T_x[i_x + j - 1] = S_x[j] < S_x[i] = S_y[i] < S_y[j] = T_y[i_y + j - 1]$ and $T_x[i_x..i_x + j - 2] = T_y[i_y..i_y + j - 2]$, where we abused the notation that $T_x[k] = (T_xT_x\cdots)[k]$ for a $k \in [1..2|T_x|]$. In both cases (a) and (b), $T_x[i_x..] \prec_{\text{lex}} T_x[i_y..]$. □

Exactly as in the SAIS recursion step, we map each LMS inf-substring to its respective meta-character via its $\prec_{\text{LMS}}$-rank, obtaining a string $R^{(1)}$ whose characters are $\prec_{\text{LMS}}$-ranks. The lexicographic order $\prec_{\text{lex}}$ induces a natural order on the strings whose characters are drawn from the $\prec_{\text{LMS}}$-ranks. With that, we can determine the Lyndon factorization on $R^{(1)}$, which is given by the following connection:

Lemma 3.4. There is a one-to-one correspondence between Lyndon factors of R and $R^{(1)}$, meaning that each Lyndon factor of $R^{(1)}$ generates a Lyndon factor in R by expanding each of its $\prec_{\text{LMS}}$-ranks to the characters of the respective LMS inf-substring (while omitting the last character if it is the beginning of another LMS inf-substring), and vice-versa by contracting the characters of R to non-terminals.

Proof. We first observe that each LMS inf-substring is contained in $T_x[1..|T_x|]T_x[1]$ for an $x \in [1..t]$. Now, let L be a Lyndon factor of $R^{(1)}$ with $L = r_1 \cdots r_\ell$ such that each r_i is a $\prec_{\text{LMS}}$-rank. Suppose that there is a $d \in [1..\ell - 1]$ such that $r_1 \cdots r_d$ expands to a suffix $T_x[s..|T_x|]$ of T_x (again omitting the last character of each expanded LMS inf-substring) and $r_{d+1} \cdots r_\ell$ expands to a prefix P of T_{x+1}. Since L is a Lyndon word, $r_1 \cdots r_d \prec_{\text{lex}} r_1 \cdots r_\ell \prec_{\text{lex}} r_{d+1} \cdots r_\ell$. Hence, $T_x[s..|T_x|] \prec_{\text{LMS}} T_x[s..|T_x|]T_x[1] \prec_{\text{LMS}} P$, and with Lemma 3.3, $T_x[1..] \prec_{\text{lex}} T_x[s..] \prec_{\text{lex}} T_{x+1}[1..]$, contradicting the Lyndon factorization of R with Lemma 3.1.

Finally, suppose that a Lyndon factor L_1 of $R^{(1)}$ expands to a proper prefix of a Lyndon factor T_x. Let L_2 be its subsequent Lyndon factor, which has to end inside T_x according to the above observation. Then $L_2 \prec_{\text{lex}} L_1$, which means that T_x contains an inf-suffix smaller than T_x due to Lemma 3.1, contradicting that T_x is a Lyndon factor. □

Thanks to Lemma 3.4, we do not have to compute the Lyndon factorization of $R^{(1)}$ needed in the recursive step, but can infer it from the Lyndon factorization of R. Additionally, we have the property that the order of the LMS inf-substrings in the recursive step only depends on the Lyndon factors they are (originally) contained in. It remains to show how the $\prec_{\text{LMS}}$-ranks of the LMS inf-substrings can be computed:

Lemma 3.5. We can compute the $\prec_{\text{LMS}}$-ranks of all LMS inf-substrings in linear time.

Proof. We follow the proof of [64, Theorem 3.12]. The idea is to know the $\prec_{\text{lex}}$-order among some smallest $\mathtt{S}^*$ inf-suffixes with which we can induce the $\prec_{\text{LMS}}$-ranks of all LMS inf-substrings. Here, we use the one-to-one correlation between each LMS inf-substring $R[i..j]$ and the respective $\mathtt{S}^*$ inf-suffix $R[i..]$ by using the starting position i for identification. To compute the order of the (traditional) LMS substrings, it sufficed to know the lexicographically smallest $\mathtt{S}^*$ suffix (cf. Fig. 4), which can be determined by appending an artificial character such as $\mathtt{\$}$ to R with the property that it is smaller than all other characters appearing in R. Here, we need to know the order of at least one $\mathtt{S}^*$ inf-suffix per Lyndon factor. That is because an inf-suffix can only induce the order of another inf-suffix of the *same* Lyndon word. However, this is not a problem since we know that the inf-suffix starting with a Lyndon factor T_x is smaller in $\prec_\omega$-order than all other inf-suffixes of T_x, for each $x \in [1..t]$. In particular, we know that $T_x \succ_{\text{lex}} T_{x+1}$ is equivalent to $T_x \succ_\omega T_{x+1}$ due to [13, Thm. 8], and hence we know the $\prec_{\text{lex}}$-ranks among all inf-suffixes starting with the Lyndon factors.[8] In what follows, we use the inf-suffixes starting with the Lyndon factors to induce the $\prec_{\text{LMS}}$-ranks of all LMS inf-substrings.

[7] This is a counterpart to the property that the $\prec_{\text{LMS}}$-ranks of the LMS substrings determine the $\prec_{\text{lex}}$-order of the $\mathtt{S}^*$ suffixes [64, Theorem 3.12].

[8] Since T_t is the smallest Lyndon word, we have the invariants that $\mathsf{SA}_\circ[1] = \mathsf{b}_R(T_t)$ and $\mathsf{BBWT}[1] = R[\mathsf{e}_R(T_t)] = R[|R|]$.

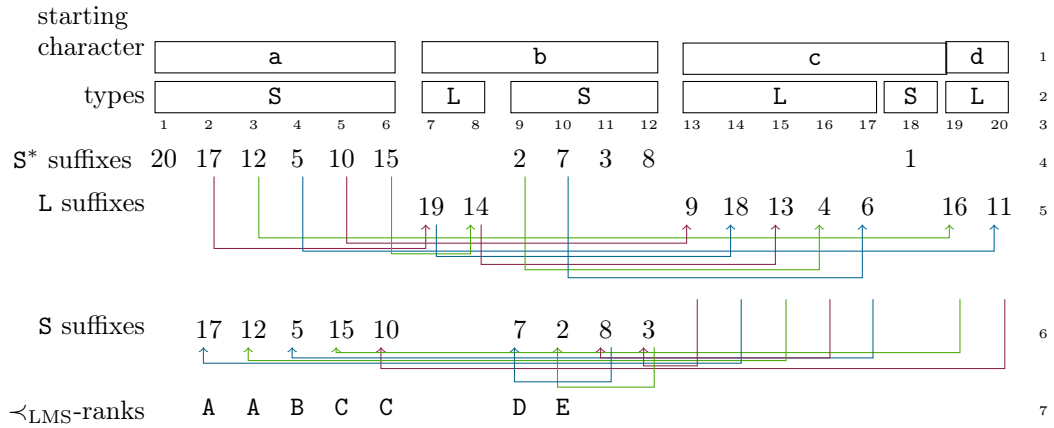

Figure 10: Inducing LMS inf-substrings. Thanks to the Lyndon factorization, we know the $\prec_\omega$-order of the inf-suffixes starting with the Lyndon factors, which is $T[20..] \prec_\omega T[17..] \prec_\omega T[12..] \prec_\omega T[5..] \prec_\omega T[2..] \prec_\omega T[1..]$. We insert the starting positions of these inf-suffixes in this order into their respective buckets, and fill the $\mathtt{S}^*$ buckets with the rest of $\mathtt{S}^*$ inf-suffixes by an arbitrary order (here we used the text order). Like Fig. 4, the $\mathtt{S}^*$ (resp. $\mathtt{L}$) suffixes induce the $\mathtt{L}$ (resp. $\mathtt{S}$) suffixes in Row 5 (resp. Row 6), but we skip those belonging to Lyndon factors of length one, since each of them is always stored at the leftmost position of its respective bucket. In the last row, we assign each LMS inf-substring a non-terminal based on its $\prec_{\mathrm{LMS}}$-rank, but omitting those that correspond to factors of length one.

However, the inducing only works if we include all text positions: While an ordered suffix $R[i..|R|]$ induces the order of $R[i-1..|R|]$ in the traditional SAIS, here we want an inf-suffix $R[i..]$ to induce the order of $R[i-1..]$. For that, we define a superset of the LMS inf-substrings, whose elements are called LMS-prefixes [64, Sect. 3.4]: Let $i \in [\mathsf{b}_R(T_x)..\mathsf{e}_R(T_x)]$ for an $x \in [1..t]$ be a text position, and let $j > i$ be the next $\mathtt{S}^*$ position in R. Then the *LMS-prefix* P_i starting at position i is $P_i := R[i..j]$ if $j \leq \mathsf{e}_R(T_x)$ or $P_i := R[i..j-1]\mathsf{b}_R(T_x)$ if $j = \mathsf{b}_R(T_{x+1})$. In particular, if i is the starting position of an LMS inf-substring S, then $P_i = S$. The LMS-prefixes inherit the types ($\mathtt{L}$ or $\mathtt{S}$) from their starting positions. We show that we can compute the $\prec_{\mathrm{LMS}}$-ranks of all P_i's by induced sorting:

Initialize the Suffix Array. We create $\mathsf{SA}_\circ$ of size $|R|$ to store the $\prec_{\mathrm{LMS}}$-ranks of all LMS-prefixes, where the entries are initially empty. Like in SAIS, we divide $\mathsf{SA}_\circ$ into buckets, and put the LMS-prefixes corresponding to the LMS inf-substrings into the $\mathtt{S}$ buckets of the respective starting characters in lexicographically sorted order. See also Fig. 10 for an example.

Inducing $\mathtt{L}$ LMS-prefixes. We scan the suffix array from left to right, and take action whenever we access a non-empty value i stored in $\mathsf{SA}_\circ$: Given $i \in [\mathsf{b}_R(T_x)..\mathsf{e}_R(T_x)]$ and $i^- = i-1$ or $i^- = \mathsf{e}_R(T_x)$ for $i = \mathsf{b}_R(T_x)$, we insert i^- into the $\mathtt{L}$ bucket of the character $T_x[i^-]$ if $R[i^-..]$ is an $\mathtt{L}$ inf-suffix. By doing so, we compute the $\prec_{\mathrm{LMS}}$ -order of all $\mathtt{L}$ LMS-prefixes in ascending lexicographic order per $\mathtt{L}$ bucket. The correctness follows by induction over the number k of inserted $\mathtt{L}$ LMS-prefixes. Since we know that all LMS-prefixes $P_{\mathsf{b}_R(T_x)}$ for $x \in [1..t]$ starting with the Lyndon factors are stored correctly in $\prec_{\mathrm{LMS}}$-order, and each of them is preceded by an $\mathtt{L}$ LMS-prefix, we perform the insertion of the first $\mathtt{L}$ LMS-prefix correctly, which is induced by the lexicographically smallest $\mathtt{S}^*$ LMS-prefix $P_{T_t[1]}$. For the induction step, assume that there is a $k > 1$ such that when we append the $(k+1)$-th $\mathtt{L}$ LMS-prefix P_i into its corresponding bucket, we have stored an $\mathtt{L}$ LMS-prefix P_j with larger $\prec_{\mathrm{LMS}}$-rank in the same bucket. In this case, we have that $R[i] = R[j]$, $P_{j+1} \succ_{\mathrm{LMS}} P_{i+1}$ and P_{j+1} is stored to the left of P_{i+1}. This implies that when we scanned $\mathsf{SA}_\circ$ from left to right, before appending P_i to its bucket, we already made a mistake.

The inducing step for the $\mathtt{S}$ LMS-prefixes works exactly in the same way by symmetry. Finally, we scan the computed $\mathsf{SA}_\circ$, and for each pair of subsequent positions i and j with $i < j$ corresponding to the starting positions of two LMS inf-suffixes, we perform a character-wise comparison whether the LMS inf-substring starting at i is $\prec_{\mathrm{LMS}}$-smaller than the one starting at j. By doing so, we can compute the

$\prec_{\text{LMS}}$-ranks of all LMS inf-substrings in linear time because the number of character comparisons is bounded by the number of characters covered by all LMS inf-substrings, which is $\mathcal{O}(|R|)$. □

With Lemma 3.5, we can determine the $\prec_\omega$ -order of the $\mathtt{S}^*$ inf-suffixes R. It is left to perform the induction step to induce first the order of the $\mathtt{L}$ inf-suffixes, and subsequently the $\mathtt{S}$ inf-suffixes, which we do in the same manner as SAIS, but access $(T_xT_x\cdots)[i^-]$ instead of $R[i-1]$ when accessing a suffix array entry with value i, where x chosen such that $i \in [\mathsf{b}_R(T_x)..\mathsf{e}_R(T_x)]$ and $i^- = i-1$ or $i^- = \mathsf{e}_R(T_x)$ if $i = \mathsf{b}_R(T_x)$. The correctness follows by construction: Instead of partitioning the suffixes into LMS substrings (maybe omitting a prefix of R with $\mathtt{L}$ suffixes), we refine the Lyndon factors into a partitioning of LMS inf-substrings.

Lyndon Factors of Length One. It is left to reintroduce the Lyndon factors of lengths one to obtain the complete $\mathsf{SA}_\circ$ of R. Remember that we omitted these factors at the recursive call. After the recursive call, we reinsert each of them at the smallest position in the $\mathtt{S}$ bucket of its respective starting character. By doing so, we correctly sort them due to the following observation: Suppose that there is a Lyndon factor consisting of a single character $\mathtt{b}$ (the following holds if $\mathtt{b} \in \Sigma$ or if $\mathtt{b}$ is a rank of an LMS substring considered in the recursive call). All LMS inf-substrings larger than one starting with $\mathtt{b}$ are larger than $\mathtt{bb}$ in the $\prec_\omega$-order because such an LMS inf-substring starting with $R[i]$ having type $\mathtt{S}^*$ is lexicographically smaller than $R[i+1..]$. Consequently, $\mathtt{bb}\cdots \prec_{\text{lex}} R[i..] = \mathtt{b}R[i+1..]$ since $\mathtt{b}\cdots \prec_{\text{lex}} R[i+1..]$. Thus, the Lyndon factor consisting of the single character $\mathtt{b}$ does not have to be tracked further in the recursive call since we know that its rank precedes the ranks of all other LMS inf-substrings starting with $\mathtt{b}$.

Time Complexity. By omitting Lyndon factors in the recursive calls, reducing R to a string R' where no two subsequent inf-suffixes $R[i..]$ and $R[i+1..]$ are $\mathtt{S}^*$, we can bound the maximum number of all $\mathtt{S}^*$ inf-suffixes by $n/2$ for the recursive call. After the recursion, we can insert all omitted LMS inf-substrings into the order returned by the recursive call by placing them at the beginnings of the respective $\mathtt{S}$ buckets. Hence, we obtain that $\mathcal{T}(n) = \mathcal{T}(n/2) + \mathcal{O}(n) = \mathcal{O}(n)$, where $\mathcal{T}(n)$ is the time complexity for computing a circular suffix array of length n. Note that the omission of the single character Lyndon factors is crucial for obtaining this time complexity. Without, there may be more than $n/2$ many $\mathtt{S}^*$ inf-suffixes, and because we keep the same Lyndon factorization in all recursive levels, we could have $\Theta(n)$ LMS inf-suffixes at each recursion level. The final step of computing the BBWT of T from the circular suffix array $\mathsf{SA}_\circ$ of R can be done in linear time with a linear scan of $\mathsf{SA}_\circ$ as described in Sect. 3.2.1.

3.2.5 Space Complexity

Given that $f = \sum_{x=1}^{t} \tau_x$ is the number of all non-composed Lyndon factors $F_1 \cdots F_f$, the algorithm of Lemma 2.2 computing the Lyndon factorization online only needs to maintain three integer variables of $\mathcal{O}(\lg n)$ bits to find $F_1 \cdots F_f$. We can represent the non-composed Lyndon factorization by a bit vector B of length n marking the ending position of each factor F_x ($x \in [1..f]$) with a one. We additionally create a bit vector B_2 of length f, and mark the first occurrence of each non-composed Lyndon factor F_x in B_2 for $x \in [1..f]$ such that B_2 stores t ones. Then the x-th '1' in B_2 corresponds to the x-th composed Lyndon factor T_x, and the number of '0's between the x-th and $(x+1)$-th '1' in B_2 is $\tau_x - 1$. It is now possible to replace T by R and store the Lyndon factorization of R in B (and resizing B to length $|R|$) since we can restore T later with B_2. (Alternatively, we can simulate R having T and B_2.) This saves at least $(f-t)\lg\sigma \geq f - t$ bits, such that our working space is at most $n + t + n\lg\sigma$ bits including the text space, before starting the actual algorithm computing $\mathsf{SA}_\circ$. Building a rank-support data structure on B helps us to identify the Lyndon factor covering a text position of R in constant time [46]. Since a recursive call of SAIS works on a text instance of at most $|R|/2$ characters, we can rebuild B from scratch by running the algorithm of Lemma 2.2 at the start of each recursive call. In total, we can maintain the Lyndon factorization in $n + o(n)$ bits with $\mathcal{O}(n)$ total time throughout all recursive calls. When a recursive call ends, we need to insert the omitted Lyndon factors of length one into the list of sorted $\mathtt{S}^*$ inf-suffixes. But this can be done with a linear scan of the sorted $\mathtt{S}^*$ inf-suffixes and their initial characters, since we know that the omitted Lyndon factors have to be inserted at the first position among all inf-suffixes sharing the same initial character. Additionally, we can achieve this within the space used for storing the circular suffix array $\mathsf{SA}_\circ$, since all $\mathtt{S}^*$ inf-suffixes use up at most half of the positions of the inf-suffix array. Overall, we have an algorithm running with $n + t + o(n)$ bits on top of our modified SAIS,

which uses $\mathcal{O}(\sigma \lg n)$ bits of working space additionally to $\mathsf{SA}_\circ$. If σ is not constant, one may consider an option to get rid of this additional space requirement. Luckily, we can do so with the in-place suffix array construction algorithm of Goto [40] (or similarly with [54]), which is a variation of SAIS, storing an implicit representation of these $\mathcal{O}(\sigma \lg n)$ bits within the space of $\mathsf{SA}_\circ$. Since B_2 is only needed for the final step computing the BBWT of T, we can compute $\mathsf{SA}_\circ$ with $n + o(n)$ additional bits of working space, and BBWT with $|\mathsf{SA}_\circ| + n + t + o(n)$ additional bits of working space, where $|\mathsf{SA}_\circ| = n \lg n$ denotes the size of $\mathsf{SA}_\circ$ in bits.

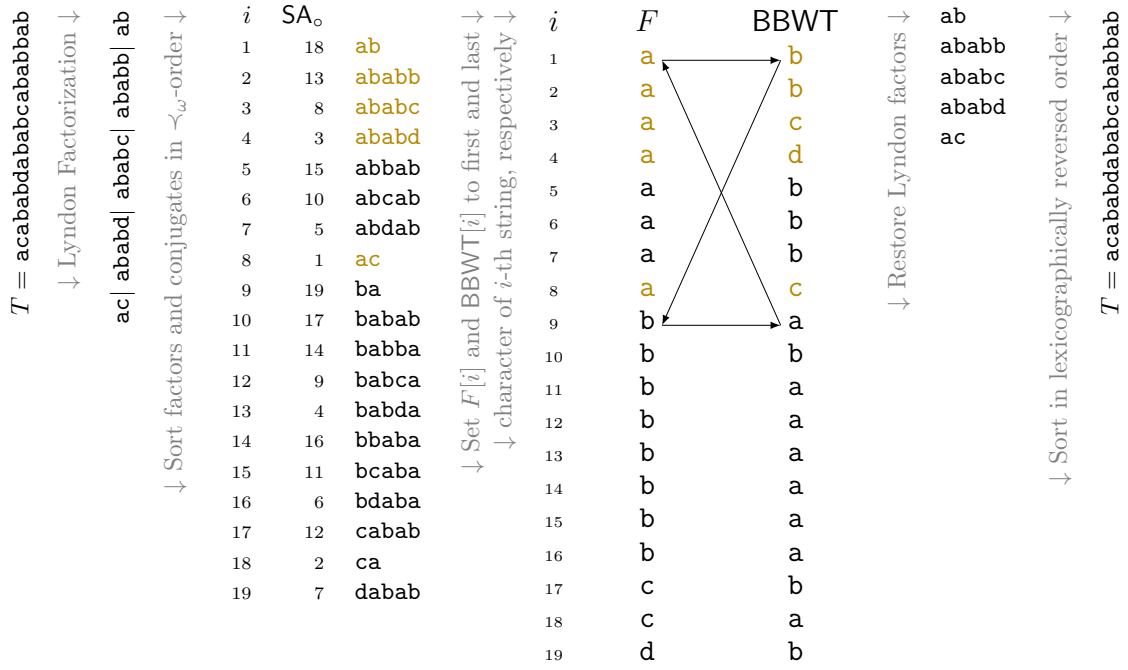

Figure 11: Constructing BBWT and restoring the original input. The Lyndon factors are highlighted (■). Middle: Restoring the Lyndon factor ab with the backward search, where the array F is defined by $F[i] := \mathsf{c}$ if $C[c-1]+1 \leq i \leq C[c]$. Right: Lyndon factors of T restored by visiting all cycles of BBWT.

Theorem 3.6. We can construct the bijective BWT of a string of length n in linear time.

Proof. Given a string T of length n, we compute its Lyndon factorization in $\mathcal{O}(n)$ time with Lemma 2.2. Subsequently, we build a representation R of T where duplicate Lyndon factors are removed. Our modified SAIS algorithm computes the circular suffix array $\mathsf{SA}_\circ$ of R in linear time. With a linear scan over the $\mathsf{SA}_\circ$ (and random access to both R and T) we can compute the BBWT of T as explained in Sect. 3.2.1. □

4 Indexing

Our task in this section is to build an index on our constructed BBWT. For the sake of explanatory purposes we switch to the different running example $T = \mathtt{acababdababcababbab}$, for which we illustrate the construction and the inversion of its BBWT in Fig. 11.

To find patterns, our index applies the same backward search as the FM-index [28], which we briefly review. Prior to that, we define some necessary concepts and data structures:

Additional Definitions. The *longest common prefix (LCP)* of two strings S and T is the longest string that is a prefix of both S and T. The length of the LCP of two strings S and T is given by the function $\mathsf{lcp}(S,T)$ returning an integer ℓ such that $T[1..\ell] = S[1..\ell]$ and either (a) $T[\ell+1] \neq S[\ell+1]$ or (b) $\ell = \min(|T|,|S|)$ holds.

Support Data Structures. Given a string $T \in \Sigma^*$, a character $\mathsf{c} \in \Sigma$, and an integer j, the *rank* query $T.\mathsf{rank}_\mathsf{c}(j)$ counts the occurrences of c in $T[1..j]$, and the *select* query $T.\mathsf{select}_\mathsf{c}(j)$ gives the position of the j-th c in T. We stipulate that $\mathsf{rank}_\mathsf{c}(0) = \mathsf{select}_\mathsf{c}(0) = 0$. An occurrence of a substring S in T is treated as a sub-interval of $[1..|T|]$ such that $S = T[\mathsf{b}_T(S)..\mathsf{e}_T(S)]$.

FM-Index. The FM-index uses BWT with the following auxiliary data structures:

- an array C with $\sigma \lg n$ bits, where $C[c]$ is the number of occurrences of those characters in T that are smaller than c (for each character c with $1 \le c \le \sigma$), and
- a data structure that supports rank queries on BWT.

The FM-index allows to query the rank of $T[i-1..]$ when having the rank of $T[i..]$ at hand, which is called LF-mapping, and formally defined as $\mathsf{LF}[i] = C[\mathsf{BWT}[i]] + \mathsf{BWT}.\mathsf{rank}_{\mathsf{BWT}[i]}(i)$.

Given a pattern P whose characters are drawn from Σ, the occurrences of P in T are represented by $\mathsf{range}(P)$ storing an interval of SA such that $\mathsf{SA}[i]$ is a starting position of an occurrence of P for each $i \in \mathsf{range}(P)$. More formally, $\mathsf{range}(P)$ denotes the range in BWT such that

$$T[\mathsf{SA}[j]..\mathsf{SA}[j] + |P| - 1] = P \text{ if and only if } j \in \mathsf{range}(P). \tag{1}$$

We obtain $\mathcal{I}_i = \mathsf{range}(P[i..])$ from $\mathcal{I}_{i+1} = \mathsf{range}(P[i+1..])$ with a *backward search* step

$$\mathsf{b}_T(\mathcal{I}_i) = C[P[i]] + \mathsf{BWT}.\mathsf{rank}_{P[i]}(\mathsf{b}_T(\mathcal{I}_{i+1}) - 1) + 1 \text{ and } \mathsf{e}_T(\mathcal{I}_i) = C[P[i]] + \mathsf{BWT}.\mathsf{rank}_{P[i]}(\mathsf{e}_T(\mathcal{I}_{i+1})). \tag{2}$$

Here, $\mathsf{b}_T(\mathcal{I}_i)$ and $\mathsf{e}_T(\mathcal{I}_i)$ denote the beginning and the end of $\mathcal{I}_i$, i.e., $\mathcal{I}_i = [\mathsf{b}_T(\mathcal{I}_i)..\mathsf{e}_T(\mathcal{I}_i)]$. We stipulate that the range of the empty string is $[1..n]$. Starting with the range of the empty string $\mathsf{range}(P[|P|+1..])$ and applying Eq. (2) iteratively, we can find all occurrences of the pattern P in T with $|P|$ rank operations.

In particular, let p_b be the position of the first occurrences of $P[i]$ in BWT succeeding $\mathsf{b}_T(\mathcal{I}_i)$ and p_e the position of the first occurrences of $P[i]$ in BWT preceding $\mathsf{e}_T(\mathcal{I}_i)$ (such that $P[i] = \mathsf{BWT}[p_b] = \mathsf{BWT}[p_e]$). Then $\mathsf{b}_T(\mathcal{I}_{i+1}) = \mathsf{LF}[p_b]$ and $\mathsf{e}_T(\mathcal{I}_{i+1}) = \mathsf{LF}[p_e]$ if $p_b \le p_e$.

The LF-mapping with the extended BWT and the inversion of the extended BWT has already been studied by Mantaci et al. [59, Section 3]. In particular, subsequent positions in the eBWT within the same character run are mapped contiguously and in the same order by the backward search step. However, the extended BWT reports the cyclic occurrences of the pattern appearing in each of the cyclic primitive strings in the input multiset. When building an index on the BBWT for classic pattern matching on the original string, reporting these cyclic occurrences is not desired. Accidentally finding these cyclic occurrences happens when *rewinding* from the first position of a factor to its last position. In the rest of the paper, let LF denote the LF mapping with the BBWT of T. Suppose that we matched an occurrence of $P[i+1..]$ starting at position $j+1$ in T.

- If both text positions j and $j+1$ are contained in a Lyndon factor F_x for an integer x with $1 \le x \le t$, the LF mapping
$$\mathsf{LF}[\mathsf{ISA}_\circ[j+1]] = C[P[i]] + \mathsf{BBWT}.\mathsf{rank}_{P[i]}(\mathsf{ISA}_\circ[j+1]) \tag{3}$$
yields the occurrence of $P[i..]$ starting at position j in T. Here $\mathsf{ISA}_\circ$ denotes the *inverse circular suffix array* of T, which is defined as the inverse of $\mathsf{SA}_\circ$, i.e., $\mathsf{ISA}_\circ[\mathsf{SA}_\circ[i]] = i$ for every $i \in [1..n]$.
- Otherwise, j and $j+1$ are contained in two different Lyndon factors. Let F_x be the Lyndon factor with $\mathsf{b}_T(F_x) = j+1$ (and hence the text position j is contained in F_{x-1}). Then the LF mapping gives $\mathsf{ISA}_\circ[\mathsf{e}_T(F_x)]$, i.e., the starting position of the last conjugate of F_x (in SA-order, cf. the cycle representing the Lyndon factor $\mathtt{ab}$ in Figure 11).

We call the second case *rewinding*, as LF counts down from the i-th conjugate to the $(i-1)$-th conjugate, but *rewinds* from the zeroth-conjugate (i.e., F_x itself) to the last conjugate. Whenever we expect that no rewinding will happen, we can find a pattern with the backward search of the FM-index:

Lemma 4.1. Given a text T and a pattern P such that each occurrence of P in T is contained in a Lyndon factor of T, we can compute these occurrences with the backward search of the FM-index on the BBWT with $2|P|$ rank operations or $|P|$ backward search steps.

Proof. Since all occurrences of P are contained in Lyndon factors of T, the backward search finds no occurrence of $P[i..]$ starting at the beginning $\mathsf{b}_T(F_x)$ of a Lyndon factor F_x in T, for $2 \le i \le |P|$ and $1 \le x \le t$. □

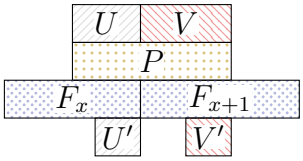

Figure 12: Setting of the proof of Lemma 4.3.

4.1 Lyndon Patterns

We first focus on the special case that the pattern itself is a Lyndon word. Subsequently, we show the general case (Sect. 4.2) by applying the Lyndon factorization to the pattern P and introduce an enhancement to the backward search for obtaining $\mathsf{range}(P[i..])$ from $\mathsf{range}(P[i+1..])$ in the case that the suffix $P[i+1..]$ starts with a Lyndon factor of T. For all this we need a little helper lemma:

Lemma 4.2 ([27, Prop. 1.10])**.** The longest prefix of T that is a Lyndon word is the first Lyndon factor F_1 of T. Given $\mathsf{LynF}(T) = \{F_1, \ldots, F_f\}$, $\mathsf{LynF}(T) = \{F_1\} \cup \mathsf{LynF}(F_2 \cdots F_f)$.

Lemma 4.3. Let T be a string with $\mathsf{LynF}(T) = \{F_1, \ldots, F_f\}$, and let P be a pattern. If P is a Lyndon word, then there is no occurrence of P in T that crosses the border of two Lyndon factors, i.e., each occurrence of P in T is contained in a Lyndon factor F_x $(1 \leq x \leq t)$.

Proof. Assume to the contrary that $P = UV$, where $U \in \Sigma^+$ is a suffix of F_x and $V \in \Sigma^+$ is a prefix of $F_{x+1} \cdots F_f$ for an integer x with $1 \leq x < t$. This setting is illustrated in Fig. 12.

Since F_x is the longest Lyndon prefix of $F_x \cdots F_f$ (see Lemma 4.2), it is not possible that $U = F_x$ (otherwise we could extend F_x to UV to form a longer Lyndon word). We conclude that U is a proper suffix of F_x. Since F_x is a Lyndon word, we have $F_x \prec U'$ for every proper suffix U' of F_x (including U). This implies that $F_xV \prec U'V$, and in particular $F_xV \prec UV$. Since the pattern P is a Lyndon word, we have $V' \succ P = UV \succ F_xV$ for every suffix V' of V (including V itself).

Putting everything together, we have that F_xV is lexicographically smaller than its proper suffixes, and F_xV thus is a Lyndon word. However, this again contradicts the setting that F_x is the longest Lyndon prefix of $F_x \cdots F_f$. □

Combining this result with Lemma 4.1 yields:

Corollary 4.4. Given a pattern P that is a Lyndon word, we can find all its occurrences with $|P|$ rank operations of the FM-index built on BBWT.

4.2 General Case

To find arbitrary patterns, we need to understand what happens during the rewinding. For that we show that they have to happen at consecutive Lyndon factors with the following lemma.

Lemma 4.5. If a pattern P is a prefix of F_x and F_f, then P is a prefix of F_y for each integer y with $x \leq y \leq f$.

Proof. Suppose that we matched $P[i..]$ in T with the backward search. Further, suppose that an occurrence of $P[i..]$ starts at position $\mathsf{b}_T(F_y)$ in T. Then we claim that F_y belongs to a consecutive set of Lyndon factors $F_x, \ldots, F_f$ with $x \leq y \leq f$ such that there is an occurrence of $P[i..]$ starting at position $\mathsf{b}_T(F_{y'})$ in T for each Lyndon factor $F_{y'}$ with $x \leq y' \leq f$. Figure 13 visualizes this setting. Assume that our claim is not true. Then there is an index y' with $x \leq y' \leq f$ for which there is no occurrence of $P[i..]$ starting at position $\mathsf{b}_T(F_{y'})$ in T. This contradicts $F_x \succeq F_{y'} \succeq F_f$. □

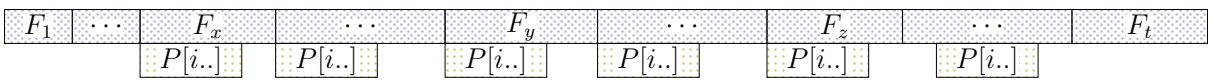

Figure 13: Suffix $P[i..]$ of pattern P matches the beginnings of some Lyndon factors of T. These Lyndon factors $F_x, \ldots, F_f$ are all consecutive.

Suppose that we matched $P[i..]$ and that there are occurrences of $P[i..]$ starting with Lyndon factors of T. These Lyndon factors are consecutive according to Lemma 4.5. Let these Lyndon factors be $F_x, \ldots, F_f$. Moreover, $P[i..]$ starts with a Lyndon factor of P according to Lemma 4.3, i.e., $P[i..] = \mathrm{lfs}_P(w)$ for an integer w with $1 \leq w \leq p$. A further backward search step causes a rewinding for all occurrences of $P[i..]$ starting at $\mathsf{b}_T(F_x), \ldots, \mathsf{b}_T(F_f)$, where the following cases can occur:

- If $T[\mathsf{e}_T(F_f)] = P[i-1]$, but $T[\mathsf{b}_T(F_{f+1})..]$ does not have $\mathrm{lfs}_P(w)$ as a prefix, then the backward search carries on a *false occurrence*.
- If $T[\mathsf{b}_T(F_x) - 1] = P[i-1]$, we would expect that the backward search reports that an occurrence of $P[i-1..]$ starts at $T[\mathsf{b}_T(F_x) - 1]$ (we assume that $T[\mathsf{e}_T(F_x)] = P[i-1]$). However, this is not the case because of the rewinding, either reporting the text position $\mathsf{e}_T(F_x)$ or dismissing this occurrence if $T[\mathsf{e}_T(F_x)] \neq P[i-1]$. In either case, we say that there is a *missed occurrence* of $P[i-1..]$ starting at $\mathsf{b}_T(F_x) - 1$.
- If $T[\mathsf{e}_T(F_y)] = P[i-1]$ but $T[\mathsf{e}_T(F_{y+1})] \neq P[i-1]$ for an integer y with $x \leq y \leq f-1$, then the rewinding discards the occurrence of $P[i..]$ starting at $T[\mathsf{b}_T(F_{y+1})]$ although $T[\mathsf{b}_T(F_{y+1}) - 1] = T[\mathsf{e}_T(F_y)] = P[i-1]$. This looks like that the occurrence of $P[i..]$ starting at $T[\mathsf{b}_T(F_{y+1})]$ becomes a missed occurrence. However, since $T[\mathsf{b}_T(F_y)]$ and $T[\mathsf{b}_T(F_{y+1})]$ are the starting positions of occurrences of $P[i..]$, the occurrence of $P[i..]$ starting at $T[\mathsf{b}_T(F_y)]$ takes over the job from the occurrence starting at $T[\mathsf{b}_T(F_{y+1})]$ after the rewinding, i.e., we obtain the starting position $T[\mathsf{e}_T(F_y)] = T[\mathsf{b}_T(F_{y+1}) - 1]$ of the occurrence of $P[i-1..]$ after rewinding it.
- Similarly, the setting $T[\mathsf{e}_T(F_y)] \neq P[i-1]$ but $T[\mathsf{e}_T(F_{y+1})] = P[i-1]$ for an integer y with $x \leq y \leq f-1$ seems to cause a false occurrence after rewinding the occurrence of $P[i..]$ starting at $\mathsf{b}_T(F_y)$, but actually this occurrence takes over the job from the occurrence starting at $T[\mathsf{b}_T(F_{y+1})]$.
- In all other cases, for $x+1 \leq y \leq f$, $T[\mathsf{e}_T(F_{y-1})]\mathrm{lfs}_P(w) = T[\mathsf{e}_T(F_y)]\mathrm{lfs}_P(w)$, i.e., the rewound positions are beginning positions of occurrences of $P[i-1..]$, where the occurrence of $P[i..]$ starting at $\mathsf{b}_T(F_{y-1})$ takes the job from the occurrence starting at $\mathsf{b}_T(F_y)$ after the rewinding.

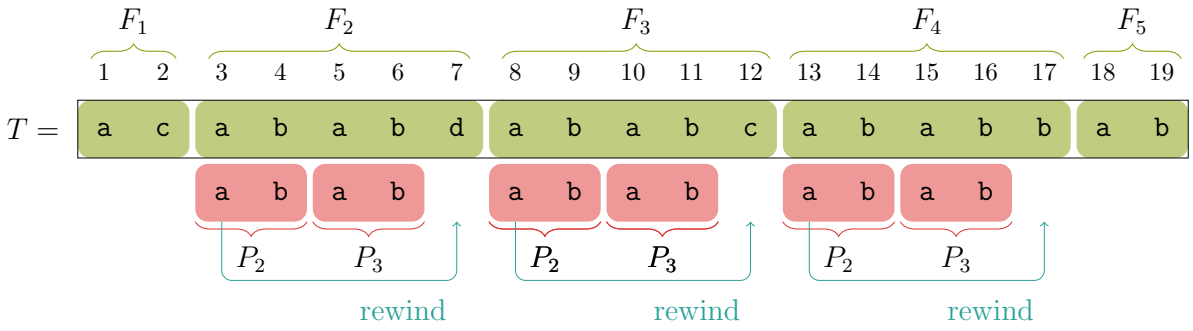

Figure 14: Backward search of a pattern P with $\mathrm{LynF}(P) = \{P_1, P_2 = \mathtt{ab}, P_3 = \mathtt{ab}\}$ in our running example $T = \mathtt{acababdababcababbab}$ after $|P_2P_3|$ steps. The sub-pattern P_2P_3 has occurrences starting at the starting positions of the Lyndon factors F_2, F_3, and F_4 of the text. The effects of the rewinding depend on P_1. If P_1 ends with $\mathtt{c}$, then we derive a *missed* occurrence from $\mathrm{lfs}_P(2)$ and F_2. If P_1 ends with $\mathtt{b}$, then we derive a *false* occurrence from $\mathrm{lfs}_P(2)$ and F_4.

In what follows, we study ways to limit the number of false and missed occurrences. We say that a false (resp. missed) occurrence of P is *derived from* P_w and F_f (resp. F_x) if it emerges on the rewinding at $T[\mathsf{b}_T(F_f)]$ (resp. $T[\mathsf{b}_T(F_x)]$). See Fig. 14 for an example. According to Lemma 4.3 there are at most p rewindings, and hence at most p false and missed occurrences. (We lower this upper bound in the subsequent section). The false occurrences can be easily maintained in a separate list, in which each element corresponds to a false occurrence (more precisely, applying SA to such an element yields its corresponding starting position in the text). Each element of the list is subject to the backward search (Eq. (3)) like the range itself (Eq. (2)). Whenever a backward search step of an element of the list yields

not an occurrence (e.g., we obtain the element $\mathsf{ISA}_\circ[j]$ by a backward search step from $P[i+1..]$ to $P[i..]$, but find out that $T[j] \neq P[i]$), then the false occurrence will also vanish from the range such that we no longer need to manage that element. Similarly, we keep track of the missed occurrences. For that, we take advantage of the fact that the entries of BBWT corresponding to Lyndon factors are lexicographically sorted (see the dark yellow marked entries in Fig. 11). To move from the beginning of a Lyndon factor to the end of its preceding Lyndon factor, it suffices to locate the previously larger Lyndon factor and apply a backward search step on it (to intentionally cause a rewinding). For that, we add a bit vector B_L marking the entries in BBWT corresponding to a Lyndon factor (and not to one of its conjugates) with '1'. Then $B_\mathrm{L}.\mathsf{select}_1(t-x+1)$ corresponds to F_x and the position $\mathsf{ISA}_\circ[\mathsf{b}_T(F_x)-1] = \mathsf{ISA}_\circ[\mathsf{e}_T(F_{x-1})]$ is found by applying a backward search step to $B_\mathrm{L}.\mathsf{select}_1(t-x)$. Again, we keep the missed occurrences in a list whose elements are (each individually) subject to the backward search. Finally, when we want to report all occurrences of the complete pattern, we take the computed range $\mathsf{range}(P)$, add all elements of the list of missed occurrences, and remove all elements of the list of the false occurrences. By doing so, we can restore the property of Eq. (1). With the lists for the missed and false occurrences and the bit vector B_L, we can state the following theorem generalizing the backward search for arbitrary patterns.

Example 4.6. In what follows, we present a step-by-step example for pattern matching. For the purpose of explanation, let $\mathcal{F}$ and $\mathcal{M}$ denote the set of false occurrences and missed occurrences, respectively. We denote with $\mathsf{b}(\mathcal{I})$ and $\mathsf{e}(\mathcal{I})$ the starting and ending positions of an interval $\mathcal{I}$, i.e., $\mathcal{I} = [\mathsf{b}(\mathcal{I})..\mathsf{e}(\mathcal{I})]$.

We reuse our running example $T = \texttt{acababdababcababbab}$ already appeared in Figs. 11 and 14 for the BBWT construction and the matching of $\texttt{b} \cdot \texttt{ab} \cdot \texttt{ab}$, respectively. Let us recall that the Lyndon factorization of T is given by $T = F_1 \cdot F_2 \cdot F_3 \cdot F_4 \cdot F_5$, where $F_1 = \texttt{ac}$, $F_2 = \texttt{ababd}$, $F_3 = \texttt{ababc}$, $F_4 = \texttt{ababb}$ and $F_5 = \texttt{ab}$. For our example, C and $\mathsf{BBWT}.\mathsf{rank}_\mathtt{X}(i)$ with $\mathtt{X} \in \Sigma, i \in [1..n]$ evaluates as follows, where $\mathsf{SA}_\circ$ is given in Fig. 11, and the inverse $\mathsf{SA}_\circ^{-1}$ of $\mathsf{SA}_\circ$ is obtained by $\mathsf{SA}_\circ^{-1}[\mathsf{SA}_\circ[i]] = i$.

X	C[X]
a	0
b	8
c	16
d	18

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$T[j]$	a	c	a	b	a	b	d	a	b	a	b	c	a	b	a	b	b	a	b
$\mathsf{SA}_\circ^{-1}[j]$	8	18	4	13	7	16	19	3	12	6	15	17	2	11	5	14	10	1	9

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$\mathsf{SA}_\circ[i]$	18	13	8	3	15	10	5	1	19	17	14	9	4	16	11	6	12	2	7
$\mathsf{BBWT}[i]$	b	b	c	d	b	b	b	c	a	b	a	a	a	a	a	a	b	a	b

$\mathsf{BBWT}.\mathsf{rank}_\mathtt{X}(i)$

X \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a	0	0	0	0	0	0	0	0	1	1	2	3	4	5	6	7	7	8	8
b	1	2	2	2	3	4	5	5	5	6	6	6	6	6	6	6	7	7	8
c	0	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2
d	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

We now consider matching two different types of patterns.

Example (a). Consider the pattern $P = \texttt{acab}$.

P has a single occurrence in T starting at the position 1 represented by $\mathsf{SA}_\circ[8] = 1$. Its Lyndon factorization is given as $P = P_1 \cdot P_2$, where $P_1 = \texttt{ac}$ and $P_2 = \texttt{ab}$. We search for the occurrences of P by iteratively processing all suffixes of P and consecutively applying Eq. (2).

$\mathcal{I}_i$	Suffix	$\mathsf{b}(\mathcal{I}_i)$	$\mathsf{e}(\mathcal{I}_i)$	$\mathcal{M}$	Remarks
$\mathcal{I}_5$	ε	1	19		
$\mathcal{I}_4$	b	9	16		
$\mathcal{I}_3$	ab	1	7		
$\mathcal{I}_2$	cab	17	17	{18}	- occurrences of cab in T cross Lyndon factor boundaries - have missed occurrence of cab between F_1F_2 with $T[\mathsf{SA}_\circ[18]] = T[2] = \texttt{c}$ - (the ending position of F_1 in $\mathsf{SA}_\circ$ is stored at index 18)
$\mathcal{I}_1$	acab	8	7	{8}	- resulting interval is empty - perform the backward search for each missed occurrence in $\mathcal{M}$

Comments. For $\mathcal{I}_3$, we matched $P[3..]$ with the beginning of F_x for all $x \in [2..5]$ (we have also other matches inside factors, which are, however, not of importance here). A backward search step can cause a

missed occurrence at the end of F_1 and a false occurrence at the end of F_5. We check that as follows: For the backward search for $P[2] = \mathtt{c}$ leading us to $\mathcal{I}_2$, we observe that we have one missed occurrence at F_1 because F_1's last character is `c`. However, we have no false occurrence at F_5 because F_5's last character is not `c`. The BBWT position 17 is *not* a false occurrence because it maps to the last position of F_3 ($\mathsf{SA}_\circ[17] = 12$), and we previously matched $P[3..]$ with a prefix of F_4.
Result. Our algorithm reports the empty interval and the set of missed occurrences $\mathcal{M} = \{8\}$. We conclude that there exists exactly one occurrence of the pattern (i.e., the missed occurrences) starting at $\mathsf{SA}_\circ[8] = 1$ in the text.
Variation. If the pattern is `cab`, then we can stop after the computation of the interval $\mathcal{I}_2$ with $\mathsf{b}(\mathcal{I}_2) = 17$ and $\mathsf{e}(\mathcal{I}_2) = 17$ and the set of missed occurrences $\mathcal{M} = \{18\}$. Hence, there are two occurrences of the pattern `cab` starting at $\mathsf{SA}_\circ[17] = 12$ and $\mathsf{SA}_\circ[18] = 2$. The former occurrence is not a false occurrence since F_3 is not the last Lyndon factor in the range when we matched the pattern suffix `ab`.

Example (b). Consider the pattern $P = \mathtt{babab}$, a specialization of Fig. 14.

Its Lyndon factorization is given as $P = P_1 \cdot P_2 \cdot P_3$, where $P_1 = \mathtt{b}$, $P_2 = \mathtt{ab}$ and $P_3 = \mathtt{ab}$. P has no occurrence in T, and thus we expect that searching leads to an empty interval in $\mathsf{SA}_\circ(T)$. As we will see, this is not the case — rather the subtraction of the false occurrences from the returned interval gives an empty interval. To start with, we search for the occurrences of P by iteratively processing all suffixes of P and consecutively applying Eq. (2).

$\mathcal{I}_i$	Suffix	$\mathsf{b}(\mathcal{I}_i)$	$\mathsf{e}(\mathcal{I}_i)$	$\mathcal{F}$	Remarks
$\mathcal{I}_6$	ε	1	19		
$\mathcal{I}_5$	`b`	9	16		
$\mathcal{I}_4$	`ab`	1	7		
$\mathcal{I}_3$	`bab`	9	13	$\{9\}$	- occurrences of `bab` in T cross the boundaries between factors P_1 and P_2 of P and the factors F_2, F_3, F_4 and F_5 of T - derive a false occurrence from P_2 and F_5
$\mathcal{I}_2$	`abab`	1	4	$\{1\}$	- perform the backward search for each false occurrence in $\mathcal{F}$
$\mathcal{I}_1$	`babab`	9	10	$\{9, 10\}$	- perform the backward search for each false occurrence in $\mathcal{F}$ - derive another false occurrence by rewinding F_4 because $T[\mathsf{e}(F_4)] = \mathtt{b}$ but F_5 does not start with `babab`

Comments. The range $\mathcal{I}_4$ is the same as $\mathcal{I}_3$ in the previous example (a). We check missed/false occurrences as follows: For the backward search for $P[3] = \mathtt{b}$ leading us to $\mathcal{I}_3$, we observe that we have no missed occurrence at F_1 because F_1's last character not `b`. However, we have a false occurrence at F_5 because F_5's last character is `b`. The BBWT position 10 is *not* a false occurrence because it maps to the last position of F_4 ($\mathsf{SA}_\circ[10] = 17$), and we previously matched $P[4..]$ with a prefix of F_5.
Result. Our algorithm reports the interval $[9..10]$ and the set of false occurrences $\mathcal{F} = \{9, 10\}$. To obtain the final result we return for the query, we need to merge both information, i.e., we need to subtract $\mathcal{F}$ from the reported interval. Thus, we obtain an empty set of matches, which we return as our result. This coincides with the backward search on the classic BWT, where the final interval is empty for this pattern.
Variation. For an alternative and final example, let us consider searching the pattern suffix `abab`. Then we can stop the search after computing the interval $\mathcal{I}_2$ with $\mathsf{b}(\mathcal{I}_2) = 1$ and $\mathsf{e}(\mathcal{I}_2) = 4$. The set of false occurrences $\mathcal{F} = \{1\}$. Thus, the three occurrences of `abab` in T are starting at $\mathsf{SA}_\circ[2] = 13, \mathsf{SA}_\circ[3] = 8$, and $\mathsf{SA}_\circ[4] = 3$.

Theorem 4.7. Given a text T and a pattern P, we can compute all occurrences of P in T with the FM-index built on BBWT with $\mathcal{O}(|P|p)$ rank operations, where p is the number of Lyndon factors of P.

In the following, we improve the $\mathcal{O}(|P|p)$ bound on the number of rank operations. There is a problem with matching a pattern whose Lyndon factorization consists of the same Lyndon factor that is equal to some Lyndon factors of the text. An example for such a case is given by $T = P = \mathtt{a}^n$. Here, P has n Lyndon factors, and therefore our current upper bound on the number of rank operations stated in

Thm. 4.7 is only $\mathcal{O}(n^2)$. Multiple occurrences of the same Lyndon factors (a) in the text as well as (b) in the pattern make the matching difficult. However, as we will see, we can cope with both individually.

First, we start with (a) the text; (b) is treated in Sect. 4.3. Our solution is to build the bijective BWT on all *distinct* Lyndon factors of T (along with their conjugates), remembering the number of occurrences of a Lyndon factor, such that the Lyndon factorization $T = F_1 \cdots F_f$ becomes $T = T_1^{\tau_1} \cdots T_t^{\tau_t}$, where $T_1, \cdots T_t$ are distinct Lyndon words with $T_x \prec T_{x+1}$ for $1 \leq x \leq t-1 \leq f-1$, and for every $1 \leq x \leq t$ it holds that (a) $\tau_j \geq 1$ and (b) there is an integer y with $y \geq x$ such that $T_x = F_y$. Remembering the definition in Sect. 3.2.1, the set $\{T_1^{\tau_1}, \ldots, T_t^{\tau_t}\}$ is called the *composed* Lyndon factorization of T. Given $T_x = F_y$, we stipulate that $\mathsf{b}_T(T_x)$ is the starting position of the leftmost Lyndon factor $F_{y-\tau_x+1}$ with $F_{y-\tau_x+1} = T_x$. For instance, the composed Lyndon factorization of $T = \mathtt{bbababababa}$ is $T = T_1^2\, T_2^3\, T_3$ with $T_1 = \mathtt{b}$, $T_2 = \mathtt{ab}$, and $T_3 = \mathtt{a}$. The starting position $\mathsf{b}_T(T_2)$ is 3.

Now suppose that the Lyndon factor P_w occurs k_w times in $\mathsf{LynF}(P)$, and suppose that P_w is the rightmost occurrence of them, i.e., $P_{w-k_w} \neq P_{w-k_w+1} = \ldots = P_{w-1} = P_w \neq P_{w+1}$. Whenever we match $\mathsf{lfs}_P(w)$ with the beginning of the rightmost Lyndon factor F_y equal to T_x with $|T_x| \leq |\mathsf{lfs}_P(w)|$ occurring τ_x times in T, we can directly match $P_w^{k_w-1}\mathsf{lfs}_P(w)$ if $\tau_x \geq k_w$, skipping the backward search for $P[\mathsf{b}_P(P_{w-k_w+1})..\mathsf{b}_P(P_w)]$ such that we directly match $P[\mathsf{b}_P(P_{w-k_w+1})..]$ for one occurrence O starting at $T[\mathsf{b}_T(T_x)]$. For that, we assumed that $T_x = P_{w-j}$ for every integer j with $0 \leq j \leq k_w - 1$. This is true due to the following lemma:

Lemma 4.8. Given an occurrence of P_w that starts at position $\mathsf{b}_T(F_x)$ in T, $\mathsf{lfs}_P(w)$ is not a proper prefix of F_x if and only if $P_w = F_x$.

Proof. Assume that $\mathsf{lfs}_P(w)$ is not a proper prefix of F_x. By switching the roles of P and T in Lemma 4.3, we obtain the result that F_x cannot cross the border between P_w and P_{w+1}. Since $|\mathsf{lfs}_P(w)| \geq |F_x|$, it holds that $P_w = F_x$ (otherwise we could extend F_x to a longer Lyndon factor). □

The further matching of the occurrence O is conducted separately to the backward search with the range of occurrences $\mathsf{range}(P[\mathsf{b}_P(P_w)..])$. If $\tau_x < k_w$, then we cannot extend the currently matched occurrence, and thus can ignore to follow this occurrence. We call this technique of skipping consecutive Lyndon factors a *composed jump*.

The composed jump allows us to proceed as follows: We only count missed occurrences O that were not derived from a missed occurrence (i.e., an occurrence belonging to the range and not part of the list of missed occurrences), which we call in the following *freshly* missed occurrences. We do not count a missed occurrence O' that is derived from a missed occurrence O. Instead, we only update the position of O to O' in the list of missed occurrences. This is justified as we cannot create a freshly missed occurrence during a later backward search step:

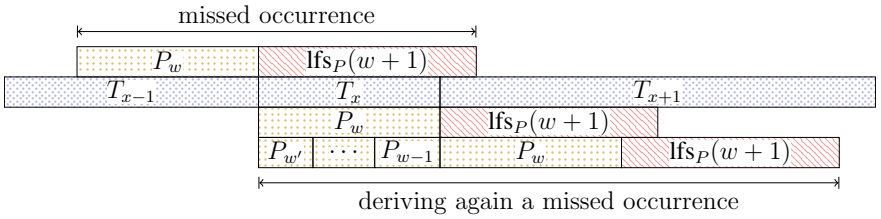

Figure 15: Setting of the proof of Lemma 4.9 where a false occurrence from $\mathsf{lfs}_P(w')$ and T_x is derived after a false occurrence was derived from $\mathsf{lfs}_P(w)$ and T_x with $w' < w$. However, this is not possible since then $T_{x-1} = T_x$.

Lemma 4.9. Let an occurrence of $\mathsf{lfs}_P(w)$ start at position $\mathsf{b}_T(T_x)$ in T and let $|T_x| < |\mathsf{lfs}_P(w)|$. If there is a missed occurrence derived from $\mathsf{lfs}_P(w)$ and T_x, there is no $w' < w$ such that $\mathsf{lfs}_P(w')$ and T_x derive a *freshly* missed occurrence.

Proof. By Lemma 4.8, $P_w = T_x$. Assume that there is a freshly missed occurrence derived from $\mathsf{lfs}_P(w')$ and T_x for the largest such w' with $w' < w$. Then $P_{w'} \cdots P_{w-1} = P_w$ and $\mathsf{lfs}_P(w') = P_w\mathsf{lfs}_P(w) = P_wP_w\mathsf{lfs}_P(w+1)$. Hence, $P_w = T_x$ is a suffix of T_{x-1} (cf. Fig. 15). Since T_{x-1} is a Lyndon word with $T_x \succeq T_{x-1}$, $T_{x-1} = T_x = P_w = P_{w-1}$ must hold. However, this contradicts the distinctness of T_x in the composed Lyndon factorization. □

4.3 Improving the Number of Ranks

In this section, we study the case of multiple occurrences of the same Lyndon factor in the pattern to improve the bound to $\mathcal{O}(|P|p')$ rank operations, where p' is the number of *different* Lyndon factors of P. For that we show two lemmas:

Figure 16: Setting of the proof of Lemma 4.10 that seems to derive a false occurrence. A necessary condition to derive a false occurrence from $\mathrm{lfs}_P(w)$ and T_x is that T_x is the last Lyndon factor having $\mathrm{lfs}_P(w)$ as a prefix (left). Since T_x must be border-free, $T_x = P_w$ holds (right).

Lemma 4.10. If $P_{w-1} = P_w = P_{w+1}$, then a *false* occurrence derived from $\mathrm{lfs}_P(w)$ disappears after matching $|P_w|$ characters.

Proof. Assume that there is a false occurrence derived from $\mathrm{lfs}_P(w)$ and T_x. Then (a) an occurrence of $\mathrm{lfs}_P(w)$ starts at position $\mathsf{b}_T(T_x)$ in T and (b) P_{w-1} is a suffix of T_x. See also Fig. 16. Since a Lyndon word is border-free, $T_x = P_w$. However, we derived a false occurrence from $\mathrm{lfs}_P(w)$ and T_x such that T_{x+1} cannot start with P_{w+1}. This is a contradiction, since we found an occurrence of $\mathrm{lfs}_P(w)$ starting at position $\mathsf{b}_T(T_x)$ in T, $T_x = P_w$, and therefore T_{x+1} must start with P_{w+1}. □

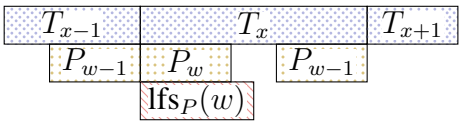

Figure 17: Setting of the proof of Lemma 4.11 that seems to derive a missed occurrence.

Lemma 4.11. Given an occurrence of P_w starts at position $\mathsf{b}_T(T_x)$ in T, a *missed* occurrence derived from $\mathrm{lfs}_P(w)$ and T_x disappears after matching $|P_w|$ characters if $|\mathrm{lfs}_P(w)| \leq |T_x|$ and $P_{w-1} = P_w$.

Proof. Suppose that there is a missed occurrence derived from $\mathrm{lfs}_P(w)$ and T_x. We have the following setting, which is sketched in Fig. 17:

- T_x is the leftmost Lyndon factor of the composed Lyndon factorization of T that starts with $\mathrm{lfs}_P(w)$, and
- P_{w-1} is a suffix of T_{x-1}.

$$\begin{aligned} \text{Then } P_w \preceq \mathrm{lfs}_P(w) &\preceq T_x && (P_w \text{ is a prefix of } \mathrm{lfs}_P(w) \text{ and } \mathrm{lfs}_P(w) \text{ is a prefix of } T_x) \\ &\prec T_{x-1} && (\text{Definition of the composed Lyndon factorization}) \\ &\preceq P_{w-1}, && (T_{x-1} \text{ is a Lyndon word and } P_{w-1} \text{ one of its suffixes}) \end{aligned}$$

contradicting the assumption $P_{w-1} = P_w$. □

A conclusion is that all longest consecutive appearances of the same Lyndon factors $P_w = \ldots = P_{w+j}$ for integers $1 \leq w \leq p$ and $j \geq 0$ can cause at most one newly missed and one false occurrence in total (which we need to keep track of). In other words, we know that we only have to care about p' freshly missed and false occurrences. Thus, we can improve the number of rank operations to $\mathcal{O}(|P|p')$.

4.4 Longest Pre-Lyndon Word

To obtain $\mathcal{O}(|P| \lg |P|)$ rank operations, we need the notion of the *longest pre-Lyndon suffix* λ_P, which is the smallest integer such that $\mathrm{lfs}_P(w+1)$ is a prefix of P_w for every $\lambda_P \leq w \leq p$. In our running example (cf. Fig. 11), $\lambda_T = 4$ since F_5 is a prefix of F_4, but F_4 is not a prefix of F_3.

Lemma 4.12. The string $\mathrm{lfs}_P(w)$ is a pre-Lyndon word for every $\lambda_P \leq w \leq p$.

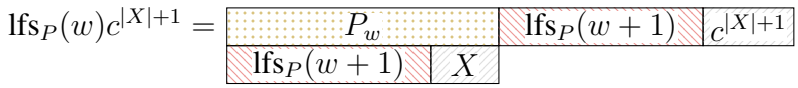

Figure 18: Setting of the proofs of Lemmas 4.12 and 4.16, where $\mathrm{lfs}_P(w+1)c^{|X|+1}$ is a Lyndon word because $\mathrm{lfs}_P(w+1)$ is a prefix of P_w and $c^{|X|+1} \succ X$.

Proof. Since $\mathrm{lfs}_P(w+1)$ is a prefix of P_w, there is a suffix X of P_w with $P_w = \mathrm{lfs}_P(w+1)X$ (cf. Fig. 18). Given a $c \in \Sigma$ with $c^{|X|+1} \succ X$, $\mathrm{lfs}_P(w)c^{|X|+1} = \mathrm{lfs}_P(w+1)X\mathrm{lfs}_P(w+1)c^{|X|+1}$ is a Lyndon word. □

We borrow the following facts from literature:

Lemma 4.13 ([44, Lemma 11])**.** P_w is *not* a proper prefix of $\mathrm{lfs}_P(w+1)$ for every integer w with $1 \le w \le \lambda_P - 1$.

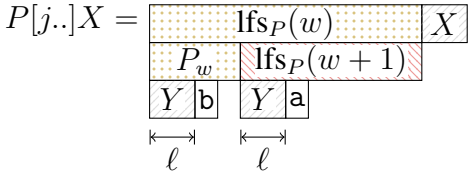

Figure 19: Setting of the proofs of Cor. 4.14 and Lemma 4.17, where a and b are characters with a $\prec$ b, and $Y = P_w[1..\ell] = \mathrm{lfs}_P(w+1)[1..\ell]$. There is no such string X that $P[j..]X$ is a Lyndon word.

Corollary 4.14. $\mathrm{lfs}_P(\lambda_P)$ is the longest pre-Lyndon suffix of P.

Proof. Assume that there is a longer pre-Lyndon suffix $P[j..]$. This suffix has to start with a Lyndon factor P_w for an integer w with $1 \le w \le p$, otherwise $\mathrm{lfs}_P(w) \prec P[j..]$ with w such that $\mathsf{b}_T(P_w) < j \le \mathsf{e}_T(P_w)$ (since every proper suffix of P_w is lexicographically larger than P_w), and therefore $\mathrm{lfs}_P(w)$ would be a longer pre-Lyndon suffix.

According to Lemma 4.13, there is an $\ell := \mathsf{lcp}(P_w, \mathrm{lfs}_P(w+1))$ with $\ell < \min(|P_w|, |\mathrm{lfs}_P(w+1)|)$. Then $P_w[\ell+1] > \mathrm{lfs}_P(w+1)[\ell+1]$ and therefore $\mathsf{conj}_{|P_w|}(\mathrm{lfs}_P(w)X) = \mathrm{lfs}_P(w+1)XP_w \prec P_w\mathrm{lfs}_P(w+1)X = \mathrm{lfs}_P(w)X$, regardless of the choice of the string X (cf. Fig. 19). □

Lemma 4.15 ([44, Lemma 12])**.** $p' - \lambda_P = \mathcal{O}(\lg |P|)$ where p' is the number of *distinct* Lyndon factors of P.

The next lemmas show the usefulness of λ_P:

Lemma 4.16. Given an integer w with $1 \le w \le \lambda_P - 1$, $\mathrm{lfs}_P(w+1)$ is not a prefix of P_w.

Proof. Assume to the contrary that $\mathrm{lfs}_P(w+1)$ is a prefix of P_w, and $P_w = \mathrm{lfs}_P(w+1)X$ for a string $X \in \Sigma^+$. Given a character $c \in \Sigma$ with $c^{|X|+1} \succ X$, $\mathrm{lfs}_P(w)c^{|X|+1} = \mathrm{lfs}_P(w+1)X\mathrm{lfs}_P(w+1)c^{|X|+1}$ is a Lyndon word, contradicting the fact that $\mathrm{lfs}_P(\lambda_P)$ is the longest pre-Lyndon word of P (cf. Cor. 4.14 and Fig. 18). □

Lemma 4.17. Given an occurrence of $\mathrm{lfs}_P(w)$ with $w < \lambda_P$ that starts at position $\mathsf{b}_T(T_x)$ in T for an integer x with $1 \le x \le t$, we have $T_x = P_w$.

Proof. Since $\mathrm{lfs}_P(\lambda_P)$ is the longest pre-Lyndon suffix of P (see Cor. 4.14), $\mathrm{lfs}_P(w)$ is not a pre-Lyndon suffix of P. According to Lemma 4.16, $\mathrm{lfs}_P(w+1)$ is not a prefix of P_w. Let $\ell := \mathsf{lcp}(P_w, \mathrm{lfs}_P(w+1)) < \min(|P_w|, |\mathrm{lfs}_P(w+1)|)$. Then $P_w[\ell+1] > \mathrm{lfs}_P(w+1)[\ell+1]$ according to the Lyndon factorization of P, and therefore P_w is the longest Lyndon word of T having an occurrence that starts at position $\mathsf{b}_T(T_x)$ in T, i.e., $T_x = P_w$. That is because a longer Lyndon factor T_x would contain $P_w\mathrm{lfs}_P(w+1)[1..\ell+1]$, which is lexicographically larger than $\mathsf{conj}_{|P_w|}(P_w\mathrm{lfs}_P(w+1)[1..\ell+1]) = \mathrm{lfs}_P(w+1)[1..\ell+1]P_w$ (cf. Fig. 19). □

The above lemmas allow us to derive the following consequence:

Corollary 4.18. There is no freshly missed occurrence derived from $\mathrm{lfs}_P(w)$ for every integer w with $w < \lambda_P$. If one of those $\mathrm{lfs}_P(w)$ derives a false occurrence, then this false occurrence disappears until the range $\mathsf{range}(P)$ is matched.

Proof. Suppose there is an occurrence of $\mathrm{lfs}_P(w)$ starting at position $\mathsf{b}_T(T_x)$ in T, for a composed Lyndon factor T_x with $1 \leq x \leq t$. According to Lemma 4.17, $P_w = T_x$. Since $w < \lambda_P \leq p$, we have that $|T_x| = |P_w| < |\mathrm{lfs}_P(w)|$. Then with Lemma 4.9 we obtain that $\mathrm{lfs}_P(w)$ and T_x cannot derive a freshly missed occurrence. By Lemma 4.13, P_{w-1} is not a proper prefix of $\mathrm{lfs}_w(P)$, such that after matching $\mathsf{lcp}(P_{w-1}, \mathrm{lfs}_w(P))$ characters, a possibly derived false occurrence will disappear, and thus does not need to be tracked. □

With Lemma 4.15 we obtain:

Theorem 4.19. Given a text T and a pattern P, we can compute all occurrences of P in T with the FM-index built on the bijective BWT of the composed Lyndon factors of T with $\mathcal{O}(|P| \lg |P|)$ rank operations.

Finally, we explain how to detect whether an occurrence of a suffix of the pattern starts at the beginning of a Lyndon factor of the text. For that, after each backward search step, we use the bit vector B_L introduced in Sect. 4.2 marking now the entries corresponding to the *composed* Lyndon factors in BBWT. If $\mathsf{range}(P[i..]) = [b..e]$ and $B_\mathrm{L}.\mathsf{rank}_1(e) - B_\mathrm{L}.\mathsf{rank}_1(b-1)$ is positive, then there is an occurrence of $P[i..]$ starting at position $\mathsf{b}_T(T_x)$ in T (after applying a composed jump) for every x with $t - B_\mathrm{L}.\mathsf{rank}_1(e) + 1 \leq x \leq t - B_\mathrm{L}.\mathsf{rank}_1(b-1)$. In this case, $P[i..] = \mathrm{lfs}_P(w)$ for an integer w with $1 \leq w \leq p$ due to Lemma 4.3.

The bit vector B_L can be stored in space linear to the number of runs of the BBWT. That is because [18, Corollary 2] have shown that the number of distinct primitive input strings to the eBWT is at most the number of runs in the eBWT, which directly translates to the fact that the number of distinct Lyndon factors is at most the number of runs in the BBWT. Hence, our bit vector B_L of length n contains at most r bits, and therefore a compressed representation [67, 69] using $H_0(B_\mathrm{L})n$ bits can be represented in $\mathcal{O}(r \lg n)$ bits if $r = o(n)$ since then $H_0(B_\mathrm{L}) = r/n \lg(n/r) + (n-r)/n \cdot \lg(n/(n-r)) = r/n \cdot (\lg(n/r) + \mathcal{O}(1))$.

Similarly, we can store the multiplicities $\tau_1, \ldots, \tau_{t'}$ of the distinct Lyndon factors in an integer array of length at most r.

4.5 Time and Space Complexities for Indexing

Having the backward search technique of Thm. 4.19, we can augment the bijective BWT with rank/select support data structures to gain the ability for pattern matching.

Theorem 4.20. Given a text T of length n whose characters are drawn from an alphabet of size σ, we can build a text index on the bijective BWT of T in $\mathcal{O}(n)$ time. The index uses $r_{\mathsf{BBWT}}(H_0(\mathsf{BBWT}_R) + 1 + H_0(B_\mathrm{L})) + o(r_{\mathsf{BBWT}}(H_0(\mathsf{BBWT}_R) + 1)) + \mathcal{O}(\sigma \lg n)$ bits, where BBWT_R is the bijective BWT of $R = T_1 \cdots T_t$ and $r_{\mathsf{BBWT}} := |\mathsf{BBWT}_R|$. The text index can count all occurrences of a pattern P in $\mathcal{O}(|P| \lg |P| \lg \sigma)$ time. For that, the index returns a range and a list of positions in SA corresponding to starting positions of suffixes having P as a prefix.

Proof. After building the BBWT in linear time with Thm. 3.6, we index the BBWT with a data structure that supports rank and select queries in $\mathcal{O}(\lg \sigma)$ time necessary for the backward search. For that, we use the Huffman-shaped wavelet tree of Grossi et al. [41], which can be constructed in $\mathcal{O}(r_{\mathsf{BBWT}})$ time. Further, we compute the bit vector B_L and compress it to $r_{\mathsf{BBWT}} H_0(B_\mathrm{L}) + o(r_{\mathsf{BBWT}})$ bits of space with the RRR bit vector representation [69], which supports rank and select operations in constant time. Finally, the array C is stored in a plain form using $\sigma \lg n$ bits. □

If we are allowed to spend more time for the construction, then there exist different data structures that can use less space and answer rank/select queries quicker [9].

To actually locate the matched positions in the text, we can use the r-index of Gagie et al. [32] who apply run-length compression on the traditional BWT and store a suffix array entry for each run in the BWT, thus achieving $\mathcal{O}(r \lg n)$ bits additional cost for this suffix array sampling, where r is the number of runs in the BWT. It is straight-forward to adapt this technique for the bijective BWT: For that, we keep B_L, but run-length compress BBWT. The time bounds remain the same if $\sigma = \mathcal{O}(\lg^{\mathcal{O}(1)} n)$ [57].

Following the steps of Boucher et al. [18] who provide an adaptation of the r-index to the eBWT that uses space linear in the number of runs of the eBWT, we can so do similar for obtaining an indexing data structure whose space is linear to the number of runs of the BBWT. Given that r is the number of runs in the BBWT, the main ingredients for an r-index are

1. a data structure supporting backward search steps on the run-length compressed BBWT,

2. the samples of the circular suffix array at the run borders of the BBWT, and

3. a predecessor data structure on these samples for supporting the Φ array, defined by

$\Phi[i] = \mathsf{SA}_\circ[\mathsf{ISA}_\circ[i] - 1]$ for $\mathsf{ISA}_\circ[i] > 1$ and $\Phi[i] = \mathsf{SA}_\circ[n]$ otherwise, where $\mathsf{ISA}_\circ$ denotes the inverse of $\mathsf{SA}_\circ$.

By using Thm. 4.20 for 1, and the predecessor data structure of Belazzougui and Navarro [9] for representing 1 as well as 3, we obtain

Theorem 4.21. Given a text T of length n whose characters are drawn from an alphabet of size σ, we can build a text index on BBWT_R using $\mathcal{O}(r)$ words, where r is the number of character of the BBWT of $R = T_1 \cdots T_t$, denoted by BBWT_R. Given a pattern P, the text index can count all occurrences of P in $\mathcal{O}(|P| \lg |P| \lg \log_w(\sigma + n/r))$ time, and locate all occ_P occurrences of P with additional $\mathcal{O}(\mathsf{occ}_P \lg \log_w n/r)$ time.

It seems possible to improve the time bounds by a novel data structure of Nishimoto and Tabei [63] to $\mathcal{O}(|P| \lg |P| \lg \log_w \sigma)$ for counting and $\mathcal{O}(|P| \lg |P| \lg \log_w \sigma + \mathsf{occ}_P)$ for locating.

4.6 Inversion

Finally, we show how to convert the BBWT back to the original text having the BBWT represented by a wavelet tree [41]. For that we follow the steps of Mantaci et al. [59][Proposition 15] for the eBWT. Having just the BBWT, Köppl et al. [49] presented an in-place conversion algorithm taking $\mathcal{O}(n^2)$ time. We can perform the inversion faster in time linear to the output length, multiplied by the query time of the wavelet tree. The idea is to process the positions marked by B_L from left to right. For each such marked position i, we apply recursively the backward search on this position and note down the visited characters in reverse order, until revisiting $\mathsf{BBWT}[i]$. Subsequently, we move to the next position marked by B_L. By doing so, we retrieve T_t for the first marked position in B_L, then T_{t-1} for the second, and so on. Having the multiplicities $\tau_1, \ldots, \tau_t$ at hand, we can output T in reverse order $T[n] \cdot T[n-1] \cdots T[1]$ in a streaming fashion.

If we do not have B_L, we create a bit vector B_V for marking already visited characters in the BBWT, and start at an arbitrary position $\mathsf{BBWT}[i]$ with the backward search as above. This time we stop when revisiting a position already marked in B_V. We subsequently continue with any position that has not yet been marked by B_V. By doing so, we obtain t primitive strings. However, there is a one-to-one correlation with the distinct Lyndon factors of T in that the Lyndon conjugate of each such primitive string is one of the distinct Lyndon factors. What is left to do is to find these Lyndon conjugates, sort them in descending order, multiply them by their multiplicities, and concatenate them to obtain T. To this end, in linear-time with constant working space, we can find the Lyndon conjugate of each returned string [72]. Finally, we use a string sorting algorithm that sorts these obtain Lyndon conjugates descendingly in lexicographic order, in time linear to the number of characters, which is n.

5 Experimental results

Moving from theory to practice, we here study the construction in Sect. 3 from a practical point of view. We omit results for the index (Sect. 4) since, as we will later observe in the evaluation (cf. Table 6), nearly all datasets we evaluated have very few Lyndon factors. That means that the number of false and missed occurrences, if there are any, is quite low for random patterns. In particular, the pattern has to be of a peculiar shape to obtain a value $p' - \lambda_P$ within $\omega(1)$ (cf. Lemma 4.15). Additionally, the difference on the number of runs between the BWT and the BBWT is marginal, at least for the datasets we evaluated (cf. Table 6), except for Fib41 and rs.13, where the number of runs of the BBWT is considerably larger. We thus expect that the final sizes of indexing data structures leveraging character runs in the BWT keep roughly the same when switching to the BBWT. Therefore, we postpone any practical study of the proposed index data structure.

5.1 Implementation

We have implemented our BBWT construction algorithm in C++. The implementation is publicly available at GitHub[9]. We parameterized (a) the input and the output data type as well as (b) the type used to represent numbers with C++ templates. The former (a) allows us to process data based not only on ASCII characters but also utf-32 and other alphabet formats with fixed bit widths. The latter (b) has a major impact on the memory consumption.

Our BBWT construction relies on the circular suffix array computation. Therefore, we need to be able to correctly represent all indices of $\mathsf{SA}_\circ$ and store them all in memory. Depending on the input data size one can decide whether to use 16, 32 or 64-bit data types to address input lengths up to 2^{16}, 2^{32}, and 2^{64} characters, respectively. Thanks to that, the amount of the memory consumed by the representation of $\mathsf{SA}_\circ$ can be significantly reduced if the maximal considered input length is known at compile time.

5.2 Evaluation

We compared the running time and memory consumption with the three alternative BBWT implementations, namely OpenBWT 2.0.0 by Yuta Mori[10] lFGSACA by Jannik Olbrich[11] and mk_bwts by Neal Burns[12]. The results are presented in Table 4, where we named our solution BBWT.

For the experiments we used a machine with two Intel(R) Xeon(R) Platinum 8260 CPU 2.40GHz processors and 1TB RAM running Debian Linux 12.0 bookworm (kernel 5.18.5-1). All programs were compiled using `gcc` compiler version 11.3.0 with the flag `-Ofast` and the flags originally used by the authors.

We ran all implementations on the datasets of the Pizza&Chili Corpus[13]. Each program ran five times for each dataset; we here report the average running time among those. The running time and the total memory usage have been measured using the GNU Time tool[14].

Since some implementations are based on 32-bit signed integer types, we limited the input text length up to 2^{31} characters, rather than to engage in any deeper source code modifications to avoid the possible degrading of the original performance.

From Table 4, we can observe that our implementation is not as performant as existing software with respect the running time. This is because we consider our code as a reference implementation for BBWT construction with focus on clarity and readability of the source code. In that sense, we believe that some effort put into algorithm engineering will improve both the algorithm running time and its memory consumption. Comparing with the other solutions, the best memory bounds can be achieved with OpenBWT. Among the selected datasets for evaluation, we observed that lFGSACA is the fastest solution for most of the datasets; on the downside, its memory consumption is the largest. While mk_bwts is the second fastest solution in most cases, it is the slowest on the 41-st Fibonacci word Fib41. This algorithm leverages the fact that entries in the suffix array and the circular suffix array are mostly the same or have small distances. Since it moves mismatching entries in a bubble-sort fashion, quadratic running time in the worst-case seems possible. However, it is an open question whether we can find such an input sequence, on which this algorithm exhibits a quadratic running time.

To evaluate the performance of our implementation in more detail we measured the running time (using C++ `std::chrono` library) separately for three phases of the BBWT construction algorithm:

- Phase_1 – Lyndon factorization of the input data,
- Phase_2 – computation of $\mathsf{SA}_\circ$,
- Phase_3 – retrieval of BBWT from $\mathsf{SA}_\circ$.

Moreover, we counted the number of recursive calls for the $\mathsf{SA}_\circ$ computation and the maximal number of distinct characters needed to encode LMS-substrings for each of those recursive calls. The results are presented in Table 5. Notable is that the Lyndon factorization thanks to Duval's algorithm takes only a

[9] https://github.com/mmpiatkowski/bbwt
[10] https://encode.su/attachment.php?attachmentid=1405
[11] https://gitlab.com/qwerzuiop/lfgsaca
[12] https://github.com/NealB/Bijective-BWT
[13] http://pizzachili.dcc.uchile.cl/
[14] https://www.gnu.org/software/time/

Table 4: Time and memory consumption for different BBWT construction implementations. n is the text length, T the running time (seconds), M the amount of memory consumed (kilobytes). A hyphen in the case of lFGSACA indicates that the computation failed for this particular file.

Pizza & Chili Corpus

		BBWT		lFGSACA		OpenBWT		mk_bwts	
file	n	T	M	T	M	T	M	T	M
dblp.xml	296 135 874	54.91	2 256 040	26.38	5 496 876	49.65	1 742 196	30.59	2 604 052
dna	403 927 746	90.52	3 103 732	38.95	7 905 588	89.17	2 373 844	61.21	3 551 440
english.1G	1 073 741 824	280.95	8 490 844	110.00	19 924 996	305.59	6 300 692	163.32	9 438 488
pitches	55 832 855	6.88	451 096	4.90	1 038 028	6.21	334 248	4.22	492 004
proteins	1 184 051 855	328.38	9 440 584	125.48	21 972 132	356.44	6 953 004	194.87	10 408 088
sources	210 866 607	36.82	1 627 748	20.21	3 915 088	33.23	1 244 716	20.01	1 854 776

Pizza & Chili Repetitive Corpus

		BBWT		lFGSACA		OpenBWT		mk_bwts	
file	n	T	M	T	M	T	M	T	M
Escherichia_Coli	112 689 515	19.83	829 012	10.45	2 093 388	18.11	665 336	13.37	991 748
cere	461 286 644	92.73	3 255 312	42.37	8 997 800	92.75	2 713 972	60.00	4 055 812
coreutils	205 281 778	35.52	1 503 224	18.76	3 811 436	30.94	1 205 956	20.87	1 805 568
dblp.xml.00001.1	104 857 600	15.80	768 344	11.66	1 948 044	14.24	619 344	11.26	922 780
dblp.xml.00001.2	104 857 600	15.94	768 352	11.62	1 948 040	15.61	627 564	10.93	922 892
dblp.xml.0001.1	104 857 600	15.73	768 960	11.69	1 948 040	14.48	621 392	11.04	923 004
dblp.xml.0001.2	104 857 600	15.98	769 144	11.66	1 948 040	15.30	623 424	10.95	922 860
dna.001.1	104 857 600	17.27	752 696	12.46	2 052 784	16.12	621 464	12.48	923 040
einstein.de.txt	92 758 441	15.32	680 392	8.15	1 723 316	14.44	550 744	8.44	816 796
einstein.en.txt	467 626 544	102.51	3 427 596	43.16	8 678 512	106.08	2 755 432	49.44	4 111 292
english.001.2	104 857 600	18.43	782 608	12.87	1 948 044	17.18	621 388	11.37	922 900
fib41	267 914 296	39.12	2 149 612	18.42	5 373 296	39.51	1 578 968	102.69	2 356 144
influenza	154 808 555	27.46	1 119 068	13.33	2 874 912	23.88	914 100	16.89	1 361 976
kernel	257 961 616	47.59	1 867 988	24.15	4 788 584	43.38	1 520 708	28.70	2 268 832
para	429 265 758	88.79	3 062 384	40.18	8 386 328	86.85	2 524 388	103.75	3 774 380
proteins.001.1	104 857 600	19.92	785 232	12.77	1 947 652	17.39	625 468	12.45	922 908
rs.13	216 747 218	30.94	1 697 988	15.57	4 447 540	30.08	1 278 944	54.51	1 906 484
sources.001.2	104 857 600	16.83	771 752	12.26	1 947 656	14.96	623 388	9.84	922 940
tm29	268 435 456	41.74	2 045 692	26.70	5 423 596	41.81	1 588 216	61.88	2 360 840
world_leaders	46 968 181	4.56	305 764	3.63	880 880	4.10	278 420	2.40	414 112

Table 5: Detailed evaluation of the BBWT construction algorithm. n is the text length, σ is the input alphabet size, $Phase_1$, $Phase_2$ and $Phase_3$ are the running times (seconds) of the subsequent phases of $\mathsf{SA}_\circ$ algorithm construction, Rec is the number of recursion levels in $\mathsf{SA}_\circ$ construction, $\max_\sigma$ is the maximal size of the alphabet used for encoding distinct LMS substrings in $\mathsf{SA}_\circ$ construction.

Pizza & Chili Corpus							
file	n	σ	$Phase_1$	$Phase_2$	$Phase_3$	Rec	$\max_\sigma$
DBLP.XML	296 135 874	97	0.31	49.61	5.10	7	3 763 047
DNA	403 927 746	16	0.35	81.54	7.54	14	13 120 265
ENGLISH	2 210 395 553	239	0.92	258.45	22.37	13	29 617 376
PITCHES	55 832 855	133	0.50	6.51	0.31	9	2 736 240
PROTEINS	1 184 051 855	27	1.46	305.36	24.17	12	46 516 571
SOURCES	210 866 607	230	0.21	34.94	2.88	11	5 201 041

Pizza & Chili Repetitive Corpus							
file	n	σ	$Phase_1$	$Phase_2$	$Phase_3$	Rec	$\max_\sigma$
ESCHERICHIA_COLI	112 689 515	15	0.30	17.63	1.64	12	1 162 864
CERE	461 286 644	5	1.37	82.14	8.41	12	909 922
COREUTILS	205 281 778	236	0.18	31.89	2.95	13	590 967
DBLP.XML.00001.1	104 857 600	89	0.11	14.20	1.35	12	23 903
DBLP.XML.00001.2	104 857 600	89	0.11	14.29	1.36	12	23 903
DBLP.XML.0001.1	104 857 600	89	0.11	14.15	1.37	11	33 051
DBLP.XML.0001.2	104 857 600	89	0.11	14.27	1.34	11	33 195
DNA.001.1	104 857 600	5	0.28	15.64	1.35	9	176 533
EINSTEIN.DE.TXT	92 758 441	117	0.83	13.36	1.24	12	14 207
EINSTEIN.EN.TXT	467 626 544	139	0.41	92.61	8.90	12	37 814
ENGLISH.001.2	104 857 600	106	0.91	16.74	1.37	9	180 085
FIB41	267 914 296	2	0.31	33.82	4.67	20	2
INFLUENZA	154 808 555	15	0.42	24.94	2.39	9	308 990
KERNEL	257 961 616	160	0.27	42.94	3.96	13	387 045
PARA	429 265 758	5	1.29	79.22	7.98	11	1 238 537
PROTEINS.001.1	104 857 600	21	0.14	17.12	1.36	9	186 458
RS.13	216 747 218	2	0.26	26.15	3.59	14	5
SOURCES.001.2	104 857 600	98	0.11	15.24	1.26	9	167 767
TM29	268 435 456	2	0.37	36.38	4.63	17	5
WORLD_LEADERS	46 968 181	89	0.42	4.52	0.38	10	71 259

tiny fraction of the total computation time. From that we expect that our algorithm, adapted to the eBWT computation, also practically competes with other algorithms that directly compute the eBWT from a multiset of primitive strings. That is because our precomputation step of Phase_1 has marginally impact on the running time, and the modification to find the Lyndon conjugate of each input string (instead of computing the Lyndon factorization) is lightweight.

6 Conclusion

In the first part of this article, we proposed an algorithm computing the bijective Burrows–Wheeler transform (BBWT) in linear time. Consequently, we can also compute the extended Burrows–Wheeler transform (eBWT) within the same time bounds by a linear-time reduction of the problem to compute the eBWT to computing the BBWT.

Our trick was to first reduce our input text T to a text R by removing all duplicate Lyndon factors. Second, we slightly modified the suffix array – induced sorting (SAIS) algorithm to compute the $\prec_\omega$-order

of the conjugates of all Lyndon factors of R instead of the $\prec_{\text{lex}}$-order of all suffixes of R. For that, we introduced the notion of inf-suffixes and inf-substrings, and adapted the typing system of L, S, and S* types from SAIS. By some properties of the Lyndon factors, we could show that there are only some border cases, where a text position receives a different type in our modification. Thanks to that, we could directly translate the induced-sorting techniques of SAIS, and obtain the correctness of our result.

In the second part, we studied the problem of indexing the BBWT. While this is easier for the extended BWT, we additionally had to take care of omitting circular occurrences and tracking occurrences that cross the boundary of Lyndon factors. Thanks to the properties of the Lyndon factorization, we could bound the number of these occurrences by $\mathcal{O}(\lg |P|)$. Our index is therefore by a multiplicative factor of $\mathcal{O}(\lg |P|)$ slower than the FM-index on the traditional BWT.

7 Open Problems

The BBWT is bijective in the sense that it transforms a string of Σ^n into another string of Σ^n while preserving distinctness. Consequently, given a string of length n, there is an integer $k \geq 1$ with $\mathsf{BBWT}^k(T) = \mathsf{BBWT}^{k-1}(\mathsf{BBWT}(T)) = T$. With our presented algorithm we can compute the smallest such number k in $\mathcal{O}(nk)$ time. However, we wonder whether we can compute this number faster, possible by scanning only the text in $\mathcal{O}(n)$ time independent of k.

7.1 BBWT Based on Different Factorizations

We also wonder whether we can define the BBWT for the generalized Lyndon factorization [26]. Contrary to the Lyndon factorization, the generalized Lyndon factorization uses a different order, called the *generalized lexicographic order* $\prec_{\text{gen}}$. In this order, two strings $S, T \in \Sigma^*$ are compared character-wise like in the lexicographic order. However, the generalized lexicographic order $\prec_{\text{gen}}$ can use different orders $<_1, <_2, \ldots$ for each text position, i.e., $S \prec_{\text{gen}} T$ if and only if S is a proper prefix of T or there is an integer ℓ with $1 \leq \ell \leq \min(|S|, |T|)$ such that $S[1..\ell-1] = T[1..\ell-1]$ and $S[\ell] <_\ell T[\ell]$.

Other factorization that maybe capable for defining a bijective BWT similar structure are the Galois factorization [26, 70], inverse Lyndon factorization [12], and the Nyldon factorization [22]. The Galois and Nyldon factorization define a unique bijection from a string to a multiset of primitive, cyclic words, and therefore can be used to define a BBWT variant with the observation made in our introduction. Unfortunately, inverse Lyndon words are not necessarily primitive, and hence the inverse Lyndon factorization may produce non-primitive words, which need to be considered with additional caution.

It would be interesting to prove or disprove whether the computed set of factors of any former mentioned factorization can form a type of BBWT that can be inverted, used for pattern matching, or can be used for text compression.

7.2 Compressibility

Table 6 shows the run lengths of the BBWT in juxtaposition with the BWT for the files from Calgary Corpus[15], Canterbury Corpus[16], Pizza&Chili Corpus[17], and Silesia Corpus[18]. Here, we counted the number of runs in BWT without the terminal symbol (dollar sign). We observe that for these datasets, the number of runs of both transforms are roughly equal, and we expect that to hold for other kinds of datasets as long as the Lyndon factors are few in numbers.

Although the number of runs in the BBWT or BWT are always smaller than the input text length for the files in Table 6, Mantaci et al. [61, Theorem 8] could construct, for a given rational number $\gamma \in (0, 2]$, a binary Lyndon word for which the ratio between the number of character runs in the BWT and the number of character runs in the original text is γ, and have shown that this ratio is tight (i.e., the number of runs can at most double after a BWT application). This result directly translates to the BBWT, since the BBWT and the BWT are identical when the input text is a Lyndon word. Related is question about the ratio between the number of runs in the BBWT and the BBWT applied on the reverse input text.

[15] http://www.data-compression.info/Corpora/CalgaryCorpus/
[16] http://www.data-compression.info/Corpora/CanterburyCorpus/
[17] http://pizzachili.dcc.uchile.cl/
[18] http://www.data-compression.info/Corpora/SilesiaCorpus/index.html

Giuliani et al. [39] modified the Fibonacci words with an additionally appended character to show that there exists a family of binary strings for which this ratio for the BWT is $\Theta(\lg n)$. A similar approach leads to the same ratio for the BBWT [11].

Finally, a theoretical analysis between the runs of the classic BWT and the bijective BWT would be more than welcomed. From Table 6, we can empirically observe that the number of Lyndon words tend to be few in real-world datasets. Even when changing the alphabet order, the Lyndon words tend to be few in numbers [3] such that we are not confident to expect large changes between the number of runs in the BWT and BBWT on practical datasets.

Regarding compressibility with respect to the empirical entropy, Ferragina et al. [30] have shown that the BWT of an input string T can be compressed up to the k-th order empirical entropy of T for any $k > 0$. A similar result for the BBWT is still unknown.

Table 6: Number of runs of the BBWT compared to the BWT for various data sets. n is the text length, σ the alphabet size, f the number of Lyndon factors, t the number of distinct Lyndon factors. The last two columns r_{BBWT} and r_{BWT} show the number of character runs in the BBWT and BWT, respectively.

Calgary Corpus								
file	n	σ	f	t	r_{BBWT}	r_{BWT}	n/r_{BBWT}	n/r_{BWT}
BIB	111 261	81	6	6	36 971	36 964	3.009	3.010
BOOK1	768 771	82	12	12	386 264	386 263	1.990	1.990
BOOK2	610 856	96	27	27	239 378	239 367	2.552	2.552
GEO	102 400	256	20	8	65 781	65 778	1.557	1.557
NEWS	377 109	98	24	24	158 607	158 592	2.378	2.378
OBJ1	21 504	256	991	6	10 616	10 616	2.026	2.026
OBJ2	246 814	256	10	10	78 814	78 814	3.132	3.132
PAPER1	53 161	95	9	9	22 146	22 140	2.400	2.401
PAPER2	82 199	91	16	16	36 689	36 687	2.240	2.241
PAPER3	46 526	84	14	14	22 569	22 566	2.062	2.062
PAPER4	13 286	80	6	6	6904	6903	1.924	1.925
PAPER5	11 954	91	6	6	5938	5935	2.013	2.014
PAPER6	38 105	93	15	15	16 048	16 046	2.374	2.375
PIC	513 216	159	36 319	4	64 691	64 690	7.933	7.933
PROGC	39 611	92	12	12	15 709	15 707	2.522	2.522
PROGL	71 646	87	77	7	19 446	19 442	3.684	3.685
PROGP	49 379	89	12	12	12 825	12 823	3.850	3.851
TRANS	93 695	99	228	13	19 456	19 453	4.816	4.816

Canterbury Corpus								
file	n	σ	f	t	r_{BBWT}	r_{BWT}	n/r_{BBWT}	n/r_{BWT}
ALICE29.TXT	152 089	74	3	3	66 903	66 902	2.273	2.273
ASYOULIK.TXT	125 179	68	2	2	62 366	62 364	2.007	2.007
CP.HTML	24 603	86	8	8	9201	9198	2.674	2.675
FIELDS.C	11 150	90	13	13	3417	3409	3.263	3.271
GRAMMAR.LSP	3721	76	8	6	1340	1344	2.777	2.769
KENNEDY.XLS	1 029 744	256	9	9	234 842	234 838	4.385	4.385
LCET10.TXT	426 754	84	6	6	165 712	165 709	2.575	2.575
PLRABN12.TXT	481 861	81	6	6	243 558	243 557	1.978	1.978
PTT5	513 216	159	36 319	4	64 691	64 690	7.933	7.933
SUM	38 240	255	13	10	13 262	13 262	2.883	2.883
XARGS.1	4227	74	9	9	2009	2008	2.104	2.105

Silesia Corpus								
file	n	σ	f	t	r_{BBWT}	r_{BWT}	n/r_{BBWT}	n/r_{BWT}
DICKENS	10 192 446	100	17	17	4 374 629	4 374 598	2.330	2.330
MOZILLA	51 220 480	256	8322	16	19 498 071	19 498 064	2.627	2.627
MR	9 970 564	256	179	23	3 444 892	3 444 884	2.894	2.894
NCI	33 553 445	62	6	6	2 195 819	2 195 816	15.281	15.281
OOFFICE	6 152 192	256	15 201	8	3 215 176	3 215 179	1.913	1.913
OSDB	10 085 684	256	15	15	3 224 389	3 224 386	3.128	3.128
REYMONT	6 627 202	256	26	26	1 935 978	1 935 966	3.423	3.423
SAMBA	21 606 400	256	10 304	17	5 220 562	5 220 551	4.139	4.139
SAO	7 251 944	256	6	6	5 524 741	5 524 740	1.313	1.313
X-RAY	8 474 240	256	7	7	4 796 213	4 796 213	1.767	1.767
XML	5 345 280	104	5955	8	581 963	581 955	9.185	9.185

Pizza & Chili Repetitive Corpus								
file	n	σ	f	t	r_{BBWT}	r_{BWT}	n/r_{BBWT}	n/r_{BWT}
fib41	267 914 296	2	21	21	41	2	6 534 495.024	89 304 765.333
rs.13	216 747 218	2	27	27	123	75	1 762 172.504	2 889 962.907
tm29	268 435 456	2	41	41	81	81	3 314 017.975	3 314 017.975
dblp.xml.00001.1	104 857 600	89	7	7	172 500	172 487	607.870	607.916
dblp.xml.00001.2	104 857 600	89	23	23	175 626	175 616	597.051	597.085
dblp.xml.0001.1	104 857 600	89	9	9	240 550	240 533	435.908	435.939
dblp.xml.0001.2	104 857 600	89	21	21	270 213	270 203	388.055	388.070
dna.001.1	104 857 600	5	18	18	1 716 857	1 716 806	61.075	61.077
english.001.2	104 857 600	106	29	29	1 449 562	1 449 517	72.337	72.340
proteins.001.1	104 857 600	21	19	19	1 278 237	1 278 199	82.033	82.035
sources.001.2	104 857 600	98	50	50	1 213 519	1 213 426	86.408	86.414
Escherichia_Coli	112 689 515	15	13	13	15 044 536	15 044 485	7.490	7.490
cere	461 286 644	5	21	21	11 574 705	11 574 639	39.853	39.853
coreutils	205 281 778	236	17	17	4 684 513	4 684 458	43.821	43.822
einstein.de.txt	92 758 441	117	21	21	101 391	101 369	914.859	915.057
einstein.en.txt	467 626 544	139	59	59	290 279	290 237	1610.955	1611.189
influenza	154 808 555	15	10	10	3 022 821	3 022 820	51.213	51.213
kernel	257 961 616	160	32	32	2 791 456	2 791 366	92.411	92.414
para	429 265 758	5	1238	22	15 636 838	15 636 738	27.452	27.452
world_leaders	46 968 181	89	13	12	573 506	573 485	81.897	81.900

Pizza & Chili Corpus								
file	n	σ	f	t	r_{BBWT}	r_{BWT}	n/r_{BBWT}	n/r_{BWT}
dblp.xml	296 135 874	97	15	15	41 037 558	41 037 553	7.216	7.216
dna	403 927 746	16	18	18	243 492 872	243 492 866	1.659	1.659
english	2 210 395 553	239	18	18	658 301 004	658 301 008	3.358	3.358
pitches	55 832 855	133	39	39	23 430 040	23 429 976	2.383	2.383
proteins	1 184 051 855	27	30	30	441 858 493	441 858 468	2.680	2.680
sources	210 866 607	230	31	31	47 896 880	47 896 806	4.403	4.403

7.3 Alphabet Reordering

Recently, Gibney and Thankachan [37] showed that finding an order of the alphabet such that the number of Lyndon factors of a given string is minimized or maximized is NP-complete. This is an important but negative result for finding an advantage of the BBWT over the BWT, since the hope is to find a way to increase the number of Lyndon factors and therefore the chances of having multiple equal factors that are contracted to a single composed factor in our proposed BBWT index. However, it is left open, whether we can find an efficient algorithm that approximates the alphabet order maximizing the number of Lyndon factors.

Interestingly, the context adaptive transformation minimizing the number of runs in the BWT based on local orderings-based transformations, can be found in time linear in the input text length [36, Theorem 23].

Another direction would be to find a string family for which we $\mathsf{SA}_\circ$ and SA differ, for instance, with a relatively high Hamming distance.

7.4 $\mathsf{SA}_\circ$ in constant space

Finally, we pose as an open problem whether we can construct $\mathsf{SA}_\circ$ in-place. The problem boils down to finding an in-place Lyndon factorization algorithm that can compute the Lyndon factors online from the text end to its beginning, because that is the same direction in which we can determine all `L` and `S`

inf-suffixes as well as the in linear time. A linear-time algorithm for the computation in this direction is presented in [4]; however this algorithm needs $\mathcal{O}(n)$ words of working space.

Acknowledgements This research was supported by JSPS KAKENHI with grant numbers `JP21K17701`, `JP23H04378`, and `JP20H04141`.

References

[1] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows–Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 2008.

[2] O. Y. Ahmed, M. Rossi, T. Gagie, C. Boucher, and B. Langmead. SPUMONI 2: improved classification using a pangenome index of minimizer digests. *Genome Biology*, 24(1):122, 2023. doi: 10.1186/s13059-023-02958-1.

[3] M. K. Albertini and F. A. Louza. Practical evaluation of Lyndon factors via alphabet reordering. *Mathematics*, 11(1), 2023. doi: 10.3390/math11010139.

[4] G. Badkobeh, M. Crochemore, J. Ellert, and C. Nicaud. Back-to-front online Lyndon forest construction. In *Proc. CPM*, volume 223 of *LIPIcs*, pages 13:1–13:23, 2022. doi: 10.4230/LIPIcs.CPM.2022.13.

[5] U. Baier. Linear-time suffix sorting - A new approach for suffix array construction. In *Proc. CPM*, volume 54 of *LIPIcs*, pages 23:1–23:12, 2016. doi: 10.4230/LIPIcs.CPM.2016.23.

[6] H. Bannai, J. Kärkkäinen, D. Köppl, and M. Piatkowski. Indexing the bijective BWT. In *Proc. CPM*, volume 128 of *LIPIcs*, pages 17:1–17:14, 2019. doi: 10.4230/LIPIcs.CPM.2019.17.

[7] H. Bannai, J. Kärkkäinen, D. Köppl, and M. Piątkowski. Constructing the bijective and the extended Burrows–Wheeler transform in linear time. In *Proc. CPM*, volume 191 of *LIPIcs*, pages 7:1–7:16, 2021. doi: 10.4230/LIPIcs.CPM.2021.7.

[8] M. J. Bauer, A. J. Cox, and G. Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483:134–148, 2013. doi: 10.1016/j.tcs.2012.02.002.

[9] D. Belazzougui and G. Navarro. Optimal lower and upper bounds for representing sequences. *ACM Trans. Algorithms*, 11(4):31:1–31:21, 2015. doi: 10.1145/2629339.

[10] N. Bertram, J. Ellert, and J. Fischer. Lyndon words accelerate suffix sorting. In *Proc. ESA*, volume 204 of *LIPIcs*, pages 15:1–15:13, 2021. doi: 10.4230/LIPIcs.ESA.2021.15.

[11] E. Biagi, D. Cenzato, Z. Liptak, and G. Romana. On the number of equal-letter runs of the bijective Burrows–Wheeler transform. In *Proc. ICTCS*, page to appear, 2023.

[12] P. Bonizzoni, C. De Felice, R. Zaccagnino, and R. Zizza. Inverse Lyndon words and inverse Lyndon factorizations of words. *Adv. Appl. Math.*, 101:281–319, 2018. doi: 10.1016/j.aam.2018.08.005.

[13] S. Bonomo, S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. Sorting conjugates and suffixes of words in a multiset. *Int. J. Found. Comput. Sci.*, 25(8):1161, 2014. doi: 10.1142/S0129054114400309.

[14] S. Böttcher, A. Bültmann, R. Hartel, and J. Schlüßler. Fast insertion and deletion in compressed texts. In *Proc. DCC*, page 393, 2012. doi: 10.1109/DCC.2012.50.

[15] S. Böttcher, A. Bültmann, R. Hartel, and J. Schlüßler. Implementing efficient updates in compressed big text databases. In *Proc. DEXA*, volume 8056 of *LNCS*, pages 189–202, 2013. doi: 10.1007/978-3-642-40173-2_17.

[16] C. Boucher, T. Gagie, A. Kuhnle, B. Langmead, G. Manzini, and T. Mun. Prefix-free parsing for building big BWTs. *Algorithms Mol. Biol.*, 14(1):13:1–13:15, 2019. doi: 10.1186/s13015-019-0148-5.

[17] C. Boucher, D. Cenzato, Z. Lipták, M. Rossi, and M. Sciortino. Computing the original eBWT faster, simpler, and with less memory. In *Proc. SPIRE*, volume 12944 of *LNCS*, pages 129–142, 2021. doi: 10.1007/978-3-030-86692-1_11.

[18] C. Boucher, D. Cenzato, Z. Lipták, M. Rossi, and M. Sciortino. r-indexing the eBWT. In *Proc. SPIRE*, volume 12944 of *LNCS*, pages 3–12, 2021. doi: 10.1007/978-3-030-86692-1_1.

[19] C. Boucher, T. Gagie, T. I, D. Köppl, B. Langmead, G. Manzini, G. Navarro, A. Pacheco, and M. Rossi. PHONI: Streamed matching statistics with multi-genome references. In *Proc. DCC*, pages 193–202, 2021. doi: 10.1109/DCC50243.2021.00027.

[20] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[21] D. Cenzato and Z. Lipták. A theoretical and experimental analysis of BWT variants for string collections. In *Proc. CPM*, volume 223 of *LIPIcs*, pages 25:1–25:18, 2022. doi: 10.4230/LIPIcs.CPM.2022.25.

[22] É. Charlier, M. Philibert, and M. Stipulanti. Nyldon words. *J. Comb. Theory, Ser. A*, 167:60–90, 2019. doi: 10.1016/j.jcta.2019.04.002.

[23] 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.

[24] D. Díaz-Domínguez and G. Navarro. A grammar compressor for collections of reads with applications to the construction of the BWT. In *Proc. DCC*, pages 83–92, 2021. doi: 10.1109/DCC50243.2021.00016.

[25] D. Díaz-Domínguez and G. Navarro. Efficient construction of the BWT for repetitive text using string compression. *Inf. Comput.*, 294:105088, 2023.

[26] F. Dolce, A. Restivo, and C. Reutenauer. On generalized Lyndon words. *Theor. Comput. Sci.*, 777: 232–242, 2019. doi: 10.1016/j.tcs.2018.12.015.

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

[28] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398, 2000. doi: 10.1109/SFCS.2000.892127.

[29] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi: 10.1145/1082036.1082039.

[30] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *J. ACM*, 52(4):688–713, 2005. doi: 10.1145/1082036.1082043.

[31] N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965.

[32] T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. SODA*, pages 1459–1477, 2018. doi: 10.1137/1.9781611975031.96.

[33] I. M. Gessel and C. Reutenauer. Counting permutations with given cycle structure and descent set. *J. Comb. Theory, Ser. A*, 64(2):189–215, 1993.

[34] I. M. Gessel, A. Restivo, and C. Reutenauer. A bijection between words and multisets of necklaces. *Eur. J. Comb.*, 33(7):1537–1546, 2012. doi: 10.1016/j.ejc.2012.03.016.

[35] R. Giancarlo, G. Manzini, A. Restivo, G. Rosone, and M. Sciortino. The alternating BWT: An algorithmic perspective. *Theor. Comput. Sci.*, 812:230–243, 2020. doi: 10.1016/j.tcs.2019.11.002.

[36] R. Giancarlo, G. Manzini, A. Restivo, G. Rosone, and M. Sciortino. A new class of string transformations for compressed text indexing. *ArXiv 2205.05643*, 2022.

[37] D. Gibney and S. V. Thankachan. Finding an optimal alphabet ordering for Lyndon factorization is hard. In *Proc. STACS*, volume 187 of *LIPIcs*, pages 35:1–35:15, 2021. doi: 10.4230/LIPIcs.STACS.2021.35.

[38] J. Y. Gil and D. A. Scott. A bijective string sorting transform. *ArXiv 1201.3077*, 2012.

[39] S. Giuliani, S. Inenaga, Z. Lipták, N. Prezza, M. Sciortino, and A. Toffanello. Novel results on the number of runs of the Burrows–Wheeler-transform. In *Proc. SOFSEM*, volume 12607 of *LNCS*, pages 249–262, 2021. doi: 10.1007/978-3-030-67731-2_18.

[40] K. Goto. Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. In *Proc. PSC*, pages 111–125, 2019.

[41] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.

[42] W. Hon, C. Lu, R. Shah, and S. V. Thankachan. Succinct indexes for circular patterns. In *Proc. ISAAC*, volume 7074 of *LNCS*, pages 673–682, 2011. doi: 10.1007/978-3-642-25591-5_69.

[43] W. Hon, T. Ku, C. Lu, R. Shah, and S. V. Thankachan. Efficient algorithm for circular Burrows–Wheeler transform. In *Proc. CPM*, volume 7354 of *LNCS*, pages 257–268, 2012. doi: 10.1007/978-3-642-31265-6_21.

[44] T. I, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. *Theor. Comput. Sci.*, 656:215–224, 2016. doi: 10.1016/j.tcs.2016.03.005.

[45] M. Ito, H. Inoue, and K. Taura. Fragmented BWT: an extended BWT for full-text indexing. In *Proc. SPIRE*, volume 9954 of *LNCS*, pages 97–109, 2016. doi: 10.1007/978-3-319-46049-9_10.

[46] G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554, 1989. doi: 10.1109/SFCS.1989.63533.

[47] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. doi: 10.1145/1217856.1217858.

[48] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005. doi: 10.1016/j.jda.2004.08.002.

[49] D. Köppl, D. Hashimoto, D. Hendrian, and A. Shinohara. In-place bijective Burrows–Wheeler transforms. In *Proc. CPM*, volume 161 of *LIPIcs*, pages 21:1–21:15, 2020. doi: 10.4230/LIPIcs.CPM.2020.21.

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

[51] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, 2012. doi: 10.1038/nmeth.1923.

[52] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinform.*, 26(5):589–595, 2010. doi: 10.1093/bioinformatics/btp698.

[53] R. Li, C. Yu, Y. Li, T. W. Lam, S. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinform.*, 25(15):1966–1967, 2009. doi: 10.1093/bioinformatics/btp336.

[54] Z. Li, J. Li, and H. Huo. Optimal in-place suffix sorting. In *Proc. SPIRE*, volume 11147 of *LNCS*, pages 268–284, 2018. doi: 10.1007/978-3-030-00479-8_22.

[55] F. A. Louza, S. Mantaci, G. Manzini, M. Sciortino, and G. P. Telles. Inducing the Lyndon array. In *Proc. SPIRE*, volume 11811 of *LNCS*, pages 138–151, 2019. doi: 10.1007/978-3-030-32686-9_10.

[56] R. C. Lyndon. On Burnside's problem. *Transactions of the American Mathematical Society*, 77(2):202–215, 1954.

[57] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.*, 12(1):40–66, 2005.

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

[59] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows–Wheeler transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007. doi: 10.1016/j.tcs.2007.07.014.

[60] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. Suffix array and Lyndon factorization of a text. *J. Discrete Algorithms*, 28:2–8, 2014. doi: 10.1016/j.jda.2014.06.001.

[61] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. Burrows–Wheeler transform and run-length enconding. In *Proc. WORDS*, volume 10432 of *LNCS*, pages 228–239, 2017. doi: 10.1007/978-3-319-66396-8_21.

[62] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):2:1–2:61, 2007. doi: 10.1145/1216370.1216372.

[63] T. Nishimoto and Y. Tabei. Optimal-time queries on BWT-runs compressed indexes. In *Proc. ICALP*, volume 198 of *LIPIcs*, pages 101:1–101:15, 2021. doi: 10.4230/LIPIcs.ICALP.2021.101.

[64] G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011. doi: 10.1109/TC.2010.188.

[65] T. Ohno, Y. Takabatake, T. I, and H. Sakamoto. A faster implementation of online run-length Burrows–Wheeler transform. In *Proc. IWOCA*, volume 10765 of *LNCS*, pages 409–419, 2017. doi: 10.1007/978-3-319-78825-8_33.

[66] J. Olbrich, E. Ohlebusch, and T. Büchler. On the optimisation of the GSACA suffix array construction algorithm. In *Proc. SPIRE*, volume 13617 of *LNCS*, pages 99–113, 2022. doi: 10.1007/978-3-031-20643-6_8.

[67] M. Patrascu. Succincter. In *Proc. FOCS*, pages 305–313, 2008. doi: 10.1109/FOCS.2008.83.

[68] A. Policriti and N. Prezza. Computing LZ77 in run-compressed space. In *Proc. DCC*, pages 23–32, 2016. doi: 10.1109/DCC.2016.30.

[69] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi: 10.1145/1290672.1290680.

[70] C. Reutenauer. Mots de Lyndon généralisés. *Séminaire Lotharingien de Combinatoire*, 54(B54h): 1–16, 2005.

[71] M. Rossi, M. Oliva, B. Langmead, T. Gagie, and C. Boucher. MONI: A pangenomic index for finding maximal exact matches. *J. Comput. Biol.*, 29(2):169–187, 2022. doi: 10.1089/cmb.2021.0290.

[72] Y. Shiloach. Fast canonization of circular strings. *J. Algorithms*, 2(2):107–121, 1981. doi: 10.1016/0196-6774(81)90013-4.