Algorithms to Compute the Lyndon Array*

Frantisek Franek[1], A. S. M. Sohidull Islam[2], M. Sohel Rahman[3], and William F. Smyth[1,3,4]

[1] Algorithms Research Group
Department of Computing & Software
McMaster University, Hamilton, Canada
{franek/smyth}@mcmaster.ca
[2] School of Computational Science & Engineering
McMaster University, Hamilton, Canada
sohansayed@gmail.com
[3] Department of Computer Science & Engineering
Bangladesh University of Engineering & Technology
msrahman@cse.buet.ac.bd
[4] School of Engineering & Information Technology
Murdoch University, Perth, Australia

Abstract. In the Lyndon array $\boldsymbol{\lambda} = \boldsymbol{\lambda}_{\boldsymbol{x}}[1..n]$ of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, $\boldsymbol{\lambda}[i]$ is the length of the longest Lyndon word starting at position i of $\boldsymbol{x}$. The computation of $\boldsymbol{\lambda}$ has recently become of great interest, since it was shown (Bannai *et al.*, **The "Runs" Theorem**) that the runs in $\boldsymbol{x}$ are computable in linear time from $\boldsymbol{\lambda}_{\boldsymbol{x}}$. Here we describe two algorithms for computing $\boldsymbol{\lambda}_{\boldsymbol{x}}$ based on previous results known in different context, but for which no explicit exposition in this context had been given. These two algorithms execute in $\mathcal{O}(n^2)$ time in the worst case. The third algorithm presented that executes in $\Theta(n)$ time had been suggested and discussed previously, and we provide a more substantial discussion and prove of correctness for one of its steps. This algorithm achieves its linearity at the expense of prior computation of both the suffix array and the inverse suffix array of $\boldsymbol{x}$. We then go on to sketch a new algorithm and its two variants that avoids prior computation of global data structures and indicate that in worst-case these algorithms perform in $\mathcal{O}(n \log n)$ time.
Keywords: string, Lyndon word, Lyndon array, Lyndon factorization

1 Introduction

If $\boldsymbol{x} = \boldsymbol{uv}$ for some $\boldsymbol{u}$ and nonempty $\boldsymbol{v}$, then $\boldsymbol{vu}$ is said to be the $|\boldsymbol{u}|^{\text{th}}$ ***rotation*** of $\boldsymbol{x}$, written $\boldsymbol{vu} = R_{|\boldsymbol{u}|}(x)$. If there exists a string $\boldsymbol{u}$ and an integer $e > 1$ such that $\boldsymbol{x} = \boldsymbol{u}^e$, then $\boldsymbol{x}$ is said to be a ***repetition***; otherwise $\boldsymbol{x}$ is ***primitive***. A primitive string $\boldsymbol{x}$ that is lexicographically strictly least among all its rotations $R_k(\boldsymbol{x}), k = 0, 1, \ldots, |\boldsymbol{x}|-1$, is said to be a ***Lyndon word***.

The ***Lyndon array*** $\boldsymbol{\lambda} = \boldsymbol{\lambda}_{\boldsymbol{x}}[1..n]$ of a given nonempty string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ gives at each position i the length of the longest Lyndon word starting at i. Note that equivalently we could store in the ith position of the Lyndon array the end position of the longest Lyndon word starting in i. We will use the notation $\mathcal{L}[i]$ to indicate the end position for the longest Lyndon word starting at i.

$$\begin{array}{rl} & \begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array} \\ \boldsymbol{x} = & \begin{array}{cccccccccc} a & b & a & a & b & a & b & a & a & b \end{array} \\ \boldsymbol{\lambda} = & \begin{array}{cccccccccc} 2 & 1 & 5 & 2 & 1 & 2 & 1 & 3 & 2 & 1 \end{array} \\ \mathcal{L} = & \begin{array}{cccccccccc} 2 & 2 & 7 & 5 & 5 & 7 & 7 & 10 & 10 & 10 \end{array} \end{array} \tag{1}$$

* This work was supported in part by the Natural Sciences & Engineering Research Council of Canada. The authors wish to thank Maxime Crochemore and Hideo Bannai for helpful discussions.

Frantisek Franek, A. S. M. Sohidull Islam, M. Sohel Rahman, William F. Smyth: Algorithms to Compute the Lyndon Array, pp. 172–184.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8

Since being Lyndon really depends on the order of the underlying alphabet of the string, the Lyndon array of a string will change when we change the order of the alphabet. The Lyndon array has recently become of interest since Bannai *et al.* [2] showed that the two Lyndon arrays, one with respect to a given order of the alphabet and the other with respect to the inverse of that order, can be used to compute all the maximal periodicities ("runs") in a string in linear time.

In this paper we describe four algorithms to compute $\boldsymbol{\lambda_x}$. Section 2 makes various observations that apply generally to the Lyndon array and its computation. In Section 3 we describe two algorithms that are based on previous results known in a different context, and we present them here explicitly in the context of computing Lyndon arrays. These two algorithms perform in $\mathcal{O}(n^2)$ time in the worst case, where n is the length of the input string. Despite the high worst case complexity, in practice these algorithms perform very well as they are simple and straightforward to implement and do not require any complicated data structures; they could be characterized almost as in-place. The third algorithm discussed in this section had been described previously and we provide a more substantial discussion and prove correctness of one of its steps that we could not find anywhere in the literature. This algorithm is simple and worst-case linear-time, but requires suffix array construction and so is a little slower. Section 4 describes two variants of an algorithm we designed that uses only elementary data structures (no suffix arrays). One variant is $\mathcal{O}(n^2)$ in the worst case, the other indicates $\mathcal{O}(n \log n)$ time, but with no clear advantage in processing time. Section 5 describes the results of preliminary experiments on the algorithms; Section 6 outlines future work.

2 Preliminaries

Here we make various observations that apply to the algorithms described below.

Observation 1 *Let $\boldsymbol{x} = \boldsymbol{w_1 w_2 \cdots w_k}$ be the Lyndon decompostion [5,9] of $\boldsymbol{x}$, with Lyndon words $\boldsymbol{w_1} \geq \boldsymbol{w_2} \geq \cdots \geq \boldsymbol{w_k}$. Then every Lyndon word $\boldsymbol{x}[i..\mathcal{L}[i]]$ of length $\boldsymbol{\lambda}[i]$ is a substring of some $\boldsymbol{w_h}$, $h \in 1..k$.*

Proof. For some $h \in 1..k-1$, consider $\boldsymbol{w_h}$ with a nonempty proper suffix $\boldsymbol{v_h}$, and for some $t \in 1..k-h$, consider $\boldsymbol{w_{h+t}}$ with a nonempty prefix $\boldsymbol{u_{h+t}}$. Since $\boldsymbol{w_h}$ is a Lyndon word, $\boldsymbol{w_h} < \boldsymbol{v_h}$, and by lexorder, $\boldsymbol{u_{h+t}} \leq \boldsymbol{w_{h+t}}$. Thus $\boldsymbol{v_h} > \boldsymbol{w_h} \geq \boldsymbol{w_{h+t}} \geq \boldsymbol{u_{h+t}}$, and so $\boldsymbol{v_h w_{h+1} \cdots w_{h+t-1} u_{h+t}}$ cannot be a Lyndon word for any choice of h or t. □

Therefore to compute $\mathcal{L}_{\boldsymbol{x}}$ it suffices to consider separately each distinct element $\boldsymbol{w_h}$ in the Lyndon decomposition of $\boldsymbol{x}$. Hence, without loss of generality we suppose that $\boldsymbol{x}$ is a Lyndon word and write it in the form $\boldsymbol{x_1 x_2 \cdots x_m}$, where for each $r \in 1..m$, $|\boldsymbol{x_r}| = \ell_r$ and

$$\boldsymbol{x_r}[1] \leq \boldsymbol{x_r}[2] \leq \cdots \leq \boldsymbol{x_r}[\ell_r], \tag{2}$$

while for $1 \leq r < m$,

$$\boldsymbol{x_r}[\ell_r] > \boldsymbol{x_{r+1}}[1]. \tag{3}$$

We call $\boldsymbol{x_r}$ a ***range*** in $\boldsymbol{x}$ and the boundary between $\boldsymbol{x_r}$ and $\boldsymbol{x_{r+1}}$ a ***drop***. We identify a position j in range $\boldsymbol{x_r}$, $1 \leq j \leq \ell_r$, with its equivalent position i in $\boldsymbol{x}$ by writing $i = S_{r,j} = \sum_{r'=1}^{r-1} \ell_{r'} + j$.

Observation 2 *Let $i = S_{r,j}$ be a position in $\boldsymbol{x}$ that corresponds to position j in range $\boldsymbol{x_r}$.*

(*a*) *If $\boldsymbol{x_r}[j] = \boldsymbol{x_r}[\ell_r]$, then $\mathcal{L}[i] = i$.*
(*b*) *Otherwise, $\mathcal{L}[i] = i'$, where i' is the final position in some range $\boldsymbol{x_{r'}}$, $r' \geq r$; that is, $i' = \sum_{s=1}^{r'} \ell_s$.*

Proof. (a) is an immediate consequence of (2) and (3). To prove (b), suppose that $\boldsymbol{x}[i..\mathcal{L}[i]]$ is a maximum-length Lyndon word, where $\mathcal{L}[i]$ falls within range r' but $\mathcal{L}[i] < i'$. Since by (2) $\boldsymbol{x}[\mathcal{L}[i]] \leq \boldsymbol{x}[\mathcal{L}[i]+1]$, there are two consecutive Lyndon words $\boldsymbol{x}[i..\mathcal{L}[i]], \boldsymbol{x}[\mathcal{L}[i]+1]$ that by the Lyndon decomposition theorem [5] can be merged into a single Lyndon word $\boldsymbol{x}[i..\mathcal{L}[i]+1]$. Thus $\boldsymbol{x}[i..\mathcal{L}[i]]$ is not maximum-length, a contradiction. □

We see then that if $\boldsymbol{x_r}[j] < \boldsymbol{x_r}[\ell_r]$, then $\boldsymbol{x_r}[j..\ell_r]$ is a (not necessarily maximum-length) Lyndon word, and for $i = S_{r,j}$, $\mathcal{L}[i] \geq S_{r,\ell_r}$:

$$\begin{array}{rl} & \begin{array}{cccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \end{array} \\ \boldsymbol{x} = & a\ \ a\ \ a\ b \mid a\ a\ b \mid a\ b \mid\ a\ \ a\ \ b\ \ b \\ \mathcal{L} = & 13\ 13\ 4\ \ 4\ \ 9\ 7\ \ 7\ \ 9\ \ 9\ \ 13\ 13\ 12\ 13 \end{array} \tag{4}$$

More generally, the integer interval $\langle i, \mathcal{L}[i] \rangle = i..\mathcal{L}[i]$ satisfy a "Monge" property that is exploited by Algorithm NSV* (Section 4):

Observation 3 *Suppose positions i, j in $\boldsymbol{x}[1..n]$ satisfy $1 \leq i < j \leq n$. Then either $\mathcal{L}[i] \leq j$ or $\mathcal{L}[i] \geq \mathcal{L}[j]$: the intervals $\langle i, \mathcal{L}[i] \rangle$ and $\langle j, \mathcal{L}[j] \rangle$ are not overlapping.*

Proof. Suppose two such intervals do overlap. Then the maximum-length Lyndon words $\boldsymbol{w_1} = \boldsymbol{x}[i..\mathcal{L}[i]]$ and $\boldsymbol{w_2} = \boldsymbol{x}[j..\mathcal{L}[j]]$ have a nonempty overlap, so that we can write $\boldsymbol{w_1} = \boldsymbol{uv}$, $\boldsymbol{w_2} = \boldsymbol{vv'}$ for some nonempty $\boldsymbol{v}$. But then, by well-known properties of Lyndon words, $\boldsymbol{w_1} < \boldsymbol{v} < \boldsymbol{w_2} < \boldsymbol{v'}$, implying that $\boldsymbol{w_1 v'}$ is a Lyndon word, contradicting the assumption that $\boldsymbol{w_1}$ is of a maximal length. □

Expressing a string in terms of its ranges has the same useful lexorder property that writing it in terms of its letters does:

Observation 4 *Suppose strings $\boldsymbol{x}$ and $\boldsymbol{y}$ are expressed in terms of their ranges: $\boldsymbol{x} = \boldsymbol{x_1 x_2 \cdots x_m}$, $\boldsymbol{y} = \boldsymbol{y_1 y_2 \cdots y_n}$. Suppose further that for some least integer $r \in 1..\min(m, n)$, $\boldsymbol{x_r} \neq \boldsymbol{y_r}$. Then $\boldsymbol{x} < \boldsymbol{y}$ (respectively, $\boldsymbol{x} > \boldsymbol{y}$) according as $\boldsymbol{x_r} < \boldsymbol{y_r}$ (respectively, $\boldsymbol{x_r} > \boldsymbol{y_r}$).*

Proof. If $\boldsymbol{x_r} < \boldsymbol{y_r}$, then either

(*a*) $\boldsymbol{x_r}$ is a nonempty proper prefix of $\boldsymbol{y_r}$; or
(*b*) there is some least position j such that $\boldsymbol{x_r}[j] < \boldsymbol{y_r}[j]$.

In case (a), if $r = m$, then $\boldsymbol{x}$ is actually a prefix of $\boldsymbol{y}$, so that $\boldsymbol{x} < \boldsymbol{y}$, while if $r < m$, then by (3), $\boldsymbol{x_{r+1}}[1] < \boldsymbol{y_r}[|\boldsymbol{x_r}|+1]$, and again $\boldsymbol{x} < \boldsymbol{y}$. In case (b) the result is immediate. The proof for $\boldsymbol{x_r} > \boldsymbol{y_r}$ is similar. □

3 Basic Algorithms

Here we outline three algorithms for which no clear exposition in the context of Lyndon arrays is available in the literature. We remark that the Lyndon array computation is equivalent to "Lyndon bracketing", for which an $\mathcal{O}(n^2)$ algorithm was described in [17].

3.1 Folklore — Iterated MaxLyn

This algorithm, see Figure 1, is based on Duval's linear time algorithm for Lyndon factorization, [9] – it is the application of its first step which we refer to as **MaxLyn** since it returns the size of the longest Lyndon word starting at that position. This process is iterated for all positions in the input string and this thus gives immediately $O(|x|^2)$ worst case complexity for an input string x. Since Duval's algorithm is in-place, this algorithm is simple and almost in-place, except the space for the Lyndon array. Below, we sketch the reasons the algorithm provides the correct answer.

For a string $\boldsymbol{x}$ of length n, recall that the ***prefix table*** $\pi[1..n]$ is an integer array in which for every $i \in 1\,..\,n$, $\pi[i]$ is the length of the longest substring beginning at position i of $\boldsymbol{x}$ that matches a prefix of $\boldsymbol{x}$. Given a nonempty string $\boldsymbol{x}$ on alphabet Σ, let us define $\boldsymbol{x'} = \boldsymbol{x}\$$, where the sentinel $\$ < \mu$ for every letter $\mu \in \Sigma$.

Observation 5 *$\boldsymbol{x}$ is a Lyndon word if and only if for every $i \in 2\,..\,n$, $\boldsymbol{x'}[1+k] < \boldsymbol{x'}[i+k]$, where $k = \pi[i]$.*

This result forms the basis of the algorithm given in Figure 1 that computes the length $\max \in 1\,..\,n-j+1$ of the longest Lyndon factor at a given position j in $\boldsymbol{x}[1..n]$. Its efficiency is a consequence of the instruction $i \leftarrow i+k+1$ that skips over positions in the range $i+1\,..\,i+k-1$, effectively assuming that for every position i^* in that range, $i^* + \pi[i^*] \leq i+k$. Lemma 11, given in Appendix 1, justifies this assumption. Simply repeating MaxLyn at every position j of $\boldsymbol{x}$ gives a simple, fast $\mathcal{O}(n^2)$ time and $\mathcal{O}(1)$ additional space algorithm to compute $\boldsymbol{\lambda_x}$.

```
procedure MaxLyn(x[1..n], j, Σ, ≺) : integer
i ← j + 1; max ← 1
while i ≤ n do
  k ← 0
  while x'[j + k] = x'[i + k] do
    k ← k + 1
  if x'[j + k] ≺ x'[i + k] then
    i ← i + k + 1; max ← i − 1
  else
    return max
```

Figure 1. Algorithm MaxLyn

Recent work on the prefix table [4,6] has confirmed its importance as a data structure for string algorithms. In this context it is interesting to find that Lyndon words $\boldsymbol{x}$ can be characterized in terms of $\pi_{\boldsymbol{x}}$:

Observation 6 *Suppose $\boldsymbol{x} = \boldsymbol{x}[1\,..\,n]$ is a string on alphabet Σ such that $\boldsymbol{x}[1]$ is the least letter in $\boldsymbol{x}$. Then $\boldsymbol{x}$ is a Lyndon word over Σ if and only if for every $i \in 2\,..\,n$,*

(a) $i + \pi_{\boldsymbol{x}}[i] < n+1$; and

(*b*) *for every* $j \in i+1\,..\,i+\pi_{\boldsymbol{x}}[i]-1$, $j+\pi_{\boldsymbol{x}}[j] \leq i+\pi_{\boldsymbol{x}}[i]$.

In Appendix 1, the reader can find an additional result that justifies the strategy employed by **MaxLyn** (Figure 1).

3.2 Recursive Duval Factorization: Algorithm RDuval

Rather than independently computing the maximum-length Lyndon factor at each position i, as MaxLyn does, Algorithm RDuval recursively computes the Lyndon decomposition, [9], into maximum factors, at each step taking advantage of the fact that $\mathcal{L}[i]$ is known for the first position i in each factor, then recomputing with the first letters removed. This again gives immediate worst case complexity of $O(n^2)$. We consider it only because it allows for a more refined discussion of the complexity in special cases for strings over binary alphabets giving an average case complexity of $\mathcal{O}(n \log n)$, see below.

By Observation 1, whenever $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is a Lyndon word, we know that $\mathcal{L}[1] = n$. Thus computing the Lyndon decomposition $\boldsymbol{x} = \boldsymbol{w_1 w_2} \cdots \boldsymbol{w_k}$, $\boldsymbol{w_1} \geq \boldsymbol{w_2} \geq \cdots \geq \boldsymbol{w_k}$, allows us to assign $\boldsymbol{\lambda}[i_j] = |\boldsymbol{w_j}|$, where i_j is the first position of $\boldsymbol{w_j}$, $j = 1, 2, \ldots, k$.

Algorithm RDuval applies this strategy recursively, by assigning $\boldsymbol{\lambda}[i_j] \leftarrow |\boldsymbol{w_j}|$, then removing the first letter i_j from each $\boldsymbol{w_j}$ to form $\boldsymbol{w'_j}$, to which the Lyndon decomposition is applied in the next recursive step. This process continues until each Lyndon word is reduced to a single letter.

The asymptotic time required for RDuval is bounded above by n times the maximum depth of the recursion, thus $O(n^2)$ in the worst case — consider, for example, the string $\boldsymbol{x} = a^{n-1}b$. However, to estimate expected behaviour, we can make use of a result of Bassino *et al.* [3]. Given a Lyndon word $\boldsymbol{w}$, they call $\boldsymbol{w} = \boldsymbol{uv}$ the ***standard factorization*** of $\boldsymbol{w}$ if $\boldsymbol{u}$ and $\boldsymbol{v}$ are both Lyndon words and $\boldsymbol{v}$ is of maximum size. They then show that if $\boldsymbol{w}$ is a binary string ($\Sigma = \{a, b\}$), the average length of $\boldsymbol{v}$ is asymptotically $3|\boldsymbol{w}|/4$. Thus each recursive application of RDuval yields a left Lyndon factor of expected length $|\boldsymbol{w}|/4$ and a remainder of length $3|\boldsymbol{w}|/4$ to be factored further. It follows that the expected number of recursive calls of RDuval is $\mathcal{O}(\log_{4/3} n)$. Hence

Lemma 7 *On binary strings RDuval executes in* $O(n \log_{4/3} n)$ *time on average.*

Example 8 *For*

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	*a*	*a*	*b*	*a*	*a*	*b*	*b*	*a*	*b*	*b*	*a*	*b*
$\boldsymbol{\lambda} =$	12	2	1	9	3	1	1	3	1	1	2	1

the factors considered are first 1–12, then

- *2–3 and 4–12 in the first level of recursion;*
- *3, 5–7, 8–10 and 11–12 in the second level;*
- $6, 7, 9, 10, 12$ *in the third level.*

Positions are assigned as follows: $\boldsymbol{\lambda}[1] \leftarrow 12; \boldsymbol{\lambda}[2] \leftarrow 2, \boldsymbol{\lambda}[4] \leftarrow 9; \boldsymbol{\lambda}[3] \leftarrow 1, \boldsymbol{\lambda}[5] \leftarrow 3, \boldsymbol{\lambda}[8] \leftarrow 3, \boldsymbol{\lambda}[11] \leftarrow 2; \boldsymbol{\lambda}[6] \leftarrow 1, \boldsymbol{\lambda}[7] \leftarrow 1, \boldsymbol{\lambda}[9] \leftarrow 1, \boldsymbol{\lambda}[10] \leftarrow 1, \boldsymbol{\lambda}[12] \leftarrow 1.$

3.3 NSV Applied to the Inverse Suffix Array

The idea of the "next smaller value" (NSV) array for a given array $\boldsymbol{x}$ had been proposed in various forms and under various names [1,10,11,15].

Definition 9 (Next Smaller Value) *Given an array $\boldsymbol{x}[1..n]$ of ordered values, $NSV = NSV_{\boldsymbol{x}}[1..n]$ is the **next smaller value array** of $\boldsymbol{x}$ if and only if for every $i \in 1..n$, $NSV[i] = j$, where*

(a) *for every $h \in 1..j-1$, $\boldsymbol{x}[i] \leq \boldsymbol{x}[i+h]$; and*
(b) *either $i+j = n+1$ or $\boldsymbol{x}[i] > \boldsymbol{x}[i+j]$.*

Example 10

$$\begin{array}{rl} & \begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array} \\ \boldsymbol{x} = & 3\,8\,7\,10\,2\,1\,4\,9\,6\,5 \\ NSV_{\boldsymbol{x}} = & 4\,1\,2\;\;1\;\;1\,5\,4\,1\,1\;\;1 \end{array}$$

As shown in various contexts in [11], $\mathrm{NSV}_{\boldsymbol{x}}$ can be computed in $\Theta(n)$ time using a stack. Our main observation here, also mentioned in [12], is that $\boldsymbol{\lambda}_{\boldsymbol{x}}$ can be computed merely by applying NSV to the inverse suffix array $\mathrm{ISA}_{\boldsymbol{x}}$. Proof of this claim can be found in Appendix 2; here we present the very simple $\Theta(n)$-time, $\Theta(n)$-space algorithm for this calculation:

procedure NSVISA($\boldsymbol{x}[1\,..\,n]$) : $\boldsymbol{\lambda}_{\boldsymbol{x}}[1\,..\,n]$
Compute $SA_{\boldsymbol{x}}$ (see [14,16])
Compute $\mathrm{ISA}_{\boldsymbol{x}}$ from $\mathrm{SA}_{\boldsymbol{x}}$ in place (see [16])
$\boldsymbol{\lambda}_{\boldsymbol{x}} \leftarrow \mathrm{NSV}(\mathrm{ISA}_{\boldsymbol{x}})$ (in place)

Figure 2. Apply NSV to $\mathrm{ISA}_{\boldsymbol{x}}$

4 Elementary Computation of $\boldsymbol{\lambda}_{\boldsymbol{x}}$ Using Ranges

In this section we describe an approach to the computation of $\boldsymbol{\lambda}_{\boldsymbol{x}}$ that applies a variant of the NSV idea to the ranges of $\boldsymbol{x}$. Figure 3 gives pseudocode for Algorithm NSV* that uses the NSV stack ACTIVE to compute $\boldsymbol{\lambda}$. The processing identifies ranges in a single left-to-right scan of $\boldsymbol{x}$, making use of two range comparison routines, COMP and MATCH. COMP compares adjacent individual ranges $\boldsymbol{x_r}$ and $\boldsymbol{x_{r+1}}$, returning $\delta_1 = -1, 0, +1$ according as $\boldsymbol{x_r} < \boldsymbol{x_{r+1}}$, $\boldsymbol{x_r} = \boldsymbol{x_{r+1}}$, $\boldsymbol{x_r} > \boldsymbol{x_{r+1}}$. MATCH similarly returns δ_2 for adjacent *sequences* of ranges; that is,

$$\begin{aligned} \boldsymbol{X_r} &= \boldsymbol{x_r x_{r+1}} \cdots \boldsymbol{x_{r+s}}, \text{ for some } s \geq 1; \\ \boldsymbol{X_{r+s+1}} &= \boldsymbol{x_{r+s+1} x_{r+s+2}} \cdots \boldsymbol{x_{r+s+t}}, \text{ for some } t \geq 1. \end{aligned}$$

Algorithm NSV* is based on the idea encapsulated in Lemma 15 of Appendix 2, the main basis of the correctness of Algorithm NSVISA (see Figure 2). We process $\boldsymbol{x}$ from left to right, using a stack ACTIVE initialized with index 1. At each iteration, the top of the stack (say, j) is compared with the current index (say, i). In particular, we need to compare $\boldsymbol{s_x}(i)$ with $\boldsymbol{s_x}(j)$, where $\boldsymbol{s_x}(i) \equiv \boldsymbol{x}[i..n]$. As long as $\boldsymbol{s_x}(i) \succeq \boldsymbol{s_x}(j)$, NSV* pushes the current index and continues to the next. When $\boldsymbol{s_x}(i) \prec \boldsymbol{s_x}(j)$, it pops the stack and puts appropriate values in the corresponding indices of $\boldsymbol{\lambda}_{\boldsymbol{x}}$.

```
procedure NSV* (x, λ)
nextequal ← 0^n; period ← 0^n
push(ACTIVE) ← 1
▷ x[n+1] = $, a letter smaller than any in Σ.
for i ← 2 to n+1 do
  prev ← 0; j ← peek(ACTIVE)
▷ COMP compares suffixes specified by i, j of two ranges.
  δ1 ← COMP(x[j], x[i]); δ2 ← 1
    while (δ1 ≥ 0 and δ2 > 0) do
      if δ1 = 0 then δ2 ← MATCH(x[j], x[i])
      if δ2 > 0 then
        if prev = 0 or nextequal[j] ≠ prev then λ[j] ← i−j
        else
          λ[j] ← offset ← prev−j
          if period[prev] = 0 then
            if λ[prev] > offset then
              λ[j] ← λ[j]+λ[prev]
          else
            if nextequal[j] = prev and offset ≠ λ[prev] then
              λ[j] ← λ[j]+period[prev]
          if λ[prev] = offset then
        ▷ Current position is a part of periodic substring
            if period[prev] = 0 then
              period[j] ← period[prev] + 2 × offset
            else
              period[j] ← period[prev]+offset
        pop(ACTIVE)
        prev ← j; j ← peek(ACTIVE)
        ▷ Empty stack implies termination.
        if j = 0 then EXIT
        δ1 ← COMP(x[j], x[i])
▷ Finished processing i — it goes to stack.
  if δ2 = 0 then nextequal[j] ← i
  push(ACTIVE) ← i
```

Figure 3. Computing $\boldsymbol{\lambda}_x$ using modified NSV

As noted above, especially Observations 1–3, ranges are employed to expedite these suffix comparisons.

Two auxiliary arrays, **nextequal** and **period**, are required to handle situations in which MATCH finds that a suffix of a previous range at position j equals the current range at position i. Thus, when $\delta_2 = 0$, the algorithm assigns nextequal$[j] \leftarrow i$ before i is pushed onto ACTIVE. Then when a later MATCH yields $\delta_2 = 0$, the value of **period** — that is, the extent of the following periodicity — may need to be set or adjusted, as shown in the following example:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\boldsymbol{x}$ =	*a*	*a*	*a*	*b*	*a*	*a*	*b*	*a*	*a*	*b*	*a*	*a*	*b*	*a*	*b*
nextequal =	0	5	0	0	8	0	0	11	0	0	0	14	0	0	0
period =	0	12	0	0	9	0	0	6	0	0	0	4	0	0	0

A straightforward implementation of COMP and MATCH could require a number of letter comparisons equal to the length of the shorter of the two sequences of ranges being matched. However, by performing $\Theta(n)$-time preprocessing, we can compare two ranges in $\mathcal{O}(\sigma)$ time, where $\sigma = |\Sigma|$ is the alphabet size. Given $\Sigma = \{\mu_1, \mu_2, \ldots, \mu_\sigma\}$,

we define Parikh vectors $P_r[1..\sigma]$, where $P_r[j]$ is the number of occurrences of μ_j in range $\boldsymbol{x_r}$. Since ranges are monotone nondecreasing in the letters of the alphabet, it is easy to compute all the $P_r, r = 1, 2, \ldots, m$, in linear time in a single scan of $\boldsymbol{x}$. Similarly, during the processing of each range $\boldsymbol{x_r}$, any value $P_{r,j}$, the Parikh vector of the suffix $\boldsymbol{x_r}[j..\ell_r]$, can be computed in constant time for each position considered. Thus we can determine the lexicographical order of any two ranges (or part ranges) $\boldsymbol{x_r}$ and $\boldsymbol{x_{r'}}$ in $\mathcal{O}(\sigma)$ time rather than time $\mathcal{O}(\max(\ell_r, \ell_{r'}))$. The variant of NSV* that uses Parikh vectors is called PNSV*; otherwise NPNSV* for Not Parikh.

In Appendix 3 we describe briefly another approach to this suffix comparison problem, which we believe achieves run time $\mathcal{O}(n \log n)$ by maintaining a simple data structure requiring $\mathcal{O}(n \log n)$ space.

Now consider the worst case behaviour of Algorithm NSV*. Given the initial string $\boldsymbol{x_0} = a^h b a^h c_0,\ h \geq 1, c_0 > b > a$, let $\boldsymbol{x_k^{(h)}} = \boldsymbol{x_k} = \boldsymbol{x_{k-1}} \boldsymbol{x^*_{k-1}},\ k = 1, 2, \ldots$, with $\boldsymbol{x^*_{k-1}}$ identical to $\boldsymbol{x_{k-1}}$ except in the last position, where the letter $c_k > c_{k-1}$ replaces c_{k-1}. Then $\boldsymbol{x_k}$ has length $n = (h+1)m$, where $m = 2^{k+1}$ is the number of ranges in $\boldsymbol{x_k}$. We believe and are working towards a proof that $\boldsymbol{x_k}$ is a worst-case input for Algorithm NSV*, which requires $\mathcal{O}(n \log n)$ range matches in such cases. Since PNSV* compares two ranges in $\mathcal{O}(\sigma)$ time, it therefore would require $\mathcal{O}(\sigma n \log n)$ time in the worst case, thus $\mathcal{O}(n \log n)$ for constant σ.

5 Preliminary Experimental Results

We have done some preliminary tests on the implementations of the two variants of NSV*, with and without employing Parikh vectors. The equipment used was an Intel(R) Core i3 at 1.8GHz and 4GB main memory under a 64-bit Windows 7 operating system. For each length $10000, 20000, \ldots, 100000$ we generated 500 random strings for alphabets of sizes $\sigma = 2, 4$ and 8. The results indicate, that at least for random strings, the processing time seems linear. The processing time for "with Parikh vectors" is greater because of the initial pre-processing. The data and the corresponding graphs are in Figures 4..11 below.

Processing time versus length of the string for the program without Parikh vectors

length of the input strings	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
alphabet size 2	0.001122	0.002265	0.003335	0.004444	0.005552	0.006682	0.007813	0.008936	0.010117	0.011167
alphabet size 4	0.001103	0.002233	0.003345	0.004464	0.0056	0.006677	0.007792	0.008903	0.010036	0.011157
alphabet size 8	0.001067	0.002112	0.003181	0.004255	0.005337	0.006372	0.007418	0.008522	0.009646	0.010653

Figure 4. Processing times in seconds for the implementation without Parikh vectors

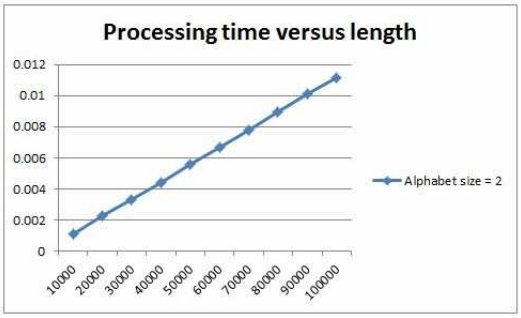

Figure 5. Processing times for random strings over the binary alphabet; without Parikh vectors

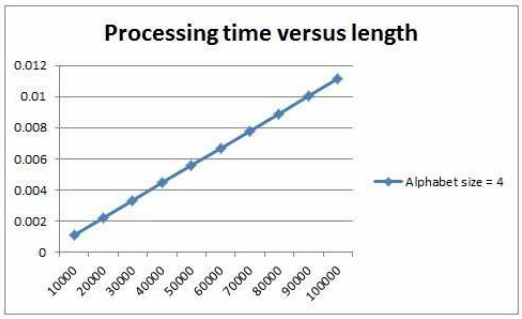

Figure 6. Processing times for random strings over the alphabet of size 4; without Parikh vectors

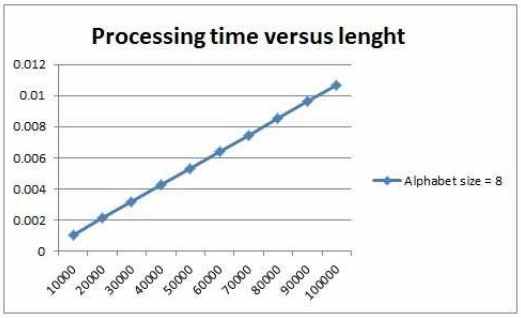

Figure 7. Processing times for random strings over the alphabet of size 8; without Parikh vectors

Processing time versus length of the string for the program with Parikh vectors

length of the input strings	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
alphabet size 2	0.001368	0.002724	0.004072	0.005413	0.006782	0.008304	0.009465	0.010868	0.012201	0.013669
alphabet size 4	0.001898	0.003782	0.005694	0.007516	0.009498	0.011421	0.01319	0.015226	0.017124	0.019068
alphabet size 8	0.002385	0.004721	0.007053	0.009409	0.011874	0.014219	0.016546	0.018967	0.021337	0.024008

Figure 8. Processing times in seconds for the implementation with Parikh vectors

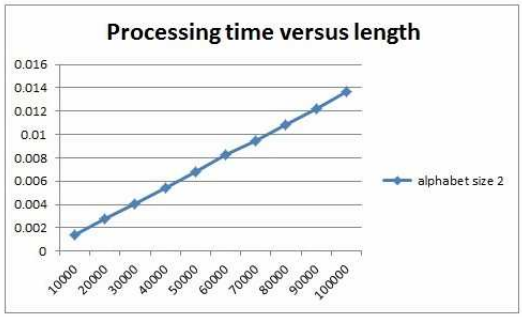

Figure 9. Processing times for random strings over the binary alphabet; with Parikh vectors

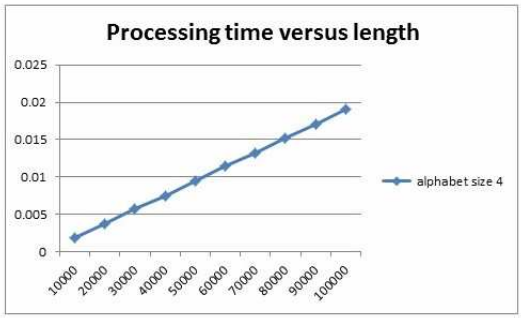

Figure 10. Processing times for random strings over the alphabet of size 4; with Parikh vectors

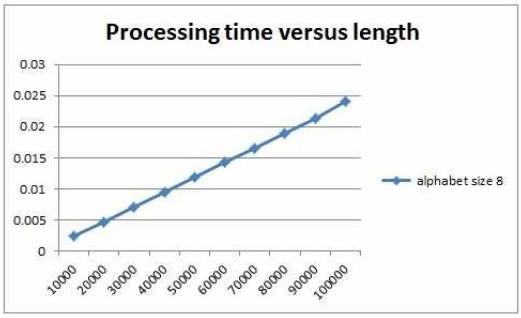

Figure 11. Processing times for random strings over the alphabet of size 8; with Parikh vectors

6 Future Work

There is reason to believe [13] that the Lyndon array computation is less hard than suffix array construction. Thus the authors conjecture that there is a linear-time elementary algorithm (no suffix arrays) to compute the Lyndon array.

References

1. S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe: *Nearest common ancestors: a survey and new distributed algorithm*, in Proc. 1th Annual ACM Symp. on Parallel Algorithms & Architectures, 2002, pp. 258–264.
2. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *The "runs" theorem*, 2014, *arXiv:1406.0263v6*.
3. F. Bassino, J. Clément, and C. Nicaud: *The standard factorization of Lyndon words: an average point of view*. Discrete Mathematics, 290(1) 2005, pp. 1–25.
4. W. Bland, G. Kucherov, and W. F. Smyth: *Prefix table construction and conversion*. Proc. 24th Internat. Workshop on Combinatorial Algs. (IWOCA), 2013, pp. 41–53.
5. 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.
6. M. Christodoulakis, P. J. Ryan, W. F. Smyth, and S. Wang: *Indeterminate strings, prefix arrays and undirected graphs*. Theoretical Comput. Sci., 600 2015, pp. 34–48.
7. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*, Cambridge University Press, New York, NY, USA, 2007.
8. M. Crochemore and W. Rytter: *Jewels of stringology*, World Scientific, 2002.
9. J.-P. Duval: *Factorizing words over an ordered alphabet*. J. Algorithms, 4(4) 1983, pp. 363–381.
10. J. Fischer, V. Mäkinen, and G. Navarro: *An(other) entropy-based compressed suffix tree*, in 19th Annual Symp. on Combinatorial Pattern Matching, vol. 5029 of Lecture Notes in Computer Science, Springer, 2008, pp. 152–165.
11. K. Goto and H. Bannai: *Simpler and faster Lempel-Ziv factorization*, in Data Compression Conference, 2013, pp. 133–142.
12. C. Hohlweg and C. Reutenauer: *Lyndon words, permutations and trees*. Theor. Comput. Sci., 307(1) 2003, pp. 173–178.
13. D. Kosolobov: *Lempel-Ziv factorization may be harder than computing all runs*, in Proc. 32nd Symp. on Theoretical Aspects of Computer Science, 2015, *arXiv:1409.5641*.
14. G. Nong, S. Zhang, and W. H. Chan: *Linear suffix array construction by almost pure induced–sorting*. Data Compression Conference, 0 2009, pp. 193–202.
15. E. Ohlebusch and S. Gog: *Lempel-Ziv factorization revisited*, in 22nd Annual Symp. on Combinatorial Pattern Matching, vol. 6661 of Lecture Notes in Computer Science, Springer, 2011, pp. 15–26.
16. S. J. Puglisi, W. F. Smyth, and A. H. Turpin: *A taxonomy of suffix array construction algorithms*. ACM Comput. Surv., 39(2) July 2007, pp. 1–31.
17. J. Sawada and F. Ruskey: *Generating Lyndon brackets: an addendum to "Fast algorithms to generate necklaces, unlabeled necklaces amd irreducible polynomials over GF(2)"*. J. Algorithms, 46 2003, pp. 21–26.

Appendix 1

The following result justifies the strategy employed in Algorithm MaxLyn (Figure 1):

Lemma 11 *Suppose that for some position i in a Lyndon word $\boldsymbol{x}[1..n]$, $k = \pi[i] \geq 2$. Then for every $j \in i+1\,..\,i+k-1$, $\pi[j] \leq i+k-j$.*

Proof. The result certainly holds for $i+k = n+1$, so we consider $i+k \leq n$. Assume that for some $j \in i+1\,..\,i+k-1$, $\pi[j] > i+k-j$. It follows that

$$\boldsymbol{x}[1\,..\,i+k-j+1] = \boldsymbol{x}[j\,..\,i+k], \tag{5}$$

while $\boldsymbol{x}[j-i+1\,..\,k] = \boldsymbol{x}[j\,..\,i+k-1]$. Since $\boldsymbol{x}$ is Lyndon, therefore $\boldsymbol{x}[1+k] \prec \boldsymbol{x}[i+k]$, and so we find that

$$\boldsymbol{x}[j-i+1\,..\,1+k] \prec \boldsymbol{x}[j\,..\,i+k]. \tag{6}$$

From (5) and (6) we see that $\boldsymbol{x}[1..k+1]$ has suffix $\boldsymbol{x}[j-i+1..k+1]$ satisfying $\boldsymbol{x}[j-i+1..k+1] \prec \boldsymbol{x}[1..i+k-j+1]$, contradicting the assumption that $\boldsymbol{x}$ is Lyndon. □

Appendix 2

Here we prove Theorem 12 that justifies th algorithm given in Figure 2:

Theorem 12 *For a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on alphabet Σ, totally order by $\prec$, let $ISA = ISA^{\prec}_{\boldsymbol{x}}$. Then for every $i \in 1..n$, the substring $\boldsymbol{x}[i..j]$ is a longest Lyndon factor with respect to $\prec$ if and only if*

(a) *for every $h \in i+1..j$, $ISA[j] < ISA[h]$; and*
(b) *either $j = n$ or $ISA[j+1] < ISA[i]$.*

The following well-known result is needed to prove Lemma 14:

Lemma 13 (Duval, Lemma 1.6, [9]) *Suppose $\boldsymbol{x} \in \Sigma^+$, where Σ is an alphabet totally ordered by $\prec$. Let $\boldsymbol{x} = \boldsymbol{u}^r\boldsymbol{u_1}b$, where $\boldsymbol{u}$ is nonempty, $r \geq 1$, $\boldsymbol{u_1}$ a possibly empty proper prefix of $\boldsymbol{u}$, and the letter $b \neq \boldsymbol{u}[|\boldsymbol{u_1}|+1]$.*

(a) *If $b \prec \boldsymbol{u}[|\boldsymbol{u_1}|+1]$, then $\boldsymbol{u}$ is a longest Lyndon prefix of $\boldsymbol{xy}$ for any $\boldsymbol{y}$;*
(b) *if $b \succ \boldsymbol{u}[|\boldsymbol{u_1}|+1]$, then $\boldsymbol{x}$ is Lyndon with respect to $\prec$.*

For a given string $\boldsymbol{x}[1..n]$, let $\boldsymbol{s_x}(i) = \boldsymbol{x}[i..n]$ denote the suffix of $\boldsymbol{x}$ beginning at position i. When clear from context we write just $\boldsymbol{s}(i)$.

Lemma 14 *Consider a string $\boldsymbol{x} = \boldsymbol{x}[1\,..\,n]$ over alphabet Σ totally ordered by $\prec$. Let $\boldsymbol{x}[i\,..\,j]$ be the longest Lyndon factor of $\boldsymbol{x}$ starting at i. Then $\boldsymbol{s_x}(i) \prec \boldsymbol{s_x}(k)$ for every $k \in i+1..j$ and either $j = n$ or $\boldsymbol{s_x}(j+1) \prec \boldsymbol{s_x}(i)$.*

Proof. Because $\boldsymbol{x}[i\,..\,j]$ is Lyndon, therefore for any $i < k \leq j$, $\boldsymbol{x}[i\,..\,j] \prec \boldsymbol{x}[k\,..\,j]$ and so $\boldsymbol{s}(i) \prec s(k)$. If $j = n$, we are done. So we may assume $j < n$, and we want to show that $\boldsymbol{s}(j{+}1) \prec \boldsymbol{s}(i)$. Suppose then that $\boldsymbol{s}(j{+}1) \not\prec \boldsymbol{s}(i)$. Since $\boldsymbol{s}(i)$ and $\boldsymbol{s}(j{+}1)$ are distinct, it follows that $\boldsymbol{s}(i) \prec \boldsymbol{s}(j{+}1)$. If we let $d = lcp(\boldsymbol{s}(i), \boldsymbol{s}(j{+}1)) + 1$, two cases arise:

(a) $0 \leq d \leq j - i$.
Here $i \leq i + d \leq j$. Thus $\boldsymbol{x}[i\,..\,i{+}d{-}1] = \boldsymbol{x}[j{+}1\,..\,j{+}d]$ and $\boldsymbol{x}[i{+}d] \prec \boldsymbol{x}[j{+}1{+}d]$, and so for $j < k \leq j{+}1{+}d$, $\boldsymbol{x}[i\,..\,j{+}1{+}d] \prec \boldsymbol{x}[k\,..\,j{+}1{+}d]$. Since $\boldsymbol{x}[i\,..\,j]$ is Lyndon, $\boldsymbol{x}[i\,..\,j] \prec \boldsymbol{x}[k\,..\,j]$ and so $\boldsymbol{x}[i\,..\,j{+}1{+}d] \prec \boldsymbol{x}[k\,..\,j{+}1{+}d]$ for any $i < k \leq j$. Thus $\boldsymbol{x}[i\,..\,j{+}1{+}d]$ is Lyndon, contradicting the assumption that $\boldsymbol{x}[i\,..\,j]$ is the longest Lyndon factor starting at i.

(b) $0 < j - i \leq d$.
Let $d = r(j{-}i){+}d_1$, where $0 \leq d_1 < j - i$. Then $r \geq 1$ and $\boldsymbol{x}[i\,..\,j{+}1{+}d] = \boldsymbol{u}^r\boldsymbol{u_1}b$ where $\boldsymbol{u} = \boldsymbol{x}[i\,..\,j]$,

$$\boldsymbol{u_1} = \boldsymbol{x}[j{+}r(j{-}i){+}1\,..\,j{+}r(j{-}i){+}d_1{-}1] = \boldsymbol{x}[j{+}r(j{-}i){+}1\,..\,j{+}d{-}1]$$

is a prefix of $\boldsymbol{x}[i\,..\,j]$, and $\boldsymbol{x}[i{+}d] \prec \boldsymbol{x}[j{+}1{+}d]$, so that by Lemma 13 (b), $\boldsymbol{x}[i\,..\,j{+}1{+}d]$ is Lyndon, contradicting the assumption that $\boldsymbol{x}[i\,..\,j]$ is the longest Lyndon factor starting at i.

Thus $\boldsymbol{s}(j{+}1) \prec \boldsymbol{s}(i)$, as required. □

Lemma 15 describes the property of being a longest Lyndon factor of a string $\boldsymbol{x}$ in terms of relationships between corresponding suffixes.

Lemma 15 *Consider a string $\boldsymbol{x} = \boldsymbol{x}[1\,..\,n]$ over an alphabet Σ with an ordering $\prec$. A substring $\boldsymbol{x}[i\,..\,j]$ is a longest Lyndon factor of $\boldsymbol{x}$ with respect to $\prec$ if and only if $\boldsymbol{s_x}(i) \prec \boldsymbol{s_x}(k)$ for every $k \in i + 1..j$ and either $j = n$ or $\boldsymbol{s_x}(j{+}1) \prec \boldsymbol{s_x}(i)$.*

Proof. Let (A) denote $\{\boldsymbol{x}[i\,..\,j]$ *is a longest Lyndon factor of* $\boldsymbol{x}\}$ and let (B) denote $\{\boldsymbol{s}(i) \prec \boldsymbol{s}(k)$ *for any* $1 \leq k \leq j$ *and* $\boldsymbol{s}(j{+}1) \prec \boldsymbol{s}(i)\}$. Then (A) $\Rightarrow$ (B) follows from Lemma 14, so we need to prove that (B) $\Rightarrow$ (A).

Suppose then that (B) holds, and let $\boldsymbol{x}[i\,..\,k]$ be a longest Lyndon factor of $\boldsymbol{x}$ starting at position i. If $k < j$, then by Lemma 14, $\boldsymbol{s}(k{+}1) \prec \boldsymbol{s}(i)$, a contradiction since $k{+}1 \leq j$. If $k > j$, then by Lemma 14, $\boldsymbol{s}(i) \prec \boldsymbol{s}(j{+}1)$ because $j{+}1 \leq k$, which again gives us a contradiction. Thus $k = j$ and $\boldsymbol{x}[i\,..\,j]$ is a longest Lyndon factor of $\boldsymbol{x}$. □

Now we reformulate Lemma 15 in terms of the inverse suffix array ISA of $\boldsymbol{x}$ using the relationship that $\boldsymbol{s}(i) \prec s(j) \iff \mathrm{ISA}[i] < \mathrm{ISA}[j]$, thus yielding Theorem 12, as required. Hence the Lyndon array can be computed in a simple three-step algorithm, as shown in Figure 2, that executes in $\theta(n)$ time and uses only one additional array of integers.

Appendix 3

Here we describe a simple data structure that yields an alternative approach to Algorithm NSV*, based on the comparison of longest Lyndon factors as described in Lemma 15. The ***dictionary of basic factors*** [7,8] of string $\boldsymbol{x}[1..n]$ consists of a

sequence of arrays $\mathcal{D}_t, 0 \leq t \leq \log n$. The array $\mathcal{D}_t$ records information about factors of $\boldsymbol{x}$ of length 2^t — that is, the basic factors. In particular, $\mathcal{D}_t[i]$ stores the rank of $\boldsymbol{x}[i..i + 2^t - 1]$, so that

$$\boldsymbol{x}[i..i + 2^t - 1] \preceq \boldsymbol{x}[i..i + 2^t - 1] \Leftrightarrow \mathcal{D}_t[i] \leq \mathcal{D}_t[i].$$

This dictionary requires $O(n \log n)$ space and can be constructed in $O(n \log n)$ time as follows. $\mathcal{D}_0$ contains information about consecutive symbols of $\boldsymbol{x}$ and hence can be computed in $O(n \log n)$ time by sorting all the symbols appearing in $\boldsymbol{x}$ and mapping them to numbers from 1 and onward. Once $\mathcal{D}_t$ is computed, we can easily compute $\mathcal{D}_{t+1}$ by spending $O(n)$ time on a radix sort, because $u[i..i + 2^{t+1} - 1]$ is in fact a concatenation of the factors $u[i..i + 2^t - 1]$ and $u[i + 2^t..i + 2^{t+1} - 1]$.

Once this dictionary is computed, we can compare any two factors by comparing two appropriate overlapping basic factors (i.e., factors having length power of two), which is done by checking the corresponding $\mathcal{D}$ array from the dictionary. This will require constant time and hence each suffix-suffix comparison can be done in constant time.