In-Place Bijective Burrows Wheeler Transformations

Dominik Köppl
Department of Informatics, Kyushu University
Japan Society for Promotion of Science (JSPS)
dominik.koeppl@inf.kyushu-u.ac.jp

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

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

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

Abstract

One of the most well-known variants of the Burrows-Wheeler transform (BWT) is the bijective BWT (BBWT), which does not need to store the artificial dollar sign or any information regarding the location of the end of the text in the transformed string. In this paper, we present algorithms constructing the BBWT or restoring the text from 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.

2012 ACM Subject Classification Theory of computation; Mathematics of computing → Combinatorics on words

Keywords and phrases In-Place Algorithms, Burrows-Wheeler transform, Lyndon words

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding *Dominik Köppl*: JSPS KAKENHI Grant Number JP18F18120.

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 [20], 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 by Policriti and Prezza [27]. The BWT and its run-length compressed representation have been intensively studied during the past decades (e.g., [1, 12, 14] and the references therein). Contrary to that, a variant, called the *bijective* BWT (BBWT) [15], is far from being well-studied despite its mathematically appealing characteristics. As a matter of fact, we are only aware of one index data structure based on the BBWT [2] and of two non-trivial construction algorithms [3, 5] of the (uncompressed) BBWT, both with the need of additional data structures. This gives rise to the following questions:

- Is there a connection between the BBWT and the BWT?
- Can we compute the run-length compressed BBWT in run-length compressed space directly?

For the former question, we shed more light on the connection between the BWT and the BBWT by quadratic time in-place conversion algorithms in Sect. 6 constructing the BWT from the BBWT, or vice versa. We can also perform these conversions in the run-length compressed setting in space linear to the number of the character runs (cf. Sect. 5), solving the latter question.

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 [21] 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 [16] 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 [27] 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. [26].

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 all Lyndon factors sorted with respect to the $\prec_\omega$ order. Bannai et al. [3] recently revealed an indirect connection between the bijective BWT and suffix sorting by presenting an $\mathcal{O}(n)$ time BBWT construction algorithm based on SAIS [25]. With dynamic data structures like a dynamic wavelet tree [24], Bonomo et al. [5]

could give an algorithm computing the BBWT in $\mathcal{O}(n \lg n / \lg\lg n)$ time. Both construction algorithms need however data structures taking $\mathcal{O}(n \lg n)$ bits of space. However, the latter can work in-place by simulating the LF mapping on the BBWT (cf. Sect. 4), which we focus on in Sect. 6.1.

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 $[\mathsf{b}(I)..\mathsf{e}(I)] = I$ 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 with $1 \le j \le |T|$, 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 $k \ge 2$.

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 $1 \le i \le j \le |T|$, 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 called the *j-th suffix* of T, and denoted with $T[j..]$. An occurrence of a substring S in T is treated as a sub-interval of $[1..|T|]$ such that $S = T[\mathsf{b}(S)..\mathsf{e}(S)]$. 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 $\prec_{\text{lex}}$. Given two string S and T, then $S \prec_{\text{lex}} T$ if S is a prefix of T or there exists an integer ℓ with $1 \le \ell \le \min(|S|, |T|)$ such that $S[1..\ell-1] = T[1..\ell-1]$ and $S[\ell] < T[\ell]$. Next we define the *$\prec_\omega$ order*: We write $S \prec_\omega T$ if the infinite concatenation $S^\omega := SSS\cdots$ is lexicographically smaller than $T^\omega := TTT\cdots$. For instance, $\mathtt{ab} \prec_{\text{lex}} \mathtt{aba}$ but $\mathtt{aba} \prec_\omega \mathtt{ab}$.

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

3.2 Lyndon Words

Given a string $T = T[1..n]$, its i-th *conjugate* $\mathsf{conj}_i(T)$ is defined as $T[i+1..n]T[1..i]$ for an integer i with $0 \le i \le n-1$. We say that T and all of its conjugates belong to the *conjugate class* $\mathsf{conj}(T) := \{\mathsf{conj}_0(T), \ldots, \mathsf{conj}_{n-1}(T)\}$. If a conjugate class contains *exactly* one conjugate that is lexicographically smaller than all other conjugates, then this conjugate is called a *Lyndon word* [19]. Equivalently, a string T is said to be a Lyndon word if and only if $T \prec_{\text{lex}} S$ for every proper suffix S of T.

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, and (b) $T_x \succeq_{\text{lex}} T_{x+1}$ for each $x \in [1..t)$. Each Lyndon word T_x is called a *Lyndon factor*.

▶ **Lemma 1** ([10, Algo. 2.1]). *The Lyndon factorization of T can be computed in $\mathcal{O}(n)$ such that the Lyndon factors $T_1, \ldots, T_t$ are output one by one in increasing order. For that it suffices to scan the text linearly from left to right while keeping a constant number of pointers to positions in the text that can move one position forward at one time.*

For what follows, we fix a string $T[1..n]$ 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 $\mathsf{BBWT}[i]$ 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 decoding is the following:

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

Proof. There is no conjugate of a Lyndon factor that is smaller than the smallest Lyndon factor T_t since $T_t \preceq_{\text{lex}} T_x \prec_{\text{lex}} T_x[j..]$ for every $j \in [2..|T_x|]$ and every $x \in [1..t]$. Therefore, T_t is the smallest string among all conjugates of all Lyndon factors. Hence, $\mathsf{BBWT}[i]$ 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. Originally, the BWT is defined as reading the last characters of all cyclic rotations of T (without $\$$) sorted lexicographically. Here, we call the resulting string BWT°. BWT° is equivalent to BWT if T contains a unique delimiter $\$$ smaller than all other appearing characters. We further write BWT_P (and analogously BBWT_P or BWT°_P) to denote the BWT of P for a string P. In what follows, we review means to simulate a linear scan of the text in forward or backward manner by the BWT of T and then translate this result to BBWT.

4 Backward and Forward Search

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 to $T[i+1]$, 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 to $T[i-1]$, we can use the backward search 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 $c \in [1..\sigma]$). 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] restoring the text from BWT in-place in $\mathcal{O}(n^{2+\epsilon})$ time uses the result of Munro and Raman [23] 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], this time bound can be improved to $\mathcal{O}(n^2)$ time in the RAM model.

If we allow more space, it is still advantageous to favor storing C instead of F if $\sigma = o(n)$ as 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. When working with the run-length compressed BWT, we want however a data structure that works in the run-length compressed space while supporting queries and updates more efficiently than the in-place approach:

4.1 Run-length Compressed Wavelet Trees

Given a run-length compressed string T with r character runs, there is an $\mathcal{O}(r \lg n)$ bits representation of T that supports access, rank, select, insertions, and deletions in $\mathcal{O}(\lg r / \lg \lg r)$ time. This data structure needs $\mathcal{O}(\lg r)$ time for any of those operations, and is described by Policriti and Prezza [27, 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 [24] 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 time if updates are restricted to be in-/decremental.

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

▶ **Corollary 3** ([27, Thm. 2]). *We can construct the* RLBWT *in* $\mathcal{O}(r_{\mathsf{BWT}} \lg n)$ *bits of space online on the reversed text in* $\mathcal{O}(n \lg r / \lg \lg r)$ *time.*

We can define LF and FL similarly for BBWT with the following peculiarity:

4.2 Search with the Bijective BWT

The major difference to the BWT is that 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 keeps within the same Lyndon factor). This fact was exploited for circular pattern matching [18], but is not of interest here. Instead, we follow the analysis of the so-called *rewindings* [2, 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 search step at such an entry results in a rewinding, i.e., we 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) by applying the LF mapping. We use this property with Lemma 2 in the following for reading the Lyndon factors from T individually in the order $T_t, \ldots, T_1$.

5 Run-Length Compressed Conversions

In this section, we consider BWT and BBWT represented in the run-length compressed wavelet tree representation of Sect. 4.1 taking respectively $\mathcal{O}(r_{\mathsf{BWT}} \lg n)$ 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. The run-length compressed wavelet trees support a backward search step in $\mathcal{O}(\lg r / \lg \lg r)$ time with $r := \max(r_{\mathsf{BWT}}, r_{\mathsf{BBWT}})$. Our goal of this section is to 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.

5.1 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. 3 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 search 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.

5.2 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.[1] This time complexity also covers a forward search 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 since this algorithm only needs forward access, where the position i with $\mathsf{BWT}[i] = \$$ is our starting point, as $\mathsf{FL}[i]$ 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 use the backward search 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

[1] 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 [17] 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.

To \ From	T	BWT	BBWT	BWT$^\circ$
T	\	[9, Fig. 3]	Sect. 6.3	Sect. 6.2
BWT	[9, Fig. 2]	\	Sect. 6.3	
BBWT	Sect. 6.1	Sect. 6.4	\	
BWT$^\circ$	Sect. 6.1			\

Table 1 Overview of in-place conversions in focus of Sect. 6 working in quadratic time.

(for recursing on T_{x+1}). Inserting a Lyndon factor into RLBBWT works exactly as sketched by Bonomo et al. [5, Thm. 17] or in Algo. 1 in the appendix.

5.3 From Run-Length Compressed Text to RLBWT

Finally, we want to study how much space is needed to compute RLBWT in $r_{\mathsf{BWT}} \lg \sigma$ bits (without a wavelet tree) having the text also run-length compressed. For that, we use the algorithm of Crochemore et al. [9, Fig. 2] to compute the RLBWT in-place in $\mathcal{O}(n^2)$ online on the reversed text. Given that r_T is the number of character runs in the run-length compressed text T, we can compute RLBWT with $\max(3r_T/2, r_{\mathsf{BWT}}) + \mathcal{O}(1)$ words of working space and $\mathcal{O}(n^2)$ time: Suppose $\mathtt{b} := T[i] = \cdots = T[k]$ is a character run. Then adding this character run to RLBWT induces at most three character runs in BWT. That is because (1) the context of $T[k]$ (i.e., the suffix $T[k+1..]$ succeeding $T[k]$) starting with $T[k+1]\cdots$ can be arbitrary, but (2) the context of $T[j]$ starts always with one or multiple $\mathtt{b}$'s, for $j \in [i..k)$. Finally, (3) the character $T[i-1]$ whose context has a common prefix with $T[j]$ for $j \in [i..k)$ can create another character run in BBWT. Hence, adding a character run of T to BWT can cause at most three new character runs in BWT. After parsing half of the text, we need $3r_T/2 + \mathcal{O}(1)$ words of working space.

6 In-Place Conversions

We now study various in-place conversions that work in quadratic time by accessing LF or FL in $\mathcal{O}(n)$ time having only stored BWT, BBWT, or BWT$^\circ$. We note that the conversions from the text also work in the comparison model, while restore the text or converting two different transformations 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 $\epsilon > 0$ [23]. We start with the conversion between BWT$^\circ$ and the text T (Sects. 6.1 and 6.2), where we show (a) that we can construct BWT$^\circ$ 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. 6.3 how to restore the text form BBWT, which allows us to also convert BBWT to BWT with the algorithm of Crochemore et al. [9, Fig. 2]. Finally, we show a conversion from BWT to BBWT in Sect. 6.4. An overview is given in Table 1.

6.1 Computing BWT$^\circ$

We can compute BWT$^\circ$ from T with the algorithm of Bonomo et al. [5] computing the extended BWT [22]. The extended BWT is the BWT defined on a set of primitive strings. It coincides with BBWT if this set of primitive strings is the set of Lyndon factors of T [5, Thm. 14]. We briefly describe their algorithm for computing the BBWT (cf. Fig. 1 and Algo. 1 in the appendix): 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 search $p \leftarrow \mathsf{LF}[p] + 1$, and insert $T_x[j]$ at $\mathsf{BBWT}[p]$. To see why this computes BBWT, we note that the last character of the most recently inserted Lyndon factor T_x is always the first character in $\mathsf{BBWT}_{T_1 \cdots T_x}$ according to Lemma 2. By recursively inserting the preceding character at the place returned by a backward search step, we precisely insert this character at the position where we would expect it (another backward search step from the same position p would then return the inserted character). Using only n backward search steps and n insertions, this algorithm works in-place in $\mathcal{O}(n^2)$ time by simulating LF as described in Sect. 4.

Consequently, we can build BWT° if T is a Lyndon word since in this case BWT° and BBWT coincide. It is easy to generalize this to work for a general string T. First, if T is primitive, then we can compute its so-called Lyndon conjugate $\mathsf{conj}_j(T)$ in $\mathcal{O}(n)$ time with the following two lemmata:

▶ **Lemma 4** ([10, Prop. 1.3]). *Given two Lyndon words S and T, ST is a Lyndon word if $S \prec_{\text{lex}} T$.*

▶ **Lemma 5.** *Given a primitive string T, we can find its* Lyndon conjugate *$\mathsf{conj}_j(T)$ in $\mathcal{O}(n)$ time with $\mathcal{O}(\lg n)$ bits of space, where a conjugate is called Lyndon if it is a Lyndon word. The Lyndon word of a primitive string is uniquely defined.*

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 4, $T_t T_1$ is a Lyndon word since $T_t \prec_{\text{lex}} T_1$, and so is $T_t T_1 \cdots T_{t-1}$. Hence, we have found T's Lyndon conjugate. ◀

Since BWT° is identical to $\mathsf{BBWT}_{\mathsf{conj}_j(T)}$, we are done by running the algorithm of Bonomo et al. [5] on $\mathsf{conj}_j(T)$. 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°, we only need to make each character in BWT°_P to a character run of length k, i.e., if $\mathsf{BWT}_P[i] = \mathsf{c}$, we append c^k to BWT° for increasing $i \in [1..|P|]$. 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. We summarized these steps in the pseudo code of Algo. 2 in the appendix.

6.2 Restoring T from BWT°

To revert BWT°, we use the techniques of Crochemore et al. [9, Fig. 3] restoring T from BWT in-place. For 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 move p to $\mathsf{BWT}[\mathsf{FL}[i]]$ and subsequently move $\mathsf{BWT}^\circ[i]$ to the left of T. The algorithm works exactly as [9, Fig. 3] if we do an additional backward search at p for inserting $. After recursing n times, we have converted BWT° to T. More involving is restoring T from BBWT or converting BBWT to BWT, which we tackle next.

6.3 Computing BWT or T from BBWT

Similarly to Sect. 5.1, we read the Lyndon factors from BBWT in the order $T_t, \ldots, T_1$, and move each read Lyndon factor directly to BWT 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 $\mathsf{BWT}_{T_{x+1} \cdots T_t}$, producing $\mathsf{BBWT}_{T_1 \cdots T_{x-1}}$ and $\mathsf{BWT}_{T_x \cdots T_t}$. This allows us to recurse by reading always the

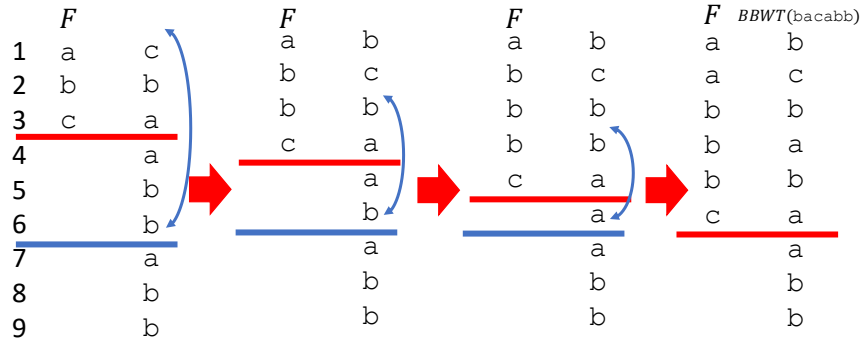

Figure 1 Computing BBWT from our running example $T = \texttt{bacabbabb}$ in four steps (visualized by four columns), cf. Sect. 6.1. In each column, the characters from the top to the red line form the currently built BBWT. The characters below that up to the blue line are under consideration of being merged into BBWT. The blue line is always before the beginning of the next yet unread Lyndon factor. *First column*: We have already computed the BBWT of $T_1T_2 = \texttt{bac}$, which is $\texttt{cba}$. In the following we want to add the next Lyndon factor $T_3 = \texttt{abb}$ to it. For that, we prepend its last character to the currently constructed BBWT. *Second column*: We move the last character above the blue 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 $T_1T_2T_3$ in the *forth column*.

last Lyndon factor T_x stored in $\mathsf{BBWT}_{T_1 \cdots T_x}$, whose position we know thanks to Lemma 2. In detail, we start at $\mathsf{BBWT}[1]$ and perform backward search steps until we return back to this position. The number of steps tells us how long T_x is. Next, we want to remove the entry in BBWT corresponding to $\mathsf{conj}_{|T_x|-j}(T_x)$ for increasing $j \in [1..|T_x| - 1]$ and append it to an increasing text buffer stored in the space of BBWT. For that, we apply the algorithm of Sect. 6.2 with the pointer $p = \mathsf{FL}[1]$, which now extracts only T_x from BBWT due to the rewinding (cf. Sect. 4.2). Subsequently, we run the in-place BWT construction algorithm of Crochemore et al. [9, Fig. 2] on the text buffer. We continue with the remaining BBWT, and merge the text buffer with the already computed BWT. We can do so since the used BWT construction algorithm works online on the reversed text. If we want to convert BBWT to T, we do not have to convert the text buffer to BWT, but instead prepend the contents of the text buffer to the already restored part of the text.

6.4 Computing BBWT from BWT

Like in Sect. 5.2, 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 detected the first Lyndon factor T_1. Let $\texttt{f}$ denote its last character. Further let $i_{\texttt{f}}$ and $i_{\$}$ be the position of the last character of T_1 and the last character of T, respectively, such that $\mathsf{BWT}[i_{\texttt{f}}] = \texttt{f}$ and $\mathsf{BWT}[i_{\$}] = \$$ (cf. Fig. 2). Let $p := \mathsf{LF}[i_{\texttt{f}}]$ such that $\mathsf{F}[p] = \texttt{f}$ 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, $T_1 \succeq_{\text{lex}} T_2$. Consequently, the suffix $T[\mathsf{b}(T_2)..]$ (the context of $\mathsf{BWT}[i_{\texttt{f}}]$) is lexicographically smaller than the suffix $T[\mathsf{b}(T_1)..]$ (the context of $\mathsf{BWT}[i_{\$}]$), i.e., $i_{\texttt{f}} < i_{\$}$.

Our aim is to change BWT such that a forward or backward search within the characters belonging to T_1 always results in a cycle. Informally, we want to cut out T_1 from BWT, which additionally allows us to recursively continue with the FL mapping to find the end of the next Lyndon factor T_2.[2] For that, we exchange $\mathsf{BWT}[i_{\$}]$ with $\mathsf{BWT}[i_{\texttt{f}}]$ (cf. Fig. 3). Then

[2] As a matter of fact, if we now want to restore the text with the modified BWT by LF, we would only produce $T_2 \cdots T_t$.

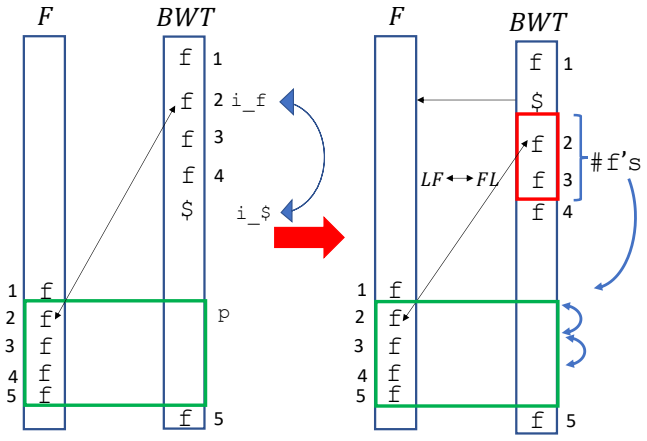

Figure 2 Setting of Sect. 6.4 with focus on forming a cycle for a Lyndon factor ending with `f` in BWT. *Left*: We exchange $\mathsf{BWT}[i_{\mathtt{f}}]$ with $\mathsf{BWT}[i_{\$}]$ with the aim to form a cycle. *Right*: To obtain this cycle we additionally need to swap at the top of the green rectangle as many entries as there are `f`'s in the red rectangle $\mathsf{BWT}[i_{\mathtt{f}}+1..i_{\$}-1]$. The green rectangle starts with $\mathsf{LF}[i_{\mathtt{f}}]$ (evaluated before the exchange) and ends with the largest position i with $\mathsf{F}[i]=\mathtt{f}$.

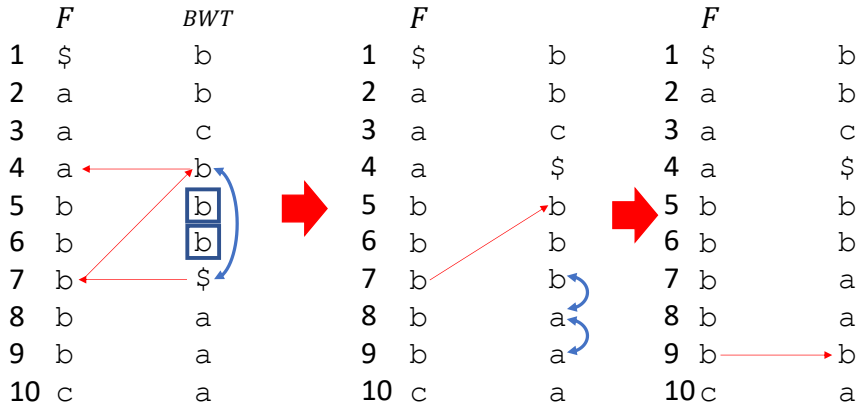

Figure 3 Computing BWT from BBWT (cf. Sect. 6.4) of our running example $T=$ `bacabbabb$`. At the *left*, we find the first Lyndon factor $T_1=$ `b` of T by forward search steps with FL. There are two `b`'s that are between the exchanged `b` at $\mathsf{BWT}[4]$ and `$`. Therefore, we need to swap two elements below of $\mathsf{LF}[4]=7$, as shown at the *middle*. This gives a cycle in the *right* figure. We can recurse, as the LF mapping of `$` now yields the second character of T.

the character $T[\mathsf{e}(T_1)+1]$ (i.e., the first character of T_2) becomes the next character of `$` in terms of the forward search ($\mathsf{BWT}[\mathsf{FL}[i_{\mathtt{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 $i\in(i_{\mathtt{f}}..i_{\$}]$. 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 $\mathsf{BWT}[i_{\mathtt{f}}]$), 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 $\mathsf{F}[i]\neq\mathtt{f}$. This may not be sufficient if the characters we swap are identical (cf. Fig. 4). In such a case, we recurse on $\mathsf{LF}[p]$ to swap non-identical characters, see also Algo. 3 in the appendix.

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 $\mathsf{BWT}[i_{\mathtt{f}}]$ changes the values of $\mathsf{BWT.rank}_{\mathtt{f}}(j)$ for every $j\in[i_{\mathtt{f}}..i_{\$}]$ by one. In particular, $\mathsf{BWT.select}_{\mathtt{f}}$ changes for those `f`'s in BWT that are between $i_{\mathtt{f}}$ and $i_{\$}$. Hence, the number of swaps m is the number of all positions $k\in[i_{\mathtt{f}}+1..i_{\$}-1]$ with $\mathsf{BWT}[k]=\mathtt{f}$. The swaps are performed at $\mathsf{BWT}[p..]$ since there starts the first F entry whose mapping has changed.

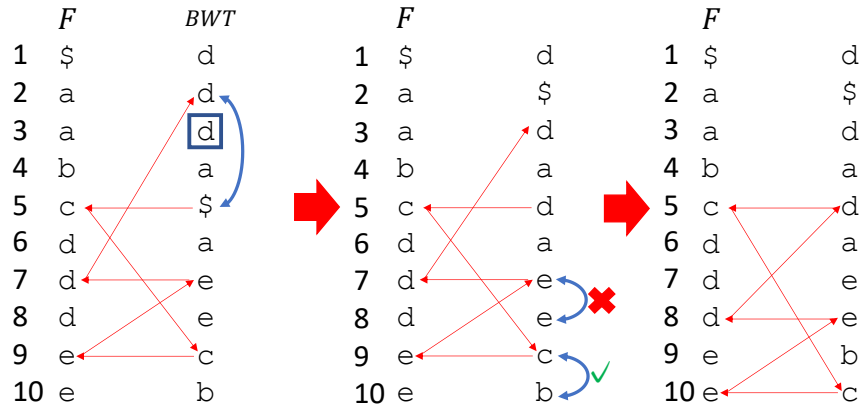

Figure 4 Special case for computing BWT from BBWT (cf. Sect. 6.4) with the different example string $T :=$ `cedabedad$` having $T_1 =$ `ced` as its first Lyndon factor. *Left:* We find the first Lyndon factor $T_1 =$ `ced` of T by forward search steps with FL. Its last character is stored at BWT[3]. *Middle*: By exchanging `$` with the last character of T_1 in BWT, the LF mapping for the third `d` in F becomes invalid. However, there is only a character run of $T_1[|T_1|-1] =$ `e` in BWT of the $T_1[|T_1|] =$ `d` interval in F starting with $p = 7$. So we recurse on LF$[p]$ to find characters different from $T_1[|T_1|-2] =$ `c` to swap in the respective $T_1[|T_1|-1] =$ `e` interval. *Right:* We have created a cycle with the characters of the first Lyndon factor. A forward search step at `$` gives the first character of the next Lyndon factor.

However, if 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 the interval in which F is `f` (the `f` interval of F) to swap elements below this run with elements starting with this run. If the number of effective swaps (i.e., a swap not exchanging two identical characters) is less than m, we recurse on the $T_1[|T_1|-1]$ interval of F starting at LF$[p]$ (using the LF value before the swapping).

To see why this restores the LF mapping for the remaining part of the text $T_2 \cdots T_t$, we examine a substring $x_1x_2\mathtt{f} \in \Sigma^3$ that is represented in BWT (before changing it) with $\mathsf{BWT}[p+1] = x_2, \mathsf{BWT}[\mathsf{LF}[p+1]] = x_1, \mathsf{BWT}[\mathsf{FL}[p+1]] = \mathtt{f}$, and $i_0 := \mathsf{FL}[p+1] \in [i_{\mathtt{f}}+1..i_{\$}-1]$. Then FL$[p]$ becomes i_0 after exchanging BWT$[i_{\$}]$ with BWT$[i_{\mathtt{f}}]$. If we swap BWT$[p]$ with BWT$[p+1]$, then LF$[p]$ is still i_0, but BWT[LF$[p]$] becomes x_1 such that BWT still represents this substring $x_1x_2\mathtt{f}$ (we only swap entries in the `f` interval of F such that BWT[FL$[p]$] is still `f`). With the same argument, we can generalize this proof from $p+1$ to $p+j$ within the `f` interval of F or for the swaps in the recursive call in the case that the number of effective swaps in the `f` interval of F is too small.

7 Open Problems

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

At `https://github.com/daikihashimoto/BWT_to_BBWT`, we have some preliminary implementations available giving empirical evidence of our conversions.

[3] That is because in this case $p = i_{\$}$, and hence, BWT$[p]$ was `$` but is now `f` $= T_1[1]$.

References

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

2 H. Bannai, J. Kärkkäinen, D. Köppl, and M. Piatkowski. Indexing the bijective BWT. In *Proc. CPM*, volume 128 of *LIPIcs*, pages 17:1–17:14, 2019.

3 H. Bannai, J. Kärkkäinen, D. Köppl, and M. Piatkowski. Indexing the bijective BWT. *ArXiv 1911.06985*, 2019.

4 P. Bille, A. R. Christiansen, P. H. Cording, I. L. Gørtz, F. R. Skjoldjensen, H. W. Vildhøj, and S. Vind. Dynamic relative compression, dynamic partial sums, and substring concatenation. *Algorithmica*, 80(11):3207–3224, 2018.

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

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

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

8 K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, IV. The quotient groups of the lower central series. *Annals of Mathematics*, pages 81–95, 1958.

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

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

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

12 P. Ferragina, G. Manzini, and S. M. Muthukrishnan. *The Burrows-Wheeler Transform: Ten Years Later*. DIMACS, 2004.

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

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

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

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

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

18 W. Hon, T. Ku, C. Lu, R. Shah, and S. V. Thankachan. Efficient algorithm for circular Burrows-Wheeler transform. In *Proc. CPM*, volume 7354 of *LNCS*, pages 257–268, 2012.

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

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

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

22 S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows-Wheeler transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007.

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

24 G. Navarro and Y. Nekrich. Optimal dynamic sequence representations. *SIAM J. Comput.*, 43(5):1781–1806, 2014.

426 **25** G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array
427 construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.
428 **26** T. Ohno, Y. Takabatake, T. I, and H. Sakamoto. A faster implementation of online
429 run-length Burrows-Wheeler transform. In *Proc. IWOCA*, volume 10765 of *LNCS*, pages
430 409–419, 2017.
431 **27** A. Policriti and N. Prezza. LZ77 computation based on the run-length encoded BWT.
432 *Algorithmica*, 80(7):1986–2011, 2018.

A Pseudo Codes

Algorithm 1 Computing BBWT from T [5, Algo. 13], cf. Sect. 6.1

```
1  foreach Lyndon factor F_x with x = 1 up to t do
2      prepend T_x[|T_x|] to BBWT
3      p ← 1                                   ▷ insert position in BBWT
4      foreach i = |T_x| − 1 down to 1 do
5          p ← LF[p] + 1
6          insert T_x[i] at BBWT[p]
```

Algorithm 2 Computing BWT° in-place, cf. Sect. 6.1

```
1  assume that T is a Lyndon word
2  n ← |T|
3  c ← T[n]
4  T[n] ← T[n − 1]
5  T[n − 1] ← c
6  lastpos ← n
7  lastchar ← T[n]
8  for i = n − 1 to 1 do
9      count ← 0; rank ← 0
10     for j = i to n do
11         if T[j] < lastchar then
12             count ← count + 1
13     for j = i to lastpos do
14         if T[j] = lastchar then
15             rank ← rank + 1
16     lastpos ← i + count + rank
17     lastchar ← T[i − 1]
18     for j = i − 1 to lastpos do
19         T[j] ← T[j + 1]
20 return T                                    ▷ now storing BWT°
```

Algorithm 3 Computing BBWT from BWT in-place, cf. Sect. 6.4

```
1  x ← 1
2  while LF[i_$] ≠ $ where i_$ is the position of $ in BWT do
3      i_$ ← position of $ in BWT, i_f ← LF[i_$]
4      while i_f does not correspond to T_x[|T_x|] do                ▷ use Lemma 1
5          i_f ← i_f + 1
6      p ← position in BWT such that FL[p] = i_f
7      invariants:
         BWT[i_$] = T_x[|T_x|], BWT[FL[i_$]] = T_x[1], F[p] = T_x[|T_x|], BWT[p] = T_x[|T_x| − 1]
8      exchange BWT[i_f] with BWT[i_$]
9      m ← |{i ∈ [i_f + 1..i_$ − 1] with BWT[i] = T_x[|T_x|]}|
10     depth ← 0
11     p′ ← LF[p]                          ▷ save LF[p] as it may change while m > 0 do
12         dst ← p, src ← p + 1
13         while BWT[src] = T_x[|T_x| − depth − 1] do src ← src + 1
14         while F[src] = T_x[|T_x| − depth] do
15             swap BWT[src] with BWT[dst]
16             increment src and dst, decrement m
17         increment depth, p ← p′, p′ ← LF[p′]
18     x ← x + 1                                          ▷ done with T_1 ··· T_{x−1}
19 invariant: x = t
```