The Linear Equivalence of the Suffix Array and the Partially Sorted Lyndon Array

Frantisek Franek[1], Asma Paracha[1], and William F. Smyth[1,2]

[1] Department of Computing & Software
McMaster University, Hamilton, Canada
{franek/paracham/smyth}@mcmaster.ca
[2] School of Engineering & Information Technology
Murdoch University, Perth, Australia

Abstract. In 2015 Uwe Baier presented a linear-time algorithm that directly sorts the suffixes of a string, the first such algorithm that is not recursive. In fact, his approach implicitly gives quite a bit more: it includes a linear-time elementary algorithm for computing what turns out to be a partially sorted version of the Lyndon array, and then shows how this can be used to sort the suffixes. At the same time, it is known that the Lyndon array can be computed in linear time from the suffix array. This paper extends these aspects of Baier's work to establish the *linear equivalence* of certain orderings of maximal Lyndon substrings and of fully sorted suffixes. By this terminology we mean that each data structure can be transformed into the other by a simple linear-time computation.

Keywords: string; Lyndon string; suffix; suffix array; Lyndon array; sorted Lyndon array; partially sorted Lyndon array

1 Introduction

In [1,2] Baier described a linear-time algorithm to sort the suffixes of a string, the first such algorithm that is not recursive. In fact, his approach implicitly gives much more: it includes a linear-time elementary algorithm for computing a partially sorted version of the Lyndon array, and it shows how this partial sort can be used to yield a complete sort of the suffixes. (Baier does not in his paper or his thesis make explicit reference to Lyndon substrings or to the Lyndon array.) On the other hand, it is known that the regular (unsorted) Lyndon array can be computed in linear time from the suffix array [6,5]. Thus there is some sort of *linear equivalence* between certain orderings of the Lyndon array and the suffix array, a relationship that we make precise in this paper.

Baier's algorithm works in two phases: in the first phase the suffixes of the input string are distributed into "groups" (that actually correspond to a partial sort of entries in the Lyndon array); then in the second phase the suffix array of the input string is computed from the groups. This paper deals mainly with the second phase: we show that the groups of suffixes output by Baier's Phase 1 are in fact an arrangement of the maximal Lyndon substrings of the input string, and further that this arrangement leads naturally, in linear time, to the suffix array. We also show how to go in the reverse direction; that is, how to compute the groups from the suffix array.

In the next section we introduce the ideas and notation that we use — most importantly, precise definitions for various notions of "groups of suffixes". In Section 3 two main theorems are presented, showing the linear equivalence of a *partially sorted Lyndon array* and the *suffix array* of a string.

Frantisek Franek, Asma Paracha, William F. Smyth: The Linear Equivalence of the Suffix Array and the Partially Sorted Lyndon Array, pp. 77–84.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

2 Preliminaries

In the literature, *string* and *word* are used interchangeably. But *word* may be used to refer to a substring, as in: `let` k `be the length of the longest Lyndon word in` $\boldsymbol{x}$ `starting at position` i. The terms *subword*, *substring*, and *factor* are also often used interchangeably. To prevent confusion, we will strictly use *string* and *substring*.

We use the array notation for strings indexing from 1; that is, $\boldsymbol{x} = \boldsymbol{x}[1..n]$ indicates that string $\boldsymbol{x}$ consists of n alphabet symbols and thus has length n. Strings are given in bold to distinguish them from other entities such as numbers and alphabet symbols: for example, $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and $\boldsymbol{x}[i] = a$. The *length* of a string $\boldsymbol{x}$ is denoted by $|\boldsymbol{x}|$. An *empty string* of length 0 is denoted by ε. The notation $\boldsymbol{x}[i..j)$ is used as an abbreviation for $\boldsymbol{x}[i..j-1]$.

The symbol $\boldsymbol{xy}$ denotes the *concatenation* of $\boldsymbol{x}$, $\boldsymbol{y}$; in particular, $\boldsymbol{x}[1..n] = \boldsymbol{x}[1]\boldsymbol{x}[2]\cdots\boldsymbol{x}[n]$. If $\boldsymbol{x} = \boldsymbol{uvw}$, $\boldsymbol{u}$ is called a *prefix* of $\boldsymbol{x}$, $\boldsymbol{v}$ a *substring* of $\boldsymbol{x}$, $\boldsymbol{w}$ a *suffix* of $\boldsymbol{x}$. A prefix (substring, suffix) is *trivial* if it is empty, *proper* if not equal to $\boldsymbol{x}$.

The symbol $\mathcal{A}(\boldsymbol{x})$ denotes the *alphabet* of the string $\boldsymbol{x}$; that is, the set of all distinct symbols occurring in $\boldsymbol{x}$. A string $\boldsymbol{x}$ is said to be *over* an alphabet $\mathcal{B}$, denoted by $\boldsymbol{x} \in \mathcal{B}^*$, if $\mathcal{A}(\boldsymbol{x}) \subseteq \mathcal{B}$. If $\prec$ is a total order of $\mathcal{B}$, it can be naturally extended to a total *lexicographic* order of $\mathcal{B}^*$ (*lexorder* for short) by a simple rule: $\boldsymbol{x} \prec \boldsymbol{y}$ if either $\boldsymbol{x}$ is a proper prefix of $\boldsymbol{y}$, or $\boldsymbol{x}[1..j) = \boldsymbol{y}[1..j)$ and $\boldsymbol{x}[j] \prec \boldsymbol{y}[j]$ for some j, $1 < j \leq \min\{|\boldsymbol{x}|, |\boldsymbol{y}|\}$.

If a string $\boldsymbol{x} = \boldsymbol{uv}$, then $\boldsymbol{vu}$ is called a *rotation* of $\boldsymbol{x}$. The rotation is *trivial* if either $\boldsymbol{u}$ or $\boldsymbol{v}$ is empty. A string $\boldsymbol{x}$ is *Lyndon* [3] if $\boldsymbol{x} \prec \boldsymbol{y}$ for any non-trivial rotation of $\boldsymbol{x}$, where as above we suppose a total order $\prec$ on $\mathcal{A}(\boldsymbol{x})$. Clearly any string of length 1 is Lyndon, thus called a *trivial* Lyndon string.

Observation 1 ([4,8]) *For any $\boldsymbol{x} = \boldsymbol{x}[1..n]$, $n > 1$, the following are equivalent:*

1. *$\boldsymbol{x}$ is a non-trivial Lyndon string;*
2. *$\boldsymbol{x}[1..n] \prec \boldsymbol{x}[k..n]$ for any $1 < k \leq n$;*
3. *$\boldsymbol{x}[1..k) \prec \boldsymbol{x}[k..n]$ for any $1 < k \leq n$;*
4. *there is $1 < k \leq n$ so that $\boldsymbol{x}[1..k) \prec \boldsymbol{x}[k..n]$, both $\boldsymbol{x}[1..k)$ and $\boldsymbol{x}[k..n]$ are Lyndon.*

Item 4 of this observation is the basis for the definition of the *standard factorization* of a Lyndon string $\boldsymbol{x}$, given by $\boldsymbol{x}[1..k)\boldsymbol{x}[k..n]$ where k is the smallest integer such that $\boldsymbol{x}[1..k)$ and $\boldsymbol{x}[k..n]$ are both Lyndon.

A string is *primitive* if it is not a concatenation of two or more copies of a smaller string. A *border* of a string $\boldsymbol{x}$ is a prefix that is also a suffix; a border is *trivial* if it is empty, *proper* if it is not $\boldsymbol{x}$ itself. If $\boldsymbol{x}$ has only trivial or improper borders, it is said to be *unbordered*.

Observation 2 *For any string $\boldsymbol{x}$: $\boldsymbol{x}$ is Lyndon $\not\Rightarrow$ unbordered $\not\Rightarrow$ primitive.*

Suppose that a substring $\boldsymbol{u}$ of $\boldsymbol{x}$ is Lyndon. Then $\boldsymbol{u}$ is said to be *maximal Lyndon in $\boldsymbol{x}$* if it is not a proper prefix of any Lyndon substring of $\boldsymbol{x}$. Occasionally, we may abbreviate *maximal Lyndon* as *maxLyn*.

Theorem 1 ([6], Hohlweg and Reutenauer) *Any substring $\boldsymbol{x}[i..k]$ of string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is maximal Lyndon if and only if $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[j..n]$ for any j satisfying $i < j \leq k$, and either $k = n$ or $\boldsymbol{x}[k+1..n] \prec \boldsymbol{x}[i..n]$.*

The Lyndon array was introduced in [5] — it is closely related to the *Lyndon tree* of [6]:

Definition 1 *For a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, the* Lyndon array *of $\boldsymbol{x}$ is an integer array $L[1..n]$ such that $L[i] = j$ if and only if j is the length of the maximal Lyndon substring at i.*

We now introduce the *Lyndon grouping array*, the *partially sorted Lyndon array*, and the *sorted Lyndon array*. All three are two-dimensional arrays $\mathcal{L}[1..2][1..n]$, but for brevity we use $\mathcal{L}_1[i]$ instead of $\mathcal{L}[1][i]$, $\mathcal{L}_2[i]$ instead of $\mathcal{L}[2][i]$.

Definition 2 *(See Figure 1.) Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ be a string of length n. The* Lyndon grouping array *of $\boldsymbol{x}$ is a two-dimensional integer array $\mathcal{L}[1..2][1..n]$ such that*

1. *$\mathcal{L}_1[1..n]$ is a permutation of $1..n$;*
2. *if $\mathcal{L}_2[i] > 0$, then the maximal Lyndon substring starting at $\mathcal{L}_1[i]$ has length $\mathcal{L}_2[i]$;*
3. *if $\mathcal{L}_2[i] = 0$, then the maximal Lyndon substring starting at $\mathcal{L}_1[i]$ has length $\mathcal{L}_2[j]$ where j is the greatest integer less than i such that $\mathcal{L}_2[j] > 0$.*

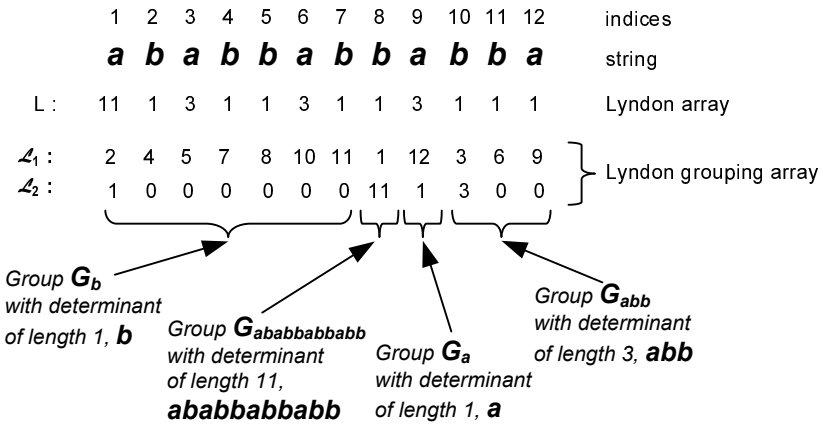

Figure 1. A Lyndon grouping array for ***ababbabbabba***

Thus, a Lyndon grouping array just partitions the positions of a string into groups determined by identical maxLyn substrings: all indices in the same group are starting positions of the same maxLyn substring. Baier calls this substring the *context* of the group [1,2] — here we will use the term *determinant*; we denote a group with a determinant $\boldsymbol{u}$ as $G_{\boldsymbol{u}}$. Note that the Lyndon grouping array is not unique; that is, for given $\boldsymbol{x}$ there may exist several such arrays with different orderings.

Lemma 1 *Let $\mathcal{L}[1..2][1..n]$ be a Lyndon grouping array of $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Then the Lyndon array L of $\boldsymbol{x}$ can be computed from $L[1..2]$ in $\emptyset(n)$ steps.*

Proof. Replacing zeros in $\mathcal{L}_2$ with the value at the start of each group yields $L[\mathcal{L}_1[i]] = \mathcal{L}_2[i]$ for all $i \in 1..n$:

for $i = 1$ **to** n **do**
  if $\mathcal{L}_2[i] \neq 0$ **then** $m \leftarrow \mathcal{L}_2[i]$
  $L[\mathcal{L}_1[i]] \leftarrow m$

□

Note that the Lyndon array may provide weaker information than a Lyndon grouping array, as the Lyndon array can be computed from a Lyndon grouping array in linear time, but we do not know at this point how to compute in linear time a Lyndon grouping array from the Lyndon array.

Definition 3 *(See Figure 2.) A* partially sorted Lyndon array *of* $\boldsymbol{x}$ *is a Lyndon grouping array whose groups are sorted in ascending lexorder; that is,*

4. *For* $i < j$ *such that* $\mathcal{L}_2[i] > 0$, $\mathcal{L}_2[j] > 0$, $\boldsymbol{x}\big[\mathcal{L}_1[i]..\mathcal{L}_1[i]+\mathcal{L}_2[i]-1\big] \prec \boldsymbol{x}\big[\mathcal{L}_1[j]..\mathcal{L}_1[j]+\mathcal{L}_2[j]-1\big]$.

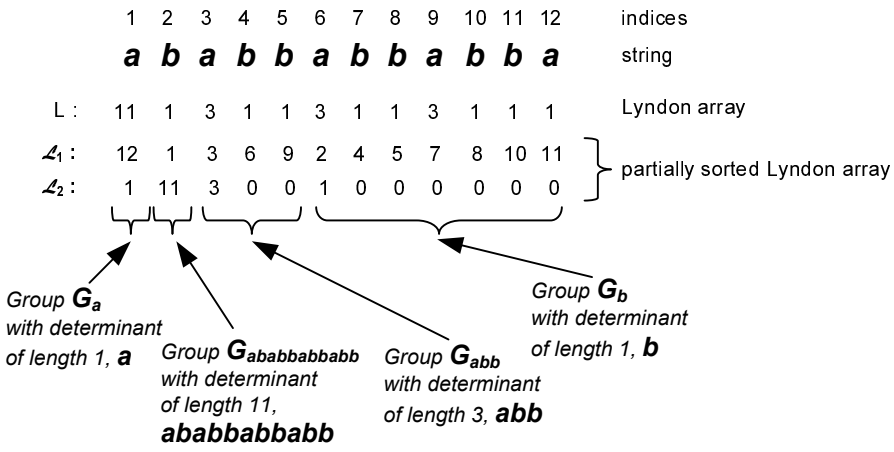

Figure 2. A partially sorted Lyndon array for ***ababbabbabba***

In Figure 2 the determinants a, $ababbabbabb$, abb, and b of the groups are sorted in ascending lexorder. However, the indices within the groups need not be in any particular order, though in our example they happen to fall in ascending order of position. Like the Lyndon grouping array, a partially sorted Lyndon array may not be unique.

Definition 4 *(See Figure 3.) A* sorted Lyndon array *of* $\boldsymbol{x}$ *is a partially sorted Lyndon array whose indices are ordered within each group in the* perfect order *according to the lexorder of the corresponding suffixes; that is,*

5. *If* $\mathcal{L}_1[i]$ *and* $\mathcal{L}_1[j]$ *belong to the same group,* $i < j \iff \boldsymbol{x}\big[\mathcal{L}_1[i]..n\big] \prec \boldsymbol{x}\big[\mathcal{L}_1[j]..n\big]$.

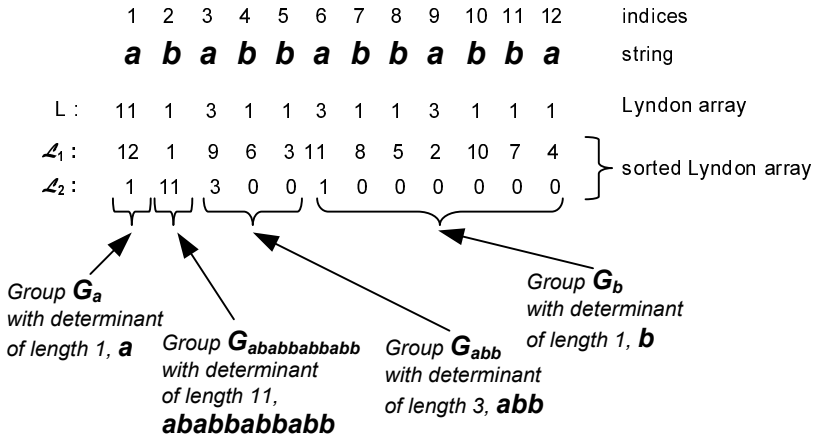

Figure 3. The sorted Lyndon array of ***ababbabbabba***

Definition 5 *(See Figure 4.)*

(a) Given $\boldsymbol{x} = \boldsymbol{x}[1..n]$*, the integer array* $SA[1..n]$ *is the* suffix array *of* $\boldsymbol{x}$ *iff the entries of* SA *form a permutation of* $1..n$ *and for every* $1 \leq i < n$, $\boldsymbol{x}\big[SA[i]..n\big] \prec \boldsymbol{x}\big[SA[i{+}1]..n\big]$.

(b) The lcp array *associated with* SA *is an integer array* $lcp[1..n]$ *in which* $lcp[i]$ *is the size of the longest common prefix of* $\boldsymbol{x}\big[SA[i]..n\big]$ *and* $\boldsymbol{x}\big[SA[i{-}1]..n\big]$ *for any* $1 < i \leq n$.

(c) The inverse suffix array $ISA[1..n]$ *is an integer array such that* $SA[i] = j$ *iff* $ISA[j] = i$.

Note that if $\mathcal{L}[1..2][1..n]$ is a sorted Lyndon array of $\boldsymbol{x}$, then in fact $\mathcal{L}_1[1..n]$ is the suffix array of $\boldsymbol{x}$. Thus, a sorted Lyndon array is unique, unlike a Lyndon grouping array and a partially sorted Lyndon array. Therefore we speak of **the** sorted Lyndon array of $\boldsymbol{x}$.

	1	2	3	4	5	6	7	8	9	10	11	12	
	a	**b**	**a**	**b**	**b**	**a**	**b**	**b**	**a**	**b**	**b**	**a**	string
SA :	12	1	9	6	3	11	8	5	2	10	7	4	suffix array
lcp :	-	1	2	4	6	0	2	5	8	1	3	6	lcp array
ISA :	2	9	5	12	8	4	11	7	3	10	6	1	inverse suffix array

Figure 4. suffix array, inverse suffix array, and lcp array of ***ababbabbabba***

3 Main Results

In this section we present the two main results tying together partially sorted Lyndon arrays and the suffix array of a string.

Theorem 2 *Let* $SA[1..n]$ *be the suffix array of a string* $\boldsymbol{x} = \boldsymbol{x}[1..n]$*. The sorted Lyndon array of* $\boldsymbol{x}$ *can be computed from* $\boldsymbol{x}$ *and* SA *in* $\mathcal{O}(n)$ *steps.*

Proof. As just observed, the top array $\mathcal{L}_1[1..n]$ is exactly the suffix array of $\boldsymbol{x}$. Thus we need only compute $\mathcal{L}_2[1..n]$. First we compute the inverse suffix array ISA from SA in $\mathcal{O}(n)$ steps. Then, as noted in [6] and explained in [5], we compute the Lyndon array $L[1..n]$ of $\boldsymbol{x}$ from ISA, also in $\emptyset(n)$ steps, using the next smaller value (NSV) algorithm. Thus we set $\mathcal{L}_2[i] = L\big[\mathcal{L}_1[i]\big]$ for every i.

To complete the calculation, we need only set the $\mathcal{L}_2$ values to zero except for the first entry in each group. For that we can use the $\mathcal{O}(n)$-time algorithm of Kasai *et al.* [7] to compute the lcp array. Then, for every i, if $lcp(\mathcal{L}_1[i], \mathcal{L}_1[i+1]) \geq \mathcal{L}_2[i]$ and $\mathcal{L}_2[i-1] = \mathcal{L}_2[i]$, we change the value of $\mathcal{L}_2[i]$ to 0. □

The reversed calculation is in essence Baier's Phase 2 algorithm. However, we will describe a different algorithm based on the same ideas. Though it is more complex to implement and requires more working memory than Baier's, it has the potential to be faster. This statement has not been verified by empirical testing, it is just based on the analysis of the implementation of the two algorithms. The actual testing will require to excise the second step from Uwe Baier's implementation.

The several following definitions are introduced only for use in the proof of Lemma 2. Thus they are not presented formally. We give them here because they are too complex to be included in the proof itself.

The *delta operator* is defined as follows: for $i \in G_{\boldsymbol{u}}$, $\Delta(i) = i+|\boldsymbol{u}|$. If $\Delta(i) \leq n$, then consider $\boldsymbol{v}$, the maxLyn substring at the position $\Delta(i)$. If $\boldsymbol{u}$ were lexicographically smaller than $\boldsymbol{v}$, then $\boldsymbol{uv}$ would be Lyndon, contradicting the maximality of $\boldsymbol{u}$. Thus, $\boldsymbol{v} \preceq \boldsymbol{u}$. It follows that for $i \in G_{\boldsymbol{u}}$ $\begin{cases} \Delta(i) = n+1, \text{ or} \\ \Delta(i) \in G_{\boldsymbol{v}} \text{ for some maxLyn } \boldsymbol{v} \prec \boldsymbol{u}, \text{ or} \\ \Delta(i) \in G_{\boldsymbol{u}}. \end{cases}$
The groups form a partition of the set of indices. Through the delta operator we define the Δ-refinement of this partition: let $\boldsymbol{u}$, $\boldsymbol{v}$ be maxLyn substrings of $\boldsymbol{x}$ so that $\boldsymbol{v} \preceq \boldsymbol{u}$, then we define the *subgroup* $G^{\boldsymbol{v}}_{\boldsymbol{u}} = \{i \in G_{\boldsymbol{u}} : \Delta(i) \in G_{\boldsymbol{v}}\}$, while we define the *subgroup* $G^{\$}_{\boldsymbol{u}} = \{i \in G_{\boldsymbol{u}} : \Delta(i) = n+1\}$.

It follows that each group $G_{\boldsymbol{u}}$ is a disjoint union of non-empty subgroups $G^{\boldsymbol{v}}_{\boldsymbol{u}}$ for all maxLyn $\boldsymbol{v} \preceq \boldsymbol{u}$ and possibly $G^{\$}_{\boldsymbol{u}}$. If $i \in G^{\boldsymbol{v}_1}_{\boldsymbol{u}}$ and $j \in G^{\boldsymbol{v}_2}_{\boldsymbol{u}}$, and $\boldsymbol{v}_1 \prec \boldsymbol{v}_2 \preceq \boldsymbol{u}$, then $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[j..n]$, as $\boldsymbol{x}[i..n] = \boldsymbol{u}\boldsymbol{v}_1\boldsymbol{w}_1$ for some $\boldsymbol{w}_1$, and $\boldsymbol{x}[j..n] = \boldsymbol{u}\boldsymbol{v}_2\boldsymbol{w}_2$ for some $\boldsymbol{w}_2$. Since $|G^{\$}_{\boldsymbol{u}}| \leq 1$, if $i \in G^{\$}_{\boldsymbol{u}}$ and $i \neq j \in G_{\boldsymbol{u}}$, then $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[j..n]$, as $\boldsymbol{x}[i..n] = \boldsymbol{u}$ and $\boldsymbol{x}[j..n] = \boldsymbol{u}\boldsymbol{w}$ for some $\boldsymbol{w}$. Thus, if we separately perfectly order the subgroup $G^{\boldsymbol{v}}_{\boldsymbol{u}}$ for each maxLyn $\boldsymbol{v} \prec \boldsymbol{u}$, then the group $G_{\boldsymbol{u}}$ will be perfectly ordered, as an important property of each subgroup $G^{\boldsymbol{v}}_{\boldsymbol{u}}$, $\boldsymbol{v} \prec \boldsymbol{u}$, is the fact that a perfect order of the group $G_{\boldsymbol{v}}$ induces a perfect order on $G^{\boldsymbol{v}}_{\boldsymbol{u}}$: we simply let i precede j only if $\Delta(i)$ precedes $\Delta(j)$. Similarly, a perfect order of $G^{1}_{\boldsymbol{u}}$, which is defined as the disjoint union of all subgroups of $G_{\boldsymbol{u}}$ except $G^{\boldsymbol{u}}_{\boldsymbol{u}}$, induces a perfect order on $G^{\boldsymbol{u}}_{\boldsymbol{u}}$.

For example, consider $\boldsymbol{x} = abb\,abb\,aa\,abb\,abb\,abb$ with $G_{abb} = \{1, 4, 9, 12, 15\}$. $G^{\$}_{abb} = \{15\}$, $G^{aaabbabbabb}_{abb} = \{4\}$, $G^{abb}_{abb} = \{1, 9, 12\}$ and $G_{abb} = G^{\$}_{abb} \cup G^{aaabbabbabb}_{abb} \cup G^{abb}_{abb}$. A perfect order of $G^{\$}_{abb}$ is 15, a perfect order of $G^{aaabbabbabb}_{abb}$ is 4. The elements of $G^{\$}_{abb}$ will be listed first, the elements of $G^{aaabbabbabb}_{abb}$. The perfect order of $G^{\$}_{abb} \cup G^{aaabbabbabb}_{abb} = \{15, 4\}$ determines the order of $G^{abb}_{abb} = \{1, 9, 12\}$. Now, $\Delta(1) = 1+3 = 4$, $\Delta(9) = 9+3 = 12$, and $\Delta(12) = 12+3 = 15$. Thus, 12 goes before 1, and then goes 9, i.e. the perfect order of G_{abb} is 15, 4, 12, 1, 9.

Lemma 2 *Let $\mathcal{L}[1..2][1..n]$ be a partially sorted Lyndon array of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Then in $\mathcal{O}(n)$ steps we can order the items in the groups to obtain the sorted Lyndon array.*

We only present here a sketch of the proof, as a complete proof would require an analysis of the code of the algorithm and so would exceed the scope of this contribution. However, for the interested reader, a `C++` implementation is available at `http://www.cas.ca/~franek/research.html/ub.cpp` for viewing, analysis, and testing.

Proof. We can achieve the desired ordering of $\mathcal{L}[1..2][1..n]$ by computing the suffix array SA of $\boldsymbol{x}$ and copying it into $\mathcal{L}_1[1..n]$.

First we compute triples $(I[i], G[i], SG[i])$ for $i \in 1..n$, where $I[i] = \mathcal{L}_1[i]$, $G[i]$ represent group (we are using integers $1..n$ to represent groups, and using $\Delta(i)$ we compute the subgroups (we are using integers $0..n$ to represents the subgroups). This can be achieved in two traversals.

Then we use a radix sort to sort the triples to be ascending in G and within each group to be ascending in SG. This can be achieved in six traversals.

In two traversals we can compute the inverse Δ relation, i.e. $i \in \Delta^{-1}(j)$ iff $\Delta(i) = j$.

Then we traverse the inverse Δ relation Δ^{-1} and record the indices as we encounter them. As explained in the text before this lemma, the perfect order of the previous groups induces a perfect order on the current group via the Δ operator. □

Theorem 3 *Let $\mathcal{L}[1..2][1..n]$ be a partially sorted Lyndon array of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$. The suffix array $SA[1..n]$ of $\boldsymbol{x}$ can be computed from $\boldsymbol{x}$ and $\mathcal{L}$ in $\mathcal{O}(n)$ steps.*

Proof. Using Lemma 2, we can compute the sorted Lyndon array $\mathcal{L}[1..2][1..n]$ of $\boldsymbol{x}$ by perfectly ordering $\mathcal{L}$. As previously noted, $\mathcal{L}[1][1..n]$ is then the suffix array of $\boldsymbol{x}$. □

4 Conclusion

Three arrays — *Lyndon grouping array*, *partially sorted Lyndon array*, *sorted Lyndon array* — have been introduced to formalize the notion of what is meant by *sorting the maximal Lyndon substrings*. The mutual relationship of these arrays has been examined and we have shown in what way the sorting of all maximal Lyndon substrings and sorting of suffixes of a string relate to each other.

Uwe Baier observed in [1,2], that his algorithm was slower than the state-of-the-art suffix sorting algorithms. He ascribed that to the early stages of the existence of his non-recursive approach and conjectured that with time, the approach would become more refined and thus faster. In essence, Phase 1 in Uwe Baier's algorithm is a direct construction of a partially sorted Lyndon array, which in Phase 2 is perfectly ordered to give the suffix array. The proof of Theorem 2 actually shows how much extra work is needed to get from the suffix array to a sorted Lyndon array. Thus, it seems to us that computing a partially sorted Lyndon array is essentially a harder task than "plain sorting" of the suffixes. So, maybe, no algorithm for computing a partially sorted Lyndon array can be as fast as sorting of suffixes, which in no way detracts from Uwe Baier's discovery of the deep connection hitherto unnoticed between the order of maximal Lyndon substrings and the order suffixes of a string.

In the diagram in Fig. 5, the arrow represent "simple linear computation". The diagram summarizes the relationships among the various arrays we were investigating. The two arrows with ? represent open questions: *Can a Lyndon array be used in a simple linear computation to compute a Lyndon grouping array?* and *Can a Lyndon grouping array be used in a simple linear computation to compute a sorted Lyndon array?* Note that Phase 1 of Uwe Baier's algorithm basically says **Yes** to both these questions. However it is not using any Lyndon array or Lyndon grouping array, it just computes it directly from the string. Maybe, having a Lyndon array or Lyndon grouping array can simplify the computation. From our point of view, having been interested in computation of Lyndon arrays, answer to the first question is much more interesting.

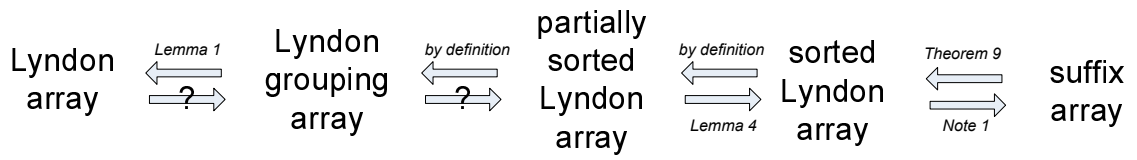

Figure 5.

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada Discovery Grants (Franek, Smyth). The authors wish to thank Christoph Diegelmann who in reaction to questions posed in [5] brought the work of Uwe Baier to their attention and indicated that Phase 1 of Baier's algorithm in fact computes the Lyndon array of the input string.

References

1. U. Baier: *Linear-time suffix sorting — a new approach for suffix array construction.* M.Sc. Thesis, University of Ulm, 2015.
2. U. Baier: *Linear-time suffix sorting — a new approach for suffix array construction*, in 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016), R. Grossi and M. Lewenstein, eds., vol. 54 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2016, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 23:1–23:12.
3. K. T. Chen, R. H. Fox, and R. C. Lyndon: *Free differential calculus. iv. the quotient groups of the lower central series.* Annals of Mathematics, 68(1) 1958, pp. 81–95.
4. J.-P. Duval: *Factorizing words over an ordered alphabet.* J. Algorithms, 4(4) 1983, pp. 363–381.
5. F. Franek, A. S. Islam, M. S. Rahman, and W. Smyth: *Algorithms to compute the Lyndon array*, in Proceedings of Prague Stringology Conference 2016, PSC'16, 2016, pp. 172–184.
6. C. Hohlweg and C. Reutenauer: *Lyndon words, permutations and trees.* Theoretical Computer Science, 307(1) 2003, pp. 173 – 178, {WORDS}.
7. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in Combinatorial Pattern Matching: 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1–4, 2001 Proceedings, A. Amir, ed., Berlin, Heidelberg, 2001, Springer Berlin Heidelberg, pp. 181–192.
8. B. Smyth: *Computing Patterns in Strings*, Pearson Addison-Wesley, 2003.