arXiv:1905.12987v2 [cs.DS] 26 Jul 2019

Inducing the Lyndon Array

Felipe A. Louza[1], Sabrina Mantaci[2], Giovanni Manzini[3,4], Marinella Sciortino[2], and Guilherme P. Telles[5]

[1] Department of Computing and Mathematics, University of São Paulo, Brazil
louza@usp.br
[2] Dipartimento di Matematica e Informatica, University of Palermo, Italy
{sabrina.mantaci,marinella.sciortino}@unipa.it
[3] University of Eastern Piedmont, Alessandria, Italy
[4] IIT CNR, Pisa, Italy
giovanni.manzini@uniupo.it
[5] Institute of Computing, University of Campinas, Brazil
gpt@ic.unicamp.br

Abstract. In this paper we propose a variant of the induced suffix sorting algorithm by Nong (TOIS, 2013) that computes simultaneously the Lyndon array and the suffix array of a text in $O(n)$ time using $\sigma+O(1)$ words of working space, where n is the length of the text and σ is the alphabet size. Our result improves the previous best space requirement for linear time computation of the Lyndon array. In fact, all the known linear algorithms for Lyndon array computation use suffix sorting as a preprocessing step and use $O(n)$ words of working space in addition to the Lyndon array and suffix array. Experimental results with real and synthetic datasets show that our algorithm is not only space-efficient but also fast in practice.

Keywords: Lyndon array, Suffix array, induced suffix sorting, lightweight algorithms

1 Introduction

The suffix array is a central data structure for string processing. Induced suffix sorting is a remarkably powerful technique for the construction of the suffix array. Induced sorting was introduced by Itoh and Tanaka [10] and later refined by Ko and Aluru [11] and by Nong *et al.* [19,18]. In 2013, Nong [17] proposed a space efficient linear time algorithm based on induced sorting, called SACA-K, which uses only $\sigma + O(1)$ words of working space, where σ is the alphabet size and the working space is the space used in addition to the input and the output. Since a small working space is a very desirable feature, there have been many algorithms adapting induced suffix sorting to the computation of data structures related to the suffix array, such as the Burrows-Wheeler transform [21], the Φ-array [8], the LCP array [4,14], and the document array [13].

The Lyndon array of a string is a powerful tool that generalizes the idea of Lyndon factorization. In the Lyndon array (LA) of string $T = T[1] \dots T[n]$ over

the alphabet Σ, each entry $\mathsf{LA}[i]$, with $1 \leq i \leq n$, stores the length of the longest Lyndon factor of T starting at that position i. Bannai *et al.* [2] used Lyndon arrays to prove the conjecture by Kolpakov and Kucherov [12] that the number of runs (maximal periodicities) in a string of length n is smaller than n. In [3] the authors have shown that the computation of the Lyndon array of T is strictly related to the construction of the Lyndon tree [9] of the string $\$T$ (where the symbol \$ is smaller than any symbol of the alphabet Σ).

In this paper we address the problem of designing a space economical linear time algorithm for the computation of the Lyndon array. As described in [5,15], there are several algorithms to compute the Lyndon array. It is noteworthy that the ones that run in linear time (cf. [1,3,5,6,15]) use the sorting of the suffixes (or a partial sorting of suffixes) of the input string as a preprocessing step. Among the linear time algorithms, the most space economical is the one in [5] which, in addition to the $n \log \sigma$ bits for the input string plus $2n$ words for the Lyndon array and suffix array, uses a stack whose size depends on the structure of the input. Such stack is relatively small for non pathological texts, but in the worst case its size can be up to n words. Therefore, the overall space in the worst case can be up to $n \log \sigma$ bits plus $3n$ words.

In this paper we propose a variant of the algorithm SACA-K that computes in linear time the Lyndon array as a by-product of suffix array construction. Our algorithm uses overall $n \log \sigma$ bits plus $2n + \sigma + O(1)$ words of space. This bound makes our algorithm the one with the best worst case space bound among the linear time algorithms. Note that the $\sigma + O(1)$ words of working space of our algorithm is optimal for strings from alphabets of constant size. Our experiments show that our algorithm is competitive in practice compared to the other linear time solutions to compute the Lyndon array.

2 Background

Let $T = T[1] \ldots T[n]$ be a string of length n over a fixed ordered alphabet Σ of size σ, where $T[i]$ denotes the i-th symbol of T. We denote $T[i, j]$ as the factor of T starting from the i-th symbol and ending at the j-th symbol. A suffix of T is a factor of the form $T[i, n]$ and is also denoted as T_i. In the following we assume that any integer array of length n with values in the range $[1, n]$ takes n words ($n \log n$ bits) of space.

Given $T = T[1] \ldots T[n]$, the *i-th rotation* of T begins with $T[i + 1]$, corresponding to the string $T' = T[i + 1] \ldots T[n]T[1] \ldots T[i]$. Note that, a string of length n has n possible rotations. A string T is a *repetition* if there exists a string S and an integer $k > 1$ such that $T = S^k$, otherwise it is called *primitive*. If a string is primitive, all of its rotations are different.

A primitive string T is called a *Lyndon word* if it is the lexicographical least among its rotations. For instance, the string $T = abanba$ is not a Lyndon word, while it is its rotation $aabanb$ is. A *Lyndon factor* of a string T is a factor of T that is a Lyndon word. For instance, anb is a Lyndon factor of $T = abanba$.

Definition 1. *Given a string $T = T[1] \dots T[n]$, the Lyndon array (LA) of T is an array of integers in the range $[1, n]$ that, at each position $i = 1, \dots, n$, stores the length of the longest Lyndon factor of T starting at i:*

$$\mathsf{LA}[i] = \max\{\ell \mid T[i, i+\ell-1] \text{ is a Lyndon word}\}.$$

The suffix array (SA) [16] of a string $T = T[1] \dots T[n]$ is an array of integers in the range $[1, n]$ that gives the lexicographic order of all suffixes of T, that is $T_{\mathsf{SA}[1]} < T_{\mathsf{SA}[2]} < \dots < T_{\mathsf{SA}[n]}$. The inverse suffix array (ISA) stores the inverse permutation of SA, such that $\mathsf{ISA}[\mathsf{SA}[i]] = i$. The suffix array can be computed in $O(n)$ time using $\sigma + O(1)$ words of working space [17].

Usually when dealing with suffix arrays it is convenient to append to the string T a special end-marker symbol \$ (called sentinel) that does not occur elsewhere in T and \$ is smaller than any other symbol in Σ. Here we assume that $T[n] = \$$. Note that the values $\mathsf{LA}[i]$, for $1 \le i \le n-1$ do not change when the symbol \$ is appended at the position n. Also, string $T = T[1] \dots T[n-1]\$$ is always primitive.

Given an array of integers A of size n, the next smaller value (NSV) array of A, denoted $\mathsf{NSV_A}$, is an array of size n such that $\mathsf{NSV_A}[i]$ contains the smallest position $j > i$ such that $\mathsf{A}[j] < \mathsf{A}[i]$, or $n+1$ if such a position j does not exist. Formally:

$$\mathsf{NSV_A}[i] = \min\big\{\{n+1\} \cup \{i < j \le n \mid \mathsf{A}[j] < \mathsf{A}[i]\}\big\}.$$

As an example, in Figure 1 we consider the string $T = banaananaanana\$$, and its Suffix Array (SA), Inverse Suffix Array (ISA), Next Smaller Value array of the ISA ($\mathsf{NSV_{ISA}}$), and Lyndon Array (LA). We also show all the Lyndon factors starting at each position of T.

If the SA of T is known, the Lyndon array LA can be computed in linear time thanks to the following lemma that rephrases a result in [9]:

Lemma 1. *The factor $T[i, i+\ell-1]$ is the longest Lyndon factor of T starting at i iff $T_i < T_{i+k}$, for $1 \le k < \ell$, and $T_i > T_{i+\ell}$. Therefore, $\mathsf{LA}[i] = \ell$.* □

Lemma 1 can be reformulated in terms of the inverse suffix array [5], such that $\mathsf{LA}[i] = \ell$ iff $\mathsf{ISA}[i] < \mathsf{ISA}[i+k]$, for $1 \le k < \ell$, and $\mathsf{ISA}[i] > \mathsf{ISA}[i+\ell]$. In other words, $i + \ell = \mathsf{NSV_{ISA}}[i]$. Since given ISA we can compute $\mathsf{NSV_{ISA}}$ in linear time using an auxiliary stack [7,20] of size $O(n)$ words, we can then derive LA, in the same space of $\mathsf{NSV_{ISA}}$, in linear time using the formula:

$$\mathsf{LA}[i] = \mathsf{NSV_{ISA}}[i] - i, \text{ for } 1 \le i \le n. \tag{1}$$

Overall, this approach uses $n \log \sigma$ bits for T plus $2n$ words for LA and ISA, and the space for the auxiliary stack.

Alternatively, LA can be computed in linear time from the Cartesian tree [22] built for ISA [3]. Recently, Franek *et al.* [6] observed that LA can be computed in linear time during the suffix array construction algorithm by Baier [1] using overall $n \log \sigma$ bits plus $2n$ words for LA and SA plus $2n$ words for auxiliary

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
SA =	15	14	9	4	12	7	2	10	5	1	13	8	3	11	6
ISA =	10	7	13	4	9	15	6	12	3	8	14	5	11	2	1
$\mathsf{NSV_{ISA}}$ =	2	4	4	9	7	7	9	9	14	12	12	14	14	15	16
LA =	1	2	1	5	2	1	2	1	5	2	1	2	1	1	1

Fig. 1. SA, ISA, $\mathsf{NSV_{ISA}}$, LA and all Lyndon factors for $T = banaananaanana\$$

integer arrays. Finally, Louza *et al.* [15] introduced an algorithm that computes LA in linear time during the Burrows-Wheeler inversion, using $n \log \sigma$ bits for T plus $2n$ words for LA and an auxiliary integer array, plus a stack with twice the size as the one used to compute $\mathsf{NSV_{ISA}}$ (see Section 4).

Summing up, the most economical linear time solution for computing the Lyndon array is the one based on (1) that requires, in addition to T and LA, n words of working space plus an auxiliary stack. The stack size is small for non pathological inputs but can use n words in the worst case (see also Section 4). Therefore, considering only LA as output, the working space is $2n$ words in the worst case.

2.1 Induced Suffix Sorting

The algorithm SACA-K [17] uses a technique called induced suffix sorting to compute SA in linear time using only $\sigma + O(1)$ words of working space. In this technique each suffix T_i of $T[1,n]$ is classified according to its lexicographical rank relative to T_{i+1}.

Definition 2. *A suffix T_i is S-type if $T_i < T_{i+1}$, otherwise T_i is L-type. We define T_n as S-type. A suffix T_i is LMS-type (leftmost S-type) if T_i is S-type and T_{i-1} is L-type.*

The type of each suffix can be computed with a right-to-left scanning of T [18], or otherwise it can be computed on-the-fly in constant time during Nong's algorithm [17, Section 3]. By extension, the type of each symbol in T can be classified according to the type of the suffix starting with such symbol. In particular $T[i]$ is LMS-type if and only if T_i is LMS-type.

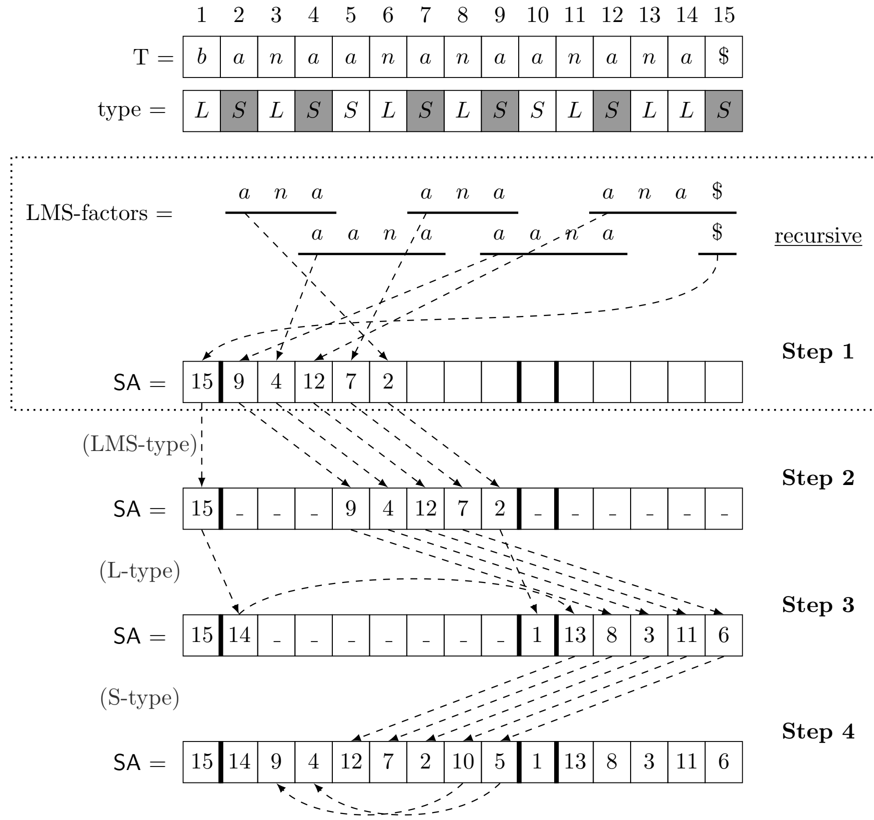

Fig. 2. Induced suffix sorting steps (SACA-K) for $T = banaananaanana\$$

Definition 3. *An LMS-factor of T is a factor that begins with a LMS-type symbol and ends with the following LMS-type symbol.*

We remark that LMS-factors do not establish a factorization of T since each of them overlaps with the following one by one symbol. By convention, $T[n, n]$ is always an LMS-factor. The LMS-factors of $T = banaananaanana\$$ are shown in Figure 2, where the type of each symbol is also reported. The LMS types are the grey entries. Notice that in SA all suffixes starting with the same symbol $c \in \Sigma$ can be partitioned into a c-bucket. We will keep an integer array $\mathsf{C}[1, \sigma]$ where $\mathsf{C}[c]$ gives either the first (head) or last (tail) available position of the c-bucket. Then, whenever we insert a value into the head (or tail) of a c-bucket, we increase (or decrease) $\mathsf{C}[c]$ by one. An important remark is that within each c-bucket S-type suffixes are larger than L-type suffixes. Figure 2 shows a running example of algorithm SACA-K for $T = banaananaanana\$$.

Given all LMS-type suffixes of $T[1, n]$, the suffix array can be computed as follows:

Steps:

1. Sort all LMS-type suffixes recursively into SA^1, stored in $\mathsf{SA}[1, n/2]$.
2. Scan SA^1 from right-to-left, and insert the LMS-suffixes into the tail of their corresponding c-buckets in SA.
3. Induce L-type suffixes by scanning SA left-to-right: for each suffix $\mathsf{SA}[i]$, if $T_{\mathsf{SA}[i]-1}$ is L-type, insert $\mathsf{SA}[i]-1$ into the head of its bucket.
4. Induce S-type suffixes by scanning SA right-to-left: for each suffix $\mathsf{SA}[i]$, if $T_{\mathsf{SA}[i]-1}$ is S-type, insert $\mathsf{SA}[i]-1$ into the tail of its bucket.

Step 1 considers the string T^1 obtained by concatenating the lexicographic names of all the consecutive LMS-factors (each different string is associated with a symbol that represents its lexicographic rank). Note that T^1 is defined over an alphabet of size $O(n)$ and that its length is at most $n/2$. The SACA-K algorithm is applied recursively to sort the suffixes of T^1 into SA^1, which is stored in the first half of SA. Nong *et al.* [18] showed that sorting the suffixes of T^1 is equivalent to sorting the LMS-type suffixes of T. We will omit details of this step, since our algorithm will not modify it.

Step 2 obtains the sorted order of all LMS-type suffixes from SA^1 scanning it from right-to-left and bucket sorting then into the tail of their corresponding c-buckets in SA. Step 3 induces the order of all L-type suffixes by scanning SA from left-to-right. Whenever suffix $T_{\mathsf{SA}[i]-1}$ is L-type, $\mathsf{SA}[i]-1$ is inserted in its final (corrected) position in SA.

Finally, Step 4 induces the order of all S-type suffixes by scanning SA from right-to-left. Whenever suffix $T_{\mathsf{SA}[i]-1}$ is S-type, $\mathsf{SA}[i]-1$ is inserted in its final (correct) position in SA.

Theoretical costs. Overall, algorithm SACA-K runs in linear time using only an additional array of size $\sigma + O(1)$ words to store the bucket array [17].

3 Inducing the Lyndon array

In this section we show how to compute the Lyndon array (LA) during Step 4 of algorithm SACA-K described in Section 2.1. Initially, we set all positions $\mathsf{LA}[i] = 0$, for $1 \leq i \leq n$. In Step 4, when SA is scanned from right-to-left, each value $\mathsf{SA}[i]$, corresponding to $T_{\mathsf{SA}[i]}$, is read in its final (correct) position i in SA. In other words, we read the suffixes in decreasing order from $\mathsf{SA}[n], \mathsf{SA}[n-1], \ldots, \mathsf{SA}[1]$. We now show how to compute, during iteration i, the value of $\mathsf{LA}[\mathsf{SA}[i]]$.

By Lemma 1, we know that the length of the longest Lyndon factor starting at position $\mathsf{SA}[i]$ in T, that is $\mathsf{LA}[\mathsf{SA}[i]]$, is equal to ℓ, where $T_{\mathsf{SA}[i]+\ell}$ is the next suffix (in text order) that is smaller than $T_{\mathsf{SA}[i]}$. In this case, $T_{\mathsf{SA}[i]+\ell}$ will be the first suffix in $T_{\mathsf{SA}[i]+1}, T_{\mathsf{SA}[i]+2} \ldots, T_n$ that has not yet been read in SA, which means that $T_{\mathsf{SA}[i]+\ell} < T_{\mathsf{SA}[i]}$. Therefore, during Step 4, whenever we read $\mathsf{SA}[i]$,

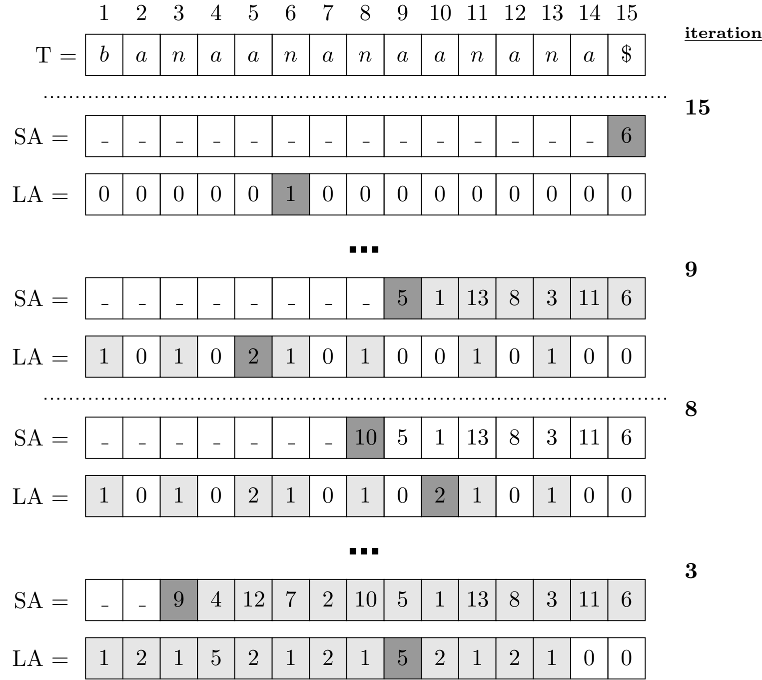

Fig. 3. Running example for $T = banaananaanana\$$.

we compute $\mathsf{LA}[\mathsf{SA}[i]]$ by scanning $\mathsf{LA}[\mathsf{SA}[i]+1,n]$ to the right up to the first position $\mathsf{LA}[\mathsf{SA}[i]+\ell]=0$, and we set $\mathsf{LA}[\mathsf{SA}[i]]=\ell$.

The correctness of this procedure follows from the fact that every position in $\mathsf{LA}[1,n]$ is initialized with zero, and if $\mathsf{LA}[\mathsf{SA}[i]+1], \mathsf{LA}[\mathsf{SA}[i]+2], \ldots, \mathsf{LA}[\mathsf{SA}[i]+\ell-1]$ are no longer equal to zero, their corresponding suffixes has already been read in positions larger than i in $\mathsf{SA}[i,n]$, and such suffixes are larger (lexicographically) than $T_{\mathsf{SA}[i]}$. Then, the first position we find $\mathsf{LA}[\mathsf{SA}[i]+\ell]=0$ corresponds to a suffix $T_{\mathsf{SA}[i]+\ell}$ that is smaller than $T_{\mathsf{SA}[i]}$, which was still not read in SA. Also, $T_{\mathsf{SA}[i]+\ell}$ is the next smaller suffix (in text order) because we read $\mathsf{LA}[\mathsf{SA}[i]+1,n]$ from left-to-right.

Figure 3 illustrates iterations $i=15$, 9, and 3 of our algorithm for $T = banaananaanana\$$. For example, at iteration $i=9$, the suffix T_5 is read at position $\mathsf{SA}[9]$, and the corresponding value $\mathsf{LA}[5]$ is computed by scanning $\mathsf{LA}[6], \mathsf{LA}[7], \ldots, \mathsf{LA}[15]$ up to finding the first empty position, which occurs at $\mathsf{LA}[7=5+2]$. Therefore, $\mathsf{LA}[5]=2$.

At each iteration $i=n, n-1, \ldots, 1$, the value of $\mathsf{LA}[\mathsf{SA}[i]]$ is computed in additional $\mathsf{LA}[\mathsf{SA}[i]]$ steps, that is our algorithm adds $O(\mathsf{LA}[i])$ time for each iteration of SACA-K.

Therefore, our algorithm runs in $O(n \cdot \mathsf{avelyn})$ time, where $\mathsf{avelyn} = \sum_{i=1}^{n} \mathsf{LA}[i]/n$. Note that computing LA does not need extra memory on top of the space for $\mathsf{LA}[1,n]$. Thus, the working space is the same as SACA-K, which is $\sigma + O(1)$ words.

Lemma 2. *The Lyndon array and the suffix array of a string $T[1,n]$ over an alphabet of size σ can be computed simultaneously in $O(n \cdot \mathsf{avelyn})$ time using $\sigma + O(1)$ words of working space, where avelyn is equal to the average value in $\mathsf{LA}[1,n]$.* □

In the next sections we show how to modify the above algorithm to reduce both its running time and its working space.

3.1 Reducing the running time to $O(n)$

We now show how to modify the above algorithm to compute each LA entry in constant time. To this end, we store for each position $\mathsf{LA}[i]$ the next smaller position ℓ such that $\mathsf{LA}[\ell] = 0$. We define two additional pointer arrays $\mathsf{NEXT}[1,n]$ and $\mathsf{PREV}[1,n]$:

Definition 4. *For $i = 1, \ldots, n-1$, $\mathsf{NEXT}[i] = \min\{\ell | i < \ell \leq n$ and $\mathsf{LA}[\ell] = 0\}$. In addition, we define $\mathsf{NEXT}[n] = n+1$.*

Definition 5. *For $i = 2, \ldots, n$, $\mathsf{PREV}[i] = \ell$, such that $\mathsf{NEXT}[\ell] = i$ and $\mathsf{LA}[\ell] = 0$. In addition, we define $\mathsf{PREV}[1] = 0$.*

The above definitions depend on LA and therefore NEXT and PREV are updated as we compute additional LA entries. Initially, we set $\mathsf{NEXT}[i] = i+1$ and $\mathsf{PREV}[i] = i-1$, for $1 \leq i \leq n$. Then, at each iteration $i = n, n-1, \ldots, 1$, when we compute $\mathsf{LA}[j]$ with $j = \mathsf{SA}[i]$ setting:

$$\mathsf{LA}[j] = \mathsf{NEXT}[j] - j \tag{2}$$

we update the pointers arrays as follows:

$$\mathsf{NEXT}[\mathsf{PREV}[j]] = \mathsf{NEXT}[j], \quad \text{if } \mathsf{PREV}[j] > 0 \tag{3}$$

$$\mathsf{PREV}[\mathsf{NEXT}[j]] = \mathsf{PREV}[j], \quad \text{if } \mathsf{NEXT}[j] < n+1 \tag{4}$$

The cost of computing each LA entry is now constant, since only two additional computations (Equations 3 and 4) are needed. Because of the use of the arrays PREV and NEXT the working space of our algorithm is now $2n + \sigma + O(1)$ words.

Theorem 1. *The Lyndon array and the suffix array of a string $T[1,n]$ over an alphabet of size σ can be computed simultaneously in $O(n)$ time using $2n+\sigma+O(1)$ words of working space.* □

3.2 Getting rid of a pointer array

We now show how to reduce the working space of Section 3.1 by storing only one array, say $\mathsf{A}[1,n]$, keeping $\mathsf{NEXT}/\mathsf{PREV}$ information together. In a glace, we store NEXT initially into the space of $\mathsf{A}[1,n]$, then we reuse $\mathsf{A}[1,n]$ to store the (useful) entries of PREV.

Note that, whenever we write $\mathsf{LA}[j]=\ell$, the value in $\mathsf{A}[j]$, that is $\mathsf{NEXT}[j]$ is no more used by the algorithm. Then, we can reuse $\mathsf{A}[j]$ to store $\mathsf{PREV}[j+1]$. Also, we know that if $\mathsf{LA}[j]=0$ then $\mathsf{PREV}[j+1]=j$. Therefore, we can redefine PREV in terms of A:

$$\mathsf{PREV}[j]=\begin{cases} j-1 & \text{if } \mathsf{LA}[j-1]=0\\ \mathsf{A}[j-1] & \text{otherwise.}\end{cases} \tag{5}$$

The running time of our algorithm remains the same since we have added only one extra verification to obtain $\mathsf{PREV}[j]$ (Equation 5). Observe that whenever $\mathsf{NEXT}[j]$ is overwritten the algorithm does not need it anymore. The working space is therefore reduced to $n+\sigma+O(1)$ words.

Theorem 2. *The Lyndon array and the suffix array of a string $T[1,n]$ over an alphabet of size σ can be computed simultaneously in $O(n)$ time using $n+\sigma+O(1)$ words of working space.* □

3.3 Getting rid of both pointer arrays

Finally, we show how to use the space of $\mathsf{LA}[1,n]$ to store both the auxiliary array $\mathsf{A}[1,n]$ and the final values of LA. First we observe that it is easy to compute $\mathsf{LA}[i]$ when T_i is an L-type suffix.

Lemma 3. $\mathsf{LA}[j]=1$ *iff T_j is an L-type suffix, or $i=n$.*

Proof. If T_j is an L-type suffix, then $T_j > T_{j+1}$ and $\mathsf{LA}[j]=1$. By definition $\mathsf{LA}[n]=1$. □

Notice that at Step 4 during iteration $i=n,n-1,\dots,1$, whenever we read an S-type suffix T_j, with $j=\mathsf{SA}[i]$, its succeeding suffix (in text order) T_{j+1} has already been read in some position in the interval $\mathsf{SA}[i+1,n]$ (T_{j+1} have induced the order of T_j). Therefore, the LA-entries corresponding to S-type suffixes are always inserted on the left of a block (possibly of size one) of non-zero entries in $\mathsf{LA}[1,n]$.

Moreover, whenever we are computing $\mathsf{LA}[j]$ and we have $\mathsf{NEXT}[j]=j+k$ (stored in $\mathsf{A}[j]$), we know the following entries $\mathsf{LA}[j+1],\mathsf{LA}[j+2],\dots,\mathsf{LA}[j+k-1]$ are no longer zero, and we have to update $\mathsf{A}[j+k-1]$, corresponding to $\mathsf{PREV}[j+k]$ (Equation 5). In other words, we update PREV information only for right-most entry of each block of non empty entries, which corresponds to a position of an L-type suffix because S-type are always inserted on the left of a block.

Then, at the end of the modified Step 4, if $\mathsf{A}[i]<i$ then T_i is an L-type suffix, and we know that $\mathsf{LA}[i]=1$. On the other hand, the values with $\mathsf{A}[i]>i$ remain

equal to $\mathsf{NEXT}[i]$ at the end of the algorithm. And we can use them to compute $\mathsf{LA}[i] = \mathsf{A}[i] - i$ (Equation 2).

Thus, after the completion of Step 4, we sequentially scan $\mathsf{A}[1, n]$ overwriting its values with LA as follows:

$$\mathsf{LA}[j] = \begin{cases} 1 & \text{if } \mathsf{A}[j] < j \\ \mathsf{A}[j] - j & \text{otherwise.} \end{cases} \quad (6)$$

The running time of our algorithm is still linear, since we added only a linear scan over $\mathsf{A}[1, n]$ at the end of Step 4. On the other hand, the working space is reduced to $\sigma + O(1)$ words, since we need to store only the bucket array $\mathsf{C}[1, \sigma]$.

Theorem 3. *The Lyndon array and the suffix array of a string of length n over an alphabet of size σ can be computed simultaneously in $O(n)$ time using $\sigma + O(1)$ words of working space.* □

Note that the bounds on the working space given in the above theorems assume that the output consists of SA and LA. If one is interested in LA only, then the working space of the algorithm is $n+\sigma+O(1)$ words which is still smaller that the working space of the other linear time algorithms that we discussed in Section 2.

4 Experiments

We compared the performance of our algorithm, called SACA-K+LA, with algorithms to compute LA in linear time by Franek *et al.* [5,9] (NSV-Lyndon), Baier [1,6] (Baier-LA), and Louza *et al.* [15] (BWT-Lyndon). We also compared a version of Baier's algorithm that computes LA and SA together (Baier-LA+SA). We considered the three linear time alternatives of our algorithm described in Sections 3.1–3.3. We tested all three versions since one could be interested in the fastest algorithm regardless of the space usage. We used four bytes for each computer word so the total space usage of our algorithms was respectively $17n$, $13n$ and $9n$ bytes. We included also the performance of SACA-K [17] to evaluate the overhead added by the computation of LA in addition to the SA.

The experiments were conducted on a machine with an `Intel Xeon` Processor `E5-2630` v3 20M Cache 2.40-GHz, 384 GB of internal memory and a 13 TB SATA storage, under a 64 bits `Debian GNU/Linux 8` (kernel 3.16.0-4) OS. We implemented our algorithms in ANSI C. The time was measured with `clock()` function of C standard libraries and the memory was measured using `malloc_count` library[6]. The source-code is publicly available at `https://github.com/felipelouza/lyndon-array/`.

We used string collections from the Pizza & Chili dataset[7]. In particular, the datasets `einstein-de`, `kernel`, `fib41` and `cere` are highly repetitive texts[8], and

[6] `https://github.com/bingmann/malloc_count`
[7] `http://pizzachili.dcc.uchile.cl/texts.html`
[8] `http://pizzachili.dcc.uchile.cl/repcorpus.html`

			LA			LA and SA				SA
dataset	σ	$n/2^{20}$	NSV-Lyndon [9]	Baier-LA [1,6]	BWT-Lyndon [15]	Baier-LA+SA [1,6]	SACA-K+LA-17n	SACA-K+LA-13n	SACA-K+LA-9n	SACA-K [17]
pitches	133	53	**0.15**	0.20	0.20	0.26	0.26	0.22	**0.18**	0.13
sources	230	201	**0.26**	0.28	0.32	0.37	0.46	0.41	**0.34**	0.24
xml	97	282	**0.29**	0.31	0.35	0.42	0.52	0.47	**0.38**	0.27
dna	16	385	0.39	**0.28**	0.49	**0.43**	0.69	0.60	0.52	0.36
english.1GB	239	1,047	0.46	**0.39**	0.56	**0.57**	0.84	0.74	0.60	0.42
proteins	27	1,129	0.44	**0.40**	0.53	0.66	0.89	0.69	**0.58**	0.40
einstein-de	117	88	0.34	**0.28**	0.38	**0.39**	0.57	0.54	0.44	0.31
kernel	160	246	0.29	**0.29**	0.39	**0.38**	0.53	0.47	**0.38**	0.26
fib41	2	256	0.34	**0.07**	0.45	**0.18**	0.66	0.57	0.46	0.32
cere	5	440	0.27	**0.09**	0.33	**0.17**	0.43	0.41	0.35	0.25
bbba	2	100	0.04	**0.02**	0.05	**0.03**	0.05	0.04	**0.03**	0.03

Table 1. Running time (μs/input byte).

the `english.1G` is the first 1GB of the original `english` dataset. We also created an artificial repetitive dataset, called `bbba`, consisting of a string T with 100×2^{20} copies of b followed by one occurrence of a, that is, $T = b^{n-2}a\$$. This dataset represents a worst-case input for the algorithms that use a stack (NSV-Lyndon and BWT-Lyndon).

Table 1 shows the running time of each algorithm in μs/input byte. The results show that our algorithm is competitive in practice. In particular, the version SACA-K+LA-9n was only about 1.35 times slower than the fastest algorithm (Baier-LA) for non-repetitive datasets, and 2.92 times slower for repetitive datasets. Also, the performance of SACA-K+LA-9n and Baier-LA+SA were very similar. Finally, the overhead of computing LA in addition to SA was small: SACA-K+LA-9n was 1.42 times slower than SACA-K, whereas Baier-LA+SA was 1.55 times slower than Baier-LA, on average. Note that SACA-K+LA-9n was consistently faster than SACA-K+LA-13n and SACA-K+LA-17n, so using more space does not yield any advantage.

Table 2 shows the peak space consumed by each algorithm given in bytes per input symbol. The smallest values were obtained by NSV-Lyndon, BWT-Lyndon and SACA-K+LA-9n. In details, the space used by NSV-Lyndon and BWT-Lyndon was $9n$ bytes plus the space used by the stack. The stack space was negligible (about 10KB) for almost all datasets, except for **bbba** where the stack used $4n$ bytes for NSV-Lyndon and $8n$ bytes for BWT-Lyndon (the

dataset	σ	$n/2^{20}$	LA: NSV-Lyndon [9]	LA: Baier-LA [1,6]	LA: BWT-Lyndon [15]	LA and SA: Baier-LA+SA [1,6]	LA and SA: SACA-K+LA-17n	LA and SA: SACA-K+LA-13n	LA and SA: SACA-K+LA-9n	SA: SACA-K [17]
`pitches`	133	53	**9**	17	**9**	17	17	13	**9**	5
`sources`	230	201	**9**	17	**9**	17	17	13	**9**	5
`xml`	97	282	**9**	17	**9**	17	17	13	**9**	5
`dna`	16	385	**9**	17	**9**	17	17	13	**9**	5
`english.1GB`	239	1,047	**9**	17	**9**	17	17	13	**9**	5
`proteins`	27	1,129	**9**	17	**9**	17	17	13	**9**	5
`einstein-de`	117	88	**9**	17	**9**	17	17	13	**9**	5
`kernel`	160	246	**9**	17	**9**	17	17	13	**9**	5
`fib41`	2	256	**9**	17	**9**	17	17	13	**9**	5
`cere`	5	440	**9**	17	**9**	17	17	13	**9**	5
`bbba`	2	100	**13**	17	17	17	17	13	**9**	5

Table 2. Peak space (bytes/input size).

number of stack entries is the same, but each stack entry consists of a pair of integers). On the other hand, our algorithm, SACA-K+LA-9n, used exactly $9n + 1024$ bytes for all datasets.

5 Conclusions

We have introduced an algorithm for computing simultaneously the suffix array and Lyndon array (LA) of a text using induced suffix sorting. The most space-economical variant of our algorithm uses only $n + \sigma + O(1)$ words of working space making it the most space economical LA algorithm among the ones running in linear time; this includes both the algorithm computing the SA and LA and the ones computing only the LA. The experiments have shown our algorithm is only slightly slower than the available alternatives, and that computing the SA is usually the most expensive step of all linear time LA construction algorithms. A natural open problem is to devise a linear time algorithm to construct only the LA using $o(n)$ words of working space.

Acknowledgments

The authors thank Uwe Baier for kindly providing the source codes of algorithms Baier-LA and Baier-LA+SA, and Prof. Nalvo Almeida for granting access to the machine used for the experiments.

Funding: F.A.L. was supported by the grant #2017/09105-0 from the São Paulo Research Foundation (FAPESP). G.M. was partially supported by PRIN grant 2017WR7SHH, by INdAM-GNCS Project 2019 *Innovative methods for the solution of medical and biological big data* and by the LSBC_19-21 Project from the University of Eastern Piedmont. S.M. and M.S. are partially supported by MIUR-SIR project CMACBioSeq *Combinatorial methods for analysis and compression of biological sequences* grant n. RBSI146R5L. G.P.T. acknowledges the support of Brazilian agencies Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES).

References

1. Baier, U.: Linear-time suffix sorting — a new approach for suffix array construction. In: Proc. Annual Symposium on Combinatorial Pattern Matching (CPM). pp. 23:1–23:12 (2016)
2. Bannai, H., I, T., Inenaga, S., Nakashima, Y., Takeda, M., Tsuruta, K.: The "runs" theorem. SIAM J. Comput. **46**(5), 1501–1514 (2017)
3. Crochemore, M., Russo, L.M.: Cartesian and Lyndon trees. Theoretical Computer Science (2018). https://doi.org/10.1016/j.tcs.2018.08.011
4. Fischer, J.: Inducing the LCP-Array. In: Proc. Workshop on Algorithms and Data Structures (WADS). pp. 374–385 (2011)
5. Franek, F., Islam, A.S.M.S., Rahman, M.S., Smyth, W.F.: Algorithms to compute the Lyndon array. In: Proc. PSC. pp. 172–184 (2016)
6. Franek, F., Paracha, A., Smyth, W.F.: The linear equivalence of the suffix array and the partially sorted Lyndon array. In: Proc. PSC. pp. 77–84 (2017)
7. Goto, K., Bannai, H.: Simpler and faster Lempel Ziv factorization. In: 2013 Data Compression Conference, DCC 2013, Snowbird, UT, USA, March 20-22, 2013. pp. 133–142 (2013)
8. Goto, K., Bannai, H.: Space efficient linear time Lempel-Ziv factorization for small alphabets. In: Proc. IEEE Data Compression Conference (DCC). pp. 163–172 (2014)
9. Hohlweg, C., Reutenauer, C.: Lyndon words, permutations and trees. Theor. Comput. Sci. **307**(1), 173–178 (2003)
10. Itoh, H., Tanaka, H.: An efficient method for in memory construction of suffix arrays. In: Proceedings of the sixth Symposium on String Processing and Information Retrieval (SPIRE '99). pp. 81–88. IEEE Computer Society Press (1999)
11. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Proc. 14th Symposium on Combinatorial Pattern Matching (CPM '03). pp. 200–210. Springer-Verlag LNCS n. 2676 (2003)
12. Kolpakov, R.M., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: Proc. FOCS. pp. 596–604 (1999)

13. Louza, F.A., Gog, S., Telles, G.P.: Inducing enhanced suffix arrays for string collections. Theor. Comput. Sci. **678**, 22–39 (2017)
14. Louza, F.A., Gog, S., Telles, G.P.: Optimal suffix sorting and LCP array construction for constant alphabets. Inf. Process. Lett. **118**, 30–34 (2017)
15. Louza, F.A., Smyth, W.F., Manzini, G., Telles, G.P.: Lyndon array construction during Burrows-Wheeler inversion. J. Discrete Algorithms **50**, 2–9 (2018)
16. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing **22**(5), 935–948 (1993)
17. Nong, G.: Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets. ACM Trans. Inf. Syst. **31**(3), 15 (2013)
18. Nong, G., Zhang, S., Chan, W.H.: Two efficient algorithms for linear time suffix array construction. IEEE Trans. Comput. **60**(10), 1471–1484 (2011)
19. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: Proc. IEEE Data Compression Conference (DCC). pp. 193–202 (2009)
20. Ohlebusch, E.: Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction. Oldenbusch Verlag (2013)
21. Okanohara, D., Sadakane, K.: A linear-time Burrows-Wheeler transform using induced sorting. In: Proc. International Symposium on String Processing and Information Retrieval (SPIRE). pp. 90–101 (2009)
22. Vuillemin, J.: A unifying look at data structures. Commun. ACM **23**(4), 229–239 (1980)