Algorithms to Compute the Lyndon Array Revisited

Michael Liut
(joint work with F. Franek)

Advanced Optimization Laboratory
Department of Computing and Software
McMaster University, Hamilton, Ontario, Canada

August 2019

Prague Stringology Conference 2019
Czech Technical University

Outline

Background

- The motivation for having an efficient algorithm for identifying all maximal Lyndon substrings of a string comes from the work of *Bannai et al.* on the runs conjecture.

- In 2015, *Bannai et al.* presented a method of L-roots to prove the maximum number of runs conjecture $\rho(n) < n$.

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

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

- In 2017, *Franek et al.* demonstrated linear co-equivalence of sorting suffixes and sorting maximal Lyndon substrings of a string; based on a novel suffix sorting algorithm introduced by *Baier*.

- Noticed by *Diegelmann*, Phase I of *Baier*'s suffix sort identifies and sorts all maximal Lyndon substrings.

  "Sorting suffixes" is (in a sense) equivalent to "sorting maximal Lyndon substrings", which increased the interest of efficiently computing maximal Lyndon substrings.

What is a 'Lyndon word'?

Definition

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

If $\boldsymbol{x} = \boldsymbol{uv}$, then $\boldsymbol{vu}$ is called a *rotation* of $\boldsymbol{x}$; if either $\boldsymbol{u} = \varepsilon$ or $\boldsymbol{v} = \varepsilon$, then the *rotation* is called ***trivial***.

A non-empty string $\boldsymbol{x}$ is ***primitive*** if there are no string $\boldsymbol{y}$ and no integer $k \geq 2$ so that $\boldsymbol{x} = \boldsymbol{y}^k = \underbrace{\boldsymbol{yy}\cdots\boldsymbol{y}}_{k\ \text{times}}$

McMaster University

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*)

McMaster University

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 $\Rightarrow$ unbordered $\Rightarrow$ primitive

McMaster University

The Lyndon array

- The maximal Lyndon substrings of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ can be best encoded by the ***Lyndon array***: an integer array $\mathcal{L}[1..n]$ so that for any $i \in 1..n$, where $\mathcal{L}[i] =$ is the length of the maximal Lyndon substrings starting at position i.

maximal Lyndon substrings:

a b b a b a b a a a b a

Lyndon array:

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

McMaster University

Overview

Our research group over the last 4-years have presented a series of papers at PSC on the topic of maximal Lyndon substrings:

- **2016** an overview of then-current algorithms for computing the Lyndon array.
- **2017** linear co-equivalency of sorting suffixes and sorting maximal Lyndon substrings.
- **2018** an elementary linear algorithm to identify and sort all maximal Lyndon substrings, inspired by Phase I of Baier's algorithm.
- **2019** today, completes the series and summarizes what has transpired, introducing new algorithms, and showing some empirical comparisons.

Iterated Duval algorithm (IDLA)

- Presented in PSC 2016, based on Duval's work on Lyndon factorization.
- Though called "Iterated Duval", it is actually the `maxLyn(x)` procedure which is iterated:
  - IDLA applies `maxLyn(x)` to every position, while
  - Duval's factorization algorithm `maxLyn(x)` is applied to the position immediately after the maximal Lyndon prefix currently computed.

Worst-Case Complexity

$\mathcal{O}(|\boldsymbol{x}|^2)$

McMaster University

Recursive Duval algorithm (RDLA)

- Presented in PSC 2016, also based on Duval's work on Lyndon factorization (applied recursively).

For example:

If $\boldsymbol{x}[1..i_1], \boldsymbol{x}[i_1+1..i_2]...\boldsymbol{x}[i_k+1..n]$ is a Lyndon factorization of $\boldsymbol{x}$, the algorithm is recursively applied to $\boldsymbol{x}[2..i_1]$, to $\boldsymbol{x}[i_1+2..i_2]$, ..., to $\boldsymbol{x}[i_k+2..n]$, etc.

Worst-Case Complexity

$\mathcal{O}(|\boldsymbol{x}|^2)$

Special Binary Alphabet
Average Case Complexity

$\mathcal{O}(|\boldsymbol{x}|\ log(|\boldsymbol{x}|))$

Algorithmic scheme based on suffix sorting (SSLA)

- Presented in PSC 2016, based of the work of Hohlweg and Reutenauer in 2003. They characterized maximal Lyndon substrings in terms of the relationships of their suffixes.
- The Lyndon array of $\boldsymbol{x}$ is the Next Smaller Value (NSV) array of the inverse suffix array.
- The scheme is as follows:
  1. sort the suffixes;
  2. from the resulting suffix array compute the inverse suffix array; and
  3. apply NSV to the inverse suffix array.

SSLA continued

- Computing the inverse suffix array, and applying *NSV*, are 'naturally' linear. Computing the suffix array can be implemented to be linear.
- Time and space are dominated by the first step (computation of the suffix array).

Worst-Case Complexity

$\mathcal{O}(\boldsymbol{x})$

For linear suffix sorting, the input strings must be over constant or integer alphabets.

McMaster University

Algorithmic scheme based on Burrows-Wheeler transform (BWLA)

- Not presented in PSC 2016, published in JDA 2018.
- The Lyndon array is computed in a linear procedure from the Burrows-Wheeler transform of the input string during the transform's inversion.
- However, the Burrows-Wheeler transform is computed via suffix sorting so this is another approach based on suffix sorting.

Worst-Case Complexity
$\mathcal{O}(\boldsymbol{x})$

McMaster University

Baier's suffix sort Phase I inspired algorithm (BSLA)

- Presented in PSC 2018, based on *Diegelmann*'s observation that Phase I of *Baier*'s suffix sort identifies and sorts all maximal Lyndon substrings.
- In comparison to PSC 2018, the following improvements were made:
  i. simplified and streamlined analysis of the working of the algorithm; and
  ii. the implementation has been significantly improved.

Worst-Case Complexity

$\mathcal{O}(\boldsymbol{x})$

McMaster University

$\tau-$reduction algorithm (TRLA)

- The idea of the algorithm follows Farach's approach for the linear algorithm for suffix tree construction.
- The scheme for computing the Lyndon array works as follows:
  1. compute $\tau(\boldsymbol{x})$ reduction of the input string $\boldsymbol{x}$;
  2. by recursion compute the Lyndon array of $\tau(\boldsymbol{x})$; and
  3. from the Lyndon array of $\tau(\boldsymbol{x})$ compute the Lyndon array of $\boldsymbol{x}$.

Worst-Case Complexity

$\Theta(\boldsymbol{x}\ log(\boldsymbol{x}))$

McMaster University

0 1 1 0 2 3 1 2 2

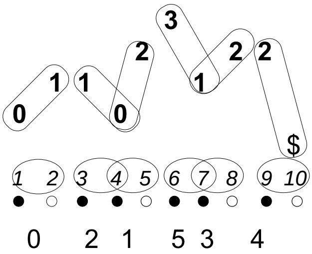


Figure: τ-reduction of string **011023122**

The rounded rectangles indicate symbol τ-pairs, the ovals indicate the τ-pairs below are the colour labels of positions, at the bottom is the τ-reduction

- For any string $\boldsymbol{x}$ of size at least 2, $\frac{1}{2}|\boldsymbol{x}| \le |\tau(\boldsymbol{x})| \le \frac{2}{3}|\boldsymbol{x}|$.

- Let $\mathcal{B}(\boldsymbol{x})$ denote the set of all black positions of $\boldsymbol{x}$.
- $$1..|\tau(\boldsymbol{x})| \underset{\mathrm{t}}{\overset{\mathrm{b}}{\rightleftarrows}} \mathcal{B}(\boldsymbol{x})$$

  b and t are bijections so that $\mathrm{b}(\mathrm{t}(j)) = j$ and $\mathrm{t}(\mathrm{b}(i)) = i$.
- We can define the Lyndon array alternatively as an integer array $\mathcal{L}'[1..n]$ so that $\mathcal{L}'[i] = j$ when $\boldsymbol{x}[i..j]$ is a maximal Lyndon substring.
- The relationship between the two definitions is straightforward: $\mathcal{L}'[i] = \mathcal{L}[i] + i - 1$, or $\mathcal{L}[i] = \mathcal{L}'[i] - i + 1$.

Theorem

Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$, $\mathcal{L}'_{\tau(\boldsymbol{x})}[1..m]$ be the Lyndon array of $\tau(\boldsymbol{x})$, and $\mathcal{L}'_{\boldsymbol{x}}[1..n]$ be the Lyndon array of $\boldsymbol{x}$. Then for any black $i \in 1..n$,

$$\mathcal{L}'_{\boldsymbol{x}}[i] = \begin{cases} b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]\big) & \text{if } \boldsymbol{x}[b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]\big)+1] \preceq \boldsymbol{x}[i] \\ b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]\big)+1 & \text{otherwise.} \end{cases}$$

McMaster University

```
L'_x[n] ← n
for i ← n − 1 downto 2
   if L'[i] = nil then
      if x[i] ≻ x[i + 1] then
         L'[i] ← i
      else
         if L'[i − 1] = i − 1 then
            stop ← n
         else
            stop ← L'[i − 1]
         L'[i] ← L'[i + 1]
         while L'[i] < stop do
            if x[i..L'[i]] ≺ x[L'[i] + 1..L'[L'[i] + 1]] then
               L'[i] ← L'[L'[i] + 1]
            else
               break
```

Figure: Computing missing values (white positions)

Empirical Analysis

- There were 4 categories of datasets:

binary	random tight binary strings over the alphabet $\{0, 1\}$;
4-ary	random tight 4-ary strings (kind of random DNA) over the alphabet $\{0, 1, 2, 3\}$;
26-ary	random tight 26-ary strings (kind of random English) over the alphabet $\{0, 1, ..., 25\}$; and
integer	random tight strings over integer alphabets.

- Each of the dataset contained 500 randomly generated strings of the same length.
- For each category, there were datasets for length: 10, 50, 10^2, $5{\cdot}10^2$, ..., 10^5, $5{\cdot}10^5$, and 10^6.

McMaster University

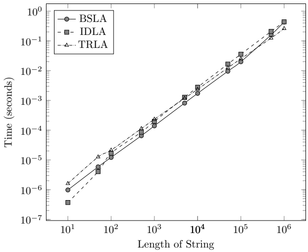
- All of the algorithms have been implemented in `C++` and are made publicly available:
  `https://www.cas.mcmaster.ca/~franek/research.html` and
  `https://github.com/MichaelLiut/Computing-LyndonArray`.

- Memory: 32GB (DDR4 @ 2400 MHz)
  CPU: 8 x Intel Xeon E5-2687W v4 @ 3.00GHz
  OS: Linux version 2.6.18-419.el5 (gcc version 4.1.2)

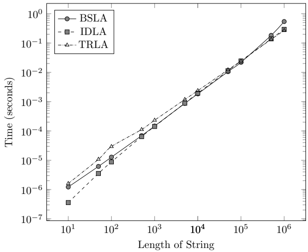
- Programs were compiled without any form of additional optimization.

- *The average time for each dataset was computed and used in the following graphs.*

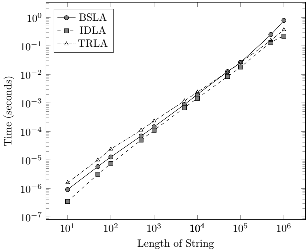
McMaster University



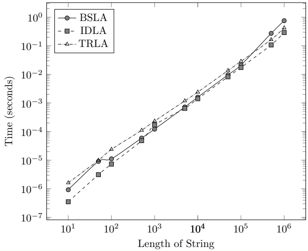
random binary



random 4-ary



random 26-ary



random integer

McMaster University

Conclusion

Let's recap what we've discussed:

- An overview of current algorithms for computing maximal Lyndon substrings and new developments since PSC 2016:
  - the algorithmic scheme based on the computation of the inverse Burrows-Wheeler transform (BWLA);
  - the linear algorithm inspired by Phase I of Baier's algorithm (BSLA); and
  - the novel algorithm based on $\tau-$reduction (TRLA).
- The performance and empirical analysis of three of the presented algorithms: IDLA, BSLA, and TRLA, on various datasets.

THANK YOU