# LPF computation revisited

Maxime Crochemore, Lucian Ilie, Costas S. Iliopoulos, Marcin Kubica, Wojciech Rytter, Tomasz Walen

# LPF computation revisited*

Maxime Crochemore[1,4], Lucian Ilie**[2], Costas S. Iliopoulos[1,5], Marcin Kubica[3], Wojciech Rytter***[3,6], and Tomasz Waleń[3]

[1] Dept. of Computer Science, King's College London, London WC2R 2LS, UK
[2] Dept. of Computer Science, University of Western Ontario, London, Ontario, N6A 5B7, Canada
[3] Institute of Informatics, Warsaw University, ul. Banacha 2, 02-097 Warszawa, Poland
[4] Université Paris-Est, France
[5] Digital Ecosystems & Business Intelligence Institute, Curtin University of Technology, Perth WA 6845, Australia
[6] Dept. of Math. and Informatics, Copernicus University, Torun, Poland

**Abstract.** We present efficient algorithms for storing past segments of a text. They are computed using two previously computed read-only arrays (SUF and LCP) composing the Suffix Array of the text. They compute the maximal length of the previous factor (subword) occurring at each position of the text in a table called LPF. This notion is central both in many conservative text compression techniques and in the most efficient algorithms for detecting motifs and repetitions occurring in a text.
The main results are: a linear-time algorithm that computes explicitly the permutation that transforms the LCP table into the LPF table; a time-space optimal computation of the LPF table; and an $O(n \log n)$ strong in-place computation of the LPF table.
**Keywords**: longest previous factor, suffix array, Ziv-Lempel factorisation, text compression, detection of repetitions.

## 1 Longest Previous Factor

We consider a string $y$ of length $n$ on the alphabet $A$: $y = y[0\,..\,n-1]$. The problem is to compute the Longest Previous Factor table defined, for $0 \le i < n$, by

$$\mathsf{LPF}[i] = \max\{k \mid y[i\,..\,i+k-1] \text{ occurs at a position } j < i\}.$$

For example, the text $y =$ abaababababbabbb has the following LPF table.

| position $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $y[i]$ | a | b | a | a | b | a | b | a | b | b | a | b | b | b |
| LPF$[i]$ | 0 | 0 | 1 | 3 | 2 | 4 | 3 | 2 | 1 | 4 | 3 | 2 | 2 | 1 |

* Research supported in part by the Royal Society, UK.
** Research supported in part by NSERC.
*** Supported by grant N206 004 32/0806 of the Polish Ministry of Science and Higher Education.

The problem may be regarded as an extension of the Ziv-Lempel factorisation (LZ77) of a string as defined in [20]. A string $y$ is decomposed into factors, called phrases, $u_0$, $u_1$, ..., $u_k$, for which $y = u_0u_1 \cdots u_k$ and defined informally by: $u_i = va$ where $a$ is a letter and $v$ is the longest segment of $u_0u_1 \cdots u_i$ occurring both at position $|u_0u_1 \cdots u_{i-1}|$ and before it in $y$. The factorisation is used in several adaptive compression methods which encode carefully re-occurrences of phrases by pointers or integers (see [18] or [19]). The factorisation yields more powerful compressors than the factorisation in [21] (called LZ78) where a phrase is an extension by a single letter of a previous phrase. But the LZ77 factorisation is more difficult to compute.

It is clear that the LZ77 factorisation comes readily when the LPF table is available, which is a remarkable application of the table (see [5]). This implies a linear-time solution for LZ77 factorisation on integer alphabets. Previous solutions using a Suffix Tree [17] or a Suffix Automaton [2] of the text not only run in time $O(|y| \log |A|)$ but these data structures are more space-expensive than the Suffix Array.

A slight variant of string parsing, whose relation with the LZ77 factorisation is analysed in [1], plays an important role in String Algorithms. The intuitive reason is that, when processing a string on-line, the work done on an element of the factorisation can usually be skipped because already done on one of its previous occurrences. A typical application of this idea resides in algorithms to compute repetitions in strings (see [2, 15, 14]). For example, the algorithm in [14] reports all maximal repetitions (called runs) occurring in a string in $O(|y| \log |A|)$ time. It runs in linear time if a Suffix Array is used instead of a Suffix Tree [4]. Indeed the technique seems to be the only technique that leads to linear-time algorithms independently of the alphabet size for this type of question.

Suffix Arrays provide an ideal data structure to solve many questions requiring an index on all the factors of a string. Introduced by Manber and Myers [16] the structure can be built in linear-time by different methods [10, 12, 13, 8] for sorting the suffixes of the text possibly adding the method of [11] to compute the Longest Common Prefix table. The result holds if the text is drawn from an integer alphabet, that is, if the alphabet of the text can be sorted in linear time (otherwise the $\Omega(n \log n)$ lower bound for sorting applies).

The notion of Longest Previous Factor has been introduced by Franek et al. as part of their concept of a Quasi Suffix Array (their $\pi$ array is the LPF table) in [7], where the authors presented a direct computation running in $O(n \log n)$ average time. A naive computation of the LPF table, either on the text itself or on its Suffix Array, as done by the algorithm LPF-NAIVE in Section 5, leads to quadratic effective running time on many inputs.

In this article we intensively use the Suffix Array of the text to be processed, and we consider only linear-time solutions. A graphical representation of the Suffix Array structure helps understand the design of the algorithms. The Longest Previous Factor is coined in [4], where a linear-time computation is described and applied to LZ77 factorisation. Another version running on-line on the Suffix Array of the text and requiring only $O(\sqrt{n})$ extra memory space is shown in [5].

We improve on the previous results and show that the computation of the Longest Previous Factor table of a text from its Suffix Array can be implemented to run in linear time with only a constant amount of additional memory space. Thus, the method is time-space optimal.

The next section introduces the necessary material for the design of Longest Previous Factor computations. First, an algorithm similar to the one in [4] is described. The algorithm is regarded in Section 3 as using the Suffix Array like a sorting network. Section 4 shows how the computation can be done on-line on the Suffix Array and Section 5 describes the time-space optimal algorithm for constructing the table.

## 2 Using a Suffix Array

The Suffix Array of the text $y$ is a data structure used for indexing its content. It comprises two tables that we denote $\mathsf{SUF}$ and $\mathsf{LCP}$ and that are defined as follows.

The table $\mathsf{SUF}$ stores the list of position on $y$ associated with the sorted list of its suffixes in increasing lexicographic order. That is, the table is such that

$$y[\mathsf{SUF}[0]\,..\,n-1] < y[\mathsf{SUF}[1]\,..\,n-1] < \cdots < y[\mathsf{SUF}[n-1]\,..\,n-1].$$

Thus, indices on $\mathsf{SUF}$ are ranks of the suffixes in their sorted list.

The second table $\mathsf{LCP}$ is also indexed by the ranks of suffixes and stores the longest common prefixes between consecutive suffixes in the sorted list. Let $\mathsf{lcp}(i,j) =$ longest common prefix of $y[i\,..\,n-1]$ and $y[j\,..\,n-1]$, for two positions $i$ and $j$ on $y$. Then, $\mathsf{LCP}[0] = 0$ and, for $0 < r < n$, we set

$$\mathsf{LCP}[r] = |\mathsf{lcp}(\mathsf{SUF}[r-1], \mathsf{SUF}[r])|.$$

(The actual Suffix Array contains indeed about $n$ additional LCP values used for binary searching the suffixes.)

*Example 1.* For the text $y =$ `abaababababbabbb` we get the Suffix Array:

| rank $r$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\mathsf{SUF}[r]$ | 2 | 0 | 3 | 5 | 7 | 10 | 13 | 1 | 4 | 6 | 9 | 12 | 8 | 11 |
| $\mathsf{LCP}[r]$ | 0 | 1 | 3 | 4 | 2 | 3 | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 2 |

The computation of the Suffix Array of $y$ can be done in time $O(n \log n)$ in the comparison model [16] (see [3, 6, 9]). If the text is on an alphabet of integers in the range $[0, n^c]$ for some constant $c$, the Suffix Array can be built in time $O(n)$ [10, 12, 13, 8] (see also [3]).

*Graphic representation* The Suffix Array of the text $y$ has a nice graphic representation that helps understand the algorithms computing the $\mathsf{LPF}$ table. The abscissae axis refers to ranks of suffixes and the ordinates axis refers to their positions. The sorted list of suffixes is plotted by their positions, and consecutive positions are linked by an edge whose label is the associated LCP value. Figure 1 shows the Suffix Array representation for the text of Example 1.

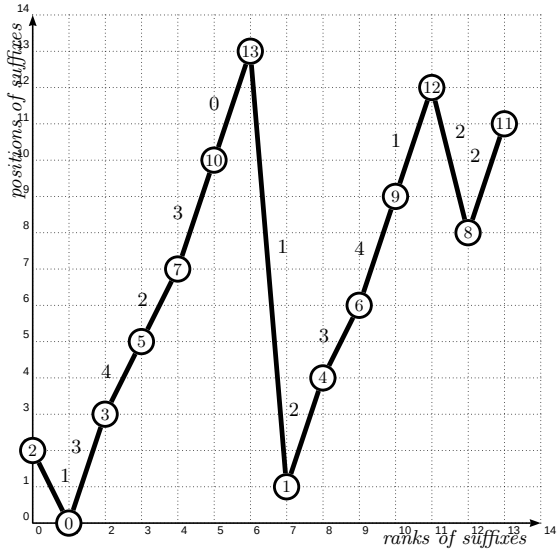


**Fig. 1.** Graph of the permutation of positions on `abaababababbabbb` (Example 1) in the lexicographic order of its suffixes. Labels of edges are Longest Common Prefix lengths between consecutive suffixes.

*Simple algorithm* The $\mathsf{LCP}$ table satisfies a simple property consequence of the lexicographic order: the LCP value between two positions at ranks $r$ and $t$, $r < t$, is the minimal value in $\mathsf{LCP}[r+1\,..\,t]$.

For a rank $r$, let us define $\mathsf{prev}[r]$ as the largest rank $s$, $s < r$, for which $\mathsf{SUF}[s] < \mathsf{SUF}[r]$ if it exists, and as $-1$ otherwise. Let us also define the dual notion $\mathsf{next}[r]$ as the smallest rank $t$, $r < t$, for which $\mathsf{SUF}[t] < \mathsf{SUF}[r]$ if it exists, and as $n$ otherwise. The above property implies that to compute $\mathsf{LPF}[\mathsf{SUF}[r]]$ there is no need to look at ranks smaller than $\mathsf{prev}[r]$ or greater than $\mathsf{next}[r]$, that is

$$\mathsf{LPF}[\mathsf{SUF}[r]] = \max\{\, |\mathsf{lcp}(\mathsf{SUF}[r], \mathsf{SUF}[\mathsf{prev}[r]])|,\ |\mathsf{lcp}(\mathsf{SUF}[r], \mathsf{SUF}[\mathsf{next}[r]])| \,\} \quad (1)$$

where undefined LCP values are set to 0. In particular, if a position $i$ is a "peak" at rank $r$ in the graphic representation of the Suffix Array ($\mathsf{SUF}[r] = i$) we get

$$\mathsf{LPF}[i] = \max\{\mathsf{LCP}[r], \mathsf{LCP}[r+1]\}.$$

And the minimum of the two values is the LCP between positions at ranks $r-1$ and $r+1$ if defined. This gives the idea underlying the next algorithm.

Figure 2 illustrates a step of the algorithm when the input is the text of Example 1. The algorithm LPF-SIMPLE computes the two tables $\mathsf{prev}$ and $\mathsf{next}$, which implement indeed a double-linked list of ranks. The algorithm runs in linear time but requires several extra arrays in addition to its input and output. Namely, the array $\mathsf{ISU}$, inverse of $\mathsf{SUF}$, that provides the rank of a position, and the two arrays $\mathsf{prev}$ and $\mathsf{next}$.

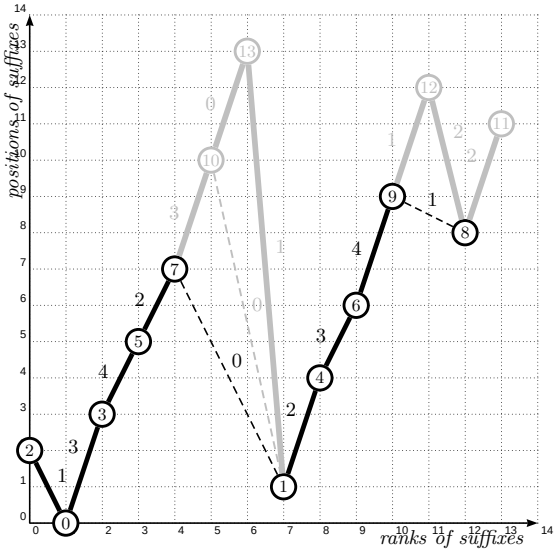


**Fig. 2.** Illustration of the run of the algorithm LPF-SIMPLE on the text `abaababababbabbb` (see Figure 1) after processing positions 13, 12, 11, 10. Gray positions and gray edges are no longer considered. The next step is to process position 9 at rank 10. According to the weights of edges pending from 9, we have $\mathsf{LPF}[9] = \max\{4, 1\} = 4$. Positions 6 and 8 will be linked, through the prev and next arrays on their ranks, with an edge of weight $\min\{4, 1\} = 1$, which is indeed $\mathsf{LPF}[8]$.

LPF-SIMPLE($\mathsf{SUF}, \mathsf{LCP}, n$)

```
 1  LCP-copy ← LCP
 2  LCP-copy[n] ← 0
 3  ISU ← inverse SUF
 4  for r ← 0 to n − 1 do
 5        prev[r] ← r − 1
 6        next[r] ← r + 1
 7  for i ← n − 1 downto 0 do
 8        r ← ISU[i]
 9        LPF[i] ← max{LCP-copy[r], LCP-copy[next[r]]}
10        LCP-copy[next[r]] ← min{LCP-copy[r], LCP-copy[next[r]]}
11        if prev[r] ≥ 0 then
12              next[prev[r]] ← next[r]
13        if next[r] < n then
14              prev[next[r]] ← prev[r]
15  return LPF
```

**Proposition 1.** *The algorithm* LPF-SIMPLE *computes the LPF array of a text of length $n$ from its Suffix Array in linear time and space. It requires $3n + c$ integer cells in addition to its input and output.*

The array next can alternatively be pre-computed by the following procedure whose linear-time behaviour is an interesting exercise left to the reader. The procedure also computes the array $\mathsf{LPF}_>$ defined in [4] and that accounts for larger ranks only. It is defined, for $r = 0, \ldots, n-1$, by

$$\mathsf{LPF}_>[r] = |\mathsf{lcp}(\mathsf{SUF}[r], \mathsf{SUF}[\mathsf{next}[r]])|.$$

```
NEXT(SUF, LCP, n)
 1  next[n − 1] ← n
 2  LPF>[n − 1] ← 0
 3  for r ← n − 2 downto 0 do
 4        t ← r + 1
 5        ℓ ← LCP[t]
 6        while t < n and SUF[t] > SUF[r] do
 7              ℓ ← min{ℓ, LPF>[t]}
 8              t ← next[t]
 9        next[r] ← t
10        LPF>[r] ← ℓ
11  return next, LPF>
```

The computation of the dual prev and $\mathsf{LPF}_<$ arrays is done symmetrically. When both arrays $\mathsf{LPF}_<$ and $\mathsf{LPF}_>$ are available, the computation of the LPF table is an application of identity 1 that rewrites, for position $i$ at rank $r$, as

$$\mathsf{LPF}[i] = \max(\mathsf{LPF}_>[r], \mathsf{LPF}_<[r]).$$

## 3 Sorting network

It has been noticed in [4] that the content of the LPF table is the same as that of the LCP table up to some permutation. A question arises then: what permutation is it? Does it depend on the text, its Suffix Array or maybe just its LCP array? It turns out that, for fixed SUF and LCP arrays, it does not depend on the actual text. It is possible to construct such a sorting network, whose shape depends only on the Suffix Array, that transforms the LCP array into the LPF table. This observation leads to the algorithm LPF-SORTING, producing LPF by permutating the elements of LCP.

In the algorithm LPF-SORTING below we assume that the table next introduced in the previous section has been pre-computed by the procedure NEXT (in which instructions related to $\mathsf{LPF}_>$ are useless and may be removed, as well as the parameter LCP). Table next is used to compute, for each position $i$, the next closest position $\mathsf{nextp}[i]$ in the Suffix Array that is smaller than $i$, that is,

$$\mathsf{nextp}[\mathsf{SUF}[r]] = \mathsf{SUF}[\min\{t \mid t > r \text{ and } \mathsf{SUF}[t] < \mathsf{SUF}[r]\}].$$

Initially, the LPF table is a copy of the LCP array permuted according to the SUF table. The algorithm sorts the array by permuting its elements to get the LPF table.

```
LPF-SORTING(SUF, LCP, n)
1  SUF[n] ← −1
2  for r ← 0 to n − 1 do
3        LPF[SUF[r]] ← LCP[r]
4        nextp[SUF[r]] ← SUF[next[r]]
5  for i ← n − 1 downto 0 do
6        if (nextp[i] ≥ 0) and (LPF[i] < LPF[nextp[i]]) then
7              EXCHANGE(LPF[i], LPF[nextp[i]])
8  return LPF
```

Note that the elements of LPF are exchanged (lines 6–7) only if they are not in increasing order. Which elements are compared, depends on values in nextp, and this in turn depends only on the Suffix Array. So, for a given Suffix Array, one can construct a sorting network implementing lines 5–7 of the algorithm LPF-SORTING. Hence, the following proposition holds:

**Proposition 2.** *For a given Suffix Array of a text, there exists a sorting network processing a sequence of $n$ numbers in such a way that it transforms the LCP table into the LPF table. Moreover, the shape of the sorting network depends only on the Suffix Array, but not on its LCP table nor on the actual text.*

As it is done for any sorting procedure, instead of directly computing the LPF table, the algorithm LPF-SORTING can equivalently produce explicitly the permutation $\pi$ to transform the LCP array into the LPF table, that is, the permutation that satisfies $\mathsf{LPF}[i] = \mathsf{LCP}[\pi[i]]$.

The algorithm LPF-SORTING uses only one integer array in addition to its input and output since the next table it uses is substituted for the nextp table. This yields the next statement.

**Proposition 3.** *The algorithm* LPF-SORTING *computes the LPF table of a text of length $n$ from its Suffix Array in linear time and space. It requires $n+c$ integer cells in addition to its input and output.*

## 4 On-line computation

Techniques of the previous sections to compute the LPF table are simple but space consuming. In this section and the next one we address this issue. We show that a computation on-line on the Suffix Array using a stack reduces the memory space to only $O(\sqrt{n})$ for a text of length $n$.

The design of the on-line computation still relies on the property used for the algorithm of Section 2 and related to "peaks" (see lines 6-8 below). It relies additionally on another straightforward property that we describe now. Assume that a position $\mathsf{SUF}[r]$ at rank $r$ satisfies $\mathsf{LCP}[r] \geq \mathsf{LCP}[r+1]$. Then, no position after it in the list can provide a larger LCP value and therefore we get $\mathsf{LPF}[\mathsf{SUF}[r]] = \mathsf{LCP}[r]$. This is implemented in lines 10-11 of the algorithm LPF-ON-LINE.

Note that lines 15 to 18 may be removed from the algorithm LPF-ON-LINE if the Suffix Array can be extended to rank $n$ and initialised to $\mathsf{SUF}[n] = -1$ and $\mathsf{LCP}[n] = 0$. But we prefer the present design that is compatible with the algorithm of the next section.

```
LPF-ON-LINE(SUF, LCP, n)
 1  EMPTYSTACK(S)
 2  for r ← 0 to n − 1 do
 3       r-lcp ← LCP[r]
 4       while not EMPTY(S) do
 5            (t, t-lcp) ← TOP(S)
 6            if SUF[r] < SUF[t] then
 7                 LPF[SUF[t]] ← max{t-lcp, r-lcp}
 8                 r-lcp ← min{t-lcp, r-lcp}
 9                 POP(S)
10            elseif (SUF[r] > SUF[t]) and (r-lcp ≤ t-lcp) then
11                 LPF[SUF[t]] ← t-lcp
12                 POP(S)
13            else break
14       PUSH(S, (r, r-lcp))
15  while not EMPTY(S) do
16       (t, t-lcp) ← TOP(S)
17       LPF[SUF[t]] ← t-lcp
18       POP(S)
19  return LPF
```

*Stack size* The extra memory space used by the algorithm LPF-ON-LINE to compute the LPF table of a text is occupied by the stack and a constant number of integer variables. To evaluate the total size required by the algorithm it is then important to determine the maximal size of the stack for a text of length $n$. It is proved in [5] that this quantity is $O(\sqrt{n})$.

For most values of $n$ there are plenty of texts for which the stack reaches its maximal size. But if $n$ is of the form $k(k+1)/2$, that is, if it is the sum of the first $k$ positive integers, then there is a unique string on the alphabet $\{\mathtt{a}, \mathtt{b}\}$ (with $\mathtt{a} < \mathtt{b}$) giving the maximal size stack. This word is $\mathtt{aabab}^2 \cdots \mathtt{ab}^{k-1}$ and the maximal stack size is $k$.

The next table shows maximal stack sizes for texts of lengths 4 to 19:

| length $n$ | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| max-stack-size | 2 | 2 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 6 | 6 |

**Proposition 4.** *The algorithm* LPF-ON-LINE *computes the* LPF *array of a text of length $n$ from its Suffix Array in linear time and $O(\sqrt{n})$ space. It requires less than $2\sqrt{2n} + c$ integer cells in addition to its input and output.*

## 5 Time-space optimal implementation

In this section we show that the computation of the LPF table of a text can be implemented with only constant memory space in addition to the Suffix Array of the text and its LPF table. The underlying property used for this purpose is the $O(\sqrt{n})$ stack size reported in the previous section. The property allows an implementation of the stack inside the LPF table for a sufficiently large part of the text. The rest of the computation for the remaining positions is done in a more time-expensive manner but for a small part of the text. This preserves the linear running time of the whole computation.

LPF-OPTIMAL(SUF, LCP, $n$)
1 ▷ it is assumed that $n \geq 8$
2 $K \leftarrow \lfloor n - 2\sqrt{2n} \rfloor$
3 ▷ next three procedures share the same LPF table
4 LPF-ON-LINE(SUF, LCP, $K$)
5 LPF-NAIVE(SUF, LCP, $K, n$)
6 LPF-ANCHORED(SUF, LCP, $K, n$)
7 **return** LPF

It is rather clear that only constant extra memory space is required to implement the strategy retained by the algorithm LPF-OPTIMAL. The choice of the parameter $K$ is a result of the previous section and is done to let enough space in the LPF array to implement the stack used by the algorithm LPF-ON-LINE.

Although not done here, the choice of the parameter $K$ can be dynamic and done during the algorithm LPF-ON-LINE as $n - 2k - 1$, where $k$ is the size of the stack just before executing line 15. This certainly reduces the actual running time but does not improve its asymptotic evaluation.

In the algorithm LPF-OPTIMAL, the stack of the procedure LPF-ON-LINE is implemented in the LPF table. Access to the table is done via the SUF array. Doing so, the stack is treated like a continuous space LPF[SUF[$K \,..\, n-1$]]. Elements are stored sequentially so that the elementary stack operations (empty, top, push, pop) are all executed in constant time. Therefore, the running time of the first step is $O(n)$ (indeed $O(K)$) as for the algorithm LPF-ON-LINE.

*Example 2.* The next table shows the content of the LPF array at two stages of the run of the procedure LPF-ON-LINE on Example 1: (i) immediately after processing rank 5 and (ii) at the end of the first step.

| | rank $r$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (i) | LPF[SUF[$r$]] | 1 | | 3 | 4 | | | | | 5 | 3 | 4 | 2 | 1 | 0 |
| (ii) | LPF[SUF[$r$]] | 1 | 0 | 3 | 4 | 2 | 3 | 1 | | | | 8 | 2 | 7 | 0 |

Row (i) shows that LPF values of positions 2, 3, 5 at respective ranks 0, 2, 3 have already been computed. The part LPF[SUF[$8 \,..\, 13$]] of the table stores the content of the stack: $((1,0), (4,2), (5,3))$. Row (ii) shows that values have already been computed for positions at ranks 0 to 6. The content of the stack is $((7,0), (8,2))$. On this example, $K$ could have been set dynamically to 5.

*Naive computation* The second step of the algorithm LPF-OPTIMAL processes the Suffix Array from rank $K$. For each rank $r$, the values $\mathsf{prev}[r]$, $\mathsf{next}[r]$ and the corresponding LCP values are computed starting from $r$ and going backward and forward, respectively. The code is given below for the sake of completeness but it contains no algorithmic high value.

The process takes $O(n-K)$ time per rank, and hence the total running time of the step is $O((n-K)^2)$, which is $O(n)$ since $K = n - 2\sqrt{2n}$.

```
LPF-NAIVE(SUF, LCP, K, n)
 1  LCP[n] ← 0
 2  for r ← K to n − 1 do
 3       left ← LCP[r]
 4       s ← r − 1
 5       while s ≥ K and SUF[s] > SUF[r] do
 6            left ← min{left, LCP[s]}
 7            s ← s − 1
 8       if s = K − 1 then
 9            left ← 0
10       t ← r + 1
11       right ← LCP[t]
12       while t < n and SUF[t] > SUF[r] do
13            t ← t + 1
14            right ← min{right, LCP[t]}
15       LPF[SUF[r]] ← max{left, right}
16  return LPF
```

*Completing the computation* The first two steps of the algorithm LPF-OPTIMAL process independently two segments of the Suffix Array. The last step consists in joining their results, which requires updating some LPF values. Indeed, for a rank $r$, $r < K$, $\mathsf{next}[r]$ can be in $[K, n-1]$, which implies that the computation of $\mathsf{LPF}[\mathsf{SUF}[r]]$ might not be achieved. The same phenomenon happens for a rank in the second part, whose associated $\mathsf{prev}$ rank is smaller than $K$.

The next algorithm updates all LPF values and completes the whole computation. It assumes that the $\mathsf{LPF}$ calculation has been done independently on parts $\mathsf{LPF}[\mathsf{SUF}[0\,..\,K-1]]$ and $\mathsf{LPF}[\mathsf{SUF}[K\,..\,n-1]]$ of the array, which is realised by the first two steps on the algorithm LPF-OPTIMAL.

The running time of this last step LPF-ANCHORED is obviously linear, $O(n)$, as are the other steps of the algorithm LPF-OPTIMAL.

The first conclusion of the section is the following statement.

**Theorem 1.** *The Longest Previous Factor table of a text of length $n$ on an integer alphabet can be built from its Suffix Array in time $O(n)$ (independently of the alphabet size) with a constant amount of extra memory space.*

```
LPF-ANCHORED(SUF, LCP, K, n)
 1  s ← K − 1
 2  t ← K
 3  ℓ ← LCP[K]
 4  while (s ≥ 0) and (t < n) do
 5        ▷ invariant: ℓ = min LCP[s + 1, ..., r]
 6        if SUF[s] < SUF[t] then
 7              LPF[SUF[t]] ← max{LPF[SUF[t]], ℓ}
 8              t ← t + 1
 9              ℓ ← min{ℓ, LCP[t]}
10        else LPF[SUF[s]] ← max{LPF[SUF[s]], ℓ}
11              ℓ ← min{ℓ, LCP[s]}
12              s ← s − 1
13  return LPF
```

The algorithm LPF-OPTIMAL uses the Suffix Array of the input text in a read-only manner but does not use the LPF table in a write-only manner. If this last condition is to be satisfied, the question remains of whether there exists a linear-time LPF table construction running with constant extra space. We get this feature if the algorithm LPF-ANCHORED is applied recursively by dividing the Suffix Array into two equal parts. The running time becomes $O(n \log n)$ in the model of computation allowing priority writes.

**Theorem 2.** *The Longest Previous Factor table of a text of length $n$ on an integer alphabet can be built from its read-only Suffix Array in time $O(n \log n)$ (independently of the alphabet size) with a constant amount of extra memory space and with a write-only output.*

Despite the use of the output as auxiliary storage in the ultimate linear-time algorithm, the series of algorithms described in the article provide a large range of efficient solutions that meet many practical needs.

## 6 Acknowledgements

Authors warmly thank German Tischler for his careful inspection of the algorithms described in the article.

## References

1. J. Berstel and A. Savelli. Crochemore factorization of Sturmian and other infinite words. In R. Kralovic and P. Urzyczyn, editors, *Mathematical Foundations of Computer Science*, volume 4162 of *Lecture Notes in Computer Science*, pages 157–166. Springer, 2006.
2. M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45(1):63–86, 1986.

3. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, Cambridge, UK, 2007. 392 pages.
4. M. Crochemore and L. Ilie. Computing Longest Previous Factor in linear time and applications. *Inf. Process. Lett.*, 106(2):75–80, 2008.
5. M. Crochemore, L. Ilie, and W. F. Smyth. A simple algorithm for computing the Lempel-Ziv factorization. In J. A. Storer and M. W. Marcellin, editors, *18th Data Compression Conference*, pages 482–488. IEEE Computer Society, Los Alamitos, CA, 2008.
6. M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific Publishing, Hong-Kong, 2002. 310 pages.
7. F. Franek, J. Holub, W. F. Smyth, and X. Xiao. Computing quasi suffix arrays. *Journal of Automata, Languages and Combinatorics*, 8(4):593–606, 2003.
8. G. Nong, S. Zhang, and W. H. Chan. Linear Time Suffix Array Construction Using D-Critical Substrings, In G. Kucherov and E. Ukkonen, editors, *Combinatorial Pattern Matching*, volume 5577 of *Lecture Notes in Computer Science*, pages 54–67. Springer, 2009.
9. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, UK, 1997. 534 pages.
10. J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003.
11. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2001.
12. D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199. Springer, 2003.
13. P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2003.
14. R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *FOCS*, pages 596–604, 1999.
15. M. G. Main. Detecting leftmost maximal periodicities. *Discret. Appl. Math.*, 25:145–153, 1989.
16. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
17. M. Rodeh, V. R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *J. ACM*, 28(1):16–24, 1981.
18. J. Storer and T. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.
19. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, New York, 1994.
20. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
21. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.