Back-to-Front Online Lyndon Forest Construction

Golnaz Badkobeh
Goldsmiths University of London, United Kingdom

Maxime Crochemore
Université Gustave Eiffel, France and King's College London, United Kingdom

Jonas Ellert
Technical University of Dortmund, Germany

Cyril Nicaud
Université Gustave Eiffel, France

Abstract

A Lyndon word is a word that is lexicographically smaller than all of its non-trivial rotations (e.g. `ananas` is a Lyndon word; `banana` is not a Lyndon word due to its smaller rotation `abanan`). The Lyndon forest (or equivalently Lyndon table) identifies maximal Lyndon factors of a word, and is of great combinatoric interest, e.g. when finding maximal repetitions in words. While optimal linear time algorithms for computing the Lyndon forest are known, none of them work in an online manner. We present algorithms that compute the Lyndon forest of a word in a reverse online manner, processing the input word from back to front. We assume a general ordered alphabet, i.e. the only elementary operations on symbols are comparisons of the form less-equal-greater. We start with a naive algorithm and show that, despite its quadratic worst-case behavior, it already takes expected linear time on words drawn uniformly at random. We then introduce a much more sophisticated algorithm that takes linear time in the worst-case. It borrows some ideas from the offline algorithm by Bille et al. (ICALP 2020), combined with new techniques that are necessary for the reverse online setting. While the back-to-front approach for this computation is rather natural (see Franek and Liut, PSC 2019), the steps required to achieve linear time are surprisingly intricate. We envision that our algorithm will be useful for the online computation of maximal repetitions in words.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Mathematics of computing → Combinatorics on words; Mathematics of computing → Combinatorial algorithms

Keywords and phrases Lyndon factorisation, Lyndon forest, Lyndon table, Lyndon array, right Lyndon tree, Cartesian tree, standard factorisation, online algorithms

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Supplementary Material https://github.com/jonas-ellert/right-lyndon

1 Lyndon words

A Lyndon word is a word that is lexicographically smaller than all of its non-trivial rotations (e.g. `ananas` is a Lyndon word; `banana` is not a Lyndon word due to its smaller rotation `abanan`). The Lyndon table or equivalently the right Lyndon forest of a word (generalized from the right Lyndon tree of a Lyndon word) identifies the longest Lyndon prefix of each suffix of the word (a precise definition follows later). The article explores the complexity of algorithms for building the Lyndon table or forest of a word over a general ordered alphabet. The only elementary operations on letters of the alphabet are comparisons of the form less-equal-greater. The presented algorithms process the input word $y[0\mathinner{.\,.}n-1]$ in a reverse online manner. When accessing position $y[i]$ for the first time, they have already computed the Lyndon table of $y[i+1\mathinner{.\,.}n-1]$.

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Background and applications. Introduced in the field of combinatorics on words by Lyndon (see [26, 24]) and used in algebra, Lyndon words introduce structural elements in plain sequences of symbols, provided there is an ordering on the set of symbols. They have shown their usefulness for designing efficient algorithms on words. For example, they underpinned the notion of critical positions in words [24], the two-way string matching [14] and rotations of periodic words [8].

The right Lyndon tree of a Lyndon word y (based on the factorisation $y = uv$, where v is the lexicographically smallest, or equivalently longest, proper Lyndon suffix of y) is by definition related to the sorted list of suffixes of y. Hohlweg and Reutenauer [21] showed that the right Lyndon tree is the Cartesian tree (see [30]) built from ranks of suffixes in their lexicographically sorted list (see also [15]). The list corresponds to the standard permutation of suffixes of the word and is the main component of its suffix array (see [27]), one of the major data structures for text indexing. The relation between suffix arrays and properties of Lyndon words is used by Mantaci et al. [28], by Baier [2] and by Bertram et al. [6] to compute the suffix array, as well as by Louza et al. [25] to induce the Lyndon table.

If the suffix array is given, the Lyndon table can be constructed in linear time (e.g. [19, Algorithm NSVISA]). In fact, the method is similar to the standard algorithms that build a Cartesian tree from the ranks of suffixes in their lexicographical order. However, computing the suffix array on general ordered alphabets requires $\Omega(n \lg n)$ time due to the well-known information-theoretic lower bound on comparison sorting. A linear-time algorithm that directly computes the Lyndon table (on general ordered alphabets, without requiring the suffix array) was designed by Bille et al. [7]. While it processes the word from left to right, it is not an online algorithm because it may need to look ahead arbitrarily far in the word in order to determine an entry of the Lyndon table.

The Lyndon forest is closely related to runs (maximal periodicities) in words, and is not only essential for showing theoretical properties of runs, but also for their efficient computation. Bannai et al. [3] used the Lyndon table to solve the conjecture of Kolpakov and Kucherov [22] stating that there are less than n runs in a length-n word, following a result in [12]. The key result in [3] is that every run in a word y contains as a factor a Lyndon root, according either to the alphabet ordering or its inverse, that corresponds to a node of the associated Lyndon forest. Since the Lyndon forest has a linear number of nodes according to the length of y, browsing all its nodes leads to a linear-time algorithm to report all the runs occurring in y. However, the time complexity of this technique also depends on the time required to build the forest and to extend a potential run root to an actual run. This is feasible on a linearly-sortable alphabet using an efficient longest common extension (LCE) technique (e.g. [18]). Kosolobov [23] conjectured a linear-time algorithm computing all runs for a word over a general ordered alphabet. An almost linear-time solution was given in [11], but the final positive answer is by Ellert and Fischer [17] who combined the Lyndon table algorithm by Bille et al. [7] with a new linear-time computation technique for the LCEs.

Our contributions. We present algorithms that compute the Lyndon table or Lyndon forest of a word over a general ordered alphabet. They scan the input word $y[0 \mathinner{..} n-1]$ in a reverse online manner, i.e. from the end to the beginning. When accessing $y[i]$ for the first time, they have already computed the Lyndon forest or table of the word $y[i+1 \mathinner{..} n-1]$. Processing the word in a reverse manner is rather natural (see Franek and Liut [20]), but it requires intricate algorithmic techniques to obtain an efficient algorithm.

In Section 2, we present a naive algorithm for computing the Lyndon table that takes quadratic time in the worst case. We also provide a simple linear time algorithm that computes the Lyndon forest from the Lyndon table. As shown in Section 3, the naive algorithm runs in expected linear time if we fix an alphabet and a length, and then choose a random word of the chosen length over the chosen alphabet. Finally, in Section 4, we introduce a more sophisticated algorithm for constructing the Lyndon table. It takes optimal linear time in the worst case, and uses the following techniques. First, the use of both next and previous smaller suffix tables, and skipping symbol comparisons when computing longest common extensions (LCEs) associated with these tables (similarly to what has been done in [7]). Second, the additional acceleration of the LCE computation by exploiting and reusing previously computed values. Third, additional steps to ensure that we reuse each computed value at most once, which ultimately results in the linear running time.

We envision that our algorithm will be useful as a tool for the online computation of runs. For example, it could lead to an online version of the runs algorithm presented in [17].

Remark. The design of the right Lyndon tree construction contrasts with the dual question of left Lyndon tree construction (see [1] and references therein). The latter is done by a far simpler algorithm than the algorithm in Section 4. But its use to build a right Lyndon tree, as done in the Lyndon bracketing by Sawada and Ryskey [29] and in [19] by Franek et al., does not seem to lead to a linear right Lyndon tree construction.

Basic definitions

Let A be an alphabet with an ordering $<$ and A^+ be the set of non-empty words with the lexicographical ordering induced by $<$. The length of a word y is denoted by $|y|$. The empty word of length 0 is denoted by ϵ. The concatenation of two words u and v is denoted by uv. The e times concatenation of a word u is written as u^e (e.g. $u^3 = uuu$). If for non-empty words u, v, y it holds $y = uv$, then we say that uv (formally (u, v)) is a non-trivial factorisation of y; the word vu is a non-trivial rotation (or conjugate) of y; the word u is a proper non-empty prefix of y; and word v is a proper non-empty suffix of y. A word y is primitive if it has $|y| - 1$ distinct non-trivial rotations, or equivalently if it cannot be written as $y = u^e$ for some word u and integer $e \geq 2$. If there are non-empty u and v such that $y = uv = vu$, then y is non-primitive. A word u is strongly less than a word v, denoted by $u \ll v$, if there are words r, s and t, and letters a and b satisfying $u = ras$, $v = rbt$ and $a < b$. The word u is smaller than a word v, denoted by $u < v$, if either $u \ll v$ or u is a proper prefix of v, i.e. $v = ur$ for some non-empty word r. A Lyndon word is a non-empty word defined as follows:

▶ **Proposition 1** ([16, Proposition 1.2]). *Both of the following equivalent conditions define a Lyndon word y: (i) $y < vu$, for every non-trivial factorisation uv of y, (ii) $y < v$, for every proper non-empty suffix v of y.*

Note that the conditions in the definition trivially hold if $|y| = 1$, i.e. a single symbol is always a Lyndon word. The main feature of Lyndon words stands in the theorem by Chen, Fox and Lyndon (see [24]), which states that every word in A^+ can be uniquely factorised into Lyndon words.

▶ **Theorem 2** (Lyndon factorisation). *Any non-empty word y may be written uniquely as a lexicographically weakly decreasing product of Lyndon words, i.e. $y = x_1x_2 \cdots x_m$, where each x_k is a Lyndon word and $x_1 \geq x_2 \geq \cdots \geq x_m$.*

A fundamental property of the Lyndon factorisation is the fact that suffix x_m of y is both the longest and the lexicographically smallest Lyndon suffix of y.

2 Lyndon tree and Lyndon table construction

The structure of the Lyndon tree of a Lyndon word derives from the next property (see [24]).

▶ **Property 3.** *Let y be a Lyndon word and $y = uv$, where v is the smallest or equivalently the longest proper Lyndon suffix of y. Then u is a Lyndon word.*

The (right) standard factorisation of a Lyndon word y of length $n > 1$ is the pair (u, v) of Lyndon words, simply denoted by $u \cdot v$, where u and v are as in Property 3.

The (right) Lyndon tree $\mathcal{R}(y)$ of a Lyndon word y represents recursively its complete (right) standard factorisation. It is a binary tree with $2|y| - 1$ nodes: its leaves are positions on the word and internal nodes correspond to concatenations of two consecutive Lyndon factors of the word, which as such can be viewed as inter-positions. More precisely, $\mathcal{R}(y) = \langle 0 \rangle$ if $|y| = 1$, and otherwise $\mathcal{R}(y) = \langle \mathcal{R}(u), \mathcal{R}(v) \rangle$, where $u \cdot v$ is the standard factorisation of y. The next algorithm gives a straightforward construction of a right Lyndon tree.

LyndonTree(y Lyndon word of length n)

1 $\mathcal{F} \leftarrow$ stack containing only the empty word ϵ
2 **for** $i \leftarrow n - 1$ **downto** 0 **do**
3     $(u, \mathcal{R}(u), v) \leftarrow (y[i], \langle i \rangle, \mathcal{F}.peek())$
4   **while** $u < v$ **do**
5         $\mathcal{R}(uv) \leftarrow \langle \mathcal{R}(u), \mathcal{R}(v) \rangle$
6         $(u, v) \leftarrow (uv, \mathcal{F}.\textit{pop-and-then-peek}())$
7     $\mathcal{F}.push(u)$
8 **return** $\mathcal{R}(y)$

Algorithm LyndonTree scans the word from right to left. Just after executing an iteration of the for loop, the suffix $y[i \mathinner{..} n-1]$ of y is decomposed into its Lyndon factorisation $z_1 \cdot z_2 \cdots z_m$. In this moment, the stack contains exactly the elements $z_1, z_2, \ldots, z_m$ (z_1 being the topmost element), and variable u stands for the first factor z_1 of the decomposition.

With an appropriate implementation of the tree, the algorithm runs in linear time if the test $u < v$ at line 4 is done in constant time. However, in the worst case, the algorithm runs in quadratic time; this is when the test is performed by mere letter comparisons. For example, if $y = \mathtt{a}^k\mathtt{ca}^{k+1}\mathtt{b}$ then each factor $\mathtt{a}^\ell\mathtt{c}$ is compared with the prefix $\mathtt{a}^{\ell+1}$ of $\mathtt{a}^{k+1}\mathtt{b}$).

Lyndon forest and Lyndon table

If $x_1x_2 \ldots x_m$ is the Lyndon factorisation of the non-empty word y, then its (right) Lyndon forest is defined by the list $\mathcal{R}(x_1)$, $\mathcal{R}(x_2)$, $\ldots$, $\mathcal{R}(x_m)$, i.e. the list of Lyndon trees of the Lyndon factors (the Lyndon forest is a single tree if and only if y is a Lyndon word). One could compute the Lyndon forest by simply first computing the Lyndon factorisation, and then building the tree for each factor. A more elegant method computes the forest from the Lyndon table (sometimes called Lyndon array) of the word. It is denoted by *Lyn* (l in [3], $\mathcal{L}$ in [20] and λ in [19, 7]) and is defined, for each position i on y, by

$$\mathit{Lyn}[i] = \max\{|w| \mid w \text{ is a Lyndon prefix of } y[i \mathinner{..} n-1]\}.$$

An example is provided in Figure 1 (we will explain the labeling of forest nodes and the table *root* in a moment; for now, we focus on *Lyn*). The Lyndon factorisation of y deduces easily from *Lyn*. Indeed, if i is the starting position of a factor of the decomposition, the next factor

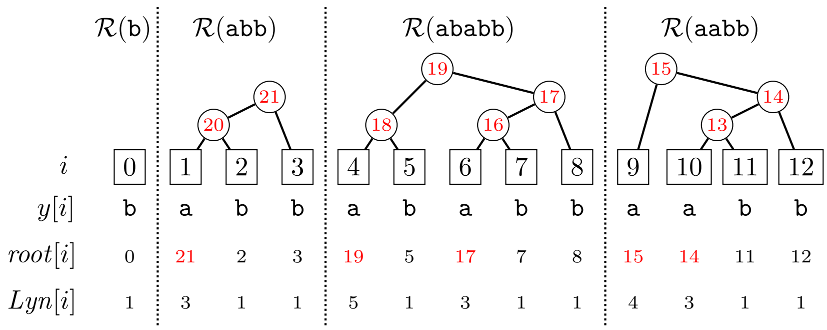


Figure 1 Lyndon forest and Lyndon table of the word $y =$ `babbababbaabb`. It holds $Lyn[4] = 5$ because `ababb` is the longest Lyndon factor starting at position 4.

starts at position $i + Lyn[i]$, which is the first position of the next smaller suffix of $y[i\,..\,n-1]$ (see Lemma 8). For the example above, we get positions 0, $(1 = 0 + Lyn[0])$, $(4 = 1 + Lyn[1])$ and $(9 = 4 - Lyn[4])$, corresponding to the Lyndon factorisation `b` · `abb` · `ababb` · `aabb` of y.

Algorithm LyndonTable shown below computes Lyn using the same scheme as Algorithm LyndonTree. It scans the input word from right to left and implicitly concatenates two adjacent Lyndon factors u and v to form a Lyndon factor uv when $u < v$.

```
LyndonTable(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
```

More details (including a proof of correctness) regarding this algorithm can be found in [13, Problem 87]). The worst-case running time is $O(|y|^2)$ in the letter-comparison model. Note, however, that the comparison of Lyndon words $y[i\,..\,j-1] < y[j\,..\,j + Lyn[j] - 1]$ in line 3 can be replaced by a suffix comparison $y[i\,..\,n-1] < y[j\,..\,n-1]$ (see [3, 15] and also [13, Problem 87]). Then, if the suffixes of y are previously sorted and their ranks are stored in a table, Algorithm LyndonTable runs in linear time. The precomputation takes linear time if y is drawn from a linearly-sortable alphabet (see suffix arrays in [10, Chapter 4]).

Obtaining the Lyndon forest from the table. A possible data structure that implements the Lyndon forest uses tables $root$, $left$ and $right$. Implicitly, the leaves are positions $0, \ldots, n-1$ on y. For each position i, $root[i]$ is the root of the largest subtree whose leftmost leaf is i. Thus, if x_k is a factor of the Lyndon factorisation of y that start at position i, then $root[i]$ is the root of the Lyndon tree of x_k. Internal nodes are integers larger than $n-1$. The left and right child of an internal node m are respectively $left[m]$ and $right[m]$. An example of this representation is provided in Figure 1.

As demonstrated by the example, $Lyn[i]$ is exactly the size of the subtree that is rooted in $root[i]$. This makes it easy to translate the Lyndon forest to the Lyndon table and vice versa. Algorithm LyndonForest below shows the conversion from Lyndon table to forest. Note that the time required by the algorithm, apart from the time needed to compute Lyn, is linear in n. This is due to the fact that each iteration of the inner loop creates a new internal node, and there are at most $n-1$ such nodes.

```
LyndonForest(y non-empty word of length n)
Lyn ← LyndonTable(y)
m ← n                              ▷ next available integer for a new internal node
for i ← n − 1 downto 0 do
    (root[i], j) ← (i, i + 1)
    while j < i + Lyn[i] do
        (left[m], right[m]) ← (root[i], root[j])
        (root[i], m) ← (m, m + 1)
        j ← j + Lyn[j]
return (root, left, right)
```

3 Average running time for building a Lyndon table

In this section, we prove that the average running time of Algorithm LyndonTable is linear, for the uniform distribution on size-n words, on any alphabet. The result also applies to Algorithm NaiveLyn of the next section and implies that the construction of a Lyndon forest can also be done in average linear time. We assume the alphabet $A = \{a_0, a_1, \dots a_{\sigma-1}\}$ of size $\sigma \geq 2$, equipped with the order $a_0 < a_1 < \dots < a_{\sigma-1}$. (Note that LyndonTable takes worst-case linear time for $\sigma = 1$, because then it holds $Lyn[i] = 1$ for all i, which means that the comparison in line 3 takes constant time).

For any positive n, let $\mathbb{P}_n$ be the uniform probability on A^n, where every word u has probability $\mathbb{P}_n(u) = \sigma^{-n}$. Formally, we are considering a sequence of uniform distributions, one for each size n; as we seek to obtain a result for every σ, we therefore have to consider that $\sigma = \sigma_n$ also depends on n (even if it can be the constant sequence and want a result for a fixed alphabet). In the following, we just require that $\sigma_n \geq 2$ for every n and write σ instead of σ_n for readability. Observe that allowing an alphabet of unbounded size is a completely different framework than in the series of articles [5, 9] dealing with the expected properties of uniform random Lyndon words, where the alphabet has a fixed size. In particular, for large σ_n, a uniform random word ressembles a uniform random permutation, whose typical statistics are greatly different from the ones of a uniform random word on, say, four letters.

Let i, j be two integers with $0 \leq i < j < n$. The random variable C_{ij} is defined as the number of comparisons performed between $y[i\,..\,j-1]$ and $y[j\,..\,j + Lyn[j] - 1]$ at line 3 of Algorithm LyndonTable, for a random word y: if the algorithm does not compare these two factors, then $C_{ij} = 0$, otherwise it is the number of letter comparisons performed by the naive algorithm that scans both words from left to right until it can decide. Since this is the only step of Algorithm LyndonTable where letter comparisons are performed, the total number of such comparisons performed by the algorithm is the random variable $C = \sum_{i<j} C_{ij}$. We also define $C_j = \sum_{i<j} C_{ij}$. In this section, we are interested in estimating the expectation $\mathbb{E}_n[C]$ of C for uniform random words of length n.

By linearity of the expectation, we have $\mathbb{E}_n[C] = \sum_j \mathbb{E}_n[C_j]$. Our proof consists in bounding from above the contribution of each $\mathbb{E}_n[C_j]$, using $\mathbb{E}_n[C_j] = \sum_{i<j} \mathbb{E}_n[C_{ij}]$. Let us first consider the cases where the position j is near the end of the word, as it has to be considered separately in our analysis and since it illustrates well the kind of techniques used for the main part of the proof.

▶ **Lemma 4.** *If $j \geq n - 3\log_2 n$ then $\mathbb{E}_n[C_j] = \mathcal{O}(\log^2 n)$.*

Proof. We make the following observations: (i) the factors starting at indices i and j can only be compared once by the algorithm, (ii) the factors compared at line 3 are always Lyndon words, and (iii) the number of letter comparisons performed for given i and j, if any, is at most $3\log_2 n$ since $j \geq n - 3\log_2 n$. So for every word y, the number of comparisons $C_{ij}(y)$ is bounded from above by $D_{ij}(y)$, where $D_{ij}(y)$ is 0 if $y[i\,..\,j-1]$ is not Lyndon and $3\log_2 n$ otherwise. Therefore $\mathbb{E}_n[C_{ij}] \leq \mathbb{E}_n[D_{ij}]$.

Let $\ell = j - i$ be the length of the factor $y[i\,..\,j-1]$. Since a non-primitive word cannot be a Lyndon word, if $\mathcal{L}$ and $\mathcal{P}$ respectively denote the set of Lyndon words and of primitive words, we have

$$\mathbb{P}_n(y[i\,..\,j-1] \in \mathcal{L}) = \mathbb{P}_\ell(\mathcal{L}) = \mathbb{P}_\ell(\mathcal{L} \cap \mathcal{P}) = \mathbb{P}_\ell(\mathcal{L} \mid \mathcal{P})\mathbb{P}_\ell(\mathcal{P}) \leq \mathbb{P}_\ell(\mathcal{L} \mid \mathcal{P}).$$

But $\mathbb{P}_\ell(\mathcal{L} \mid \mathcal{P}) = \frac{1}{\ell}$ as all rotations of a primitive word are equally likely, and only one of them is a Lyndon word. Hence $y[i\,..\,j-1] \in \mathcal{L}$ with probability at most $\frac{1}{\ell}$. This yields that $\mathbb{E}_n[C_{ij}] \leq \mathbb{E}_n[D_{ij}] \leq \frac{3\log_2 n}{\ell}$, and if we sum the contributions of all possible i, we have the announced result as $\mathbb{E}_n[C_j] \leq \sum_{i=0}^{j-1} \frac{3\log_2 n}{j-i} \leq 3\log_2 n \cdot (\log j + 1) = \mathcal{O}(\log^2 n)$. ◀

So when we sum the contributions of the $\mathbb{E}_n[C_j]$ for $j \geq n - 3\log_2 n$, the expected total number of comparisons is $\mathcal{O}(\log^3 n)$, hence sublinear. Thus we can focus on estimating $\mathbb{E}_n[C_j]$ for j sufficiently far away from the end of the word. We need the following lemma.

The proof is given in Appendix A.1

▶ **Lemma 5.** *Let Λ be an ordered alphabet with $|\Lambda| \geq 2$ letters, $\# \notin \Lambda$ be a new letter that is greater than every letter of Λ, and y be a word selected from Λ^t uniformly at random. Then there exists a constant $c \geq 1$ such that the probability that $y\#$ is a Lyndon word is at most $\frac{c}{t}$.*

Our main technical result is stated in the following proposition.

▶ **Proposition 6.** *There exists a constant γ such that $\mathbb{E}_n[C_j] \leq \gamma$, for all $j < n - 3\log_2 n$.*

Proof. Recall that if $E \subseteq A^n$, the indicator function of E is the random variable $\mathbb{1}_E$ that values 1 for elements of E and 0 otherwise. The first step of the proof is to look at the factor of a random word whose positions range from j to $j + \lfloor 3\log_2 n \rfloor - 1$. Let $\mathcal{A} \subseteq A^n$ denote the set of words y such that $y[j\,..\,j + \lfloor 3\log_2 n \rfloor - 1] = a_0^{\lfloor 3\log_2 n \rfloor}$, i.e. it consists of the repetition of the smallest letter. Let $\overline{\mathcal{A}}$ denote the complement of $\mathcal{A}$ in A^n. We have $C_j = \mathbb{1}_{\mathcal{A}} \cdot C_j + \mathbb{1}_{\overline{\mathcal{A}}} \cdot C_j$, hence $\mathbb{E}_n[C_j] = \mathbb{E}_n[\mathbb{1}_{\mathcal{A}} \cdot C_j] + \mathbb{E}_n[\mathbb{1}_{\overline{\mathcal{A}}} \cdot C_j]$.

Since for all word $y \in A^n$ we have $C_j(y) \leq C(y) \leq n^2$, it holds that $\mathbb{1}_{\mathcal{A}}(y) \cdot C_j(y) \leq \mathbb{1}_{\mathcal{A}}(y)\, n^2$ and therefore $\mathbb{E}_n[\mathbb{1}_{\mathcal{A}} \cdot C_j] \leq n^2\, \mathbb{E}_n[\mathbb{1}_{\mathcal{A}}] = n^2\, \mathbb{P}_n(\mathcal{A}) = \frac{n^2}{\sigma^{\lfloor 3\log_2 n \rfloor}}$. As $\sigma \geq 2$, this yields that $\mathbb{E}_n[\mathbb{1}_{\mathcal{A}} \cdot C_j] \leq \frac{2}{n}$.

We now focus on $\overline{\mathcal{A}}$. Let $\mathcal{B}_{k,s}^{(n)}$ denote the set of words y in A^n such that there exist a positive integer $k \leq \lfloor 3\log_2 n \rfloor$ and a letter $a_s \neq a_0$ such that $a_0^{k-1}a_s$ is a prefix of the suffix of y that starts at position j. Any word of $\overline{\mathcal{A}}$ has a factor of the form $a_0^{k-1}a_s$ starting at position j, so we have the following partition:

$$\overline{\mathcal{A}} = \bigsqcup_{k=1}^{\lfloor 3\log_2 n \rfloor} \bigsqcup_{s=1}^{\sigma-1} \mathcal{B}_{k,s}^{(n)},$$

and therefore $\mathbb{E}_n[\mathbb{1}_{\overline{\mathcal{A}}} C_j] = \sum_{k=1}^{\lfloor 3\log_2 n \rfloor} \sum_{s=1}^{\sigma-1} \mathbb{E}_n[\mathbb{1}_{\mathcal{B}_{k,s}^{(n)}} C_j]$.

Fix k and s. We want to bound from above the contribution of $\mathcal{B}_{k,s}^{(n)}$ to $\mathbb{E}_n[C_j]$ by summing the $\mathbb{E}_n[\mathbb{1}_{\mathcal{B}_{k,s}^{(n)}} C_{ij}]$ for $i < j$.

For a given index $i < j - k$ (the cases $j - k \leq i < j$ will be studied separately), when the algorithm works on an input $y \in \mathcal{B}_{k,s}^{(n)}$, if it compares $y[i\,..\,j-1]$ and $y[j\,..\,j + Lyn[j] - 1]$ then necessarily:

(i) The factor $y[i\,..\,j-1]$ is a Lyndon word.
(ii) There is no occurrence of $a_0^{k-1}a_t$ with $t < s$ in $y[i+1\,..\,j-1]$, otherwise the Lyndon factor starting at position j would have already merged, before reaching index i.

In our proof, we ask for weaker conditions, which is not an issue as we are looking for an upper bound for $\mathbb{E}_n[C_j]$. For this, we split the factor $y[i\,..\,j-1]$ of length $\ell = j-i$ into $\lambda = \lfloor \ell/k \rfloor$ blocks $y_0, \ldots, y_{\lambda-1}$ of k letters, and one remaining block z of length $\ell \mod k$ if ℓ is not a multiple of k:

$$\forall m \in \{0,\ldots,\lambda-1\} : y_m = y[i+mk\,..\,i+m(k+1)-1], \text{ so that } y[i\,..\,j-1] = y_0\cdots y_{\lambda-1}\cdot z.$$

The number λ of blocks is at least 1, as we only consider the indices i smaller than $j-k$ for now. Observe that, as y is a uniform random word, the y_m's are uniform and independent random words of length k. Condition (ii) implies that none of the y_m is smaller than $a_0^{k-1}a_s$ for $m \geq 1$. We distinguish two cases, depending on whether y_0 is smaller than $a_0^{k-1}a_s$ or not, and define, for $i < j-k$, the following sets

$$\mathcal{B}^{<}_{k,s,i} = \{x \in \mathcal{B}^{(n)}_{k,s} \mid y_0 < a_0^{k-1}a_s \text{ and } \forall m \in \{1,\ldots,\lambda-1\}, y_m \geq a_0^{k-1}a_s\},$$
$$\mathcal{B}^{\geq}_{k,s,i} = \{x \in \mathcal{B}^{(n)}_{k,s} \mid \forall m \in \{0,\ldots,\lambda-1\}, y_m \geq a_0^{k-1}a_s \text{ and } y[i\,..\,j-1] \in \mathcal{L}\}.$$

As a consequence of Condition (i) and Condition (ii), if $y \in \mathcal{B}^{(n)}_{k,s}$ and $C_{ij}(y) \neq 0$ (the algorithm compares the factors at positions i and j), then necessarily $y \in \mathcal{B}^{<}_{k,s,i}$ or $y \in \mathcal{B}^{\geq}_{k,s,i}$. Therefore $\mathbb{P}_n(C_{ij} \neq 0 \text{ and } \mathcal{B}^{(n)}_{k,s}) \leq \mathbb{P}_n(\mathcal{B}^{<}_{k,s,i}) + \mathbb{P}_n(\mathcal{B}^{\geq}_{k,s,i})$.

- The probability of $\mathcal{B}^{<}_{k,s,i}$ is $\mathbb{P}_n(\mathcal{B}^{<}_{k,s,i}) = \frac{1}{\sigma^k}\frac{s}{\sigma^k}\left(\frac{\sigma^k-s}{\sigma^k}\right)^{\lambda-1}$, as the probability to be in $\mathcal{B}^{(n)}_{k,s}$ is σ^{-k} and as there are s words of length k smaller than $a_0^{k-1}a_s$.
- To compute the probability of $\mathcal{B}^{\geq}_{k,s,i}$ observe the y_m's are uniform independent elements of $\Lambda_{k,s} = \{w \in A^k : w \geq a_0^{k-1}a_s\}$ and z is an independent and uniform word of length $\ell \mod k$ on A. Moreover, if $y[i\,..\,i+j-1]$ is a Lyndon word, then $y_m y_{m+1}\cdots y_{\lambda-1}z$ is greater than $y[i\,..\,i+j-1]$ for every $1 \leq m < \lambda$. Now consider $y_0\cdots y_{\lambda-1}$ as a size-λ word on the alphabet $\Lambda_{k,s}$; the latter property implies that $y_0\ldots y_{\lambda-1}\#$ is smaller than $y_m y_{m+1}\cdots y_{\lambda-1}\#$, where $\#$ is a new letter greater than all the letters of $\Lambda_{k,s}$. This weakens the condition that $y[i\,..\,j-1]$ is a Lyndon word, but still provides a useful upper bound: by Lemma 5 a proportion of at most $\frac{c}{\lambda}$ of the possibilities satisfy the property (in fact we cannot apply Lemma 5 if $k=1$ and $s=\sigma-1$, but the inequality trivially holds as Condition (i) forces $i=j-1$ and $c \geq 1$). Putting all together, the probability of $\mathcal{B}^{\geq}_{k,s,i}$ is bounded from above by $\frac{1}{\sigma^k}\left(\frac{\sigma^k-s}{\sigma^k}\right)^{\lambda}\frac{c}{\lambda}$.

So we established that, for $i < j-k$, $\mathbb{P}_n(C_{ij} \neq 0 \text{ and } \mathcal{B}^{(n)}_{k,s}) \leq q(k,s,\lambda)$ where

$$q(k,s,\lambda) = \frac{s}{\sigma^{2k}}\left(\frac{\sigma^k-s}{\sigma^k}\right)^{\lambda-1} + \frac{c}{\lambda\sigma^k}\left(\frac{\sigma^k-s}{\sigma^k}\right)^{\lambda}. \tag{1}$$

Observe that we only used conditions on the letters of indices smaller than $j+k$ to estimate the probability $\mathbb{P}_n(C_{ij} \neq 0 \text{ and } \mathcal{B}^{(n)}_{k,s})$. Hence, conditioned by $C_{ij} \neq 0$ and $\mathcal{B}^{(n)}_{k,s}$, the suffix $y[j+k\,..\,n-1]$ of y is a uniform random word of A^{n-j-k}. For any $z \in A^{j+k}$, consider a random word $y \in A^n$ that admits z as a prefix, which is the same as saying that $y = zz'$ where z' is a uniform random word of length $n-j-k$. The number of comparisons C_{ij} performed by the algorithm between $y[i\,..\,j-1]$ and $y[j\,..\,j+Lyn[j]-1]$ can be bounded from above by $k+1+D_{i+k,j+k}(y)$, where $D_{i+k,j+k}$ is the length of the longest common prefix between $y[i+k\,..\,n-1]$ and $y[j+k\,..\,n-1]$: we get the upper bound by considering that the k first comparisons are successful and by discarding the conditions on the lengths of the factors. If we fix z, the law of $1+D_{i+k,j+k}$ is a truncated geometric law of parameter $1-\frac{1}{\sigma}$ (we count the number of Bernoulli trials of parameter $1-\frac{1}{\sigma}$ until we get a success, i.e. a mismatch, or reach the end of the word). This can be in turn bounded from above by a geometric law of parameter $1-\frac{1}{\sigma}$. Hence $1+D_{i+k,j+k} \leq \mathrm{Geom}(1-\frac{1}{\sigma})$, so $\mathbb{E}[1+D_{i+k,j+k}] \leq \frac{\sigma}{\sigma-1} \leq 2$. Let pref_m be the random variable that associates to a word its prefix of length m. For $i < j-k$, this yields

$$\mathbb{E}_n[\mathbb{1}_{\mathcal{B}^{(n)}_{k,s}} \cdot C_{ij}] \leq \sum_{z\in\mathcal{B}^{(j+k)}_{k,s}} \mathbb{E}_n[\mathbb{1}_{\mathrm{pref}_{j+k}=z} \cdot C_{ij}] \leq \sum_{\substack{z\in\mathcal{B}^{(j+k)}_{k,s}\\ C_{ij}(z)\neq 0}} \frac{k+2}{\sigma^{j+k}}$$

$$= (k+2)\cdot \mathbb{P}_n(C_{ij}\neq 0 \text{ and } \mathcal{B}^{(n)}_{k,s}) \leq (k+2)\cdot q(k,s,\lambda).$$

Now we have to sum the contributions for all $i < j-k$, all $1 \leq s < \sigma-1$ and all $k \leq \lfloor 3\log_2 n\rfloor$. After tedious but elementary computations of sums, we obtain that there exists some positive constant γ_1 such that

The details are given in Appendix A.2

$$\sum_{k=1}^{\lfloor 3\log_2 n\rfloor} \sum_{s=1}^{\sigma-1} \sum_{i=0}^{j-k-1} \mathbb{E}_n[\mathbb{1}_{\mathcal{B}^{(n)}_{k,s}} \cdot C_{ij}] \leq \gamma_1. \tag{2}$$

We still have to estimate the contribution of the indices i such that $j-k \leq i < j$. For this, we just use Condition (i): $y[i\,..\,j-1]$ must be a Lyndon word for C_{ij} to be positive, which happens with probability at most $\frac{1}{\ell}$. The comparisons performed by the algorithm are evaluated as previously, by bounding them by an independent geometric law of parameter $1-\frac{1}{\sigma}$ plus k, yielding, as the probability that $y[j\,..\,j+k-1] = a_0^{k-1}a_s$ is σ^{-k}:

$$\mathbb{E}_n[\mathbb{1}_{\mathcal{B}^{(n)}_{k,s}} \cdot C_{ij}] \leq \frac{k+2}{\ell\sigma^k}.$$

Using the same kind of elementary techniques as for Eq. (2), we get that there exists a constant γ_2 such that

The details are given in Appendix A.3

$$\sum_{k=1}^{\lfloor 3\log_2 n\rfloor} \sum_{s=1}^{\sigma-1} \sum_{i=j-k}^{j-1} \mathbb{E}_n[\mathbb{1}_{\mathcal{B}^{(n)}_{k,s}} \cdot C_{ij}] \leq \gamma_2. \tag{3}$$

As a consequence, for all $j \leq n-3\log_2 n$ we have $\mathbb{E}_n[C_j] \leq \gamma_1+\gamma_2$, concluding the proof. ◀

Eventually we can state the main result of this section whose proof directly follows from Proposition 6 and Lemma 4, by summing for all j.

▶ **Theorem 7.** *For any $n \geq 1$ let A_n be an alphabet having $\sigma_n \geq 2$ letters. For the uniform distribution on words of length n over A_n, the expected number of comparisons performed by Algorithm* LyndonTable *(and by its variant* NaiveLyn *below) is linear.*

4 Linear computation of the Lyndon table

To describe the algorithm that computes the Lyndon table *Lyn* in linear time, we proceed in four steps. First, we consider the next smaller suffix table, which contains the same information as the table *Lyn*, and its dual version, the previous smaller suffix table. They form an important element in the left-to-right solution introduced by Bille et al. [7]. Second, we adapt another component of the left-to-right solution, namely skipping some letter comparisons when lexicographically comparing suffixes. We achieve this by efficiently computing the longest common extension (LCE) of the relevant suffixes, which is the length of their longest common prefix (see e.g. [10, Chapter 4]). In the third step, we show how to compute the LCEs even faster by reusing previously computed values. The fourth step completes the algorithm with small adjustments that lead to an overall linear running time.

4.1 Next and previous smaller suffix tables

From now on, we prepend and append an infinitely small sentinel symbol $y[-1] = y[n] = \$$ to the input word $y[0\,..\,n-1]$. This simplifies the description of algorithms, e.g. by ensuring that for any two positions it holds $y[i\,..\,n] < y[j\,..\,n] \iff y[i\,..\,n] \ll y[j\,..\,n]$. Note that this does not affect the lexicographical order of suffixes (i.e. $y[i\,..\,n-1] < y[j\,..\,n-1] \iff y[i\,..\,n] < y[j\,..\,n]$). As mentioned earlier, our algorithm uses the previous and next smaller suffix tables *pss* and *nss*, which are closely related to the Lyndon table. For each position i, where $0 \le i < n$, these tables store the (starting) positions of the closest lexicographically smaller suffixes, formally defined by

$$\begin{aligned} pss[i] &= \max\{j \mid j < i \text{ and } y[j\,..\,n] < y[i\,..\,n]\}, \text{ and} \\ nss[i] &= \min\{j \mid j > i \text{ and } y[j\,..\,n] < y[i\,..\,n]\}. \end{aligned}$$

The tables *Lyn* and *nss* are equivalent; indeed, the longest Lyndon prefix of $y[i\,..\,n-1]$ is exactly $y[i\,..\,nss[i]-1]$. Additionally, $y[pss[i]\,..\,i-1]$ is a Lyndon word.

▶ **Lemma 8** ([19, Lemma 15]). *For any position i of a word y, it holds $Lyn[i] = nss[i] - i$.*

▶ **Corollary 9** ([7, Lemma 4] and Lemma 8 above). *For any position i of a word y, both $y[pss[i]\,..\,i-1]$ and $y[i\,..\,nss[i]-1]$ are Lyndon words.*

An important property of nearest smaller suffixes is that they do not *intersect*. If we draw directed edges underneath the word such that for each position i there are two outgoing edges, one to position $pss[i]$ and one to position $nss[i]$, then the resulting drawing is a planar embedding. Formally, this is expressed by the following lemma.

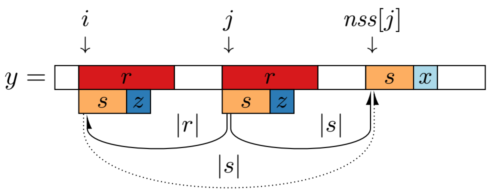
▶ **Lemma 10.** *If $i < j < nss[i]$, then $pss[j] \ge i$ and $nss[j] \le nss[i]$.*

Proof. Because of $i < j < nss[i]$, and by the definition of *nss*, it holds $y[nss[i]\,..\,n] < y[i\,..\,n] < y[j\,..\,n]$. This implies $pss[j] \ge i$ and $nss[j] \le nss[i]$. ◀

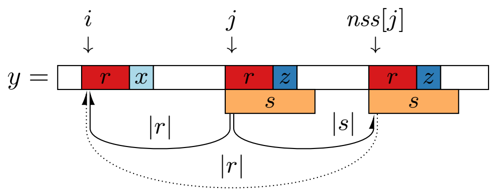
In the remainder of the section, we show how to simultaneously compute the tables *pss* and *nss* in right-to-left order. The core of our solution is a simple folklore algorithm for the linear time computation of next and previous smaller *values*, where we substitute value comparisons for lexicographical suffix comparisons. The process is similar to the linear-time construction of the Cartesian tree [30] or the LRM-tree [4].

NaiveLyn(y non-empty word $y[0\,..\,n-1]$ with sentinels $y[-1] = y[n] = \$$)

for $i \leftarrow n-1$ **downto** 0 **do**
  $j \leftarrow i+1$
  while $y[i\,..\,n] < y[j\,..\,n]$ **do**
    $(pss[j], j) \leftarrow (i, nss[j])$ ▷ $j \leftarrow j + Lyn[j]$
  $(nss[i], pss[i]) \leftarrow (j, -1)$ ▷ $Lyn[i] \leftarrow j - i$
return (pss, nss)



(a) If $|r| > |s|$ then $\text{LCE}(i, nss[j]) = |s|$ and $y[nss[j]\,..\,n] < y[i\,..\,n]$, and thus $nss[i] = nss[j]$.



(b) If $|r| < |s|$ then $\text{LCE}(i, nss[j]) = |r|$ and $y[nss[j]\,..\,n] > y[i\,..\,n]$, and thus $pss[nss[j]] = i$.

Figure 2 Skipping symbols comparisons when $y[i\,..\,n] < y[j\,..\,n]$. (Best viewed in colour.)

The algorithm merely adapts Algorithm LyndonTable to the computation of *nss*, up to the use of sentinels, and adds the computation of *pss*. In fact, if we omit the assignment of *pss* entries and apply Lemma 8 (i.e. if we replace lines 4–5 with their comments), then we essentially obtain Algorithm LongestLyndon described in [15, Algorithm 5] and in [13, Problem 87]. As shown in [15], the algorithm correctly computes the table *Lyn* (or respectively *nss*), and performs no more than $2n-2$ lexicographical suffix comparisons in line 3. Note that the loop in line 3 maintains the invariant that all suffixes $y[k\,..\,n]$ with $i < k < j$ are lexicographically larger than both $y[i\,..\,n]$ and $y[j\,..\,n]$. This means that the computation of *pss* is correct.

If we could lexicographically compare suffixes in constant time, then NaiveLyn would take $\mathcal{O}(n)$ time (as already mentioned in Section 2 for Algorithm LyndonTable). However, for each suffix comparison, we first have to determine the length $\text{LCE}(i,j) = \min\{\ell \mid \ell \geq 0 \text{ and } y[i+\ell] \neq y[j+\ell]\}$ of the longest common prefix of two suffixes. By the definition of the lexicographical order, $y[i\,..\,n] < y[j\,..\,n]$ is equivalent to $y[i+\text{LCE}(i,j)] < y[j+\text{LCE}(i,j)]$. If we compute $\text{LCE}(i,j)$ by naive scanning, then a single suffix comparison might require $\Omega(n)$ individual symbol comparisons, and for some inputs the algorithm will take $\Omega(n^2)$ time (e.g. for the pathological word $y = \mathtt{a}^n$).

4.2 Skipping symbol comparisons

Now we will accelerate the algorithm by exploiting previously computed LCEs. First, we show how to save comparisons for a single fixed value of i, i.e. for a single iteration of the outer loop of NaiveLyn. During that iteration, we may have to compute the LCE between i and multiple different values of j. Assume that we have just computed $\text{LCE}(i,j) = |r|$ (with $y[j\,..\,n] = ru$ for some $u \in A^*$), and we discovered that $y[i\,..\,n] < y[j\,..\,n]$. Then during the next iteration of the inner loop we have to evaluate whether $y[i\,..\,n] < y[nss[j]\,..\,n]$, thus we have to compute $\text{LCE}(i, nss[j])$. Since we previously computed $nss[j]$, we must have also computed the value $\text{LCE}(j, nss[j]) = |s|$ (with $y[j\,..\,n] = sv$ for some $v \in A^*$). We observe that exactly one of the following cases holds.

- It holds $|r| > |s|$ (as depicted in Figure 2a), such that $r = szw$ for some $z \in A$ and $w \in A^*$. Let $x = y[nss[j] + |s|]$, then from $y[j\,..\,n] > y[nss[j]\,..\,n]$ follows $x < z$ and thus $sx \ll sz$. Since $y[i\,..\,n]$ has prefix $r = szw$, we have $\text{LCE}(i, nss[j]) = |s|$ and $y[nss[j]\,..\,n] < y[i\,..\,n]$. Note that this implies $nss[i] = nss[j]$, such that this is the last iteration of the inner loop.
- It holds $|r| < |s|$ (as depicted in Figure 2b), such that $s = rzw$ for some $z \in A$ and $w \in A^*$. Let $x = y[i + |r|]$, then from $y[j\,..\,n] < y[i\,..\,n]$ follows $x < z$ and thus $rx \ll rz$. Since $y[nss[j]\,..\,n]$ has prefix $s = rzw$, we have $\text{LCE}(i, nss[j]) = |r|$ and $y[nss[j]\,..\,n] > y[i\,..\,n]$. Note that this implies $pss[nss[j]] = i$.
- It holds $r = s$ and thus $\text{LCE}(i, nss[j]) \geq |s|$.

The algorithm LceLyn shown below is a modification of NaiveLyn and exploits the new insights. It uses two auxiliary arrays *plce* and *nlce*, in which we store $plce[i] = \text{LCE}(pss[i], i)$ and $nlce[i] = \text{LCE}(i, nss[i])$ (we update the values whenever we assign $nss[i]$ and $pss[i]$ respectively). In iteration i of the outer loop, we compute the first LCE value $\ell = \text{LCE}(i, j)$ with $j = i + 1$ in constant time by exploiting the fact that for any index j it holds either $nss[j] = j + 1$ or $pss[j + 1] = j$. Thus, if $y[i\,..\,n]$ starts with a run of a single symbol, then we have previously computed the length of this run, and we can simply assign $\text{LCE}(i, j) \leftarrow 1 + \text{LCE}(j, j + 1)$ (lines 3–5). Whenever we reach the head of the inner loop, we have already computed the value $\ell = \text{LCE}(i, j)$. Thus, we can determine the lexicographical order of $y[i\,..\,n]$ and $y[j\,..\,n]$ by comparing their first mismatching symbol (line 6). If $y[i\,..\,n] < y[j\,..\,n]$, then we enter the inner loop and assign $pss[j] \leftarrow i$ and $plce[j] \leftarrow \ell$. For the next iteration of the inner loop, we have to compute $\text{LCE}(i, nss[j])$. Here we exploit our previous observations. If $\ell > nlce[j]$ (i.e. $|r| > |s|$ in terms of Figure 2a), then it holds $\text{LCE}(i, nss[j]) = nlce[j]$ and we continue with the next (and final) iteration of the inner loop with $\ell \leftarrow nlce[j]$ and $j \leftarrow nss[j]$ (lines 8–10). If $\ell < nlce[j]$ (i.e. $|r| < |s|$ in terms of Figure 2b), then it already holds $\ell = \text{LCE}(i, nss[j])$ and we continue with the next iteration of the inner loop with $j \leftarrow nss[j]$ (lines 11–12). Only if $\ell = nlce[j]$ we may need additional steps to compute $\text{LCE}(i, nss[j])$. However, in this case $\text{LCE}(i, nss[j]) \geq \ell$, and we can skip the first ℓ symbol comparisons when computing $\text{LCE}(i, nss[j])$. We compute the remaining part of the LCE by naive scanning, thus taking additional $\text{LCE}(i, nss[j]) - \ell + 1$ symbol comparisons (lines 13–15).

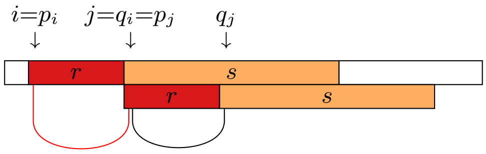
LceLyn(y non-empty word $y[0\,..\,n-1]$ with sentinels $y[-1] = y[n] = \$$)

1 **for** $i \leftarrow n-1$ **downto** 0 **do**
2     $j \leftarrow i + 1$
3   **if** $y[i] \neq y[j]$ **then** $\ell \leftarrow 0$
4   **elseif** $nss[j] = j + 1$ **then** $\ell \leftarrow 1 + nlce[j]$
5   **elseif** $pss[j+1] = j$ **then** $\ell \leftarrow 1 + plce[j + 1]$
6   **while** $y[i + \ell] < y[j + \ell]$ **do**
7         $(pss[j], plce[j]) \leftarrow (i, \ell)$
8   **if** $\ell > nlce[j]$ **then**
9             $\ell \leftarrow nlce[j]$
10             $j \leftarrow nss[j]$
11   **elseif** $\ell < nlce[j]$ **then**
12             $j \leftarrow nss[j]$
13   **elseif** $\ell = nlce[j]$ **then**
14             $j \leftarrow nss[j]$
15             $\ell \leftarrow \ell + \text{SCAN-LCE}(i + \ell, j + \ell)$ $\triangleright$ $\ell \leftarrow \text{EXTEND}(i, j, \ell)$
16     $(nss[i], nlce[i], pss[i]) \leftarrow (j, \ell, -1)$
17 **return** (pss, nss)

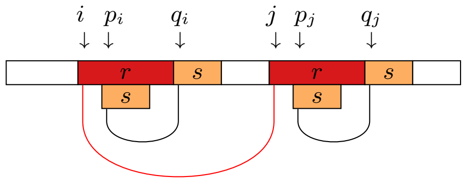
Apart from the time needed for executing line 15, algorithm LceLyn takes $\mathcal{O}(n)$ time.

4.3 Extending common prefixes with already computed LCEs

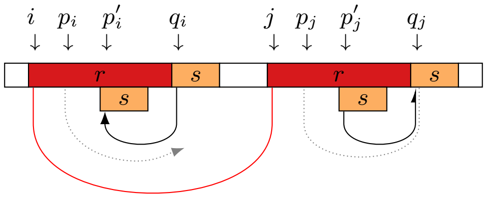
In this section, we accelerate the instruction at line 15 by replacing the naive computation of $\ell + \text{SCAN-LCE}(i + \ell, j + \ell)$ with a more sophisticated extension technique $\text{EXTEND}(i, j, \ell)$, for which we once more exploit previously computed LCEs. The technique requires $\ell > 0$. If



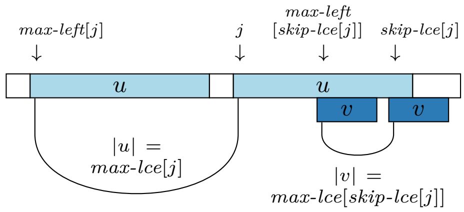
(a) $|s| = \text{LCE}(p_j, q_j)$.



(b) $|s| = \min(\text{LCE}(p_i, q_i), \text{LCE}(p_j, q_j))$.



(c) $|s| = \min(\text{LCE}(p'_i, q_i), \text{LCE}(p'_j, q_j))$.



(d) Relation between *skip-lce* and *max-lce*.

Figure 3 Extending common prefixes between $y[i \mathinner{..} n]$ and $y[j \mathinner{..} n]$ using a known lower bound $\ell = |r| \leq \text{LCE}(i, j)$ (a-c), and visualization of *skip-lce* (d). (Best viewed in colour.)

$\ell = 0$, then we simply perform one additional symbol comparison to test $y[i] = y[j]$, which either establishes $\ell = 1$, or terminates the LCE computation in constant time.

Assume that we just reached line 15, i.e. we have to compute $\text{LCE}(i, j)$, and we have already established $\text{LCE}(i, j) \geq \ell$. If $y[q_i] \neq y[q_j]$ with $q_i = i + \ell$ and $q_j = j + \ell$, then $\text{LCE}(i, j) = \ell$, i.e. no further computation is necessary. Otherwise, let $p_j \in \{j, \ldots, q_j - 1\}$ be some index with either $nss[p_j] = q_j$ or $pss[q_j] = p_j$. Such index always exists because it trivially holds either $nss[q_j - 1] = q_j$ or $pss[q_j] = q_j - 1$; we will explain how to choose p_j later. Let $p_i = p_j - (j - i)$. We compute $\text{LCE}(i, j)$ with exactly one of three methods.

1. If $p_i = i$ and $q_i = j$, then $p_i = p_j - (j - i)$ implies $p_j = p_i + j - i = j = q_i$ (as depicted in Figure 3a). From $\text{LCE}(i, j) \geq \ell$ follows $\text{LCE}(i, j) = \ell + \text{LCE}(i + \ell, j + \ell)$. Note that $\text{LCE}(i{+}\ell, j{+}\ell) = \text{LCE}(q_i, q_j) = \text{LCE}(p_j, q_j)$. Since we chose p_j such that either $nss[p_j] = q_j$ or $pss[q_j] = p_j$, we have already computed $\text{LCE}(p_j, q_j)$ and stored it either in $nlce[p_j]$ or in $plce[q_j]$. Thus we can compute $\text{LCE}(i, j) = \ell + \text{LCE}(p_j, q_j)$ in constant time.
2. If the first case does not apply, then we check whether either $nss[p_i] = q_i$ or $pss[q_i] = p_i$. If one of the two holds (as depicted in Figure 3b), then we have already computed $\ell_i = \text{LCE}(p_i, q_i)$ and stored it in $nlce[p_i]$ or in $plce[q_i]$. Analogously, we have already computed $\ell_j = \text{LCE}(p_j, q_j)$ and stored it in $nlce[p_j]$ or in $plce[q_j]$.
   - **a.** If $\ell_i = \ell_j$, we have established that $\text{LCE}(i, j) \geq \ell + \max(1, \ell_j)$. In this case, we say that *we use* $\text{LCE}(p_j, q_j)$ *to extend* $\text{LCE}(i, j)$. If ℓ_j is large, then we saved many symbol comparisons. We continue by recursively repeating the extension technique with $\ell \leftarrow \ell + \max(1, \ell_j)$. (We can always increase ℓ by at least 1 because we only reach this point if we initially ensured $y[q_i] = y[q_j]$.)
   - **b.** If $\ell_i \neq \ell_j$, then $\text{LCE}(i, j) = \ell + \min(\ell_i, \ell_j)$, and no further computation is necessary.
3. If none of the other two cases applies, then let $p'_i = pss[q_i]$ and $p'_j = p'_i + (j - i)$. This case is similar to case 2, but this time we use $\ell_i = \text{LCE}(p'_i, q_i)$ and $\ell_j = \text{LCE}(p'_j, q_j)$ (as depicted in Figure 3c). First, let us show that we have actually already computed these LCEs. Note that $pss[q_j] = p_j$ or $nss[p_j] = q_j$ and Corollary 9 imply that $y[p_j \mathinner{..} q_j - 1] = y[p_i \mathinner{..} q_i - 1]$ is a Lyndon word. Therefore, Lemma 8 implies $nss[p_i] \geq q_i$. We cannot have $nss[p_i] = q_i$ because then we would have used case 2 instead of case 3. Thus $nss[p_i] > q_i$, and due

to Lemma 10 it holds $p'_i = pss[q_i] \geq p_i$. Again, we cannot have $p'_i = p_i$ because then we would have used case 2 instead of case 3, i.e. it holds $p'_i > p_i$ and consequently $p'_j > p_j$. Finally, from $p'_i = pss[q_i]$ and Corollary 9 follows that $y[p'_i \mathinner{.\,.} q_i - 1] = y[p'_j \mathinner{.\,.} q_j - 1]$ is a Lyndon word, such that Lemma 8 implies $nss[p'_j] \geq q_j$. Since $p_j < p'_j < q_j$ and Lemma 10 imply $nss[p'_j] \leq q_j$, we must have $nss[p'_j] = q_j$. Therefore, we have already computed $\ell_i = \textsc{lce}(p'_i, q_i) = plce[q_i]$ and $\ell_j = \textsc{lce}(p'_j, q_j) = nlce[p'_j]$, which means that we can compute $\textsc{lce}(i, j) = \ell + \min(\ell_i, \ell_j)$ in constant time.
Note that even if $\ell_i = \ell_j$, it cannot be that $\textsc{lce}(i, j) > \ell + \ell_j$. Since ℓ_i is associated with a *previous* smaller value and ℓ_j is associated with a *next* smaller value, it holds $y[q_i + \ell_j] > y[p'_i + \ell_j] = y[p'_j + \ell_j] > y[q_j + \ell_j]$.

In cases 1, 2b and 3, we take constant time to finish the computation of $\textsc{lce}(i, j)$. Since we compute less than $2n$ LCEs, the total time spent for these cases is $\mathcal{O}(n)$. In case 2a, we take constant time to increase the known lower bound of $\textsc{lce}(i, j)$ by $\textsc{lce}(p_j, q_j)$ (however, when computing $\textsc{lce}(i, j)$ we may run into this case repeatedly). Thus we should choose p_j such that we maximize $\textsc{lce}(p_j, q_j)$. For this purpose, we maintain two auxiliary arrays *max-left* and *max-lce* (initialized with -1). Whenever we compute an LCE value $\textsc{lce}(i, j)$ (where by design of the algorithm it always holds $i < j$), we check whether $\textsc{lce}(i, j) > \textit{max-lce}[j]$. If this condition holds, then we assign $\textit{max-left}[j] \leftarrow i$ and $\textit{max-lce}[j] \leftarrow \textsc{lce}(i, j)$. For the extension technique, we always choose $p_j = \textit{max-left}[q_j]$. However, we may use each LCE value $\textsc{lce}(p_j, q_j)$ to extend multiple other LCEs, which may result in super linear computation time. We show how to avoid this in the following section.

4.4 Linear time extension of common prefixes

In this section, we ensure that we use each $\textsc{lce}(p_j, q_j)$ to extend at most one LCE. This imposes a linear upper bound on the number of recursive calls to $\textsc{extend}$, because each recursive call is preceded by an extension. For this, we need the following dynamic array.

▶ **Definition 11.** $\textit{skip-lce}[j] = \min\{k \mid k > j \text{ and } (k + \textit{max-lce}[k]) > (j + \textit{max-lce}[j])\}$.

A schematic drawing of *skip-lce* is provided in Figure 3d. We maintain these values in linear time using simple techniques. Whenever we have to update some value *max-lce*$[j]$, due to $\textsc{lce}(i, j)$ for some i, we inherently discover the new value of *skip-lce*$[j]$[1]. We may have to additionally assign $\textit{skip-lce}[x] \leftarrow j$ for some values $x \in \{i + 1, \ldots, j - 1\}$, but the total number of such updates is linear[2]. The technical details can be found in our well-documented implementation. Using *skip-lce*, the only two changes needed to achieve linear time are:

- In line 15 of LceLyn, we call $\textsc{extend}(i, j, 0)$ if and only if $\ell = 0$, and otherwise we call $\textsc{extend}(i, j, \textit{skip-lce}[j] - j)$. We can replace ℓ with $\textit{skip-lce}[j] - j$ because in this moment it holds $\textit{skip-lce}[j] - j \leq \textit{max-lce}[j] = \ell$. The special handling of $\ell = 0$ ensures that we compare the symbols $y[i]$ and $y[j]$.
- In case 2a of the extension technique, we replace $\ell \leftarrow \ell + \max(1, \ell_j)$ with $\ell \leftarrow \ell + \textit{skip-lce}[q_j] - q_j$. This is possible due to $\textit{skip-lce}[q_j] - q_j \leq \max(1, \textit{max-lce}[q_j]) = \max(1, \ell_j)$.

It may seem counter-intuitive that this improves the execution time, since we no longer extend by $\ell_j = \textit{max-lce}[q_j]$ but by the possibly shorter length $\textit{skip-lce}[q_j] - q_j$. However, this

[1] see `https://github.com/jonas-ellert/right-lyndon/blob/main/right-lyndon-extension-linear.hpp`, lines 88–135 (to be replaced by software heritage permanent link in published version)
[2] see same link as above, lines 52–71

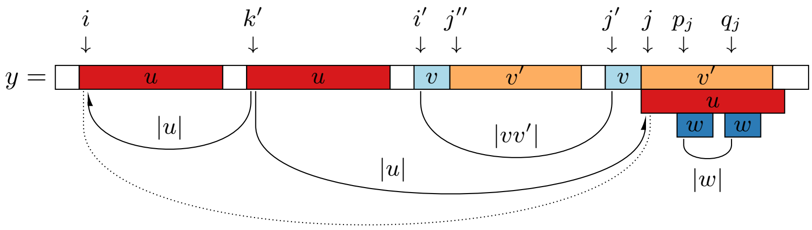


Figure 4 Supplementary drawing for the proof of Lemma 14. (Best viewed in colour.)

guarantees that we use each LCE in at most one extension step. A crucial observation for showing this is that q_j only assumes values that can be obtained by repeatedly applying *skip-lce* to j. For any j, we define the repeated application of the skip function as $\textit{skip-lce}^1[j] = \textit{skip-lce}[j]$ and for integer $e > 1$ as $\textit{skip-lce}^e[j] = \textit{skip-lce}^{e-1}[\textit{skip-lce}[j]]$. We then write $q_j = \textit{skip-lce}^*[j]$ if and only if $\exists e : q_j = \textit{skip-lce}^e[j]$. The following observation is a direct consequence of Definition 11 and extends to the helpful intermediate Lemma 13.

▶ **Observation 12.** *If $q_j = \textit{skip-lce}[j]$, then for every k with $j \le k < q_j$ it holds $k + \textit{max-lce}[k] < q_j + \textit{max-lce}[q_j]$.*

▶ **Lemma 13.** *If $q_j = \textit{skip-lce}^*[j]$, then for every k with $j \le k < q_j$ it holds $k + \textit{max-lce}[k] < q_j + \textit{max-lce}[q_j]$.*

▶ **Lemma 14.** *When running* LceLyn *with the modified extension technique using skip-lce, if we use $\textsc{lce}(p_j, q_j)$ for an extension, then we will never use it for an extension again.*

Proof. For the sake of contradiction, assume that we use $\textsc{lce}(p_j, q_j)$ more than once: first to compute $\textsc{lce}(i', j')$, and then again to compute $\textsc{lce}(i, j)$ for some i, j with $i \neq i'$ or $j \neq j'$. We have two cases to consider according to the position of j with respect to j':

Case $j < j'$: Since we use $\textsc{lce}(p_j, q_j)$ to extend $\textsc{lce}(i, j)$, it holds $q_j = \textit{skip-lce}^*[j]$. However, Lemma 13 implies that $j' + \textit{max-lce}[j'] < q_j + \textit{max-lce}[q_j]$ (note that $\textit{max-lce}[q_j]$ has not changed). This contradicts the assumption that we used $\textsc{lce}(p_j, q_j)$ to extend $\textsc{lce}(i', j')$.

Case $j \ge j'$: This case is accompanied by a schematic drawing in Figure 4. If we are comparing i and j then either $\textit{nss}[i] = j$ or $\textit{pss}[j] = i$, which implies $y[k\,..\,n] > y[j\,..\,n]$ for all k with $i < k < j$. Let $y[k'\,..\,n]$ be the lexicographically smallest suffix amongst the suffixes starting between positions i and j, then it holds $\textit{nss}[k'] = j$. Thus, at the time we compute $\textsc{lce}(i, j)$, we have computed $\textsc{lce}(k', j)$ already, and it holds $\textit{max-lce}[j] \ge \textsc{lce}(k', j)$. Let $j'' = i' + j - j'$, then due to our choice of k' it holds $y[j''\,..\,n] \ge y[k'\,..\,n] > y[j\,..\,n]$ and thus $\textsc{lce}(j'', j) \le \textsc{lce}(k', j)$. Therefore, we have $j + \textit{max-lce}[j] \ge j + \textsc{lce}(k', j) \ge j + \textsc{lce}(j'', j) = j' + \textsc{lce}(i', j') \ge q_j + \textsc{lce}(p_j, q_j)$. However, according to Lemma 13, $j + \textit{max-lce}[j] \ge q_j + \textsc{lce}(p_j, q_j)$ contradicts $q_j = \textit{skip-lce}^*[j]$, which means that we do not use $\textsc{lce}(p_j, q_j)$ to extend $\textsc{lce}(i, j)$. ◀

We now state the main results, which are direct consequences of Lemma 14 and Lemma 8.

▶ **Theorem 15.** *Algorithm* LceLyn *using the modified extension technique computes the previous and next smaller suffix tables of a word over a general ordered alphabet. It does so in a back-to-front online manner, and in linear time with respect to the length of the word.*

▶ **Corollary 16.** *Theorem 15 also holds when computing the Lyndon table and forest.*

Proof. For the Lyndon table we output *Lyn* alongside *nss*, using Lemma 8. For the Lyndon forest, we additionally interleave the computation with Algorithm LyndonForest. ◀

References

1 G. Badkobeh and M. Crochemore. Left Lyndon tree construction. In J. Holub and J. Zdárek, editors, *Prague Stringology Conference 2020, Prague, Czech Republic, August 31-September 2, 2020*, pages 84–95. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2020. https://arxiv.org/abs/2011.12742.

2 U. Baier. Linear-time Suffix Sorting - A New Approach for Suffix Array Construction. In *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, pages 23:1–23:12, 2016.

3 H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. The "runs" theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017.

4 J. Barbay, J. Fischer, and G. Navarro. Lrm-trees: Compressed indices, adaptive sorting, and compressed permutations. *Theor. Comput. Sci.*, 459:26–41, 2012.

5 F. Bassino, J. Clément, and C. Nicaud. The standard factorization of lyndon words: an average point of view. *Discret. Math.*, 290(1):1–25, 2005.

6 N. Bertram, J. Ellert, and J. Fischer. Lyndon words accelerate suffix sorting. In P. Mutzel, R. Pagh, and G. Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 15:1–15:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

7 P. Bille, J. Ellert, J. Fischer, I. L. Gørtz, F. Kurpicz, J. I. Munro, and E. Rotenberg. Space efficient construction of Lyndon arrays in linear time. In A. Czumaj, A. Dawar, and E. Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 14:1–14:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

8 D. Breslauer, T. Jiang, and Z. Jiang. Rotations of periodic strings and short superstrings. *J. Algorithms*, 24(2):340–353, 1997.

9 P. Chassaing and L. Mercier. The height of the lyndon tree. *Discrete Mathematics & Theoretical Computer Science*, 2013.

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

11 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.

12 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.

13 M. Crochemore, T. Lecroq, and W. Rytter. *125 Problems in Text Algorithms*. Cambridge University Press, 2021. 334 pages.

14 M. Crochemore and D. Perrin. Two-way string-matching. *J. Assoc. Comput. Mach.*, 38(3):651–675, 1991.

15 M. Crochemore and L. M. S. Russo. Cartesian and Lyndon trees. *Theoretical Computer Science*, 806:1–9, February 2020.

16 J. Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.

17 J. Ellert and J. Fischer. Linear time runs over general ordered alphabets. In N. Bansal, E. Merelli, and J. Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 63:1–63:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

18 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.

19 F. Franek, A. S. M. S. Islam, M. S. Rahman, and W. F. Smyth. Algorithms to compute the Lyndon array. In J. Holub and J. Zdárek, editors, *Proceedings of the Prague Stringology Conference 2016, Prague, Czech Republic, August 29-31, 2016*, pages 172–184. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2016.

20 F. Franek and M. Liut. Algorithms to compute the Lyndon array revisited. In J. Holub and J. Zdárek, editors, *Prague Stringology Conference 2019, Prague, Czech Republic, August 26-28, 2019*, pages 16–28. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2019.

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

22 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.

23 D. Kosolobov. Computing runs on a general alphabet. *Inf. Process. Lett.*, 116(3):241–244, 2016.

24 M. Lothaire. *Combinatorics on Words*. Addison-Wesley, 1983. Reprinted in 1997.

25 F. A. Louza, S. Mantaci, G. Manzini, M. Sciortino, and G. P. Telles. Inducing the Lyndon array. In N. R. Brisaboa and S. J. Puglisi, editors, *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings*, volume 11811 of *Lecture Notes in Computer Science*, pages 138–151. Springer, 2019.

26 R. C. Lyndon. On Burnside problem I. *Trans. Amer. Math. Soc.*, 77:202–215, 1954.

27 U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In D. S. Johnson, editor, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*, pages 319–327. SIAM, 1990.

28 S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. Sorting suffixes of a text via its lyndon factorization. In J. Holub and J. Zdárek, editors, *Proceedings of the Prague Stringology Conference 2013, Prague, Czech Republic, September 2-4, 2013*, pages 119–127. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2013.

29 J. Sawada and F. Ruskey. Generating Lyndon brackets.: An addendum to: Fast algorithms to generate necklaces, unlabeled necklaces and irreducible polynomials over GF(2). *J. Algorithms*, 46(1):21–26, 2003.

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

A Appendix

The following inequalities (obtained by comparing sums and integrals) will be needed.

▶ **Lemma 17.** *Let* $H_n = \sum_{i=1}^{n} \frac{1}{i}$ *and* $L_n = \sum_{i=1}^{n} \log i$*. For all* $n \geq 1$*,* $H_n \leq \log n + 1$ *and* $L_n \geq n \log n - n$*.*

Proof. The mapping $x \mapsto \log x$ is increasing, so for any $i \geq 1$ and any $x \in [i, i+1]$, we have $\log x \leq \log(i+1)$. We integrate on the length-1 interval $[i, i+1]$ to get $\int_i^{i+1} \log x \, dx \leq \log(i+1)$. Summing for i ranging from 1 to $n-1$ gives

$$\int_1^n \log x \, dx \leq \sum_{i=2}^{n} \log i = L_n,$$

which concludes the proof since $\int_1^n \log x \, dx = n \log n - n + 1$. The proof is similar for H_n by considering the decreasing map $x \mapsto \frac{1}{x}$. ◀

A.1 Proof of Lemma 5

Proof. Any Lyndon word w of length at least 2 can uniquely be written [16] as $w = u^+ vb$ where u is a Lyndon word, va is a prefix of u, and a and b are letters such that $a < b$. Thus, if $y\#$ is a Lyndon word, then y can be uniquely written $y = ux$ where x is the longest border of y (or the empty word if y is a Lyndon word). As y is entirely defined by its associated u and its length t, this yields a bijection between the words y such that $y\# \in \mathcal{L}$ and the union of the Lyndon words of length i, for i ranging from 1 to t. Let $\kappa = |\Lambda|$. As we already established that $\left|\mathcal{L} \cap \Lambda^i\right| \leq \frac{\kappa^i}{i}$, we have

$$\mathbb{P}_t(y\# \in \mathcal{L}) \leq \frac{1}{\kappa^t} \sum_{i=1}^{t} \frac{\kappa^i}{i} = \sum_{i=0}^{t-1} \frac{\kappa^{-i}}{t-i} = \frac{1}{t} \sum_{i=0}^{t-1} \frac{\kappa^{-i}}{1 - i/t}.$$

Observe that for any $x \in [0, \frac{t-1}{t}]$, we have $\frac{1}{1-x} \leq 1 + tx$, and therefore

$$\frac{1}{t} \sum_{i=0}^{t-1} \frac{\kappa^{-i}}{1 - i/t} \leq \frac{1}{t} \sum_{i=0}^{t-1} (1+i)\kappa^{-i} \leq \frac{c}{t},$$

if we set $c = \sum_{i=0}^{\infty} (1+i)\kappa^{-i}$. This concludes the proof. ◀

A.2 Proof of Equation (2)

Recall that $\lambda = \lfloor \frac{j-i}{k} \rfloor$, so that there are k values of i for each λ. Hence

$$\sum_{i=0}^{j-k-1} (k+2)\, q\left(k, s, \left\lfloor \frac{j-i}{k} \right\rfloor\right) \leq k(k+2) \sum_{\lambda=1}^{\lceil j/k \rceil} q\left(k, s, \lambda\right).$$

We consider separately the two terms coming from Eq. (1):

$$\begin{aligned}
\sum_{k=1}^{\lfloor 3\log_2 n \rfloor} \sum_{s=1}^{\sigma-1} k(k+2) \sum_{\lambda=1}^{\lceil j/k \rceil} \frac{s}{\sigma^{2k}} \left(\frac{\sigma^k - s}{\sigma^k}\right)^{\lambda-1} &\leq \sum_{k=1}^{\infty} \sum_{s=1}^{\sigma-1} \frac{k(k+2)s}{\sigma^{2k}} \sum_{\lambda=1}^{\infty} \left(1 - \frac{s}{\sigma^k}\right)^{\lambda-1} \\
&= \sum_{k=1}^{\infty} \sum_{s=1}^{\sigma-1} \frac{k(k+2)s}{\sigma^{2k}} \frac{1}{1 - (1 - s/\sigma^k)} \\
&= \sum_{k=1}^{\infty} \sum_{s=1}^{\sigma-1} \frac{k(k+2)}{\sigma^k} \\
&\leq \sum_{k=1}^{\infty} \frac{k(k+2)}{2^{k-1}} =: \nabla_2.
\end{aligned}$$

For the second part of Eq. (1), we have:

$$\sum_{k=1}^{\lfloor 3\log_2 n\rfloor}\sum_{s=1}^{\sigma-1}k(k+2)\sum_{\lambda=1}^{\lfloor j/k\rfloor}\frac{c}{\lambda\sigma^k}\left(\frac{\sigma^k-s}{\sigma^k}\right)^{\lambda} \leq \sum_{k=1}^{\infty}\sum_{s=1}^{\sigma-1}\frac{ck(k+2)}{\sigma^k}\sum_{\lambda=1}^{\infty}\frac{1}{\lambda}\left(1-\frac{s}{\sigma^k}\right)^{\lambda}$$
$$= \sum_{k=1}^{\infty}\sum_{s=1}^{\sigma-1}\frac{ck(k+2)}{\sigma^k}\log\left(\frac{1}{1-(1-s/\sigma^k)}\right)$$
$$\leq \sum_{k=1}^{\infty}\sum_{s=1}^{\sigma-1}\frac{ck(k+2)}{\sigma^k}\log\left(\frac{\sigma^k}{s}\right).$$

We have

$$\sum_{k=1}^{\infty}\sum_{s=1}^{\sigma-1}\frac{ck(k+2)}{\sigma^k}\log\left(\frac{\sigma^k}{s}\right) = \sum_{s=1}^{\sigma-1}\frac{3c}{\sigma}\log\left(\frac{\sigma}{s}\right) + \sum_{k=2}^{\infty}\sum_{s=1}^{\sigma-1}\frac{ck(k+2)}{\sigma^k}\log\left(\frac{\sigma^k}{s}\right).$$

We use Lemma 17 for the first sum:

$$\sum_{s=1}^{\sigma-1}\log\left(\frac{\sigma}{s}\right) = (\sigma-1)\log\sigma - \sum_{s=1}^{\sigma-1}\log s \leq (\sigma-1)\log\sigma - (\sigma-1)\log(\sigma-1)+\sigma-1$$
$$= (\sigma-1)\left(\log\left(\frac{\sigma}{\sigma-1}\right)+1\right) \leq (1+\log 2)\sigma.$$

Thus

$$\sum_{s=1}^{\sigma-1}\frac{3c}{\sigma}\log\left(\frac{\sigma}{s}\right) \leq 3c(1+\log 2) =: \nabla_2.$$

Finally

$$\sum_{k=2}^{\infty}\sum_{s=1}^{\sigma-1}\frac{ck(k+2)}{\sigma^k}\log\left(\frac{\sigma^k}{s}\right) \leq \sum_{k=2}^{\infty}\sum_{s=1}^{\sigma-1}\frac{ck^2(k+2)\log\sigma}{\sigma^k}$$
$$\leq \frac{\log\sigma}{\sigma}\sum_{k=2}^{\infty}\frac{ck^2(k+2)}{\sigma^{k-2}} \leq \sum_{k=2}^{\infty}\frac{ck^2(k+2)}{2^{k-2}} =: \nabla_3.$$

This concludes the proof by setting $\gamma_1 = \nabla_1 + \nabla_2 + \nabla_3$.

A.3 Proof of Equation (3)

Using Lemma 17 we have

$$\sum_{k=1}^{\lfloor 3\log_2 n\rfloor}\sum_{s=1}^{\sigma-1}\sum_{\ell=1}^{k-1}\frac{k+2}{\ell\sigma^k} \leq \frac{\sigma-1}{\sigma}\sum_{k=1}^{\lfloor 3\log_2 n\rfloor}\frac{(k+2)(\log k+1)}{\sigma^{k-1}}$$
$$\leq \sum_{k=0}^{\infty}\frac{(k+3)(\log(k+1)+1)}{\sigma^k}$$
$$\leq \sum_{k=0}^{\infty}\frac{(k+3)(\log(k+1)+1)}{2^k} =: \gamma_2.$$