Linear Time Runs Over General Ordered Alphabets

Jonas Ellert
Department of Computer Science, Technical University of Dortmund, Germany

Johannes Fischer
Department of Computer Science, Technical University of Dortmund, Germany

Abstract

A run in a string is a maximal periodic substring. For example, the string `bananatree` contains the runs `anana` = $(\texttt{an})^{5/2}$ and `ee` = $\texttt{e}^2$. There are less than n runs in any length-n string, and computing all runs for a string over a linearly-sortable alphabet takes $\mathcal{O}(n)$ time (Bannai et al., SIAM J. Comput. 2017). Kosolobov conjectured that there also exists a linear time runs algorithm for general ordered alphabets (Inf. Process. Lett. 2016). The conjecture was almost proven by Crochemore et al., who presented an $\mathcal{O}(n\alpha(n))$ time algorithm (where $\alpha(n)$ is the extremely slowly growing inverse Ackermann function). We show how to achieve $\mathcal{O}(n)$ time by exploiting combinatorial properties of the Lyndon array, thus proving Kosolobov's conjecture. This also positively answers the at least 29-year-old question whether square-freeness can be tested in linear time over general ordered alphabets (Breslauer, PhD thesis, Columbia University 1992).

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases String algorithms, Lyndon array, runs, squares, longest common extension, general ordered alphabets, combinatorics on words

Digital Object Identifier 10.4230/LIPIcs.ICALP.2021.63

Category Track A: Algorithms, Complexity and Games

Supplementary Material *Software (Source Code)*: `https://github.com/jonas-ellert/linear-time-runs/`; archived at `swh:1:snp:aa7b1dc6293d939b5ec6e554d3102b15a518b7e7`

1 Introduction and Related Work

A run in a string S is a maximal periodic substring. For example, the string S = `bananatree` contains exactly the runs `anana` = $(\texttt{an})^{5/2}$ and `ee` = $\texttt{e}^2$. Identifying such repetitive structures in strings is of great importance for applications like text compression, text indexing and computational biology (for a general overview see [8]). To name just one example, runs in human genes (called maximal tandem repeats) are involved with a number of neurological disorders [5]. In 1999, Kolpakov and Kucherov showed that the maximum number $\rho(n)$ of runs in a length-n string is bounded by $\mathcal{O}(n)$, and provided a word RAM algorithm that outputs all runs in linear time [18]. The algorithm is based on the Lempel-Ziv factorization and only achieves $\mathcal{O}(n)$ time for *linearly-sortable alphabets*, i.e. alphabets that are totally ordered and for which a sequence of σ alphabet symbols can be sorted in $\mathcal{O}(\sigma)$ time. Since then, it has been an open question whether there exists a linear time runs algorithm for *general ordered alphabets*, i.e. totally ordered alphabets for which the order of any two symbols can be determined in constant time. Any such algorithm must not use the Lempel-Ziv factorization, since for general ordered alphabets of size σ it cannot be constructed in $o(n \lg \sigma)$ time [19].

Kolpakov and Kucherov also conjectured that the maximum number of runs is bounded by $\rho(n) < n$, which started a 15 year-long search for tighter upper bounds of $\rho(n)$. Rytter was the first to give an explicit constant with $\rho(n) < 5n$ [25]. After multiple incremental improvements of this bound (e.g. [7, 9, 24]), Bannai et al. [2] finally proved the conjecture by showing $\rho(n) < n$ for arbitrary alphabets, which was subsequently even surpassed for binary texts [12]. (The current best bound for binary alphabets is $\rho(n) < \frac{183}{193}n$ [17].)

© Jonas Ellert and Johannes Fischer;
licensed under Creative Commons License CC-BY 4.0

On the algorithmic side, Bannai et al. also provided a new linear time algorithm that computes all the runs [2]. While (just like the algorithm by Kolpakov and Kucherov) it only achieves the time bound for linearly-sortable alphabets, it no longer relies on the Lempel-Ziv factorization. Instead, the main effort of the algorithm lies in the computation of $\Theta(n)$ *longest common extensions (LCEs)*; given two indices $i, j \in [1, n]$, their LCE is the length of the longest common prefix of the suffixes $S[i..n]$ and $S[j..n]$. For linearly-sortable alphabets, we can precompute a data structure in $\mathcal{O}(n)$ time that answers arbitrary LCE queries in constant time (see e.g. [11]), thus yielding a linear time runs algorithm. Kosolobov showed that for general ordered alphabets any batch of $\mathcal{O}(n)$ LCEs can be computed in $\mathcal{O}(n \lg^{2/3} n)$ time, and conjectured the existence of a linear time runs algorithm for general ordered alphabets [20]. Gawrychowski et al. improved this result to $\mathcal{O}(n \lg \lg n)$ time [14]. Finally, Crochemore et al. noted that the required LCEs satisfy a special non-crossing property. They showed how to compute $\mathcal{O}(n)$ non-crossing LCEs in $\mathcal{O}(n\alpha(n))$ time, resulting in an $\mathcal{O}(n\alpha(n))$ time algorithm that computes all runs over general ordered alphabets [10] (where α is the inverse Ackermann function).

This is also the asymptotically fastest known algorithm for testing whether a string is square-free. A square is a substring $\alpha\alpha$ for some non-empty word α. A string is square-free if and only if it contains no runs (because every square is contained in a run, and every run contains at least one square). The question whether square-freeness over general ordered alphabets can be tested in linear time dates back at least to Breslauer's PhD dissertation [4, Section 4.4], which was published almost 30 years ago.[1] Testing for square-freeness over general *unordered* alphabets (where only constant time equality testing of symbols is permitted) takes at least $\Omega(n \lg n)$ symbol comparisons [22].

Our Contributions. We show how to compute the LCEs required by the algorithm by Bannai et al. in $\mathcal{O}(n)$ time and space, resulting in the first linear time runs algorithm for general ordered alphabets. Thus we prove Kosolobov's conjecture, and provide the first linear time algorithm to test for square-freeness over general ordered alphabets. Our solution differs from all previous approaches in the sense that it cannot answer a sequence of *arbitrary* non-crossing LCE queries. Instead, our algorithm is specifically designed *exactly for the LCEs required by the runs algorithm.* This allows us to utilize powerful combinatorial properties of the *Lyndon array* (a definition follows in Section 2) that do not generally hold for arbitrary non-crossing LCE sequences.

Even though the main contribution of our work is the improved asymptotic time bound, it is worth mentioning that our algorithm is also very fast in practice. On modern hardware, computing all runs for a text of length $10^7 (= 10\text{MB})$ takes only one second.

A Note on the Model. As mentioned earlier, our algorithm runs in linear time for *general ordered alphabets*, whereas previous algorithms achieve this time bound only when the alphabet is *linearly-sortable.* This is comparable with the distinction between *comparison-based* and *integer* sorting: while in the comparison-model sorting n items requires $\Omega(n \lg n)$ time, integer sorting is faster ($O(n\sqrt{\lg \lg n})$ time [16] and sometimes even linear, e.g. when the word width w satisfies $w = \mathcal{O}(\lg n)$ and one can use radix sort, or when $w \geq (\lg^{2+\epsilon} n)$ [1]). Whereas it is a major open problem whether integer sorting can always be done in linear time, this paper settles a symmetric open problem for the computation of runs.

[1] We thank the reviewer who kindly pointed out to us that this was an open problem.

The remainder of the paper is structured as follows: First, we introduce the basic notation, definitions, and auxiliary lemmas (Section 2). Then, we give a simplified description of the runs algorithm by Bannai et al. and show how the required LCEs relate to the Lyndon array (Section 3). Our linear time algorithm to compute the LCEs is described in Section 4. We discuss additional practical aspects and experimental results in Section 5.

2 Preliminaries

Our analysis is performed in the word RAM model (see e.g. [15]), where we can perform fundamental operations (logical shifts, basic arithmetic operations etc.) on words of size w bits in constant time. For an input of size n we assume $\lceil \log_2 n \rceil \leq w$. We write $[i, j] = [i, j+1) = (i-1, j] = (i-1, j+1)$ with $i, j \in \mathbb{N}$ to denote the set of integers $\{x \mid x \in \mathbb{N} \land i \leq x \leq j\}$.

Strings. Let Σ be a finite and totally ordered set. A *string* S of length $|S| = n$ over the *alphabet* Σ is a sequence $S[1] \dots S[n]$ of n *symbols* (also called *characters*) from Σ. The alphabet is called a *general ordered alphabet* if order testing (i.e. evaluating $\sigma_1 < \sigma_2$ for $\sigma_1, \sigma_2 \in \Sigma$) is possible in constant time. For $i, j \in [1, n]$, we use the interval notation $S[i..j] = S[i..j+1) = S(i-1..j] = S(i-1..j+1)$ to denote the *substring* $S[i] \dots S[j]$. If however $i > j$, then $S[i..j]$ denotes the *empty string* ϵ. The substring $S[i..j]$ is called *proper* if $S[i..j] \neq S$. A *prefix* of S is a substring $S[1..j]$ (including $S[1..0] = \epsilon$), while the *suffix* S_i is the substring $S[i..n]$ (including $S_{n+1} = \epsilon$). Given two strings S and T of length n and m respectively, their concatenation is defined as $ST = S[1] \dots S[n]T[1] \dots T[m]$. For any positive integer k, the k-times concatenation of S is denoted by S^k. Let $\ell_{\max} = \min(n, m)$. The *longest common prefix (LCP)* of S and T has length $\textsc{lcp}(S, T) = \max\{\ell \mid \ell \in [0, \ell_{\max}] \land S[1..\ell] = T[1..\ell]\}$, while the *longest common suffix* has length $\textsc{lc-suff}(S, T) = \max\{\ell \mid \ell \in [0, \ell_{\max}] \land S_{n-\ell+1} = T_{m-\ell+1}\}$. Let $\ell' = \textsc{lcp}(S, T)$. For a string S of length n and indices $i, j \in [1, n]$, we define the *longest common right-extension (R-LCE) and left-extension (L-LCE)* as $\textsc{lce}_r(i, j) = \textsc{lcp}(S_i, S_j)$ and $\textsc{lce}_\ell(i, j) = \textsc{lc-suff}(S[1..i], S[1..j])$ respectively. The total order on Σ induces a *lexicographical order* $\prec$ on the strings over Σ in the usual way. Given three suffixes, we can deduce properties of their R-LCEs from their lexicographical order:

▶ **Lemma 1.** *Let $S_i \prec S_j \prec S_k$ be suffixes of a string, then it holds $\textsc{lce}_r(i, k) \leq \textsc{lce}_r(i, j)$ and $\textsc{lce}_r(i, k) \leq \textsc{lce}_r(j, k)$.*

Proof. Assume $\ell = \textsc{lce}_r(i, j) < \textsc{lce}_r(i, k)$, then $S_i[1..\ell] = S_j[1..\ell] = S_k[1..\ell]$ and $S_j[\ell+1] \neq S_i[\ell+1] = S_k[\ell+1]$. This implies $S_i \prec S_j \Leftrightarrow S_k \prec S_j$, which contradicts $S_i \prec S_j \prec S_k$. The proof of $\textsc{lce}_r(i, k) \leq \textsc{lce}_r(j, k)$ works analogously. ◀

Repetitions and Runs. Let S be a string and let $S[i..j]$ be a non-empty substring. We say that $p \in \mathbb{N}^+$ is a *period* of $S[i..j]$ if and only if $\forall x \in [i, j-p] : S[x] = S[x+p]$. If additionally $(j - i + 1) \geq p$, then $S[i..i+p)$ is called *string period* of $S[i..j]$. Furthermore, p is called *shortest period* of $S[i..j]$ if there is no $q \in [1, p)$ that is also a period of $S[i..j]$. Analogously, a string period of $S[i..j]$ is called *shortest string period* if there is no shorter string period of $S[i..j]$. A *run* is a triple $\langle i, j, p \rangle$ such that p is the shortest period of $S[i..j]$, $(j - i + 1) \geq 2p$ (i.e. there are at least two consecutive occurrences of the shortest string period $S[i..i+p)$), and neither $\langle i-1, j, p \rangle$ nor $\langle i, j+1, p \rangle$ satisfies these properties (assuming $i > 1$ and $j < n$, respectively).

Lyndon Words and Nearest Smaller Suffixes. For a length-n string S and $i \in [1, n]$, the string $S_i S[1..i)$ is called *cyclic shift* of S, and *non-trivial cyclic shift* if $i > 1$. A *Lyndon word* is a non-empty string that is lexicographically smaller than any of its non-trivial cyclic shifts, i.e. $\forall i \in [2, n] : S \prec S_i S[1..i)$. The Lyndon array of S identifies the longest Lyndon word starting at each position of S.

▶ **Definition 2** (Lyndon Array). *Given a string S of length n, its Lyndon array $\lambda[1, n]$ is defined by $\forall i \in [1, n] : \lambda[i] = \max\{j - i + 1 \mid j \in [i, n] \wedge S[i..j]$ is a Lyndon word$\}$.*

An alternative representation of the Lyndon array is the next-smaller-suffix array.

▶ **Definition 3** (Next Smaller Suffixes). *Given a string S of length n, its* next-smaller-suffix (NSS) array *is defined by $\forall i \in [1, n] : \mathsf{nss}[i] = \min\{j \mid j = n + 1 \vee (j \in (i, n] \wedge S_i \succ S_j)\}$. If $\mathsf{nss}[i] \leq n$, then $S_{\mathsf{nss}[i]}$ is called the* next smaller suffix *of S_i.*

▶ **Lemma 4** (Lemma 15 [13]). *The longest Lyndon word starting at any position $i \in [1, n]$ of a length-n string S is exactly the substring $S[i..\mathsf{nss}[i])$, i.e. $\forall i \in [1, n] : \lambda[i] = \mathsf{nss}[i] - i$.*

An example of the Lyndon and NSS array is provided in Figure 1a. The NSS edges in the example do not intersect. This property also holds in the general case:

▶ **Lemma 5.** *Let $i \in [1, n]$ and $i' \in [i, \mathsf{nss}[i])$. Then it holds $\mathsf{nss}[i'] \leq \mathsf{nss}[i]$.*

Proof. Due to $i' \in [i, \mathsf{nss}[i])$ and Definition 3 it holds $S_{i'} \succ S_{\mathsf{nss}[i]}$. Assume that the lemma does not hold, then we have $\mathsf{nss}[i] \in (i', \mathsf{nss}[i'])$ and Definition 3 implies $S_{i'} \prec S_{\mathsf{nss}[i]}$. ◀

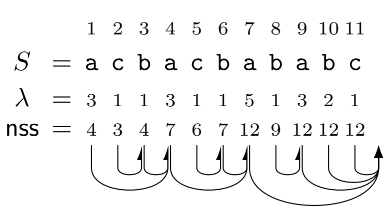
3 The Runs Algorithm Revisited

In this section, we recapitulate the main ideas of the algorithm by Bannai et al. [2], which is the basis of our solution for general ordered alphabets. The key insight is that every run is *rooted* in a longest Lyndon word, allowing us to compute all runs from the Lyndon array.

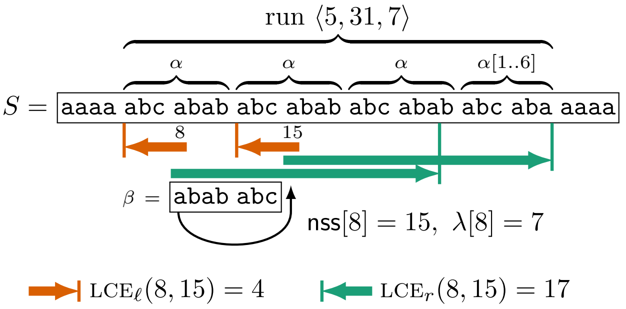
▶ **Definition 6.** *Let $\langle i, j, p\rangle$ be a run in a string S. We say that $\langle i, j, p\rangle$ is* (lexicographically) decreasing *if and only if $S_i \succ S_{i+p}$. Otherwise, $\langle i, j, p\rangle$ is* (lexicographically) increasing.

▶ **Lemma 7.** *Let $\langle i, j, p\rangle$ be a decreasing run, then there is exactly one index $i_0 \in [i..i + p)$ such that $\lambda[i_0] = p$.*

Proof. Consider any $i_0 \in [i, i+p)$. By the definition of runs, we have $S[i..i_0) = S[i+p..i_0+p)$. Since the run is decreasing it follows $S_i \succ S_{i+p} \iff S[i..i_0)S_{i_0} \succ S[i + p..i_0 + p)S_{i_0+p} \iff S_{i_0} \succ S_{i_0+p}$. This implies $\mathsf{nss}[i_0] \leq i_0 + p$, and due to Lemma 4 also $\lambda[i_0] \leq p$. Next, we show that there is at least one index $i_0 \in [i..i + p)$ such that $S[i_0..i_0 + p)$ is a Lyndon word. Let $\alpha = S[i..i+p)$. Assume that the described index i_0 does not exist, then from $S[i..i+2p) = \alpha\alpha$ follows that no cyclic shift of α is a Lyndon word. Let β be a lexicographically minimal cyclic shift of α, then this shift is not unique (otherwise β would be a Lyndon word), and thus there must be a cyclic shift $\beta_k\beta[1..k) = \beta[1..k)\beta_k$ with $k > 1$. This however implies that β is of the form $\beta = \mu^k$ for some string μ and an integer $k > 1$ (see Lemma 3 in [21]), which contradicts the fact that α is the *shortest* string period of the run. Finally, let $\alpha_k\alpha[1..k)$ with $k \in [1, p]$ be the unique lexicographically smallest cyclic shift of α (and thus a Lyndon word), then it is easy to see that only $i_0 = i + k - 1$ satisfies $\lambda[i_0] = p$. ◀



(a) String $S =$ acbacbababc, its Lyndon array λ, and its NSS array nss.



(b) Decreasing run $\langle 5, 31, 7\rangle$ with $S[5..31] = (\texttt{abcabab})^{27/7}$. The run has shortest string period $\alpha =$ abcabab, and is rooted in position 8 (with longest Lyndon word $\beta = S[8..15) = \alpha_4\alpha[1..3] =$ abababc).

Figure 1 An edge from text position a to text position b indicates $\mathsf{nss}[a] = b$.

▶ **Definition 8** (Root of a Run). *Let $\langle i, j, p\rangle$ be a decreasing run, and let $i_0 \in [i..i + p)$ be the unique index with $\lambda[i_0] = p$ (as described in Lemma 7). We say that $\langle i, j, p\rangle$ is* rooted *in i_0.*

An example of a decreasing run and its root is provided in Figure 1b. Note that our notion of a root differs from the L-roots introduced by Crochemore et al. [6]. While an L-root is *any* length-p Lyndon word contained in the run, our root is exactly the *leftmost* one.

Given a longest Lyndon word $S[i_0..\mathsf{nss}[i_0])$ of length $p = \mathsf{nss}[i_0] - i_0 = \lambda[i_0]$, it is easy to determine whether i_0 is the root of a decreasing run. We simply try to extend the periodicity as far as possible to both sides by using the LCE functions. For this purpose, we only need to compute $l = \text{LCE}_\ell(i_0, \mathsf{nss}[i_0])$ and $r = \text{LCE}_r(i_0, \mathsf{nss}[i_0])$. Let $i = i_0 - l + 1$ and $j = \mathsf{nss}[i_0] + r - 1$, then clearly the substring $S[i..j]$ has smallest period p, and we cannot extend the substring to either side without breaking the periodicity. Thus, if $j - i + 1 \geq 2p$ then $\langle i, j, p\rangle$ is a run. Note that this run is only rooted in i_0 if additionally $i_0 \in [i..i + p)$ (or equivalently $l \leq p$) holds. For the index $i_0 = 8$ in Figure 1b, we have $l = \text{LCE}_\ell(8, 15) = 4$ and $r = \text{LCE}_r(8, 15) = 17$. Therefore, the run starts at position $i = 8 - 4 + 1 = 5$ and ends at position $j = 15 + 17 - 1 = 31$. From $l = 4 \leq 7 = p$ follows that 8 is actually the root.

Since each decreasing run is rooted in exactly one index, we can find all decreasing runs by checking for each index whether it is the root of a run. This procedure is outlined in Algorithm 1. First, we compute the NSS array (line 2). Then, we investigate one index $i_0 \in [1, n]$ at a time (line 3), and consider it as the root of a run with period $p = \mathsf{nss}[i_0] - i_0$ (line 4). If the left-extension covers an entire period (i.e. $\text{LCE}_\ell(i_0, \mathsf{nss}[i_0]) > p$), then we have already investigated the root of the run in an earlier iteration of the for-loop, and no further action is required (line 5). Otherwise, we compute the left and right border of the potential run as described earlier (lines 6–7). If the resulting interval has length at least $2p$, then we have discovered a run that is rooted in i_0 (lines 8–9).

Time and space complexity. The NSS array can be computed in $\mathcal{O}(n)$ time and space for general ordered alphabets [3]. Assume for now that we can answer L-LCE and R-LCE queries in constant time, then clearly the rest of the algorithm also requires $\mathcal{O}(n)$ time and space. The correctness of the algorithm follows from Lemma 7 and the description. We have shown:

Algorithm 1 Compute all decreasing runs.

Input: String S of length n.
Output: Set R of all decreasing runs in S.
1: $R \leftarrow \emptyset$
2: compute array nss
3: **for** $i_0 \in [1, n]$ **with** $\mathsf{nss}[i_0] \neq n + 1$ **do**
4: $\quad p \leftarrow \mathsf{nss}[i_0] - i_0$
5: $\quad$ **if** $\text{LCE}_\ell(i_0, \mathsf{nss}[i_0]) \leq p$ **then**
6: $\quad\quad i \leftarrow i_0 - \text{LCE}_\ell(i_0, \mathsf{nss}[i_0]) + 1$
7: $\quad\quad j \leftarrow \mathsf{nss}[i_0] + \text{LCE}_r(i_0, \mathsf{nss}[i_0]) - 1$
8: $\quad\quad$ **if** $j - i + 1 \geq 2p$ **then**
9: $\quad\quad\quad R \leftarrow R \cup \{\langle i, j, p\rangle\}$

▶ **Lemma 9.** *Let S be a string of length n over a general ordered alphabet, and let nss be its NSS array. We can compute all decreasing runs of S in $\mathcal{O}(n) + t(n)$ time and $\mathcal{O}(n) + s(n)$ space, where $t(n)$ and $s(n)$ are the time and space needed to compute $\text{LCE}_\ell(i, \mathsf{nss}[i])$ and $\text{LCE}_r(i, \mathsf{nss}[i])$ for all $i \in [1, n]$ with $\mathsf{nss}[i] \neq n + 1$.*

In order to also find all *increasing* runs, we only need to rerun the algorithm with reversed alphabet order. This way, previously increasing runs become decreasing.

4 Algorithm for Computing the LCEs

In this section, we show how to precompute the LCEs required by Algorithm 1 in linear time and space. Our approach is asymmetric in the sense that we require different algorithms for L-LCEs and R-LCEs (whereas previous approaches usually compute L-LCEs by applying the R-LCE algorithm to the reverse text). However, for both directions we use similar properties of the Lyndon array that are shown in Lemmas 10 and 11 and visualized in Figure 2a.

▶ **Lemma 10.** *Let $i \in [1, n]$ and $j = \mathsf{nss}[i] \neq n + 1$. If $\text{LCE}_r(i, j) \geq (j - i)$, then it holds $\text{LCE}_r(j, j + (j - i)) = \text{LCE}_r(i, j) - (j - i)$ and $\mathsf{nss}[j] = j + (j - i)$.*

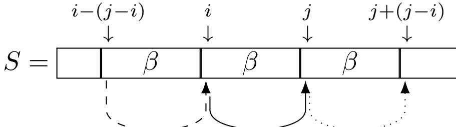
Proof. From $\text{LCE}_r(i, j) \geq (j - i)$ follows $\text{LCE}_r(i, j) = (j - i) + \text{LCE}_r(j, j + (j - i))$, which is equivalent to $\text{LCE}_r(j, j + (j - i)) = \text{LCE}_r(i, j) - (j - i)$. It remains to be shown that $\mathsf{nss}[j] = j + (j - i)$. Due to $\mathsf{nss}[i] = j$ it holds $S_i \succ S_j$. Since $S_i \succ S_j$ and $\text{LCE}_r(i, j) \geq (j - i)$, we have $S_{i+(j-i)} \succ S_{j+(j-i)}$, which implies $\mathsf{nss}[j] \leq j + (j - i)$. Note that $\mathsf{nss}[i] = j$ and Lemma 4 imply that $S[i..j) = S[j..j + (j - i))$ is a Lyndon word. Thus it holds $\lambda[j] \geq (j - i)$, or equivalently $\mathsf{nss}[j] \geq j + (j - i)$. ◀

▶ **Lemma 11.** *Let $i \in [1, n]$ and $j = \mathsf{nss}[i] \neq n + 1$. If $\text{LCE}_\ell(i, j) > (j - i)$, then it holds $\text{LCE}_\ell(i - (j - i), i) = \text{LCE}_\ell(i, j) - (j - i)$ and $\mathsf{nss}[i - (j - i)] = i$.*

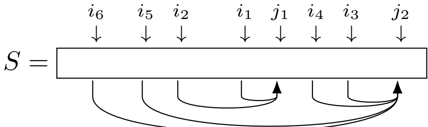
Proof. Analogous to Lemma 10. ◀

4.1 Computing the R-LCEs

First, we will briefly describe our general technique for computing LCEs, and our method of showing the linear time bound. Assume for this purpose that we want to compute $\ell = \text{LCE}_r(i, j)$ with $i < j$. It is easy to see that we can determine ℓ by performing $\ell + 1$ individual character comparisons (by simultaneously scanning the suffixes S_i and S_j from left



(a) Lemmas 10 and 11. The dotted edge follows from $\text{LCE}_r(i, j) \geq (j - i)$ (Lemma 10). The dashed edge follows from $\text{LCE}_\ell(i, j) > (j - i)$ (Lemma 11).



(b) Relative order of R-LCE computations from first to last: $\text{LCE}_r(i_1, j_1)$, $\text{LCE}_r(i_2, j_1)$, $\text{LCE}_r(i_3, j_2)$, $\text{LCE}_r(i_4, j_2)$, $\text{LCE}_r(i_5, j_2)$, $\text{LCE}_r(i_6, j_2)$.

Figure 2 As before, an edge from text position a to text position b indicates $\mathsf{nss}[a] = b$.

to right until we find a mismatch). Whenever we use this naive way of computing an LCE, we *charge* one character comparison to each of the indices from the interval $[j, j + \ell)$. This way, we account for ℓ character comparisons. Since we want to compute $\mathcal{O}(n)$ R-LCE values in $\mathcal{O}(n)$ time, we can afford a constant time overhead (i.e. a constant number of unaccounted character comparisons) for each LCE computation. Thus, there is no need to charge the $(\ell + 1)$-th comparison to any index. At the time at which we want to compute ℓ, we may already know some lower bound $k \leq \ell$. In such cases, we simply skip the first k character comparisons and compute $\ell = k + \text{LCE}_r(i + k, j + k)$. This requires $\ell - k + 1$ character comparisons, of which we charge $\ell - k$ to the interval $[j + k..j + \ell)$.

Ultimately, we will show that all R-LCE values $\text{LCE}_r(i, j)$ with $i \in [1, n]$ and $j = \mathsf{nss}[i] \neq n + 1$ can be computed in a way such that each text position gets charged at most once, which results in the desired linear time bound. From now on, we refer to i as the *left index* and j as the *right index* of the R-LCE computation. Our algorithm computes the R-LCEs in the following order (a visualization is provided in Figure 2b): We consider the possible right indices $j \in [2, n]$ one at a time and in *increasing* order. For each right index j, we then consider the corresponding left indices i with $\mathsf{nss}[i] = j$ in *decreasing* order (we will see how to efficiently deduce this order from the Lyndon array later).

Assume that we are computing the R-LCEs in the previously described order, and let $\ell = \text{LCE}_r(i, j)$ with $j = \mathsf{nss}[i] \neq n + 1$ be the next value that we want to compute. The set of indices that we have already considered as left indices for LCE computations is $I = \{x \mid (\mathsf{nss}[x] < j) \vee ((\mathsf{nss}[x] = j) \wedge (i < x))\}$. For example, when we compute $\text{LCE}_r(i_4, j_2)$ in Figure 2b it holds $\{i_1, i_2, i_3\} \subseteq I$. At this point in time, the rightmost text position that we have already inspected is $\vec{c} = \max_{x \in I}(\mathsf{nss}[x] + \text{LCE}_r(x, \mathsf{nss}[x]))$ if $I \neq \emptyset$, or $\vec{c} = 1$ otherwise. Due to the nature of our charging method, we have not charged any indices from the interval $[\vec{c}, n]$ yet. Thus, in order to show that we can compute all LCEs without charging any index twice, it suffices to show how to compute $\ell = \text{LCE}_r(i, j)$ without charging any index from the interval $[1, \vec{c})$. If $j \geq \vec{c}$ then we naively compute ℓ and charge the character comparisons to the interval $[j, j + \ell)$, thus only charging previously uncharged indices. The new value of $\vec{c}$ is $j + \ell$. If however $j < \vec{c}$, then the computation of ℓ depends on the previously computed LCEs, which we describe in the following.

Let $\ell' = \text{LCE}_r(i', j')$ with $j' = \mathsf{nss}[i']$ be the *most recently* computed R-LCE that satisfies $j' + \ell' = \vec{c}$. Our strategy for computing ℓ depends on the position of i relative to i' and j'. First, note that $i \notin [i', j')$ because otherwise Lemma 5 implies $j \leq j'$, which contradicts our order of computation. This leaves us with three possible cases (as before, a directed edge from text position a to text position b indicates $\mathsf{nss}[a] = b$):

Case R1: $\boldsymbol{i < i'}$ (possibly $j' = j$)

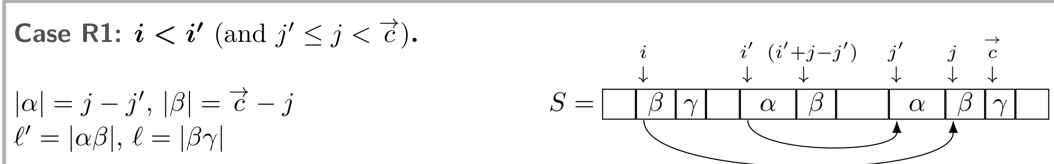
Case R2: $\boldsymbol{i = j'}$

Case R3: $\boldsymbol{i > j'}$

Now we explain the cases in detail. Each case is accompanied by a schematic drawing. We strongly advise the reader to study the drawings alongside the description, since they are essential for an easy understanding of the matter.

Case R1: $\boldsymbol{i < i'}$ (and $j' \le j < \vec{c}$).

$|\alpha| = j - j'$, $|\beta| = \vec{c} - j$
$\ell' = |\alpha\beta|$, $\ell = |\beta\gamma|$



Due to $i < (i' + j - j') < j = \mathsf{nss}[i]$ we have $S_j \prec S_i \prec S_{i'+j-j'}$. From Lemma 1 follows $\vec{c} - j = \text{LCE}_r(i' + j - j', j) \le \text{LCE}_r(i, j) = \ell$, i.e. both S_i and S_j start with β. Since now we know a lower bound $\vec{c} - j \le \ell$ of the desired LCE value, we can skip character comparisons during its computation. Later, we will see that the same bound also holds for most of the other cases. Generally, whenever we can show $\vec{c} - j \le \ell$ we use the following strategy. We compute $\ell = (\vec{c} - j) + \text{LCE}_r(i + (\vec{c} - j), \vec{c})$ using $\ell - (\vec{c} - j) + 1$ character comparisons, of which we charge $\ell - (\vec{c} - j)$ to the interval $[\vec{c}, j + \ell)$. Thus we only charge previously uncharged positions. We continue with $i' \leftarrow i$, $j' \leftarrow j$, $\ell' \leftarrow \ell$, and $\vec{c} \leftarrow j + \ell$.

Case R2: $\boldsymbol{i = j'}$**.** We divide this case into two subcases.

Case R2a: $\boldsymbol{\ell' < j' - i'}$.

$|\alpha| = j - j'$, $|\beta| = \vec{c} - j$

From $j < \vec{c} \implies j - i < \vec{c} - i = \ell'$ and $\ell' < j' - i'$ follows $i' + j - i < j' = i$. Therefore, $\mathsf{nss}[i'] = i$ and Definition 3 imply $S_i \prec S_{i'+j-1}$. Due to $\mathsf{nss}[i] = j$ we also have $S_j \prec S_i$, such that it holds $S_j \prec S_i \prec S_{i'+j-1}$. It is easy to see that $S_{i'+j-i}$ and S_j share a prefix β of length $\text{LCE}_r(i' + j - i, j) = \vec{c} - j$. In fact, also S_i has prefix β because Lemma 1 implies that $\text{LCE}_r(i' + j - i, j) \le \text{LCE}_r(i, j) = \ell$. Thus it holds $\vec{c} - j \le \ell$, which allows us to use the strategy from Case R1.

Case R2b: $\boldsymbol{\ell' \ge j' - i'}$.

$|\beta| = j' - i'$, $\ell = \ell' - |\beta|$

Due to $\ell' \ge j' - i'$, Lemma 10 implies $j = i + (j' - i')$ and $\ell = \ell' - (j' - i')$. Since i', j', and ℓ' are known, we can compute ℓ in constant time without performing any character comparisons. We continue with $i' \leftarrow i$, $j' \leftarrow j$, and $\ell' \leftarrow \ell$ (leaving $\vec{c}$ unchanged).

Case R3: $i > j'$. This is the most complicated case, and it is best explained by dividing it into three subcases. Let $d = j' - i'$, $i'' = i - d$, $j'' = j - d$, and $\ell'' = \text{LCE}_r(i'', j'')$. (In this situation it is implied that $j'' \leq j'$ because otherwise $\ell' = \text{LCE}_r(i', j')$ would not be the *most recently* computed R-LCE that satisfies $j' + \ell' = \vec{c}$. However, since our proof does not rely on this property, we will not explain it in more detail.)

Case R3a: $\mathsf{nss}[i''] \neq j''$:

$|\alpha| = \ell'$, $|\beta| = |\gamma| = \vec{c} - j$
$\ell'' \geq |\beta|$, $\ell \geq |\beta|$

First, note that $S[i'..i' + \ell') = S[j'..\vec{c})$ implies $S[i..j) = S[i''..j'')$. From $\mathsf{nss}[i] = j$ follows that $S[i..j) = S[i''..j'')$ is a Lyndon word. Thus, due to Lemma 4 and $\mathsf{nss}[i''] \neq j''$ it holds $\mathsf{nss}[i''] > j''$, which implies $S_{i''} \prec S_{j''}$. Let $\beta = S[i''..i'' + \vec{c} - j) = S[i..i + \vec{c} - j)$ and let $\gamma = S[j''..i' + \ell') = S[j..\vec{c})$. From $S_{i''} \prec S_{j''}$ follows $\beta \preceq \gamma$, while $S_i \succ S_j$ implies $\beta \succeq \gamma$. Thus it holds $\beta = \gamma$, and therefore $\text{LCE}_r(i, j) \geq |\gamma| = \vec{c} - j$. This means that we can use the strategy from Case R1.

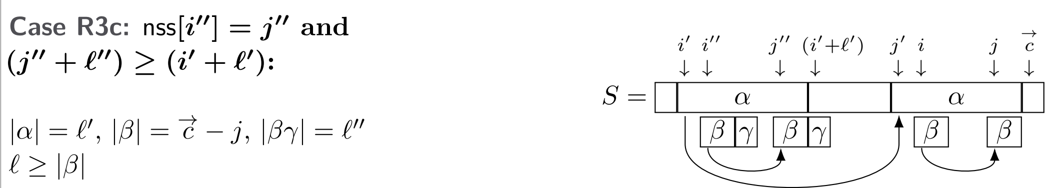
Case R3b: $\mathsf{nss}[i''] = j''$ and $(j'' + \ell'') < (i' + \ell')$:

$|\alpha| = \ell'$, $|\beta| = \ell'' = \ell$

Due to $\ell'' = \text{LCE}_r(i'', j'')$, there is a shared prefix $\beta = S[i''..i''+\ell'') = S[j''..j''+\ell'')$ between $S_{i''}$ and $S_{j''}$, and the first mismatch between the two suffixes is $S[i'' + \ell''] \neq S[j'' + \ell'']$. Because of $(j'' + \ell'') < (i' + \ell')$, both the shared prefix and the mismatch are contained in $S[i'..i' + \ell')$ (i.e. in the first occurrence of α). If we consider the substring $S[j'..j' + \ell')$ instead (i.e. the second occurrence of α), then S_i and S_j clearly also share the prefix $\beta = S[i..i + \ell'') = S[j..j + \ell'')$, with the first mismatch occurring at $S[i + \ell''] \neq S[j + \ell'']$. Thus it holds $\ell = \ell''$. Due to $\mathsf{nss}[i''] = j''$ and our order of R-LCE computations, we have already computed ℓ''. Therefore, we can simply assign $\ell \leftarrow \ell''$ and continue without changing i', j', ℓ', and $\vec{c}$.

Case R3c: $\mathsf{nss}[i''] = j''$ and $(j'' + \ell'') \geq (i' + \ell')$:

$|\alpha| = \ell'$, $|\beta| = \vec{c} - j$, $|\beta\gamma| = \ell''$
$\ell \geq |\beta|$



This situation is similar to Case R3b. There is a shared prefix $\beta = S[i''..i'' + \vec{c} - j) = S[j''..i'+\ell')$ between the suffixes $S_{i''}$ and $S_{j''}$. They may share an even longer prefix $\beta\gamma$, but only the first $|\beta| = \vec{c} - j$ symbols of their LCP are contained in $S[i'..i' + \ell')$ (i.e. in the first occurrence of α). If we consider the substring $S[j'..j'+\ell')$ instead (i.e. the second occurrence of α), then S_i and S_j clearly also share at least the prefix $\beta = S[i..i + \vec{c} - j) = S[j..\vec{c})$. Thus it holds $\vec{c} - j \leq \ell$, and we can use the strategy from Case R1.

We have shown how to compute ℓ without charging any index twice. It follows that the total number of character comparisons for all R-LCEs is $\mathcal{O}(n)$.

A Simple Algorithm for R-LCEs. While the detailed differentiation between the six subcases helps to show the correctness of our approach, it can be implemented in a significantly simpler way (see Algorithm 2). At all times, we keep track of j', $\vec{c}$ and the distance $d = j' - i'$ (line 1). We consider the indices $j \in [2, n]$ in increasing order (line 2). For each index j, we then consider the indices i with $\mathsf{nss}[i] = j$ in decreasing order (line 3). Each time we want to compute an R-LCE value $\ell = \text{LCE}_r(i, j)$, we first check whether Case R3b applies (line 4). If it does, then we simply copy the previously computed R-LCE value $\text{LCE}_r(i-d, j-d)$ (line 5). Otherwise, we either compute the LCE naively (if $j \geq \vec{c}$), or we have to apply the strategy from Case R1 (since all other cases except for Case R2b use this strategy; in Case R2b it holds $\vec{c} - j = \ell$, which means that it can also be solved with the strategy from Case R1). If $j \geq \vec{c}$ then in lines 7–8 we have $k = 0$, and thus we naively compute $\text{LCE}_r(i, j)$ by scanning. If however $j < \vec{c}$, then we have $k = \vec{c} - j$, and we skip k character comparisons. In any case, we update the values j', $\vec{c}$, and d accordingly (line 9).

Algorithm 2 Compute all R-LCEs.

Input: String S of length n and its NSS array nss.
Output: R-LCE value $\text{LCE}_r(i, \mathsf{nss}[i])$ for each index $i \in [1, n]$ with $\mathsf{nss}[i] \neq n + 1$.

```
1: j' ← 0;  c⃗ ← 1;  d ← 0
2: for j ∈ [2, n] in increasing order do
3:     for i with nss[i] = j ≠ n + 1 in decreasing order do
4:         if (   i, j ∈ (j', c⃗)
               ∧ nss[i − d] = j − d
               ∧ j + LCE_r(i − d, j − d) < c⃗  ) then
5:             LCE_r(i, j) ← LCE_r(i − d, j − d)        ▷ retrieve LCE in constant time
6:         else
7:             k ← max(c⃗, j) − j
8:             LCE_r(i, j) ← k + NAIVE-SCAN-LCE_r(i + k, j + k)
9:             j' ← j;  c⃗ ← j + LCE_r(i, j);  d ← j − i
```

The correctness of the algorithm follows from the description of Cases 1–3. Since for each left index i we have to store at most one R-LCE, we can simply maintain the LCEs in a length-n array, where the i-th entry is $\text{LCE}_r(i, \mathsf{nss}[i])$. This way, we use linear space and can access the R-LCE that is required in line 5 in constant time. Apart from the at most n character comparisons that we charge to the indices, we only need a constant number of additional primitive operations per computed R-LCE. The order of iteration can be realized by first generating all $(i, \mathsf{nss}[i])$-pairs, and then using a linear time radix sorter to sort the pairs in increasing order of their second component and decreasing order of their first component. We have shown:

▶ **Lemma 12.** *Given a string of length n and its NSS array nss, we can compute $\text{LCE}_r(i, \mathsf{nss}[i])$ for all indices $i \in [1, n]$ with $\mathsf{nss}[i] \neq n + 1$ in $\mathcal{O}(n)$ time and space.*

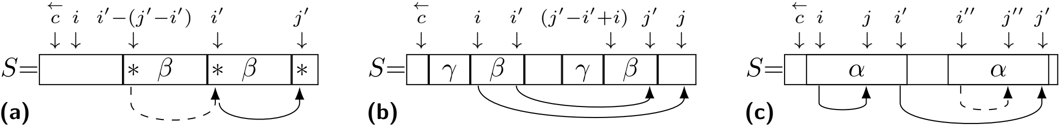


Figure 3 Illustration of the proofs of the three properties in Section 4.2.

4.2 Computing the L-LCEs

Our solution for the L-LCEs is similar to the one for R-LCEs, but differs in subtle details. We generally compute $\ell = \text{LCE}_\ell(i, j)$ by simultaneously scanning the prefixes $S[1..i]$ and $S[1..j]$ from right to left until we find the first mismatch. This takes $\ell + 1$ character comparisons, of which we charge ℓ comparisons to the interval $(i - \ell, i]$. As before, if some lower bound $k \leq \ell$ is known then we skip k character comparisons. In this case, we compute the L-LCE as $\ell = k + \text{LCE}_\ell(i - k, j - k)$, and charge $\ell - k$ comparisons to the interval $(i - \ell, i - k]$.

Again, we will show how to compute all values $\text{LCE}_\ell(i, \mathsf{nss}[i])$ with $i \in [1, n]$ and $\mathsf{nss}[i] \neq n + 1$ such that each index gets charged at most once. In contrast to the more complex R-LCE iteration order, we can simply compute the L-LCE values in *decreasing* order of i. Thus, when we want to compute $\ell = \text{LCE}_\ell(i, j)$ with $j = \mathsf{nss}[i] \neq n + 1$, we have already considered the indices $I = \{x \mid x \in (i, n] \wedge \mathsf{nss}[x] \neq n + 1\}$ as left indices of L-LCE computations. The leftmost text position that we have already inspected so far at this point is $\overleftarrow{c} = \min_{x \in I}(x - \text{LCE}_\ell(x, \mathsf{nss}[x]))$ if $I \neq \emptyset$, or $\overleftarrow{c} = n$ otherwise. Due to our charging method, we have not charged any index from the interval $[1, \overleftarrow{c}]$ yet. Thus, we only have to show how to compute ℓ without charging indices from $(\overleftarrow{c}, n]$. Let $\ell' = \text{LCE}_\ell(i', j')$ be the most recently computed L-LCE that satisfies $i' - \ell' = \overleftarrow{c}$. If $i \leq \overleftarrow{c}$ then we compute ℓ naively and charge the character comparisons to the interval $(i - \ell, i]$ (thus only charging previously uncharged indices). If however $i > \overleftarrow{c}$, then our strategy is more complicated. Before explaining it in detail, we show three important properties that hold in the present situation.

First, we show that $i \geq i' - (j' - i')$. Assume the opposite (as visualized in Figure 3a), then from $\overleftarrow{c} = i' - \ell' < i$ follows $\ell' > j' - i'$. Thus, Lemma 11 implies $\mathsf{nss}[i' - (j' - i')] = i'$ (dashed edge) and $\text{LCE}_\ell(i' - (j' - i'), i') = \ell' - (j' - i')$. Due to our order of computation and $i < i' - (j' - i')$ we must have already computed this L-LCE. However, it holds $i' - (j' - i') - \text{LCE}_\ell(i' - (j' - i'), i') = \overrightarrow{c}$, which contradicts the fact that $\ell' = \text{LCE}_\ell(i', j')$ is the *most recently* computed L-LCE with $i' - \ell' = \overleftarrow{c}$.

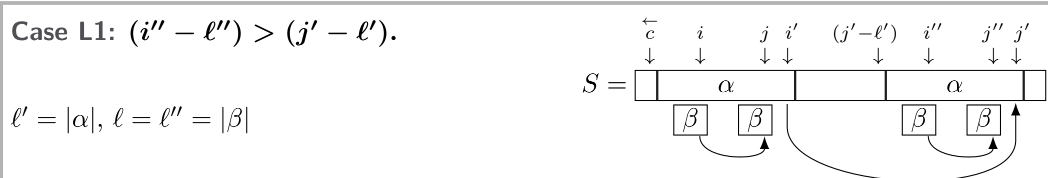
Next, we show that $j \leq i'$. First, note that $j \notin (i', j')$, since due to $i < i'$ we would otherwise contradict Lemma 5. Thus we only have to show $j < j'$. Assume for this purpose that $j \geq j'$ (as visualized in Figure 3b). From $j' - i' + i \in (i, \mathsf{nss}[i])$ and Definition 3 follows $S_i \prec S_{j'-i'+i}$. Because of $\text{LCE}_\ell(i', j') > (i' - i)$ it holds $S[i..i'] = S[j' - i' + i..j'](= \beta)$. Thus $S_i \prec S_{j'-i'+i}$ implies $S_{i'} \prec S_{j'}$, which contradicts the fact that $\mathsf{nss}[i'] = j'$.

Lastly, let $d = j' - i'$, $i'' = i + d$, and $j'' = j + d$ (as visualized in Figure 3c). Now we show that $\mathsf{nss}[i''] = j''$ (dashed edge in the figure). Because of $\alpha = S(\overleftarrow{c}..i'] = S(j' - \ell'..j']$ it holds $S[i..j) = S[i''..j'')$. From $\mathsf{nss}[i] = j$ and Lemma 4 follows that $S[i''..j'')$ is a Lyndon word, and thus $\mathsf{nss}[i''] \geq j''$. We have already shown that $i \geq i' - (j' - i')$, which implies $i'' \geq i'$. Due to $\mathsf{nss}[i'] = j'$ and $i'' \in [i', j')$ it follows from Lemma 5 that $\mathsf{nss}[i''] \leq j'$. Now assume $\mathsf{nss}[i''] \in (j'', j']$, then $S[i''..\mathsf{nss}[i'']) = S[i..j + (\mathsf{nss}[i''] - j''))$ is a Lyndon word, which contradicts the fact that $S[i..j)$ is the longest Lyndon word starting at position i. Thus, we have ruled out all possible values of $\mathsf{nss}[i'']$ except for j''.

Now we show how to compute ℓ. We keep using the definition of i'' and j'' from the previous paragraph. Furthermore, let $\ell'' = \text{LCE}_\ell(i'', j'')$. There are two possible cases.

Case L1: $(i'' - \ell'') > (j' - \ell')$.

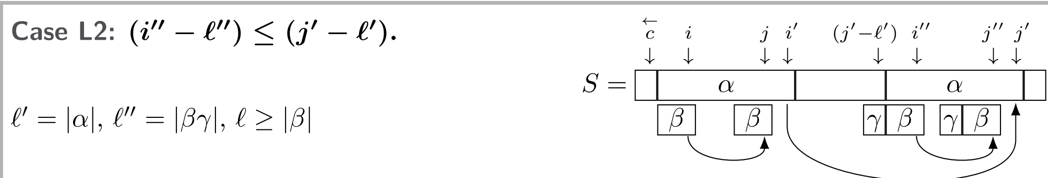
$\ell' = |\alpha|$, $\ell = \ell'' = |\beta|$



Due to $\ell'' = \text{LCE}_\ell(i'', j'')$, the prefixes $S[1..i'']$ and $S[1..j'']$ share the suffix $\beta = S(i'' - \ell''..i''] = S(j'' - \ell''..j'']$, and the first (from the right) mismatch between these prefixes is $S[i'' - \ell''] \neq S[j'' - \ell'']$. Both the shared suffix and the mismatch are contained in $S(j' - \ell'..j']$ (i.e. in the right occurrence of α). If we consider the substring $S(\overleftarrow{c}..i']$ instead (i.e. the left occurrence of α), then $S[1..i]$ and $S[1..j]$ clearly also share the suffix $\beta = S(i - \ell''..i] = S(j - \ell''..j]$, with the first mismatch occurring at $S[i - \ell''] \neq S[j'' - \ell]$. Thus it holds $\ell = \ell''$. Due to $\mathsf{nss}[i''] = j''$ and our order of L-LCE computations, we have already computed ℓ''. Therefore, we can simply assign $\ell \leftarrow \ell''$ and continue without changing i', j', ℓ', and $\overleftarrow{c}$.
(Note that possibly $i'' \neq i' \wedge j'' = j'$. We provide a sketch in Figure 4a.)

Case L2: $(i'' - \ell'') \leq (j' - \ell')$.

$\ell' = |\alpha|$, $\ell'' = |\beta\gamma|$, $\ell \geq |\beta|$



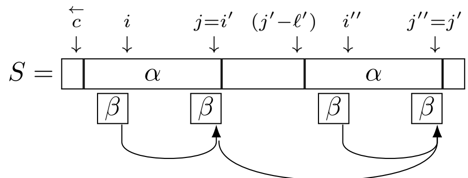
This situation is similar to Case L1. There is a shared suffix $\beta = S(j' - \ell'..i''] = S(j'' - (i - \overleftarrow{c})..j'']$ between the prefixes $S[1..i'']$ and $S[1..j'']$. They may share an even longer suffix $\gamma\beta$, but only the rightmost $|\beta| = i' - \overleftarrow{c}$ symbols of this suffix are contained in $S(j' - \ell'..j']$ (i.e. in the right occurrence of α). If we consider the substring $S(\overleftarrow{c}..i']$ instead (i.e. the left occurrence of α), then $S[1..i]$ and $S[1..j]$ clearly also share the suffix $\beta = S(\overleftarrow{c}..i] = S(j - (i - \overleftarrow{c})..j]$. Thus it holds $i - \overleftarrow{c} \leq \ell$, and we can skip the first $i - \overleftarrow{c}$ character comparisons by computing the LCE as $\ell = (i - \overleftarrow{c}) + \text{LCE}_\ell(\overleftarrow{c}, j + \overleftarrow{c} - i)$. We charge $\ell - (i - \overleftarrow{c})$ character comparisons to the previously uncharged interval $(i - \ell, \overleftarrow{c}]$, and continue with $i' \leftarrow i$, $j' \leftarrow j$, $\ell' \leftarrow \ell$, and $\overleftarrow{c} \leftarrow i - \ell$.
(Note that possibly $i'' \neq i' \wedge j'' = j'$ or even $i'' = i' \wedge j'' = j'$. We provide schematic drawings in Figures 4b and 4c.)

We have shown how to compute ℓ without charging any index twice. It follows that the total number of character comparisons for all LCEs is $\mathcal{O}(n)$. For completeness, we outline a simple implementation of our approach in Algorithm 3. Lines 4–5 correspond to Case L1. If $i \leq \overleftarrow{c}$, then lines 7–9 compute the LCE naively. Otherwise, they correspond to Case L2.

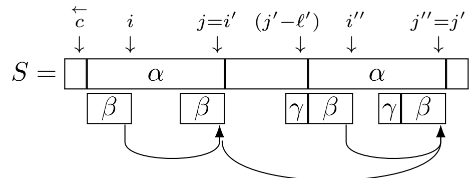
▶ **Lemma 13.** *Given a string of length n and its NSS array* nss*, we can compute $\text{LCE}_\ell(i, \mathsf{nss}[i])$ for all indices $i \in [1, n]$ with $\mathsf{nss}[i] \neq n + 1$ in $\mathcal{O}(n)$ time and space.*

▶ **Corollary 14.** *Given a string of length n over a general ordered alphabet, we can find all runs in the string in $\mathcal{O}(n)$ time and space.*

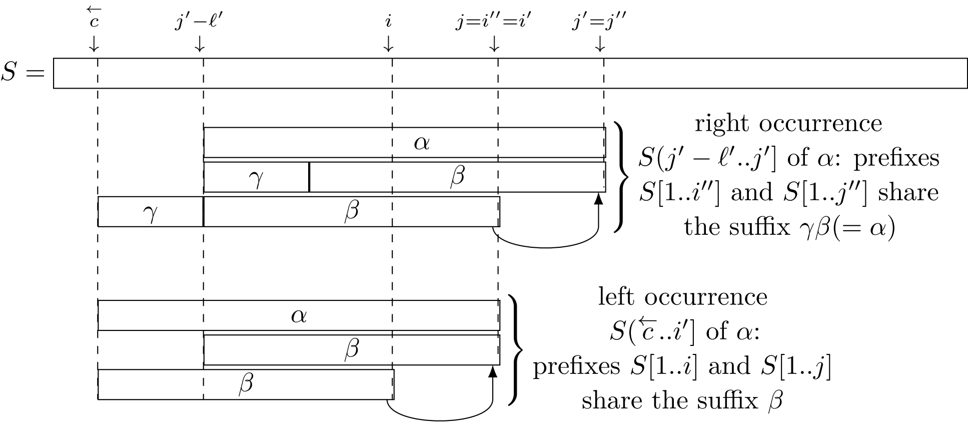
Proof. Computing the increasing runs takes $\mathcal{O}(n)$ time and space due to Lemmas 9, 12, and 13. For decreasing runs, we only have to reverse the order of the alphabet and rerun the algorithm. ◀



(a) Case L1 with $i'' \neq i'$ and $j'' = j'$.



(b) Case L2 with $i'' \neq i'$ and $j'' = j'$.



(c) Case L2 with $i'' = i'$ and $j'' = j'$.

Figure 4 Additional drawings for Cases L1 and L2.

Algorithm 3 Compute all L-LCEs.

Input: String S of length n and its NSS array nss.
Output: L-LCE value $\text{LCE}_\ell(i, \mathsf{nss}[i])$ for each index $i \in [1, n]$ with $\mathsf{nss}[i] \neq n + 1$.

1: $i' \leftarrow 0;\quad \overleftarrow{c} \leftarrow n;\quad d \leftarrow 0$
2: **for** $i \in [1, n]$ **with** $\mathsf{nss}[i] \neq n + 1$ in decreasing order **do**
3: $\quad j \leftarrow \mathsf{nss}[i]$
4: $\quad$ **if** $i \in (\overleftarrow{c}, i') \wedge i - \text{LCE}_\ell(i + d, j + d) > \overleftarrow{c}$ **then**
5: $\quad\quad \text{LCE}_\ell(i, j) \leftarrow \text{LCE}_\ell(i + d, j + d)$ ▷ retrieve LCE in constant time
6: $\quad$ **else**
7: $\quad\quad k \leftarrow i - \min(\overleftarrow{c}, i)$
8: $\quad\quad \text{LCE}_\ell(i, j) \leftarrow k + \text{NAIVE-SCAN-LCE}_\ell(i - k, j - k)$
9: $\quad\quad i' \leftarrow i;\quad \overleftarrow{c} \leftarrow i - \text{LCE}_\ell(i, j);\quad d \leftarrow j - i$

Table 1 Throughput achieved by our runs algorithm using an AMD EPYC 7452 processor. We repeated each experiment five times and use the median throughput as the final result (the minimum and maximum throughputs were almost identical to the median). All numbers are truncated to one decimal place.

Text n in MiB	t_{49} [23] 1077 MiB	sources 201 MiB	pitches 53 MiB	proteins 1024 MiB	dna 385 MiB	english 1024 MiB	xml 282 MiB	ecoli 107 MiB	cere 439 MiB	fib41 255 MiB	rs.13 206 MiB	tm29 256 MiB
runs/100n	94.4	4.7	11.7	7.0	25.3	2.4	3.4	24.4	23.6	76.3	92.7	83.3
MiB/s	15.0	11.4	11.0	10.9	8.8	10.5	12.8	9.0	9.2	15.4	15.1	15.6

5 Practical Implementation

We implemented our algorithm for the runs computation in C++17 and evaluated it by computing all runs on texts from the natural, real repetitive, and artificial repetitive text collections of the Pizza-Chili corpus[2]. Additionally, we used the binary run-rich strings proposed by Matsubara et al. [23] as input. Table 1 shows the throughput that we achieve, i.e. the number of input bytes (or equivalently input symbols) that we process per second. On the string `tm29` we achieve the highest throughput of 15.6 MiB/s. The lowest throughput of 8.8 MiB/s occurs on the text `dna`. Generally, we perform better for run-rich strings.

Lastly, it is noteworthy that our new method of LCE computation leads to a remarkably simple implementation of the runs algorithm. In fact, the entire implementation *including the computation of the NSS array* needs only 250 lines of code. We achieve this by interleaving the computation of the R-LCEs with the computation of the NSS array, which also improves the practical performance. For technical details we refer to the source code, which is publicly available on GitHub[3].

6 Conclusion and Open Questions

We have shown the first linear time algorithm for computing all runs over a general ordered alphabet. The algorithm is also very fast in practice and remarkably easy to implement. It is an open question whether our techniques could be used for the computation of runs on tries, where the best known algorithms require super-linear time even for linearly-sortable alphabets (see e.g. [26]).

References

1 Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):74–93, 1998. doi:10.1006/jcss.1998.1580.

2 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.

3 Philip Bille, Jonas Ellert, Johannes Fischer, Inge Li Gørtz, Florian Kurpicz, J. Ian Munro, and Eva Rotenberg. Space efficient construction of Lyndon arrays in linear time. In *Proceedings of the 47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*, pages 14:1–14:18, Saarbrücken, Germany, July 2020. doi:10.4230/LIPIcs.ICALP.2020.14.

[2] http://pizzachili.dcc.uchile.cl/texts.html, http://pizzachili.dcc.uchile.cl/repcorpus.html
[3] https://github.com/jonas-ellert/linear-time-runs/

4 Dany Breslauer. *Efficient String Algorithmics*. PhD thesis, Columbia University, New York, USA, 1992. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.9146.

5 Helen Budworth and Cynthia T. McMurray. *A Brief History of Triplet Repeat Diseases*, volume 1010 of *Methods in Molecular Biology*, pages 3–17. Springer, 2013. doi:10.1007/978-1-62703-411-1_1.

6 M. Crochemore, C.S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń. Extracting powers and periods in a word from its runs structure. *Theoretical Computer Science*, 521:29–41, 2014. doi:10.1016/j.tcs.2013.11.018.

7 Maxime Crochemore and Lucian Ilie. Maximal repetitions in strings. *Journal of Computer and System Sciences*, 74(5):796–807, 2008. doi:10.1016/j.jcss.2007.09.003.

8 Maxime Crochemore, Lucian Ilie, and Wojciech Rytter. Repetitions in strings: Algorithms and combinatorics. *Theoretical Computer Science*, 410(50):5227–5235, 2009. doi:10.1016/j.tcs.2009.08.024.

9 Maxime Crochemore, Lucian Ilie, and Liviu Tinta. The “runs” conjecture. *Theoretical Computer Science*, 412(27):2931–2941, 2011. doi:10.1016/j.tcs.2010.06.019.

10 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Ritu Kundu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Near-optimal computation of runs over general alphabet via non-crossing lce queries. In *Proceedings of the 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016)*, pages 22–34, Beppu, Japan, October 2016. doi:10.1007/978-3-319-46049-9_3.

11 Johannes Fischer and Volker Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006)*, pages 36–48, Barcelona, Spain, 2006. doi:10.1007/11780441_5.

12 Johannes Fischer, Stepan Holub, Tomohiro I, and Moshe Lewenstein. Beyond the runs theorem. In Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz, editors, *String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings*, volume 9309 of *Lecture Notes in Computer Science*, pages 277–286. Springer, 2015. doi:10.1007/978-3-319-23826-5_27.

13 Frantisek Franek, A. S. M. Sohidull Islam, Mohammad Sohel Rahman, and William F. Smyth. Algorithms to compute the Lyndon array. In *Proceedings of the Prague Stringology Conference 2016 (PSC 2016)*, pages 172–184, Prague, Czech Republic, 2016. URL: http://www.stringology.org/event/2016/p15.html.

14 Pawel Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Walen. Faster longest common extension queries in strings over general alphabets. In *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, pages 5:1–5:13, Tel Aviv, Israel, 2016. doi:10.4230/LIPIcs.CPM.2016.5.

15 Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS 98)*, pages 366–398, Paris, France, February 1998. doi:10.1007/BFb0028575.

16 Yijie Han and M. Thorup. Integer sorting in $\mathcal{O}(n\sqrt{\log\log n})$ expected time and linear space. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2002)*, pages 135–144, Vancouver, Canada, 2002. doi:10.1109/SFCS.2002.1181890.

17 Stepan Holub. Prefix frequency of lost positions. *Theor. Comput. Sci.*, 684:43–52, 2017. doi:10.1016/j.tcs.2017.01.026.

18 R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS 1999)*, pages 596–604, New York, NY, USA, 1999. doi:10.1109/SFFCS.1999.814634.

19 Dmitry Kosolobov. Lempel-Ziv factorization may be harder than computing all runs. In *Proceedings of the 32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, pages 582–593, Munich, Germany, 2015. doi:10.4230/LIPIcs.STACS.2015.582.

20 Dmitry Kosolobov. Computing runs on a general alphabet. *Information Processing Letters*, 116(3):241–244, 2016. doi:10.1016/j.ipl.2015.11.016.

21 R. C. Lyndon and M. P. Schützenberger. The equation $a^m = b^n c^p$ in a free group. *Michigan Mathematical Journal*, 9(4):289–298, 1962. doi:10.1307/mmj/1028998766.

22 Michael G Main and Richard J Lorentz. An o(n log n) algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984. doi:10.1016/0196-6774(84)90021-X.

23 Wataru Matsubara, Kazuhiko Kusano, Hideo Bannai, and Ayumi Shinohara. A series of run-rich strings. In Adrian Horia Dediu, Armand Mihai Ionescu, and Carlos Martín-Vide, editors, *Proceedings of the 3rd International Conference on Language and Automata Theory and Applications (LATA 2009)*, pages 578–587, Tarragona, Spain, 2009. doi:10.1007/978-3-642-00982-2_49.

24 Simon J. Puglisi, Jamie Simpson, and W.F. Smyth. How many runs can a string contain? *Theoretical Computer Science*, 401(1):165–171, 2008. doi:10.1016/j.tcs.2008.04.020.

25 Wojciech Rytter. The number of runs in a string: Improved analysis of the linear upper bound. In *Proceedings of the 24th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2006)*, pages 184–195, Marseille, France, 2006. doi:10.1007/11672142_14.

26 Ryo Sugahara, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing runs on a trie. In *Proceedings of the 30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*, volume 128, pages 23:1–23:11, Pisa, Italy, June 2019. doi:10.4230/LIPIcs.CPM.2019.23.