Inducing the Lyndon Array

Felipe A. Louza[1] Sabrina Mantaci[2] Giovanni Manzini[3,4]
Marinella Sciortino[2] Guilherme P. Telles[5]

[1]Department of Computing and Mathematics, University of São Paulo, Brazil

[2]Dipartimento di Matematica e Informatica, Università di Palermo, Italy

[3]University of Eastern Piedmont, Alessandria, Italy

[4]IIT CNR, Pisa, Italy

[5]Institute of Computing, University of Campinas, Brazil

SPIRE 2019
Segovia, Spain - October 7th, 2019

Our contribution

- We propose an algorithm to compute simultaneously the Lyndon Array LA and the Suffix Array SA of a text.

Our contribution

- We propose an algorithm to compute simultaneously the Lyndon Array LA and the Suffix Array SA of a text.
    - It is a variant of the algorithm by Nong *et al.*, 2013 for suffix array construction based on *induced suffix sorting*

Our contribution

- We propose an algorithm to compute simultaneously the Lyndon Array LA and the Suffix Array SA of a text.
    - It is a variant of the algorithm by Nong *et al.*, 2013 for suffix array construction based on *induced suffix sorting*
    - Time complexity: $O(n)$, where n is the length of the text
    - Working space: $\sigma + O(1)$ words, where σ is the alphabet size

Our contribution

- We propose an algorithm to compute simultaneously the Lyndon Array LA and the Suffix Array SA of a text.
  - It is a variant of the algorithm by Nong *et al.*, 2013 for suffix array construction based on *induced suffix sorting*
  - Time complexity: $O(n)$, where n is the length of the text
  - Working space: $\sigma + O(1)$ words, where σ is the alphabet size
- Our result improves the previous best space requirement for linear time computation of the Lyndon array.
- Experimental results with real and synthetic datasets show that our algorithm is not only space-efficient but also fast in practice.

Main objects: Lyndon Word and Lyndon Array

- $T = T[1] \dots T[n]$ is a string of length n over a fixed ordered alphabet Σ of size σ.
- $T[i, j]$ denotes the factor of T starting from the i-th symbol and ending at the j-th symbol.
- The i-th rotation of T begins with $T[i+1]$, corresponding to the string $T' = T[i+1, n]T[1, i]$.
    - *banbaa* is the 1-st rotation of *abanba*
- A string of length n has n possible rotations.
- A string T is *primitive* if it has n distinct rotations
- A primitive string T is called a *Lyndon word* if it is the lexicographical least among its rotations.
    - *aabanb* is a Lyndon word

Main objects: Lyndon Word and Lyndon Array

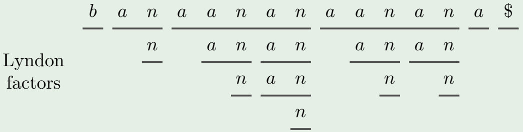
Definition

Given a string $T = T[1] \ldots T[n]$, the Lyndon array (LA) of T is an array of integers in the range $[1, n]$ that, at each position $i = 1, \ldots, n$, stores the length of the longest Lyndon factor of T starting at i:

$$\mathsf{LA}[i] = \max\{\ell \mid T[i, i+\ell-1] \text{ is a Lyndon word}\}.$$

Example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
LA =	1	2	1	5	2	1	2	1	5	2	1	2	1	1	1



Several aspects of Lyndon array

- Lyndon array generalizes the Lyndon factorization of a text: a string can be uniquely factorized in a non-increasing sequence of Lyndon words (Chen-Fox-Lyndon, 1958).

Several aspects of Lyndon array

- Lyndon array generalizes the Lyndon factorization of a text: a string can be uniquely factorized in a non-increasing sequence of Lyndon words (Chen-Fox-Lyndon, 1958).
- Bannai *et al.* (2017) used Lyndon arrays to prove the very known conjecture by Kolpakov and Kucherov (1999): the number of runs (maximal periodicities) in a string of length n is smaller than n.

Several aspects of Lyndon array

- Lyndon array generalizes the Lyndon factorization of a text: a string can be uniquely factorized in a non-increasing sequence of Lyndon words (Chen-Fox-Lyndon, 1958).
- Bannai *et al.* (2017) used Lyndon arrays to prove the very known conjecture by Kolpakov and Kucherov (1999): the number of runs (maximal periodicities) in a string of length n is smaller than n.
- The computation of the Lyndon array of a text T is strictly related to the construction of the Lyndon tree of the string $\$T$[1] (Crochemore, Russo, 2018)

---

[1] the symbol \$ is smaller than any symbol of the alphabet Σ

Several aspects of Lyndon array

- Lyndon array generalizes the Lyndon factorization of a text: a string can be uniquely factorized in a non-increasing sequence of Lyndon words (Chen-Fox-Lyndon, 1958).
- Bannai *et al.* (2017) used Lyndon arrays to prove the very known conjecture by Kolpakov and Kucherov (1999): the number of runs (maximal periodicities) in a string of length n is smaller than n.
- The computation of the Lyndon array of a text T is strictly related to the construction of the Lyndon tree of the string $\$T$[1] (Crochemore, Russo, 2018)
- Lyndon array can be used as preprocessing for efficient constructions of suffix array of a text (Bayer, 2016).

---

[1] the symbol $\$$ is smaller than any symbol of the alphabet Σ

Several aspects of Lyndon array

- Lyndon array generalizes the Lyndon factorization of a text: a string can be uniquely factorized in a non-increasing sequence of Lyndon words (Chen-Fox-Lyndon, 1958).
- Bannai *et al.* (2017) used Lyndon arrays to prove the very known conjecture by Kolpakov and Kucherov (1999): the number of runs (maximal periodicities) in a string of length n is smaller than n.
- The computation of the Lyndon array of a text T is strictly related to the construction of the Lyndon tree of the string $\$T$[1] (Crochemore, Russo, 2018)
- Lyndon array can be used as preprocessing for efficient constructions of suffix array of a text (Bayer, 2016).
- Known efficient constructions for Lyndon array involve other structures...

---

[1] the symbol \$ is smaller than any symbol of the alphabet Σ

Sorting suffixes of a string T

- T_i denotes the suffix $T[i, n]$ of T. We assume that $T[n] = \$$.
- Suffix Array SA: It is an array of integers in the range $[1, n]$ that gives the lexicographic order of all suffixes of T, that is $T_{\mathrm{SA}[1]} < T_{\mathrm{SA}[2]} < \cdots < T_{\mathrm{SA}[n]}$.
- The suffix array can be computed in $O(n)$ time using $\sigma + O(1)$ words of working space (see for instance, Nong et al, 2013).

Example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
SA =	15	14	9	4	12	7	2	10	5	1	13	8	3	11	6

Lyndon Array and Suffix array

- If the SA of T is known, the Lyndon array LA can be computed in linear time by using:

Lemma

The factor $T[i, i+\ell-1]$ *is the longest Lyndon factor of* T *starting at* i *iff* $T_i < T_{i+k}$, *for* $1 \leq k < \ell$, *and* $T_i > T_{i+\ell}$. *Therefore,* $\mathrm{LA}[i] = \ell$.

Lyndon Array and Suffix array

- If the SA of T is known, the Lyndon array LA can be computed in linear time by using:

Lemma

The factor $T[i, i+\ell-1]$ *is the longest Lyndon factor of* T *starting at* i *iff* $T_i < T_{i+k}$*, for* $1 \leq k < \ell$*, and* $T_i > T_{i+\ell}$*. Therefore,* $\mathrm{LA}[i] = \ell$*.*

- Previous Lemma is useful when Inverse Suffix Array ISA is used: it stores the inverse permutation of SA, such that $\mathrm{ISA}[\mathrm{SA}[i]] = i$.

Lyndon Array and Suffix array

- If the SA of T is known, the Lyndon array LA can be computed in linear time by using:

Lemma

The factor $T[i, i+\ell-1]$ *is the longest Lyndon factor of* T *starting at* i *iff* $T_i < T_{i+k}$, *for* $1 \leq k < \ell$, *and* $T_i > T_{i+\ell}$. *Therefore,* $\mathrm{LA}[i] = \ell$.

- Previous Lemma is useful when Inverse Suffix Array ISA is used: it stores the inverse permutation of SA, such that $\mathrm{ISA}[\mathrm{SA}[i]] = i$.
- By using a result by Hohlweg and Reutenauer (2003) that combines ISA and the notion Next Smallest Value NSV of an array, Lyndon array can be computed in linear time (Franek *et al.* 2016).

Efficient Constructions of LA up to now

- Franek et al. 2016: Construct ISA and $\mathrm{NSV}_{\mathrm{ISA}}$ in linear time by using an auxiliary stack.
  Space: $n \log \sigma$ bits for $T + 2n$ words for LA and ISA, and the space for the auxiliary stack. The stack size is n words in the worst case.

Efficient Constructions of LA up to now

- Franek et al. 2016: Construct ISA and $\mathsf{NSV}_{\mathsf{ISA}}$ in linear time by using an auxiliary stack.
  Space: $n \log \sigma$ bits for T + $2n$ words for LA and ISA, and the space for the auxiliary stack. The stack size is n words in the worst case.
- Crochemore, Russo 2018: LA can be computed in linear time from the Cartesian tree built for ISA.

Efficient Constructions of LA up to now

- Franek et al. 2016: Construct ISA and $\mathrm{NSV}_{\mathrm{ISA}}$ in linear time by using an auxiliary stack.
  Space: $n \log \sigma$ bits for $T + 2n$ words for LA and ISA, and the space for the auxiliary stack. The stack size is n words in the worst case.
- Crochemore, Russo 2018: LA can be computed in linear time from the Cartesian tree built for ISA.
- Franek et al, 2017: LA can be computed in linear time during the Baier's suffix array construction algorithm
  Space: $n \log \sigma$ bits plus $2n$ words for LA and SA plus $2n$ words for auxiliary integer arrays.

Efficient Constructions of LA up to now

- Franek et al. 2016: Construct ISA and $\mathsf{NSV}_{\mathsf{ISA}}$ in linear time by using an auxiliary stack.
  Space: $n \log \sigma$ bits for $T + 2n$ words for LA and ISA, and the space for the auxiliary stack. The stack size is n words in the worst case.
- Crochemore, Russo 2018: LA can be computed in linear time from the Cartesian tree built for ISA.
- Franek et al, 2017: LA can be computed in linear time during the Baier's suffix array construction algorithm
  Space: $n \log \sigma$ bits plus $2n$ words for LA and SA plus $2n$ words for auxiliary integer arrays.
- Louza *et al.*, 2018: LA is computed in linear time during the Burrows-Wheeler inversion, using $n \log \sigma$ bits for T plus $2n$ words for LA and an auxiliary integer array, plus a stack with twice the size as the one used to compute $\mathsf{NSV}_{\mathsf{ISA}}$.

Our LA construction

- It is based on Induced Suffix Sorting $\mathrm{SACA\text{-}K}$ proposed by Nong *et al.* in 2013.
- $\mathrm{SACA\text{-}K}$ constructs the suffix array in linear time and $\sigma + O(1)$ words of working space.
- We propose a variant of $\mathrm{SACA\text{-}K}$ algorithm to compute in linear time the Lyndon array as by-product. It uses $\sigma + O(1)$ words of working space.
- Our strategy is optimal for strings from alphabet of constant size.

LMS factors in SACA-K algorithm

- Each suffix T_i of $T[1, n]$ is classified according to its lexicographical rank relative to T_{i+1}.
- A suffix T_i is S-type if $T_i < T_{i+1}$, otherwise T_i is L-type.
- A suffix T_i is LMS-type (leftmost S-type) if T_i is S-type and T_{i-1} is L-type.
- The type of each suffix can be computed with a right-to-left scanning of T

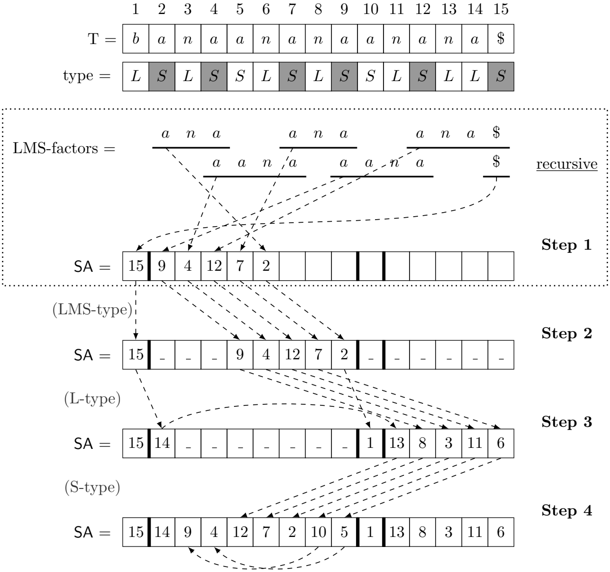
LMS factors in SACA-K algorithm

- Each suffix T_i of $T[1, n]$ is classified according to its lexicographical rank relative to T_{i+1}.
- A suffix T_i is S-type if $T_i < T_{i+1}$, otherwise T_i is L-type.
- A suffix T_i is LMS-type (leftmost S-type) if T_i is S-type and T_{i-1} is L-type.
- The type of each suffix can be computed with a right-to-left scanning of T
- $T[i]$ is LMS-type if and only if T_i is LMS-type.
- A LMS-factor of T is a factor that begins with a LMS-type symbol and ends with the following LMS-type symbol.

SACA-K in 4 steps

1. Sort all LMS-type suffixes recursively into SA_1, stored in $\mathrm{SA}[1, n/2]$.
2. Scan SA_1 from right-to-left, and insert the LMS-suffixes into the tail of their corresponding c-buckets (containing the suffixes starting with c) in SA.
3. Induce L-type suffixes by scanning SA left-to-right: for each suffix $\mathrm{SA}[i]$, if $T_{\mathrm{SA}[i]-1}$ is L-type, insert $\mathrm{SA}[i]-1$ into the head of its bucket.
4. Induce S-type suffixes by scanning SA right-to-left: for each suffix $\mathrm{SA}[i]$, if $T_{\mathrm{SA}[i]-1}$ is S-type, insert $\mathrm{SA}[i]-1$ into the tail of its bucket.

An example


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
T = b a n a a n a n a a n a n a $
type = L S L S S L S L S S L S L L S
LMS-factors =
a n a
a n a
a n a $
a a n a
a a n a
$
recursive
Step 1
SA = 15 9 4 12 7 2
(LMS-type)
Step 2
SA = 15 _ _ _ 9 4 12 7 2 _ _ _ _ _ _
(L-type)
Step 3
SA = 15 14 _ _ _ _ _ _ _ 1 13 8 3 11 6
(S-type)
Step 4
SA = 15 14 9 4 12 7 2 10 5 1 13 8 3 11 6

Key points in our strategy

- We set all positions $\text{LA}[i] = 0$, for $1 \leq i \leq n$.

Key points in our strategy

- We set all positions $\mathrm{LA}[i] = 0$, for $1 \leq i \leq n$.
- In Step 4, when SA is scanned from right-to-left, each value $\mathrm{SA}[i]$, corresponding to $T_{\mathrm{SA}[i]}$, is read in its final (correct) position i (in decreasing order) in SA.

Key points in our strategy

- We set all positions $\mathsf{LA}[i] = 0$, for $1 \leq i \leq n$.
- In Step 4, when SA is scanned from right-to-left, each value $\mathsf{SA}[i]$, corresponding to $T_{\mathsf{SA}[i]}$, is read in its final (correct) position i (in decreasing order) in SA.
- By Lemma, $\mathsf{LA}[\mathsf{SA}[i]] = \ell$ iff $T_{\mathsf{SA}[i]+\ell}$ is the next suffix (in text order) that is smaller than $T_{\mathsf{SA}[i]}$.

Key points in our strategy

- We set all positions $\mathsf{LA}[i] = 0$, for $1 \leq i \leq n$.
- In Step 4, when SA is scanned from right-to-left, each value $\mathsf{SA}[i]$, corresponding to $T_{\mathsf{SA}[i]}$, is read in its final (correct) position i (in decreasing order) in SA.
- By Lemma, $\mathsf{LA}[\mathsf{SA}[i]] = \ell$ iff $T_{\mathsf{SA}[i]+\ell}$ is the next suffix (in text order) that is smaller than $T_{\mathsf{SA}[i]}$.
- $T_{\mathsf{SA}[i]+\ell}$ is the first suffix in $T_{\mathsf{SA}[i]+1}, T_{\mathsf{SA}[i]+2} \ldots, T_n$ that has not yet been read in SA.

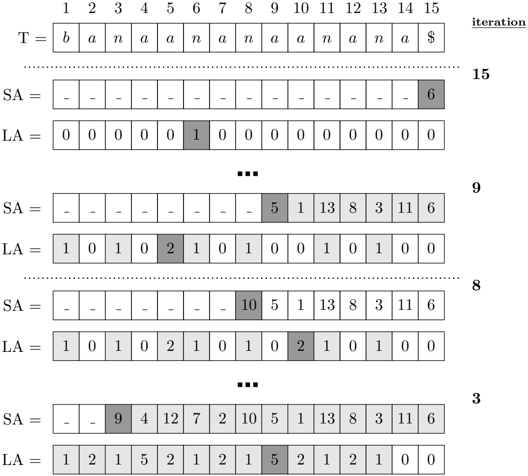
Key points in our strategy

- We set all positions $\mathsf{LA}[i] = 0$, for $1 \leq i \leq n$.
- In Step 4, when SA is scanned from right-to-left, each value $\mathsf{SA}[i]$, corresponding to $T_{\mathsf{SA}[i]}$, is read in its final (correct) position i (in decreasing order) in SA.
- By Lemma, $\mathsf{LA}[\mathsf{SA}[i]] = \ell$ iff $T_{\mathsf{SA}[i]+\ell}$ is the next suffix (in text order) that is smaller than $T_{\mathsf{SA}[i]}$.
- $T_{\mathsf{SA}[i]+\ell}$ is the first suffix in $T_{\mathsf{SA}[i]+1}, T_{\mathsf{SA}[i]+2} \ldots, T_n$ that has not yet been read in SA.
- Therefore, during Step 4 we compute $\mathsf{LA}[\mathsf{SA}[i]]$ by scanning $\mathsf{LA}[\mathsf{SA}[i]+1, n]$ to the right up to the first position $\mathsf{LA}[\mathsf{SA}[i]+\ell] = 0$

Key points in our strategy

- We set all positions $\mathsf{LA}[i] = 0$, for $1 \leq i \leq n$.
- In Step 4, when SA is scanned from right-to-left, each value $\mathsf{SA}[i]$, corresponding to $T_{\mathsf{SA}[i]}$, is read in its final (correct) position i (in decreasing order) in SA.
- By Lemma, $\mathsf{LA}[\mathsf{SA}[i]] = \ell$ iff $T_{\mathsf{SA}[i]+\ell}$ is the next suffix (in text order) that is smaller than $T_{\mathsf{SA}[i]}$.
- $T_{\mathsf{SA}[i]+\ell}$ is the first suffix in $T_{\mathsf{SA}[i]+1}, T_{\mathsf{SA}[i]+2} \dots, T_n$ that has not yet been read in SA.
- Therefore, during Step 4 we compute $\mathsf{LA}[\mathsf{SA}[i]]$ by scanning $\mathsf{LA}[\mathsf{SA}[i] + 1, n]$ to the right up to the first position $\mathsf{LA}[\mathsf{SA}[i] + \ell] = 0$
- We set $\mathsf{LA}[\mathsf{SA}[i]] = \ell$.

Running example



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	**iteration**
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$	
SA =	_	_	_	_	_	_	_	_	_	_	_	_	_	_	6	**15**
LA =	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	
								...								
SA =	_	_	_	_	_	_	_	_	5	1	13	8	3	11	6	**9**
LA =	1	0	1	0	2	1	0	1	0	0	1	0	1	0	0	
SA =	_	_	_	_	_	_	_	10	5	1	13	8	3	11	6	**8**
LA =	1	0	1	0	2	1	0	1	0	2	1	0	1	0	0	
								...								
SA =	_	_	9	4	12	7	2	10	5	1	13	8	3	11	6	**3**
LA =	1	2	1	5	2	1	2	1	5	2	1	2	1	0	0	

Running time and space

Proposition

The Lyndon array and the suffix array of a string $T[1, n]$ over an alphabet of size σ can be computed simultaneously in $O(n \cdot \text{avelyn})$ time using $\sigma + O(1)$ words of working space, where avelyn is equal to the average value in $\text{LA}[1, n]$.

Question

Is it possible to reduce the running time to $O(n)$?

Result

Yes, each LA entry can be computed in constant time.

Reducing running time to $O(n)$

- We use two additional pointer arrays NEXT$[1, n]$ and PREV$[1, n]$:

Definition

For $i = 1, \ldots, n-1$, NEXT$[i] = \min\{\ell | i < \ell \leq n$ and LA$[\ell] = 0\}$. In addition, we define NEXT$[n] = n + 1$.

Definition

For $i = 2, \ldots, n$, PREV$[i] = \ell$, such that NEXT$[\ell] = i$ and LA$[\ell] = 0$. In addition, we define PREV$[1] = 0$.

In other words, NEXT$[i]$ points to the next smaller position ℓ in LA equal to zero, and PREV$[i]$ is the inverse pointer.

Reducing running time to $O(n)$

- We set $\mathrm{NEXT}[i] = i + 1$ and $\mathrm{PREV}[i] = i - 1$, for $1 \leq i \leq n$.
- At each iteration $i = n, n-1, \ldots, 1$, we compute $\mathrm{LA}[j]$ with $j = \mathrm{SA}[i]$ by setting:

$$\mathrm{LA}[j] = \mathrm{NEXT}[j] - j \tag{1}$$

- We update the pointers arrays as follows:

$$\mathrm{NEXT}[\mathrm{PREV}[j]] = \mathrm{NEXT}[j], \quad \text{if } \mathrm{PREV}[j] > 0 \tag{2}$$

$$\mathrm{PREV}[\mathrm{NEXT}[j]] = \mathrm{PREV}[j], \quad \text{if } \mathrm{NEXT}[j] < n + 1 \tag{3}$$

Theorem

The Lyndon array and the suffix array of a string $T[1, n]$ over an alphabet of size σ can be computed simultaneously in $O(n)$ time using $2n + \sigma + O(1)$ words of working space.

Getting rid of a pointer

- We store only one array, say $\mathsf{A}[1, n]$, keeping NEXT/PREV information together.
- The array NEXT is initially stored into the space of $\mathsf{A}[1, n]$, then we reuse $\mathsf{A}[1, n]$ to store the (useful) entries of PREV.
- Note that when we write $\mathsf{LA}[j] = \ell$, the value in $\mathsf{A}[j]$, that is $\mathsf{NEXT}[j]$ is no more used by the algorithm. Then, we can reuse $\mathsf{A}[j]$ to store $\mathsf{PREV}[j + 1]$.
- PREV can be computed in terms of A and LA:

$$\mathsf{PREV}[j] = \begin{cases} j - 1 & \text{if } \mathsf{LA}[j-1] = 0 \\ \mathsf{A}[j-1] & \text{otherwise.} \end{cases} \tag{4}$$

Getting rid of both pointer arrays

- The space of LA$[1, n]$ is used to store both the auxiliary array A$[1, n]$ and the final values of LA.

Getting rid of both pointer arrays

- The space of $\mathsf{LA}[1, n]$ is used to store both the auxiliary array $\mathsf{A}[1, n]$ and the final values of LA.

Lemma

$\mathsf{LA}[j] = 1$ *iff* T_j *is an L-type suffix, or* $i = n$.

Getting rid of both pointer arrays

- The space of $\mathsf{LA}[1, n]$ is used to store both the auxiliary array $\mathsf{A}[1, n]$ and the final values of LA.

Lemma

$\mathsf{LA}[j] = 1$ *iff* T_j *is an L-type suffix, or* $i = n$*.*

Lemma

The LA*-entries corresponding to S-type suffixes are always inserted on the left of a block (possibly of size one) of non-zero entries in* $\mathsf{LA}[1, n]$*.*

Getting rid of both pointer arrays

- The space of $\mathrm{LA}[1, n]$ is used to store both the auxiliary array $\mathrm{A}[1, n]$ and the final values of LA.

Lemma

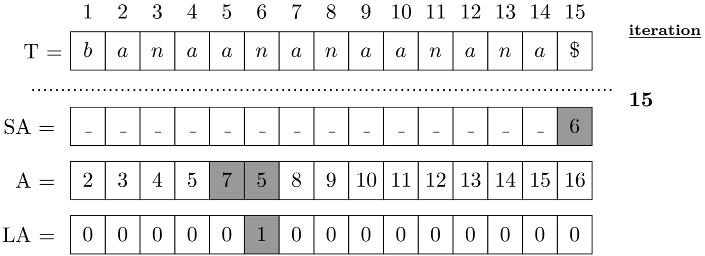
$\mathrm{LA}[j] = 1$ *iff* T_j *is an L-type suffix, or* $i = n$*.*

Lemma

The LA*-entries corresponding to S-type suffixes are always inserted on the left of a block (possibly of size one) of non-zero entries in* $\mathrm{LA}[1, n]$*.*

- We can update PREV information only for right-most entry of each block of non empty entries, which corresponds to a position of an L-type suffix.

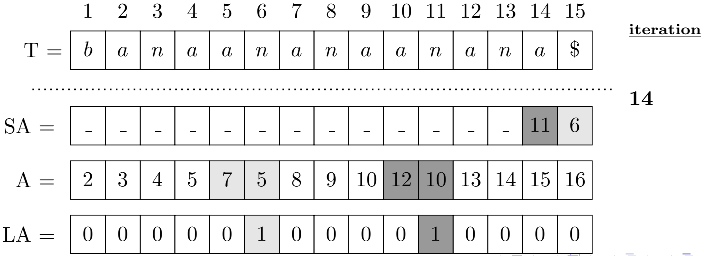
Running example



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
SA =	_	_	_	_	_	_	_	_	_	_	_	_	_	_	6
A =	2	3	4	5	7	5	8	9	10	11	12	13	14	15	16
LA =	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

iteration

15



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
SA =	_	_	_	_	_	_	_	_	_	_	_	_	_	11	6
A =	2	3	4	5	7	5	8	9	10	12	10	13	14	15	16
LA =	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0

iteration

14

Running example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
SA =	_	_	_	_	_	_	_	_	_	_	_	_	3	11	6
A =	2	4	2	5	7	5	8	9	10	12	10	13	14	15	16
LA =	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0

iteration

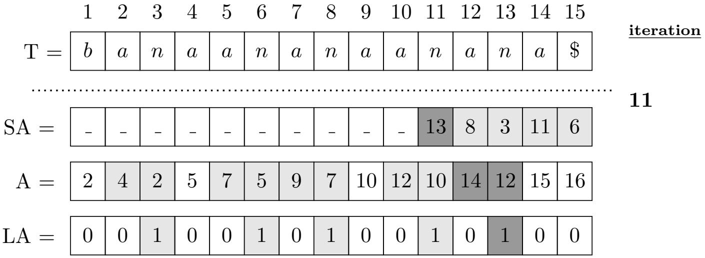
13

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
SA =	_	_	_	_	_	_	_	_	_	_	_	8	3	11	6
A =	2	4	2	5	7	5	9	7	10	12	10	13	14	15	16
LA =	0	0	1	0	0	1	0	1	0	0	1	0	0	0	0

iteration

12

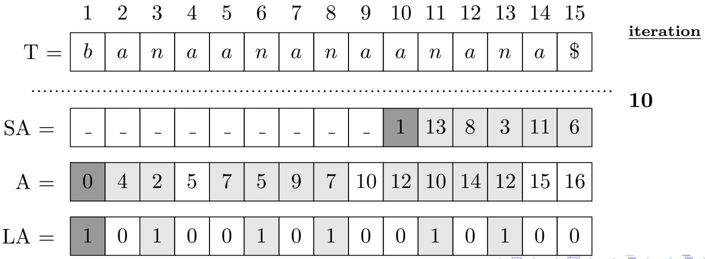
Running example



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
SA =	_	_	_	_	_	_	_	_	_	_	13	8	3	11	6
A =	2	4	2	5	7	5	9	7	10	12	10	14	12	15	16
LA =	0	0	1	0	0	1	0	1	0	0	1	0	1	0	0

iteration

11

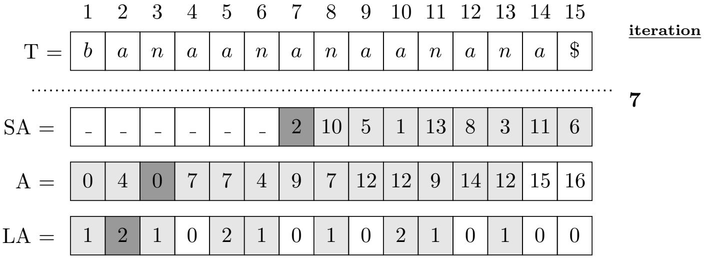


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
SA =	_	_	_	_	_	_	_	_	_	1	13	8	3	11	6
A =	0	4	2	5	7	5	9	7	10	12	10	14	12	15	16
LA =	1	0	1	0	0	1	0	1	0	0	1	0	1	0	0

iteration

10

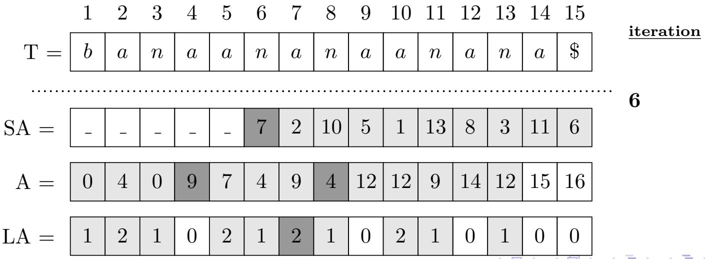
Running example



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
SA =	_	_	_	_	_	_	2	10	5	1	13	8	3	11	6
A =	0	4	0	7	7	4	9	7	12	12	9	14	12	15	16
LA =	1	2	1	0	2	1	0	1	0	2	1	0	1	0	0

iteration

7



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
SA =	_	_	_	_	_	7	2	10	5	1	13	8	3	11	6
A =	0	4	0	9	7	4	9	4	12	12	9	14	12	15	16
LA =	1	2	1	0	2	1	2	1	0	2	1	0	1	0	0

iteration

6

Running example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
SA =	_	14	9	4	12	7	2	10	5	1	13	8	3	11	6
A =	0	4	0	9	7	4	9	0	14	12	9	14	0	0	16
LA =	1	2	1	5	2	1	2	1	5	2	1	2	1	1	0

iteration

2

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
SA =	15	14	9	4	12	7	2	10	5	1	13	8	3	11	6
A =	0	4	0	9	7	4	9	0	14	12	9	14	0	0	16
LA =	1	2	1	5	2	1	2	1	5	2	1	2	1	1	1

iteration

1

Getting rid of both pointer arrays

After the modified Step 4, we sequentially scan $\mathsf{A}[1, n]$ overwriting its values with LA as follows:

$$\mathsf{LA}[j] = \begin{cases} 1 & \text{if } \mathsf{A}[j] < j \\ \mathsf{A}[j] - j & \text{otherwise.} \end{cases} \tag{5}$$

Theorem

The Lyndon array and the suffix array of a string of length n over an alphabet of size σ can be computed simultaneously in $O(n)$ time using $\sigma + O(1)$ words of working space.

Experimental results

			LA			LA and SA				SA
dataset	σ	$n/2^{20}$	NSV-Lyndon[2]	Baier-LA[3]	BWT-Lyndon[4]	Baier-LA+SA [2]	SACA-K+LA-17n	SACA-K+LA-13n	SACA-K+LA-9n	SACA-K[5]
`pitches`	133	53	**0.15**	0.20	0.20	0.26	0.26	0.22	**0.18**	0.13
`sources`	230	201	**0.26**	0.28	0.32	0.37	0.46	0.41	**0.34**	0.24
`xml`	97	282	**0.29**	0.31	0.35	0.42	0.52	0.47	**0.38**	0.27
`dna`	16	385	0.39	**0.28**	0.49	**0.43**	0.69	0.60	0.52	0.36
`english.1GB`	239	1,047	0.46	**0.39**	0.56	**0.57**	0.84	0.74	0.60	0.42
`proteins`	27	1,129	0.44	**0.40**	0.53	0.66	0.89	0.69	**0.58**	0.40
`einstein-de`	117	88	0.34	**0.28**	0.38	**0.39**	0.57	0.54	0.44	0.31
`kernel`	160	246	0.29	**0.29**	0.39	**0.38**	0.53	0.47	**0.38**	0.26
`fib41`	2	256	0.34	**0.07**	0.45	**0.18**	0.66	0.57	0.46	0.32
`cere`	5	440	0.27	**0.09**	0.33	**0.17**	0.43	0.41	0.35	0.25
`bbba`	2	100	0.04	**0.02**	0.05	**0.03**	0.05	0.04	**0.03**	0.03

Table: Running time (μs/input byte).

---

[2] Franek *et al.* 2016

[3] Baier 2016, Franek *et al.* 2017

[4] Louza *et al.* 2018

[5] Nong *et al.* 2013

Experimental Results

			LA			LA and SA				SA
dataset	σ	$n/2^{20}$	NSV-Lyndon[6]	Baier-LA[7]	BWT-Lyndon[8]	Baier-LA+SA[7]	SACA-K+LA-17n	SACA-K+LA-13n	SACA-K+LA-9n	SACA-K[9]
`pitches`	133	53	**9**	17	**9**	17	17	13	**9**	5
`sources`	230	201	**9**	17	**9**	17	17	13	**9**	5
`xml`	97	282	**9**	17	**9**	17	17	13	**9**	5
`dna`	16	385	**9**	17	**9**	17	17	13	**9**	5
`english.1GB`	239	1,047	**9**	17	**9**	17	17	13	**9**	5
`proteins`	27	1,129	**9**	17	**9**	17	17	13	**9**	5
`einstein-de`	117	88	**9**	17	**9**	17	17	13	**9**	5
`kernel`	160	246	**9**	17	**9**	17	17	13	**9**	5
`fib41`	2	256	**9**	17	**9**	17	17	13	**9**	5
`cere`	5	440	**9**	17	**9**	17	17	13	**9**	5
`bbba`	2	100	**13**	17	17	17	17	13	**9**	5

Table: Peak space (bytes/input size).

---

[6] Franek *et al.* 2016

[7] Baier 2016, Franek *et al.* 2017

[8] Louza *et al.* 2018

[9] Nong *et al.* 2013

Conclusion

- We have introduced an algorithm for computing simultaneously the suffix array and Lyndon array (LA) of a text using induced suffix sorting.
- The most space-economical variant of our algorithm uses only $n + \sigma + O(1)$ words of working space making it the most space economical LA algorithm among the ones running in linear time; this includes both the algorithm computing the SA and LA and the ones computing only the LA.
- By experiments[10], our algorithm is only slightly slower than the available alternatives, and that computing the SA is usually the most expensive step of all linear time LA construction algorithms
- A natural open problem is to design a linear time algorithm to construct only the LA using $o(n)$ words of working space.

---

[10] The source-code is publicly available at `https://github.com/felipelouza/lyndon-array/`

Thanks for your attention!

Next Smaller Value

- Next Smaller Value of an array A NSV_A: Given an array A of size n, it is an array of size n such that $\mathsf{NSV}_\mathsf{A}[i]$ contains the smallest position $j > i$ such that $\mathsf{A}[j] < \mathsf{A}[i]$, or $n+1$ if such a position j does not exist. Formally:

$$\mathsf{NSV}_\mathsf{A}[i] = \min\big\{\{n+1\} \cup \{i < j \le n \mid \mathsf{A}[j] < \mathsf{A}[i]\}\big\}.$$

Example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A =	10	7	13	4	9	15	6	12	3	8	14	5	11	2	1
NSV =	2	4	4	9	7	7	9	9	14	12	12	14	14	15	16

Lyndon Array from SA (Hohlweg, Reutenauer 2003)

- If the SA of T is known, the Lyndon array LA can be computed in linear time by using:

Lemma

The factor $T[i, i+\ell-1]$ *is the longest Lyndon factor of* T *starting at* i *iff* $T_i < T_{i+k}$, *for* $1 \leq k < \ell$, *and* $T_i > T_{i+\ell}$. *Therefore,* $\mathrm{LA}[i] = \ell$.

- Previous Lemma can be reformulated in terms of ISA or NSV:

Lemma

$\mathrm{LA}[i] = \ell$ *if and only if* $\mathrm{ISA}[i] < \mathrm{ISA}[i+k]$ *for each* $1 \leq k < \ell$ *and* $\mathrm{ISA}[i] > \mathrm{ISA}[i+\ell]$.

Lemma

$\mathrm{LA}[i] = \ell$ *if and only if* $i + \ell = \mathrm{NSV}_{\mathrm{ISA}}[i]$

An example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
SA =	15	14	9	4	12	7	2	10	5	1	13	8	3	11	6
ISA =	10	7	13	4	9	15	6	12	3	8	14	5	11	2	1
$\mathrm{NSV}_{\mathrm{ISA}}$ =	2	4	4	9	7	7	9	9	14	12	12	14	14	15	16
LA =	1	2	1	5	2	1	2	1	5	2	1	2	1	1	1