Near-Optimal Computation of Runs over General Alphabet via Non-Crossing LCE Queries

Maxime Crochemore[1](✉), Costas S. Iliopoulos[1], Tomasz Kociumaka[2], Ritu Kundu[1], Solon P. Pissis[1], Jakub Radoszewski[1,2], Wojciech Rytter[2], and Tomasz Waleń[2]

[1] Department of Informatics, King's College London, London, UK
{maxime.crochemore,costas.iliopoulos,ritu.kundu,solon.pissis}@kcl.ac.uk

[2] Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Warsaw, Poland
{kociumaka,jrad,rytter,walen}@mimuw.edu.pl

Abstract. Longest common extension queries (LCE queries) and runs are ubiquitous in algorithmic stringology. Linear-time algorithms computing runs and preprocessing for constant-time LCE queries have been known for over a decade. However, these algorithms assume a linearly-sortable integer alphabet. A recent breakthrough paper by Bannai et al. (SODA 2015) showed a link between the two notions: all the runs in a string can be computed via a linear number of LCE queries. The first to consider these problems over a general ordered alphabet was Kosolobov (*Inf. Process. Lett.*, 2016), who presented an $\mathcal{O}(n(\log n)^{2/3})$-time algorithm for answering $\mathcal{O}(n)$ LCE queries. This result was improved by Gawrychowski et al. (CPM 2016) to $\mathcal{O}(n \log\log n)$ time. In this work we note a special *non-crossing* property of LCE queries asked in the runs computation. We show that any n such non-crossing queries can be answered on-line in $\mathcal{O}(n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann function, which yields an $\mathcal{O}(n\alpha(n))$-time algorithm for computing runs.

1 Introduction

Runs (also called *maximal repetitions*) are a fundamental type of repetitions in a string as they represent the structure of all repetitions in a string in a succinct way. A run is an inclusion-maximal periodic factor of a string in which the shortest period repeats at least twice. A crucial property of runs is that their maximal number in a string of length n is $\mathcal{O}(n)$. This fact was already observed by Kolpakov and Kucherov [15,16] who conjectured that this number

T. Kociumaka—Supported by Polish budget funds for science in 2013–2017 as a research project under the 'Diamond Grant' program.

J. Radoszewski—Newton International Fellow.

W. Rytter and T. Waleń—Supported by the Polish National Science Center, grant no. 2014/13/B/ST6/00770.

S. Inenaga et al. (Eds.): SPIRE 2016, LNCS 9954, pp. 22–34, 2016.
DOI: 10.1007/978-3-319-46049-9_3

is actually smaller than n, which was known as the runs conjecture. Due to the works of several authors [6–8,12,19–21] more precise bounds on the number of runs have been obtained, and finally in a recent breakthrough paper [2] Bannai et al. proved the runs conjecture, which has since then become the runs theorem (even more recently in [10] the upper bound of $0.957n$ was shown for binary strings).

Perhaps more important than the combinatorial bounds is the fact that the set of all runs in a string can be computed efficiently. Namely, in the case of a linearly-sortable alphabet Σ (e.g., $\Sigma = \{1, \dots, \sigma\}$ with $\sigma = n^{\mathcal{O}(1)}$) a linear-time algorithm based on Lempel-Ziv factorization [15,16] was known for a long time. In the recent papers of Bannai et al. [1,2] it is shown that to compute the set of all runs in a string, it suffices to answer $\mathcal{O}(n)$ longest common extension (LCE) queries. An LCE query asks, for a pair of suffixes of a string, for the length of their longest common prefix. In the case of $\sigma = n^{\mathcal{O}(1)}$ such queries can be answered on-line in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time preprocessing that consists of computing the suffix array with its inverse, the LCP table and a data structure for range minimum queries on the LCP table; see e.g. [5]. The algorithms from [1,2] use (explicitly and implicitly, respectively) an intermediate notion of Lyndon tree (see [3,13]) which can, however, also be computed using LCE queries.

Let $T_{\mathrm{LCE}}(n)$ denote the time required to answer on-line n LCE queries in a string. In a very recent line of research, Kosolobov [17] showed that, for a general ordered alphabet, $T_{\mathrm{LCE}}(n) = \mathcal{O}(n(\log n)^{2/3})$, which immediately leads to $\mathcal{O}(n(\log n)^{2/3})$-time computation of the set of runs in a string. In [11] a faster, $\mathcal{O}(n \log \log n)$-time algorithm for answering n LCE queries has been presented which automatically leads to $\mathcal{O}(n \log \log n)$-time computation of runs.

Runs have found a number of algorithmic applications. Knowing the set of runs in a string of length n one can compute in $\mathcal{O}(n)$ time all the local periods and the number of all squares, and also in $\mathcal{O}(n + T_{\mathrm{LCE}}(n))$ time all distinct squares provided that the suffix array of the string is known [9]. Runs were also used in a recent contribution on efficient answering of internal pattern matching queries and their applications [14].

Our Results. We observe that the computation of a Lyndon tree of a string and furthermore the computation of all the runs in a string can be reduced to answering $\mathcal{O}(n)$ LCE queries that are *non-crossing*, i.e., no two queries $\mathrm{LCE}(i, j)$ and $\mathrm{LCE}(i', j')$ are asked with $i < i' < j < j'$ or $i' < i < j' < j$. Let $T_{\mathrm{ncLCE}}(n)$ denote the time required to answer n such queries on-line in a string of length n over a general ordered alphabet. We show that $T_{\mathrm{ncLCE}}(n) = \mathcal{O}(n\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function. As a consequence, we obtain $\mathcal{O}(n\alpha(n))$-time algorithms for computing the Lyndon tree, the set of all runs, the local periods and the number of all squares in a string over a general ordered alphabet.

Our solution relies on a trade-off between two approaches. The results of [11] let us efficiently compute the LCEs if they are short, while LCE queries with similar arguments and a large answer yield structural properties of the string, which we discover and exploit to answer further such queries.

Our approach for answering non-crossing LCE queries is described in three sections: in Sect. 3 we give an overview of the data structure, in Sect. 4 we present the details of the implementation, and in Sect. 5 we analyse the complexity of answering the queries. The applications including runs computation are detailed in Sect. 6.

2 Preliminaries

Strings. Let Σ be a finite ordered alphabet of size σ. A string w of length $|w| = n$ is a sequence of letters $w[1] \dots w[n]$ from Σ. By $w[i, j]$ we denote the *factor* of w being a string of the form $w[i] \dots w[j]$. A factor $w[i, j]$ is called proper if $w[i, j] \neq w$. A factor is called a *prefix* if $i = 1$ and a *suffix* if $j = n$. We say that p is a *period* of w if $w[i] = w[i + p]$ for all $i = 1, \dots, n - p$. If p is a period of w, the prefix $w[1, p]$ is called a *string period* of w.

By an interval $[\ell, r]$ we mean the set of integers $\{\ell, \dots, r\}$. If w is a string of length n, then an interval $[a, b]$ is called a *run* in w if $1 \leq a < b \leq n$, the shortest period p of $w[a, b]$ satisfies $2p \leq b - a + 1$ and none of the factors $w[a - 1, b]$ and $w[a, b + 1]$ (if it exists) has the period p. An example of a run is shown in Fig. 1.

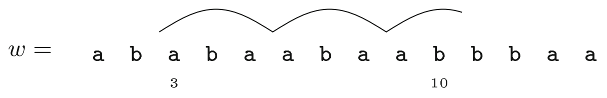

Fig. 1. Example of a run [3, 10] with period 3 in the string $w =$ `ababaabaabbbaa`. This string contains also other runs, e.g. $[10, 12]$ with period 1 and $[1, 5]$ with period 2.

Lyndon Words and Trees. By $\prec = \prec_0$ we denote the order on Σ and by $\prec_1$ we denote the reverse order on Σ. We extend each of the orders $\prec_r$ for $r \in \{0, 1\}$ to a lexicographical order on strings over Σ. A string w is called an *r-Lyndon word* if $w \prec_r u$ for every non-empty proper suffix u of w. The *standard factorization* of an r-Lyndon word w is a pair (u, v) of r-Lyndon words such that $w = uv$ and v is the longest proper suffix of w that is an r-Lyndon word.

The *r-Lyndon tree* of an r-Lyndon word w, denoted as $LTree_r(w)$, is a rooted full binary tree defined recursively on $w[1, n]$ as follows:

– $LTree_r(w[i, i])$ consists of a single node labeled with $[i, i]$
– if $j - i > 1$ and (u, v) is the standard factorization of $w[i, j]$, then the root of $LTree_r(w)$ is labeled by $[i, j]$, has left child $LTree_r(u)$ and right child $LTree_r(v)$.

See Fig. 2 for an example. We can also define the r-Lyndon tree of an arbitrary string. Let $\$_0, \$_1$ be special characters smaller than and greater than all the letters from Σ, respectively. We then define $LTree_r(w)$ as $LTree_r(\$_r w)$; note that $\$_r w$ is an r-Lyndon word.

LCE Queries. For two strings u and v, by $\text{lcp}(u, v)$ we denote the length of their longest common prefix. Let w be a string of length n. An LCE query $\text{LCE}(i, j)$

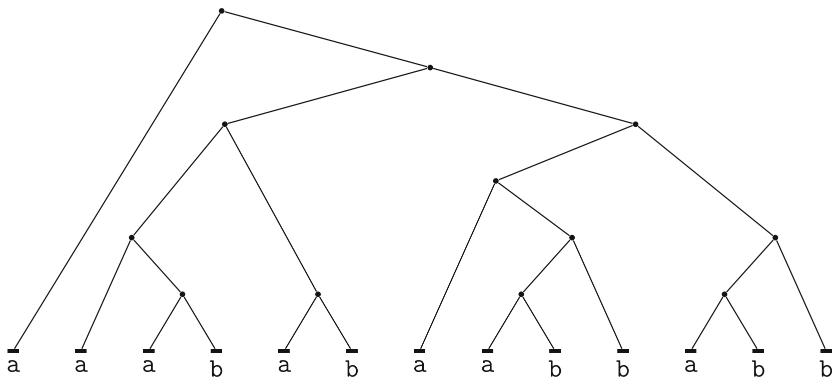

Fig. 2. The Lyndon tree $LTree_0(w)$ of a Lyndon word $w = \texttt{aaababaabbabb}$.

computes $\mathrm{lcp}(w[i,n], w[j,n])$. An ℓ-limited LCE query Limited-LCE$_{\leq \ell}(i,j)$ computes $\min(\mathrm{LCE}(i,j), \ell)$. Such queries can be answered efficiently as follows; see Lemma 14 in [11].

Lemma 1 ([11]). *A sequence of q queries* Limited-LCE$_{\leq \ell_p}(i_p, j_p)$ *can be answered on-line in $\mathcal{O}((n + \sum_{p=1}^{q} \log \ell_p)\alpha(n))$ time over a general ordered alphabet.*

The following observation shows a relation between LCE queries and periods in a string that we use in our data structure; for an illustration see Fig. 3.

Observation 2. *Assume that the factors $w[a, d_A - 1]$ and $w[b, d_B - 1]$ have the same string period, but neither $w[a, d_A]$ nor $w[b, d_B]$ has this string period. Then*

$$\mathrm{LCE}(a,b) = \begin{cases} \min(d_A - a, d_B - b) & \textit{if } d_A - a \neq d_B - b, \\ d_A - a + \mathrm{LCE}(d_A, d_B) & \textit{otherwise.} \end{cases}$$

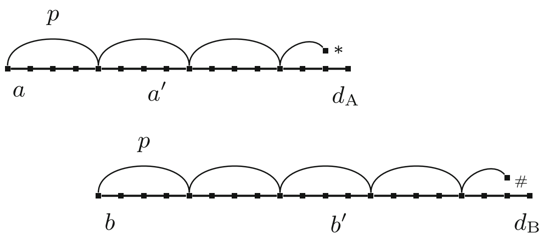

Fig. 3. In this example figure $d_A - a = 14$, $d_B - b = 18$, and $p = 4$. We have $\mathrm{LCE}(a,b) = 14$ and $\mathrm{LCE}(a', b') = 8 + \mathrm{LCE}(d_A, d_B)$.

Non-Crossing Pairs. For a positive integer n, we define the set of pairs

$$P_n = \{(a,b) \in \mathbb{Z}^2 : 1 \leq a \leq b \leq n\}.$$

Pairs (a, b) and (a', b') are called *crossing* if $a < a' < b < b'$ or $a' < a < b' < b$. A subset $S \subseteq P_n$ is called *non-crossing* if it does not contain crossing pairs.

A graph G is called *outerplanar* if it can be drawn on a plane without crossings in such a way that all vertices belong to the unbounded face. An outerplanar graph on n vertices has less than $2n$ edges (at most $2n - 3$ for $n \geq 2$).

Fact 3. *A non-crossing set of pairs $S \subseteq P_n$ has less than $3n$ elements.*

Proof. We associate $S \backslash \{(a, a) : 1 \leq a \leq n\}$ with a plane graph on vertices $\{1, \ldots, n\}$ drawn on a circle in this order, and edges represented as straight-line segments. The non-crossing property of pairs implies that these segments do not intersect. Thus, the graph drawing is outerplanar, and therefore the number of edges is less than $2n$. Accounting for the pairs of the form (a, a), we get the claimed upper bound. □

For a set of pairs $S = \{(a_i, b_i) : 1 \leq i \leq k\}$ and a positive integer t, by $\lceil S/t \rceil$ we denote the set $\{(\lceil \frac{a_i}{t} \rceil, \lceil \frac{b_i}{t} \rceil) : 1 \leq i \leq k\}$.

Observation 4. *If S is non-crossing, then $\lceil S/t \rceil$ is also non-crossing.*

3 High-Level Description of the Data Structure

We say that a sequence of LCE(a, b) queries, for $a \leq b$, is *non-crossing* if the underlying collection of pairs (a, b) is non-crossing. In this section, we give an overview of our data structure, which answers a sequence of q non-crossing LCE queries on-line in $\mathcal{O}(q + n \cdot \alpha(n))$ total time.

The data structure is composed of $\lceil \log n \rceil$ levels. Function LCE$^{(i)}(a, b)$ corresponds to the level i and returns LCE(a, b). In the computation it may make calls to LCE$^{(i+1)}(a, b)$. However, we make sure that the total number of such calls is bounded. Each original LCE(a, b) query is first asked at the level 0.

The implementation of LCE$^{(i)}(a, b)$ consists of two phases. If LCE$(a, b) \geq 3 \cdot 2^i$, then this LCE$^{(i)}$ query is called *relevant*; otherwise it is called *short*. In the first phase, we check the type of the query via a Limited-LCE$_{\leq 3 \cdot 2^i}(a, b)$ query. This lets us immediately answer short queries. In the second phase, we know that the query is relevant, and we try to deduce the answer based on data gathered while processing *similar* queries or to learn some information useful for answering future *similar* queries by asking LCE$^{(i+1)}$ queries.

We shall say that LCE$^{(i)}$ queries for (a, b) and (a', b') are *similar* if $\lceil \frac{a}{2^i} \rceil = \lceil \frac{a'}{2^i} \rceil$ and $\lceil \frac{b}{2^i} \rceil = \lceil \frac{b'}{2^i} \rceil$. Each equivalence class of this relation is processed by an independent component, called a *block-pair*. A *block* at level i is an interval of the form $[x \cdot 2^i + 1, (x + 1) \cdot 2^i]$, and a *block-pair* is a data structure identified by a pair (A, B) of blocks. If a relevant LCE$^{(i)}(a, b)$ query satisfies $a \in A$ and $b \in B$ for some block-pair (A, B), we say that the block-pair is *responsible* for the query or that the query *concerns* the block-pair. As we show in Sect. 5, the pairs of interval right endpoints of block-pairs at each level are non-crossing (whereas LCE$^{(i)}$ queries that will be asked for $i \geq 1$ are non necessarily non-crossing).

The implementation of a block-pair, summarized in the lemma below, is given in Sect. 4.

Lemma 5. *Consider a sequence of relevant* $\mathrm{LCE}^{(i)}$ *queries concerning a block-pair* (A, B). *The block-pair can answer these queries on-line in worst-case constant time plus the time to answer at most four* $\mathrm{LCE}^{(i+1)}(a, b)$ *queries, such that each either corresponds to the currently processed* $\mathrm{LCE}^{(i)}$ *query or satisfies* $a < b \le a + 2^{i+1}$.

Structural conditions stated in Lemma 5 let us characterize the set of queries passed to the next level. The complexity analysis in Sect. 5 relies on this characterization.

4 Block-Pair Implementation

Our aim in this section is to prove Lemma 5. Information stored by a block-pair changes through the course of the algorithm, and the implementation of the query algorithm depends on what is currently stored. We distinguish four states of a block pair (A, B) at level i. Figure 4 illustrates two of the states.

state(A, B)	*description*
initial	No auxiliary data is stored
visited(a_0, b_0, L)	$a_0 \in A$ and $b_0 \in B$ are the arguments of the first query that concerns this block pair, $L = \mathrm{LCE}(a_0, b_0) \ge 3 \cdot 2^i$
full$(d_\mathrm{A}, d_\mathrm{B})$	$\exists_{p \in [1, 2^{i+1}]}$: $w[\max A, d_\mathrm{A} - 1]$ and $w[\max B, d_\mathrm{B} - 1]$ have common period p and length at least $p + 2^i$, but neither $w[\max A, d_\mathrm{A}]$ nor $w[\max B, d_\mathrm{B}]$ has period p
full$^+(d_\mathrm{A}, d_\mathrm{B}, L')$	As in **full**$(d_\mathrm{A}, d_\mathrm{B})$ plus $L' = \mathrm{LCE}(d_\mathrm{A}, d_\mathrm{B})$.

4.1 Initial State

In this state, we simply forward the query to the level $i + 1$, return the obtained $\mathrm{LCE}(a, b)$ value, and change the state to **visited**$(a, b, \mathrm{LCE}(a, b))$.

Algorithm 1. Initial-$\mathrm{LCE}^{(i)}_{(A,B)}(a, b)$

```
Require: $\mathrm{LCE}^{(i)}(a, b)$ concerns $(A, B)$, whose state is initial
$L \leftarrow \mathrm{LCE}^{(i+1)}(a, b)$;                    ▷ higher level call
transform $(A, B)$ to state visited$(a, b, L)$;
return $L$;
```

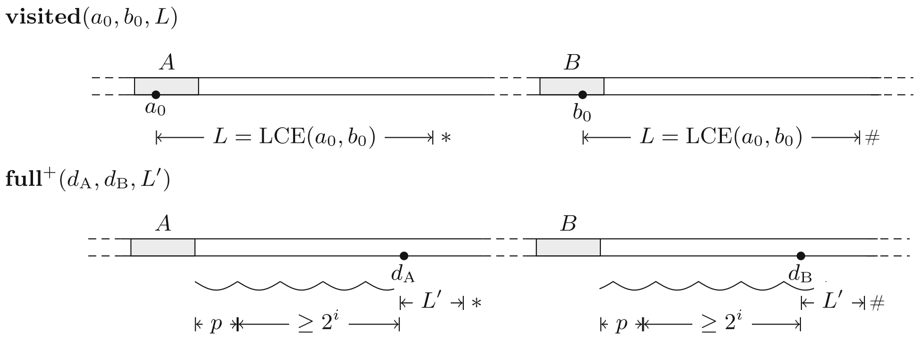

Fig. 4. Block-pair (A, B) in states **visited**(a_0, b_0, L) and **full**$^+(d_\mathrm{A}, d_\mathrm{B}, L')$.

4.2 Visited State

In state **visited**(a_0, b_0, L), we can immediately determine $\mathrm{LCE}(a, b)$ if (a, b) is a shift of (a_0, b_0). Otherwise, we apply Lemma 6 to move to state **full**.

Lemma 6. *Let* $\mathrm{LCE}^{(i)}(a, b)$, $\mathrm{LCE}^{(i)}(a', b')$ *be similar and relevant queries and let* $p = |(b - b') - (a - a')|$*. If* $p \neq 0$ *and* $b' \leq b$*, then* $\mathrm{LCE}(a, a + p) \geq 2^{i+1}$*, i.e., p is a (not necessarily shortest) period of the factor* $w[a, a + 2^{i+1} + p - 1]$*.*

Proof. We shall first prove that $\mathrm{LCE}(a, a + q) \geq 3 \cdot 2^i - (b - b')$ where $q = (b - b') - (a - a')$. First, observe that $a + q = a' + (b - b')$, and thus $\mathrm{LCE}(a + q, b) = \mathrm{LCE}(a' + (b - b'), b' + (b - b')) \geq 3 \cdot 2^i - (b - b')$ because $\mathrm{LCE}^{(i)}(a', b')$ is relevant. Since $\mathrm{LCE}^{(i)}(a, b)$ is also relevant, we have $\mathrm{LCE}(a, b) \geq 3 \cdot 2^i \geq 3 \cdot 2^i - (b - b')$. Combining these two inequalities, we immediately get $\mathrm{LCE}(a, a + q) \geq \min(\mathrm{LCE}(a, b), \mathrm{LCE}(a + q, b)) \geq 3 \cdot 2^i - (b - b')$, as claimed.

If $q > 0$, we have $q = p$, and thus $\mathrm{LCE}(a, a+p) \geq 3 \cdot 2^i - (b-b')$. Since the two $\mathrm{LCE}^{(i)}$ queries are similar, we have $3 \cdot 2^i - (b-b') \geq 2^{i+1}$, so $\mathrm{LCE}(a, a+p) \geq 2^{i+1}$. See Fig. 5 for an illustration of this case.

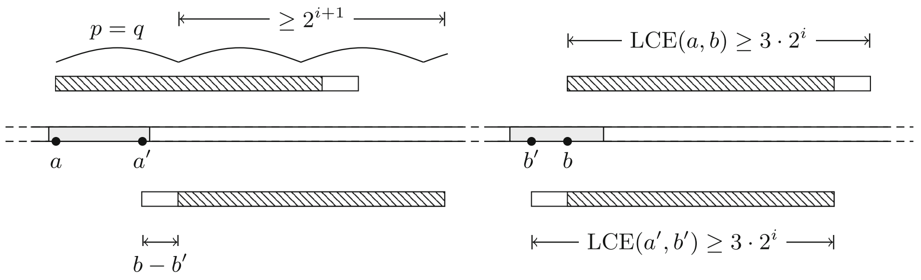
$p = q$
$\geq 2^{i+1}$
$\mathrm{LCE}(a, b) \geq 3 \cdot 2^i$
a
a'
b'
b
$b - b'$
$\mathrm{LCE}(a', b') \geq 3 \cdot 2^i$

Fig. 5. Illustration of Lemma 6: case $q > 0$. We assume that $\mathrm{LCE}(a+q, b) \leq \mathrm{LCE}(a, b)$. The marked fragments correspond to $\mathrm{LCE}(a, a + q) = \mathrm{LCE}(a + q, b)$.

Otherwise, $q = -p$, and we have $\mathrm{LCE}(a, a-p) \geq 3 \cdot 2^i - (b - b')$, which implies $\mathrm{LCE}(a+p, a) \geq 3 \cdot 2^i - (b-b') + q = 3 \cdot 2^i - (a - a')$. Again, the fact that the queries are similar yields $3 \cdot 2^i - (a - a') \geq 2^{i+1}$, and consequently $\mathrm{LCE}(a, a+p) \geq 2^{i+1}$. □

In the query algorithm, we first check if $a - a_0 = b - b_0$. If so, let us denote the common value by Δ. Note that $|\Delta| \leq 2^i$, $\mathrm{LCE}(a,b) \geq 3 \cdot 2^i$, and $\mathrm{LCE}(a_0, b_0) \geq 3 \cdot 2^i$. This clearly yields $\mathrm{LCE}(a,b) = \mathrm{LCE}(a_0, b_0) + \Delta$, which lets us compute the result in constant time.

Algorithm 2. Visited-$\mathrm{LCE}^{(i)}_{(A,B)}(a,b)$

```
Require: $\mathrm{LCE}^{(i)}(a,b)$ concerns $(A,B)$, whose state is visited$(a_0, b_0, L)$
if $a - a_0 = b - b_0$ then
    return $L + a - a_0$;
else
    $p \leftarrow |(a - a_0) - (b - b_0)|$;
    $a' \leftarrow \max A$;  $b' \leftarrow \max B$;
    $d_A \leftarrow a' + p + \mathrm{LCE}^{(i+1)}(a', a'+p)$;          ▷ higher level call
    $d_B \leftarrow b' + p + \mathrm{LCE}^{(i+1)}(b', b'+p)$;          ▷ higher level call
    transform $(A,B)$ to state full$(d_A, d_B)$;
    return Full-$\mathrm{LCE}^{(i)}_{(A,B)}(a,b)$;          ▷ recursive call on state full
```

Otherwise, our aim is to change the state of the block-pair to **full**. Lemma 6 lets us deduce that $\mathrm{LCE}(\bar{a}, \bar{a}+p) \geq 2^{i+1}$ for some $\bar{a} \in \{a, a_0\}$ and (by symmetry) $\mathrm{LCE}(\bar{b}, \bar{b}+p) \geq 2^{i+1}$ for some $\bar{b} \in \{b, b_0\}$, where $p = |(a-a_0) - (b - b_0)|$ ($\bar{a}$ and $\bar{b}$ depend on the relative order of b, b_0 and a, a_0, respectively). Let $a' = \max A$ and $b' = \max B$. We have $\mathrm{LCE}(a', a'+p) \geq 2^i$ and $\mathrm{LCE}(b', b'+p) \geq 2^i$ because $a' - 2^i < \bar{a} \leq a'$ and $b' - 2^i < \bar{b} \leq b'$. Such a situation allows for a move to state **full**. The exact values of d_A and d_B are computed using a higher level call, which lets us determine $\mathrm{LCE}(a', a'+p)$ and $\mathrm{LCE}(b', b'+p)$. Note that $p \leq 2^{i+1}$ implies that these queries satisfy the condition of Lemma 5. The answer to the initial $\mathrm{LCE}^{(i)}(a,b)$ query is computed by the routine for state **full**, which we give below.

4.3 Full States

In state **full**$^+$ we can answer every relevant query in constant time. In state **full** we can either answer the query in constant time or make the final query at level $i+1$ to transform the state to **full**$^+$; see the following lemma.

Lemma 7. *Consider a relevant* $\mathrm{LCE}^{(i)}(a,b)$ *query concerning a block-pair* (A,B) *in state* **full**(d_A, d_B) *or* **full**$^+(d_A, d_B, L')$*. Then*

$$\mathrm{LCE}(a,b) = \begin{cases} \min(d_A - a, d_B - b) & \text{if } d_A - a \neq d_B - b, \\ d_A - a + \mathrm{LCE}(d_A, d_B) & \text{otherwise.} \end{cases}$$

Proof. Let $a_0 = \max A$, $b_0 = \max B$ and let p be the witness period of the state of (A, B). Let us define $\Delta = \max(a_0 - a, b_0 - b)$, $a' = a + \Delta$, and $b' = b + \Delta$. Observe that $\Delta \le 2^i$, $a_0 \le a' \le a_0 + 2^i$, and $b_0 \le b' \le b_0 + 2^i$. The fact that the query is relevant yields $\mathrm{LCE}(a, b) \ge 3 \cdot 2^i \ge p + \Delta$, so $\mathrm{LCE}(a, b) = \Delta + \mathrm{LCE}(a', b')$ and $\mathrm{LCE}(a', b') \ge p$. Moreover, $d_A \ge p + 2^i + a_0$ and $d_B \ge p + 2^i + b_0$ implies that fragments $w[a', d_A - 1]$ and $w[b', d_B - 1]$ have length at least p, and thus they are right-maximal with period p. Consequently, the fragments $w[a', d_A - 1]$ and $w[b', d_B - 1]$ have the same string period of length p. This lets us apply Observation 2, which gives

$$\mathrm{LCE}(a', b') = \begin{cases} \min(d_A - a', d_B - b') & \text{if } d_A - a' \ne d_B - b', \\ d_A - a' + \mathrm{LCE}(d_A, d_B) & \text{otherwise.} \end{cases}$$

Since $a' = a + \Delta$, $b' = b + \Delta$, and $\mathrm{LCE}(a, b) = \Delta + \mathrm{LCE}(a', b')$, this is clearly equivalent to the claimed formula for $\mathrm{LCE}(a, b)$. □

Algorithm 3. Full-$\mathrm{LCE}^{(i)}_{(A,B)}(a, b)$

```
Require: $\mathrm{LCE}^{(i)}(a, b)$ concerns $(A, B)$, whose state is **full**$(d_A, d_B)$ or
         **full**$^+(d_A, d_B, L')$

if $d_A - a \ne d_B - b$ then
    return $\min(d_A - a, d_B - b)$;
else
    if $(A, B)$ is in state full$(d_A, d_B)$ then
        $L' \leftarrow \mathrm{LCE}^{(i+1)}(a, b) - (d_A - a)$;        ▷ higher level call
        transform $(A, B)$ to state full$^+(d_A, d_B, L')$;
    return $d_A - a + L'$;
```

4.4 Proof of Lemma 5

Lemma 5. *Consider a sequence of relevant* $\mathrm{LCE}^{(i)}$ *queries concerning a block-pair (A,B). The block-pair can answer these queries on-line in worst-case constant time plus the time to answer at most four* $\mathrm{LCE}^{(i+1)}(a, b)$ *queries, such that each either corresponds to the currently processed* $\mathrm{LCE}^{(i)}$ *query or satisfies* $a < b \le a + 2^{i+1}$.

Proof. Algorithms 1, 2 and 3 answer queries concerning the block-pair (A, B), and use constant time. The level $i + 1$ call is only made when the state changes. The original query is forwarded during a shift from state **initial** to **visited** and from state **full** to **full**$^+$, while during a shift from **visited** to **full** two LCE queries are asked, both with arguments at distance $p \le 2^{i+1}$, as claimed. □

5 Complexity Analysis

Algorithm 4 summarizes the implementation of the $\mathrm{LCE}^{(i)}(a,b)$ function. As mentioned in Sect. 3, we first compute Limited-$\mathrm{LCE}_{\le 3\cdot 2^i}(a,b)$, which might immediately give us the sought value $\mathrm{LCE}(a,b)$. Otherwise the query is relevant, and we refer to the block-pair (A,B) which is responsible for the query.

Algorithm 4. $\mathrm{LCE}^{(i)}(a,b)$

```
$\ell \leftarrow$ Limited-$\mathrm{LCE}_{\le 3\cdot 2^i}(a,b)$;
if $\ell < 3\cdot 2^i$ then                                  ▷ short query
    return $\ell$;
else                                                         ▷ relevant query
    $(A,B) \leftarrow$ block-pair responsible for the query $(a,b)$ at level $i$;
    return:
        Initial-$\mathrm{LCE}^{(i)}_{(A,B)}(a,b)$     if $(A,B)$ is in state initial
        Visited-$\mathrm{LCE}^{(i)}_{(A,B)}(a,b)$     if $(A,B)$ is in state visited
        Full-$\mathrm{LCE}^{(i)}_{(A,B)}(a,b)$        if $(A,B)$ is in state full or full$^+$
```

Let $S_i = \{(a,b) : \mathrm{LCE}^{(i)}(a,b)$ is called$\}$. Then $\lceil S_i/2^i \rceil$ corresponds to the set of pairs of interval right endpoints of block-pairs at level i.

Fact 8. *The set $\lceil S_i/2^i \rceil$ is non-crossing.*

Proof. We proceed by induction on i. The base case is trivial from the assumption on the input sequence. Lemma 5 proves that $S_{i+1} \subseteq S_i \cup \{(a,b) : a < b \le a + 2^{i+1}\}$. Hence, $\lceil S_{i+1}/2^{i+1} \rceil \subseteq \lceil \lceil S_i/2^i \rceil /2 \rceil \cup \{(a,b) : a \le b \le a+1\}$. The first component is non-crossing by the inductive hypothesis combined with Observation 4. Pairs of the form (a,a) and $(a,a+1)$ do not cross any other pair, so adding them to a non-crossing family preserves this property. □

Consequently, Fact 3 proves that the number of block-pairs responsible for a query at level $i-1$ is bounded by $\frac{3n}{2^{i-1}}$. Each of them yields at most 4 queries at level i. This leads straight to the following bound.

Observation 9. $|S_i| \le \frac{24n}{2^i}$ *for* $i \ge 1$.

If we stored the block-pairs using a hash table, we could retrieve the internal data of the block-pair responsible for (a,b) in randomised constant time. However, in the case of non-crossing LCE queries we can make this time worst-case.

Recall from Fact 3 that for a set $S \subseteq P_n$ of non-crossing pairs we can identify $S \backslash \{(a,a) : 1 \le a \le n\}$ with an outerplanar graph on vertices $\{1,\dots,n\}$. We say that a simple undirected graph has *arboricity* at most c if it can be partitioned into c forests. Outerplanar graphs have arboricity at most 2 (see [18]) which lets us use the following theorem to store $S \backslash \{(a,a) : 1 \le a \le n\}$. Membership queries for pairs (a,a) are trivial to support using an array.

Theorem 10. **([4]).** *Consider a graph of arboricity c with vertices given in advance and edges revealed on-line. One can support adjacency queries, asking to return the edge between two given vertices or **nil** if it does not exist, in worst-case $\mathcal{O}(c)$ time, with edge insertions processed in amortized constant time.*

The following corollary shows, by Fact 8, that indeed the block-pairs at each level can be retrieved in worst-case constant time.

Corollary 11. *Consider a set $S \subseteq P_n$ of non-crossing pairs arriving on-line. One can support membership queries (asking if $(a, b) \in S$ and, if so, to return data associated with this pair) in worst-case constant time with insertions processed in amortized constant time.*

Theorem 12. *In a string of length n, a sequence of q non-crossing LCE queries can be answered in total time $\mathcal{O}(q + n \cdot \alpha(n))$.*

Proof. For $i > 0$, an $\mathrm{LCE}^{(i)}$ query, excluding the $\mathrm{LCE}^{(i+1)}$ queries called, requires $\mathcal{O}(i \cdot \alpha(n))$ time for answering a Limited-$\mathrm{LCE}_{\le 3 \cdot 2^i}$ query by Lemma 1 plus $O(1)$ additional time by Lemma 5. For $i = 0$ we may compute Limited-$\mathrm{LCE}_{\le 3}$ naïvely in constant time, so the running time is constant.

The number of $\mathrm{LCE}^{(0)}$ queries is q, while the number of $\mathrm{LCE}^{(i)}$ queries for $i \ge 1$ is $\mathcal{O}(\frac{n}{2^i})$ by Observation 9. The total running time is therefore

$$\mathcal{O}\left(q + n \cdot \alpha(n) \cdot \sum_{i=1}^{\infty} \frac{i}{2^i}\right) = \mathcal{O}(q + n \cdot \alpha(n)). \qquad \square$$

6 Computing Runs

Bannai et al. [1,2] presented an algorithm for computing all the runs in a string of length n that works in time proportional to answering $\mathcal{O}(n)$ LCE queries on the string or on its reverse. As main tool they used Lyndon trees. We note here that the LCE queries asked by their algorithm can be divided into a constant number of groups, each consisting of non-crossing LCE queries. Roughly speaking, this is based on the obvious fact that intervals in a Lyndon tree form a *laminar family*, i.e., for every two they are either disjoint or one of them contains the other.

In the first phase, given a string w, the algorithm of [1,2] constructs $\mathit{LTree}_0(w)$ and $\mathit{LTree}_1(w)$. For each $r \in \{0, 1\}$, the construction of $\mathit{LTree}_r(w)$ goes from right to left. Before the k-th step (for $k = n, \ldots, 1$), we store on a stack the roots of subtrees of $\mathit{LTree}_r(w)$ that correspond to $w[k+1, n]$. Hence, the intervals corresponding to the roots on the stack are disjoint and cover the interval $[k + 1, n]$. In the k-th step we push on the stack a single node corresponding to $[k, k]$. Afterwards, as long as the stack contains at least two elements and the top element $[k, l]$ and the second to top element $[a, b]$ satisfy $w[k, l] \prec_r w[a, b]$, we pop the two subtrees from the stack and push one subtree with the root $[k, b]$. The lexicographical comparison is performed via an $\mathrm{LCE}(k, a)$ query.

Observation 13. *The* LCE *queries asked in the construction of $LTree_r(w)$ are non-crossing.*

Proof. In the k-th step of the algorithm we only ask LCE(i, j) queries for $i = k$. Suppose towards contradiction that in the course of the algorithm we ask two LCE queries with (i, j) and (i', j') such that $i < i' < j < j'$. The latter is asked at step i', and at that moment $[i', j' - 1]$ is a root of a subtree of $LTree_r(w)$. Then the former is asked at step i, and then $[i, j - 1]$ is a root of a subtree of $LTree_r(w)$. This contradicts the fact that the intervals in $LTree_r(w)$ form a laminar family. □

In the second phase, for each node $[a, b]$ of each Lyndon tree $LTree_r(w)$ we check if there is a run with period $p = b-a+1$ that contains $w[a, b]$. To this end we check how long does the periodicity with period p extend to the right and to the left of $w[a, b]$. The former obviously reduces to an LCE$(a, b + 1)$ query and the latter to an LCE query in the reverse of w, which is totally symmetric. As the intervals in $LTree_r(w)$ form a laminar family, we arrive at the following.

Observation 14. *The* LCE *queries asked when right-extending the periodicity of the intervals from $LTree_r(w)$ are non-crossing.*

By Observations 13 and 14, Theorem 12 yields the following result and its immediate corollary.

Theorem 15. *The Lyndon tree and the set of all runs in a string of length n over a general ordered alphabet can be computed in $\mathcal{O}(n\alpha(n))$ time.*

Corollary 16. *All the local periods and the number of all squares in a string of length n over a general ordered alphabet can be computed in $\mathcal{O}(n\alpha(n))$ time.*

References

1. Bannai, H., I, T., Inenaga, S., Nakashima, Y., Takeda, M., Tsuruta, K.: A new characterization of maximal repetitions by Lyndon trees. In: Indyk, P. (ed.) 26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, pp. 562–571. SIAM (2015)
2. Bannai, H., I, T., Inenaga, S., Nakashima, Y., Takeda, M., Tsuruta, K.: The "runs" theorem (2015). arXiv:1406.0263v7
3. Barcelo, H.: On the action of the symmetric group on the free Lie algebra and the partition lattice. J. Comb. Theory, Ser. A **55**(1), 93–129 (1990)
4. Brodal, G.S., Fagerberg, R.: Dynamic representations of sparse graphs. In: Dehne, F., Gupta, A., Sack, J.-R., Tamassia, R. (eds.) WADS 1999. LNCS, vol. 1663, pp. 342–351. Springer, Heidelberg (1999)
5. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press, New York (2007)
6. Crochemore, M., Ilie, L.: Analysis of maximal repetitions in strings. In: Kučera, L., Kučera, A. (eds.) MFCS 2007. LNCS, vol. 4708, pp. 465–476. Springer, Heidelberg (2007)

7. Crochemore, M., Ilie, L.: Maximal repetitions in strings. J. Comput. Syst. Sci. **74**(5), 796–807 (2008)
8. Crochemore, M., Ilie, L., Tinta, L.: Towards a solution to the "runs" conjecture. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 290–302. Springer, Heidelberg (2008)
9. Crochemore, M., Iliopoulos, C.S., Kubica, M., Radoszewski, J., Rytter, W., Waleń, T.: Extracting powers and periods in a word from its runs structure. Theor. Comput. Sci. **521**, 29–41 (2014)
10. Fischer, J., Holub, Š., I, T., Lewenstein, M.: Beyond the runs theorem. In: Iliopoulos, C., Puglisi, S., Yilmaz, E. (eds.) SPIRE 2015. LNCS, vol. 9309, pp. 277–286. Springer, Heidelberg (2015)
11. Gawrychowski, P., Kociumaka, T., Rytter, W., Waleń, T.: Faster longest common extension queries in strings over general alphabets. In: Grossi, R., Lewenstein, M. (eds.) 27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016. LIPIcs, vol. 54, pp. 5:1–5:13. Schloss Dagstuhl (2016)
12. Giraud, M.: Not so many runs in strings. In: Martín-Vide, C., Otto, F., Fernau, H. (eds.) LATA 2008. LNCS, vol. 5196, pp. 232–239. Springer, Heidelberg (2008)
13. Hohlweg, C., Reutenauer, C.: Lyndon words, permutations and trees. Theor. Comput. Sci. **307**(1), 173–178 (2003)
14. Kociumaka, T., Radoszewski, J., Rytter, W., Waleń, T.: Internal pattern matching queries in a text and applications. In: Indyk, P. (ed.) 26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, pp. 532–551. SIAM (2015)
15. Kolpakov, R.M., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: 40th Annual Symposium on Foundations of Computer Science, FOCS 1999, pp. 596–604. IEEE Computer Society (1999)
16. Kolpakov, R.M., Kucherov, G.: On maximal repetitions in words. J. Discrete Algorithms, 159–186. Special Issue of Matching Patterns, Hermes Science Publishing (2000). https://www.amazon.com/Matching-Patterns-Crochemore/dp/190339807X
17. Kosolobov, D.: Computing runs on a general alphabet. Inf. Process. Lett. **116**(3), 241–244 (2016)
18. Nash-Williams, C.S.J.A.: Decompositions of finite graphs into forests. J. London Math. Soc. **39**, 12 (1964)
19. Puglisi, S.J., Simpson, J., Smyth, W.F.: How many runs can a string contain? Theor. Comput. Sci. **401**(1–3), 165–171 (2008)
20. Rytter, W.: The number of runs in a string: improved analysis of the linear upper bound. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 184–195. Springer, Heidelberg (2006)
21. Rytter, W.: The number of runs in a string. Inf. Comput. **205**(9), 1459–1469 (2007)