A Taxonomy of LCP-Array Construction Algorithms

Johannes Fischer
TU Dortmund, Germany

Enno Ohlebusch[1]
Ulm University, Germany

Abstract

The combination of the suffix array and the LCP-array can be used to solve many string processing problems efficiently. We review some of the most important sequential LCP-array construction algorithms in random access memory.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases longest common prefix array, suffix array, Burrows-Wheeler transform

Digital Object Identifier 10.4230/OASIcs.Manzini.2025.8

1 Introduction

In a landmark paper, Weiner [43] presented the concept of suffix trees and devised the first linear-time construction algorithm. For several decades, the suffix tree was *the* data structure in stringology, because it can be used to efficiently solve a "myriad" of string processing problems [4]. For instance, with the help of the suffix tree, exact pattern matching can be done in $O(m)$ time (assuming a constant alphabet size), where m is the length of the pattern searched for. It should be pointed out that the search time is independent of the text length n.

The suffix array was devised by Manber and Myers [30] and independently by Gonnet et al. [19] under the name PAT array. Ten years later, it was shown independently and contemporaneously by Kärkkäinen and Sanders [23], Kim et al. [26], Ko and Aluru [27], and Hon et al. [20] that a direct linear-time construction of the suffix array is possible.

The LCP-array first appeared in the seminal paper of Manber and Myers [30], where it was called *Hgt* array. The authors presented an $O(n \log n)$ time LCP-array construction algorithm (LACA for short) and showed that the augmentation of classical binary search with the LCP-array enables exact pattern matching on the suffix array in $O(m + \log n)$ time. The first linear-time LACA was devised by Kasai et al. [25]. They also demonstrated the importance of the LCP-array by presenting a linear-time algorithm that simulates the bottom-up traversal of a suffix tree with a suffix array *and* the LCP-array. Their work was taken one step further by Abouelhoda et al. [1], who showed that suffix trees can be completely replaced with *enhanced* suffix arrays. An enhanced suffix array is the combination of the suffix array with augmenting data structures, which depend on the application at hand. For example, exact pattern matching can be done in $O(m)$ time with the suffix array, the LCP-array, and constant-time range minimum queries [16] on the LCP-array [1] – again assuming constant alphabet size; for non-constant alphabet sizes σ one can extend this idea to achieve $O(m \log \sigma)$ running time [15]. Algorithms on enhanced suffix arrays are not only more space efficient than those on suffix trees, but they are also faster and easier to implement. Since the LCP-array is of utmost importance in this context, it has been studied extensively.

[1] Corresponding author

Table 1 Suffix array, LCP-array, and BWT of the text $S = mississippi\$$.

k	SA	LCP	BWT	$S_{\mathsf{SA}[k]}$
1	12	$\perp$	i	$\$$
2	11	0	p	$i\$$
3	8	1	s	$ippi\$$
4	5	1	s	$issippi\$$
5	2	4	m	$ississippi\$$
6	1	0	$\$$	$mississippi\$$
7	10	0	p	$pi\$$
8	9	1	i	$ppi\$$
9	7	0	s	$sippi\$$
10	4	2	s	$sissippi\$$
11	6	1	i	$ssippi\$$
12	3	3	i	$ssissippi\$$

2 Preliminaries

Let S be a text of length n on an ordered alphabet Σ of constant size σ. We assume that S has the sentinel $\$ \in \Sigma$ at the end (and nowhere else), which is smaller than any other character. For $1 \leq i \leq n$, $S[i]$ denotes the *character at position* i in S. For $i \leq j$, $S[i,j]$ denotes the *substring* of S starting at position i and ending at position j. Furthermore, S_i denotes the i-th suffix $S[i,n]$ of S.

The *suffix array* SA of the text S is an array of integers in the range 1 to n specifying the lexicographic ordering of the n suffixes of S, that is, it satisfies $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \cdots < S_{\mathsf{SA}[n]}$. The suffix array can be built in linear time; we refer to the overview article of [37] for suffix array construction algorithms and to [35] for recent developments.

The *inverse suffix array* ISA is an array of size n such that for any i with $1 \leq i \leq n$ the equality $\mathsf{ISA}[\mathsf{SA}[i]] = i$ holds. Obviously, it takes only linear time to invert the suffix array.

The LCP-array contains the lengths of longest common prefixes between consecutive suffixes in SA. Formally, the LCP-array is an array such that $\mathsf{LCP}[1] = \perp$ and $\mathsf{LCP}[i] = \mathsf{lcp}(S_{\mathsf{SA}[i-1]}, S_{\mathsf{SA}[i]})$ for $2 \leq i \leq n$, where $\mathsf{lcp}(u,v)$ denotes the length of the longest common prefix between two strings u and v. Here the value $\mathsf{LCP}[1]$ is undefined, but – depending on the application – it may make sense to define it as 0 or -1. Table 1 shows the LCP-array of the string $S = mississippi\$$.

The Burrows-Wheeler transform [9] converts S into the string $\mathsf{BWT}[1,n]$ defined by $\mathsf{BWT}[i] = S[\mathsf{SA}[i] - 1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $\mathsf{BWT}[i] = \$$ otherwise. Given the suffix array, this transformation takes only linear time.

The ψ-array is an array of size n such that $\psi[1] = \mathsf{ISA}[1]$ and $\psi[i] = \mathsf{ISA}[\mathsf{SA}[i] + 1]$ for all i with $2 \leq i \leq n$. That is, $\psi[i]$ is the index at which the suffix $S_{\mathsf{SA}[i]+1}$ occurs in the suffix array. The ψ-array is the inverse of the LF-array, which can easily be computed in linear time from the BWT; see e.g. [34, Section 7.2.2].

The C-array is an array of size σ. For $c \in \Sigma$, the entry $C[c]$ is defined as follows: if we consider all characters in Σ that are smaller than c, then $C[c]$ is the overall number of their occurrences in S.

Algorithm 1 This procedure computes the LCP-array by Kasai et al.'s algorithm [25].

```
Input: $S$, SA, ISA
$\ell \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    $j \leftarrow \mathsf{ISA}[i]$
    if $j > 1$ then
        $k \leftarrow \mathsf{SA}[j-1]$          /* $S_k$ precedes $S_i$ in SA */
    while $S[k+\ell] = S[i+\ell]$ do
        $\ell \leftarrow \ell + 1$
    $\mathsf{LCP}[j] \leftarrow \ell$
    if $\ell > 0$ then
        $\ell \leftarrow \ell - 1$
```

3 The First Linear Time LACA

We first present Kasai et al.'s LACA [25], which computes the LCP-array from the string S, its suffix array SA, and its inverse suffix array ISA; see Algorithm 1. It starts with the longest suffix $S_i = S_1$ of S (so $i = 1$), computes $j = \mathsf{ISA}[i]$ and then $\mathsf{LCP}[j]$ by a left-to-right comparison of the characters in $S_i = S_{\mathsf{SA}[j]}$ and its immediately preceding suffix $S_k = S_{\mathsf{SA}[j-1]}$ in the suffix array. The same is done for the other suffixes S_i of S by incrementing i successively. As we shall see, some character comparisons can safely be skipped.

In our example, Algorithm 1 first compares $S_1 = \mathit{mississippi\$}$ with $S_2 = \mathit{ississippi\$}$ and sets $\mathsf{LCP}[j] = \mathsf{LCP}[6] = 0$. In the next iteration of the for-loop, it compares $S_2 = \mathit{ississippi\$}$ with $S_5 = \mathit{issippi\$}$ and sets $\mathsf{LCP}[j] = \mathsf{LCP}[5] = 4$. In the following iterations of the for-loop the underlined characters in $S = \mathit{mi}\underline{\mathit{ssi}}\mathit{ssippi\$}$ will be skipped in further comparisons of adjacent suffixes. For example, in the third iteration, Algorithm 1 must compare $S_3 = \mathit{ssissippi\$}$ with $S_6 = \mathit{ssippi\$}$ and the first three characters will be skipped in the comparison. To prove the correctness of Algorithm 1, we must show that if $\ell > 1$ characters match in iteration i of the for-loop, then at least $\ell - 1$ characters match in iteration $i + 1$. So suppose that the longest common prefix of S_i and its preceding suffix S_k is $c\omega$, where c is a character and ω is a string of length $\ell - 1 \geq 1$. In the next iteration, Algorithm 1 compares S_{i+1} with its preceding suffix, which is not necessarily S_{k+1} as in the example above. However, since $S_k = c\omega u$ is lexicographically smaller than $S_i = c\omega v$, it follows that $S_{k+1} = \omega u$ is lexicographically smaller than $S_{i+1} = \omega v$. Moreover, all the suffixes in between S_{k+1} and S_{i+1} must share the prefix ω because the suffixes are ordered lexicographically. In particular, S_{i+1} and its preceding suffix have ω as a common prefix. Therefore, $\ell - 1$ characters (namely ω) can be skipped in the comparison of S_{i+1} and its preceding suffix. If S_{k+1} immediately precedes S_{i+1} in the suffix array, then we can directly conclude that their longest common prefix is ω; otherwise further character comparisons are required to determine their longest common prefix. Obviously, S_{k+1} precedes S_{i+1} if $\mathsf{BWT}[j'] = \mathsf{BWT}[j' - 1]$, where $j' = \mathsf{ISA}[i + 1]$. This result is summarized in the next lemma, which will be important later. The first proof of Lemma 1 can be found in [25]. A value $\mathsf{LCP}[j]$ is called *reducible* if $\mathsf{BWT}[j] = \mathsf{BWT}[j - 1]$; otherwise it is called *irreducible*.

Algorithm 2 This procedure computes the LCP-array by the algorithm of Kasai et al. [25] with Manzini's [31] improvement. Initially the array LCP stores ψ. During the course of computation, the initial values in the LCP-array are overwritten by the correct values.

```
Input: $S$, SA, LCP $= \psi$
1  $j \leftarrow$ LCP[1]          /* LCP[1] $= \psi[1] =$ ISA[1] */
2  $\ell \leftarrow 0$
3  for $i \leftarrow 1$ to $n$ do
4      $next \leftarrow$ LCP[$j$]          /* LCP[$j$] $= \psi[j] =$ ISA[$i+1$] */
5      if $j > 1$ then
6          $k \leftarrow$ SA[$j-1$]          /* $S_k$ precedes $S_i$ in SA */
7      while $S[k+\ell] = S[i+\ell]$ do
8          $\ell \leftarrow \ell + 1$
9      LCP[$j$] $\leftarrow \ell$
10     if $\ell > 0$ then
11         $\ell \leftarrow \ell - 1$
12     $j \leftarrow next$
```

▶ **Lemma 1.** *Suppose that* LCP[j] *is reducible, where* $j =$ ISA[i]. *Then we have* LCP[j] $=$ LCP[ISA[$i-1$]] $- 1$.

It will now be shown that Algorithm 1 constructs the LCP-array in $O(n)$ time. We use an amortized analysis to show that the while-loop is executed at most $2n$ times. It is readily verified that this implies the linear run-time. Each comparison in the while-loop ends with a mismatch, so there are $n-1$ mismatches (redundant character comparisons) in total. If a position p in S is involved in a match, then this particular occurrence of character $S[p]$ will be skipped in further suffix comparisons. More precisely, if we have $S[k+\ell] = S[i+\ell]$ in the while-loop, then $p = i+\ell$ will not appear on the right-hand-side of a character comparison again. So there are at most n matches. To illustrate the argument, the positions in our example text that are involved in a match are marked by underlining the respective character in $S = m\underline{issis}s\underline{ip}pi\$$. Since every position is marked at most once (because it is skipped in further comparisons), it is clear that the overall number of character comparisons is $n-1$ (mismatches) plus the number of underlined characters.

In Algorithm 1, the text S as well as the arrays SA, ISA, and LCP should be randomly accessible. Under the common assumption that an entry in SA takes 4 bytes, each of the arrays SA, ISA, and LCP requires $4n$ bytes. Consequently, for an 8 bit alphabet, Algorithm 1 needs $13n$ bytes of working memory.

In the context of compressed suffix trees, Mäkinen [29, Proposition 3.2] first showed that the LCP-array can be constructed with less working space. Manzini [31] observed that the additional array ISA is superfluous because it is possible to store the required information about ISA in the LCP-array itself. If the LCP-array initially stores the ψ-array, then the cell LCP[j] contains the value ISA[$i+1$] in iteration i of the for-loop of Algorithm 2. That is, variable *next* gets the value ISA[$i+1$] in line 4 and LCP[j] can safely be overwritten in line 9. The correctness of Algorithm 2 follows directly from this fact. Algorithm 2 requires $9n$ bytes of working memory because it needs random access to the text and both input arrays.

Manzini [31] proposed another LACA, which saves even more space by overwriting the suffix array. This LACA is based on Lemma 1: if LCP[j] is reducible, then one can immediately set LCP[j] $\leftarrow \ell$ because the character comparisons in lines 7-8 of Algorithm 2 are superfluous. It follows as a consequence that in this case the assignment $k \leftarrow$ SA[$j-1$] is

i	1	2	3	4	5	6	7	8	9	10	11	12
$S[i]$	m	i	s	s	i	s	s	i	p	p	i	\$
$\Phi[i]$	2	5	6	7	8	4	9	11	10	1	12	0
$\mathsf{PLCP}[i]$	0	4	3	2	1	1	0	1	1	0	0	$\perp$

Figure 1 Φ-array and PLCP-array of $S = mississippi\$$; cf. Table 1.

superfluous, too. In other words, the value $\mathsf{SA}[j-1]$ is required only if $\mathsf{LCP}[j]$ is irreducible. In a preprocessing phase, Manzini's second LACA determines the values in the suffix array SA that are actually required and stores them in an auxiliary array. The auxiliary array is then used instead of SA and SA can safely be overwritten. However, this LACA is of limited value in practice because in most applications the suffix array is needed as well.

4 The Φ-Algorithm

Kärkkäinen, Manzini, and Puglisi [22] proposed a variant of Kasai et al.'s algorithm, which first computes a permuted LCP-array (the PLCP-array) with the help of the so-called Φ-array and then derives the LCP-array from the PLCP-array. They called it "Φ-algorithm" because it uses the Φ-array, which "is in some way symmetric to the ψ array."

The Φ-array and the PLCP-array, respectively, are defined as follows: For all i with $1 \leq i \leq n$ let

$$\Phi[i] = \begin{cases} \mathsf{SA}[\mathsf{ISA}[i]-1] & \text{if } \mathsf{ISA}[i] \neq 1 \\ 0 & \text{otherwise} \end{cases} \quad \text{and } \mathsf{PLCP}[i] = \begin{cases} \mathsf{lcp}(S_i, S_{\Phi[i]}) & \text{if } \Phi[i] \neq 0 \\ \perp & \text{otherwise} \end{cases}$$

Fig. 1 shows our example. So in the suffix array SA, the suffix S_i is immediately preceded by the suffix $S_{\Phi[i]}$ unless $\Phi[i] = 0$. Note that we have $\Phi[n] = 0$ because we assume that S is terminated by \$. For $j \neq 1$, we have

$$\mathsf{PLCP}[\mathsf{SA}[j]] = \mathsf{lcp}(S_{\mathsf{SA}[j]}, S_{\Phi[\mathsf{SA}[j]]}) = \mathsf{lcp}(S_{\mathsf{SA}[j]}, S_{\mathsf{SA}[j-1]}) = \mathsf{LCP}[j] \tag{1}$$

In case $j = 1$, Equation (1) holds also true: $\mathsf{LCP}[1] = \perp$ and $\mathsf{PLCP}[\mathsf{SA}[1]] = \mathsf{PLCP}[n] = \perp$ because $\Phi[n] = 0$. Thus, the PLCP-array is a permutation of the LCP-array. The difference between the two arrays is that the lcp-values occur in text order (position order) in the PLCP-array, whereas they occur in suffix array order (lexicographic order) in the LCP-array. The pseudocode of the Φ-algorithm is shown in Algorithm 3. Its properties (correctness, linear run-time) can be shown as in the analysis of Algorithm 1.

Experimental comparisons of Algorithm 1 and the Φ-algorithm can be found in [22, 18]. The bottom line is that the Φ-algorithm is faster because of better memory locality: it merely needs sequential access to the Φ-array and the PLCP-array in its second for-loop. Consequently, these arrays can be streamed and the number of possible cache misses is limited to $3n$: n in the computation of the Φ-array, n in line 6,[2] and n in the conversion of the PLCP-array into the LCP-array (in virtually all applications lcp-values are required to be in suffix array order). By contrast, there can be up to $4n$ cache misses in Algorithm 1 (including the computation of ISA).

[2] Remark: If $\mathsf{PLCP}[i]$ is reducible, then $\Phi[i] = \Phi[i-1]+1$ and hence $S[\Phi[i]]$ is cached.

Algorithm 3 The Φ-algorithm of Kärkkäinen et al. [22].

```
Input: $S$, SA
for $i \leftarrow 2$ to $n$ do
    $\Phi[\mathsf{SA}[i]] \leftarrow \mathsf{SA}[i-1]$
$\ell \leftarrow 0$
for $i \leftarrow 1$ to $n-1$ do
    $k \leftarrow \Phi[i]$          /* $S_k$ precedes $S_i$ in SA */
    while $S[k+\ell] = S[i+\ell]$ do
        $\ell \leftarrow \ell + 1$
    $\mathsf{PLCP}[i] \leftarrow \ell$
    if $\ell > 0$ then
        $\ell \leftarrow \ell - 1$
for $i \leftarrow 2$ to $n$ do
    $\mathsf{LCP}[i] \leftarrow \mathsf{PLCP}[\mathsf{SA}[i]]$
```

If we assume that the arrays SA, Φ, PLCP, and LCP are kept in working memory, then Algorithm 3 requires $17n$ bytes RAM. However, the algorithm merely needs random access to the Φ-array in the first for-loop, to the text in the second for-loop, and to the PLCP-array in the third for-loop. All other arrays can be streamed from or to disk. Thus, $9n$ bytes of RAM are sufficient.

Kärkkäinen et al. [22] presented another algorithm for computing the PLCP-array based on irreducible PLCP-values. We call a value $\mathsf{PLCP}[i]$ *reducible* if $S[i-1] = S[\Phi[i]-1]$; otherwise it is called *irreducible*. The proof of the next lemma is similar to that of Lemma 1.

▶ **Lemma 2.** *If* $\mathsf{PLCP}[i]$ *is reducible, then* $\mathsf{PLCP}[i] = \mathsf{PLCP}[i-1] - 1$.

The above-mentioned algorithm first determines all irreducible PLCP-values in a brute-force manner: each PLCP-value is computed by comparing the respective suffixes from the very beginning. Afterwards, it uses Lemma 2 to compute the remaining reducible PLCP-values. The authors showed that the sum of all irreducible PLCP-values is at most $2n \log n$. Hence, the overall run-time is $O(n \log n)$. An experimental comparison of this algorithm and the Φ-algorithm showed that the Φ-algorithm is faster in practice; see [22].

5 Computing the LCP-Array along with SA

The LCP-array can also be computed while sorting the suffixes lexicographically, which we show here for two well-known suffix array construction algorithms DC3 [24] and SAIS [33]. The adaptions to the LCP-array computation were described in the original article for DC3, and by Bingmann et al. for SAIS [8]. Intuitively, it seems clear that LCP-array computation during suffix array construction should be possible, as any suffix sorting algorithm must, at least implicitly, determine the lcp-values between lexicographically adjacent suffixes to determine their order, for otherwise it could not make the right decisions on the order of suffixes.

5.1 A Unified Perspective

We first look at the common features that both algorithms share, which form the basis for the LCP-computation. We assume knowledge about DC3 and SAIS, as a full exposition of those algorithms would take too much space in this article. We also leave out some details of

LCP-array construction and refer the readers to the original articles at various points; the focus here is more on a common exposition of how the computations of lcp-values can be woven into suffix sorting algorithms.

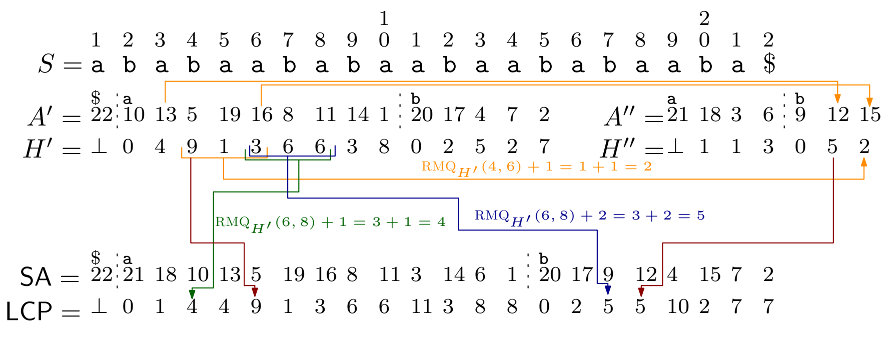
Both DC3 and SAIS are recursive algorithms, and for sorting all the suffixes of a text $S[1,n]$ they first sort a *subset* of $n' \leq n/\Omega(1)$ suffixes recursively, stored in an array $A'[1,n']$ such that $S_{A'[i]} < S_{A'[i+1]}$ for all $1 \leq i < n'$. From A', they use different mechanisms to compute the full suffix array $\mathsf{SA}[1,n]$. In DC3, A' contains all suffixes at positions $i \not\equiv 0 \pmod 3$ (see top of Fig. 2), whereas in SAIS, A' contains those positions i with $S_{i-1} > S_i < S_{i+1}$ (called LMS-suffixes for *leftmost smallest suffixes*; see top of Fig. 3 with LMS-suffixes indicated by a star). In both algorithms, we talk about "c-buckets" for an arbitrary character $c \in \Sigma$, by which we mean the subarray consisting of the suffixes starting with $c \in \Sigma$ as in counting or bucket sort – in the figures those buckets are depicted by dotted lines.

By induction, we can assume that we also get the LCP-array H' for all the lexicographically adjacent suffixes in A' from the recursion: $H'[i] = \mathsf{lcp}(S_{A'[i]}, S_{A'[i-1]})$. The details of how to compute H' differ between the two algorithms and are a bit tedious, but are not the core of the algorithmic ideas. The reason why this is not trivial is that because A' has been computed recursively from the suffix array of a reduced text S' (each character in S' representing several characters in S – 3 in DC3 (hence the name), and a non-constant number in SAIS), the lcp-values need to be *scaled* to actually represent the number of characters of shared prefixes in S, and not S'. We assume here that this scaling step has already been performed – see again Fig. 2 for DC3 and Fig. 3 for SAIS, where, in the latter case, both arrays A' and H' are already stored in the final arrays SA and LCP, as the algorithm works in-place (skip the gray values in LCP for this moment, as they will be written in the course of the algorithm that follows).

Then both SAIS and DC3 compute the suffix array SA from A', and the algorithms can be enhanced to compute the full LCP-array from H'. Both algorithms finally place suffixes into SA in a left-to-right manner (for SAIS, this happens in actually *two* passes, the second one right-to-left but symmetrically). Say the algorithm just wrote $\mathsf{SA}[x] \leftarrow i$ and now needs to compute the lcp-value h between S_i and its lexicographic predecessor S_j at position $x-1$ in SA; the result h will have to be stored at $\mathsf{LCP}[x] \leftarrow h$. As the values in SA are filled from left to right, the value $\mathsf{SA}[x-1] = j$ has been set in a previous iteration of this loop. (In SAIS, this left-to-right filling is per bucket, but the principle remains the same, as only suffixes in the same c-bucket have an lcp-value > 0.)

The easy case is if both i and j were neighbors in A'; then we know their lcp-values and we can just copy the lcp-value from H'. Consider, for example, the placement of suffix $i = 5$ and $j = 13$ at positions $x = 6$ and $x - 1 = 5$ in Fig. 2: those suffixes are also neighbors in A' at positions 4 and 3, respectively, and hence the lcp-value $H'[4] = 9$ can be copied to $\mathsf{LCP}[6] \leftarrow 9$ (left red arrow). The same situation might occur in SAIS, e.g. in Fig. 3 suffix $i = 9$ is placed to the right of $j = 6$, whose lcp-value of 2 can just be copied from H'.

The more difficult case is when i and j were not neighbors in A'. Here, the techniques differ between the algorithms, but they share some common ideas. We first compare the two *characters* $S[i]$ and $S[j]$ (if in DC3 $i + 1 \equiv 0 \pmod 3$, we also compare $S[i+1]$ and $S[j+1]$). If the 1 or 2 compared characters are all the same (otherwise the lcp-value h has been determined), it is clear that the value $h = \mathsf{lcp}(S_i, S_j)$ must be one more than h', where $h' = \mathsf{lcp}(S_{i+1}, S_{j+1})$ (possibly two more than $h'' = \mathsf{lcp}(S_{i+2}, S_{j+2})$ in DC3, see above). Now h' (or h'') can be computed by a *range minimum query* on H', which gives the minimum value between two specified array indices. More formally, the range minimum



S = a b a b a a b a b a a b a a b a b a a b a $
A′ = 22 10 13 5 19 16 8 11 14 1 20 17 4 7 2
H′ = ⊥ 0 4 9 1 3 6 6 3 8 0 2 5 2 7
A″ = 21 18 3 6 9 12 15
H″ = ⊥ 1 1 3 0 5 2
RMQ_{H′}(4, 6) + 1 = 1 + 1 = 2
RMQ_{H′}(6, 8) + 1 = 3 + 1 = 4
RMQ_{H′}(6, 8) + 2 = 3 + 2 = 5
SA = 22 21 18 10 13 5 19 16 8 11 3 14 6 1 20 17 9 12 4 15 7 2
LCP = ⊥ 0 1 4 4 9 1 3 6 6 11 3 8 8 0 2 5 5 10 2 7 7

Figure 2 Example of LCP-array computation with the DC3 suffix sorting algorithm.

query (RMQ) problem on a general array $X[1, m]$ is to find a data structure such that later, for two specified indices ℓ, r with $1 \leq \ell \leq r \leq m$, the minimum in $X[\ell, r]$ can be returned efficiently, in symbols: $\text{RMQ}_X(\ell, r) = \min\{X[k] \ : \ \ell \leq k \leq r\}$. For these queries, linear preprocessing schemes exist that can answer the queries in constant time [16]. Armed with this tool, in DC3 we can simply compute $h' = \text{RMQ}_{H'}(A'^{-1}[i+1]+1, A'^{-1}[j+1])$ (or $h'' = \text{RMQ}_{H'}(A'^{-1}[i+2]+1, A'^{-1}[j+2])$), where A'^{-1} is the inverse of A', defined by $A'[A'^{-1}[p]] = p$ for all values p actually stored in $A'[1, n']$ – note that for all the positions p queried we have $p \not\equiv 0 \pmod 3$; this is also why the distinction between h' and h'' has been made.

For example, in Fig. 2, when writing $\mathsf{SA}[4] \leftarrow 10$ to the right of $\mathsf{SA}[3] = 18$, we have $A'^{-1}[11] = 8$ and $A'^{-1}[19] = 5$ and hence compute $h' = \text{RMQ}_{H'}(6, 8) = 3$ and thus finally write $\mathsf{LCP}[4] = 3 + 1$ (green color). When writing $\mathsf{SA}[17] \leftarrow 9$ to the right of $\mathsf{SA}[16] = 17$, we compute $h'' = \text{RMQ}_{H'}(6, 8) = 3$ and thus finally write $\mathsf{LCP}[17] = 3 + 2$ (blue color).

The justification of this approach is the following: Let $Y[1, m]$ be an arbitrary array storing strings in lexicographically sorted order, and let $X[2, m]$ be an array such that $X[i]$ stores the lcp-value between strings $Y[i]$ and $Y[i-1]$ for all $2 \leq i \leq m$. Then due to the lexicographic sortedness of Y, the lcp-value between two arbitrary strings $Y[z]$ and $Y[y]$ is the minimum value from $X[z+1], \ldots, X[y]$ (assuming $z < y$), in symbols: $\mathsf{lcp}(Y[z], Y[y]) = \text{RMQ}_X(z+1, y)$. In our case, we exploit this fact with $Y[i] = S_{A'[i]}$ and $X[i] = H'[i]$.

In SAIS, the situation is different, as there is (most likely) no constant K such that $i + K$ and $j + K$ are both in A'. Instead, we make use of the fact that (in the left-to-right scan) both suffixes S_{i+1} and S_{j+1} are lexicographically smaller than S_i and S_j, respectively, and are thus already present in SA (say at positions z with $\mathsf{SA}[z] = i + 1$ and $y < z$ with $\mathsf{SA}[y] = j + 1$), including the lcp-values with their respective lexicographic predecessors *and all lcp-values in between* (the entire subarray $\mathsf{LCP}[y, z]$ has already been written). So $h' = \text{RMQ}_{\mathsf{LCP}}(y + 1, z)$ can be easily computed in the same manner as in DC3. Look at the computation of the values $\mathsf{SA}[16]$ and $\mathsf{LCP}[16]$ in Fig. 3 (orange color): We write $\mathsf{SA}[16] \leftarrow 11$ when the scanning of SA reaches position $z = 11$ with $\mathsf{SA}[11] = 12$. As $j = \mathsf{SA}[15] = 1$, which was written when at position $y = 8$, we need to set $\mathsf{LCP}[16]$ to the lcp-value of suffix S_{11} and its lexicographic predecessor S_1. The values $\mathsf{LCP}[8, 11]$ are all known (gray values in H below the $\mathtt{b}$-bucket); hence we can perform an RMQ from positions 9 to 11 in LCP, which gives a 2. We thus set $\mathsf{LCP}[16] \leftarrow 2 + 1 = 3$.

Commenting on the differences between the RMQs in DC3 and SAIS, we can say that in DC3 the RMQs are performed on the static array H' (for which optimal preprocessing schemes exist), whereas for SAIS the array LCP on which RMQs are performed grows semi-dynamically from the left to the right. We comment below how this can be done efficiently.

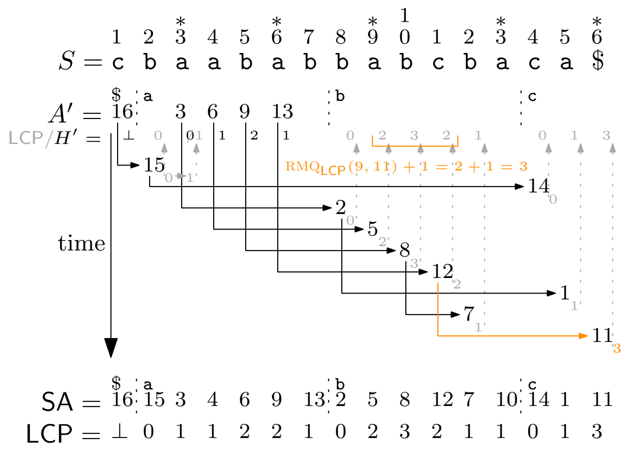

S = c b a a b a b b a b c b a c a $
A' = 16 3 6 9 13
LCP/H' =
time
RMQ_LCP(9, 11) + 1 = 2 + 1 = 3
SA = 16 15 3 4 6 9 13 2 5 8 12 7 10 14 1 11
LCP = ⊥ 0 1 1 2 2 1 0 2 3 2 1 1 0 1 3

Figure 3 Example of LCP-array computation with the SAIS suffix sorting algorithm. Note that this only shows the left-to-right scan – in a subsequent right-to-left scan the missing suffixes 10 and 4 including the affected lcp-values would also be considered; the final result is shown at the bottom.

5.2 DC3

The above description of DC3 skipped one important detail of its inner workings: when it returns from the recursion, before it finally computes the full suffix array SA, it first needs to compute an array A'' containing the lexicographic order of the suffixes starting at positions $i \equiv 0 \pmod 3$. This is done by a left-to-right scan through A': for $z = 1, \ldots, n' \approx \frac{2n}{3}$ in ascending order, if $A'[z] = i + 1 \equiv 1 \pmod 3$, place i to the first free place p in A'''s c-bucket for $c = S[i]$. (By “first free position” we mean the first position not already written in that bucket, as in classical counting- or bucket-sort algorithms.)

Having computed A'' from A', we now explain how to compute array H'' storing the lcp-values of lexicographically adjacent suffixes in A''. When placing i to $A''[p]$, let $A''[p-1] = j$ be its immediate predecessor in A'' in the c-bucket (first suffixes in a bucket are simple as their lcp-values are 0). Since both S_{i+1} and S_{j+1} are suffixes in A', we can compute $H''[p] = 1 + h''$, where $h'' = \text{RMQ}_{H'}(A'^{-1}[j] + 1, A'^{-1}[i])$. Note that this is another usage of the RMQ data structure on H' that needs to be precomputed anyway for the later steps.

For example, in Fig. 2, when we write the final value $A''[7] \leftarrow 15$ next to $A''[6] = 12$, we have $A'^{-1}[13] = 3$ and $A'^{-1}[16] = 6$, and hence compute $H''[7]$ as $\text{RMQ}_{H'}(4, 6) + 1 = 1 + 1 = 2$ (orange color).

The rest of the algorithm works exactly as explained above – with the addition that if two suffixes from A'' are placed adjacently into SA, we can look up their lcp-value directly in H''. In more detail, the final computation of SA can be viewed as merging the arrays A' and A'', as they are both sorted lexicographically and, together, contain all the suffixes from S. Hence, if in this merging the value i is written to SA$[x]$ and we already know SA$[x-1] = j$ with both $i, j \equiv 0 \pmod 3$, their lcp-value can be directly retrieved from H'' and written to LCP, as i and j are neighbors in A''. See Fig. 2 for an example, where the value SA$[18] = 12$ is written right after SA$[17] = 9$, both of which come from A'' at adjacent positions 5 and 6 with $H''[6] = 5$ (right red arrow). We note here that neither A'' nor H'' need be materialized, but can be computed on the fly during the merging process.

5.3 SAIS

We now give more details about the LCP-array computation during the SAIS-algorithm. Let us first consider the *scaling* of the lcp-values to compute H'. As a reminder, the suffixes in A' (those positions i with $S_{i-1} > S_i < S_{i+1}$) are called *LMS-suffixes*, including the sentinel character `$` at the end. The recursion works by computing the suffix array of a reduced text with meta-characters ranging from one LMS-suffix to the next. In our example, the reduced text is $T' =$ [`aaba`][`abba`][`abcba`][`aba$`][`$`] (with square brackets indicating the meta-characters), and the recursion stops immediately with $A' = [16, 3, 6, 9, 13]$, as all meta-characters are different. Now in order to see the issue with LCP-computation, look at the suffixes S_6 and S_9 from A': in A', their order was determined from the order of the meta-characters $S[6, 9] =$ [`abba`] and $S[9, 13] =$ [`abcba`], where the first one is smaller, and the lcp-value of the suffixes – in terms of meta-characers – is 0. But the meta-characters *themselves* share a common prefix, `ab` in this case. Hence, the correct lcp-value for suffixes S_6 and S_9 is 2. Here, we do not give the full details of how the array H' is actually computed correctly, but refer the reader to [8, Lemma 3.1] for the correct way to do it, and to [14, Sect. 3.3] for the way how to compute this efficiently.

For what follows, we also need to classify the other suffixes: if $S_i > S_{i+1}$, then S_i is called *L-suffix* (for "larger"); if $S_{i-1} < S_i < S_{i+1}$, then S_i is called *S-suffix* (for "smaller"). The left-to-right pass (as shown in the example in Fig. 3) actually sorts the L-suffixes from the LMS-suffixes (whose order has been computed recursively), whereas the subsequent right-to-left pass sorts the S-suffixes into the already sorted LMS-suffixes. In a c-bucket in SA for a character $c \in \Sigma$, all the L-suffixes come first, followed by a mix of S- and LMS-suffixes. E.g., at the bottom of Fig. 3 one can see that in the `a`-bucket the only L-suffix S_{15} comes first, followed by the LMS-suffix S_3, an S-suffix S_4, followed by three more LMS-suffixes S_3, S_9, S_{13} in this order.

A further detail is the computation of the lcp-value between the last L-suffix in a bucket and its lexicographic successor (which is either S or LMS). In Fig. 3, this happens in the `a`-bucket, where the L-suffix S_{15} is placed right before the LMS-suffix S_3, which had an lcp-value of 0 in H' (stored at LCP[3]) because it was the lexicographically first suffix starting with an `a` among those in A'. However, with the placement of SA[2] $\leftarrow$ 15, the value LCP[3] is not 0 anymore, but needs to be updated to 1 (second gray arrow from the left). It can be shown [8, Lemma 3.2] that in the c-bucket only the character c, possibly repeated many times, can contribute to the lcp-value of those suffixes. Hence, a *naive* LCP-computation per bucket suffices to compute these lcp-values, as any character from S can contribute at most once to this naive computation (in its corresponding bucket).

The last issue is that the RMQs needed for LCP-computation are performed on an array LCP where values are constantly being written to, no data structures are known for constant-time range minimum queries for dynamic arrays. However, the updates occur in a regular manner, namely from left to right per bucket in the left-to-right scan (and again, symmetrically from right to left in the right-to-left scan). We sketch here the solution for such *semi-dynamic RMQs* [8, Sect. 3.2] based on *LRM-trees* [5, Def. 1] – also known under the slightly misleading name *2d-Min-Heaps* [16, Def. 5.3]. We maintain an LRM-tree T_c for each bucket c, which initially contains only the LMS-suffixes of that bucket with their respective lcp-values (as computed in the recursive call). When a new L-suffix along with lcp-value h is written into its c-bucket, we climb up the rightmost path of T_c until we find an element x whose corresponding LCP-entry is strictly smaller than h (T_c has an artificial root holding lcp-value $-\infty$, which guarantees that such an element always exists). The new element is then added as x's new rightmost leaf; an easy amortized argument shows that

Table 2 BWT of $S = mississippi\$$ and its partially filled LCP-array.

i	LCP	BWT	$S_{\mathsf{SA}[i]}$
1	$\perp$	i	$\$$
2	$0\ \cancel{\perp}$	p	$i\$$
3	$\perp$	s	$ippi\$$
4	$\perp$	s	$issippi\$$
5	$\perp$	m	$ississippi\$$
6	$0\ \cancel{\perp}$	$\$$	$mississippi\$$
7	$0\ \cancel{\perp}$	p	$pi\$$
8	$\perp$	i	$ppi\$$
9	$0\ \cancel{\perp}$	s	$sippi\$$
10	$\perp$	s	$sissippi\$$
11	$\perp$	i	$ssippi\$$
12	$\perp$	i	$ssissippi\$$
13	-1		

this results in overall linear time. T_c is stored with a data structure for constant-time lowest common ancestor queries (LCAs) that supports dynamic leaf additions in $O(1)$ worst-case time [10]. Then the minimum in any range in the processed portion of the c-bucket can be found in $O(1)$ time [16, Lemma 5.5] by LCA-queries on T_c.

Interestingly, this solution for RMQs on semi-dynamic arrays is the only scenario known to the authors where LRM-trees are actually advantageous over the better known Cartesian Trees [42, Sect. 3.1], as only in LRM-trees appending elements to the array results in pure leaf additions to the tree. In a Cartesian Tree, appending a single new element might result in more complicated relinking operations on the tree topology (be it in constant time), and no data structure for constant-time LCAs is known in fully dynamic trees. A final caveat is that one would probably not want to implement this approach due to the complicated data structure for dynamic LCAs – see [8, Sect. 3.2] for alternatives that work better in practice.

6 Computing the LCP-array from the BWT

Next, we will present a LACA that is based on the BWT of S [7]. We start with some prerequisites.

Let ω be a substring of S. The ω-interval is the interval $[i, j]$ such that ω is a common prefix of $S_{\mathsf{SA}[i]}, S_{\mathsf{SA}[i+1]}, \ldots, S_{\mathsf{SA}[j]}$, but neither of $S_{\mathsf{SA}[i-1]}$ nor of $S_{\mathsf{SA}[j+1]}$. For example, in Table 2 one can see that the *miss*-interval is $[6, 6]$, while the *iss*-interval is $[4, 5]$. Since the empty string ε is a common prefix of all suffixes of S, the ε-interval is $[1, n]$. The LACA in Algorithm 4 is based on the procedure *getIntervals*, which has the following functionality: For an ω-interval $[i, j]$, the call *getIntervals*$([i, j])$ returns the list of all $c\omega$-intervals, where $c \in \Sigma$ and $c\omega$ is a substring of S. In our example, *getIntervals* applied to the ε-interval $[1, 12]$ generates the $\$$-interval $[1, 1]$, the i-interval $[2, 5]$, the m-interval $[6, 6]$, the p-interval $[7, 8]$ and the s-interval $[9, 12]$. Beller et al. [7] describe an implementation of the procedure *getIntervals* that is based on the wavelet tree of the BWT of S. A call to *getIntervals*$([i, j])$ takes $O(k \log \sigma)$ time if it returns a list with k intervals (so each interval can be generated in $O(\log \sigma)$ time).

Algorithm 4 Computation of the LCP-array based on the BWT [7].

```
Input: wavelet tree of the BWT
initialize the array LCP$[1, n+1]$ by LCP$[i] \leftarrow \perp$ for all $i$
$\ell \leftarrow -1$
initialize an empty queue $Q$
$enqueue(Q, [1, n])$          /* add the $\varepsilon$-interval to $Q$ */
$size \leftarrow 1$          /* initial size of $Q$ */
while $Q$ is not empty do
    if $size = 0$ then
        $\ell \leftarrow \ell + 1$
        $size \leftarrow |Q|$          /* current size of $Q$ */
    $[lb, rb] \leftarrow dequeue(Q)$
    $size \leftarrow size - 1$
    if LCP$[rb+1] = \perp$ then
        LCP$[rb+1] \leftarrow \ell$          /* case 1 */
        $list \leftarrow getIntervals([lb, rb])$
        foreach $[i, j]$ in $list$ do
            $enqueue(Q, [i, j])$
    else
        nothing to do          /* case 2 */
```

Algorithm 4 shows how the LCP-array of a string S can be obtained solely based on the procedure *getIntervals*. The algorithm maintains a queue Q, which initially contains the ε-interval $[1, n]$. Moreover, the variable ℓ stores the current lcp-value (initially, $\ell = -1$) and *size* memorizes how many elements there are in Q that correspond to the current lcp-value (initially, $size = 1$). Algorithm 4 computes lcp-values in increasing order (first the artificial entry LCP$[n+1] = -1$, then the 0 entries, and so on) as follows: whenever it dequeues an element from Q, say the ω-interval $[lb, rb]$ where $|\omega| = \ell + 1$, it tests whether LCP$[rb+1] = \perp$. If so, it assigns ℓ to LCP$[rb+1]$, generates all non-empty $c\omega$-intervals (where $c \in \Sigma$) and adds them to (the back of) Q. Otherwise, it does nothing.

Let us illustrate the algorithm by computing the artificial -1 entry and all 0 entries in the LCP-array of our example in Table 2. Initially, $\ell = -1$ and the queue Q contains the ε-interval $[1, 12]$. Consequently, the first interval that is pulled from the queue is the ε-interval $[1, 12]$ and *size* is decreased by one. Since LCP$[12+1] = \perp$, case 1 in Algorithm 4 applies. Thus, LCP$[13]$ is set to $\ell = -1$ and *getIntervals*$([1, 12])$ generates the \$-interval $[1, 1]$, the i-interval $[2, 5]$, the m-interval $[6, 6]$, the p-interval $[7, 8]$ and the s-interval $[9, 12]$. These intervals are put into the queue Q. In the second iteration of the while-loop, we have $size = 0$. Therefore, $\ell = -1$ is increased by one and the variable *size* gets the new value $|Q|$. At that point, the *size* of Q is 5; so five intervals correspond to the new lcp-value $\ell = 0$. The second interval that is pulled from the queue is the \$-interval $[1, 1]$ and *size* is decreased by one. Since LCP$[1+1] = \perp$, case 1 in Algorithm 4 applies. Thus, LCP$[2]$ is set to $\ell = 0$ and the i\$-interval $[2, 2]$, which is the only interval in the list returned by *getIntervals*$([1, 1])$, is added to the queue. Next, the i-interval $[2, 5]$ is dequeued (and *size* is decreased by one). Again, case 1 applies because LCP$[5+1] = \perp$. So LCP$[6]$ is set to $\ell = 0$, *getIntervals*$([2, 5])$ returns the list $[[6, 6], [7, 7], [9, 10]]$, and the intervals in the list are added to the queue Q. When the m-interval $[6, 6]$ is dequeued, *size* is decreased by one, LCP$[7]$ is set to $\ell = 0$, but no new interval is added to the queue (observe that *getIntervals* does not generate the \$$m$-interval because $\$m$ is not a substring of S). Then the p-interval is dequeued, *size* is decreased by one, $\mathsf{LCP}[9]$ is set to 0, and the intervals $[3,3]$ and $[8,8]$ (the ip- and the pp-interval) are enqueued. Finally, the s-interval $[9,12]$ is pulled from the queue. Again, *size* is decreased by one; it now has the value 0. Because $\mathsf{LCP}[12+1] = -1$, case 2 in Algorithm 4 applies. In the next iteration of the while-loop, ℓ is increased by one and *size* is set to the current size 6 of Q. At that point in time, the six elements in Q are the following intervals that correspond to the lcp-value $\ell = 1$:

$$[2,2]_{i\$}, [6,6]_{mi}, [7,7]_{pi}, [9,10]_{si}, [3,3]_{ip}, [8,8]_{pp}$$

where the notation $[lb, rb]_\omega$ indicates that the interval $[lb, rb]$ is the ω-interval. The reader is invited to compute all 1 entries in the LCP-array by executing the algorithm by hand.

▶ **Theorem 3.** *Algorithm 4 correctly computes the* LCP*-array.*

Proof. We proceed by induction on ℓ. After the first iteration of the while-loop (which leads to the artificial entry $\mathsf{LCP}[n+1] = -1$), we have $\ell = 0$ and the queue Q contains the c-interval $[lb, rb]$ for every character $c = \Sigma[k]$, where $lb = C[c]+1$ and $rb = C[d]$ with $d = \Sigma[k+1]$. The algorithm sets $\mathsf{LCP}[rb+1] = 0$ unless $rb = n$. This is certainly correct because the suffix $S_{\mathsf{SA}[rb]}$ starts with the character c and the suffix $S_{\mathsf{SA}[rb+1]}$ starts with the character d. Clearly, the LCP-array contains all entries with value 0. Let $\ell > 0$. By the inductive hypothesis, we may assume that Algorithm 4 has correctly computed all lcp-values $< \ell$. Immediately after the value of ℓ was incremented in line 8 of Algorithm 4, the queue Q solely contains intervals that correspond to the lcp-value ℓ (i.e., maximal intervals in which the suffixes share a common prefix of length $\ell+1$). Let the ωa-interval $[lb, rb]$ be in Q, where $|\omega| = \ell$ and $a \in \Sigma$. If $\mathsf{LCP}[rb+1] = \perp$, then we know from the induction hypothesis that $\mathsf{LCP}[rb+1] \geq \ell$, i.e., ω is a common prefix of the suffixes $S_{\mathsf{SA}[rb]}$ and $S_{\mathsf{SA}[rb+1]}$. On the other hand, ωa is a prefix of $S_{\mathsf{SA}[rb]}$ but not of $S_{\mathsf{SA}[rb+1]}$. Consequently, ω is the longest common prefix of $S_{\mathsf{SA}[rb]}$ and $S_{\mathsf{SA}[rb+1]}$. Hence Algorithm 4 assigns the correct value ℓ to $\mathsf{LCP}[rb+1]$.

We still have to prove that all entries of the LCP-array with value ℓ are really set. So let k, $0 \leq k < n$, be an index with $\mathsf{LCP}[k+1] = \ell$. Since $\ell > 0$, the longest common prefix of $S_{\mathsf{SA}[k]}$ and $S_{\mathsf{SA}[k+1]}$ can be written as $c\omega$, where $c \in \Sigma$, $\omega \in \Sigma^*$, and $|\omega| = \ell - 1$. Let $c\omega a$ be the length $\ell+1$ prefix of $S_{\mathsf{SA}[k]}$ and $c\omega d$ be the length $\ell+1$ prefix of $S_{\mathsf{SA}[k+1]}$, where $a \neq d$. Because ωa is a prefix of $S_{\mathsf{SA}[k]+1}$ and ωd is a prefix of $S_{\mathsf{SA}[k+1]+1}$, it follows that ω is the longest common prefix of $S_{\mathsf{SA}[k]+1}$ and $S_{\mathsf{SA}[k+1]+1}$. Let $[i,j]$ be the ω-interval, p be the index with $\mathsf{SA}[p] = \mathsf{SA}[k]+1$, and q be the index with $\mathsf{SA}[q] = \mathsf{SA}[k+1]+1$. Clearly, $i \leq p < q \leq j$. Let t, $p < t \leq q$, be the smallest index at which the corresponding suffix does not start with ωa; see Fig. 4. Consequently, $\mathsf{LCP}[t] = |\omega| = \ell - 1$. According to the inductive hypothesis, Algorithm 4 assigns the value $\ell - 1$ to $\mathsf{LCP}[t]$. Therefore, *getIntervals* is called with the ωa-interval $[s, t-1]$ for some $s \leq t$. Since ωa is a prefix of $S_{\mathsf{SA}[p]}$ and $\mathsf{BWT}[p] = c$, it follows that the $c\omega a$-interval, say $[lb, rb]$, is not empty. Moreover, $rb = k$ because $\mathsf{BWT}[r] \neq c$ for all $p < r < q$. Thus, $[lb, k]$ is in the *list* returned by *getIntervals*$([s, t-1])$. Hence it is added to the queue Q. At some point in time, $[lb, k]$ will be removed from Q and $\mathsf{LCP}[k+1]$ will be set to ℓ. ◀

▶ **Theorem 4.** *Algorithm 4 has a worst-case time complexity of* $O(n \log \sigma)$.

Proof. We use an amortized analysis to prove that each of the cases 1 and 2 can occur at most n times. Case 1 occurs as often as an entry of the LCP-array is filled, and this happens exactly n times. It remains for us to analyze how often case 2 can occur. We claim that

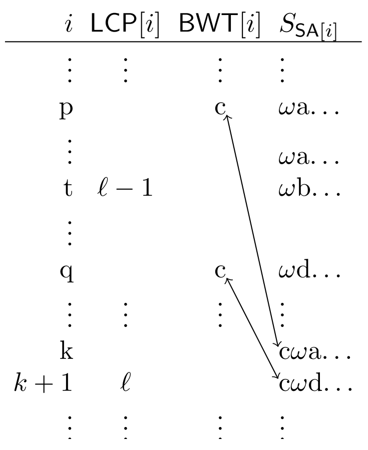


Figure 4 Correctness of Algorithm 4.

for a fixed position j, $1 \leq j \leq n$, there is at most one substring $\omega = S[i, j]$ ending at j for which the ω-interval $[lb, rb]$ belongs to case 2. If i is the largest position with $\omega = S[i, j]$ such that the ω-interval $[lb, rb]$ belongs to case 2, then none of the left-extensions of ω is generated. More precisely, none of the ω'-intervals, where $\omega' = S[i', j]$ with $1 \leq i' < i$, will be enqueued. This proves the claim. As there are only n possibilities for j, it follows that case 2 occurs at most n times. In summary, the procedure *getIntervals* can create at most $2n$ intervals because every interval belongs to exactly one case. Each interval can be generated in $O(\log \sigma)$ time, so the run-time of Algorithm 4 is $O(n \log \sigma)$. ◀

It is shown in [7] that Algorithm 4 can be implemented in such a way that it needs only $4n$ bits for the storage of the intervals in the queue. Furthermore, $n \log \sigma + o(n \log \sigma)$ bits are required for the wavelet tree and $4n$ bytes are needed for the LCP-array itself. Under the assumption that $\log \sigma = 8$, the algorithm requires $5.5n$ bytes plus $o(n)$ bits of RAM in total.

In essence, Algorithm 4 enumerates all substring-intervals in a breadth-first manner and Beller et al. [7] demonstrated how this can be done space efficiently. Belazzougui [6] showed that the enumeration can also be done space-efficiently in a depth-first manner. Prezza and Rosone [36] analyzed the space consumption of the two alternatives and came to the conclusion that Beller et al.'s algorithm is more space-efficient on large alphabets whereas Belazzougui's algorithm is more space-efficient on small alphabets. Prezza and Rosone designed an algorithm which is the combination of the two alternatives: if the alphabet size σ is larger than $\frac{\sqrt{n}}{\log^2 n}$, they apply the algorithm of Beller et al. and otherwise they use Belazzougui's algorithm to compute the LCP-array. The resulting algorithm fills the LCP-array in $O(n \log \sigma)$ time, using only $o(n \log \sigma)$ bits of working space on top of the BWT and the LCP-array [36, Theorem 3].

7 Final Remarks

We have only scratched the surface of LCP-array construction and reviewed some classic algorithms, but we did not cover results on the efficient storage of lcp-values, such as sampled [41] or compressed [13, 39] LCP-arrays. There are also variants of LCP-arrays, such as for labeled graphs [3], spectra [2], or Wheeler DFAs [11], or with mismatches [32], which were all not covered here. We also did not look at more advanced models of computation, like parallel shared or distributed memory [17, 40] or external memory [8, 21]. Finally, LCP-arrays for collections of strings have also been considered [12, 28]. All citations in this subsection should only be seen as a point of entry for further literature research, as they are far from exhaustive. We close this review article by mentioning that for large repetitive data sets the currently best practical algorithm is based on *prefix free parsing* [38, Lemma 2].

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004. doi:10.1016/S1570-8667(03)00065-0.

2 Jarno N. Alanko, Elena Biagi, and Simon J. Puglisi. Longest common prefix arrays for succinct k-spectra. In Franco Maria Nardini, Nadia Pisanti, and Rossano Venturini, editors, *String Processing and Information Retrieval - 30th International Symposium, SPIRE 2023, Pisa, Italy, September 26-28, 2023, Proceedings*, volume 14240 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2023. doi:10.1007/978-3-031-43980-3_1.

3 Jarno N. Alanko, Davide Cenzato, Nicola Cotumaccio, Sung-Hwan Kim, Giovanni Manzini, and Nicola Prezza. Computing the LCP array of a labeled graph. In Shunsuke Inenaga and Simon J. Puglisi, editors, *35th Annual Symposium on Combinatorial Pattern Matching, CPM 2024, June 25-27, 2024, Fukuoka, Japan*, volume 296 of *LIPIcs*, pages 1:1–1:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CPM.2024.1.

4 Alberto Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, pages 85–96. Springer, 1985.

5 Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. LRM-trees: Compressed indices, adaptive sorting, and compressed permutations. *Theor. Comput. Sci.*, 459:26–41, 2012. doi:10.1016/J.TCS.2012.08.010.

6 D. Belazzougui. Linear time construction of compressed text indices in compact space. In *Proc. 46th Annual ACM Symposium on Theory of Computing*, pages 148–193, 2014.

7 Timo Beller, Simon Gog, Enno Ohlebusch, and Thomas Schnattinger. Computing the longest common prefix array based on the Burrows-Wheeler transform. *Journal of Discrete Algorithms*, 18:22–31, 2013. doi:10.1016/J.JDA.2012.07.007.

8 Timo Bingmann, Johannes Fischer, and Vitaly Osipov. Inducing suffix and LCP arrays in external memory. *ACM J. Exp. Algorithmics*, 21(1):2.3:1–2.3:27, 2016. doi:10.1145/2975593.

9 M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. Research Report 124, Digital Systems Research Center, 1994.

10 Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005. doi:10.1137/S0097539700370539.

11 Alessio Conte, Nicola Cotumaccio, Travis Gagie, Giovanni Manzini, Nicola Prezza, and Marinella Sciortino. Computing matching statistics on Wheeler DFAs. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Data Compression Conference, DCC 2023, Snowbird, UT, USA, March 21-24, 2023*, pages 150–159. IEEE, 2023. doi:10.1109/DCC55655.2023.00023.

12 Lavinia Egidi, Felipe A. Louza, Giovanni Manzini, and Guilherme P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms Mol. Biol.*, 14(1):6:1–6:15, 2019. doi:10.1186/S13015-019-0140-0.

13 Johannes Fischer. Wee LCP. *Inf. Process. Lett.*, 110(8-9):317–320, 2010. doi:10.1016/J.IPL.2010.02.010.

14 Johannes Fischer. Inducing the LCP-array. In Frank Dehne, John Iacono, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures - 12th International Symposium, WADS 2011, New York, NY, USA, August 15-17, 2011. Proceedings*, volume 6844 of *Lecture Notes in Computer Science*, pages 374–385. Springer, 2011. doi:10.1007/978-3-642-22300-6_32.

15 Johannes Fischer and Volker Heun. Finding range minima in the middle: Approximations and applications. *Math. Comput. Sci.*, 3(1):17–30, 2010. doi:10.1007/S11786-009-0007-8.

16 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. `doi:10.1137/090779759`.

17 Patrick Flick and Srinivas Aluru. Parallel distributed memory construction of suffix and longest common prefix arrays. In Jackie Kern and Jeffrey S. Vetter, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 16:1–16:10. ACM, 2015. `doi:10.1145/2807591.2807609`.

18 Simon Gog and Enno Ohlebusch. Compressed suffix trees: Efficient computation and storage of LCP-values. *Journal of Experimental Algorithmics*, 18:2.1, 2013. `doi:10.1145/2444016.2461327`.

19 Gaston H. Gonnet, Ricardo Baeza-Yates, and Tim Snider. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures and Algorithms*, chapter 5, pages 66–82. Prentice-Hall, Englewood Cliffs, NJ, 1992.

20 Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 251–260, 2003. `doi:10.1109/SFCS.2003.1238199`.

21 Juha Kärkkäinen and Dominik Kempa. Better external memory LCP array construction. *ACM J. Exp. Algorithmics*, 24(1):1.3:1–1.3:27, 2019. `doi:10.1145/3297723`.

22 Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In *Proc. 20th Annual Symposium on Combinatorial Pattern Matching*, volume 5577 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2009. `doi:10.1007/978-3-642-02441-2_17`.

23 Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003. `doi:10.1007/3-540-45061-0_73`.

24 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. `doi:10.1145/1217856.1217858`.

25 Toru Kasai, Hiroki Arimura Gunho Lee, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2001. `doi:10.1007/3-540-48194-X_17`.

26 Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199. Springer-Verlag, 2003. `doi:10.1007/3-540-44888-8_14`.

27 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. Springer-Verlag, 2003. `doi:10.1007/3-540-44888-8_15`.

28 Felipe A. Louza, Guilherme P. Telles, Steve Hoffmann, and Cristina Dutra de Aguiar Ciferri. Generalized enhanced suffix array construction in external memory. *Algorithms Mol. Biol.*, 12(1):26:1–26:16, 2017. `doi:10.1186/S13015-017-0117-9`.

29 Veli Mäkinen. Compact suffix array – a space-efficient full-text index. *Fundamenta Informaticae*, 56(1-2):191–210, 2003. URL: `http://content.iospress.com/articles/fundamenta-informaticae/fi56-1-2-12`.

30 Udi Manber and Gerne Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. `doi:10.1137/0222058`.

31 Giovanni Manzini. Two space saving tricks for linear time LCP array computation. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 372–383. Springer, 2004. `doi:10.1007/978-3-540-27810-8_32`.

32 Giovanni Manzini. Longest common prefix with mismatches. 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 299–310. Springer, 2015. doi:10.1007/978-3-319-23826-5_29.

33 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011. doi:10.1109/TC.2010.188.

34 Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.

35 Jannik Olbrich, Enno Ohlebusch, and Thomas Büchler. Generic non-recursive suffix array construction. *ACM Transactions on Algorithms*, 20(2), 2024. doi:10.1145/3641854.

36 Nicola Prezza and Giovanna Rosone. Space-efficient computation of the LCP array from the Burrows-Wheeler transform. In *Proc. 30th Annual Symposium on Combinatorial Pattern Matching*, volume 128 of *Leibniz International Proceedings in Informatics*, pages 7:1–7:18, 2019. doi:10.4230/LIPICS.CPM.2019.7.

37 Simon J. Puglisi, Bill Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):Article 4, 2007.

38 Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A pangenomic index for finding maximal exact matches. *J. Comput. Biol.*, 29(2):169–187, 2022. doi:10.1089/CMB.2021.0290.

39 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.

40 Julian Shun. Fast parallel computation of longest common prefixes. In Trish Damkroger and Jack J. Dongarra, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, pages 387–398. IEEE Computer Society, 2014. doi:10.1109/SC.2014.37.

41 Jouni Sirén. Sampled longest common prefix array. In Amihood Amir and Laxmi Parida, editors, *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, volume 6129 of *Lecture Notes in Computer Science*, pages 227–237. Springer, 2010. doi:10.1007/978-3-642-13509-5_21.

42 Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980. doi:10.1145/358841.358852.

43 Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.