In-Place Bijective Burrows-Wheeler Transforms

Dominik Köppl 🄳
Department of Informatics, Kyushu University, Fukuoka, Japan
Japan Society for Promotion of Science (JSPS), Tokyo, Japan
`https://dkppl.de/`
dominik.koeppl@inf.kyushu-u.ac.jp

Daiki Hashimoto
Graduate School of Information Sciences, Tohoku University, Sendai, Japan
daiki_hashimoto@shino.ecei.tohoku.ac.jp

Diptarama Hendrian 🄳
Graduate School of Information Sciences, Tohoku University, Sendai, Japan
diptarama@tohoku.ac.jp

Ayumi Shinohara 🄳
Graduate School of Information Sciences, Tohoku University, Sendai, Japan
ayumis@tohoku.ac.jp

───── **Abstract** ─────

One of the most well-known variants of the Burrows-Wheeler transform (BWT) [Burrows and Wheeler, 1994] is the bijective BWT (BBWT) [Gil and Scott, arXiv 2012], which applies the extended BWT (EBWT) [Mantaci et al., TCS 2007] to the multiset of Lyndon factors of a given text. Since the EBWT is invertible, the BBWT is a bijective transform in the sense that the inverse image of the EBWT restores this multiset of Lyndon factors such that the original text can be obtained by sorting these factors in non-increasing order.

In this paper, we present algorithms constructing or inverting the BBWT in-place using quadratic time. We also present conversions from the BBWT to the BWT, or vice versa, either (a) in-place using quadratic time, or (b) in the run-length compressed setting using $\mathcal{O}(n \lg r / \lg \lg r)$ time with $\mathcal{O}(r \lg n)$ bits of words, where r is the sum of character runs in the BWT and the BBWT.

1 Introduction

The *Burrows-Wheeler transform* (BWT) [6] is one of the most favored options both for (a) compressing and (b) indexing data sets. On the one hand, compression programs like `bzip2` apply the BWT to achieve high compression rates. For that, they leverage the effect that the BWT built on repetitive data tends to have long character runs, which can be

compressed by run-length compression, i.e., representing a substring of ℓ a's by the tuple $(\mathtt{a}, \ell)$. On the other hand, self-indexing data structures like the FM-index [11] enhance the BWT to a full-text self-index. A combined approach of both compression and indexing is the run-length compressed FM-index [21], representing a BWT with r_{BWT} character runs, i.e., maximal repetitions of a character, run-length compressed in $\mathcal{O}(r_{\mathsf{BWT}} \lg n)$ bits. This representation can be computed directly in run-length compressed space thanks to Policriti and Prezza [30]. The BWT and its run-length compressed representation have been intensively studied during the past decades (e.g., [12, 1, 14] and the references therein). Contrary to that, a variant, called the *bijective* BWT (BBWT) [16], is far from being well-studied despite its mathematically appealing characteristics[1]. As a matter of fact, we are only aware of one index data structure based on the BBWT [3] and of two non-trivial construction algorithms [5, 2] of the (uncompressed) BBWT, both with the need of additional data structures.

In this article, we shed more light on the connection between the BWT and the BBWT by quadratic time in-place conversion algorithms in Sect. 5 constructing the BWT from the BBWT, or vice versa. We can also perform these conversions in the run-length compressed setting in $\mathcal{O}(n \lg r / \lg \lg r)$ time with space linear to the number of the character runs (cf. Sect. 4 and Thm. 3), where r is the sum of character runs in the BWT and the BBWT.

2 Related Work

Given a text T of length n, the BWT of T is the string obtained by assigning $\mathsf{BWT}[i]$ to the character preceding the i-th lexicographically smallest suffix of T (or the last character of T if this suffix is the text itself). By this definition, we can construct the BWT with any suffix array [22] construction algorithm. However, storing the suffix array inherently needs $n \lg n$ bits of space. Crochemore et al. [9] tackled this space problem with an in-place algorithm constructing the BWT in $\mathcal{O}(n^2)$ online on the reversed text by simulating queries on a dynamic wavelet tree [17] that would be built on the (growing) BWT. They also gave an algorithm for restoring the text in-place in $\mathcal{O}(n^{2+\epsilon})$ time.

In the run-length compressed setting, Policriti and Prezza [30] can compute the run-length compressed BWT having r_{BWT} character runs in $\mathcal{O}(n \lg r_{\mathsf{BWT}})$ time while using $\mathcal{O}(r_{\mathsf{BWT}} \lg n)$ bits of space. They additionally presented an adaption of the wavelet tree on run-length compressed texts, yielding a representation using $\mathcal{O}(r_{\mathsf{BWT}} \lg n)$ bits of space with $\mathcal{O}(\lg r_{\mathsf{BWT}})$ query and update time. Finally, practical improvements of the run-length compressed BWT construction were considered by Ohno et al. [29].

The BBWT is the string obtained by assigning $\mathsf{BBWT}[i]$ to the last character of the i-th smallest string in the list of all conjugates of the factors of the Lyndon factorization sorted with respect to the $\prec_\omega$ order [23, Def. 4]. Bannai et al. [2] recently revealed a connection between the bijective BWT and suffix sorting by presenting an $\mathcal{O}(n)$ time BBWT construction algorithm based on SAIS [28]. With dynamic data structures like a dynamic wavelet tree [27], Bonomo et al. [5] could devise an algorithm computing the BBWT in $\mathcal{O}(n \lg n / \lg \lg n)$ time. With nearly the same techniques, Mantaci et al. [24] presented an algorithm computing the BWT (and simultaneously the suffix array if needed) from the Lyndon factorization. All these construction algorithms need however data structures taking $\mathcal{O}(n \lg n)$ bits of space. However, the latter two (i.e., [5] and [24]) can work in-place by simulating the LF mapping (cf. Sects. 3.4 and 3.5), which we focus on in Sect. 5.1.

---

[1] The BBWT is a bijection between strings without the need of an artificial delimiter needed, e.g., to invert the BWT.

3   Preliminaries

Our computational model is the word RAM model with word size $\Omega(\lg n)$. Accessing a word costs $\mathcal{O}(1)$ time. An algorithm is called *in-place* if it uses, besides a rewriteable input, only $\mathcal{O}(\lg n)$ bits of working space. We write $[\mathsf{b}(I) \ldots \mathsf{e}(I)] = I$ for an interval I of natural numbers.

3.1   Strings

Let Σ denote an integer alphabet of size σ with $\sigma = n^{\mathcal{O}(1)}$. We call an element $T \in \Sigma^*$ a *string*. Its length is denoted by $|T|$. Given an integer $j \in [1 \ldots |T|]$, we access the j-th character of T with $T[j]$. Concatenating a string $T \in \Sigma^*$ k times is abbreviated by T^k. A string T is called *primitive* if there is no string $S \in \Sigma^+$ with $T = S^k$ for an integer k with $k \geq 2$.

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; the prefix X, substring Y, or suffix Z is called *proper* if $X \neq T$, $Y \neq T$, or $Z \neq T$, respectively. For two integers i, j with $1 \leq i \leq j \leq |T|$, let $T[i \ldots j]$ denote the substring of T that begins at position i and ends at position j in T. If $i > j$, then $T[i \ldots j]$ is the empty string. In particular, the suffix starting at position j of T is called the *j-th suffix* of T, and denoted with $T[j \ldots]$. An occurrence of a substring S in T is treated as a sub-interval of $[1 \ldots |T|]$ such that $S = T[\mathsf{b}(S) \ldots \mathsf{e}(S)]$. The *longest common prefix (LCP)* of two strings S and T is the longest string that is a prefix of both S and T.

Orders on Strings.   We denote the *lexicographic order* with $\prec_{\mathrm{lex}}$. Given two strings S and T, $S \prec_{\mathrm{lex}} T$ if S is a prefix of T or there exists an integer ℓ with $1 \leq \ell \leq \min(|S|, |T|)$ such that $S[1 \ldots \ell - 1] = T[1 \ldots \ell - 1]$ and $S[\ell] < T[\ell]$. Next we define the $\prec_\omega$ *order* of strings, which is based on the lexicographic order of infinite strings: We write $S \prec_\omega T$ if the infinite concatenation $S^\omega := SSS \cdots$ is lexicographically smaller than $T^\omega := TTT \cdots$. For instance, $\mathtt{ab} \prec_{\mathrm{lex}} \mathtt{aba}$ but $\mathtt{aba} \prec_\omega \mathtt{ab}$.

Rank and Select Queries.   Given a string $T \in \Sigma^*$, a character $\mathsf{c} \in \Sigma$, and an integer j, the *rank* query $T.\mathsf{rank_c}(j)$ counts the occurrences of c in $T[1 \ldots j]$, and the *select* query $T.\mathsf{select_c}(j)$ gives the position of the j-th c in T. We stipulate that $\mathsf{rank_c}(0) = \mathsf{select_c}(0) = 0$. A *wavelet tree* is a data structure supporting rank and select queries.

3.2   Lyndon Words

Given a string $T = T[1 \ldots n]$, its i-th *conjugate* $\mathrm{conj}_i(T)$ is defined as $T[i+1 \ldots n]T[1 \ldots i]$ for an integer $i \in [0 \ldots n-1]$. We say that T and all of its conjugates belong 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* [20]. Equivalently, a string T is said to be a Lyndon word if and only if $T \prec_{\mathrm{lex}} S$ for every proper suffix S of T [10, Prop. 1.2].

The *Lyndon factorization* [8] of $T \in \Sigma^+$ is the factorization of T into a sequence of lexicographically non-increasing Lyndon words $T_1 \cdots T_t$, where (a) each $T_x \in \Sigma^+$ is a Lyndon word for $x \in [1 \ldots t]$, and (b) $T_x \succeq_{\mathrm{lex}} T_{x+1}$ for each $x \in [1 \ldots t]$. Each Lyndon word T_x is called a *Lyndon factor*.

F — BBWT | F — BWT° | F — BWT

	F	BBWT		F	BWT°		F	BWT
1	$a_4\,b_5\,b_6$	$a_4\,b_5\,b_6$	1	$a_4\,b_5\,b_6\,a_7\,b_8\,b_9\,b_1\,a_2\,c_3$		1	$\$\ b_1\,a_2\,c_3\,a_4\,b_5\,b_6\,a_7\,b_8\,b_9$	
2	$a_7\,b_8\,b_9$	$a_7\,b_8\,b_9$	2	$a_7\,b_8\,b_9\,b_1\,a_2\,c_3\,a_4\,b_5\,b_6$		2	$a_7\,b_8\,b_9\,\$\ b_1\,a_2\,c_3\,a_4\,b_5\,b_6$	
3	$a_2\,c_3$	$a_2\,c_3$	3	$a_2\,c_3\,a_4\,b_5\,b_6\,a_7\,b_8\,b_9\,b_1$		3	$a_4\,b_5\,b_6\,a_7\,b_8\,b_9\,\$\ b_1\,a_2\,c_3$	
4	$b_6\,a_4\,b_5$	$b_6\,a_4\,b_5$	4	$b_6\,a_7\,b_8\,b_9\,b_1\,a_2\,c_3\,a_4\,b_5$		4	$a_2\,c_3\,a_4\,b_5\,b_6\,a_7\,b_8\,b_9\,\$\ b_1$	
5	$b_9\,a_7\,b_8$	$b_9\,a_7\,b_8$	$T\!\rightarrow$ 5	$b_1\,a_2\,c_3\,a_4\,b_5\,b_6\,a_7\,b_8\,b_9$		5	$b_9\,\$\ b_1\,a_2\,c_3\,a_4\,b_5\,b_6\,a_7\,b_8$	
6	$b_5\,b_6\,a_4$	$b_5\,b_6\,a_4$	6	$b_5\,b_6\,a_7\,b_8\,b_9\,b_1\,a_2\,c_3\,a_4$		6	$b_6\,a_7\,b_8\,b_9\,\$\ b_1\,a_2\,c_3\,a_4\,b_5$	
7	$b_8\,b_9\,a_7$	$b_8\,b_9\,a_7$	7	$b_9\,b_1\,a_2\,c_3\,a_4\,b_5\,b_6\,a_7\,b_8$	$T\$\!\rightarrow$ 7	$b_1\,a_2\,c_3\,a_4\,b_5\,b_6\,a_7\,b_8\,b_9\,\$$	
8	b_1	b_1	8	$b_8\,b_9\,b_1\,a_2\,c_3\,a_4\,b_5\,b_6\,a_7$		8	$b_8\,b_9\,\$\ b_1\,a_2\,c_3\,a_4\,b_5\,b_6\,a_7$	
9	$c_3\,a_2$	$c_3\,a_2$	9	$c_3\,a_4\,b_5\,b_6\,a_7\,b_8\,b_9\,b_1\,a_2$		9	$b_5\,b_6\,a_7\,b_8\,b_9\,\$\ b_1\,a_2\,c_3\,a_4$	
						10	$c_3\,a_4\,b_5\,b_6\,a_7\,b_8\,b_9\,\$\ b_1\,a_2$	

Figure 1 All three BWT variants studied in this paper applied on our running example $T =$ `bacabbabb`. *Left*: BBWT built on the last characters of the conjugates of all Lyndon words sorted in the $\prec_\omega$ order. *Middle* and *Right*: BWT° and BWT built on the lexicographically sorted conjugates of T and of $T\$$, respectively. To ease understanding, each character is marked with its position in T in subscript. Reading these positions in F of BBWT and in F of BWT gives a circular suffix array (there are multiple possibilities with $T_3 = T_4 = $ `abb`) and the suffix array (the position of $\$$ is uniquely defined as $|T\$| = 10$).

▶ **Lemma 1** ([10, Algo. 2.1]). *Given a string T of length n, there is an algorithm that outputs the Lyndon factors $T_1, \ldots, T_t$ one by one in increasing order in $\mathcal{O}(n)$ total time while keeping only a constant number of pointers to positions in T that (a) can move one position forward at one time or (b) can be set to the position of another pointer.*

For what follows, we fix a string $T[1 \mathinner{.\,.} n]$ over an alphabet Σ with size σ. We use the string $T := $ `bacabbabb` as our running example. Its Lyndon factors are $T_1 = $ `b`, $T_2 = $ `ac`, $T_3 = $ `abb`, and $T_4 = $ `abb`.

3.3 Burrows-Wheeler Transforms

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 $T_1, \ldots, T_t$ of T sorted with respect to the $\prec_\omega$ order. A property of BBWT used in this paper as a starting point for an inversion algorithm is the following:

▶ **Lemma 2** ([5, Lemma 15]). BBWT$[1] = T[n]$.

Proof. There is no conjugate of a Lyndon factor that is smaller than the smallest Lyndon factor T_t since $T_t \preceq_{\text{lex}} T_x \prec_{\text{lex}} T_x[j \mathinner{.\,.}]$ for every $j \in [2 \mathinner{.\,.} |T_x|]$ and every $x \in [1 \mathinner{.\,.} t]$. Therefore, T_t is the smallest string among all conjugates of all Lyndon factors. Hence, BBWT$[1]$ is the last character of T_t, which is $T[n]$. ◄

The BWT of T, called in the following BWT, is the BBWT of $\$T$ for a delimiter $\$ \notin \Sigma$ smaller than all other characters in T (cf. [15, Lemma 12] since $\$T$ is a Lyndon word). Originally, the BWT is defined by reading the last characters of all cyclic rotations of T (without $\$$) sorted lexicographically [6]. Here, we call the resulting string BWT°. BWT° is equivalent to BWT if T contains the aforementioned unique delimiter $\$$. We further write BWT$_P$ (and analogously BBWT$_P$ or BWT°$_P$) to denote the BWT of P for a string P.

Since BWT (and analogously BBWT or BWT°) is a permutation of T, it is natural to identify each entry of BWT with a text position: By construction BWT$[i] = T[j]$, where $T[j+1 \mathinner{.\,.}]$ is the i-th lexicographically smallest suffix, i.e., SA$[i] = j+1$, where SA is the suffix array of T. A similar relation is given between BBWT and the circular suffix array [19, 2],

which is uniquely defined up to positions of equal Lyndon factors. Figure 1 gives an example for all three variants. In what follows, we review means to simulate a linear traversal of the text in forward or backward manner by BWT, and then translate this result to BBWT.

3.4 Backward and Forward Steps

Having the location of $T[i]$ in BWT, we can compute $T[i+1]$ (i.e., $T[1]$ for $i = 1$) and $T[i-1]$ (i.e., $T[n]$ for $i = 0$) by rank and select queries. To move from $T[i]$ to $T[i+1]$, which we call a *forward step*, we can use the FL mapping:

$$\mathsf{FL}[i] := \mathsf{BWT}.\mathsf{select}_{\mathsf{F}[i]}(\mathsf{F}.\mathsf{rank}_{\mathsf{F}[i]}(i)), \tag{1}$$

where $\mathsf{F}[i]$ is the i-th lexicographically smallest character in BWT. To move from $T[i]$ to $T[i-1]$, we can use the *backward step* of the FM-index [11], which is also called LF mapping, and is defined as follows:

$$\mathsf{LF}[i] := \mathsf{F}.\mathsf{select}_{\mathsf{BWT}[i]}(\mathsf{BWT}.\mathsf{rank}_{\mathsf{BWT}[i]}(i)) = C[\mathsf{BWT}[i]] + \mathsf{BWT}.\mathsf{rank}_{\mathsf{BWT}[i]}(i), \tag{2}$$

where $C[c]$ is the number of occurrences of those characters in BWT that are smaller than c (for each character $c \in [1 \mathinner{\ldotp\ldotp} \sigma]$). We observe from the second equation of (2) that there is no need for F when having C. This is important, as we can compute $C[i]$ in $\mathcal{O}(n)$ time only having BWT available. Hence, we can compute $\mathsf{LF}[i]$ in $\mathcal{O}(n)$ time in-place. However, the same trick does not work with $\mathsf{FL}[i] = \mathsf{BWT}.\mathsf{select}_{\mathsf{F}[i]}(i - C[\mathsf{F}[i]])$. To lookup $\mathsf{F}[i]$, we can use the selection algorithm of Chan et al. [7] using BWT and $\mathcal{O}(\lg n)$ bits as working space (the algorithm restores BWT after execution) to compute an entry of F in $\mathcal{O}(n)$ time.

In summary, we can compute both $\mathsf{FL}[i]$ and $\mathsf{LF}[i]$ in-place in $\mathcal{O}(n)$ time. The algorithm of Crochemore et al. [9, Thm. 2] inverting BWT in-place in $\mathcal{O}(n^{2+\epsilon})$ time uses the result of Munro and Raman [26] computing $\mathsf{F}[i]$ in $\mathcal{O}(n^{1+\epsilon})$ time for a constant $\epsilon > 0$ in the comparison model. As noted by Chan et al. [7, Sect. 1], the time bound for the inversion can be improved to $\mathcal{O}(n^2)$ time in the RAM model under the assumption that BWT is rewritable.

If we allow more space, it is still advantageous to favor storing C instead of F if $\sigma = o(n)$ because storing F and C in their plain forms take $n \lg \sigma$ bits and $\sigma \lg n$ bits, respectively. To compute $\mathsf{FL}[i]$, we can also compute FL without F by endowing C with a predecessor data structure (which we do in Sect. 4.3).

Finally, we also need LF and FL on BBWT for our conversion algorithms. We can define LF and FL similarly for BBWT with the following peculiarity:

3.5 Steps in the Bijective BWT

The major difference to the BWT is that the LF mapping of the BBWT can contain multiple cycles, meaning that LF (or FL) recursively applied to a BBWT position would result in searching circular (more precisely, the search stays within the same Lyndon factor). This is because BBWT is the extended BWT [23, Thm. 20 and Remark 12] applied to the multiset of Lyndon factors $\{T_1, \ldots, T_t\}$. This fact was exploited for circular pattern matching [19], but is not of interest here.

Instead, we follow the analysis of the so-called *rewindings* [3, Sect. 3]: Remembering that we store the last character of all conjugates of all Lyndon factors in BBWT, we observe that the entries in BBWT representing the Lyndon factors (i.e., the last characters of the Lyndon factors) are in sorted order (starting with $T_t[|T_t|]$ and ending with $T_1[|T_1|]$). That is because the lexicographic order and the $\prec_\omega$ order are the same for Lyndon words [5,

Thm. 8]. Applying the backward step at such an entry results in a rewinding, i.e., we can move from the beginning of a Lyndon factor T_x (represented by $T_x[|T_x|]$ in BBWT) to the end of T_x (represented by $T_x[1]$ in BBWT) with one backward step. We use this property with Lemma 2 in the following sections to read the Lyndon factors from T individually in the order $T_t, \ldots, T_1$.

4   Run-Length Compressed Conversions

We now consider BWT and BBWT represented as run-length compressed strings taking $\mathcal{O}(r_{\mathsf{BWT}} \lg n)$ and $\mathcal{O}(r_{\mathsf{BBWT}} \lg n)$ bits of space, where r_{BWT} and r_{BBWT} are the number of character runs in BWT and BBWT, respectively. For $r := \max(r_{\mathsf{BWT}}, r_{\mathsf{BBWT}})$, the goal of this section is the following:

▶ **Theorem 3.** *We can convert* RLBBWT *to* RLBWT *in* $\mathcal{O}(n \lg r / \lg \lg r)$ *time using* $\mathcal{O}(r \lg n)$ *bits as working space, or vice versa.*

To prove this theorem, we need a data structure that works in the run-length compressed space while supporting rank and select queries as well as updates more efficiently than the $\mathcal{O}(n)$ time in-place approach described in Sects. 3.4 and 3.5:

4.1   Run-length Compressed Wavelet Trees

Given a run-length compressed string S of uncompressed length n with r character runs, there is an $\mathcal{O}(r \lg n)$ bits representation of S that supports access, rank, select, insertions, and deletions in $\mathcal{O}(\lg r)$ time [30, Lemma 1]. It consists of (1) a dynamic wavelet tree maintaining the starting characters of each character run and (2) a dynamic Fenwick tree maintaining the lengths of the runs. It can be accelerated to $\mathcal{O}(\lg r / \lg \lg r)$ time by using the following representations:

1. The dynamic wavelet tree of Navarro and Nekrich [27] on a text of length r uses $\mathcal{O}(r \lg r)$ bits, and supports both updates and queries in $\mathcal{O}(\lg r / \lg \lg r)$ time.
2. The dynamic Fenwick tree of Bille et al [4, Thm. 2] on r ($\lg n$)-bit numbers uses $\mathcal{O}(r \lg n)$ bits, and supports both updates and queries in constant time if updates are restricted to be in-/decremental.

The obtained time complexity of this data structure directly improves the construction of RLBWT:

▶ **Corollary 4** ([30, Thm. 2]). *We can construct the* RLBWT *in* $\mathcal{O}(r_{\mathsf{BWT}} \lg n)$ *bits of space online on the reversed text in* $\mathcal{O}(n \lg r_{\mathsf{BWT}} / \lg \lg r_{\mathsf{BWT}})$ *time.*

In the run-length compressed wavelet tree representation, RLBWT and RLBBWT support an update operation and a backward step in $\mathcal{O}(\lg r / \lg \lg r)$ time with $r := \max(r_{\mathsf{BWT}}, r_{\mathsf{BBWT}})$. This helps us to devise the following two conversions:

4.2   From RLBBWT to RLBWT

We aim for directly outputting the characters of T in reversed order since we can then use the algorithm of Cor. 4 building RLBWT online on the reversed text. We start with the first entry of BBWT (corresponding to the last Lyndon factor T_t, i.e., storing $T_t[|T_t|] = T[n]$ according to Lemma 2) and do a backward step until we come back at this first entry (i.e., we have visited all characters of T_t). During that search, we copy the read characters to RLBWT and mark in an array R of length r_{BBWT} at entry i how often we visited the i-th

character run of RLBBWT. Finally, we remove the read cycle of RLBBWT by decreasing the run lengths of RLBBWT by the numbers stored in R. By doing so, we remove the last Lyndon factor T_t from RLBBWT and consequently know that the currently first entry of BBWT must correspond to T_{t-1}. This means that we can apply the algorithm recursively on the remaining RLBBWT to extract and delete the Lyndon factors in reversed order while building RLBWT in the meantime. By removing T_t, BBWT is still a valid BBWT since BBWT becomes the BBWT of $T' := T_1 \cdots T_{t-1}$ whose Lyndon factors are the same as of T (but without T_t). Note that it is also possible to build RLBWT in forward order, i.e., computing $\mathsf{RLBWT}_{T_1 \cdots T_x}$ for increasing x by applying the algorithm of Mantaci et al. [24, Fig. 1] while omitting the suffix array construction.

4.3   From RLBWT to RLBBWT

To build BBWT, we need to be aware of the Lyndon factors of T, which we compute with Lemma 1 by simulating a forward scan on T with FL on BWT. To this end, we store the entries of the C array in a Fusion tree [13] using $\mathcal{O}(\sigma \lg n)$ bits and supporting predecessor search in $\mathcal{O}(\lg \sigma / \lg \lg \sigma) = \mathcal{O}(\lg r / \lg \lg r)$ time.[2] This time complexity also covers a forward step in RLBWT by simulating F with the Fusion tree on C. Hence, this fusion tree allows us to apply Lemma 1 computing the Lyndon factorization of T with a multiplicative $\mathcal{O}(\lg r / \lg \lg r)$ time penalty since this algorithm only needs to perform forward traversals. The starting point of such a traversal is the position i with $\mathsf{BWT}[i] = \$$ because $\mathsf{FL}[i]$ returns the first character of T. Whenever we detect a Lyndon factor T_x (starting with $x = 1$), we copy this factor to our dynamic RLBBWT. For that, we always maintain the first and the last position of T_x in memory. Having the last position of T_x, we perform backward steps on RLBWT until returning at the first position of T_x to read the characters of T_x in reversed order. Then we continue with the algorithm of Lemma 1 at the position after T_x (for recursing on T_{x+1}). Inserting a Lyndon factor into RLBBWT works exactly as sketched by Bonomo et al. [5, Thm. 17] (we review this algorithm in detail in Sect. 5.1).

5   In-Place Conversions

We finally present our in-place conversions that work in quadratic time by computing LF or FL in $\mathcal{O}(n)$ time having only stored either BWT, BBWT, or BWT$^\circ$. We note that the constructions from the text also work in the comparison model, while inverting a transform or converting two different transforms have a multiplicative $\mathcal{O}(n^\epsilon)$ time penalty as the fastest option to access F in the comparison model uses $\mathcal{O}(n^{1+\epsilon})$ time for a constant $\epsilon > 0$ [26]. We start with the construction and inversion of BWT$^\circ$ (Sects. 5.1 and 5.2), where we show (a) that we can construct BWT$^\circ$ from the text in the same manner as Bonomo et al. [5] construct BBWT, and (b) that the latter construction works also in-place. Next, we show in Sect. 5.3 how to invert BBWT with the BWT inversion algorithm of Crochemore et al. [9, Fig. 3], which allows us to also convert BBWT to BWT with the BWT construction algorithm of the same paper [9, Fig. 2]. Finally, we show a conversion from BWT to BBWT in Sect. 5.4. An overview is given in Table 1.

---

[2] We assume that the alphabet Σ is *effective*, i.e., that each character of Σ appears at least once in T. Otherwise, assume that T uses σ' characters. Then we build the static dictionary of Hagerup [18] in $\mathcal{O}(\sigma' \lg \sigma')$ time, supporting access to a character in $\mathcal{O}(\lg \lg \sigma') = \mathcal{O}(\lg \lg r)$ time, assigning each of the σ' characters an integer from $[1 .. \sigma']$. We further map RLBWT to the alphabet $[1 .. \sigma']$, which can be done in $\mathcal{O}(r)$ time by using $\mathcal{O}(r \lg n)$ space for a linear-time integer sorting algorithm.

■ **Table 1** Overview of in-place conversions in focus of Sect. 5 working in quadratic time.

To \ From	T	BWT	BBWT	BWT°
T	\	[9, Fig. 3]	Sect. 5.3	Sect. 5.2
BWT	[9, Fig. 2]	\	Sect. 5.3	
BBWT	Sect. 5.1	Sect. 5.4	\	
BWT°	Sect. 5.1			\

5.1 Constructing BWT° and BBWT

We can compute BWT° and BBWT from T with the algorithm of Bonomo et al. [5] computing the extended BWT [23]. The extended BWT is the BWT defined on a set of primitive strings. As stated in Sect. 3.5, the extended BWT coincides with BBWT if this set of primitive strings is the set of Lyndon factors of T [5, Thm. 14]. We briefly describe the algorithm of Bonomo et al. [5] for computing the BBWT (cf. Fig. 2): For each Lyndon factor T_x (starting with $x = 1$), prepend $T_x[|T_x|]$ to BBWT. To insert the remaining characters of the factor T_x, let $p \leftarrow 1$ be the position of the currently inserted character. Then perform, for each $j = |T_x| - 1$ down to 1, a backward step $p \leftarrow \mathsf{LF}[p] + 1$, and insert $T_x[j]$ at BBWT$[p]$. To understand why this computes BBWT, we observe that the last character of the most recently inserted Lyndon factor T_x is always the first character in BBWT$_{T_1 \cdots T_x}$ according to Lemma 2. By recursively inserting the preceding character at the place returned by a backward step, we precisely insert this character at the position where we would expect it (another backward step from the same position p would then return the inserted character). Using only n backward steps and n insertions, this algorithm works in-place in $\mathcal{O}(n^2)$ time by simulating LF as described in Sect. 3.4.

Consequently, we can build BWT° if T is a Lyndon word since in this case BWT° and BBWT coincide [15, Lemma 12]. That is because sorting the suffixes of T is equivalent to sorting the conjugates of T (if T is a Lyndon word, then its Lyndon factorization consists only of T itself).

It is easy to generalize this to work for a general string T. First, if T is primitive, then we compute its so-called *Lyndon conjugate*, i.e., a conjugate of T that is a Lyndon word. (The Lyndon conjugate of T is uniquely defined if T is primitive.) We can find the Lyndon conjugate of T in $\mathcal{O}(n)$ time with the following two lemmata:
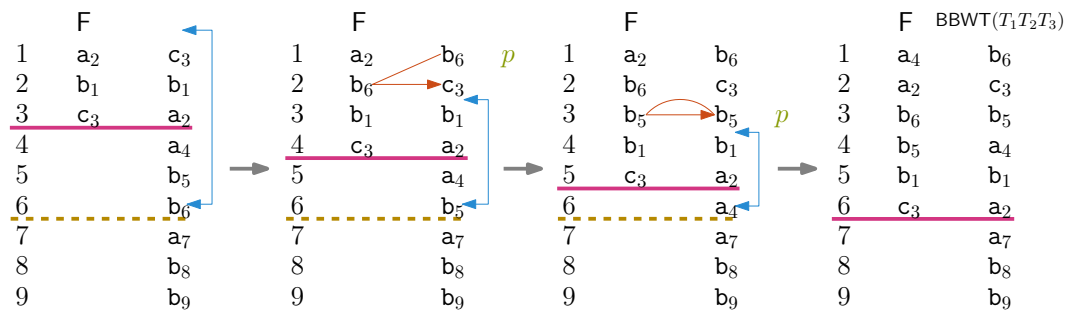
▶ **Lemma 5** ([10, Prop. 1.3]). *Given two Lyndon words S and T, ST is a Lyndon word if $S \prec_{\mathrm{lex}} T$.*

▶ **Lemma 6.** *Given a primitive string T, we can find its Lyndon conjugate in $\mathcal{O}(n)$ time with $\mathcal{O}(\lg n)$ bits of space.*

Proof. We use Lemma 1 to detect the last Lyndon factor T_t of the Lyndon factorization $T_1 \cdots T_t$ of T with $\mathcal{O}(\lg n)$ bits of working space. According to Lemma 5, $T_t T_1$ is a Lyndon word since $T_t \prec_{\mathrm{lex}} T_1$, and so is $T_t T_1 \cdots T_{t-1}$ a Lyndon word by a recursive argument. Hence, we have found T's Lyndon conjugate.   ◀

Let $\mathrm{conj}_j(T)$ be the Lyndon conjugate of T for $j \in [0 \mathbin{..} n - 1]$. Since BWT° is identical to BBWT$_{\mathrm{conj}_j(T)}$, we are done by running the algorithm of Bonomo et al. [5] on $\mathrm{conj}_j(T)$. Finally, if T is not primitive, then there is a primitive string P such that $T = P^k$ for an integer $k \geq 2$. We can compute BWT°$_P$ with the above considerations. For obtaining BWT°,

	F				F					F					F	BBWT$(T_1T_2T_3)$
1	a_2	c_3		1	a_2	b_6	p		1	a_2	b_6			1	a_4	b_6
2	b_1	b_1		2	b_6	c_3			2	b_6	c_3			2	a_2	c_3
3	c_3	a_2		3	b_1	b_1			3	b_5	b_5	p		3	b_6	b_5
4		a_4		4	c_3	a_2			4	b_1	b_1			4	b_5	a_4
5		b_5		5		a_4			5	c_3	a_2			5	b_1	b_1
6		b_6		6		b_5			6		a_4			6	c_3	a_2
7		a_7		7		a_7			7		a_7			7		a_7
8		b_8		8		b_8			8		b_8			8		b_8
9		b_9		9		b_9			9		b_9			9		b_9

Figure 2 Computing BBWT from our running example $T = $ `bacabbabb` in four steps (visualized by four columns separated by three arrows ➡), cf. Sect. 5.1. In each column, the characters from the top to the solid horizontal line (—) form the currently built BBWT. The characters below that up to the dashed horizontal line (- -) are under consideration of being merged into BBWT. This dashed line is always before the beginning of the next yet unread Lyndon factor. *First column*: We have already computed the BBWT of $T_1 T_2 = $ `bac`, which is `cba`. In the following we want to add the next Lyndon factor $T_3 = $ `abb` to it. For that, we prepend its last character to the currently constructed BBWT. *Second column*: We move the last character above the dashed line to the position $\mathsf{LF}[p] + 1$ with $p = 1$, and update $p \leftarrow \mathsf{LF}[p] + 1$. We recurse in the *third column*, and have produced the BBWT of $T_1 T_2 T_3 = $ `bacabb` in the *forth column*.

according to [25, Prop. 2], we only need to make each character in BWT_P° to a character run of length k, i.e., if $\mathsf{BWT}_P^\circ[i] = \mathsf{c}$, we append c^k to BWT° for increasing $i \in [1 .. |P|]$ (cf. [15, Thm. 13]). Checking whether T is primitive can be done in $\mathcal{O}(n^2)$ time by checking for each pair of positions their longest common prefix.

5.2 Inverting BWT°

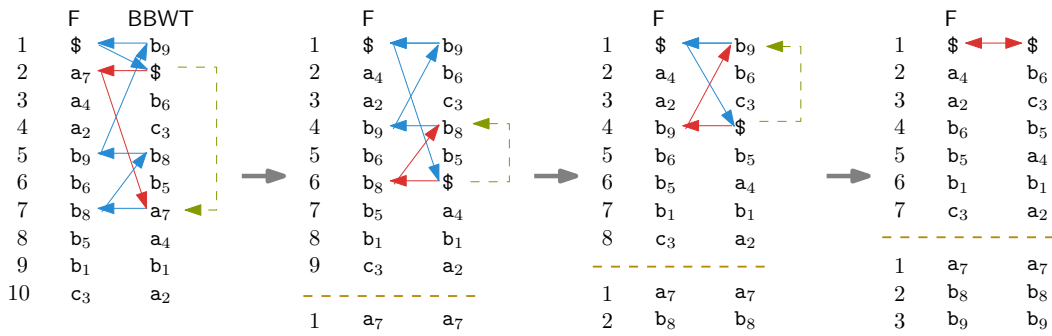To invert BWT°, we use the techniques of Crochemore et al. [9, Fig. 3] inverting BWT in-place in $\mathcal{O}(n^2)$ time. An invariant is that the BWT entry, whose FL mapping corresponds to the next character to output, is marked with a unique delimiter \$. Given that $\mathsf{BWT}[i] = \$$, the algorithm outputs $\mathsf{BWT}[\mathsf{FL}[i]]$, sets $\mathsf{BWT}[\mathsf{FL}[i]] \leftarrow \$$, removes $\mathsf{BWT}[i]$, and recurses until \$ is the last character remaining in BWT. By doing so, it restores the text in text order.

To adapt this algorithm for inverting BWT°, we additionally need a pointer p storing the first symbol of the text (since there is no unique delimiter such as \$ in general). Given that p points to $\mathsf{BWT}^\circ[i]$, we set $i \leftarrow \mathsf{FL}[i]$ and subsequently output $\mathsf{BWT}^\circ[i]$. From now on, the algorithm works exactly as [9, Fig. 3] if we set $\mathsf{BWT}^\circ[i] \leftarrow \$$ after outputting $\mathsf{BWT}^\circ[i]$. More involving is inverting BBWT or converting BBWT to BWT, which we tackle next.

5.3 Inverting BBWT

Similarly to Sect. 4.2, we read the Lyndon factors from BBWT in the order $T_t, \ldots, T_1$, and move each read Lyndon factor directly to a text buffer such that while reading the last Lyndon factor T_x for an $x \in [1 .. t]$ from $\mathsf{BBWT}_{T_1 \cdots T_x}$, we move the characters of T_x to $T_{x+1} \cdots T_t$, producing $\mathsf{BBWT}_{T_1 \cdots T_{x-1}}$ and $T_x \cdots T_t$. This allows us to recurse by reading always the last Lyndon factor T_x stored in $\mathsf{BBWT}_{T_1 \cdots T_x}$.

Here, we want to apply the inversion algorithm for BWT° described in Sect. 5.3. For adapting this algorithm to work with BBWT, it suffices to insert \$ at BBWT[2] (cf. Fig. 3). By doing so, we add \$ to the cycle of the currently last Lyndon factor T_x stored in BBWT, i.e., we enlarge the Lyndon factor T_x to $\$T_x$. That is because (a) BBWT[1] corresponds to

◼ Figure 3 Inverting BBWT of our running example $T = \texttt{bacabbabb}$ (cf. Sect. 5.3). *First Column*: We prepend the $\$$ delimiter to the last Lyndon factor T_t by inserting $\$$ at BBWT[2]. A forward step symbolized by the dashed arrow ($\dashrightarrow$) leads us from $\$$ to the first character of T_t. *Second Column*: We output $\mathsf{BBWT}[6] = T_t[1] = T[7]$, remove $\$$ and update $\mathsf{BBWT}[6] \leftarrow \$$. The output is appended to the string shown below the dashed horizontal line ($\dashleftarrow$). We continue with a forward step to access $\mathsf{BBWT}[4] = T_t[2] = T[8]$, and recurse in the *third column*. *Forth Column*: Since a forward step returns $\$$, we know that we have successfully extracted $T_t = \texttt{abb}$.
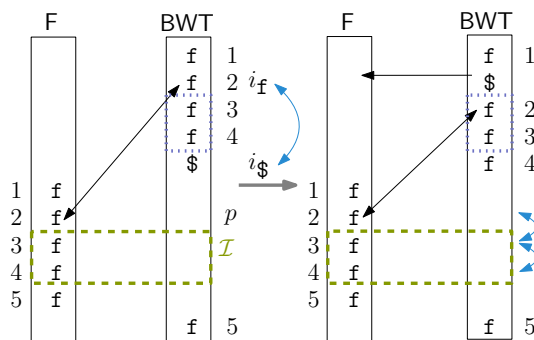
the last character $T_x[|T_x|]$ of T_x (cf. Lemma 2), and after inserting $\$$, $\mathsf{F}[1] = \$, \mathsf{F}[2] = T_x[1]$, hence $\mathsf{FL}[1] = 2$ (a forward step on the last character of T_x gives $\$$) and $\mathsf{FL}[2]$ gives the position in BBWT corresponding to $T_x[1]$. Moreover, inserting $\$$ makes BBWT the BBWT of $T' := T_1 \cdots T_{x_1} \$ T_x$, where $\$ T_x$ is the last Lyndon factor of T'. We now use the property that $\$ T_x[i \mathbin{.\,.} |T_x|]$ is a Lyndon word for each $i \in [1 \mathbin{.\,.} |T_x|]$, allowing us to perform the inversion steps of Crochemore et al. [9, Fig. 3] on BBWT. By doing so, we can remove the entry of BBWT corresponding to $\mathbf{conj}_j(T_x)$ for increasing $j \in [0 \mathbin{.\,.} |T_x| - 1]$ and prepend the extracted characters to the text buffer storing $T_{x+1} \cdots T_t$ within our working space while keeping BBWT a valid BBWT.

Instead of inverting BBWT, we can convert BBWT to BWT in-place by running the in-place BWT construction algorithm of Crochemore et al. [9, Fig. 2] on the text buffer after the extraction of each Lyndon factor. Unfortunately, this works not character-wise, but needs a Lyndon factor to be fully extracted before inserting its characters into BWT. Interestingly, for the other direction (from BWT to BBWT), we can propose a different kind of conversion that works directly on BWT without decoding it.

5.4 From BWT to BBWT on the Fly

Like in Sect. 4.3, we process the Lyndon factors of T individually to compute BBWT by scanning BWT in text order to simulate Lemma 1. Suppose that we have built BWT on $T\$ \neq \$$ with $\$$ being the $(t+1)$-th Lyndon factor of $T\$$, and suppose that we have detected the first Lyndon factor T_1. Let $\mathtt{f}$ denote the last character of T_1. Further let $i_{\mathtt{f}}$ and $i_{\$}$ be the position of the last character of T_1 and the last character of T, respectively, such that $\mathsf{BWT}[i_{\mathtt{f}}] = \mathtt{f}$ and $\mathsf{BWT}[i_{\$}] = \$$. Let $p := \mathsf{LF}[i_{\mathtt{f}}]$ such that $\mathsf{F}[p] = \mathtt{f}$ and $\mathsf{BWT}[p] = T_1[|T_1| - 1]$ if $|T_1| > 1$ or $\mathsf{BWT}[p] = \$$ otherwise. Since T_1 and T_2 are Lyndon factors, $T_1 \succeq_{\mathrm{lex}} T_2$. Consequently, the suffix $T[\mathbf{b}(T_2) \mathbin{.\,.}]$ (the context of $\mathsf{BWT}[i_{\mathtt{f}}]$) is lexicographically smaller than the suffix $T[\mathbf{b}(T_1) \mathbin{.\,.}]$ (the context of $\mathsf{BWT}[i_{\$}]$), i.e., $i_{\mathtt{f}} < i_{\$}$. Figure 4 gives an overview of the introduced setting.

Our aim is to change BWT such that a forward or backward step within the characters belonging to T_1 always results in a cycle. Informally, we want to cut T_1 out of BWT, which additionally allows us to recursively continue with the FL mapping to find the end of the

■ **Figure 4** Setting of Sect. 5.4 with focus on forming a cycle for a Lyndon factor ending with f in BWT. *Left*: We exchange $\mathsf{BWT}[i_f]$ with $\mathsf{BWT}[i_\$]$ with the aim to form a cycle. *Right*: To obtain this cycle we additionally need to swap $\mathsf{BWT}[p]$ with the elements of the dashed rectangle ($--$) corresponding to the interval $\mathcal{I}$ having the same height as the dotted rectangle ($\cdots$) covering $\mathsf{BWT}[i_f + 1 .. i_\$ - 1]$.

next Lyndon factor T_2. For that, we exchange $\mathsf{BWT}[i_\$]$ with $\mathsf{BWT}[i_f]$ (cf. Fig. 5). Then the character $T[\mathsf{e}(T_1) + 1]$ (i.e., the first character of T_2) becomes the next character of $\$$ in terms of the forward step ($\mathsf{BWT}[\mathsf{FL}[i_f]] = T[\mathsf{b}(T_2)]$), while a backwards search on the first character of T_1 yields T_1's last character (LF returns $i_\$$, but now $\mathsf{BWT}[i_\$] = T_1[|T_1|] = \mathtt{f}$). This is sufficient as long as $\mathsf{BWT}[i] \neq \mathtt{f}$ for every $i \in (i_f .. i_\$]$. Otherwise, it can happen that we change the mapping from the i-th f of F to the i-th f of BWT (or vice versa) unintentionally. In such a case, we swap some entries in BWT within the f interval of F. In detail, we conduct the exchange ($\mathsf{BWT}[i_\$]$ with $\mathsf{BWT}[i_f]$), but continue with swapping $\mathsf{BWT}[i]$ and $\mathsf{BWT}[i + 1]$ unless $\mathsf{BWT}[\mathsf{FL}[i]]$ becomes that f that corresponds to $T_1[|T_1|]$ for increasing i starting with $i = p$ until $\mathsf{F}[i] \neq \mathtt{f}$ or $\mathsf{LF}[i] \notin [i_f .. i_\$]$. This may not be sufficient if the characters we swap are identical (cf. Fig. 6). In such a case, we recurse on the $T_1[|T_1| - 1]$ interval of F.

Instead of checking whether we have created a cycle after each swap, we want to compute the exact number of swaps needed for this task. For that we note that exchanging $\mathsf{BWT}[i_\$]$ with $\mathsf{BWT}[i_f]$ decrements the values of $\mathsf{BWT.rank}_f(j)$ for every $j \in [i_f .. i_\$]$ by one. In particular, $\mathsf{BWT.select}_f$ changes for those f's in BWT that are between i_f and $i_\$$. Hence, the number of swaps m is the number of positions $k \in [i_f + 1 .. i_\$ - 1]$ with $\mathsf{BWT}[k] = \mathtt{f}$. The swaps are performed within the range $\mathcal{I}$ starting with $p + 1$ and covering all positions i with $\mathsf{LF}[i] \in [i_i .. i_\$]$ and $\mathsf{F}[i] = \mathtt{f}$ since $\mathcal{I}$ covers all entries whose mapping has changed. However, if $\mathsf{BWT}[p..]$ starts with a character run of $T[\mathsf{e}(T_1) - 1]$ (or of $T[\mathsf{b}(T_1)]$ if $|T_1| = 1$)[3], swapping the identical characters does not change BWT, and therefore has no effect of changing LF. Instead, we search the end of this run within $\mathcal{I}$ to swap the first entry i below this run with the first entry of this run, and recurse on swapping entry i with entries below of it.

Correctness. To see why the swaps restore the LF mapping for T_1 and the remaining part of the text $T_2 \cdots T_t$, we examine those substrings of T that we might no longer find with the LF mapping after exchanging $\mathsf{BWT}[i_\$]$ with $\mathsf{BWT}[i_f]$.

In detail, we examine each substring $S_j := x_j y_j \mathtt{f} \in \Sigma^3$ with $j \in [1 .. m]$ that is represented in BWT (before changing it) with $\mathsf{BWT}[p + j] = y_j$, $\mathsf{BWT}[\mathsf{LF}[p + j]] = x_j$, $\mathsf{BWT}[\mathsf{FL}[p + j]] = \mathtt{f}$, and $i_j := \mathsf{FL}[p + j] \in [i_f + 1 .. i_\$ - 1]$. Due to the LF-mapping, $\mathsf{BWT.select}_f(\mathsf{BWT.rank}_f(i_f) + j) =$

---

[3] For $|T_1| = 1$, $p = i_\$$, and hence, $\mathsf{BWT}[p]$ was $\$$ but now is $\mathtt{f} = T_1[|T_1|] = T_1[1]$.
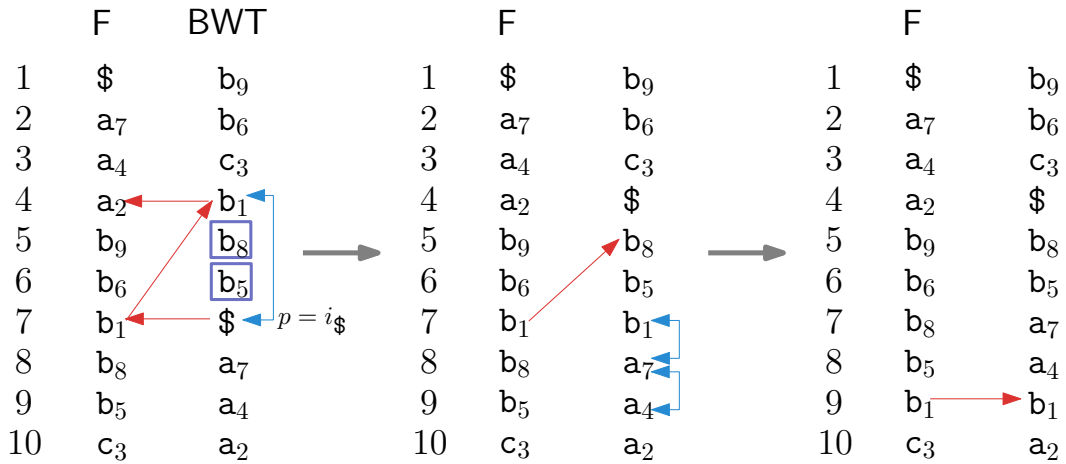
Figure 5 Computing BBWT from BWT (cf. Sect. 5.4) of our running example $T = \texttt{bacabbabb\$}$. In the *left* column, we find the first Lyndon factor $T_1 = \texttt{b}$ of T by forward steps with FL. Since $|T_1| = 1$, $p = i_\$$. We obtain the *middle* column by exchanging $\mathsf{BWT}[4]$ with $\mathsf{BWT}[7] = \$$. Since there are two $\texttt{b}$'s between $\texttt{b}$ at $\mathsf{BWT}[4]$ and $\$$ in the *left* column, we need to swap $\mathsf{BWT}[p]$ with the two elements below of it in the *middle* column. This gives a cycle in the *right* column. We can recurse since the FL mapping of $\$$ now yields the second character of T.

i_j, meaning that $\mathsf{BWT}[i_j]$ is the j-th $\texttt{f}$ in $\mathsf{BBWT}[i_\texttt{f} + 1 \mathinner{.\,.} i_\$ - 1]$, which stores m $\texttt{f}$'s. After exchanging $\mathsf{BWT}[i_\$]$ with $\mathsf{BWT}[i_\texttt{f}]$, $\mathsf{FL}[p + j]$ becomes i_{j+1} for $j \in [0 \mathinner{.\,.} m]$ with $i_{m+1} := i_\$$. However, for all $i > p + m$, $\mathsf{FL}[i]$ did not change. Hence, we only have to focus on the range $\mathcal{I} = [p + 1 \mathinner{.\,.} p + m]$.

First, suppose that $y_1 = \mathsf{BWT}[p + 1] \neq \mathsf{BWT}[p]$. If we swap $\mathsf{BWT}[p]$ with $\mathsf{BWT}[p + 1]$, then $\mathsf{LF}[p]$ is still i_1, but $\mathsf{BWT}[\mathsf{LF}[p]]$ becomes x_1 such that we have fixed the substring $x_1 y_1 \texttt{f}$. This also works in a more general setting: If $y_j = \mathsf{BWT}[p+j] \neq \mathsf{BWT}[p]$ for every $j \in [1 \mathinner{.\,.} m]$, we can perform m swaps like above for all m entries in $\mathsf{BWT}[\mathcal{I}]$ to fix all substrings S_j.

Now suppose that $y_j = \mathsf{BWT}[p + j] = \mathsf{BWT}[p]$ for $j \in [1 \mathinner{.\,.} \ell]$ with the largest possible $\ell \in [2 \mathinner{.\,.} m]$. Let $k > p + \ell$ be the first entry with $\mathsf{BWT}[k] \neq \mathsf{BWT}[p]$. First, suppose that $k \in \mathcal{I}$. Then $\mathsf{F}[k] = \texttt{f}$, and swapping $\mathsf{BWT}[k]$ with $\mathsf{BWT}[p]$ restores the LF mapping for the substrings S_j with $j \in [1 \mathinner{.\,.} \ell]$ since this swap decrements $\mathsf{BWT}.\mathrm{rank}_{y_j}[p + j]$ by one for every $j \in [1 \mathinner{.\,.} \ell]$. We recurse on swapping $\mathsf{BWT}[k]$ with the following BBWT entries in $\mathcal{I}$ until all m substrings got restored. Finally, if $k \geq p + m$, then all y_i are equal such that we can find the x_i in BWT consecutively stored at positions with an F value of y_i. Thus, we can apply the swaps there recursively.

Time Complexity. Fixing a Lyndon factor T_x, we spend $\mathcal{O}(|\mathcal{I}|)$ time for the swaps in $\mathsf{BWT}[\mathcal{I}]$, and perform the swaps recursively at most $|T_x|$ times, where we need additionally $\mathcal{O}(n)$ time per recursion step for computing $\mathsf{LF}[p]$, summing up to $\mathcal{O}(|T_x|(|\mathcal{I}|+n)) = \mathcal{O}(n|T_x|)$ time. Since $\sum_{x=1}^{t} |T_x| = n$, we yield $\mathcal{O}(n^2)$ total time.

6   Open Problems

Our algorithm of Sect. 5.3 converts BBWT to BWT, Lyndon factor by Lyndon factor. It would be interesting to find another conversion that works character-wise. Here, our inversion algorithm extracts a Lyndon factor in text order from BBWT, while the used BWT construction algorithm parses the text in reverse text order.
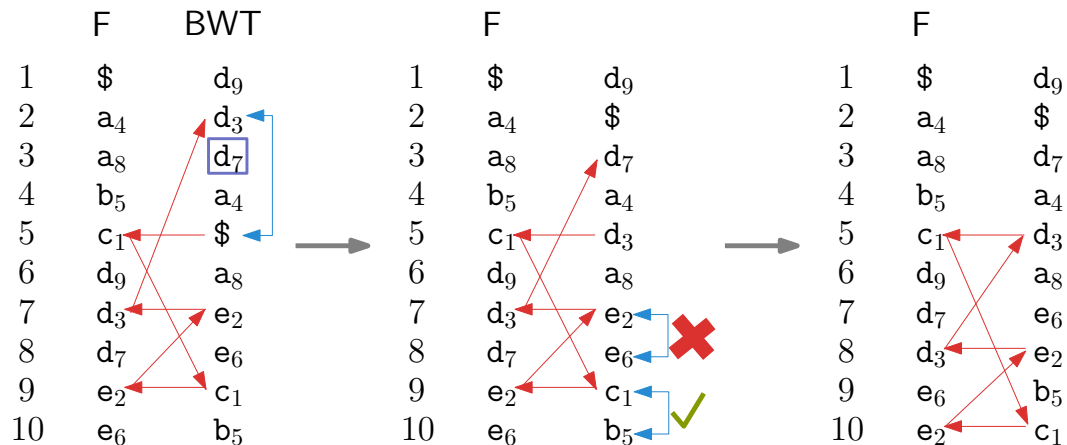
Figure 6 Special case for computing BWT from BBWT (cf. Sect. 5.4) with the different example string $T\$:= \mathtt{cedabedad\$}$ having $T_1 = \mathtt{ced}$ as its first Lyndon factor. *Left column*: We find the first Lyndon factor $T_1 = \mathtt{ced}$ of T by forward steps with FL. Its last character is stored at BWT[2]. By exchanging $\$$ with the last character of T_1 in BWT, we obtain the middle column. *Middle column*: The LF mapping for the third d in F becomes invalid. However, there is only a character run of $T_1[|T_1| - 1] = \mathtt{e}$ in BWT of the $T_1[|T_1|] = \mathtt{d}$ interval $[7 . . 8]$ in F starting with $p = 7$. So we recurse on LF[p] to find characters different from $T_1[|T_1| - 2] = \mathtt{c}$ to swap in the respective $T_1[|T_1| - 1] = \mathtt{e}$ interval $[9 . . 10]$. *Right Column*: We have created a cycle with the characters of the first Lyndon factor. A forward step at $\$$ gives the first character of the next Lyndon factor.

Crochemore et al. [9, Sect. 4] proposed a space and time trade-off algorithm based on their in-place techniques computing or inverting BWT. We are positive that it should be possible to adapt their techniques for computing or inverting BBWT or BWT° with a trade-off parameter.

From the combinatorial perspective, we question whether the number of distinct Lyndon words of T is bounded by the runs in BBWT. If we can affirm this question, it would be possible to adapt the BBWT based index data structure [3] for RLBBWT using $\mathcal{O}(r_{\mathsf{BBWT}} \lg n)$ bits of space because this solution needs a bit vector with rank and select support marking the positions in BBWT corresponding to the distinct Lyndon factors. If this number is at most the number of runs r_{BBWT}, then we can store this bit vector entropy-compressed in $\mathcal{O}(r \lg n)$ bits when $r_{\mathsf{BBWT}} = o(n)$ since $nH_0(r) = n \lg(n/(n-r)) + r \lg((n-r)/r) \le n \lg r \Leftrightarrow r \lg((n-r)/r) \le n \lg(r(n-r)/n)$ for $r = r_{\mathsf{BBWT}}$.

Speaking of RLBBWT, we wonder whether we can construct RLBBWT online in run-length compressed space similar to Cor. 4. With the run-length compressed wavelet tree, the algorithm of Bonomo et al. [5, Thm. 17] works in $\mathcal{O}(n \lg r_{\mathsf{BBWT}} / \lg \lg r_{\mathsf{BBWT}})$ time with $\max_{x \in [1 . . t]} |T_x| + \mathcal{O}(r_{\mathsf{BBWT}} \lg n)$ bits of space by reading each Lyndon factor of the text individually.

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. Constructing the bijective BWT. *ArXiv 1911.06985*, 2019.

3   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.

4   Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj, and Søren Vind. Dynamic relative compression, dynamic partial sums, and substring concatenation. *Algorithmica*, 80(11):3207–3224, 2018.

5   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.

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

7   Timothy M. Chan, J. Ian Munro, and Venkatesh Raman. Selection and sorting in the "restore" model. *ACM Trans. Algorithms*, 14(2):11:1–11:18, 2018.

8   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.

9   Maxime Crochemore, Roberto Grossi, Juha Kärkkäinen, and Gad M. Landau. Computing the Burrows-Wheeler transform in place and in small space. *J. Discrete Algorithms*, 32:44–52, 2015.

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

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

12   Paolo Ferragina, Giovanni Manzini, and S. Muthu Muthukrishnan. *The Burrows-Wheeler Transform: Ten Years Later*. DIMACS, 2004.

13   Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

14   Travis Gagie, Giovanni Manzini, Gonzalo Navarro, and Jens Stoye. 25 Years of the Burrows-Wheeler Transform (Dagstuhl Seminar 19241). *Dagstuhl Reports*, 9(6):55–68, 2019.

15   Raffaele Giancarlo, Antonio Restivo, and Marinella Sciortino. From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization. *Theor. Comput. Sci.*, 387(3):236–248, 2007.

16   Joseph Yossi Gil and David Allen Scott. A bijective string sorting transform. *ArXiv 1201.3077*, 2012.

17   Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.

18   Torben Hagerup. Fast deterministic construction of static dictionaries. In *Proc. SODA*, pages 414–418, 1999.

19   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.

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

21   Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.*, 12(1):40–66, 2005.

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

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

24   Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Suffix array and Lyndon factorization of a text. *J. Discrete Algorithms*, 28:2–8, 2014.

25   Sabrina Mantaci, Antonio Restivo, and Marinella Sciortino. Burrows-Wheeler transform and Sturmian words. *Inf. Process. Lett.*, 86(5):241–246, 2003.

26   J. Ian Munro and Venkatesh Raman. Selection from read-only memory and sorting with minimum data movement. *Theor. Comput. Sci.*, 165(2):311–323, 1996.

27   Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. *SIAM J. Comput.*, 43(5):1781–1806, 2014.

28   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.

29   Tatsuya Ohno, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A faster implementation of online run-length Burrows-Wheeler transform. In *Proc. IWOCA*, volume 10765 of *LNCS*, pages 409–419, 2017.

30   Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80(7):1986–2011, 2018.