Suffix array and Lyndon factorization of a text*

Sabrina Mantaci[1], Antonio Restivo[1], Giovanna Rosone[1], and Marinella Sciortino[1]

[1]University of Palermo, Palermo, Italy

Abstract

The main goal in this paper is to highlight the relationship between the suffix array of a text and its Lyndon factorization. It is proved in [**?**] that one can obtain the Lyndon factorization of a text from its suffix array. Conversely, here we show a new method for constructing the suffix array of a text that takes advantage of its Lyndon factorization. The surprising consequence of our results is that, in order to construct the suffix array, the local suffixes inside each Lyndon factor can be separately processed, allowing different implementative scenarios, such as online, external and internal memory, or parallel implementations. Based on our results, the algorithm that we propose sorts the suffixes by starting from the leftmost Lyndon factors, even if the whole text or the complete Lyndon factorization are not yet available.

1 Introduction

In this paper we propose a strategy for the construction of the suffix array that takes advantage of a very close relationship between the sorting of the suffixes (the suffix array) of a text and a particular decomposition of the text itself, known as the *Lyndon factorization*.

 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, Marinella Sciortino, Suffix array and Lyndon factorization of a text, Journal of Discrete Algorithms, Volume 28, 2014, Pages 2-8, ISSN 1570-8667, https://doi.org/10.1016/j.jda.2014.06.001. Partially supported by Italian MIUR Project PRIN 2010LYA9RH, "Automi e Linguaggi Formali: Aspetti Matematici e Applicativi".

The suffix array is a very popular data structure in text algorithms, first introduced in [**?**], used both in theoretical studies and for practical applications. A recent quick tour on suffix arrays can be found in [**?**].

The Lyndon factorization is a text decomposition introduced in [**?**] having several applications in algebra and combinatorics on words. There exists a linear time algorithm, due to Duval [**?**], for computing the Lyndon factorization of a given text.

In [**?**], the authors show a method to deduce the Lyndon factorization of a text from its suffix array. Conversely, in this paper a construction of the suffix array (SA) and the Burrows-Wheeler transform (BWT) [**?**] of a text w from the Lyndon factorization of w is given. In [**?**], combinatorial aspects connecting three important data structures in string algorithms, i.e. the suffix array, the Burrows-Wheeler transform, and the extended BWT of a multiset of words [**?**, **?**], are studied. A variant of the BWT has been proposed in [**?**, **?**] by combining the Lyndon factorization of a text and the extended BWT of the multiset of the Lyndon factors. Note that, differently from our approach, the output of such a transformation in general does not coincide with the BWT of a text. For instance, if $w = cbabacaac$, the Lyndon factorization is $c|b|abac|aac$, the output of the variant in [**?**, **?**] is $ccababaac$, whereas BWT produces $cbbacaaca$.

In the literature, there exist several techniques for sorting suffixes in order to compute the SA, and, in general, they require that the whole text is available. The Lyndon factorization and some combinatorial properties proved in this paper allow the sorting of the suffixes of w ("global suffixes") by using the sorting of the suffixes inside each block of the decomposition ("local suffixes").

The main theorem in this paper states that if u is a concatenation of consecutive Lyndon factors of a word w, then the mutual order of two local suffixes in u is maintained when they are extended as global suffixes. This result could suggest new strategies for the computation of the SA and the BWT.

We give one of the possible implementations of a strategy that, at the same time, incrementally computes from left to right both the SA and the BWT, without the whole text or the complete Lyndon factorization being available.

In Section **??** we give the fundamental notions and results concerning combinatorics on words, the Lyndon factorization and the SA. In Section **??** we first introduce the notions of global and local suffixes and we prove the main theorem. In Section **??** we describe an algorithm that uses the above results to incrementally construct the SA and the BWT of a text

from left to right, and we discuss about possible implementations. Section **??** is devoted to some further developments and conclusions. A preliminary version of the results given in the present paper can be found in [**?**].

2 Preliminaries

Let $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ be a finite alphabet with $c_1 < c_2 < \ldots < c_\sigma$. Given a finite word $w = a_1 a_2 \cdots a_n$, $a_i \in \Sigma$ for $i = 1, \ldots, n$, a *factor* of w is written as $w[i, j] = a_i \cdots a_j$. A factor $w[1, j]$, for $j = 1, \ldots, n$, is called *prefix*, while a factor $w[i, n]$, for $i = 1, \ldots, n$, is called *suffix*. We say that $x, y \in \Sigma^*$ are *conjugate* if $x = uv$ and $y = vu$ for some $u, v \in \Sigma^*$. Recall that conjugacy is an equivalence relation. A word is *primitive* if all of its conjugates are distinct.

Given a text w of length n, the suffix array (SA) of w is an array of integers ranging from 1 to n specifying the lexicographic ordering of the suffixes of the string w. That is, $SA[j] = i$ if and only if $w[i, n]$ is the j-th suffix of w in ascending lexicographical order. For instance, if $w =$ *mathematics* then

$$SA(w) = [2,\ 7,\ 10,\ 5,\ 4,\ 9,\ 1,\ 6,\ 11,\ 3,\ 8].$$

We recall that in many applications it is useful to append the symbol \$ to the end of the word w, where \$ is considered as a symbol smaller than any other letter in the text alphabet.

A *Lyndon* word is a primitive word which is the minimum in its conjugacy class, with respect to the lexicographic order relation. In [**?**, **?**], one can find a linear algorithm that for any word $w \in \Sigma^*$ computes the Lyndon word in its conjugacy class. We call it *the Lyndon word of* w.

Note that, in general, the sorting of two conjugates of a word w starting in two given positions could be different from the sorting of the suffixes starting in the same positions, but, as consequence of the properties of Lyndon words, when w is a Lyndon word, then the two sorting coincide (cf. [**?**, Lemma 12]). Therefore, a Lyndon word can equivalently be defined as a word that is the minimum of its suffixes, with respect to the lexicographic order relation. Lyndon words are involved in an important factorization property of words.

Theorem 2.1. *[**?**] Every word $w \in \Sigma^+$ has a unique factorization $w = l_1 l_2 \cdots l_k$ such that $l_1 \geq_{lex} \cdots \geq_{lex} l_k$ is a non-increasing sequence of Lyndon words.*

We call such a decomposition the *Lyndon factorization* of a word and it can be computed in linear time (see for instance [**?**, **?**]). Lyndon factorization has been realized also in parallel (cf. [**?**, **?**]) and in external memory (cf. [**?**]).

A method to deduce the Lyndon factorization of a text from its SA has been already given in [**?**]. In particular in the paper the notion of suffix permutation is used.

Recall that the *suffix permutation* (cf. [**?**]) of a word $w = a_1 \cdots a_n$ is the permutation π_w over $\{1, \ldots, n\}$, where $\pi_w(i)$ is the rank of the suffix $w[i, n]$ in the set of the lexicographically sorted suffixes of w. In other words the suffix permutation π_w is the inverse permutation defined by the suffix array $SA(w)$. Given a permutation π over $\{1, \ldots, n\}$, an integer i $(1 \leq i \leq n)$ is a *left-to-right minimum* of π if either $i = 1$ or $\pi(j) > \pi(i)$, for all $j < i$. The method described in [**?**] is synthesized in the following theorem.

Theorem 2.2. *Let w be a word, let $i_1 = 1, i_2, \ldots, i_k$ be the positions of the left-to-right minima of the suffix permutation π_w. Then the values $i_1, i_2, \ldots, i_k$ correspond to the starting positions of the factors in the Lyndon factorization of w.*

Since nowadays several $O(n)$ algorithms for computing the SA are known, this implies that an algorithm based on Theorem **??** can be considered as an efficient alternative method to Duval's.

3 Suffix array of a text through its Lyndon factorization

In this section, we deal with a problem which is dual with respect to Theorem **??**. In particular, we show how the knowledge of the Lyndon factorization of a text helps in the process of sorting its suffixes, a fundamental step for computing the suffix array.

We recall that, in general, one can establish the order relation between two suffixes of a text, by looking for the first symbol mismatch. We show that, if the Lyndon factorization of the text is known, it is sufficient to look at the suffixes of some particular factors, reducing in this way the needed number of symbol comparisons.

Let $w \in \Sigma^*$ and let $w = l_1 l_2 \cdots l_k$ be its Lyndon factorization. For each factor l_r, we denote by $first(l_r)$ and $last(l_r)$ the position of the first and the last character, respectively, of the factor l_r in w. Let u be a factor of w. We denote by $suf_u(i) = w[i, last(u)]$ and we call it *local suffix* at the position

i with respect to u. Note that $suf_w(i) = w[i, n]$ and we call it *global suffix* of w at the position i. We write $suf(i)$ instead of $suf_w(i)$ when there is no danger of ambiguity.

Definition 3.1. Let w be a word and let u be a factor of w. We say that the sorting of local suffixes with respect to u is *compatible* with the sorting of the global suffixes if for all i, j with $first(u) \leq i < j \leq last(u)$,

$$suf_u(i) < suf_u(j) \iff suf(i) < suf(j).$$

Notice that in general, taken an arbitrary factor of a word w, the sorting of its suffixes is not compatible with the sorting of the suffixes of w. Consider for instance the word $w = abababb$ and its factor $u = ababa$. Then $suf_u(1) = ababa > a = suf_u(5)$ whereas $suf(1) = abababb < abb = suf(5)$.

Theorem 3.2. *Let $w \in \Sigma^*$ and let $w = l_1 l_2 \cdots l_k$ be its Lyndon factorization. Let $u = l_r l_{r+1} \cdots l_s$ with $1 \leq r \leq s \leq k$. Then the sorting of the local suffixes with respect to u is compatible with the sorting of the global suffixes.*

Proof. Let i and j be two indexes with $i < j$ both contained in u. We just need to prove that $suf(i) > suf(j) \iff suf_u(i) > suf_u(j)$. Let $x = w[j, last(l_s)]$ and $y = w[i, i + |x| - 1]$.

Suppose that $suf(i) > suf(j)$. Then $y \geq x$ by the definition of lexicographic order. If $y > x$ there is nothing to prove. If $x = y$, then $suf_u(j)$ is prefix of $suf_u(i)$, so by the definition of lexicographic order $suf_u(i) > suf_u(j)$.

Suppose now that $suf_u(i) > suf_u(j)$. This means that $y \geq x$. If $y > x$ there is nothing to prove. If $x = y$, the index $i + |x| - 1$ is in some Lyndon factor l_m with $r \leq m \leq s$, then $l_r \geq l_m \geq l_s$. We denote $z = w[i + |x|, last(l_m)]$. Then $suf(i) = xzl_{m+1} \cdots l_k > xl_{s+1} \cdots l_k = suf(j)$, since $z > l_m$ (because l_m is a Lyndon word) and $l_m \geq l_{s+1}$ (since the factorization is a sequence of non increasing factors). □

The above theorem states, in other words, that the mutual order of the suffixes of w starting in two positions i and j is the same as the mutual order of the "local" suffixes starting in i and j inside each block obtained as concatenation of consecutive Lyndon factors including i and j.

As particular case, the theorem is also true when the two suffixes start in the same Lyndon factor.

Similar considerations used to prove Theorem **??** can be used to give an alternative proof of the result of Theorem **??**.

We recall that, if L_1 and L_2 denote two sorted lists of elements taken from any well ordered set, the operation $merge(L_1, L_2)$ computes the sorted list of elements in L_1 and L_2.

A consequence of previous theorem is stated in the following corollary.

Corollary 3.3. *Let $sort(l_1 l_2 \cdots l_r)$ and $sort(l_{r+1} l_{r+2} \cdots l_k)$ denote the sorted lists of the suffixes of $l_1 l_2 \cdots l_r$ and the suffixes $l_{r+1} l_{r+2} \cdots l_k$, respectively. Then $sort(l_1 l_2 \cdots l_k) = merge(sort(l_1 l_2 \cdots l_r), sort(l_{r+1} l_{r+2} \cdots l_k))$.*

Theorem **??** also gives a bound on the number of symbol comparisons needed to obtain the order relation between two global suffixes.

Remark 3.4. Let i and j be two positions in the word w. If $i < j$, let us denote by $lcp(i, j)$ the length of the longest common prefix between the global suffixes $w[i, n]$ and $w[j, n]$. Let l_r and l_s $(r < s)$ be the Lyndon factors in the Lyndon factorization containing respectively i and j. Let u be the smallest concatenation of consecutive Lyndon factors containing both l_r and l_s, i.e. $u = l_r l_{r+1} \cdots l_s$. The previous theorem states that in order to get the mutual order between the global suffixes $w[i, n]$ and $w[j, n]$ one needs $\min(lcp(i, j) + 1, m)$ symbol comparisons, where m denotes the length of the rightmost local suffix with respect to u (i.e. the one starting at the position j).

The following example shows that m can be much smaller than $lcp(i, j) + 1$.

Example 3.5. Let $w = abaaaabaaaaabaaaabaaaaaab$. The Lyndon factorization of w is $ab|aaaab|aaaaabaaaab|aaaaaab$.
Consider the global suffixes $w[2, 25] = b|aaaab|aaaaabaaaab|aaaaaab$ and $w[13, 25] = baaaab|aaaaaab$. We have that the first mismatch between $w[2, 25]$ and $w[13, 25]$ can be found after 12 symbol comparisons.

$$\begin{array}{rccccccc} & & & m & & & lcp(2,13)+1 & \\ & & & \downarrow & & & \downarrow & \\ w[2,25] = & baaaa & b & aaaaa & b & aaaabaaaaaab & \\ w[13,25] = & baaaa & b & aaaaa & a & b & \end{array}$$

Let $u = w[2, 18] = ab|aaaab|aaaaabaaaab|$. By Theorem **??** we just need to perform $m = 6 < 12 = lcp(2, 13) + 1$ symbol comparisons. So, even if $w[2, 7] = w[13, 18]$, the mutual order is established by the Lyndon properties, indeed we can state that the Lyndon word starting at the position 19, $w[19, 25] = aaaaaab$, is smaller than of the suffix starting at the position 8.

4 An incremental left-to-right computation of the suffix array of a text

The results of previous section suggest a versatile technique that can be easily adapted to different implementative scenarios. In fact, if the Lyndon factorization of a word $w = l_1 l_2 \cdots l_k$ is given, the suffix array of each of its Lyndon factors can be computed separately by using any known algorithm. The resulting sorted lists have to be merged in a second step in order to obtain the sorted list of all the suffixes of w. The efficiency of the algorithm will depend on the merging strategy. A possible strategy for the merging is the one presented in [**?**], where Lyndon factors in the Lyndon factorization are considered one-by-one from left to right.

A surprising property of these techniques based on the Lyndon factorization is that one can determine the mutual lexicographic order among the global suffixes of a text by considering the local suffixes of each Lyndon factor from left to right, even before that the whole text is available.

Here we present a new strategy, integrated with the Duval's algorithm for Lyndon decomposition, that processes the symbols of w from left to right in order to find the Lyndon factors and, when a Lyndon factor l_i is found, it inserts the local suffixes of l_i, from the rightmost one to the leftmost one, in the sorted list of the already considered local suffixes. The algorithm stops when the last Lyndon factor l_k is processed. Consequently, when the complete Lyndon factorization of w into $l_1 l_2 \cdots l_k$ is determined, the suffix array is definitively computed.

In order to find the mutual lexicographic order among all local suffixes of the text, at each step of the algorithm we use an array of characters that contains the Burrows-Wheeler Transform (BWT) [**?**] of the processed text. Recall that an end-of-string symbol \$ (smaller than any other letter) is usually appended to the end of the input text, when the BWT of the text is constructed. The BWT is intuitively described as follows: given a text $v \in \Sigma^*$, $bwt(v\$)$ is a word obtained by sorting the list of the suffixes of $v\$$ and by concatenating the symbols (circularly) preceding every suffix of $v\$$ in the sorted list. For instance, if $v = mathematics$ then $bwt(v\$) = smmihtt\$ecaa$.

In the sequel, we consider the symbol \$ appended to the input text w. Let us denote by $\mathcal{L}$ the list, initially empty and incrementally constructed, of the lexicographically sorted local suffixes of $w\$$.

An intuitive description of the steps of our strategy, when the Lyndon factorization of $w\$$ is $l_1 l_2 \cdots l_k\$$, is the following.

- In the first step ($i = 1$), find the first Lyndon word l_1 of the decomposi-

tion of $w\$$. Then any known algorithm can be applied to compute the suffix array of the first Lyndon factor. In analogy with the following steps, proceed as follows:

  - append the symbol \$ to l_1;
  - insert the local suffix \$ in $\mathcal{L}$;
  - insert the remaining local suffixes of $l_1\$$, by proceeding from right to left, in $\mathcal{L}$ according with the lexicographic order.

- For $i = 2, \ldots, k$, find the i-th Lyndon word l_i of the decomposition of $w\$$ and:

  - append the symbol \$ to l_i;
  - replace in $\mathcal{L}$ the local suffix \$ with the local suffix $l_i\$$;
  - insert the local suffix \$ of $l_i\$$ in the first position of $\mathcal{L}$;
  - insert the remaining local suffixes of $l_i\$$, by proceeding from right to left, in $\mathcal{L}$, according with the lexicographic order.

- At step $i = k + 1$, we should insert in $\mathcal{L}$ the suffix \$ of $w\$$, but it has been already inserted in $\mathcal{L}$ during the k-th step (as suffix of $l_k\$$).

Our on-line computation of the suffix array (named BUILD_SA) takes its cue from the first variant of Duval's algorithm for Lyndon factorization. This variant uses only three variables for a complete computation. The variable q contains the index of the current input letter, and the variable p represents an index such that when the letter a_q is processed, one has $a_1 \cdots a_{p-1} = a_{q-p+1} \cdots a_{q-1}$ and $a_1 \cdots a_{q-1}$ is a Lyndon word. The variable h is introduced in such a way that the remaining suffix is $a_{h+1} \cdots a_n$. The procedure continues until all Lyndon factors are found. Note that the indexes p, q are moved from left to right and each input symbol is read at most twice.

Since the function BUILD_SA essentially consists in Duval's algorithm equipped of the insertion strategy of the suffixes of the Lyndon factors, we explicitly describe only the function INSERT (see Figure **??**). Whenever a Lyndon factor l_i is constructed, we call the function INSERT in order to add, at the i-th step, the positions in SA and the symbols in BWT associated with the local suffixes of $l_i\$$. The function INSERT takes in input the index i, the first and the last position of l_i in w, the SA and the BWT computed in the previous step. Such a function returns the updated SA and the updated BWT.

The function NEW, used in INSERT function, takes as input an array T and a position r and allocates a new cell in T at the position r.

In the function INSERT, in order to add the local suffixes, we also use two known functions (introduced in [**?**]): RANK and C. For any character $x \in \Sigma$, let $\mathrm{C}(v, x)$ denote the number of symbols in the given word v that are smaller than x, and let $\mathrm{RANK}(v, t, x)$ denote the number of occurrences of x in the prefix of length t of v.

```
INSERT(i, first, last, SA, BWT)
 1 L = w[first] ··· w[last]; len = last − first + 1;
 2 append $ to L;
 3 if (i = 1) then
 4 |   NEW(SA, 1); SA[1] := 1;
   |   /* Insert the symbol $ in BWT                                  */
 5 |   NEW(BWT, 1); x = L[len + 1]; BWT[1] := x;
 6 γ = 0;
 7 NEW(SA, γ + 1); SA[γ + 1] := last + 1;
   /* Insert at the position γ + 1 the symbol preceding the suffix $ of L
      */
 8 NEW(BWT, γ + 1); x = L[len]; BWT[γ + 1] := x;
 9 prevSymb = x; prevPos = γ + 1;
10 for j=length to 2 do
11 |   γ =C[BWT, prevSymb]+RANK(BWT, prevPos, prevSymb);
12 |   NEW(SA, γ + 1); SA[γ + 1] := first + j − 1;
   |   /* Insert at the position γ + 1 the symbol preceding L[j, len + 1]  */
13 |   NEW(BWT, γ + 1); x = L[len + 1]; BWT[γ + 1] := x;
14 |   prevSymb = x; prevPos = γ + 1;
15 return (SA, BWT);
```

Figure 1: The algorithm to construct the SA and the BWT of $l_1 \cdots l_i\$$ by incrementally adding the positions in SA and the symbols in BWT associated with the local suffixes of $L = l_i\$$.

The following example is useful to illustrate how the SA, the BWT and, consequently, the elements of $\mathcal{L}$ (extended as global suffixes of $w\$$) are updated during a step.

Example 4.1. Let $w = abcabdaabcabb\$$ and let $abcabd|\mathbf{aabcabb}|\$$ its Lyndon factorization. During the first step, we compute the SA and the BWT of $l_1\$ = abcabd\$$ and we get $BWT = d\$caabb$ and $SA = [7, 1, 4, 2, 5, 3, 6]$.

In the second step, we have to consider all the suffixes of $\mathbf{l_2}\$ = \mathbf{aabcabb}\$$. Suppose we have already inserted the symbols and the positions associated with the local suffixes \$ and **b**\$ of $\mathbf{l_2}\$$ in BWT and SA, respectively. The situation is depicted on the left in Figure **??**. On the right, we depict the situation after the insertion of the symbol and the position associated with the local suffix **bb**\$.

SA	BWT	$\mathcal{L}$
14	**b**	\$
7	d	**a a b c a b b** \$
1	\$	$a\ b\ c\ a\ b\ d$ |**a a b c a b b** \$
4	c	$a\ b\ d$ |**a a b c a b b** \$
→ **13**	**b**	**b** \$
2	a	$b\ c\ a\ b\ d$ |**a a b c a b b** \$
5	a	$b\ d$ |**a a b c a b b** \$
3	b	$c\ a\ b\ d$ |**a a b c a b b** \$
6	b	d |**a a b c a b b** \$

⇒

SA	BWT	$\mathcal{L}$
14	**b**	\$
7	d	**a a b c a b b** \$
1	\$	$a\ b\ c\ a\ b\ d$ |**a a b c a b b** \$
4	c	$a\ b\ d$ |**a a b c a b b** \$
13	**b**	**b** \$
→ **12**	**a**	**b b** \$
2	a	$b\ c\ a\ b\ d$ |**a a b c a b b** \$
5	a	$b\ d$ |**a a b c a b b** \$
3	b	$c\ a\ b\ d$ |**a a b c a b b** \$
6	b	d |**a a b c a b b** \$

Figure 2: For presentation simplicity, we show the local suffixes in $\mathcal{L}$ extended as global suffixes. The three columns represent the partial SA and partial BWT before and after the (implicit) insertion of the new suffix **bb**\$ of $\mathbf{l_2}\$$. The position $\gamma + 1$ of the new symbol (shown by the arrow $\rightarrow$ in the table on the right) is computed from the position of the last inserted symbol (shown by $\rightarrow$ in the table on the left). Hence $\gamma = \mathrm{C}(BWT, b) + \textsc{rank}(BWT, 5, b) = 3 + 2 = 5$, because in BWT there are two a's and the symbol \$ smaller than b. Moreover there are 2 suffixes starting with b that are smaller or equal to $bb\$$. So, 12 and a are inserted at the position 6 both in the SA and in the BWT (see the table on the right). At the end of this iteration, $prevSymb = a$ and $prevPos = 6$. At the next iteration $\gamma = \mathrm{C}(BWT, a) + \textsc{rank}(BWT, 6, a) = 1 + 1 = 2$. So, 11 and c will be inserted at the position 3 in the SA and the BWT, respectively.

4.1 Correctness

The correctness of our strategy can be deduced by the following lemmas. In particular, the next two lemmas show that, if $\mathcal{L}$ is the list of lexicographic sorted local suffixes of $l_1 \cdots l_{i-1}\$$, then the local suffixes \$ and $l_i\$$ of $l_1 \cdots l_i\$$ are placed in the first two positions of $\mathcal{L}$. Lemma **??** determines the positions of the remaining suffixes of $l_1 \cdots l_i\$$ in $\mathcal{L}$.

Lemma 4.2. *For each step $i \geq 2$, the local suffix $l_i\$$ replaces the local suffix $\$$ of the suffix $l_1 \cdots l_{i-1}\$$ in $\mathcal{L}$.*

Proof. During the step $i-1$, we have computed the data structures, i.e. the SA and the BWT, associated with $l_1 l_2 \cdots l_{i-1}\$$. At the step i, we have to update such data structures in order to obtain those associated with $l_1 l_2 \cdots l_{i-1} l_i\$$. In terms of the sorted list $\mathcal{L}$, we should remove the local suffix $\$$ of $l_1 l_2 \cdots l_{i-1}\$$ and insert the local suffix $l_i\$$ in $\mathcal{L}$. For the Lyndon factorization properties, the local suffix $l_i\$$ is the smallest suffix among all suffixes in $l_1 l_2 \cdots l_{i-1} l_i\$$ when its suffix $\$$ is not considered. So the previous local suffix $\$$ have to be replaced in $\mathcal{L}$ with the new local suffix $l_i\$$. An important fact is that the value in the SA and the symbol in the BWT at the first position never change, since in the previous step they are associated with the suffix $\$$, and now they are associated with the suffix $l_i\$$. □

Note that for each step i from 1 to k, the position in $\mathcal{L}$ of the suffix $\$$ of $l_i\$$ is 1. The crucial point at each step i is to establish the position where we have to insert in $\mathcal{L}$ the remaining local suffixes of $l_i\$$. The following lemma illustrates how to compute the positions in SA of these local suffixes from the rightmost one to the leftmost. In particular, it determines the position in SA of the local suffix of $l_i\$$ starting at j where j is a integer ranging from $|l_i|$ down to 2, after that the positions of the local suffixes starting at $j+1, j+2, \ldots, |l_i|$ have been determined. Let t be the position in SA of the local suffix $l_i[j+1, |l_i|]\$$ in the list $\mathcal{L}$.

Lemma 4.3. *The local suffix $l_i[j, |l_i|]\$$ is lexicographically larger than precisely γ local suffixes, where*

$$\gamma = \mathrm{C}(BWT, l_i[j]) + \textsc{rank}(BWT, t, l_i[j]),$$

where BWT is the Burrows-Wheeler Transform containing the symbols associated to the local suffixes in $\mathcal{L}$. Therefore the symbol and the value associated with $l_i[j, |l_i|]\$$ have to be inserted in position $\gamma + 1$ of BWT and SA, respectively.

Proof. Recall that $\mathrm{C}(BWT, l_i[j])$ gives the number of already (implicity) inserted suffixes starting with a symbol smaller than $l_i[j]$ that are lexicographically smaller than $l_i[j, |l_i|]\$$. Let us count now the number of suffixes starting with $l_i[j]$ and being smaller than $l_i[j, |l_i|]\$$. This is equivalent to counting how many symbols equal to $l_i[j]$ occur in $BWT[1, t]$. Such a value is given by $\textsc{rank}(BWT, t, l_i[j])$. □

At the end of the step i (from 1 to k), the arrays $BWT(l_1 l_2 \cdots l_i\$)$ and $SA(l_1 l_2 \cdots l_i\$)$ are computed. Such insertions do not affect the relative order of the local suffixes already considered in the previous steps. Note that we can stop the process at the end of each step i, i.e. after the insertion of all local suffixes of the Lyndon word l_i, so that we can obtain the suffix array of the prefix of the word w up to the Lyndon factor l_i, i.e. the word $l_1 \cdots l_i\$$. This fact is at the base of our on-line algorithm.

As a consequence of the previous lemmas, we can state the following theorem.

Theorem 4.4. *Given a text w, the* BUILD_SA(w) *algorithm correctly computes the SA and BWT of the text by using its Lyndon factorization.*

4.2 Discussion on complexity

We recall that, although the computation of the Lyndon factorization is linear, the current non-linear cost of the entire algorithm for the construction of the suffix array could make it impractical. Actually, the complexity of the algorithm depends on the time-space trade-off that one wishes to reach. More precisely, the complexity depends on the suitable data structures used for the RANK and NEW operations. For instance, in order to compute the BWT, one could use Navarro and Nekrich's recent result [**?**] on optimal representations of dynamic sequences. They show that one can insert symbols in arbitrary positions and compute the RANK function in the optimal time $O(\frac{\log n}{\log \log n})$ within essentially $nH_0(s)+O(n)$ bits of space, for a text of length n. Moreover, it is possible to give also an external memory implementation of our algorithm. Indeed, one could compute the Lyndon factorization in external memory by using, for instance, the algorithm in [**?**]. One could also implement the function INSERT by using the methods in [**?**, **?**] where disk data access are executed only via sequential scans, so that it could be adapted in order to obtain a lightweight version of our algorithm.

5 Conclusions

In this paper, we have highlighted the tight relation between the SA and the Lyndon factorization. An important consequence of the compatibility of the sorting of the local suffixes inside the Lyndon factors with the sorting of the global suffixes is that our method seems to lay out the path towards a new approach to the problem of sorting the suffixes of a text. Partitioning the

text by using its combinatorial properties allows to tackle the problem in local portions of the text, in order to extend solutions to a global dimension.

In particular, in this paper we give one of the possible implementations of this general method that works by incrementally inserting one at a time a symbol in the partial BWT and a value in the partial SA. Differently from other incremental methods [**?**, **?**, **?**, **?**] that insert the suffixes in the sorted list by proceeding from the rightmost to the leftmost one, our algorithm factorizes the word from left to right, and then, for implementative convenience, inserts in the sorted list the local suffixes of the processed factor from the rightmost to the leftmost one. The advantage of this method is that, since Duval's algorithm discovers Lyndon factors with a very small lookahead w.r.t. the end of the factor itself, this allows to start the construction of the SA even while the whole text is not yet available. As a consequence, this method allows to define online algorithms for the construction of the SA, such as, for instance, the one described in the present paper. Moreover this implies that the substitution, insertion or deletion of a symbol in the i-th Lyndon factor do not affect the mutual order of the suffixes starting into the Lyndon factors $l_1, \ldots, l_{i-2}$. Moreover, as remarked above, the independence of the mutual sorting of suffixes inside the Lyndon blocks, suggests a possible design of parallel solutions. Indeed, one could compute the Lyndon factorization in parallel way, for instance, as shown in [**?**, **?**]. The suffix array of each Lyndon factor could be also computed in parallel way, as shown in [**?**]. In this case, the efficiency of the algorithm will depend on the merging strategy. A possible strategy for this purpose is shown in [**?**]. Parallelization techniques could also be used for the sorting and merging phases.

Unfortunately, the algorithm proposed here is not competitive in terms of time complexity with respect to the existing algorithms for the construction of the suffix array. Anyway, the idea of working independently on local portions of the text in order to extend solution to global suffixes is quite new and the presented algorithm is probably subject to improvements. Further, an interesting question could be to ask whether there exist some different combinatorial decomposition where the compatibility between the sorting of local and global suffixes of the input text holds.