In-Place BWT and Lyndon Array Construction in Constant Space

Felipe A. Louza[1,2] and Arnaud Lefebvre[3⋆]

[1] Universidade Federal de Uberlândia, Uberlândia, MG, Brazil
[2] Universidade Estadual de Campinas, Campinas, SP, Brazil
louza@ufu.br
[3] Normandie Univ., 76000 Rouen, France
arnaud.lefebvre@univ-rouen.fr

Abstract. We show that the BWT and the Lyndon array of a string can be constructed simultaneously using only $O(1)$ additional memory words in the general ordered alphabet model. Our algorithm extends the in-place quadratic-time BWT construction of Crochemore *et al.* [10] by maintaining suffix ranks during the right-to-left construction, from which the Lyndon array is derived via a next-smaller-value procedure. We also consider the construction of the succinct $2n$-bit representation of the Lyndon array under the same constant-space constraints. Although not practical, our methods demonstrate that these fundamental structures admit in-place constructions under the strongest possible space bounds, even over unbounded alphabets.

Keywords: BWT, Lyndon Array, Succinct data structures, In-place algorithms, String processing

1 Introduction

The Burrows–Wheeler Transform (BWT) [8] and the Lyndon array (LA) [14] are fundamental structures in string processing. The BWT underlies compressed indexing [13] and data compression [8], while the LA captures the combinatorial structure of a string through its Lyndon factorization and plays a role in several pattern-matching [5,4] and combinatorial algorithms [2,3]

The BWT depends on the lexicographic ordering of all suffixes of the input string. A standard approach computes the BWT in $O(n)$ time after constructing the suffix array (SA) [19], which requires $O(n \log n)$ bits of extra space. Several algorithms compute the BWT directly in linear time using only $O(n \log \sigma)$ bits of working space, including the induced-sorting algorithm of Okanohara and Sadakane [23] and subsequent deterministic constructions [20,6]. Under the much stronger restriction of using only $O(1)$ additional words, however, the most efficient known algorithm is the in-place $O(n^2)$ construction of Crochemore *et al.* [10], which overwrites the input text while requiring only constant extra memory. Their method works in the general ordered alphabet model, requiring only constant-time symbol comparisons and no assumptions on the alphabet size.

The Lyndon array can likewise be computed from the suffix ordering in $O(n)$ time using $O(n \log n)$ bits of extra space [14,15,17,22]. Alternatively, it can be obtained via Duval's factorization algorithm [12] in $O(n^2)$ time using $O(1)$ extra space. More recently, Bille *et al.* [7] showed that the Lyndon array can be computed in $O(n)$ time using only $O(1)$ extra words, and in $o(n)$ extra bits for the succinct representation [18]. Their algorithm, like that of Crochemore *et al.*, operates in the general ordered alphabet model and does not depend on σ.

⋆ Corresponding author

Felipe A. Louza, Arnaud Lefebvre: In-Place BWT and Lyndon Array Construction in Constant Space, pp. 3–12.
Proceedings of PSC 2026, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07576-0 © Czech Technical University in Prague, Czech Republic

i	Sorted suffix $T_{\mathsf{SA}[i]}$	$\mathsf{SA}[i]$	$\mathsf{ISA}[i]$	$\mathsf{BWT}[i]$	$\mathsf{LA}[i]$
0	\$	6	4	A	1
1	A\$	5	3	N	2
2	ANA\$	3	6	N	1
3	ANANA\$	1	2	B	2
4	BANANA\$	0	5	\$	1
5	NA\$	4	1	A	1
6	NANA\$	2	0	A	1

$$\mathsf{LA}_{\mathsf{BP}} = ()(())(())()()$$

Figure 1. SA, ISA, BWT and LA for $T = \texttt{BANANA\$}$. The Lyndon array $\mathsf{LA}[i]$ gives the length of the longest Lyndon word starting at position i. $\mathsf{LA}_{\mathsf{BP}}$ gives the succinct $2n$-bit representation of LA.

Our contribution. In this paper, we extend the quadratic in-place BWT algorithm of Crochemore *et al.* [10] to *simultaneously* compute the Lyndon array, while still using only $O(1)$ extra space. The resulting algorithm constructs both structures in $O(n^2)$ time using constant additional space. Compared with the linear-time algorithm of Bille *et al.* [7], our approach is asymptotically slower but computes the BWT *in place* while simultaneously constructing the Lyndon array using only $O(1)$ additional space. We further show that the succinct $2n$-bit representation of the Lyndon array can be constructed under the same constant-space constraints in $O(n^{3+\epsilon})$ time. In contrast, the construction of Bille *et al.* [7] achieves linear time by allowing additional $o(n)$ bits of working space. Alternatively, the succinct Lyndon array can be computed in $O(n/\log_\sigma n)$ time using extra space, as shown in [1].

2 Preliminaries

Let Σ be an ordered alphabet of σ symbols, which may be unbounded. We assume symbols in Σ are compared in constant-time. Let \$ be a special symbol (end marker) not in Σ that is lexicographically smaller than every symbol in Σ. We denote by Σ^* the set of all finite nonempty strings over Σ, and we define $\Sigma^*\$ = \{\, T\$ \mid T \in \Sigma^* \,\}$.

For a string $T = T[0]T[1]\dots T[n-1]$ over $\Sigma^*\$$, of length n, we write $T[i,j] = T[i]T[i+1]\cdots T[j]$ for the substring starting at position i and ending at j, where $0 \le i \le j < n$. A *prefix* of T is a substring of the form $T[0,k]$, and a *suffix* is a substring of the form $T[k,n-1]$, with $0 \le k < n$. We denote the suffix that begins at position i by T_i.

The *suffix array* (SA) [19] of $T \in \Sigma^*\$$ is an array $\mathsf{SA}[0,n-1]$ that contains a permutation of $\{0,\dots,n-1\}$ such that

$$T_{\mathsf{SA}[0]} < T_{\mathsf{SA}[1]} < \cdots < T_{\mathsf{SA}[n-1]}$$

in lexicographic order. The *inverse suffix array* (ISA) is the inverse permutation defined by $\mathsf{ISA}[\mathsf{SA}[i]] = i$. Hence, $\mathsf{ISA}[k]$ gives the lexicographic rank of the suffix T_k.

The *Burrows–Wheeler Transform* (BWT) [8] of a string T is a permutation of its characters that tends to cluster equal symbols. It can be defined using the suffix

array as

$$\mathsf{BWT}[i] = \begin{cases} T[\mathsf{SA}[i]-1], & \text{if } \mathsf{SA}[i] > 0, \\ \$, & \text{otherwise.} \end{cases}$$

A *Lyndon word* [9] is a nonempty string that is strictly smaller in lexicographic order than all of its proper nonempty suffixes. Every string can be uniquely factorized as a concatenation of Lyndon words in nonincreasing lexicographic order (the *Lyndon factorization*).

The *Lyndon array* (LA) [14] (also called Lyndon table [11]) of a string $T \in \Sigma^*\$$ is an array $\mathsf{LA}[0, n-1]$ that stores, for each position i, the length of the longest Lyndon word starting at i. Formally,

$$\mathsf{LA}[i] = \min\{\, j - i \mid j > i,\ T[i, j-1] \text{ is not smaller than } T[j, n-1] \,\},$$

or equivalently, the Lyndon array can be computed from the inverse suffix array using the *next smaller value* (NSV) function:

$$\mathsf{LA}[i] = \mathrm{NSV}_{\mathsf{ISA}}(i) - i, \tag{1}$$

where

$$\mathrm{NSV}_{\mathsf{ISA}}(i) = \min\{\, j > i \mid \mathsf{ISA}[j] < \mathsf{ISA}[i] \,\}, \tag{2}$$

and $\mathrm{NSV}_{\mathsf{ISA}}(i) = n$ if no such j exists.

Figure 1 shows the SA, ISA, BWT and Lyndon array for $T = \texttt{BANANA\$}$.

The Lyndon array can be represented as a succinct balanced-parentheses sequence (BPS) [21] that requires only $2n$ bits of space. Introduced in [18], when augmented with $o(n)$ additional bits, this representation allows every value $\mathsf{LA}[i]$ to be retrieved in $O(1)$.

Let T be a string of length n and let $\mathsf{LA}[0..n-1]$ be its Lyndon array. Define an array $\mathsf{R}[0..n-1]$ by

$$\mathsf{R}[i] = i + \mathsf{LA}[i]. \tag{3}$$

The array R satisfies $\mathsf{R}[i] > i$ for all i and induces a set of properly nested intervals $[i, \mathsf{R}[i])$, which induces an ordered tree, called *Lyndon tree* [11], whose structure can be encoded as BPS.

The BPS encoding of the nested intervals, stored in $\mathsf{LA}_{\mathsf{BP}}[0, 2n-1]$, is computed by scanning positions $i = 0, 1, \ldots, n-1$ of LA. Let pos denote the index of the next free position in $\mathsf{LA}_{\mathsf{BP}}$, initially set to 0. When processing a position i, an opening parenthesis "(" corresponding to i is written at position $\mathsf{LA}_{\mathsf{BP}}[\mathsf{pos}]$. The matching closing parenthesis ")" is written at position $\mathsf{LA}_{\mathsf{BP}}[\mathsf{pos} + 2 \cdot \mathsf{LA}[i] - 1]$. Afterwards, pos is advanced to the smallest index such that $\mathsf{LA}_{\mathsf{BP}}[\mathsf{pos}]$ is still undefined. Because each position of LA contributes exactly one opening and one matching closing parenthesis, the resulting BPS contains n matching pairs and therefore occupies exactly $2n$ bits.

Given the BPS $\mathsf{LA}_{\mathsf{BP}}[0, 2n-1]$. Let $\mathsf{open}(i)$ (resp. $\mathsf{close}(i)$) denote the position of the i-th opening (resp. closing) parenthesis in $\mathsf{LA}_{\mathsf{BP}}$. By construction, we have

$$\mathsf{LA}[i] = \frac{\mathsf{close}(i) - \mathsf{open}(i) + 1}{2} \tag{4}$$

Using standard balanced-parentheses navigation operations, the position of $\mathsf{open}(i)$ and $\mathsf{close}(i)$ can be computed in $O(1)$ time when $\mathsf{LA}_{\mathsf{BP}}$ is augmented with $o(n)$ additional bits, and thus every value $\mathsf{LA}[i]$ can be retrieved in constant time (see [18] for details).

	s	BWT	sorted suffixes
$s \to$	0	?	BANANA$
	1	A	$
	2	N	A$
	3	N	ANA$
$p \to$	4	$	ANANA$
	5	A	NA$
	6	A	NANA$

	s	BWT	sorted suffixes
	0	A	$
	1	N	A$
	2	N	ANA$
	3	B	ANANA$
$r \to$	4	$	BANANA$
	5	A	NA$
	6	A	NANA$

Figure 2. After Step 1 (left) and Step 4 (right) for $T =$ `BANANA$`, $s = 0$.

Figure 1 also shows the succinct $2n$-bit representation of the Lyndon array for $T =$ `BANANA$`.

3 In-Place BWT

The in-place BWT algorithm of Crochemore *et al.* [10] computes the transform in quadratic time using only constant extra space, overwriting the input string from right to left with its BWT as the construction proceeds.

The BWT is computed inductively on the length of the suffix. Let $BWT(T_s)$ denote the BWT of the suffix $T_s = T[s, n-1]$. The base cases are the two rightmost suffixes:

$$BWT(T_{n-2}) = T_{n-2}, \quad BWT(T_{n-1}) = T_{n-1}.$$

For the inductive step, assume that $BWT(T_{s+1})$ is already stored in $T[s+1, n-1]$. The algorithm computes $BWT(T_s)$ by inserting the new suffix T_s in its correct lexicographic position among the suffixes $T_{s+1}, \ldots, T_{n-1}$. It relies on the fact that the position of the end marker \$ in $BWT(T_{s+1})$ corresponds to the rank of the suffix T_{s+1} among the current suffixes.

The algorithm consists of four steps, for iterations $s = n-2, n-3, \ldots, 0$.

1. **Find the position of the end marker.** Locate the position p of \$ in $T[s+1, n-1]$.
2. **Compute the rank of the new suffix.** Let $c = T[s]$ be the first symbol of T_s. The local rank r of T_s among the suffixes $T_s, \ldots, T_{n-1}$ is obtained by summing the number of symbols in $T[s+1, n-1]$ that are strictly smaller than c, with the number of occurrences of c in $T[s+1, p]$.
3. **Insert T_s into the sequence.** Store c into $T[p]$, replacing the end marker \$.
4. **Shift symbols to insert a new end marker.** Shift the block $T[s+1, r]$ one position to the left and write \$ at position $T[r]$. Then, $BWT(T_s) = T_s$.

Figure 2 illustrates the execution of the in-place BWT algorithm on the text `BANANA$` for the iteration $s = 0$, showing the effect of Steps 1 and 4 on the partial $BWT(T_1)$ stored in $T[1\,..\,n-1]$.

The algorithm runs in $O(n^2)$ time and uses only $O(1)$ additional variables. Its simplicity and elegant inductive structure make it a useful foundation for extensions, such as computing the LCP array [16] or, as we show next, the inverse suffix array and then the Lyndon array, while preserving the constant-space property.

	s	BWT	ISA	sorted suffixes
$s \rightarrow$	0	?	?	BANANA$
	1	A	3	$
	2	N	5	A$
	3	N	2	ANA$
$p \rightarrow$	4	$	4	ANANA$
	5	A	1	NA$
	6	A	0	NANA$

	s	BWT	ISA	sorted suffixes
	0	A	4	$
	1	N	3	A$
	2	N	6	ANA$
	3	B	2	ANANA$
$r \rightarrow$	4	$	5	BANANA$
	5	A	1	NA$
	6	A	0	NANA$

Figure 3. After Step 1 (left) and Step 5 (right) for $T =$ `BANANA$`, $s = 0$.

4 In-place BWT along with the Lyndon Array

We now describe how to extend the in-place BWT algorithm (Section 3) to compute the BWT and the Lyndon array (LA) simultaneously, using only constant additional memory.

The key idea is to maintain, during each iteration $s = n-2, n-3, \ldots, 0$, the lexicographic ranks among the suffixes $T_s, \ldots, T_{n-1}$, stored in the $\mathsf{ISA}[s, n-1]$. At the end, the values in $\mathsf{ISA}[0, n-1]$ are converted into the Lyndon array, according to Equations 1 and 2.

The algorithm proceeds inductively on the length of the suffix, as in the original in-place BWT construction. Let $\mathsf{ISA}(T_s)$ denote the partial ISA for $T_s = T[s, n-1]$. The base cases for ISA are:

$$\mathsf{ISA}(T_{n-2}) = [1, 0], \quad \mathsf{ISA}(T_{n-1}) = [0].$$

For the inductive step, assume that we also have $\mathsf{ISA}(T_{s+1})$ already stored in $\mathsf{ISA}[s+1, n-1]$. The algorithm computes $\mathsf{ISA}(T_s)$ by storing in $\mathsf{ISA}[s]$ the rank r of the new suffix T_s inserted into $BWT(T_{s+1})$, and by updating the ISA-values of all suffix that has rank larger than r in $\mathsf{ISA}[s+1, n-1]$. Therefore, our algorithm has a new Step 5, added just after Step 4 of the in-place BWT algorithm as follows.

5. **Update the inverse suffix array.** After inserting T_s in $BWT(T_{s+1})$, set $\mathsf{ISA}[s] = r$, and increment by one all entries $\mathsf{ISA}[i]$, with $s+1 \leq i < n$, such that $\mathsf{ISA}[i] \geq r$.

Lemma 1. *Step 5 maintains the correct rank information for all suffixes currently present in $BWT(T_s)$.*

Proof. We prove the invariant by induction on s (right to left). For the base $s = n-2$ the two ranks are set correctly. Assume that before processing s the array $\mathsf{ISA}[s+1..n-1]$ stores the ranks $0, \ldots, m-1$ of $T_{s+1}, \ldots, T_{n-1}$, with $m = n-s$. Let $r \in \{0, \ldots, m\}$ be the rank of T_s among $T_s, \ldots, T_{n-1}$. Every suffix with old rank $q < r$ are shifted one position to the left and keep their rank, while those with $q \geq r$ must have its rank increased by 1, as T_s was inserted into position r; moreover $\mathsf{ISA}[s]$ must be r. Step 5 performs exactly these updates (incrementing entries with $\mathsf{ISA}[i] \geq r$ and setting $\mathsf{ISA}[s] = r$). Hence after the update $\mathsf{ISA}[s..n-1]$ stores the ranks $0, \ldots, m$ of $T_s, \ldots, T_{n-1}$, which proves the claim.

Figure 3 illustrates the execution of the algorithm on the text `BANANA$` for the iteration $s = 0$, showing the effect of Steps 1 and 5 on the partial BWT and ISA.

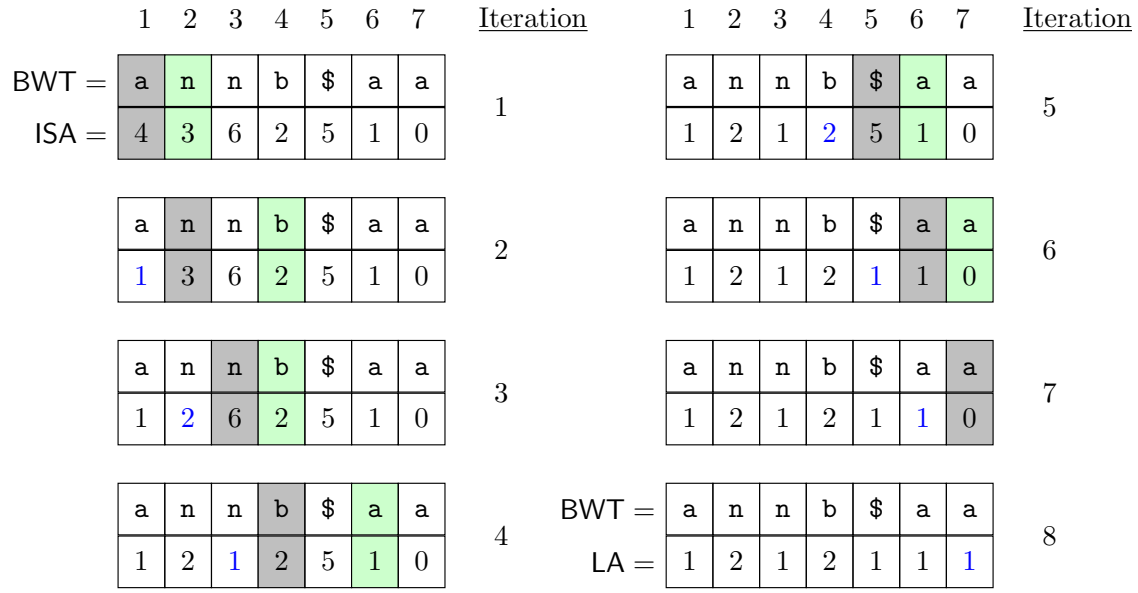


Figure 4. Example of the in-place LA construction from ISA on $T =$ `BANANA$`.

Finally, given the inverse suffix array $\mathsf{ISA}(T_0)$, we scan $\mathsf{ISA}[0, n-1]$ from left to right, overwriting each processed entry with the corresponding Lyndon array value. For each position $i = \{0, 1, \ldots, n-1\}$. Then, $\mathsf{LA}[i] = j - i$. This corresponds to computing the next smaller value (NSV) over the ISA, as described in Section 2. Since $\mathsf{ISA}[i]$ is no longer needed after processing position i, it can be overwritten by $\mathsf{LA}[i]$, yielding the final Lyndon array without allocating any additional memory.

Figure 4 illustrates how the ISA values maintained during the in-place BWT construction are subsequently overwritten by the corresponding Lyndon array values.

In the appendix, we provide a C implementation of the algorithm, including the in-place construction of the BWT (Lines 10–28), the maintenance of the inverse suffix array during the construction process (Lines 30–34), and the subsequent derivation of the Lyndon array from the computed ranks using the next-smaller-value procedure (Lines 36–43).

Theorem 2. *The BWT can be computed in-place together with Lyndon array in $O(n^2)$ time.*

Proof. The algorithm follows the same inductive structure as the in-place BWT construction. Each iteration performs a constant amount of work plus two $O(n)$ scans (Steps 1–2), and there are n iterations, yielding an overall running time of $O(n^2)$. The update of the inverse suffix array in Step 5 requires only an extra linear scan at each iteration, and the final conversion of ISA into the Lyndon array by the next-smaller-value scan also takes $O(n^2)$ time. Thus, the total time remains quadratic. The space usage is $O(1)$ words beyond the input text T and the output array LA.

5 Computing the Succinct Lyndon Array

The succinct $2n$-bit representation of the Lyndon array [18] can be computed directly from the BWT of T, using $O(1)$ additional working space. Our construction follows the in-place BWT inversion algorithm [10, Section 3]

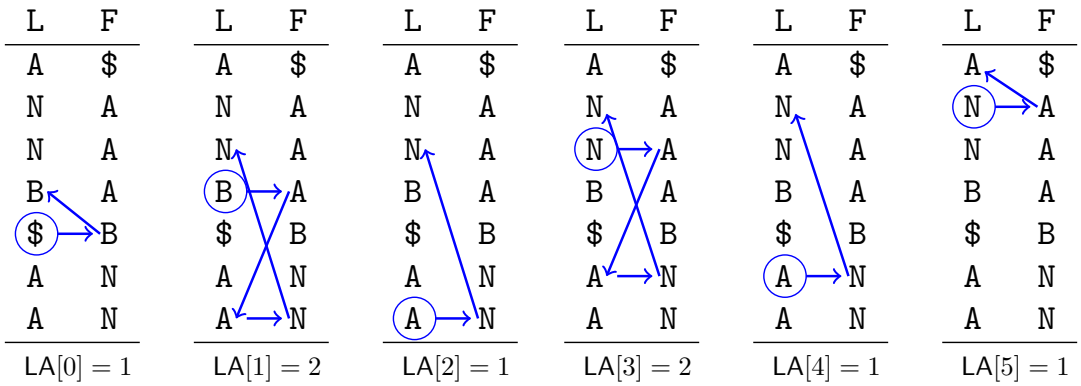


Figure 5. Computing the LA-values from the BWT of T.

Initially, $\mathsf{BWT}(T)$ is stored in the array $L[0..n-1]$, while the balanced-parentheses representation $\mathsf{LA_{BP}}[0..2n-1]$ is empty. We begin by scanning L to locate the position p of the symbol \$, which corresponds to the value $\mathsf{ISA}[0]$. For each step $i = 0, 1, \ldots, n-2$, let $p = \mathsf{ISA}[i]$. Using the in-place BWT inversion procedure, we iteratively apply the LF-mapping [8] starting from position p in order to compute the positions

$$\mathsf{ISA}[i+1],\ \mathsf{ISA}[i+2],\ \ldots,\ \mathsf{ISA}[i+j],$$

which correspond to the rank of $T_{i+1}, T_{i+2}, \ldots, T_{i+j}$. This procedure continues until the first position $q = \mathsf{ISA}[i+j]$ such that $q < p$ is encountered. By Equation 2, this index j equals $\mathrm{NSV}_{\mathsf{ISA}}(i)$, and therefore

$$\mathsf{LA}[i] = j - i.$$

This value is encoded directly into $\mathsf{LA_{BP}}[0..2n-1]$ by writing an opening parenthesis "(" at the current leftmost free position pos, and the matching closing parenthesis ")" at position $\mathsf{pos} + 2\mathsf{LA}[i] - 1$, as described in Section 2. The pointer pos is then advanced to the next free position, and the algorithm proceeds with step $i+1$. For the last position $i = n-1$, since $T[n-1] = \$$, the suffix T_{n-1} is lexicographically smaller than all other suffixes and therefore $\mathsf{LA}[n-1] = 1$, which is encoded directly in $\mathsf{LA_{BP}}$.

Figure 5 illustrates this process for $T = \texttt{BANANA\$}$. Starting from the position of the end marker, successive LF-mapping steps identify the first suffix whose lexicographic rank is smaller than the current one.

Theorem 3. *The succinct $2n$-bit Lyndon array can be computed in-place in $O(n^{3+\epsilon})$ time.*

Proof. Given $\mathsf{BWT}(T)$, the algorithm processes each position $i = 0, 1, \ldots, n-1$ by performing successive LF-mapping steps until the value $\mathsf{LA}[i]$ is determined. In the worst case, this requires $O(n)$ LF steps per position. Each LF step is computed using the in-place BWT inversion algorithm of Crochemore *et al.* [10], whose running time is $O(n^{1+\varepsilon})$ in the general ordered alphabet model, for any fixed constant $\varepsilon > 0$. Therefore, the total running time is $O(n^{3+\varepsilon})$. The construction maintains only a constant number of counters and indices in addition to the input and output arrays, and thus uses $O(1)$ extra space.

6 Conclusion

We have presented a constant-space algorithm that computes the BWT and the Lyndon array simultaneously. Our method extends the in-place BWT construction of Crochemore *et al.* [10] by maintaining the inverse suffix array incrementally and then deriving the Lyndon array. Although not intended for practical use, these results show constructions for these fundamental structures under the strongest space bounds. We also showed that the succinct $2n$-bit representation of the Lyndon array can be obtained directly from the BWT using constant space. Unlike the linear-time construction that relies on additional working memory [18,7], our procedure requires only constant extra space, at the cost of higher time complexity.

Acknowledgments

The first author was partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq), grants 408314/2023-0, 311128/2025-4 and 351599/2025-8, and by the São Paulo Research Foundation (FAPESP), process 2025/16208-7.

References

1. H. Bannai and J. Ellert: *Lyndon arrays in sublinear time*, in Proc. European Symposium on Algorithms, 2023, pp. 14:1–14:16.
2. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *A new characterization of maximal repetitions by Lyndon trees*. CoRR, abs/1406.0263 2014.
3. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *The "runs" theorem*. SIAM J. Comput., 46(5) 2017, pp. 1501–1514.
4. H. Bannai, J. Kärkkäinen, D. Köppl, and M. Piatkowski: *Constructing and indexing the bijective and extended Burrows-Wheeler transform*. Inf. Comput., 297 2024, p. 105153.
5. H. Bannai, D. Köppl, and Z. Lipták: *A survey of the bijective Burrows-Wheeler transform*, in The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday, Manzini's Festschrift, Venice, Italy, July 25, 2025, P. Ferragina, T. Gagie, and G. Navarro, eds., vol. 131 of OASIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, pp. 2:1–2:26.
6. D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen: *Linear-time string indexing and analysis in small space*. ACM Trans. Algorithms, 16(2) 2020, pp. 17:1–17:54.
7. P. Bille, J. Ellert, J. Fischer, I. L. Gørtz, F. Kurpicz, J. I. Munro, and E. Rotenberg: *Space efficient construction of Lyndon arrays in linear time*, in Proc. International Colloquium on Automata, Languages, and Programming, 2020, pp. 14:1–14:18.
8. M. Burrows and D. Wheeler: *A block-sorting lossless data compression algorithm*, tech. rep., Digital SRC Research Report, 1994.
9. K. T. Chen, R. H. Fox, and R. C. Lyndon: *Free differential calculus IV — the quotient groups of the lower central series*. Ann. Math., 68 1958, pp. 81–95.
10. M. Crochemore, R. Grossi, J. Kärkkäinen, and G. M. Landau: *Computing the Burrows-Wheeler transform in place and in small space*. J. Discret. Algorithms, 32 2015, pp. 44–52.
11. M. Crochemore and L. M. S. Russo: *Cartesian and lyndon trees*. Theor. Comput. Sci., 806 2020, pp. 1–9.
12. J. Duval: *Factorizing words over an ordered alphabet*. J. Algorithms, 4(4) 1983, pp. 363–381.
13. P. Ferragina and G. Manzini: *Opportunistic data structures with applications*, in Proc. Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society, 2000, pp. 390–398.
14. F. Franek, A. S. M. S. Islam, M. S. Rahman, and W. F. Smyth: *Algorithms to compute the Lyndon array*, in Proc. Prague Stringology Conference, 2016, pp. 172–184.
15. F. Franek, A. Paracha, and W. F. Smyth: *The linear equivalence of the suffix array and the partially sorted lyndon array*, in Proc. Prague Stringology Conference, 2017, pp. 77–84.

16. F. A. Louza, T. Gagie, and G. P. Telles: *Burrows-Wheeler transform and LCP array construction in constant space.* J. Discret. Algorithms, 42 2017, pp. 14–22.
17. F. A. Louza, S. Mantaci, G. Manzini, M. Sciortino, and G. P. Telles: *Inducing the Lyndon array*, in Proc. International Symposium on String Processing and Information Retrieval (SPIRE), 2019, pp. 138–151.
18. F. A. Louza, W. Smyth, G. Manzini, and G. P. Telles: *Lyndon array construction during Burrows–Wheeler inversion.* Journal of Discrete Algorithms, 50 2018, pp. 2–9.
19. U. Manber and E. W. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM J. Comput., 22(5) 1993, pp. 935–948.
20. J. I. Munro, G. Navarro, and Y. Nekrich: *Space-efficient construction of compressed indexes in deterministic linear time*, in Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19, P. N. Klein, ed., SIAM, 2017, pp. 408–424.
21. J. I. Munro and V. Raman: *Succinct representation of balanced parentheses and static trees.* SIAM J. Comput., 31(3) 2001, pp. 762–776.
22. D. S. Nunes, F. A. Louza, and G. P. Telles: *Space-efficient lyndon array construction from compressed texts.* Procedia Computer Science, 273 2025, pp. 413–420, XIII Latin American Algorithms, Graphs, and Optimization Symposium (LAGOS 2025).
23. D. Okanohara and K. Sadakane: *A linear-time Burrows-Wheeler transform using induced sorting*, in Proc. International Symposium on String Processing and Information Retrieval (SPIRE), 2009, pp. 90–101.

A Source Code

```
void bwt_lyndon_inplace(unsigned char *T, int n, int* LA){
  int i, j, p, r=1, s;
  unsigned char c;

  // base cases
  LA[n-2] = n-1;
  LA[n-1] = n-2;

  for (s=n-3; s>=0; s--) {
     c = T[s];

     /*steps 1 and 2*/
     r = s;
     for (i=s+1, T[i]!=END_MARKER; i++)
       if(T[i]<=c) r++;
     p = i;
     while (i<n)
       if (T[i++]<c) r++;

     /*step 3*/
     T[p]=c;

     /*step 4*/
     for (i=s; i<r; i++) {
       T[i]=T[i+1];
     }

     T[r]=END_MARKER;

     /*step 5*/
     for (i=s+1; i<n; i++) {
       if(LA[i]<=r) LA[i]--;
     }
     LA[s]= r;
  }
  /*ISA  to LA*/
  for (i=0; i<n; i++) {
    for(j=i+1; j<n; j++){
      if(LA[j]<LA[i])
        break;
    }
    LA[i]=j-i;
  }
}

```