Constructing the bijective and the extended burrows-wheeler transform in linear time

Bannai, Hideo

Bannai , H , Kärkkäinen , J , Köppl , D & Piatkowski , M 2021 , Constructing the bijective and the extended burrows-wheeler transform in linear time . in P Gawrychowski & T Starikovskaya (eds) , 32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021 . , 7 , Leibniz International Proceedings in Informatics, LIPIcs , vol. 191 , Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing , Annual Symposium on Combinatorial Pattern Matching , Wroclaw , Poland , 05/07/2021 . https://doi.org/10.4230/LIPIcs.CPM.2021.7

Constructing the Bijective and the Extended Burrows–Wheeler Transform in Linear Time

Hideo Bannai ✉ 🆔
M&D Data Science Center, Tokyo Medical and Dental University, Tokyo, Japan

Juha Kärkkäinen ✉
Helsinki Institute of Information Technology (HIIT), Finland

Dominik Köppl ✉ 🏠 🆔
M&D Data Science Center, Tokyo Medical and Dental University, Tokyo, Japan

Marcin Piątkowski ✉ 🆔
Nicolaus Copernicus University, Toruń, Poland

Abstract

The Burrows–Wheeler transform (BWT) is a permutation whose applications are prevalent in data compression and text indexing. The *bijective BWT* (BBWT) is a bijective variant of it. Although it is known that the BWT can be constructed in linear time for integer alphabets by using a linear time suffix array construction algorithm, it was up to now only conjectured that the BBWT can also be constructed in linear time. We confirm this conjecture in the word RAM model by proposing a construction algorithm that is based on SAIS, improving the best known result of $\mathcal{O}(n \lg n / \lg \lg n)$ time to linear. Since we can reduce the problem of constructing the extended BWT to constructing the BBWT in linear time, we obtain a linear-time algorithm computing the extended BWT at the same time.

1 Introduction

The Burrows–Wheeler transform (BWT) [4] is a transformation permuting the characters of a given string $T\$$, where $\$$ is a character that is strictly smaller than all characters occurring it T. The i-th entry of the BWT of $T\$$ is the character preceding the i-th lexicographically smallest suffix of $T\$$, or $\$$ if this suffix is $T\$$ itself. Strictly speaking, the BWT is not a bijection since its output contains $\$$ at an arbitrary position while it requests the input T to have $\$$ as a delimiter at its end in order to restore T. A variant, called the bijective BWT [19, 12], is a *bijective* transformation, which does not require the artificial delimiter $\$$. It is based on the Lyndon factorization [5] of T. In this variant, the output consists of the last characters of the lexicographically sorted cyclic rotations of all factors composing the Lyndon factorization of T.

In the following, we call the BWT *traditional* to ease the distinguishability of both transformations. It is well known that the traditional BWT has many applications in data compression [1] and text indexing [8, 9, 10]. Recently, such a text index was adapted to work with the bijective BWT [2].

Related Work. 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 $\{1, \ldots, n^{\mathcal{O}(1)}\}$. Let us start with the traditional BWT, which we can construct thanks to linear time suffix array construction algorithms [23, 17] in linear time. 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 $\mathsf{SA}[i] = 1$. Considering the bijective BWT, Gil and Scott [12] 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 [22, after Example 9], which is $\mathcal{O}(n^2)$. In the same paper, Mantaci et al. [22] also introduced the *extended* BWT, a generalization of the BBWT in that it is a BWT based on a set $\mathcal{S}$ of *primitive* strings, i.e., strings that are not periodic. Hon et al. [15] provided an algorithm building the extended BWT in $\mathcal{O}(n \lg n)$ time. 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. [3] presented the most recent algorithm building the bijective BWT online in $\mathcal{O}(n \lg n / \lg \lg n)$ time. In [3, 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.

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 [23] to induce the BBWT. The last one takes an already computed suffix array SA as input, and modifies SA such that reading the characters $T[\mathsf{SA}[i] - 1]$ 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.

Our Result. In this article, we present a 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.

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 $\Sigma = \{1, \ldots, \sigma\}$ with size $\sigma = n^{\mathcal{O}(1)}$.

---

[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`

$T = \texttt{cbbcacbbcadacbadacba}$ → Lyndon Factorization → $\texttt{c} \mid \texttt{bbc} \mid \texttt{acbbcad} \mid \texttt{acbad} \mid \texttt{acb} \mid \texttt{a}$ → Collect the conjugates of all Lyndon factors → ... → Sort them in $\prec_\omega$-order → ... → Set BBWT[i] to last character of i-th entry → ... The corresponding starting position in the text:

Conjugates of Lyndon factors	Sorted in $\prec_\omega$-order	BBWT	Starting position
c	a	a	20
bbc	acb	b	17
bcb	acbad	d	12
cbb	acbbcad	d	5
acbbcad	adacb	b	15
cbbcada	adacbbc	c	10
bbcadac	bac	c	19
bcadacb	badac	c	14
cadacbb	bbcadac	c	7
adacbbc	bbc	c	2
dacbbca	bcadacb	b	8
acbad	bcb	b	3
cbada	cadacbb	b	9
badac	cba	a	18
adacb	cbada	a	13
dacba	cbbcada	a	6
acb	cbb	b	4
cba	c	c	1
bac	dacba	a	16
a	dacbbca	a	11

Figure 1 Constructing BBWT of $T = \texttt{cbbcacbbcadacbadacba}$. 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. 4.1. Note that $\texttt{cbb} \prec_{\text{lex}} \texttt{cbbcada}$, but $\texttt{cbbcada} \prec_\omega \texttt{cbb}$.

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. 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 string T is called *primitive* if it cannot be written as $T = S^k$ for a string $S \in \Sigma^+$ and $k \geq 2$.

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, $\texttt{ab} \prec_{\text{lex}} \texttt{aba}$ but $\texttt{aba} \prec_\omega \texttt{ab}$. The relation $\prec_\omega$ induces an order on the set of *primitive* strings[4], which we call $\prec_\omega$-*order*.

---

[4] The order cannot be generalized to strings in general since $\texttt{a} \neq \texttt{aa}$ but neither $\texttt{a} \prec_\omega \texttt{aa}$ nor $\texttt{aa} \prec_\omega \texttt{a}$ holds.

	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
	L	S*	S	L	S*	L	S*	S	L	S*	L	S*	L	L	S*	L	S*	L	L	S*

	1	2	3	4	5	6	7	8
$T^{(1)} =$	E	C	E	D	B	D	B	A
	L	S*	L	L	S*	L	L	S*

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. 4.3 and on the left of Fig. 3.

Lyndon Words. Given a primitive string $T = T[1..n]$, its i-th *conjugate* $\mathrm{conj}_i(T)$ is defined as $T[i+1..n]T[1..i]$ for an integer $i \in [0..n-1]$. Since T is primitive, all its conjugates are distinct. We say that T and every one of its conjugates belongs to the *conjugate class* $\mathrm{conj}(T) := \{\mathrm{conj}_0(T), \ldots, \mathrm{conj}_{n-1}(T)\}$. If a conjugate class contains *exactly* one conjugate that is lexicographically smaller than all other conjugates, then this conjugate is called a *Lyndon word* [21]. 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*.

The *Lyndon factorization* [5] of $T \in \Sigma^+$ is the unique factorization of T into a sequence of Lyndon words $F_1 \cdots F_z$, where (a) each $F_x \in \Sigma^+$ is a Lyndon word, and (b) $F_x \succeq_{\mathrm{lex}} F_{x+1}$ for each $x \in [1..z)$.

▶ **Lemma 1** ([7, Algo. 2.1]). *The Lyndon factorization of a string can be computed in linear time.*

Each Lyndon word F_x for $x \in [1..z]$ is called a *Lyndon factor*. For what follows, we fix a string $T[1..n]$ over an alphabet Σ of size σ. We use the string $T := \texttt{cbbcacbbcadacbadacba}$ as our running example. Its Lyndon factorization is $\texttt{c}, \texttt{bbc}, \texttt{acbbcad}, \texttt{acbad}, \texttt{acb}, \texttt{a}$.

Bijective Burrows–Wheeler Transform. We denote the bijective BWT of T by BBWT, where BBWT$[i]$ is the last character of the i-th string in the list storing the conjugates of all Lyndon factors $F_1, \ldots, F_z$ of T sorted with respect to $\prec_\omega$. Figure 1 shows the BBWT of our running example.

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

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 S* suffix (also called LMS suffix in [23]) if $T[i-1..|T|]$ is an L suffix. The substring between two succeeding 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 S* suffixes and there is no $k \in (i..j)$ such that $T[k..|T|]$ is an 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 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).

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

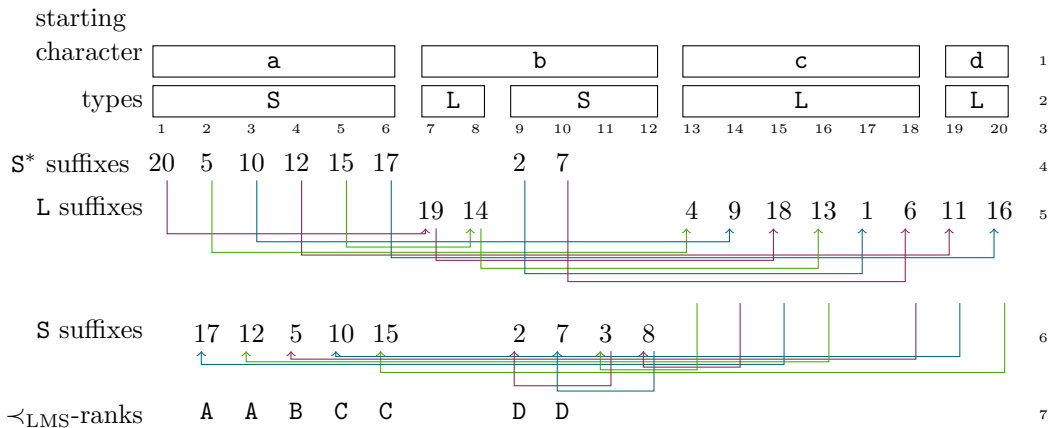| $\mathtt{S}^*$ Suffix	Contents
$T[20]$	a
$T[17..20]$	acba
$T[12..20]$	acbadacba
$T[5..20]$	acbbcadacbadacba
$T[15..20]$	adacba
$T[10..20]$	adacbadacba
$T[2..20]$	bbcacbbcadacbadacba
$T[7..20]$	bbcadacbadacba

■ **Figure 3** Ranking of the LMS substrings and the $\mathtt{S}^*$ suffixes of our running example given in Sect. 4.3 and Fig. 2. *Left*: LMS substrings assigned with non-terminals reflecting their corresponding rank in $\prec_{\mathrm{LMS}}$-order. *Right*: $\mathtt{S}^*$ suffixes of T sorted in $\prec_{\mathrm{lex}}$-order. Note that $T[5..7] = \mathtt{acb} \prec_{\mathrm{lex}} \mathtt{acba} = T[12..15] = T[17..20]$, but $\mathtt{acba} \prec_{\mathrm{LMS}} \mathtt{acb}$.

Next, Nong et al. [23, Def. 3.3] define a relation $\prec_{\mathrm{LMS}}$ 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 $S \prec_{\mathrm{LMS}} U$. For two LMS substrings S and U with $S \neq U$, either $S \prec_{\mathrm{LMS}} U$ or $U \prec_{\mathrm{LMS}} S$, even if S is a prefix of U (cf. the discussion below of Def. 3.3 in [23]). So $\prec_{\mathrm{LMS}}$ is an order on the LMS substrings. The $\prec_{\mathrm{LMS}}$-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 $\prec_{\mathrm{LMS}}$-order of the LMS substrings coincides with the lexicographic order of the suffixes starting with the LMS substrings [23, Lemma 3.8].
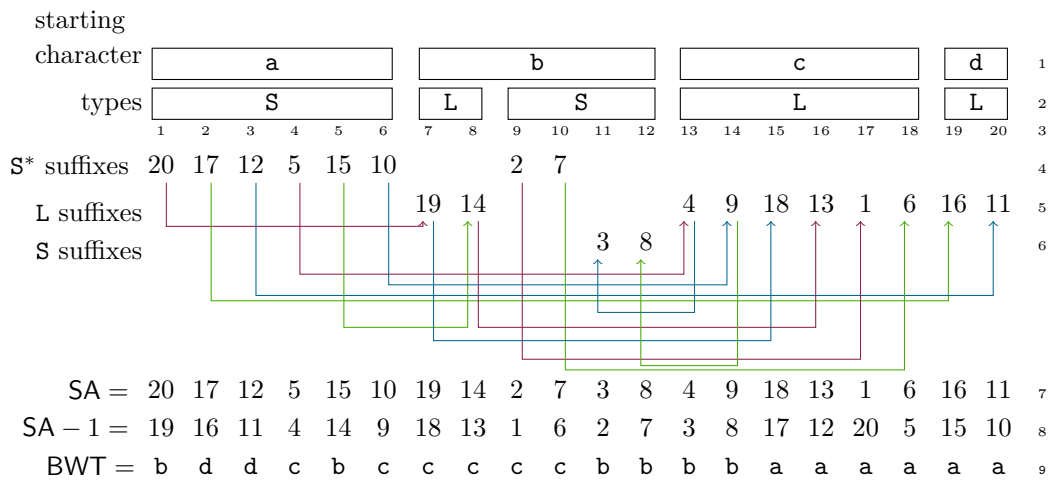
Nong et al. [23, A3.4] compute the $\prec_{\mathrm{LMS}}$-order of all LMS substrings with the induced sorting (which we describe below for the step of computing the rank of all suffixes). Figure 4 visualizes this computation on our running example. Hence, we can assign each LMS substring a rank based on the $\prec_{\mathrm{LMS}}$-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 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 $\mathtt{S}^*$ suffixes of $T^{(k)}$. The order of the $\mathtt{S}^*$ suffixes of our running example are given in Fig. 3 on the right side. Having the order of the $\mathtt{S}^*$ 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 [18, Corollary 3]. Putting $\mathtt{S}^*$ 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 $\mathtt{S}^*$ 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 $\prec_{\mathrm{lex}}$-order of the S suffixes by scanning the suffix array from right

---

[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 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 $\prec_{\mathrm{LMS}}$-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 $\mathcal{O}(|T|)$.

Figure 5 Inducing L and S suffixes from the $\prec_{\mathrm{lex}}$-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 $\prec_{\mathrm{lex}}$-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 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 $T[(\mathsf{SA}-1)[i]] = \mathrm{BWT}[i]$ 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 $T[\mathsf{SA}[1]] = T[20] = \mathtt{a}$ to the output, and then BWT, but exchanging $\mathrm{BWT}[\mathsf{SA}^{-1}[1]] = \mathrm{BWT}[17] = \mathtt{a}$ 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 $\mathtt{d}$ (i.e., $T = \ldots \mathtt{d} U \mathtt{d} \ldots \mathtt{d} V \mathtt{d} \ldots$) such that the first and the last character of both U and V start with an $\mathtt{S}^*$ 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 $\ldots \mathtt{d} U \mathtt{a} \ldots \mathtt{d} V \mathtt{d} \ldots$, for instance.

to left: When accessing the entry $\mathsf{SA}[k] = i$, write $i - 1$ to the rightmost available slot of the $\mathtt{S}$ type bucket with the character $T[i-1]$ if $T[i-1..|T|]$ is an $\mathtt{S}$ suffix. As an invariant, we always fill an $\mathtt{L}$ bucket and an $\mathtt{S}$ bucket from left to right and from right to left, respectively. So we can think of each $\mathtt{L}$ bucket and each $\mathtt{S}$ 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 induction 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 $\mathtt{S}^*$ for $i \in [1..n-1]$). This gives $\mathcal{T}(n) = \mathcal{T}(n/2) + \mathcal{O}(n) = \mathcal{O}(n)$ total time, where $\mathcal{T}(n)$ 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 exchange $\prec_{\text{lex}}$ with $\prec_{\omega}$. Although these orders are the same for Lyndon words [3, Thm. 8], they differ for LMS substrings as can be seen in Fig. 6. Hence, we need to come up with an idea to modify SAIS in such way to compute $\mathsf{SA}_\circ$.

4   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.

4.1   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):

The (composed) *Lyndon factorization* [5] of $T \in \Sigma^+$ is the factorization of T into $T_1^{\tau_1} \cdots T_t^{\tau_t} = T$, where $T_1, \ldots, T_t$ is a sequence of lexicographically decreasing Lyndon words and $\tau_x \geq 1$ for $x \in [1..t]$. 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 $\mathsf{b}(T_x)$ and $\mathsf{e}(T_x)$ denote the starting and ending position of the x-th Lyndon factor in R, i.e., $R[\mathsf{b}(T_x)..\mathsf{e}(T_x)]$ is the x-th Lyndon factor T_x of R.

Our aim is to compute the $\prec_\omega$-order of all conjugates of all Lyndon factors of R, which are given by the set $\mathcal{S} := \bigcup_{x=1}^{t} \mathrm{conj}(T_x)$. Like Hon et al. [14], we present this order in the so-called *circular suffix array* $\mathsf{SA}_\circ$ of $\{T_1, \ldots, T_t\}$, i.e., an array of length $|R|$ with $\mathsf{SA}_\circ[k] = i$ if $R[i..\mathsf{e}(T_x)]R[\mathsf{b}(T_x)..i-1]$ is the k-th smallest string in $\mathcal{S}$ with respect to $\prec_\omega$, where $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$. 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 $\mathsf{SA}_\circ[k] = i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$, we append $T[i^-]$ exactly τ_x times to BBWT, where i^- is $i-1$ or $\mathsf{e}(T_x)$ if $i = \mathsf{b}(T_x)$. (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.)

4.2 Translating Types to Inf-Suffixes

In what follows, we continue working with R defined in Sect. 4.1 instead of T. Let $R[i..]$ denote the infinite string $R[i..\mathsf{e}(T_x)]T_x T_x \cdots = \mathrm{conj}_k(T_x)\mathrm{conj}_k(T_x)\cdots$ with x such that $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$ and $k = i - \mathsf{b}(T_x)$. We say that $R[i..]$ is an *inf-suffix*. As a shorthand, we also write $T_x[i..] = \mathrm{conj}_{i-1}(T_x)\mathrm{conj}_{i-1}(T_x)\cdots$ for the inf-suffix starting at $R[\mathsf{b}(T_x) + i - 1]$. In particular, $T_x[|T_x| + 1..] = T_x[1..] = T_x T_x \cdots$.

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

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

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

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 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 2** ([3, 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. $T_x[i..|T_x|] \prec_{\mathrm{lex}} T_x[j..|T_x|].$

▶ **Lemma 3.** *Omitting all Lyndon factors of length one from R, the types of all positions match the original SAIS types, except maybe $R[1]$ and $R[\mathsf{b}(T_t) + 1..|R|]$, where $R[1..]$ and $R[|R|..|R|]$ are always an S^* inf-suffix and an S^* suffix, respectively.*

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

inf-suffxes. Assume that $R[\mathsf{b}(T_x)..]$ is an L inf-suffix for an $x \in [1..t]$. According to the definition $R[\mathsf{b}(T_x) + 1..] \prec_{\mathrm{lex}} R[\mathsf{b}(T_x)..]$, i.e., $T_x[2..] \prec_{\mathrm{lex}} T_x[1..]$, and with Lemma 2, $T_x[2..|T_x|] \prec_{\mathrm{lex}} T_x$, contradicting that T_x is a Lyndon word. Finally, $R[\mathsf{b}(T_x)..]$ is an S^* inf-suffix because $T_x \prec_{\mathrm{lex}} T_x[|T_x|]$ and hence $T_x[1..] \prec_{\mathrm{lex}} T_x[|T_x|..]$, again with Lemma 2.
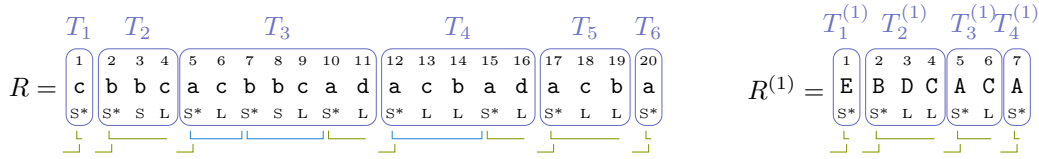
Figure 7 Splitting R and $R^{(1)}$ into LMS inf-substrings. The rectangular brackets below the types represent the LMS inf-substrings. Broken brackets denote that the corresponding LMS inf-substring ends with the first character of the Lyndon factor in which it is contained. They are colored in green (■); 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. 4.3 and on the left of Fig. 8.

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

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

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

A corollary is that $R[i..|R|] \prec_{\mathrm{lex}} R[i..]$ for $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$ and $x \in [1..t-1]$ since $T_{x+1} \prec_{\mathrm{lex}} T_x$.[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 $(T_x T_x)[i..j]$ is called an *LMS inf-substring* if and only if $T_x[i..]$ and $T_x[j..]$ are S^* inf-suffixes and there is no $k \in (i..j)$ such that $T_x[k..]$ is an 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 S^* position of T_x.

4.3 Example

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

---

[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 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
$R[10..11]R[10]$	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	-

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 S* suffixes of our running example $T = R$ given in Sect. 4.3 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*: 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.
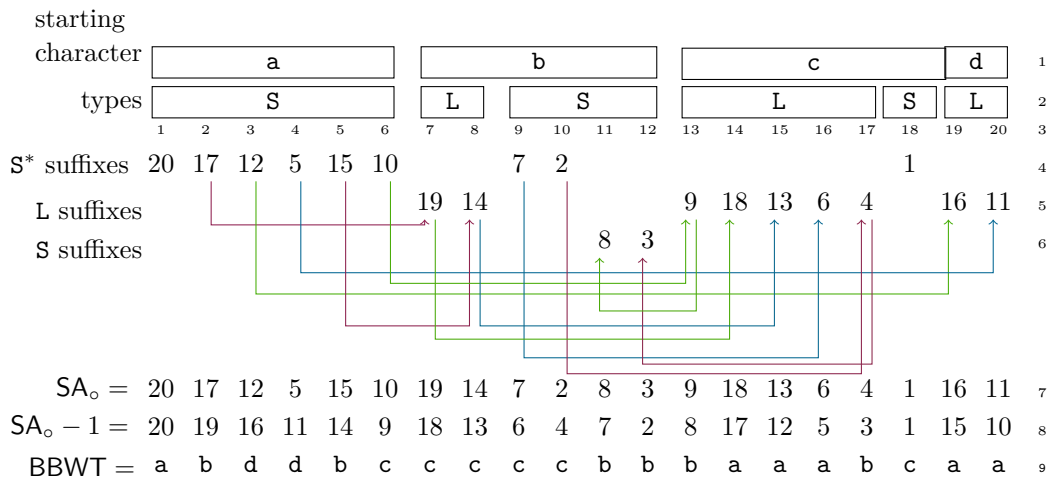
Figure 9 Inducing L and S inf-suffixes from the $\prec_{\text{lex}}$-order of the S* inf-suffixes given in Fig. 7. Rows 1 and 2 show the partitioning of $\mathsf{SA}_\circ$ into buckets, first divided by the starting characters of the respective inf-suffixes, and second by the types L and S. Row 4 is $\mathsf{SA}_\circ$ after inserting the S* inf-suffixes according to their $\prec_{\text{lex}}$-order rank obtained from the right of Fig. 8. The 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, $(\mathsf{SA}_\circ - 1)[2] = 19 = \mathsf{e}(T_5)$ since $\mathsf{SA}_\circ[2] = 17 = \mathsf{b}(T_5)$). 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.

4.4   Correctness and Time Complexity

Let us recall that our task is to compute the $\prec_\omega$-order of the conjugates $\mathrm{conj}_{i_x-1}(T_x)$ for $i_x \in [1..|T_x|]$ of all Lyndon factors $T_1, \ldots, T_t$ of R. We will frequently use that $\mathrm{conj}_{i_x-1}(T_x) \prec_\omega$ $\mathrm{conj}_{i_y-1}(T_y)$ is equivalent to $T_x[i_x..] \prec_{\mathrm{lex}} T_y[i_y..]$ for $i_x \in [1..|T_x|]$ and $i_y \in [1..|T_y|]$. We start with showing that the $\prec_{\mathrm{LMS}}$-ranks of the LMS inf-substrings determine the $\prec_{\mathrm{lex}}$-order of the $\mathtt{S}^*$ inf-suffixes[7], whenever the LMS inf-suffixes are all distinct.

▶ **Lemma 4.** *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 $S_x \prec_{\mathrm{LMS}} S_y$ then $T_x[i_x..] \prec_{\mathrm{lex}} T_y[i_y..]$.*

Proof. Given $S_x \prec_{\mathrm{LMS}} S_y$, 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 $T_x[k] = (T_x T_x \cdots)[k]$ for a $k \in [1..2|T_x|]$. In both cases (a) and (b), $T_x[i_x..] \prec_{\mathrm{lex}} T_x[i_y..]$.   ◀

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

▶ **Lemma 5.** *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 $\prec_{\mathrm{LMS}}$-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 $\prec_{\mathrm{LMS}}$-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, $r_1 \cdots r_d \prec_{\mathrm{lex}} r_1 \cdots r_\ell \prec_{\mathrm{lex}} r_{d+1} \cdots r_\ell$. Hence, $T_x[s..|T_x|] \prec_{\mathrm{LMS}} T_x[s..|T_x|]T_x[1] \prec_{\mathrm{LMS}} P$, and with Lemma 4, $T_x[1..] \prec_{\mathrm{lex}} T_x[s..] \prec_{\mathrm{lex}} T_{x+1}[1..]$, contradicting the Lyndon factorization of R with Lemma 2.

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 $L_2 \prec_{\mathrm{lex}} L_1$, which means that T_x contains an inf-suffix smaller than T_x due to Lemma 2, contradicting that T_x is a Lyndon factor.   ◀

Thanks to Lemma 5, 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 $\prec_{\mathrm{LMS}}$-ranks of the LMS inf-substrings can be computed:

▶ **Lemma 6.** *We can compute the $\prec_{\mathrm{LMS}}$-ranks of all LMS inf-substrings in linear time.*

---

[7] This is a counterpart to the property that the $\prec_{\mathrm{LMS}}$-ranks of the LMS substrings determine the $\prec_{\mathrm{lex}}$-order of the $\mathtt{S}^*$ suffixes [23, Theorem 3.12].
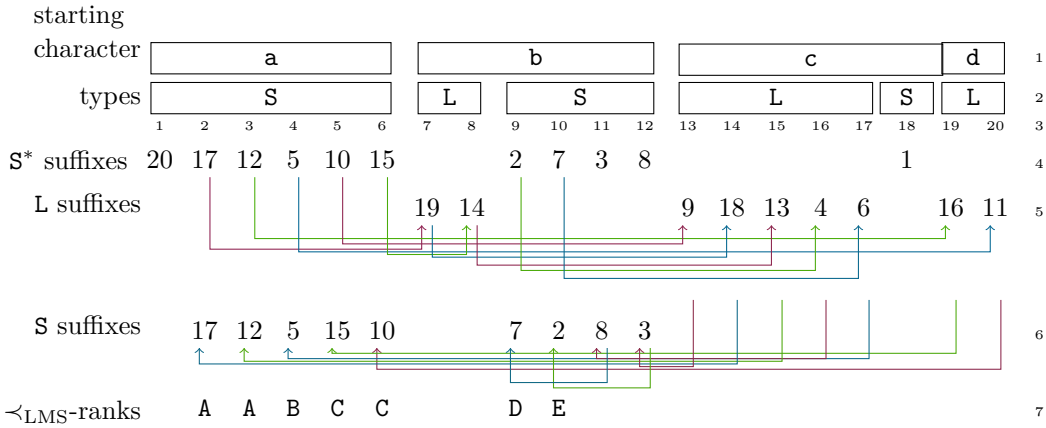
Figure 10 Inducing LMS inf-substrings. Thanks to the Lyndon factorization, we know the $\prec_\omega$-order of the inf-suffixes starting with the Lyndon factors, which is $T[20..] \prec_\omega T[17..] \prec_\omega T[12..] \prec_\omega T[5..] \prec_\omega T[2..] \prec_\omega T[1..]$. We insert the starting positions of these inf-suffixes in this order into their respective buckets, and fill the S^* buckets with the rest of S^* inf-suffixes by an arbitrary order (here we used the text order). Like Fig. 4, the S^* (resp. L) suffixes induce the L (resp. 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.

Proof. We follow the proof of [23, Theorem 3.12]. The idea is to know the $\prec_{\mathrm{lex}}$-order among some smallest S^* inf-suffixes with which we can induce the $\prec_{\mathrm{LMS}}$-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 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 S^* suffix (cf. Fig. 4), which can be determined by appending an artificial character such as \$ 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 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 $T_x \succ_{\mathrm{lex}} T_{x+1}$ is equivalent to $T_x \succ_\omega T_{x+1}$ due to [3, Thm. 8], and hence we know the $\prec_{\mathrm{lex}}$-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 $\prec_{\mathrm{LMS}}$-ranks of all LMS inf-substrings.

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 [23, Sect. 3.4]: Let $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$ for an $x \in [1..t]$ be a text position, and let $j > i$ be the next S^* position in R. Then the *LMS-prefix* P_i starting at position i is $P_i := R[i..j]$ if $j \le \mathsf{e}(T_x)$ or $P_i := R[i..j-1]\mathsf{b}(T_x)$

---

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

if $j = \mathsf{b}(T_{x+1})$. In particular, if i is the starting position of an LMS inf-substring S, then $P_i = S$. The LMS-prefixes inherit the types (L or S) from their starting positions. We show that we can compute the $\prec_{\mathrm{LMS}}$-ranks of all P_i's by induce 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 S buckets of the respective starting characters in lexicographically sorted order. See also Fig. 10 for an example.

Inducing 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 $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$ and $i^- = i-1$ or $i^- = \mathsf{e}(T_x)$ for $i = \mathsf{b}(T_x)$, we insert i^- into the L bucket of the character $T_x[i^-]$ if $R[i^-..]$ is an L inf-suffix. By doing so, we compute the $\prec_{\mathrm{LMS}}$-order of all L LMS-prefixes in ascending lexicographic order per L bucket. The correctness follows by induction over the number k of inserted L LMS-prefixes. Since we know that all LMS-prefixes $P_{\mathsf{b}(T_x)}$ 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 L LMS-prefix, we perform the insertion of the first L LMS-prefix correctly, which is induced by the lexicographically smallest 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 L LMS-prefix P_i into its corresponding bucket, we have stored an 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 did a mistake.

The inducing step for the 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 $\prec_{\mathrm{LMS}}$-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 6, we can determine the $\prec_\omega$-order of the S^* inf-suffixes R. It is left to perform the induction step to induce first the order of the L inf-suffixes, and subsequently the S inf-suffixes, which we do in the same manner as SAIS, but access $(T_x T_x \cdots)[i^-]$ instead of $R[i-1]$ when accessing a suffix array entry with value i, where x chosen such that $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$ and $i^- = i-1$ or $i^- = \mathsf{e}(T_x)$ if $i = \mathsf{b}(T_x)$. The correctness follows by construction: Instead of partitioning the suffixes into LMS substrings (maybe omitting a prefix of R with 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 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 b (the following holds if $\mathsf{b} \in \Sigma$ or if b is a rank of an LMS substring considered in the recursive call). All LMS inf-substrings larger than one starting with b are larger than bb in the $\prec_\omega$-order because such an LMS inf-substring starting with $R[i]$ having type S^* is lexicographically

smaller than $R[i+1..]$. Consequently, $\mathtt{bb}\cdots \prec_{\mathrm{lex}} R[i..] = \mathtt{b}R[i+1..]$ since $\mathtt{b}\cdots \prec_{\mathrm{lex}} R[i+1..]$. 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 simply insert all omitted LMS inf-substrings into the order returned by the recursive call by a linear scan. 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. 4.1.

4.5   Space Complexity

Given that $z = \sum_{x=1}^{t} \tau_x$ is the number of all non-composed Lyndon factors $F_1 \cdots F_z$, the algorithm of Lemma 1 computing the Lyndon factorization online only needs to maintain three integer variables of $\mathcal{O}(\lg n)$ bits to find $F_1 \cdots F_z$. 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 ($x \in [1..z]$) with a one. We additionally create a bit vector B_2 of length z, and mark the first occurrence of each non-composed Lyndon factor F_x in B_2 for $x \in [1..z]$ 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 $(z-t)\lg \sigma \geq z-t$ 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 [16]. A rank-support data structure provides support for a *rank query*, i.e., retrieving the number of ones up to a queried position in B. Since a recursive call of SAIS works on a text instance of at most $|R|/2$ characters, we can rebuild B from scratch by rerunning the algorithm of Lemma 1 on $R^{(1)}$ or after finalizing the 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 [13] (or similarly with [20]), 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.

5 Conclusion

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 – induce sorting (SAIS) algorithm to compute the $\prec_\omega$-order of the conjugates of all Lyndon factors of R instead of the $\prec_{\mathrm{lex}}$-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 induce sorting techniques of SAIS, and obtain the correctness of our result.

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.

We also wonder whether we can define the BBWT for the generalized Lyndon factorization [6]. Contrary to the Lyndon factorization, the generalized Lyndon factorization uses a different order, called the *generalized lexicographic order* $\prec_{\mathrm{gen}}$. In this order, two strings $S, T \in \Sigma^*$ are compared character-wise like in the lexicographic order. However, the generalized lexicographic order $\prec_{\mathrm{gen}}$ can use different orders $<_1, <_2, \ldots$ for each text position, i.e., $S \prec_{\mathrm{gen}} T$ 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 $S[\ell] <_\ell T[\ell]$.

Recently, Gibney and Thankachan [11] 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 the BBWT index of [2]. However, it is left open, whether we can find an efficient algorithm that approximates the alphabet order maximizing the number of Lyndon factor.

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.

───── **References** ─────

1   Donald Adjeroh, Timothy Bell, and Amar Mukherjee. *The Burrows–Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching.* Springer, 2008.

2 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Indexing the bijective BWT. In *Proc. CPM*, volume 128 of *LIPIcs*, pages 17:1–17:14, 2019.

3 Silvia Bonomo, Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Sorting conjugates and suffixes of words in a multiset. *Int. J. Found. Comput. Sci.*, 25(8):1161, 2014.

4 Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

5 Kuo Tsai Chen, Ralph H. Fox, and Roger C. Lyndon. Free differential calculus, IV. The quotient groups of the lower central series. *Annals of Mathematics*, pages 81–95, 1958.

6 Francesco Dolce, Antonio Restivo, and Christophe Reutenauer. On generalized Lyndon words. *Theor. Comput. Sci.*, 777:232–242, 2019.

7 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.

8 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398, 2000.

9 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

10 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. SODA*, pages 1459–1477, 2018.

11 Daniel Gibney and Sharma 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.

12 Joseph Yossi Gil and David Allen Scott. A bijective string sorting transform. *ArXiv 1201.3077*, 2012. `arXiv:1201.3077`.

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

14 Wing-Kai Hon, Tsung-Han Ku, Chen-Hua Lu, Rahul Shah, and Sharma V. Thankachan. Efficient algorithm for circular Burrows–Wheeler transform. In *Proc. CPM*, volume 7354 of *LNCS*, pages 257–268, 2012.

15 Wing-Kai Hon, Chen-Hua Lu, Rahul Shah, and Sharma V. Thankachan. Succinct indexes for circular patterns. In *Proc. ISAAC*, volume 7074 of *LNCS*, pages 673–682, 2011.

16 Guy Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554, 1989.

17 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

18 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005.

19 Manfred Kufleitner. On bijective variants of the Burrows–Wheeler transform. In *Proc. PSC*, pages 65–79, 2009.

20 Zhize Li, Jian Li, and Hongwei Huo. Optimal in-place suffix sorting. In *Proc. SPIRE*, volume 11147 of *LNCS*, pages 268–284, 2018.

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

22 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows–Wheeler transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007.

23 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.