https://doi.org/10.1016/j.jda.2018.08.001

Louza, F.A., Smyth, W.F., Manzini, G. and Telles, G.P. (2018) Lyndon Array construction during Burrows-Wheeler inversion. Journal of Discrete Algorithms

Lyndon Array Construction during Burrows-Wheeler Inversion

Felipe A. Louza, W.F. Smyth, Giovanni Manzini, Guilherme P. Telles

PII: S1570-8667(18)30125-4
DOI: https://doi.org/10.1016/j.jda.2018.08.001
Reference: JDA 700

To appear in: *Journal of Discrete Algorithms*

Received date: 31 October 2017
Revised date: 30 May 2018
Accepted date: 10 August 2018

Please cite this article in press as: F.A. Louza et al., Lyndon Array Construction during Burrows-Wheeler Inversion, *J. Discret. Algorithms* (2018), https://doi.org/10.1016/j.jda.2018.08.001

Lyndon Array Construction during Burrows-Wheeler Inversion

Felipe A. Louza[a,*], W. F. Smyth[b,c], Giovanni Manzini[d,e], Guilherme P. Telles[f]

[a]Department of Computing and Mathematics, University of São Paulo, Brazil
[b]Department of Computing and Software, McMaster University, Canada
[c]School of Engineering & Information Technology, Murdoch University, Australia
[d]Computer Science Institute, University of Eastern Piedmont, Italy
[e]Institute of Informatics and Telematics, CNR, Pisa, Italy
[f]Institute of Computing, University of Campinas, Brazil

Abstract

In this paper we present an algorithm to compute the Lyndon array of a string T of length n as a byproduct of the inversion of the Burrows-Wheeler transform of T. Our algorithm runs in linear time using only a stack in addition to the data structures used for Burrows-Wheeler inversion. We compare our algorithm with two other linear-time algorithms for Lyndon array construction and show that computing the Burrows-Wheeler transform and then constructing the Lyndon array is competitive compared to the known approaches. We also propose a new balanced parenthesis representation for the Lyndon array that uses $2n + o(n)$ bits of space and supports constant time access. This representation can be built in linear time using $O(n)$ words of space, or in $O(n \log n / \log\log n)$ time using asymptotically the same space as T.

Keywords: Lyndon array, Burrows-Wheeler inversion, linear time, compressed representation, balanced parentheses.

---

1. Introduction

Lyndon words were introduced to find bases of the free Lie algebra [1], and have been extensively applied in algebra and combinatorics. The term "Lyndon

---

[*] Corresponding author

Email addresses: `louza@usp.br` (Felipe A. Louza), `smyth@mcmaster.ca` (W. F. Smyth), `giovanni.manzini@uniupo.it` (Giovanni Manzini), `gpt@ic.unicamp.br` (Guilherme P. Telles)

array" was apparently introduced in [2], essentially equivalent to the "Lyndon tree" of Hohlweg & Reutenauer [3]. Interest in Lyndon arrays has been sparked by the surprising characterization of runs through Lyndon words by Bannai *et al.* [4], who were thus able to resolve the long-standing conjecture that the number of runs (maximal periodicities) in any string of length n is less than n.

The Burrows-Wheeler transform (BWT) [5] plays a fundamental role in data compression and in text indexing [6]. Embedded into a wavelet tree, the BWT is a self-index with a remarkable time/space tradeoff [7, 8]. The results on the BWT have spurred the new field of succinct or compressed data structures. In this field the challenge is to represent objects in compact space, possibly close to the information theoretic minimum, and still be able to efficiently query them [9, 10, 11].

In this article we introduce a linear time algorithm to construct the Lyndon array of a string T of length n, from an ordered alphabet of size σ, as a byproduct of Burrows-Wheeler inversion, thus establishing an apparently unremarked connection between BWT and Lyndon array construction. We experimentally compare our algorithm to other recently proposed algorithms that also compute the Lyndon array in worst-case linear time trying to give a complete picture of the available options for linear time Lyndon array computation.

Inspired by the inner working of our new algorithm, in the second part of the paper we propose a new representation of the Lyndon array consisting of a balanced parenthesis string of length $2n$. Such representation leads to a data structure of size $2n + o(n)$ bits, supporting the computation of each entry of the Lyndon array in constant time. We also show that such representation is theoretically appealing since it can be computed from T in $O(n)$ time using $O(n)$ words of space, or in $O(n \log n / \log\log n)$ time using $O(n \log \sigma)$ bits of space. To our knowledge ours is the first solution for the computation and representation of the Lyndon array in $o(n \log n)$ bits of space.

This article is organized as follows. Section 2 introduces concepts, notation and related work. Section 3 presents our construction algorithm and Section 4 shows some experimental results. Section 5 describes our balanced parenthesis representation of the Lyndon array and two construction algorithms with different time/space tradeoffs. Section 6 summarizes our conclusions.

2. Concepts, notation and related work

Let T be a string of length $|T| = n$ over an ordered alphabet Σ of size $\sigma = O(n)$. The i-th symbol of T is denoted by $T[i]$ and the substring $T[i]T[i+1]\cdots T[j]$ is denoted by $T[i,j]$, for $1 \leq i \leq j \leq n$. We assume that T always ends with a special symbol $T[n] = \$$, that doesn't appear elsewhere in T and lexicographically precedes every symbol in Σ. A prefix of T is a substring of the form $T[1,i]$ and a suffix is a substring of the form $T[i,n]$, which will be denoted by T_i. We use the symbol $\prec$ for the lexicographic order relation between strings.

The suffix array (SA) [12, 13] of a string $T[1,n]$ is an array containing the permutation of the integers in the range $[1,n]$ that gives the lexicographic order of the non-empty suffixes of T, i.e., $T[\mathsf{SA}[1],n] \prec T[\mathsf{SA}[2],n] \prec \cdots \prec T[\mathsf{SA}[n],n]$. We denote the inverse of SA as ISA, $\mathsf{ISA}[\mathsf{SA}[i]] = i$. The suffix array can be constructed in linear time using $O(\sigma)$ additional space [14].

The next smaller value array (NSV_A) defined for an array of integers $A[1,n]$ stores in $A[i]$ the position of the next value in $A[i+1,n]$ that is smaller than $A[i]$. If there is no value in $A[i+1,n]$ smaller than $A[i]$ then $\mathsf{NSV}_A[i] = n+1$. Formally, $\mathsf{NSV}_A[i] = \min(\{n+1\} \cup \{j | i < j \leq n \text{ and } A[j] < A[i]\})$. NSV may be constructed in linear time using additional memory for an auxiliary stack [15].

Lyndon array. A string T of length $n > 0$ is called a Lyndon word if it is lexicographically strictly smaller than its circular shifts [1]. Alternatively, if T is a Lyndon word and $T = uv$ is any factorization of T into non-empty strings, then $u \prec v$. The Lyndon array of a string T, denoted λ_T or simply λ when T is understood, has length $|T| = n$ and stores at each position i the length of the longest Lyndon word starting at $T[i]$.

Following [3], Franek *et al.* [2] have recently shown that the Lyndon array can be easily computed in linear time by applying the NSV computation to the inverse suffix array (ISA), such that $\lambda[i] = \mathsf{NSV}_{\mathsf{ISA}}[i] - i$, for $1 \leq i \leq$

n. Also, in a recent talk surveying Lyndon array construction, Franek and Smyth [16] quote an unpublished observation by Cristoph Diegelmann [17] that, in its first phase, the linear-time suffix array construction algorithm by Baier [18] computes a permuted version of the Lyndon array. This permuted version, called λ_{SA}, stores in $\lambda_{\mathsf{SA}}[i]$ the length of the longest Lyndon word starting at position $\mathsf{SA}[i]$ of T. Thus, including the BWT-based algorithm proposed here, there are apparently three algorithms that compute the Lyndon array in worst-case $O(n)$ time. In addition, in [4, Lemma 23] a linear-time algorithm is suggested that uses LCA/RMQ techniques to compute the Lyndon tree. The same paper also gives an algorithm for Lyndon tree calculation described as being "in essence" the same as NSV.

Burrows-Wheeler transform. The Burrows-Wheeler transform (BWT) [5, 19] is a reversible transformation that produces a permutation L of the original string T such that equal symbols of T tend to be clustered in L. The BWT can be obtained by adding each circular shift of T as a row of a conceptual matrix M', lexicographically sorting the rows of M' producing M, and concatenating the symbols in the last column of M to form L. Alternatively, the BWT can be obtained from the suffix array through the application of the relation $L[i] = T[SA[i]-1]$ if $SA[i] \neq 1$ or $L[i] = \$$ otherwise.

Burrows-Wheeler inversion, the processing of L to obtain T, is based on the LF-mapping (last-to-first mapping). Let c_F and c_L be the first and the last columns of the conceptual matrix M mentioned above. We have $LF : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ such that if $c_L[i] = \alpha$ is the k^{th} occurrence of a symbol α in c_L, then $LF(i) = j$ corresponds to the position $c_F[j]$ of the k^{th} occurrence of α in c_F.

The LF-mapping can be pre-computed in an array LF of integers in the range $[1, n]$. Given an array of integers C of length σ that stores in $C[\alpha]$ the number of symbols in T strictly smaller than α, LF can be computed in linear time using $O(\sigma \log n)$ bits of additional space [9, Alg. 7.2]. Alternatively, $LF(i)$ can be computed on-the-fly in $O(\log \sigma)$ time querying a wavelet tree [20] constructed

		sorted									sorted
	circular shifts	circular shifts									suffixes
i		F L	SA	ISA	$\mathsf{NSV}_{\mathsf{ISA}}$	LF	λ	λ_{SA}	L	$T[\mathsf{SA}[i], n]$
1	$\underline{\texttt{b}}\texttt{anana\$}$	`$banana`	7	5	2	2	1	1	`a`	$\underline{\texttt{\$}}$
2	$\underline{\texttt{an}}\texttt{ana\$b}$	`a$banan`	6	4	4	6	2	1	`n`	$\underline{\texttt{a}}\texttt{\$}$
3	$\underline{\texttt{n}}\texttt{ana\$ba}$	`ana$ban`	4	7	4	7	1	2	`n`	$\underline{\texttt{an}}\texttt{a\$}$
4	$\underline{\texttt{an}}\texttt{a\$ban}$	`anana$b`	2	3	6	5	2	2	`b`	$\underline{\texttt{an}}\texttt{ana\$}$
5	$\underline{\texttt{n}}\texttt{a\$bana}$	`banana$`	1	6	6	1	1	1	`$`	$\underline{\texttt{b}}\texttt{anana\$}$
6	$\underline{\texttt{a}}\texttt{\$banan}$	`na$bana`	5	2	7	3	1	1	`a`	$\underline{\texttt{n}}\texttt{a\$}$
7	$\underline{\texttt{\$}}\texttt{banana}$	`nana$ba`	3	1	8	4	1	1	`a`	$\underline{\texttt{n}}\texttt{ana\$}$

Figure 1: Circular shifts, sorted circular shifts, SA, ISA, $\mathsf{NSV}_{\mathsf{ISA}}$, LF, λ_{SA}, λ, L and the sorted suffixes of $T =$ `banana$`.

for c_L. Given the BWT L and the LF array, the Burrows-Wheeler inversion can be performed in linear time [9, Alg. 7.3].

Figure 1 shows the circular shifts, the sorted circular shifts, the arrays SA, ISA, $\mathsf{NSV}_{\mathsf{ISA}}$, LF, λ, λ_{SA}, the BWT L and the sorted suffixes of $T =$ `banana$`. The longest Lyndon words starting at each position i and $\mathsf{SA}[i]$ are underlined in the first and last columns of Figure 1.

3. From the BWT to the Lyndon array

Our starting point is the following characterization of the Lyndon array.

Lemma 1. *Let j be the smallest position in T after position $i < n$ such that suffix $T[j, n]$ is lexicographically smaller than suffix $T[i, n]$, that is, $j = \min\{k | i < k \leq n$ and $T[k, n] \prec T[i, n]\}$. Then the length of the longest Lyndon word starting at position i is $\lambda[i] = j - i$. If $i = n$ then $\lambda[i] = 1$.*

Proof. For $i < n$ let j be defined as above and let $w = T[i, j - 1]$. If $i = j - 1$ then $T[i]$ is a Lyndon word. If $w = uv$ then for h, $i < h < j$, let $u = T[i, h - 1]$ and $v = T[h, j-1]$. Since $h < j$ it follows that $uy = T[i, n] \succ T[h, n] = vx$, hence $u \prec v$ and $T[i, j - 1]$ is a Lyndon word. In addition, $T[j, j] \prec T[j, n] \prec T[i, n]$, hence $T[j] \leq T[i]$ and $T[i, j]$ is not a Lyndon word. □

The above lemma is at the basis of the algorithm by Franek *et al.* [2] computing $\lambda[i]$ as $\mathsf{NSV}_{\mathsf{ISA}}[i] - i$. Since $\mathsf{ISA}[i]$ is the lexicographic rank of $T[i,n]$, $j = \mathsf{NSV}_{\mathsf{ISA}}[i]$ is precisely the value used in the lemma. In this section, we use a known relationship between LF-mapping and ISA to design alternative algorithms for Lyndon array construction. Since $\mathsf{ISA}[n] = 1$, and $LF(\mathsf{ISA}[i]) = \mathsf{ISA}[i-1]$ it follows that $\mathsf{ISA}[i] = LF^{n-i}(\mathsf{ISA}[n])$ where LF^j denotes the LF map iterated j times.

Given the BWT L and the LF mapping our algorithm computes T and the Lyndon array λ from right to left. Briefly, our algorithm finds, during the Burrows-Wheeler inversion, for each position $i = n, n-1, \ldots, 1$, the first suffix $T[j,n]$ that is smaller than $T[i,n]$ and using Lemma 1 it computes $\lambda[i] = j - i$.

The complete pseudo-code appears in Algorithm 1. We remark that lines 1, 2, 7, 8, 15 and 16 are exactly the lines from the Burrows-Wheeler inversion presented in [9, Alg. 7.3]. Starting with $i = n$ and an index $pos = 1$ in the BWT, the algorithm decodes the BWT according to $LF(pos)$, keeping the visited positions whose indices are smaller than pos in a stack. The visited positions indicate the suffix ordering: a suffix visited at position i is lexicographically smaller than all suffixes visited at positions $j > i$. The stack stores pairs of integers $\langle pos, step\rangle$ corresponding to each visited position pos in iteration $step$. The stack is initialized by pushing $\langle -1, 0\rangle$.

An element $\langle pos, step\rangle$ in the stack represents the suffix $T[n - step + 1, n]$ visited in iteration $step$. At iteration $step$ the algorithm pops suffixes that are lexicographically larger than the current suffix $T[n - step + 1, n]$. Consequently, at the end of the while loop, the top element represents the next suffix (in text order) that is smaller than $T[n - step + 1, n]$ and $\lambda[step]$ is computed at line 12.

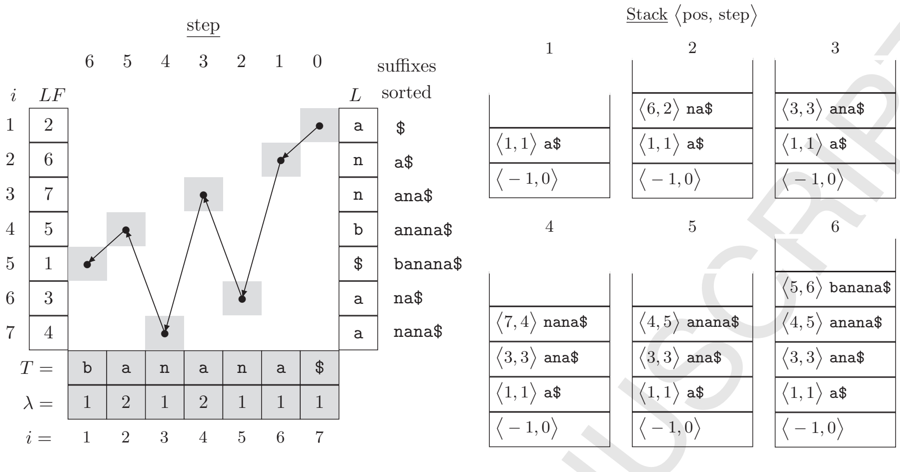
Example. Figure 2 shows a running example of our algorithm to compute the Lyndon array for string $T =$ `banana$` during its Burrows-Wheeler inversion. Before $step$ is set to 1 (lines 1–6) \$ is decoded at position n and the stack is initialized with the end-of-stack marker $\langle -1, 0\rangle$. The first loop iteration (lines 7–15) decodes `a` and finds out that the stack is empty. Then $\lambda[6] = 1$, the pair

Algorithm 1: Lyndon array construction during Burrows-Wheeler inversion

```
Data: $L[1,n]$ and $\mathsf{LF}[1,n]$
Result: $T[1,n]$ and $\lambda[1,n]$
$T[n] \leftarrow \$$
$pos \leftarrow 1$
$\lambda[n] \leftarrow 1$
$Stack \leftarrow \emptyset$
$Stack.push(\langle -1, 0\rangle)$
$step \leftarrow 1$
for $i \leftarrow n-1$ downto 1 do
    $T[i] \leftarrow L[pos]$
    while $Stack.top().pos > pos$ do
        $Stack.pop()$
    end
    $\lambda[i] \leftarrow step - Stack.top().step$
    $Stack.push(\langle pos, step\rangle)$
    $step \leftarrow step + 1$
    $pos \leftarrow LF[pos]$
end
```

$\langle 1, 1\rangle$ is pushed on the stack and $pos = LF[1] = 2$.

At the second iteration `n` is decoded and the algorithm checks if the suffix at the top of the stack (`a$`) is larger then the current suffix (`na$`). The algorithm does not pop the stack because there is no suffix lexicographically larger than the current one. Then $\lambda[5] = step - Stack.top().step = 2 - 1 = 1$. The pair $\langle 6, 2\rangle$ is pushed on the stack. At the third iteration `a` is decoded. The top element, representing suffix `na$`, is popped since it is larger then the current suffix `ana$`. Then $\lambda[4] = step - Stack.top().step = 3 - 1 = 2$ and the pair $\langle 3, 3\rangle$ is pushed. The next iterations proceed in a similar fashion.



(a) (b)

Figure 2: Example of our algorithm in the string $T =$ `banana$`. Part (a) shows the algorithms steps from right to left. The arrows illustrate the order in which suffixes are visited by the algorithm, following the LF-mapping. Part (b) shows the Stack and the corresponding suffixes at the end of each step of the algorithm.

Lemma 2. *Algorithm 1 computes the text $T[1, n]$ and its Lyndon array $\lambda[1, n]$ in $\Theta(n)$ time using $O(n)$ words of space.*

Proof. Since each instruction takes constant time, the running time is proportional to the number of stack operations, which is $O(n)$ since each text position is added to the stack exactly once. The space usage is dominated by the arrays LF, λ, and by the stack that use $O(n)$ words in total. □

4. Experiments

In this section we compare our algorithm with the linear time algorithms of Hohlweg and Reutenauer [3, 2] (NSV-Lyndon) and Baier [18] (Baier-Lyndon). Although the natural use of our algorithm is to compute the Lyndon array given the BWT, for this comparison all algorithms were adapted to compute only the Lyndon array λ given an input string $T[1, n]$. To compare our solution with the others, we compute the suffix array SA for the input string T, then we obtain

L and the LF array, and finally we construct the Lyndon array during Burrows-Wheeler inversion (Algorithm 1). This procedure will be called BWT-Lyndon. We used algorithm SACA-K [14] to construct SA in $O(n)$ time using $O(\sigma)$ working space. $\lambda[1, n]$ was computed in the same space as SA$[1, n]$ (overwriting the values) both in BWT-Lyndon and in NSV-Lyndon.

We implemented all algorithms in ANSI C. The source code is publicly available at `https://github.com/felipelouza/lyndon-array`. The experiments were executed on a 64-bit Debian GNU/Linux 8 (kernel 3.16.0-4) system with an Intel Xeon Processor E5-2630 v3 20M Cache 2.40-GHz, 384 GB of internal memory and a 13 TB SATA storage. The sources were compiled by GNU GCC version 4.9.2, with the optimizing option -O3 for all algorithms. The time was measured using the `clock()` function of C standard libraries and the peak memory usage was measured using malloc_count library[1].

We used string datasets from Pizza & Chili[2] as shown in the first three columns of Tables 1 and 2. The datasets `einstein-de`, `kernel`, `fib41` and `cere` are highly repetitive texts. The dataset `english.1gb` is the first 1GB of the original English dataset. In our experiments, each integer array of length n is stored using $4n$ bytes, and each string of length n is stored using n bytes.

Table 1 shows the running time (in seconds), the peak space memory (in bytes per input symbol) and the working space (in GB) of each algorithm.

Running time. The fastest algorithm was Baier-Lyndon, which overall spent about two-thirds of the time required by BWT-Lyndon, though the timings were much closer for larger alphabets. NSV-Lyndon was slightly faster than BWT-Lyndon, requiring about 81% of the time spent by BWT-Lyndon on average.

Peak space. The smallest peak space was obtained by BWT-Lyndon and NSV-Lyndon, which both use slightly more than $9n$ bytes. BWT-Lyndon uses $9n$ bytes to store the string T and the integer arrays SA and LF, plus the space

---

[1] `http://panthema.net/2013/malloc_count`

[2] `https://pizzachili.dcc.uchile.cl/`

used by the stack, which occupied about 11 KB in all experiments, except for dataset cere, in which the stack used 261 KB. The strings $L[1, n]$ and $T[1, n]$ are computed and decoded in the same space. NSV-Lyndon also requires $9n$ bytes to store the string T and the integer arrays SA and ISA, that plus the space of the stack used to compute NSV [15], which used exactly the same amount of memory used by the stack of BWT-Lyndon. The array NSV is computed in the same space as ISA. Baier-Lyndon uses $17n$ bytes to store T, λ and three auxiliary integer arrays of size n.

Working space. The working space is the peak space not counting the space used by the input string $T[1, n]$ and the output array $\lambda[1, n]$ ($5n$ bytes). The working space of BWT-Lyndon and NSV-Lyndon were by far the smallest in all experiments. Both algorithms use about 41% of the working space used by Baier-Lyndon. For dataset `proteins`, BWT-Lyndon and NSV-Lyndon use 7.72 GB less memory than Baier-Lyndon.

Steps (running time). Table 2 shows the running time (in seconds) for each step of algorithms BWT-Lyndon and NSV-Lyndon. Step 1, constructing SA, is the most time-consuming part of both algorithms, taking about 80% of the total time. Incidentally, this means that if the input consists of the BWT rather than T, our algorithm would clearly be the fastest. In Step 2, computing BWT is faster than computing ISA since $L[i] = T[\mathsf{SA}[i] - 1]$ is more cache-efficient than $\mathsf{ISA}[\mathsf{SA}[i]] = i$. Similarly in Step 3, computing LF is more efficient than computing NSV [15]. However, Step 4 of BWT-Lyndon, which computes λ during the Burrows-Wheeler inversion, is sufficiently slower (by a factor of 10^2) than computing λ from ISA and NSV, so that the overall time of BWT-Lyndon is larger than NSV-Lyndon, as shown in Table 1.

5. Balanced parenthesis representation of a Lyndon Array

In this section we introduce a new representation for the Lyndon array $\lambda[1, n]$ of $T[1, n]$ consisting of a balanced parenthesis string of length $2n$. The existence

Table 1: Experiments with Pizza & Chili datasets. The datasets `einstein-de`, `kernel`, `fib41` and cere are highly repetitive texts. The running time is shown in seconds. The peak space is given in bytes per input symbol. The working space is given in GB.

			running time [secs]			peak space [bytes/n]			working space [GB]		
	σ	$n/2^{20}$	BWT-Lyndon	NSV-Lyndon	Baier-Lyndon	BWT-Lyndon	NSV-Lyndon	Baier-Lyndon	BWT-Lyndon	NSV-Lyndon	Baier-Lyndon
`sources`	230	201	68	55	**57**	**9**	**9**	17	**0.79**	**0.79**	2.36
`dblp`	97	282	104	87	**90**	**9**	**9**	17	**1.10**	**1.10**	3.31
`dna`	16	385	198	160	**113**	**9**	**9**	17	**1.50**	**1.50**	4.51
`english.1gb`	239	1,047	614	504	**427**	**9**	**9**	17	**4.09**	**4.09**	12.27
`proteins`	27	1,129	631	524	**477**	**9**	**9**	17	**4.41**	**4.41**	13.23
`einstein-de`	117	88	36	32	**25**	**9**	**9**	17	**0.35**	**0.35**	1.04
`kernel`	160	246	100	75	**73**	**9**	**9**	17	**0.96**	**0.96**	2.88
`fib41`	2	256	120	93	**18**	**9**	**9**	17	**1.00**	**1.00**	2.99
`cere`	5	440	215	169	**114**	**9**	**9**	17	**1.72**	**1.72**	5.16

of this representation is not completely surprising in view of Observation 3 in [2] stating that Lyndon words do not overlap (see also the bracketing algorithm in [21]). Nevertheless, it was the inner working of the stack based algorithm of Section 3 that naturally suggested us such representation. Algorithm 2 gives an operational strategy for building such representation, and the next lemma shows how to use it to retrieve individual values of λ. In the following, given a balanced parenthesis string S, we write $\mathsf{selectopen}(S, i)$ (resp. $\mathsf{selectclose}(S, i)$) to denote the position in S of the i-th open parenthesis (resp. the position in S of the closed parenthesis closing the i-th open parenthesis).

Lemma 3. *The balanced parenthesis array λ_{BP} computed by Algorithm 2 is such that setting for $i = 1, \ldots, n$*

$$o_i = \mathsf{selectopen}(\lambda_{BP}, i), \qquad c_i = \mathsf{selectclose}(\lambda_{BP}, i) \tag{1}$$

Table 2: Experiments with Pizza & Chili datasets. The running time is reported in seconds for each step of algorithms BWT-Lyndon and NSV-Lyndon.

			Step 1	Step 2		Step 3		Step 4	
				BWT-Lyndon	NSV-Lyndon	BWT-Lyndon	NSV-Lyndon	BWT-Lyndon	NSV-Lyndon
	σ	$n/2^{20}$	SA	BWT	ISA	LF	NSV	λ	λ
`sources`	230	201	50.27	**2.28**	3.49	**0.97**	1.65	14.12	**0.13**
`dblp`	97	282	79.83	**4.02**	5.51	**1.46**	1.61	18.80	**0.18**
`dna`	16	385	145.02	**8.07**	9.99	**1.48**	4.04	43.39	**0.25**
`english.1gb`	239	1,047	459.29	**21.86**	34.35	**4.70**	10.31	127.65	**0.72**
`proteins`	27	1,129	478.13	**21.96**	34.99	**4.71**	10.47	125.73	**0.75**
`einstein-de`	117	88	28.79	**1.27**	1.91	**0.48**	0.85	5.10	**0.06**
`kernel`	160	246	68.19	**3.52**	4.92	**1.29**	2.18	27.21	**0.18**
`fib41`	2	256	85.94	**5.94**	6.55	**1.38**	0.60	26.48	**0.18**
`cere`	5	440	153.89	**9.30**	11.20	**2.24**	4.38	49.38	**0.42**

then

$$\lambda[i] = (c_i - o_i + 1)/2 \tag{2}$$

Proof. First note that at the i-th iteration we append an open parenthesis to λ_{BP} and add the value $\mathsf{ISA}[i]$ to the stack. The value $\mathsf{ISA}[i]$ is removed from the stack as soon as a smaller element $\mathsf{ISA}[j] < \mathsf{ISA}[i]$ is encountered. Since the last value $\mathsf{ISA}[n] = 1$ is the smallest element, at the end of the for loop the stack only contains the value 1, which is removed at the exit of the loop. Observing that we append a closed parenthesis to λ_{BP} every time a value is removed from the stack, at the end of the algorithm λ_{BP} indeed contains n open and n closed parentheses. Because of the use of the stack, the closing parenthesis follow a first-in last-out logic so the parenthesis are balanced.

By construction, for $i < n$, the closed parenthesis corresponding to $\mathsf{ISA}[i]$ is written immediately before the open parenthesis corresponding to $\mathsf{NSV}_{\mathsf{ISA}}[i]$. Hence, between the open and closed parenthesis corresponding to $\mathsf{ISA}[i]$ there is a pair of open/closed parenthesis for each entry k, $i < k < \mathsf{NSV}_{\mathsf{ISA}}[i]$. Hence,

Algorithm 2: Balanced parenthesis representation λ_{BP} from ISA

```
$\lambda_{BP} \leftarrow \varepsilon$
$Stack \leftarrow \emptyset$
for $i \leftarrow 1$ to $n$ do
    while $Stack.top() > \mathsf{ISA}[i]$ do
        $Stack.pop()$
        $\lambda_{BP}.append($ ")" $)$
    end
    $Stack.push(\mathsf{ISA}[i])$
    $\lambda_{BP}.append($ "(" $)$
end
$Stack.pop()$
$\lambda_{BP}.append($ ")" $)$
```

using the notation (1) and Lemma 1 it is

$$c_i - o_i - 1 \;=\; 2(\mathsf{NSV}_{\mathsf{ISA}}[i] - \mathsf{ISA}[i] - 1) \;=\; 2(\lambda[i] - 1).$$

which implies (2). Finally, for $i = n$ we have $o_n = 2n - 1$ and $c_n = 2n$, so $(c_n - o_n + 1)/2 = \lambda[n] = 1$ and the lemma follows. □

Using the range min-max tree from [22] we can represent λ_{BP} in $2n + o(n)$ bits of space and support selectopen, and selectclose in $O(1)$ time. We have therefore established the following result.

Theorem 1. *It is possible to represent the Lyndon array for a text $T[1, n]$ in $2n + o(n)$ bits such that we can retrieve every value $\lambda[i]$ in $O(1)$ time.* □

Since the new representation takes $O(n)$ bits, it is desirable to build it without storing explicitly ISA, which takes $\Theta(n)$ words. We do this borrowing the main idea from the BWT-Lyndon algorithm. In Section 3 we used the LF map to generate the ISA values right-to-left (from $\mathsf{ISA}[n]$ to $\mathsf{ISA}[1]$) from the BWT. Since in Algorithm 2 we need to generate the ISA values left-to-right, we use the inverse permutation of the LF map, known in the literature as the Ψ map. Formally, for $i = 1, \ldots, n$ $\Psi[i]$ is defined as

$$\Psi[i] = \begin{cases} \mathsf{ISA}[1] & \text{if } i = 1 \\ \mathsf{ISA}(\mathsf{SA}[i] + 1) & \text{otherwise.} \end{cases} \tag{3}$$

Lemma 4. *Assume we have a data structure supporting the* select *operation on the BWT in $O(s)$ time. Then, we can generate the values $\mathsf{ISA}[1], \ldots, \mathsf{ISA}[n]$ in $O(sn)$ time using additional $O(\sigma \log n)$ bits of space.*

Proof. By (3) it follows that $\mathsf{ISA}[1] = \Psi(1)$ and, for $i = 2, \ldots, n$, $\mathsf{ISA}[i] = \Psi(\mathsf{ISA}[i-1])$. To prove the lemma we need to show how to compute each $\Psi(i)$ in $O(s)$ time. By definition, $\Psi(i)$ is the position in L of the character prefixing row i in the conceptual matrix defining the BWT. Let $F[1, n]$ denote the binary array such that $F[j] = 1$ iff row j is the first row of the BWT matrix prefixed by some character c. Then, the character prefixing row i is given by $c_i = \mathsf{rank}_1(F, i)$ and

$$\Psi(i) = \mathsf{select}_{c_i}(L, i - \mathsf{select}_1(F, c_i) + 1).$$

The thesis follows observing that using [23] we can represent F in $\log \binom{n}{\sigma} + o(\sigma) + o(\log \log n) = O(\sigma \log n)$ bits supporting constant time rank/select queries. □

Lemma 5. *Using Algorithm 2 we can compute λ_{BP} from the BWT in $O(n)$ time using $O(n)$ words of space.*

Proof. We represent L using one of the many available data structures taking $O(n \log \sigma)$ bits and supporting constant time select queries (see [24] and references therein). By Lemma 4 we can generate the values $\mathsf{ISA}[1], \ldots, \mathsf{ISA}[n]$ in $O(n)$ overall time using $O(\sigma \log n)$ auxiliary space. Since every other operation takes constant time, the running time is proportional to the number of stack operations which is $O(n)$ since each $\mathsf{ISA}[i]$ is inserted only once in the stack. □

Note that the space usage of Algorithm 2 is dominated by the stack. Since at any given time the stack contains an increasing subsequence of ISA, if ISA were a random permutation the average stack would be $O(\sqrt{n})$ words (see [25]). Unfortunately, in the worst case, for example for $T = \mathsf{a}^{n-2}\mathsf{b\$}$, the stack may contain $n-1$ words. For this reason we now present an alternative representation for the stack that only uses $n + o(n)$ bits in the worst case and supports pop and push operations in $O(\log n/\log\log n)$ time. We represent the stack with a binary array $S[1,n]$ such that $S[1] = 1$ iff the value i is currently in the stack. Since the values in the stack are always in increasing order, S is sufficient to represent the current status of the stack. In Algorithm 2 when a new element e is added to the stack we must first delete the elements larger than e. This can be accomplished using rank/select operations. If $r_e = \mathsf{rank}_1(S, e)$ the elements to be deleted are those returned by $\mathsf{select}_1(S, r_e + i)$ for $i = 1, 2, \ldots, \mathsf{rank}_1(S, n) - r_e$. Summing up, the binary array S must support the rank/select operations in addition to changing the value of a single bit. To this end we use the dynamic array representation described in [26] which takes $n + o(n)$ bits and support the above operations in (optimal) $O(\log n/\log\log n)$ time. We have therefore established, this new time/space tradeoff for Lyndon array construction.

Lemma 6. *Using Algorithm 2 we can compute* λ_{BP} *from the BWT in* $O(n \log n/\log\log n)$ *time using* $O(n \log\sigma)$ *bits of space.* □

Finally, we point out that if the input consists of the text $T[1,n]$ the asymptotic costs do not change, since we can build the BWT from T in $O(n)$ time and $O(n\log\sigma)$ bits of space [27].

Theorem 2. *Given* $T[1,n]$ *we can compute* λ_{BP} *in* $O(n)$ *time using* $O(n)$ *words of space, or in* $O(n \log n/\log\log n)$ *time using* $O(n\log\sigma)$ *bits of space.* □

6. Summary of Results

In this paper we have described a previously unknown connection between the Burrows-Wheeler transform and the Lyndon array, and proposed a corresponding algorithm to construct the latter during Burrows-Wheeler inversion.

The algorithm is guaranteed linear-time and simple, resulting in the good practical performance shown by the experiments. If the input is stored in a BWT-based self index, our algorithm would have a clear advantage in both working space and running time, since it is the only one that uses the LF-map rather than the suffix array.

Our algorithm also inspired a new balanced parenthesis representation for the Lyndon array using $2n + o(n)$ bits supporting $O(1)$ time access. We have shown how to build this representation in linear time using $O(n)$ words of space, and in $O(n \log n / \log \log n)$ time using asymptotically the same space as T.

Over all the known algorithms surveyed in [16], probably the fastest for real world datasets and the most space-efficient is the folklore MaxLyn algorithm described in [2], which makes no use of suffix arrays and requires only constant additional space, but which however requires $\Theta(n^2)$ time in the worst-case. We tested MaxLyn on a string consisting of 10×2^{20} symbols 'a'. While the linear-time algorithms run in no more than 0.5 seconds, MaxLyn takes about 8 hours to compute the Lyndon array. Thus, the challenge that remains is to find a fast and "lightweight" worst-case linear-time algorithm for computing the Lyndon array that avoids the expense of suffix array construction.

Acknowledgments

The authors thank Uwe Baier for kindly providing the source code of algorithm Baier-Lyndon, and Prof. Nalvo Almeida for granting access to the machine used for the experiments. We also thank the anonymous reviewers for comments that improved the manuscript.

Funding. F.A.L. was supported by the grant #2017/09105-0 from the São Paulo Research Foundation (FAPESP). The work of W.F.S. was supported in part by a grant from the Natural Sciences & Engineering Research Council of Canada (NSERC). G.M. was supported by the PRIN grant 201534HNXC and INdAM-GNCS Project *Efficient algorithms for the analysis of Big Data*. G.P.T. acknowledges the support of CNPq.

References

[1] K. T. Chen, R. H. Fox, R. C. Lyndon, Free differential calculus IV — the quotient groups of the lower central series, Ann. Math. 68 (1958) 81–95.

[2] F. Franek, A. S. M. S. Islam, M. S. Rahman, W. F. Smyth, Algorithms to compute the Lyndon Array, in: Proc. Prague Stringology Conference (PSC), Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2016, pp. 172–184.

[3] C. Hohlweg, C. Reutenauer, Lyndon words, permutations and trees, Theor. Comput. Sci. 307 (1) (2003) 173–178.

[4] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, K. Tsuruta, A new characterization of maximal repetitions by Lyndon trees, CoRR abs/1406.0263.

[5] M. Burrows, D. Wheeler, A block-sorting lossless data compression algorithm, Tech. rep., Digital SRC Research Report (1994).

[6] P. Ferragina, G. Manzini, Indexing compressed text, Journal of the ACM 52 (4) (2005) 552–581.

[7] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Compressed representations of sequences and full-text indexes, ACM Transactions on Algorithms 3 (2).

[8] R. Grossi, J. S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, SIAM Journal on Computing 35 (2) (2005) 378–407.

[9] E. Ohlebusch, Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction, Oldenbusch Verlag, 2013.

[10] V. Mäkinen, D. Belazzougui, F. Cunial, A. I. Tomescu, Genome-Scale Algorithm Design, Cambridge University Press, 2015.

[11] G. Navarro, Compact Data Structures – A practical approach, Cambridge University Press, 2016.

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

[13] G. H. Gonnet, R. A. Baeza-Yates, T. Snider, New indices for text: Pat trees and pat arrays, in: Information Retrieval, Prentice-Hall, Inc., 1992, pp. 66–82.

[14] G. Nong, Practical linear-time O(1)-workspace suffix sorting for constant alphabets, ACM Trans. Inform. Syst. 31 (3) (2013) 1–15.

[15] K. Goto, H. Bannai, Simpler and faster Lempel Ziv factorization, in: Proc. IEEE Data Compression Conference (DCC), IEEE, 2013, pp. 133–142.

[16] F. Franek, W. F. Smyth, Unexpected equivalence: Lyndon array & suffix array, Talk at London Stringology Days & London Algorithmic Workshop (2017).

[17] C. Diegelmann, Personal communication (2016).

[18] U. Baier, Linear-time suffix sorting - a new approach for suffix array construction, in: Proc. Annual Symposium on Combinatorial Pattern Matching (CPM), Vol. 78 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, pp. 23:1–23:12.

[19] D. Adjeroh, T. Bell, A. Mukherjee, The Burrows-Wheeler transform: data compression, suffix arrays, and pattern matching, Springer, Boston, MA, 2008.

[20] R. Grossi, A. Gupta, J. S. Vitter, High-order entropy-compressed text indexes, in: Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, 2003, pp. 841–850.

[21] J. Sawada, F. Ruskey, Generating Lyndon brackets.: An addendum to: Fast algorithms to generate necklaces, unlabeled necklaces and irreducible polynomials over GF(2), J. Algorithms 46 (1) (2003) 21–26.

[22] G. Navarro, K. Sadakane, Fully-functional static and dynamic succinct trees, ACM Transactions on Algorithms 10 (3) (2014) article 16.

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

[24] J. Barbay, F. Claude, T. Gagie, G. Navarro, Y. Nekrich, Efficient fully-compressed sequence representations, Algorithmica 69 (1) (2014) 232–268.

[25] D. Aldous, P. Diaconis, Longest increasing subsequences: from patience sorting to the Baik-Deift-Johansson theorem, Bull. Amer. Math. Soc. 36 (1999) 413–432.

[26] M. He, J. I. Munro, Succinct representations of dynamic strings, in: Proc. International Symposium on String Processing and Information Retrieval (SPIRE), Springer, 2010, pp. 334–346.

[27] J. I. Munro, G. Navarro, Y. Nekrich, Space-efficient construction of compressed indexes in deterministic linear time, in: Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, 2017, pp. 408–424.