Algorithms to Compute the Lyndon Array

F. Franek, A.S. M. S. Islam[1], M.S. Rahman[2], & W. F. Smyth

Algorithms Research Group, Department of Computing and Software
McMaster University, Hamilton, Ontario, Canada

[1] School of Computational Science & Engineering
McMaster University, Hamilton, Canada

[2] Department of Computer Science & Engineering
Bangladesh University of Engineering & Technology

Prague Stringology Conference PSC 2016
August 2016

McMaster University

Outline

Motivation

- In 2015 *Bannai et al.* presented a method of L-roots to prove the maximum number of runs conjecture $\rho(n) < n$.
- In simple terms, an L-root of a run is any non-trivial right cyclic shift of the root of the run that happens to be Lyndon; of course it begs the question ***with respect to which ordering?***

- The ingenuity of their approach lies in the fact that either the order of the alphabet is used, or its inverse, depending on the “ending” of the run. The order is chosen so that it makes the L-root a maximal Lyndon factor (subword) starting at that position.
- The maximality of L-roots guarantee that two L-roots cannot start at the same position, hence the set of starting positions of all L-roots of a run is disjoint from the set of starting positions of L-roots for any other run.

- Thus, all runs are mapped to a partition of the indeces of the string, hence $\rho(n) < n$.

- Given all maximal Lyndon subwords of a string w.r.t. the order of the alphabet and all maximal Lyndon subwords of the string w.r.t. to the inverse order, *Bannai et al.* showed that all runs of the string can be computed in linear time.

  This is the only algorithm that does not require a prior Lempel-Ziv factorization of the string.

- The formalization of ***all maximal Lyndon subwords of a string*** $x[1..n]$ is *Lyndon array*: $L[i] = j$ where j is the length of the longest Lyndon substring starting at position i, i.e. $x[i..i{+}j{-}1]$ is a maximal Lyndon substring of x.

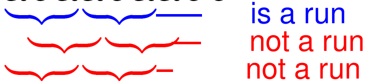
  Note that equivalently we could have defined the Lyndon array as $\lambda[i] = j$ *where* $x[i..j]$ *is a maximal Lyndon substring, i.e. the array stores the end positions of the maximal Lyndon substrings rather than their lengths. Trivially,* $\lambda[i] = i + L[i] - 1$.

- Of course, if computing the Lyndon array were more expensive than computing the Lempel-Ziv factorization, the *Bannai et al.* algorithm would not be that useful.

Basic notions

The notion of run: maximal fractional repetition that can be extended neither to the left nor to the right.

$$\cdots babaabaabb \cdots$$



Moreover the ***root*** (the repeating part) must be ***primitive*** (not a repetition)

Run can be fully described by 3 integers: starting position, ending position, and the period.

$\rho(n) = \max\{r(x) : |x| = n\}$ where $r(x)$ denotes the # of runs in the string x

Lyndon words

Definition

A string x is a *Lyndon word* if x is lexicographically strictly smaller than any non-trivial rotation of x.
Trivially true when $|x| = 1$, so-called *trivial* Lyndon word.

The following are all equivalent:

- x is a non-trivial Lyndon word
- $x[1..n] \prec x[i..n]$ for any $1 < i \leq n$
- $x[1..i] \prec x[i{+}1..n]$ for any $1 \leq i < n$
- there is $1 \leq i < n$ so that $x[1..i] \prec x[i{+}1..n]$ and both $x[1..i]$ and $x[i{+}1..n]$ are Lyndon (standard factorization *when $x[i{+}1..n]$ is the longest*)

abb is Lyndon (*abb* *bba* *bab*)

aba is not (*aba* *baa* *aab*)

abab is not (none of the rotations is strictly smallest: *abab* *baba* *abab* *baba*)

Lyndon ⇒ unbordered ⇒ aperiodic ⇒ primitive

- Lyndon words have an application to the description of free Lie algebras, this was *Lyndon's* original motivation for introducing these words.

- Linear time constant space generation of Lyndon words provides an efficient method for constructing a particular de Bruijn sequence in linear time and logarithmic space.

- Radford's theorem states that the Lyndon words are algebraically independent elements of the shuffle algebra, and generate it.

- Lyndon words correspond to aperiodic necklace class representatives and can thus be counted with Moreau's necklace-counting function.
- *Given a string s, find its Lyndon rotation*. Problem arises in chemical databases for circular molecules.The canonical representation is the lexicographically smallest rotation.

Theorem (*Chen+Fox+Lyndon*, 1958, Lyndon factorization)

For any string x there are unique Lyndon words u_1, ..., u_k so that $u_{i+1} \preceq u_i$ and $x = u_1 u_2 ... u_k$.

Duval, 1983, presented an efficient elegant linear time constant space algorithm to compute Lyndon factorization.

Each u_i is in fact a ***maximal Lyndon factor***.

Lyndon roots and L-roots

A repetition in a string can be considered a sequence of right cyclic shifts of its root:

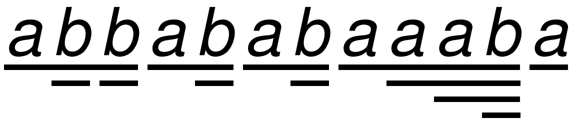
$$\begin{array}{l}
\ldots\, babbababba \ldots \\
\ldots\, babbababba \ldots \\
\ldots\, babbababba \ldots \\
\ldots\, babbababba \ldots \\
\ldots\, babbababba \ldots \\
\ldots\, babbababba \ldots
\end{array}$$

If the root of the repetition is primitive, one of the shifts is guaranteed to be Lyndon !

Hence, every run has one or more Lyndon roots !

Lyndon arrays

Lyndon factors:



Lyndon array:

3 1 1 2 1 2 1 4 3 2 1 1

Computing Lyndon array via sorting of suffixes

Lemma (*Hohlweg+Reutenauer*, 2003)

For any string $x = x[1..n]$, $x[i..j]$ is a maximal Lyndon factor of x iff $x[j{+}1..n] \prec x[i..n]$.

Definition

Suffix array $s[i]$ of a string $x = x[1..n]$ is an integer array so that $s[i] = j$ iff $x[i..n]$ is the j-th suffix in the lexicographic ordering.

Suffix array can be computed in linear time!! (*Kărkkăinen+ Sanders*, *Kim+Sim+Park+Park*, *Ko+Aluru*).

Inverse suffix array $s^{-1}[j] = i$ iff $s[i] = j$.

Can also be computed from the suffix array in linear time.

Thus, $s^{-1}[i] < s^{-1}[j]$ iff $x[i..n] \prec x[j..n]$.

Lyndon array can be computed in linear time using stack from the inverse suffix array by NSV (next smallest value) algorithm.

Lyndon array can be computed in linear time

Our contribution in the paper: though NSV had been mentioned in several papers including in *Hohlweg+Reutenauer*, we never found any formal treatment of the algorithm, so we provided a proof of the correctness of the algorithm

- A small beauty fault: computing suffix array in linear time is quite laborious and involved. It is not clear what is “simpler”: computing the Lempel-Ziv factorization or sorting the suffixes.

- Hence: a goal is to find a direct linear time algorithm to compute Lyndon array bypassing the computation of the suffix array altogether.

But, is it possible? What if by computing Lyndon array one actually sorts out the suffixes?

Holub+Islam+Smyth+F. : For a binary alphabet, except a special case when the Lyndon array is all 1's, one can determine in linear time the unique string of which it is the Lyndon array, i.e. in linear time we can sort the suffixes from the Lyndon array.

Thus, for binary strings, computing the Lyndon array seems as hard as sorting suffixes.

This is not true for alphabets of bigger sizes.

Mantaci+Restivo+Rosone+Sciortino: sorting suffixes from Lyndon decomposition.

No complexity given explicitly, but looks like $O(n^2)$.

Proposition

Let $u_1..u_k$ be the Lyndon factorization of x. Then $sort(u_1..u_k) = merge(sort(u_1..u_r), sort(u_{r+1}..u_k))$.

Can easily be reformulated in terms of Lyndon array as it gives more information than just Lyndon factorization.

The real question is whether it can be done in linear time

Duval revisited

- The elegant linear time in-place algorithm of Duval originally designed for computing Lyndon decomposition can easily be adopted for computing Lyndon array.
- The basic component of Duval's algorithm relies on the fact the Lyndon words are aperiodic and it thus identifies the longest Lyndon subword starting at a given position. For the decomposition, the next starting position is the position right after the end of the Lyndon subword just identified. But we can use it for the next adjacent position, thus obtaining $O(n^2)$ algorithm to compute Lyndon array.

- The bad thing: the worst time complexity is $O(n^2)$
  The good thing: quite simple, fast, in-place algorithm
- Quite suitable for smaller strings, though the empirical evidence for larger strings indicates that it is not that fast.
- Our contribution in the paper: nobody seems ever to be interested in using Duval's algorithm for computing Lyndon array, so we wanted explicitly to point out this additional application.

Could ranges help?

We also presented an approach based on *ranges*. A *range* in a string x is a maximal substring $x[i..j]$ so that $x[k] \preceq x[k+1]$ for any $i \leq k < j$. The ranges can be lexicographically compared in $O(|\Sigma|)$ time if we have a fixed alphabet Σ and precomputed Parikh's vectors of all ranges. The ranges can be "consolidated" into the Lyndon factors:

$$\underline{abb}\ \underline{ab\underline{abbb}}\ \underline{ab}\ \underline{ab}$$

It can be shown that the consolidation process is at least $O(|\Sigma| n \log n)$ and we have reasons to believe that it is in fact $O(|\Sigma| n \log n)$, though we have no proof yet.

What are the pitfalls: it cannot happen for binary strings, but for strings over larger alphabets, the ranges may degenerate into single letters, e.g. *hfedcb*, so in extreme cases working with ranges would be the same as working with letters. On the other hand, all such extreme cases are simple for Lyndon factors: e.g. $\underline{h}\ \underline{f}\ \underline{e}\ \underline{d}\ \underline{c}\ \underline{b}$. However $\underline{a\ \underline{h}\ \underline{f}\ \underline{e}\ \underline{d}\ \underline{c}\ \underline{b}}$.

Conclusion and future work

- We presented two well-known algorithms for an application they were not originally intended for: computing Lyndon array.
- We presented the algorithm that computes the Lyndon array from the inverse suffix array using NSV and provided a proof of correctness of NSV.
- We showed how Duval's factorization algorithm can be easily adapted to compute Lyndon array in a very simple in-place fashion in $O(n^2)$ time.

- We sketched two variants of computing Lyndon arrays based on ranges and their consolidation into Lyndon factors that are most-likely of $O(n \log n)$ complexity.
- We ran some very preliminary experimental tests on random strings to see the performances of these algorithms.

- We will determine the true complexity of the algorithms based on ranges and their consolidation.

- We will analyze the “Farah’s” approach: reduce the input string, through recursive call obtain Lyndon array of the reduced string, expand it into a partial Lyndon array for the input string, compute the missing values.
  We already have a very good reduction scheme; the problem is that the computation of the missing values is at worst $O(n\ log\ n)$, but most likely $O(n\ log\ n)$.

THANK YOU