Cartesian and Lyndon trees

Maxime Crochemore, Luís M.S. Russo

▶ **To cite this version:**

Maxime Crochemore, Luís M.S. Russo. Cartesian and Lyndon trees. Theoretical Computer Science, 2020, 806, pp.1-9. 10.1016/j.tcs.2018.08.011 . hal-04523393

HAL Id: hal-04523393
https://hal.science/hal-04523393

Submitted on 27 Mar 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Cartesian and Lyndon trees

Maxime Crochemore[a,*], Luís M. S. Russo[b]

[a] *King's College London, London WC2B 4BG, UK and Université Paris-Est, France.*
[b] *INESC-ID and the Department of Computer Science and Engineering Instituto Superior Técnico, Universidade de Lisboa.*

Abstract

The article describes the structural and algorithmic relations between Cartesian trees and Lyndon Trees. This leads to a uniform presentation of the Lyndon table of a word corresponding to the Next Nearest Smaller table of a sequence of numbers. It shows how to efficiently compute runs, that is, maximal periodicities occurring in a word.

Keywords: Lyndon Tree, Cartesian tree, runs, word

1. Introduction

The Cartesian tree introduced by Vuillemin [15] is a binary tree associated with a sequence of numbers that label its nodes. It is both a heap, with the smallest element at the root, and the sequence is recovered during a symmetric traversal of the tree.

Cartesian trees have a series of applications in addition to that introduced by Vuillemin [15] on two-dimensional images. To quote a few of them, they are used for range searching to implement range minimum queries in a sequence of numbers through the help of Lowest Common Ancestor queries in the Cartesian tree of the sequence [8]. They are also part of sorting methods that want to take advantage of partially sorted subsequences (see for example [12]).

*Corresponding author

Email addresses: Maxime.Crochemore@kcl.ac.uk (Maxime Crochemore), luis.russo@tecnico.ulisboa.pt (Luís M. S. Russo)

Lyndon trees are associated with Lyndon words, words that are lexicographically smaller than all their proper non-empty suffixes (see [13] and [2]). They also have several interesting algorithmic applications and attracted much interest in connection with the detection of runs (maximal periodicities) in words. The notion of Lyndon roots of runs, introduced for cubic runs in [5], has led to the property that there is linear number of square runs in a word. Originally conjectured by Kolpakov and Kucherov [11], it has eventually been proved by Bannai et al. [1]. They also show how to compute efficiently all the runs using implicitly the notion of Lyndon table (array), which is a side product of the Lyndon tree construction.

This article may be viewed as a follow-up of the publication by Hohlweg and Reutenauer [10] in which they show the link between the two types of trees. The bridge between them is a key property (stated in Proposition 1) that relates a local condition on the factors of the word to a global condition on its suffixes. It implies the structure of a Lyndon tree is the same as that of the Cartesian tree of the associated word suffixes ranks.

2. Cartesian tree

Let $x = (x[0], x[1], \ldots, x[n-1])$ be a sequence of numbers of length n. The Cartesian tree of x is an ordered binary tree with the following properties:

- The tree contains exactly one node for each number in x. We identify these nodes with the positions of numbers in the sequence, i.e., node i will contain the value $x[i]$, this value is labelled $X[i]$.
- An in-order traversal of the tree results in sequence x. Therefore the left subtree contains the values that occur before the root value, while the right subtree contains that values that occur after the root value. The same property holds for the remaining nodes.
- The min-heap property holds, i.e., for any node S, except the root, we have that $X[S.\texttt{Parent}] \leq X[S]$.

If the values in x are pairwise distinct then the Cartesian tree is unique. Figure 1 shows the Cartesian tree for a sample sequence x.

Let us now study a standard algorithm for computing Cartesian trees. To simplify the algorithm we insert a sentinel into the original sequence, i.e., we

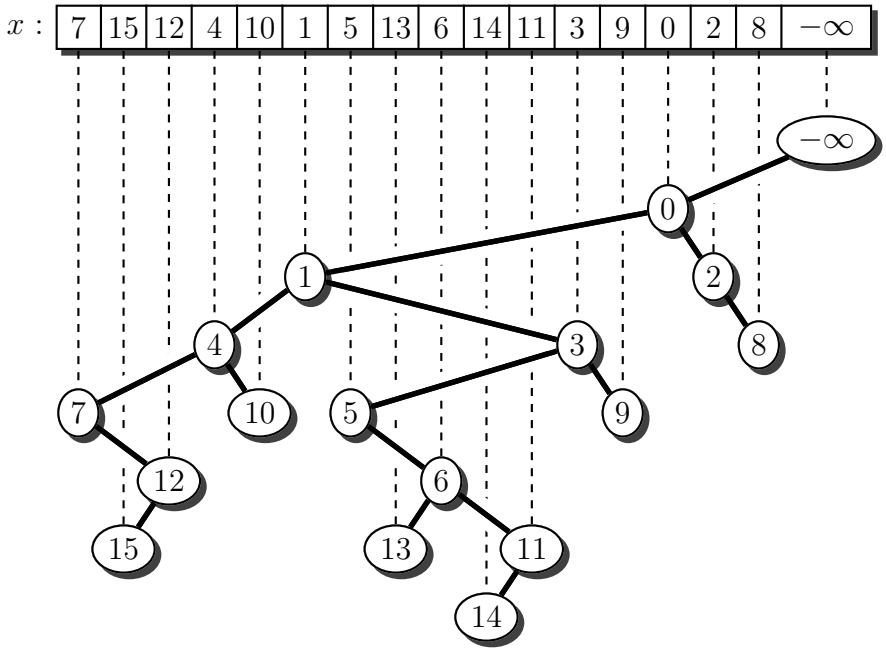

Figure 1: Sample Cartesian tree.

add a number $x[n] = -\infty$. This number is smaller than any other number that already exists in the sequence.

The algorithm proceeds from right to left, instead of left to right as usual, to fit with the Lyndon tree construction. One step i is to go up the leftmost path of the tree from $i + 1$ to find where to insert the node i. The artificial node n acts as a sentinel to simplify the design. During the traversal, going to the parent of node S is like going to the next nearest value smaller than $x[i]$. See Algorithm 1.

Figure 2 exemplifies Algorithm CartesianTree by inserting the number 5 into the tree. At each step the algorithm considers the leftmost branch of the tree, highlighted by the arrows in the picture. These arrows represent the `Parent` pointers that are used by the inner **while** cycle. In this example the **while** guard is true twice, for $5 < 13$ and $5 < 6$. The final comparison yields $5 > 3$ and therefore the cycle stops. Notice that the `Parent` pointers of 13 and 6 are not represented by arrows in the second tree, as they are no longer part of the leftmost branch.

Algorithm 1 Build a Cartesian tree.

CartesianTree(x)

```
 1  ▷ x is a non-empty sequence of n numbers.
 2  X[n] ← −∞
 3  n.LeftChild ← Null
 4  for i ← n − 1 downto 0 do
 5          S ← i + 1
 6          while x[i] < X[S] do
 7                  S ← S.Parent
 8          i.RightChild ← S.LeftChild
 9          S.LeftChild ← i
10  return labelled built tree
```

The number of comparisons executed at line 6 is linear in n. Indeed, any comparison that yields $x[i] \geq X[S]$ means that the `while` test fails and therefore occurs at most once for each i. Moreover the comparisons that yield $x[i] < X[S]$ for some position j, i.e., $x[j] = X[S]$ implies that position j will no longer be involved in a latter comparison. An alternative view of this process is that the assignment i.`RightChild` $\leftarrow$ S.`LeftChild` adds an element to the rightmost branch of the tree and that the assignments $S \leftarrow S$.`Parent` move upward on the rightmost branch of the current tree. Thus, the running time amortizes to $O(n)$. More precisely the length of this rightmost branch can be used as the value of a potential function Φ, which yields the amortized cost of 2 for the assignment of line 8 and the amortized cost of 1 for the **while** cycle of line 6.

This upwards process computed by the inner cycle, corresponds to finding the Next nearest smaller value. In our example, when given the number 5 we searched the sequence until we reached the number 3. Note that moving upwards on the tree is faster than computing linear scan from left to right, in particular we did not compare with the numbers 14 and 11.

Next nearest smaller table. The Next nearest smaller table NNS of a (non-empty) sequence y of numbers is defined as follows. For a position i on x, $i = 0, \ldots, |x| - 1$, NNS[i] is the smallest position $j > i$ of an element $x[j] < x[i]$.

$$\mathrm{NNS}[i] = \min\{j \mid i < j \leq n \text{ and } x[j] < x[i]\}.$$

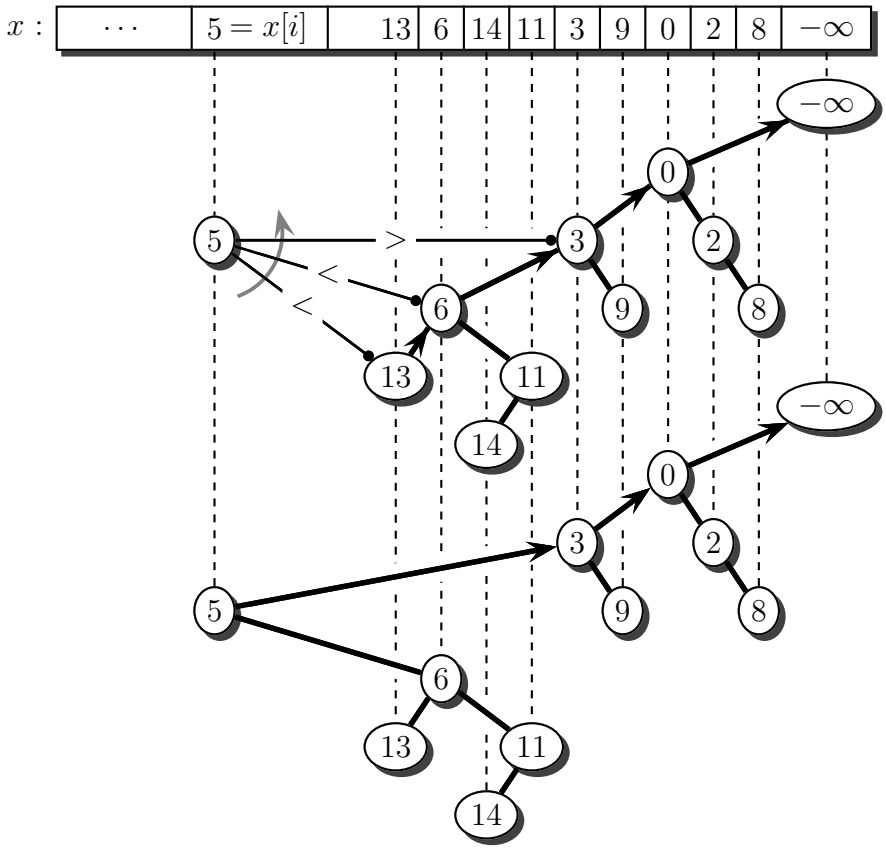

Figure 2: Illustration of the Cartesian tree construction algorithm.

Due to our assumption that $x[n] = -\infty$, the value NNS$[i]$ is n if no other value $x[j]$ is smaller than $x[i]$.

Figure 3 shows the NNS table illustrated over the Cartesian tree. We show the corresponding table below the tree. Moreover each node also shows an arrow that points to the corresponding Next nearest smaller node. When a node is a left child of its parent the arrows are simply the `parent` pointers. However when the node is a right child of its parent then the arrows are shown with dashed lines and point to an ancestor of the node that is to its right.

It is interesting to notice that the algorithm used for constructing Cartesian trees can be adapted to compute the NNS values. As illustrated by the picture when a node is a `LeftChild` then the NNS value is actually a pointer

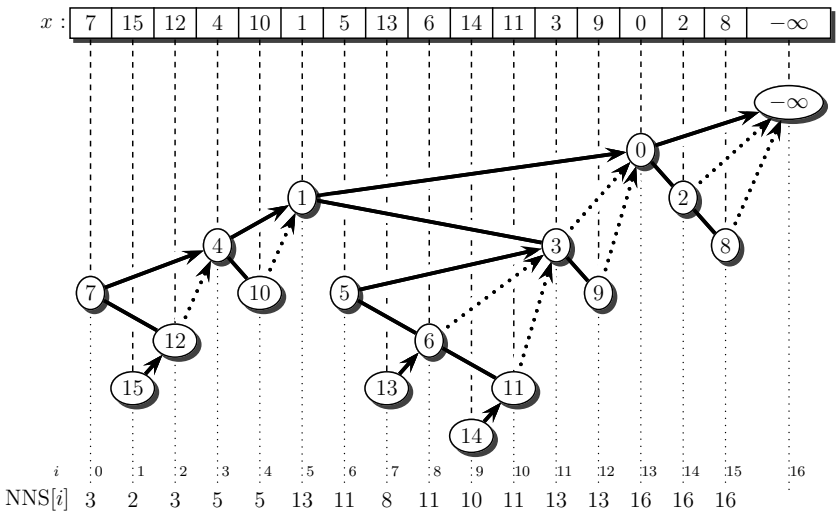

Figure 3: Illustration of the NNS table.

to its `Parent` on the tree. Now recall Algorithm CartesianTree and notice that whenever a value is inserted in the tree it is always a `LeftChild` and therefore its `Parent`, when it gets inserted, is the corresponding NNS value. Recall our example when the value of $x[6] = 5$ is inserted into the tree it becomes the `LeftChild` of node 11 (with $x[11] = 3$) therefore NNS[6] = 11. Note also in this example that when $x[6]$ is processed we have that the node 8 (with $x[8] = 6$) was a `LeftChild` of node 11 (with $x[11] = 3$) before the insertion but becomes a `RightChild` after the insertion. Still the value NNS[8] = 11 is not altered by this procedure. Algorithm 2 is a modification of Algorithm CartesianTree that uses this information to obtain the NNS values. Likewise it also runs in linear time.

3. Lyndon tree

Lyndon trees are associated with Lyndon words. Recall that a Lyndon word is a non-empty word lexicographically smaller than all its proper non-empty suffixes. The Lyndon tree of a Lyndon word y corresponds recursively to the following Lyndon (or standard) factorisation of y. If y is not a single letter, y can be written uv where v is chosen as the smallest proper non-empty suffix of y. The word u is then also a Lyndon word (see [13]).

Figure 4 shows the Lyndon tree of the word `#abbabaababbabaab`.

Algorithm 2 Compute NNS table.

NextNearestSmaller(x non-empty sequence of numbers of length n)

```
1  (x[n], NNS[n − 1]) ← (−∞, n)
2  for i ← n − 2 downto 0 do
3          j ← i + 1
4          while x[i] < x[j] do
5                  j ← NNS[j]
6          NNS[i] ← j
7  return NNS
```

Algorithm LyndonTree builds the Lyndon tree of a Lyndon word y. The hypothesis that y is a Lyndon word is not a significant restriction because any word can be turned into a Lyndon word by prepending to it a letter smaller than all letters occurring in it. We use the symbol # for this purpose in our examples. Otherwise, since any word factorises uniquely into Lyndon words, the algorithm produces the forest of Lyndon trees of the factors. We show the psedo-code in Algorithm 3.

The algorithm proceeds naturally from right to left on y to find the longest Lyndon word starting at each position i. It applies a known property: if u and v are Lyndon words and $u < v$ then uv is also a Lyndon word and satisfies $u < uv < v$.

To facilitate the presentation, variable u stores a phrase, that is, the occurrence of a Lyndon factor of y though the position of the factor is not explicitly given, and $T(u)$ is the Lyndon tree associated with this occurrence. Idem for v.

Figure 5 illustrates this processes by inserting the letter $y[i] = a$. The top of the figure shows the forest on Lyndon trees that exists before the letter is processed and the bottom shows the forest after $y[i]$ is inserted. The comparisons that are executed during this process are also shown. Note that these comparisons, in line 5, are actually lexicographical comparisons of strings, and in straightforward implementation require more than constant time.

If the comparison $u < v$ at line 5 is done by mere letter comparisons, the algorithm may run in quadratic time, for example if applied on $y = \mathtt{a}^k\mathtt{b}\mathtt{a}^k\mathtt{c}$. Each factor $\mathtt{a}^i\mathtt{b}$ is compared with the prefix $\mathtt{a}^{i+1}$ of $\mathtt{a}^k\mathtt{c}$ or with $\mathtt{a}^k\mathtt{c}$ itself, Figure 6 illustrates this argument.

However the algorithm can be implemented to run in linear time if the

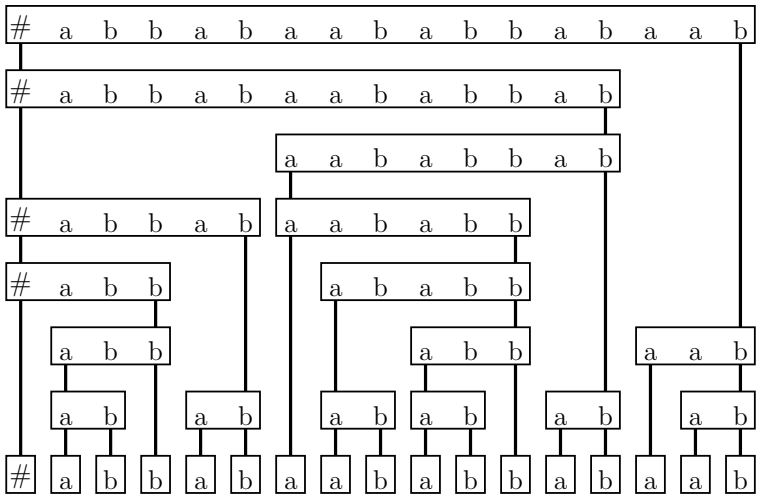

Figure 4: Lyndon tree.

test $u < v$ at line 5 is done in constant time because each execution of instructions at lines 6-8 decreases the number of Lyndon phrases, which goes from n to 1.

Lyndon table. The Lyndon table Lyn of a (non-empty) word y is defined as follows. For a position i on y, $i = 0, \ldots, |y| - 1$, Lyn$[i]$ is the length of the longest Lyndon factor of y starting at i:

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

Figure 7 shows an illustration of the Lyn table below the Lyndon tree, the arrows point to the nodes of the tree that contain the corresponding longest Lyndon factors. Moreover those nodes are also drawn with thicker lines. The line for i shows the indexing of the letters and of y, note that the symbol # is not indexed, it would correspond to index -1. For now ignore the values $i + \text{Lyn}[i]$, they will be explained in the next section.

The computation of the Lyndon table is an offspring of the previous algorithm, like the computation of the Next nearest smaller table for Algorithm CartesianTree. The procedure LongestLyndon of Algorithm 4 computes Lyn using the same right-to-left detection of Lyndon factors as above.

The Lyndon factorization of y can be obtained from the Lyn table. Note that in the previous algorithm we did not assume that the character # was

Algorithm 3 Build Lyndon tree.

```
LyndonTree(y)
1  ▷ y is a Lyndon word of length n
2  (v, T(v)) ← (y[n − 1], (y[n − 1]))
3  for i ← n − 2 downto 0 do
4          (u, T(u)) ← (y[i], (y[i]))
5          while u < v do
6                  T(uv) ← (new node, T(u), T(v))
7                  u ← uv
8                  v ← next phrase, empty word if none
9  return T(y)
```

Algorithm 4 Lyn table computation.

```
LongestLyndon(y non-empty word of length n)
1  for i ← n − 1 downto 0 do
2          (Lyn[i], j) ← (1, i + 1)
3          while j < n and y[i . . j − 1] < y[j . . j + Lyn[j] − 1] do
4                  (Lyn[i], j) ← (Lyn[i] + Lyn[j], j + Lyn[j])
5  return Lyn
```

part of the string, otherwise the factorization would be trivial, because y was a Lyndon word. After the algorithm finishes its execution, with $i = 0$, consider a final execution to insert the character #, with $i = -1$. We are interested in the values of j during this process. Initially j is $i + 1 = 0$, then the values get updated as $j + \text{Lyn}[j]$, these values are shown in the last line of Figure 7. For the running example the values form the sequence $0, 3, 5, 13, 16$, this sequence highlighted with dashed lines and arrows between the bottom two lines. The resulting factorization is `abb·ab·aababbab·aab`. This process is illustrated in the figure using dashed nodes and dashed arrows.

4. Key property

It is clear that the previous algorithms all share the same algorithmic structure. The link between the trees or their reduced versions is even tighter when the running time of the Lyndon tree construction is concerned. Indeed, the comparison between two consecutive phrases of the factorisation of y at line 5 in LyndonTree or at line 3 in LongestLyndon comes back to

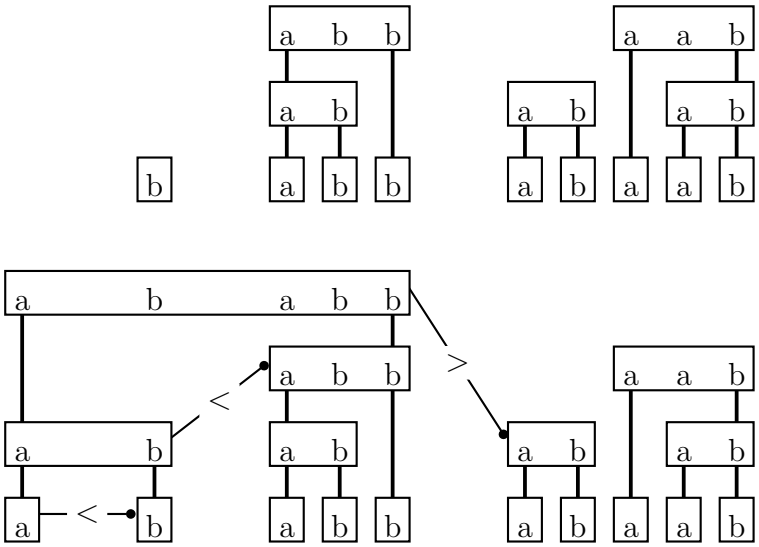

Figure 5: Illustration of comparisons for Lyndon tree construction.

considering the ranks of suffixes in alphabetic order. This is shown by the next proposition where the local comparison between two phrases is shown to be equivalent to the comparison of their associated suffixes.

In addition, the next statement also leads to a proof that the Lyndon tree of y, possibly reduced to its internal nodes, has the same structure of the Cartesian tree built from the ranks of the word suffixes, which was first noticed by Hohlweg and Reutenauer in [10].

Proposition 1. *Let u be a Lyndon word and $v \cdot v_1 \cdot v_2 \cdots v_m$ be the Lyndon factorisation of a word w. Then $u < v$ iff $uw < w$.*

Proof. In the proof we use the notation $u \ll v$ to mean that u is lexicographically smaller than v, $u < v$, and that u is not a prefix of v.

Let us consider the different cases.

Assume first $u < v$. If $u \ll v$ then $uw \ll vv_1v_2 \cdots v_m = w$. Alternatively consider the case where u is a proper prefix of v. Let $e > 0$ be the largest integer for which $v = u^e z$. Since v is a Lyndon word, z is not empty and we have $u^e < z$. Since u is not a prefix of z (by definition of e) nor z a prefix of u (because v is border-free) we have $u \ll z$. This implies $u^{e+1} \ll u^e z = v$ and then $uw < w$.

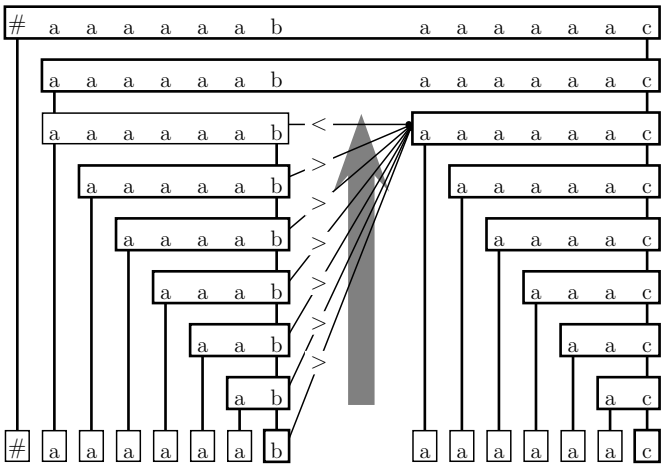

Figure 6: Quadratic time example.

Then assume $v \leq u$. If $v \ll u$ we have obviously $w < uw$. It remains to consider the situation where v is a prefix of u. If it is a proper prefix, u writes vz for a non-empty word z. We have $v < z$ because u is a Lyndon word. The word z cannot be a prefix of $t = v_1 v_2 \cdots v_m$ because v would not be the longest Lyndon prefix of w, a contradiction with a property of the factorisation. Thus, either $t \leq z$ or $z \ll t$. In the first case, if t is a prefix of z, $w = vt$ is a prefix of u and then of uw, that is, $w < uw$. In the second case, for some suffix z' of z and some factor v_k of t we have $z' \ll v_k$. The factorisation implies $v_k \leq v$. Therefore, the suffix z' of u is smaller than its prefix v, a contradiction with the fact that u is a Lyndon word. ■

The suffix array SA of a string y indicates the lexicographical order of all the suffixes of y, i.e., if $i < j$ then $y[\mathrm{SA}[i] \,..\, |y| - 1] < y[\mathrm{SA}[j] \,..\, |y| - 1]$. The Rank array is the inverse of the suffix array, thus $\mathrm{Rank}[i]$ is the rank of the suffix $x[i \,..\, |y| - 1]$ is the increasing alphabetic list of all non-empty suffixes of y (ranks run from 0 to $|y| - 1$).

Figure 8 illustrates this computation for the steps we considered throughout the paper, thus unifying the computation for both trees. The first two columns on the left are identical and contain the $\mathrm{Rank}[i]$ values. The third column contains the suffix array of the text, thus it is the inverse of the first column. The fourth column contains the indexes i. The values involved in the

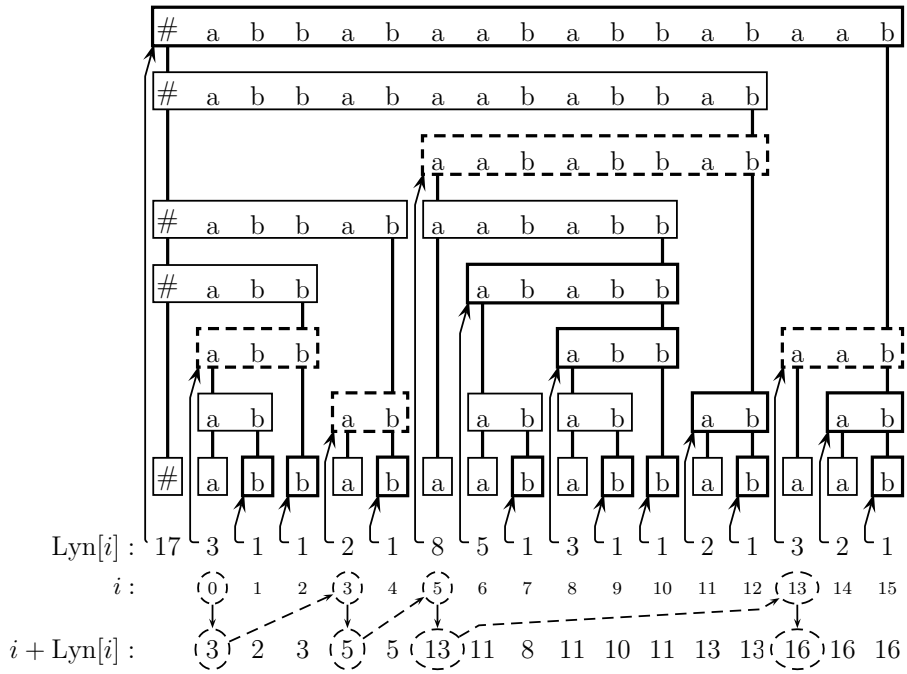

Figure 7: Lyn table.

computation are underlined. The suffixes, in lexicographical order are shown on the right, twice for illustration purposes. According to the key property, the operations on the Lyndon tree use the comparisons of suffixes on the right and the operations on the Cartesian tree use the corresponding integer comparisons of underlined numbers on the left. The comparisons on suffixes proceed bottom up, because the suffixes compared are lexicographically decreasing. On the other hand the index j on Rank is increasing and therefore the computation proceeds bottom-up in rank columns. These directions are indicated by large grey arrows.

Applying the above property to update line 3, Algorithm 5 uses a rewrite of the LongestLyndon procedure, where tables Lyn and Rank concern the input word y.

As for the running time, when the table Rank is precomputed, the comparison of words at line 3 is obtained in constant time. And since the number of comparisons is no more than $2|x| - 2$ (exactly $n - 1$ comparisons fail because $\text{Rank}[i] > \text{Rank}[j]$, which stop the while loop and no more than $n - 1$

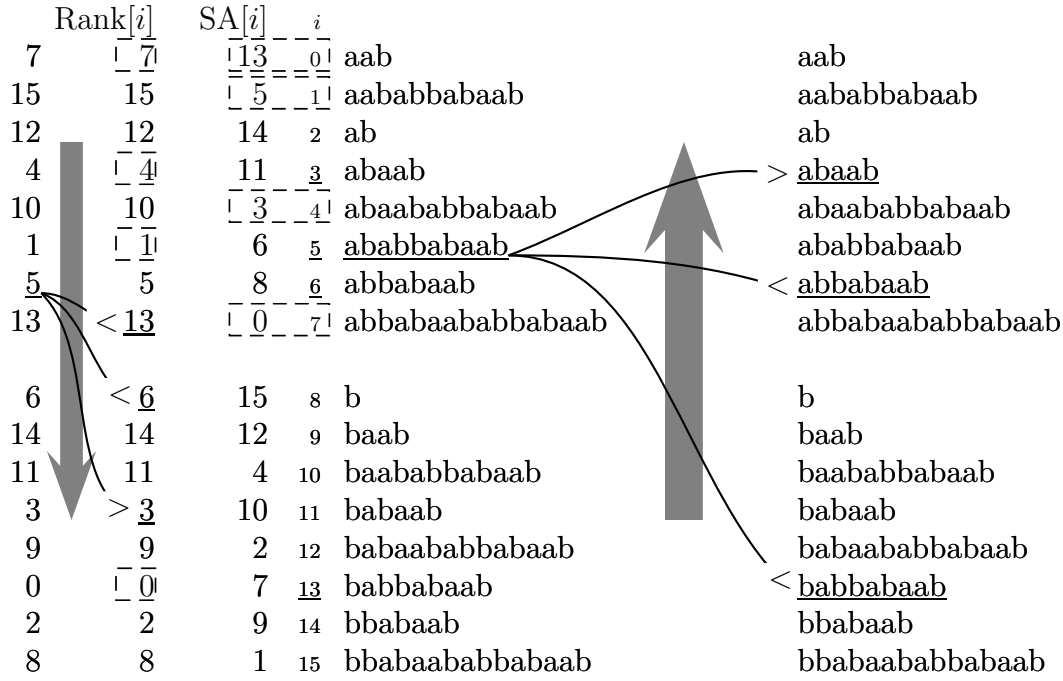

Figure 8: Rank and SA arrays in a sample computation.

comparisons succeed when $\text{Rank}[i] < \text{Rank}[j]$, since each reduces the number of Lyndon factors in the overall factorisation of y), the total running time is linear.

An immediate consequence of Proposition 1 is that the computation of the Lyndon factorisation of y can also be recovered from the left branch of the Cartesian tree. This is done by iterating NNS from position 0 on y. The next statement follows Corollary 3.1 by Holweg and Reutenauer in [10].

Corollary 2. *Associated with a word y, the sequence $(0, \text{NNS}[0], \text{NNS}^2[0], \ldots)$ (without the last value $|y|$) is the sequence of left-to-right starting positions of factors of the Lyndon factorisation of y.*

Corollary 3.1 by Holweg and Reutenauer in [10], establishes that the left-to-right starting positions of factors of the Lyndon factorisation of y can be obtained by tracking the left to right minima in SA. In Figure 8 this left-to-right process becomes top-to-bottom. Consider keeping track of the value of the minimum. Initially this value is 13, then, when $i = 1$, the value changes to 5. The value changes again to 3 at position $i = 4$ and to 0 at position

Algorithm 5 Efficient computation of the Lyn table.

LongestLyndon(y non-empty word of length n)

```
1  for i ← n − 1 downto 0 do
2          (Lyn[i], j) ← (1, i + 1)
3          while j < n and Rank[i] < Rank[j] do
4                  (Lyn[i], j) ← (Lyn[i] + Lyn[j], j + Lyn[j])
5  return Lyn
```

$i = 7$, which is the overall minimum. The resulting sequence of minima is $(13, 5, 3, 0)$, which corresponds to the Lyndon factor positions in reverse, for our running example. The sequence of positions $(0, 3, 5, 13)$ corresponds to the rank values $7, 4, 1, 0$. These values are highlighted with dashed boxes in Figure 8. Note the relation between Lyn and NNS: $\mathrm{NNS}[i] = i + \mathrm{Lyn}[i]$, since $\mathrm{Lyn}[i]$ is the smallest distance to a next rank value smaller than $\mathrm{Rank}[i]$.

5. Computing runs

Algorithm LongestLyndon extends to an algorithm for computing efficiently all runs occurring in a word.

Recall that a run in the word y is a maximal (non-extensible) occurrence of a factor, say $y[i\,..\,j]$, whose length is at least twice its (smallest) period. The main result in [1] shows that a run can be identified with a special position s, $i < s \leq j$, for which both $\mathrm{Lyn}[s]$ is the period of $y[i\,..\,j]$ and $2 \times \mathrm{Lyn}[s] \leq j - i + 1$, considering some alphabet ordering or its inverse for the table Lyn.

To compute all runs of the word y, we just have to check if the longest Lyndon factor starting at i produces a special position of a run. This is done by extending the Lyndon factor to the left and to the right according to the period of the resulting factor and using longest common extensions. This is done by computing $r = \mathrm{LCE_R}(i, i + \mathrm{Lyn}[i])$ and $\ell = \mathrm{LCE_L}(i - 1, i + \mathrm{Lyn}[i] - 1)$ when appropriate and verifying if $\ell + r \geq \mathrm{Lyn}[i]$. If the inequality holds a run can be reported.

The longest common extension to the right of two suffixes starting at positions i and j, $\mathrm{LCE_R}(i, j)$, is the size of the largest common prefix among them. In our example, for $i = 8$ and $j = 8 + \mathrm{Lyn}[8] = 8 + 3 = 11$, the corresponding suffixes are `abbabaab` and `abaab`. The resulting largest common prefix is `ab`, therefore $r = 2$. Likewise the longest common extension to the

left of two prefixes i and j, $\mathrm{LCE_L}(i,j)$, is the size of the largest common suffix among their associated prefixes. Figure 9 illustrates this process, where the run, occurrence of $\mathtt{bab}\cdot\mathtt{bab}$ is identified by starting with the Lyndon factor $\mathtt{abb}$ and obtaining $r=2$ and $\ell=1$.

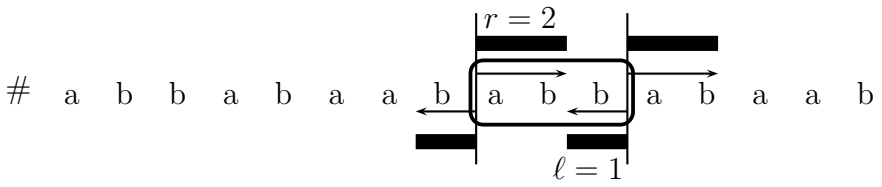

Figure 9: LCE illustration.

In Algorithm 6 we assume ℓ to be set to null if $i=0$ and r to null also if $i+\mathrm{Lyn}[i]=n$.

Algorithm 6 Computation of runs.

RUNS(y non-empty word of length n)

```
1  for i ← n − 1 downto 0 do
2        (Lyn[i], j) ← (1, i + 1)
3        while j < n and Rank[i] < Rank[j] do
4              (Lyn[i], j) ← (Lyn[i] + Lyn[j], j + Lyn[j])
5        (ℓ, r) ← (LCE_L(i − 1, i + Lyn[i] − 1), LCE_R(i, i + Lyn[i]))
6        if ℓ + r ≥ Lyn[i] then
7              output run x[i − ℓ .. i + Lyn[i] + r − 1]
```

To locate all runs, procedure RUNS has to be executed twice, for the tables corresponding to some alphabet ordering and for the tables corresponding to the inverse alphabet ordering.

Running time of RUNS. Procedure RUNS can be implemented to run in linear time $O(|y|)$ when the alphabet is linearly-sortable.

Indeed, with the hypothesis, it is known that suffixes of y can be sorted in linear time (see for example [3]). Then also the table Rank that is just the inverse of the sorted list of starting positions of the suffixes.

Again with the hypothesis, LCE queries at line 5 can be executed in constant time after a linear-time preprocessing. The reader can refer to the

review by Fischer and Heun [6] concerning LCE queries. More advanced techniques to implement them over a general alphabet and to compute runs can be found in [9, 4] and references therein.

Therefore the whole algorithm RUNS runs in linear time when the alphabet is linearly-sortable.

6. Concluding remarks

In this article we detailed the relation between the Lyndon tree of a word and the Cartesian tree of the corresponding Rank array of its suffixes. Thus pointing out a simple process to compute Lyndon trees in linear time, for linearly-sortable alphabets. We also explained how to extend this computation to locate all runs in the same time.

The relation between suffix sorting, part of the suffix array, and Lyndon factorisation is examined by Mantaci, Restivo, Rosone and Sciortino in [14]. Franek, Islam, Rahman and Smyth present several algorithms to compute the Lyndon table [7].

The structure of the Cartesian tree with its nodes labelled by numbers is richer than the structure of the Lyndon tree because it seems difficult to recover the labels without completely sorting the ranks of suffixes. This question is certainly related to the application of Cartesian trees to sorting (see for example [12]).

Acknowledgements

This work was funded in part by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

[1] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. The "runs" theorem. *CoRR*, abs/1406.0263v7, 2015.

[2] J. Berstel, A. Lauve, C. Reutenauer, and F. Saliola. *Combinatorics on Words: Christoffel Words and Repetition in Words*, volume 27 of *CRM Monograph Series*. Université de Montréal et American Mathematical Society, Dec. 2008.

[3] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. 392 pages.

[4] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, R. Kundu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen. Near-optimal computation of runs over general alphabet via non-crossing LCE queries. In S. Inenaga, K. Sadakane, and T. Sakai, editors, *String Processing and Information Retrieval - 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings*, volume 9954 of *Lecture Notes in Computer Science*, pages 22–34, 2016.

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

[6] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In M. Lewenstein and G. Valiente, editors, *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings*, volume 4009 of *Lecture Notes in Computer Science*, pages 36–48. Springer, 2006.

[7] F. Franek, A. S. M. S. Islam, M. S. Rahman, and W. F. Smyth. Algorithms to compute the lyndon array. *CoRR*, abs/1605.08935, 2016.

[8] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In R. A. DeMillo, editor, *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA*, pages 135–143. ACM, 1984.

[9] P. Gawrychowski, T. Kociumaka, W. Rytter, and T. Walen. Faster longest common extension queries in strings over general alphabets. In R. Grossi and M. Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 5:1–5:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

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

[11] R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer*

Science, FOCS '99, 17-18 October, 1999, New York, NY, USA, pages 596–604. IEEE Computer Society, 1999.

[12] C. Levcopoulos and O. Petersson. Heapsort - adapted for presorted files. In F. K. H. A. Dehne, J. Sack, and N. Santoro, editors, *Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17-19, 1989, Proceedings*, volume 382 of *Lecture Notes in Computer Science*, pages 499–509. Springer, 1989.

[13] M. Lothaire. *Combinatorics on Words*. Addison-Wesley, Reading, Mass., 1983.

[14] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. Suffix array and lyndon factorization of a text. *J. Discrete Algorithms*, 28:2–8, 2014.

[15] J. Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.