Linear Construction of a Left Lyndon Tree*

Golnaz Badkobeh† and Maxime Crochemore‡

September 21, 2021

Abstract

We extend the left-to-right Lyndon factorisation of a word to the left Lyndon tree construction of a Lyndon word. It yields an algorithm to sort the prefixes of a Lyndon word according to the infinite ordering defined by Dolce et al. (2019). A straightforward variant computes the left Lyndon forest of a word. All algorithms run in linear time on a general alphabet, that is, in the letter-comparison model.

Keywords: Lyndon tree, infinite ordering, prefix sorting, linear algorithm, general alphabet.

1 Lyndon words

In this article we consider algorithmic questions related to Lyndon words. Introduced in the field of combinatorics by Lyndon (see [14]) and used in algebra, these words have shown their usefulness for designing efficient algorithms on words. The notion of Lyndon tree associated with the decomposition of a Lyndon word has, for example, been used by Bannai et al. [2] to solve a conjecture of Kolpakov and Kucherov [12] on the maximal number of runs (maximal periodicities) in words, following a result in [4].

The key result in [2] is that every run in a word y contains as a factor a Lyndon root (according to the alphabet ordering or its inverse) that corresponds to a node of the associated Lyndon tree. Since the Lyndon tree has a linear number of nodes according to the length of y, browsing all its nodes leads to a linear-time algorithm in order to report all the runs occurring in y. However, the time complexity of this technique also depends on the time it takes to build the tree and to extend a potential run root to an actual run.

Here we consider the left Lyndon tree of a Lyndon word y. This binary tree has a single node if y is reduced to a single letter, otherwise its structure

*Revision and extension of a contribution to Prague Stringology Conference 2020 [1]

†Goldsmiths, University of London, New Cross, London SE14 6NW, UK. g.badkobeh@gold.ac.uk

‡King's College London, Informatics, 30 Aldwych, London WC2B 4BG, UK, and Université Gustave Eiffel, 77454 Marne-la-Vallée, France. Maxime.Crochemore@kcl.ac.uk

parallels recursively the left standard factorisation (see Viennot [18]) of y as uv where u is the longest proper Lyndon prefix of y.

The dual notion of right Lyndon tree of a Lyndon word y (based on the factorisation $y = uv$ where v is the longest proper Lyndon suffix of y) is strongly related to the sorted list of suffixes of y. Indeed, Hohlweg and Reutenauer [11] showed that the tree is the Cartesian tree built from ranks of suffixes in their lexicografically sorted list (see [6]). The list corresponds to the standard permutation of suffixes of the word and is the main component of its suffix array (see [15] or `en.wikipedia.org/wiki/Suffix_array`), one of the major data structures for text indexing.

Inspired by a result of Ufnarovskij [17], Dolce et al. [8] showed that the left Lyndon tree is also a Cartesian tree built from the ranks of prefixes sorted according to an ordering they call the infinite order.

The main result of this article is to show that sorting prefixes of a Lyndon word according to the infinite ordering can be attained in linear time in the letter-comparison model. This produces the prefix standard permutation of the word. The algorithm is based on the Lyndon factorisation of words by Duval [9] and it extends naturally to build the left Lyndon forest of a word. Furthermore, recovery of a word from its prefix standard permutation can be made in linear time.

Recently, Bille et al. [3] designed an algorithm to build the right Lyndon table of a word in linear time on a general alphabet, result from which the right Lyndon tree can be deduced with the same time complexity. The reverse-engineering question on this table is discussed by Nakashima et al. in [16].

Definitions

Let A be an alphabet with an ordering $<$ and A^+ be the set of non-empty words with the lexicographical ordering induced by $<$. The length of a word w is denoted by $|w|$. We say that uv (formally (u, v)) is a non-trivial factorisation of a word w if $uv = w$ and both u and v are non-empty words.

A word is said to be strongly less than a word v, denoted by $u \ll v$, if there are words r, s and t, and letters a and b satisfying $u = ras$, $v = rbt$ and $a < b$. And a word u is smaller than a word v, $u < v$, if either $u \ll v$ or u is a proper prefix of v.

In addition to the usual lexicographical ordering, the infinite ordering denoted by $\prec$ (see [7, 8]) is defined by: $u \prec v$ if $u^\infty < v^\infty$ or both $u^\infty = v^\infty$ and $|u| > |v|$. Note that the condition $u^\infty = v^\infty$ implies that u and v are powers of the same word, consequence of Fine and Wilf's Periodicity lemma (see [13, Proposition 1.3.5]).

Let $\mathcal{L}$ be the set of Lyndon words on the alphabet A. The next proposition defines Lyndon words that are not reduced to a single letter. Condition in item (i) is the original definition and condition in item (iii) is by Ufnarovskij [17].

Proposition 1 *Any of the following equivalent conditions define a Lyndon word w, $|w| > 1$: (i) $w < vu$, for any non-trivial factorisation uv of w, (ii) $w < v$, for*

any proper non-empty suffix v of w, (iii) $u^\infty < w^\infty$, for any proper non-empty prefix u of w.

2 Lyndon suffix table

Algorithms presented in the article strongly use the notion of Lyndon suffix table of a word, which is denoted by *LynS*. The table *LynS* (more accurately $LynS_y$) of a word y is defined, for each position j on y, by

$$LynS[j] = \max\{|w| \mid w \text{ is the longest Lyndon suffix of } y[0\,..\,j]\}.$$

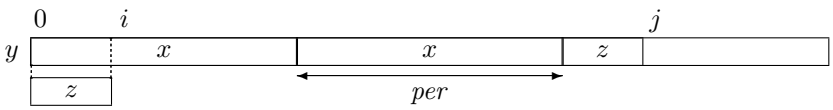
For $y = \texttt{babbababbaabb}$ on the alphabet of constant letters $\{\texttt{a}, \texttt{b}, \ldots\}$ ordered as usual $\texttt{a} < \texttt{b} < \cdots$, the *LynS* table is as follows:

j	0	1	2	3	4	5	6	7	8	9	10	11	12
$y[j]$	b	a	b	b	a	b	a	b	b	a	a	b	b
$LynS[j]$	1	1	2	3	1	2	1	2	5	1	1	3	4

Table *LynS* is the dual notion of the Lyndon table of y (also called Lyndon array) l in [2], $\mathcal{L}$ in [10] or *Lyn* in [6, 5], used to detect maximal periodicities (runs) in words: $Lyn[j]$ is the maximal length of Lyndon prefixes of $y|j\,..\,|y|-1]$.

The computation of *LynS* is a mere extension of the algorithm for testing if a word is the prefix of a Lyndon word. It includes the key point of the factorisation algorithm in [9] and is recalled first as Algorithm LyndonWordPrefix that tests if the input is a prefix of a Lyndon word and it works online. Note that the input is a Lyndon word its final period must equal its length.

```
LyndonWordPrefix(y non-empty word of length n)
 1  (per, i) ← (1, 0)
 2  for j ← 1 to n − 1 do
 3       if y[j] > y[i] then          ▷ y[i] = y[j − per]
 4            (per, i) ← (j + 1, 0)
 5       elseif y[j] < y[i] then
 6            return FALSE
 7       else i ← i + 1 mod per
 8  return TRUE
```



The key feature of the method stands in lines 3-4 of the algorithm and is illustrated in the picture above. If $y[j] > y[i] = y[j - per]$, not only the periodicity *per* of $y[0\,..\,j-1]$ breaks but it also indicates that $y[0\,..\,j]$ is a Lyndon word with period $j + 1$. This results from the following known properties (see [13]).

Proposition 2 *(i) Let z be a word and a a letter for which za is a prefix of a Lyndon word and let b be a letter with $a < b$. Then zb is a Lyndon word.*
(ii) Let u and v be two Lyndon words with $u < v$. Then uv is a Lyndon word.

Algorithm LyndonSuffixT below computes the Lyndon suffix table of a Lyndon word. (It is extended in Section 6 to compute the same table of a non-empty word.) The algorithm results from a minor modification of Algorithm LyndonWordPrefix and can be easily enhanced to compute also the period of all non-empty prefixes of the input.

```
LyndonSuffixT(y Lyndon word of length n)
 1  LynS[0] ← 1
 2  (per, i) ← (1, 0)
 3  for j ← 1 to n − 1 do
 4       if y[j] ≠ y[i] then          ▷ y[j] > y[i] = y[j − per]
 5            LynS[j] ← j + 1
 6            (per, i) ← (j + 1, 0)
 7       else LynS[j] ← LynS[i]
 8            i ← i + 1 mod per
 9  return LynS
```

Proposition 3 *Algorithm* LyndonSuffixT *computes the Lyndon suffix table of a Lyndon word of length n in time $O(n)$ in the letter-comparison model.*

Given $y =$ `ababbababbabac`, the corresponding *LynS* table and period table are as follows:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$y[j]$	a	b	a	b	b	a	b	a	b	b	a	b	a	c
$LynS[j]$	1	2	1	2	5	1	2	1	2	5	1	2	1	14
$period[j]$	1	2	2	2	5	5	5	5	5	5	5	5	5	14

3 Left Lyndon tree construction

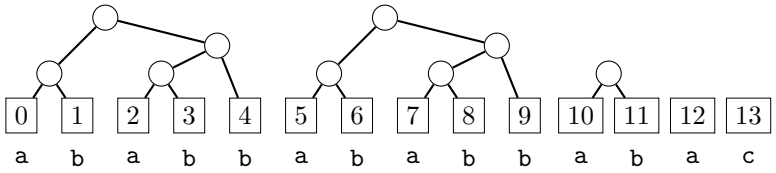
The left Lyndon tree $\mathcal{L}(y)$ of a Lyndon word y represents recursively the left standard factorisation of y. It is a binary tree whose leaves are positions on the word and internal nodes correspond to concatenations of two Lyndon factors of the word, and as such can be viewed as interpositions. Precisely, $\mathcal{L}(y) = (p)$ if $|y| = 1$ else it is $(p, \mathcal{L}(u), \mathcal{L}(v))$ where the node $p \in \{|y| \,..\, 2|y| - 2\}$ is an integer and uv is the left standard factorisation of y, that is, u is the longest proper Lyndon prefix of y (v is then a Lyndon word).

In the next algorithm, subtrees of $\mathcal{L}(y)$ are handled from positions on y as follows. The subtree associated with position j is $\mathcal{L}(y[i \,..\, j])$ where $j - i + 1 = LynS[j]$ and its root is $root[j]$. Thus, position j on y is the rightmost leaf of the

subtree and $LynS[j]$ is its tree width. Besides, the left child of an internal node q is $left(q)$ and its right child is $right(q)$.

It is known that y, as a Lyndon word with $|y| > 1$, is of the form $x^k zb$ where x is a Lyndon word of length $per = period(x^k z)$, $k > 0$, z is a proper prefix of x and b is a letter greater than letter a following z in x (za is a prefix of x) [9].

The construction of $\mathcal{L}(y)$ is achieved with the help of the table $LynS$ of y. It is done by processing y from left to right building first $\mathcal{L}(x)$ and reproducing that tree or part of it up to z. The picture displays the subtrees built for the word $(\texttt{ababb})^2\texttt{abac}$ just before processing the letter `c`.

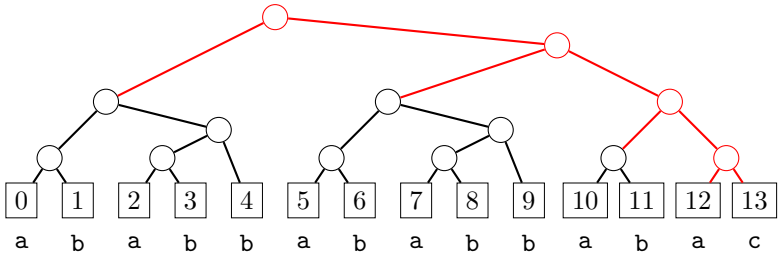


The main step of the procedure, in addition to computing $LynS$ identically as in Algorithm LyndonSuffixT above, is to aggregate partial Lyndon trees when processing the last letter b of y, which creates the final tree as a bundle of all subtrees. In fact, this step is also carried out when dealing with $x^k z$ at each position j for which $LynS[j] > 1$. In order to aggregate the subtrees, the second property of Proposition 2 is applied iteratively, processing the trees from right to left. An explicit instruction of this step is designed at lines 10-15 in Algorithm LeftLyndonTree below.

The process of bundling can be viewed as a translation into the tree structure of the proof of the key feature of Algorithm LyndonWordPrefix stated in Proposition 2 item (ii). Even so the latter algorithm deals with this process in constant time using item (i) of the proposition. Consequently, the iteration of instructions during the bundling does not affect the asymptotic running time of the present algorithm.

```
LeftLyndonTree(y Lyndon word of length n)
 1  (LynS[0], root[0]) ← (1, 0)
 2  (per, i) ← (1, 0)
 3  for j ← 1 to n − 1 do
 4      root[j] ← j
 5      if y[j] ≠ y[i] then          ▷ y[j] > y[i] = y[j − per]
 6          LynS[j] ← j + 1
 7          (per, i) ← (j + 1, 0)
 8      else LynS[j] ← LynS[i]
 9          i ← i + 1 mod per
10      (ℓ, k) ← (1, j − 1)
11      while ℓ < LynS[j] do
12          q ← new node ≥ n
13          (left[q], right[q]) ← (root[k], root[j])
14          root[j] ← q
15          (ℓ, k) ← (ℓ + LynS[k], k − LynS[k])
16  return root[n − 1]
```

The picture below shows red nodes and links created by the final round of instructions at lines 10-15 in Algorithm LeftLyndonTree.



Proposition 4 *Algorithm* LeftLyndonTree *builds the left Lyndon tree of a Lyndon word of length n in time $O(n)$ in the letter-comparison model.*

Proof. All instructions inside the **for** loop are executed in constant time except for the **while** loop. In addition, since each execution of instructions in the **while** loop takes constant time and leads to the creation of an internal node of the final tree twinned with the fact that there are eactly $n-1$ such nodes, the total (amortised) running time is $O(n)$. ■

4 Sorting prefixes

This section shows that Algorithm LeftLyndonTree can be adapted to sort the prefixes of a Lyndon word according to the infinite ordering $\prec$. This is a

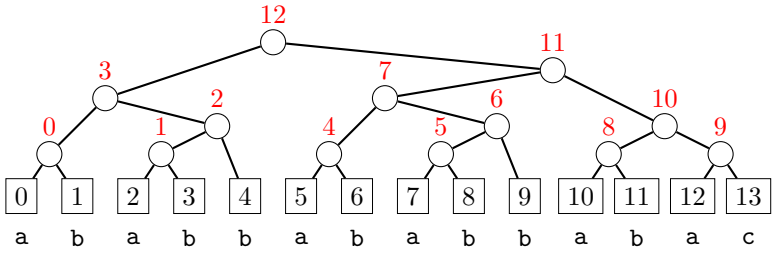
consequence of Theorem 6 below.

For the Lyndon word y, an internal node p of the left Lyndon tree $\mathcal{L}(y)$ is the root of a Lyndon subtree associated with a Lyndon factor w of y. This factor is obtained by concatenating two consecutive occurrences of Lyndon factors u and v. If the concerned occurrence of u ends at position j on y, node p is identified with the prefix of y ending at position j. The correspondence between internal nodes of the tree and proper non-empty prefixes of y is one-to-one (see picture below).

Labelling internal nodes with the $\prec$-ranks of their associated prefixes transforms the tree into a heap, i.e. ranks are increasing from leaves to the root. The relation between the infinite ordering and left Lyndon trees is established by the next result [8].

Theorem 5 (Dolce, Restivo, Reutenauer, 2019) *For a Lyndon word y, the tree of internal nodes of the left Lyndon tree $\mathcal{L}(y)$ in which nodes are labelled by the ranks of proper non-empty prefixes of y sorted according to the infinite ordering is the Cartesian tree of prefix ranks.*

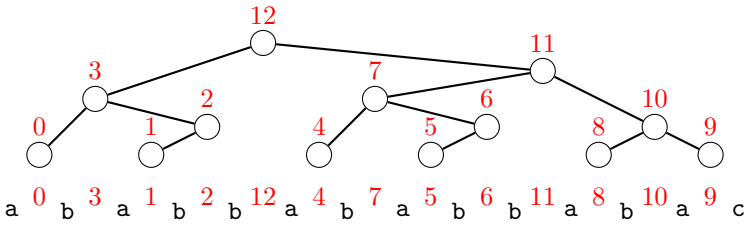
The picture below shows the left Lyndon tree of `ababbababbabac` and the $\prec$-rank labels of its internal nodes.



Denoting a non-empty prefix of y by the position of its last letter, the tables below show both $\prec$-ranks of proper non-empty prefixes of $y =$ `ababbababbabac` and its sorted list of prefixes, called the prefix standard permutation of y in [8]. They are denoted by *rank* and *psp* and are inverse of each other when considered as functions from $(0, 1, \ldots, |y| - 2)$ to itself. The sorted list is $(0, 2, 3, 1, 5, 7, 8, 6, 10, 12, 11, 9, 4)$, that is, `a` $\prec$ `aba` $\prec$ `abab` $\prec$ `ab` $\prec$ `ababba` $\prec$ `ababbaba` $\prec$ `ababbabab` $\prec$ `ababbab` $\prec$ `ababbababba` $\prec$ `ababbababbaba` $\prec$ `ababbababbab` $\prec$ `ababbababb` $\prec$ `ababb`.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$y[j]$	a	b	a	b	b	a	b	a	b	b	a	b	a	c
$rank[j]$	0	3	1	2	12	4	7	5	6	11	8	10	9	
r	0	1	2	3	4	5	6	7	8	9	10	11	12	
$psp[r]$	0	2	3	1	5	7	8	6	10	12	11	9	4	

The tree below is the Cartesian tree of prefix $\prec$-ranks of y.



The next theorem is the computational complement of Theorem 5 showing additionally that the construction of the left Lyndon tree by Algorithm LeftLyndonTree processes the nodes of the tree in a left-to-right postorder traversal.

Theorem 6 *Algorithm* LeftLyndonTree *applied to a Lyndon word y of length $n > 1$, creates and processes internal nodes of the tree $\mathcal{L}(y)$ in the order of their corresponding prefix ranks according to the infinite ordering $\prec$.*

Proof. Since word y is a Lyndon word and $|y| > 1$, it is of the form $x^k z b$ where x is a Lyndon word of length $period(x^k z)$, $k > 0$, z is a proper prefix of x and b is a letter greater than letter a following prefix z in x (see [9]).

Algorithm LeftLyndonTree processes nodes of the tree $\mathcal{L}(y)$ as follows. First it builds $\mathcal{L}(x)$ and Lyndon subtrees of the next occurrences of x in a left to right manner. Next, it builds the trees related to z. Eventually during the last bundling (run of instructions at lines 10-15) the algorithm builds $\mathcal{L}(zb)$ and follows with the nodes corresponding to the concatenations $x \cdot zb$, $x \cdot xzb$, ..., $x \cdot x^{k-1}zb$ in that order.

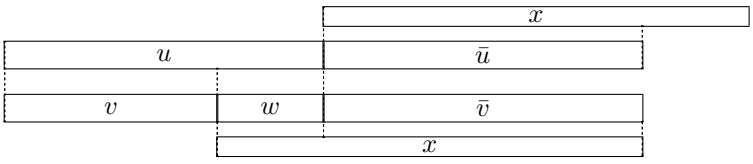
We will prove the statement by induction on the length of the period $|x|$ of $x^k z$. If $|x| = 1$, x is reduced to a single letter and y is of the form $a^k b$ for two letters a and b with $a < b$. Nodes associated with prefixes a^k, a^{k-1}, ..., a are processed in this order, which matches the $\prec$-order of prefixes, $a^k \prec a^{k-1} \prec \cdots \prec a$, as expected.

We then assume $|x| > 1$ and consider disjoint groups of non-empty proper prefixes of y. For $e = 0, 1, \ldots, k$, let

$$P_e = \{x^e u \text{ prefix of } y \mid e|x| < |x^e u| < \min\{(e+1)|x|, |y|\}\}.$$

The main part of the proof relies on the following three claims that we prove first.

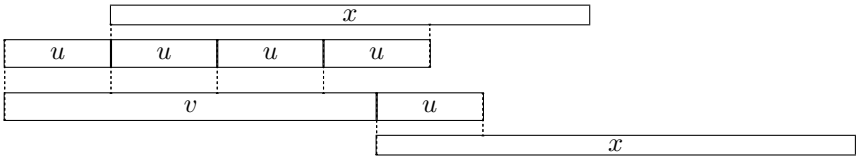
Claim 1: *prefixes $x^e u \in P_e$, $0 < e \leq k$, are in the same relative $\prec$-order as prefixes $u \in P_0$.* Let $u, v \in P_0$ with $u \prec v$ and let us show $x^e u \prec x^e v$ considering two cases.



Case $u^\infty = v^\infty$ and $|u| > |v|$. By the Periodicity lemma u, v and $v^{-1}u$ are powers of the same word. Let $w = v^{-1}u$, $\bar{v} = w^{-1}x$ and $\bar{u}$ the prefix of x of length $|\bar{v}|$ (see picture). Since x is a Lyndon word, $\bar{u} < x < \bar{v}$, which implies $ux < vx$ because w is a prefix of x. Therefore we have $(x^e u)^\infty < (x^e v)^\infty$, that is, $x^e u \prec x^e v$.

Case $u^\infty < v^\infty$. Assume u is shorter than v and let h be the largest exponent for which u^h is a prefix of v. It is a proper prefix because $u^\infty < v^\infty$ and then $w = (u^h)^{-1}v$ is not empty.

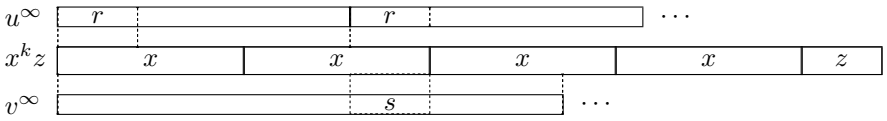
If $|u| \leq |w|$, we have $u \ll w$, which implies $ux \ll vx$ and $(x^e u)^\infty < (x^e v)^\infty$, that is, $x^e u \prec x^e v$.



If $|u| > |w|$, v is a proper prefix of u^{h+1} but u^{h+1} shorter than vu cannot be a prefix of it due to the Periodicity lemma applied on periods $|u|$ and $|v|$ of u^{h+1}. Then $u \ll wu$ and since u is a prefix of x it implies $ux \ll vx$ and $(x^e u)^\infty < (x^e v)^\infty$, that is, $x^e u \prec x^e v$ as before.

The situation in which u is longer than v is fairly symmetric and treated similarly. Therefore again $u \prec v$ implies $x^e u \prec x^e v$, which proves the claim.

Claim 2: *prefixes in P_e are $\prec$-smaller than prefixes in P_f when $0 \leq e < f \leq k$.* Let $u \in P_e$ and $v \in P_f$. We have to compare u and v according to $\prec$, that is, to compare u^∞ and v^∞.



When $e > 0$, u is longer than x. Let r be the prefix of u for which $|ur| = |x^{e+1}|$ (see picture in which $u \in P_1$ and $v \in P_2$) and s the suffix of x of the same length. Comparing u^∞ and v^∞ amounts to compare r and s because u is a prefix of v. Since r is a prefix and s a suffix of the Lyndon word x, we have $r < s$ and even $r \ll s$, then $u^\infty < v^\infty$ and $u \prec v$.

When $e = 0$, u is shorter than x. Let h be the largest integer for which u^h is a proper prefix of x. It is a proper prefix because x is a Lyndon word and $w = (u^h)^{-1}x$ is not empty. As in the proof of previous claim, u^{h+1} cannot be prefix of xu that is a prefix of v. The same conclusion follows, that is, $u^{h+1} \ll vu$ and eventually $u \prec v$.

Claim 3: *prefixes in P_e, $0 \leq e \leq k$, are $\prec$-smaller than prefixes x^f, $0 < f \leq k$.* To prove the claim, in view of the statement of Claim 2 and the fact $x^k \prec x^{k-1} \prec x$ by definition, it is enough to show that $P_k \prec x^k$. Note that if

P_k is empty the proof can be done with P_{k-1} instead, and if in addition $k = 1$ then this case is part of Claim 2.

Let $x^k u \in P_k$, $s = u^{-1}x$ and r the prefix of x of length $|s|$. As prefix and suffix of x, r and s satisfy $r < s$. Since $x^k ur < x^k us = x^{k+1}$ and r is a prefix of x, it results $(x^k u)^\infty < x^\infty$ and eventually $x^k u \prec x^k$. This proves the claim.

To summarise, claims show

$$P_0 \prec P_1 \prec \cdots \prec P_k \prec x^k \prec x^{k-1} \prec \cdots \prec x.$$

Let us go back to induction. By induction hypothesis, the result holds for internal nodes of $\mathcal{L}(x)$ corresponding to prefixes in P_0.

Consider the next occurrences of x. Since the Lyndon suffix table for each of them is copied from that of prefix x due to the instruction at line 8 in Algorithm LeftLyndonTree, the Lyndon trees of all occurrences of x have the same structure. Therefore, both from the induction hypothesis and from Claim 1, the order in which internal nodes of the eth occurrence of x are processed and created matches the $\prec$-order of prefixes in P_e, for $0 < e \leq k$.

The algorithm processes occurrences of x from left to right, which corresponds to the result of Claim 2. The treatment of zb is done at the beginning of the bundling run, which also corresponds to the fact that prefixes in P_k are $\prec$-larger than all prefixes that have been considered before.

Finally, the last part of the bundling creates nodes associated with x^k, x^{k-1}, $\ldots$, x in that order, which matches the order $x^k \prec x^{k-1} \prec \cdots \prec x$.

This ends the proof of the theorem. ■

An immediate consequence of Theorem 6 is that Algorithm LeftLyndonTree can be down-graded and adapted to compute directly the $\prec$-sorted list of non-empty proper prefixes of a Lyndon word, that is, to compute its prefix standard permutation (PSP). See the details of this adaptation in the following algorithm.

```
PrefixStandardPermutation(y Lyndon word of length n)
 1  psp ← ()
 2  (LynS[0], per, i) ← (1, 1, 0)
 3  for j ← 1 to n − 1 do
 4       if y[j] ≠ y[i] then          ▷ y[j] > y[i] = y[j − per]
 5            LynS[j] ← j + 1
 6            (per, i) ← (j + 1, 0)
 7       else LynS[j] ← LynS[i]
 8            i ← i + 1 mod per
 9       (m, k) ← (1, j − 1)
10       while m < LynS[j] do
11            psp ← psp · (j − m)
12            m ← m + LynS[k]
13            k ← k − LynS[k]
14  return psp
```

Corollary 7 *Sorting the proper non-empty prefixes of a Lyndon word of length n according to the infinite ordering $\prec$ can be done in time $O(n)$ in the letter-comparison model.*

Proof. It suffices to substitute the handling of sequence *psp* to the processing of internal nodes of the Lyndon tree in Algorithm LeftLyndonTree. The change is realised by Algorithm PrefixStandardPermutation above. ■

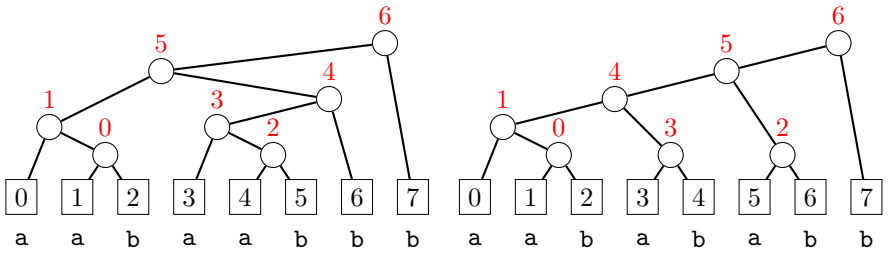
5 Reverse-engineering a PSP

This section discusses the recovery of a word of length n from a permutation of $(0, 1, \ldots, n-2)$ assumed to be its prefix standard permutation (PSP).

We first consider the case of binary words on the alphabet $\{\mathtt{a}, \mathtt{b}\}$. Function *psp* from $\mathcal{L}_n = \mathcal{L} \cap \{\mathtt{a}, \mathtt{b}\}^n$ to the set of permutations of $(0, 1, \cdots, n-2)$ is one-to-one. Thus psp^{-1} is a function from $psp(\mathcal{L}_n)$ to $\mathcal{L}_n$ and $psp^{-1}(psp(y)) = y$. To show the property, given a permutation p of $(0, 1, \cdots, n-2)$, we propose the following algorithm to recover the possible word y that admits the permutation as its PSP.

```
InversePsp(p permutation of (0, 1, ..., n − 2))
 1  rank ← inverse of p
 2  C ← Cartesian tree of rank
 3  T ← C extended with leaves to form a complete binary tree
 4  L ← labelled T: left-child leaves labelled by a others by b
 5  y ← word-label of leaves of L
 6  if psp(y) = p then
 7        return y
 8  else return p ∉ psp(L_n)
```

From the permutation $p = (1, 0, 4, 3, 5, 2, 6)$ the algorithm computes $rank = (1, 0, 5, 3, 2, 4, 6)$ and eventually the labelled Lyndon tree below left. The word label of its leaves is `aabaabbb` and effectively $psp(\mathtt{aabaabbb}) = (1, 0, 4, 3, 5, 2, 6)$.



However with the permutation $p = (1, 0, 5, 3, 2, 4, 6)$, the algorithm computes $rank = (1, 0, 4, 3, 5, 2, 6)$ and the correponding L tree (above right), which produces the word `aababbab`. But $psp(\texttt{aababbab}) = (1, 0, 3, 2, 5, 4, 6)$ is not the input permutation. This is because obviously not all the $(n-1)!$ permutations are PSPs of some binary Lyndon words (less than 2^n). It may also happen that word y built in the procedure is not even a Lyndon word.

Proposition 8 *On a binary alphabet the prefix standard permutation is a one-to-one function and computing the Lyndon word y for which $psp(y)$ is a given valid permutation can be done in linear time.*

Proof. From the above discussion and Algorithm InversePsp, the one-to-one feature is a consequence of Theorem 5. As for the running time, it comes from the linearity of all operations, especially those of the Cartesian tree construction[1] and of the prefix standard permutation computation by Algorithm PrefixStandardPermutation in Section 4. ■

On alphabets with more than two letters the function *psp* is not one-to-one. For example $(0, 2, 3, 1, 4)$ is the PSP of Lyndon words `ababbb`, `ababbc`, `ababcb` and `ababcc`, and permutation $(0, 1, 2, 3)$ is the PSP of all (Lyndon) words in $\texttt{a}\{\texttt{b}, \texttt{c}\}^4$.

Nevertheless, given the permutation $p = psp(z)$ associated with a Lyndon word z of length n, we can compute an equivalent word y whose PSP is p. The simplest approach to carry out this computation is to deal with prefix periods of the word.

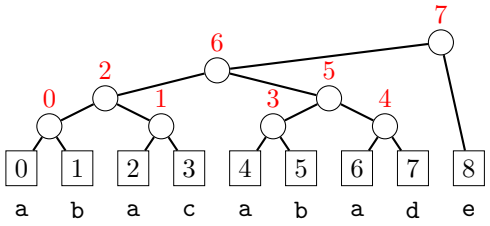
Indeed, periods of prefixes of y can be retrieved from p by looking at some positions where p is decreasing. Due to properties (proof of theorem 6, after claim 3; the only case where a longer prefix is $\prec$-smaller than a shorter prefix is when the shorter one is a period of the longer) and the definition of the prefix standard permutation, when there is a decrease in the order of prefixes it is because there is a non-empty border. Therefore scanning p from right to left enables tracing the periodicity of each proper prefix. This is how Algorithm PeriodsFromPsp computes the period table of a word from its PSP.

```
PeriodsFromPsp(p PSP of a Lyndon word of length n)
 1  q ← n
 2  for j ← n − 2 downto 1 do
 3      if j ≥ q then
 4          per[j] ← q
 5      elseif p[j] < p[j − 1] then
 6          per[j] ← q ← p[j] + 1
 7      else per[j] ← j + 1
 8  per[0] ← 1
 9  return per
```

[1] See for example `https://en.wikipedia.org/wiki/Cartesian_tree`

In the example below, positions on the PSP p, where condition at line 5 is met, are $j = 6$ and $j = 2$ corresponding respectively to periods 4 and 2.

Here is the step-by-step computation of the periods. For the following example, we start at $j = 7$, since $p[j] > p[j-1]$ then $per[j] = j + 1 = 8$, now $p[6] < p[5]$ which means $per[6] = p[6] + 1 = 4$. Next, we can move on to position $p[6] - 1$; $p[2] < p[1]$ so $per[2] = 1 + 1$, and we are done.



j	0	1	2	3	4	5	6	7	8
$y[j]$	a	b	a	c	a	b	a	d	e
$psp[j]$	0	2	1	4	6	5	3	7	
$rank[j]$	0	2	1	6	3	5	4	7	
$per[j]$	1	2	2	4	4	4	4	8	9

Another way to retrieve periods of prefixes is to look at prefix ranks according to the infinite order. To do so amounts to looking at ranks of proper Lyndon prefixes of y, because their periods are their lengths, starting with the first rank, r. Then the next Lyndon prefix is the shortest prefix having a rank greater than r, which is iterated until the end. This amounts to going up the left Lyndon tree from its leftmost leaf to its root. In the example (above) positions on the rank table that correspond to the traversal are $j = 0, 1, 3, 7$.

Following the discussion, Algorithm WordFromPsp takes as input the PSP p of a Lyndon word and builds an equivalent word, that is, a Lyndon word having the same PSP. The output is a word on the (constant) alphabet $\{\mathtt{a}, \mathtt{b}, \cdots\}$. If $y \in \mathcal{L}_2$, the output is y itself. Else, the output is the smallest lexicographic Lyndon word having the same PSP.

After the inversion of p to get the table *rank* (lines 1-2), the algorithm proceeds online on that table. It keeps information on the last highest rank met so far in variable r and on the current period of the being-built word y in variable q. Instructions at lines 5-8 implement the bottom up description on the virtual left Lyndon tree of the future output.

```
WordFromPsp(p PSP of a Lyndon word of length n)
 1  for j ← 0 to n − 1 do
 2        rank[p[j]] ← j
 3  (y, r, q) ← (a, rank[0], 1)
 4  for j ← 0 to n − 2 do
 5        if rank[j] ≤ r then
 6              y ← y · y[j − q]
 7        else y ← y · (smallest letter larger than y[j − q])
 8              (r, q) ← (rank[j], j + 1)
 9  y ← y · (smallest letter larger than y[n − 1 − q])
10  return y
```

Applied to the example whose PSP is $(0, 2, 1, 4, 6, 5, 3, 7) = psp(\texttt{abacabade})$ the algorithm produces the Lyndon word `abacabadb`. Indeed, in the initial word, letter `b` is necessarily greater than `a`, letter `c` greater than `b` and letter `d` greater than `c`. But letter `e` is only required to be greater than `a`.

Proposition 9 *Given the PSP table p of a Lyndon word,* WordFromPsp(p) *is the lexicographic smallest Lyndon word $y \in \{\texttt{a}, \texttt{b}, \cdots\}$ for which $psp(y) = p$. The computation is done in linear time.*

Note that when applied to the PSP of a half Zimin word the algorithm recovers the word itself up to an alphabetic translation. Recall that Zimin words Z_i are defined by the relations: Z_0 is the empty word and, for $i > 0$, $Z_i = Z_{i-1} \cdot a_i \cdot Z_{i-1}$, where a_i is a letter not occurring in Z_{i-1}. Then half Zimin words are $Z'_i = Z_{i-1} \cdot a_i$. Using the usual ordered constant alphabet, first half Zimin words are ϵ, `a`, `ab`, `abac`, `abacabad` and `abacabadabacabae`.

Half Zimin words contain the largest alphabet amongst the class of solution words of length n constructed by Algorithm WordFromPsp. They contain $\lfloor \log(n+1) \rfloor + 1$ distinct letters.

6 Lyndon forest

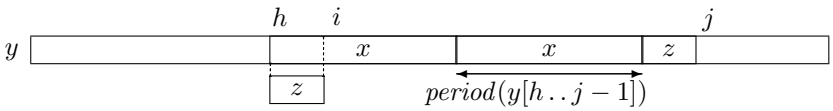
Methods of previous sections that concern Lyndon words easily extend to all (non-empty) words. Trees become forests due to the Lyndon factorisation of words. A forest is reduced to a single tree when the considered word is a Lyndon word.

The Lyndon factorisation of a non-empty word y is a decreasing list of Lyndon factors of the word. It is a list $x_1, x_2, \ldots, x_k$ for which both $x_1 x_2 \cdots x_k = y$ and $x_1 \geq x_2 \geq \cdots \geq x_k$ hold. This factorisation is unique (see [13, Theorem 5.1.5]) and the left Lyndon forest of word y is the list of left Lyndon trees $\mathcal{L}(x_1)$, $\mathcal{L}(x_2)$, …, $\mathcal{L}(x_k)$.

The factorisation and its algorithm by Duval [9] is the guiding thread of previous algorithms. Following the techniques in Section 3 the computation

of Lyndon forest also uses the Lyndon suffix table of the word. Algorithm LyndonSuffixTable deals with words that are not necessarily Lyndon words, and it can be viewed as an extension of Algorithm LyndonSuffixT.

Computing the forest from the table can then be carried out as in Section 3, therefore we only describe the table computation below.




```
LyndonSuffixTable(y non-empty word of length n)
 1  LynS[0] ← 1
 2  (per, h, i, j) ← (1, 0, 0, 1)
 3  while j < n do
 4       if y[j] < y[i] then
 5            h ← j − (i − h)
 6            LynS[h] ← 1
 7            (per, i, j) ← (1, h, h + 1)
 8       elseif y[j] > y[i] then
 9            LynS[j] ← j − h + 1
10            j ← j + 1
11            (per, i) ← (j − h, h)
12       else LynS[j] ← LynS[i]
13            (i, j) ← (h + (i − h + 1 mod per), j + 1)
14  return LynS
```

The update of Algorithm LyndonSuffixT to get Algorithm LyndonSuffixTable essentially lies in instructions on lines 4-7 in Algorithm LyndonSuffixTable. They reset the computation to the suffix $y[h\,..\,n-1]$ of the input after the factorisation of the prefix $y[0\,..\,h-1]$ is definitely achieved. Variable h becomes the starting position of the next Lyndon factor of y.

Proposition 10 *Algorithm* LyndonSuffixTable *computes the Lyndon suffix table of a word of length $n > 0$ in time $O(n)$ in the letter-comparison model.*

Proof. Let us consider the values of expression $h + j$ and show they strictly increase after each iteration of the **while** loop. The claim holds if the condition at line 4 is false, because j is incremented by at least one unit (on line 10 or on line 13) and h remains unchanged. The claim also holds if the condition at line 4 is true, because h is incremented by at least $period(y[h\,..\,j-1])$ while j is decremented by less than the same value.

Thus, since $h+j$ goes from 1 to at most $2n-1$ combined with the fact that instructions at lines 4-13 execute in constant time, the running time is $O(n)$. ■

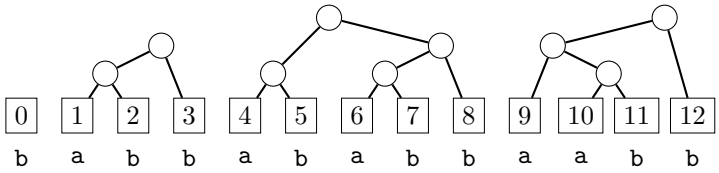
Note that the Lyndon factorisation of a word y can be retrieved from its *LynS* table by sequentially tracing back from $|y|$ starting positions of previous factors.

The list of starting positions of factors, in reverse order, is $i_k = |y| - LynS[|y|-1]$, $i_{k-1} = i_k - LynS[i_{k-1} - 1], \ldots, 0$.

The Lyndon suffix table of $y =$ `babbababbaabb` is as follows:

j	0	1	2	3	4	5	6	7	8	9	10	11	12
$y[j]$	b	a	b	b	a	b	a	b	b	a	a	b	b
$LynS[j]$	1	1	2	3	1	2	1	2	5	1	1	3	4

Starting positions of factors of its Lyndon factorisation are $9 = 13 - LynS[12]$, $4 = 9 - LynS[8]$, $1 = 4 - LynS[3]$, $0 = 1 - LynS[0]$. The figure below depicts the Lyndon forest of this example.



Algorithm LeftLyndonForest is merely adapted from the previous algorithm in order to manage Lyndon tree constructions of each factor of the Lyndon factorisation while computing the latter. The next proposition is a direct consequence of Proposition 10.

Proposition 11 *Algorithm* LeftLyndonForest *computes the Lyndon forest of a word of length $n > 0$ in time $O(n)$ in the letter-comparison model.*

```
LeftLyndonForest(y non-empty word of length n)
 1  (LynS[0], root[0]) ← (1, 0)
 2  (per, h, i, j) ← (1, 0, 0, 1)
 3  while j < n do
 4      root[j] ← j
 5      if y[j] < y[i] then
 6          h ← j − (i − h)
 7          LynS[h] ← 1
 8          (per, i, j) ← (1, h, h + 1)
 9      elseif y[j] > y[i] then
10          LynS[j] ← j − h + 1
11          j ← j + 1
12          (per, i) ← (j − h, h)
13      else LynS[j] ← LynS[i]
14          (i, j) ← (h + (i − h + 1 mod per), j + 1)
15      ▷ Bundle
16      (p, m, k) ← (root[j], 1, j − 1)
17      while m < LynS[j] do
18          q ← new node ≥ n
19          (left[q], right[q]) ← (root[k], p)
20          (p, m) ← (q, m + LynS[k])
21          k ← k − LynS[k]
22  return root[n − 1]
```

7 Conclusions

In this paper, Algorithm LyndonSuffixTable computes the Lyndon suffix table of a word. The table is an essential part of algorithm LeftLyndonTree that constructs the left Lyndon tree of a Lyndon word in linear time.

We further investigated the prefix standard permutation of a Lyndon word, initially introduced by Dolce et al. [8], and its relation to the left Lyndon tree. This study resulted in a linear-time algorithm for sorting the prefixes of a Lyndon word according to infinite ordering. In addition, we showed how to recover a word from a given permutation assumed to be a prefix standard permutation.

To achieve the results, we exhibited a strong connection between the prefix ranks and the left Lyndon tree. This connection dictates that the order in which the internal nodes of the left Lyndon tree are created and processed coincides with that of the prefix ranks according to infinite ordering and corresponds to the left-to-right postorder traversal of the tree.

We finally endeavoured to design a linear-time algorithm, LeftLyndonForest, that computes the Lyndon forest of an ordinary word.

Many interesting questions remain, among them are: Is there a connection between runs and the internal nodes of the left Lyndon forest? Is there a tight relation between the left Lyndon trees and the right Lyndon trees?

References

[1] G. Badkobeh and M. Crochemore. Left Lyndon tree construction. In J. Holub and J. Zdárek, editors, *Prague Stringology Conference 2020, Prague, Czech Republic, August 31-September 2, 2020*, pages 84–95. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2020.

[2] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. The "runs" theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017.

[3] P. Bille, J. Ellert, J. Fischer, I. L. Gørtz, F. Kurpicz, J. I. Munro, and E. Rotenberg. Space efficient construction of Lyndon arrays in linear time. In A. Czumaj, A. Dawar, and E. Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 14:1–14:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[4] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen. The maximal number of cubic runs in a word. *J. Comput. Syst. Sci.*, 78(6):1828–1836, 2012.

[5] M. Crochemore, T. Lecroq, and W. Rytter. *One Twenty Five Problems in Text Algorithms*. Cambridge University Press, 2021. In press.

[6] M. Crochemore and L. M. S. Russo. Cartesian and Lyndon trees. *Theoretical Computer Science*, 806:1–9, February 2020.

[7] F. Dolce, A. Restivo, and C. Reutenauer. On generalized Lyndon words. *Theor. Comput. Sci.*, 777:232–242, 2019.

[8] F. Dolce, A. Restivo, and C. Reutenauer. Some variations on Lyndon words. *CoRR*, abs/1904.00954, 2019.

[9] J. Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.

[10] F. Franek and M. Liut. Algorithms to compute the Lyndon array revisited. In J. Holub and J. Zdárek, editors, *Prague Stringology Conference 2019, Prague, Czech Republic, August 26-28, 2019*, pages 16–28. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2019.

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

[12] R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604. IEEE Computer Society, 1999.

[13] M. Lothaire. *Combinatorics on Words*. Addison-Wesley, 1983. Reprinted in 1997.

[14] R. C. Lyndon. On Burnside problem I. *Trans. Amer. Math. Soc.*, 77:202–215, 1954.

[15] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In D. S. Johnson, editor, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*, pages 319–327. SIAM, 1990.

[16] Y. Nakashima, T. Takagi, S. Inenaga, H. Bannai, and M. Takeda. On the size of the smallest alphabet for Lyndon trees. *Theor. Comput. Sci.*, 792:131–143, 2019.

[17] V. A. Ufnarovskij. Combinatorial and asymptotic methods in algebra. In A. Kostrikin and I. Shafarevich, editors, *Algebra VI: Combinatorial and Asymptotic Methods of Algebra. Non-Associative Structures*, volume 57 of *Encyclopaedia of Mathematical Sciences*, pages 1–196. Springer, Berlin, 2011.

[18] G. Viennot. *Algèbres de Lie libres et monoïdes libres*, volume 691 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1978.