Computing the Longest Previous Factor

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

HAL Id: hal-01806294

https://hal-upec-upem.archives-ouvertes.fr/hal-01806294

Submitted on 1 Jun 2018

Computing the Longest Previous Factor[*][**]

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. The *Longest Previous Factor* array gives, for each position
i in a string y, the length of the longest factor (substring) of y that
occurs both at i and to the left of i in y. The Longest Previous Factor
array is central in many text compression techniques as well as in the
most efficient algorithms for detecting motifs and repetitions occurring
in a text. Computing the Longest Previous Factor array requires usually
the Suffix Array and the Longest Common Prefix array. We give the
first time-space optimal algorithm that computes the Longest Previous
Factor array, given the Suffix Array and the Longest Common Prefix
arrays. We also give the first linear-time algorithm that computes the
permutation that applied to the Longest Common Prefix array produces
the Longest Previous Factor array.

Keywords: longest previous factor, suffix array, longest common prefix,
Ziv-Lempel factorization, text compression, string repetitions.

1 Introduction

The *Longest Previous Factor* array gives, for each position i in a string
y, the length of the longest factor (substring) of y the occurs both at
i and to the left of i in y. For example, for the string $y = \underline{\text{ab}}\text{ba}\underline{\text{ba}}$, the
Longest Previous Factor corresponding to position 3 is 2 because ab is
the longest factor at position 3 that appears before (at position 0). The

Longest Previous Factor array is central in many text compression techniques as well as in the most efficient algorithms for detecting motifs and repetitions occurring in a text. The problem of computing the Longest Previous Factor array may be regarded as an extension of the Ziv-Lempel factorization (LZ77) of a string as defined in [21].

The LZ77 factorization is used in several adaptive compression methods which encode carefully re-occurrences of phrases by pointers or integers (see [19, 20]). It yields more powerful compressors than the factorization in [22] (called LZ78) where a phrase is an extension by a single letter of a previous phrase. But the LZ77 factorization is more difficult to compute. One remarkable application of the Longest Previous Factor array is that the LZ77 factorization can be easily computed from it (see [4]). This implies a linear-time solution for LZ77 factorization on integer alphabets. Previous solutions, using a Suffix Tree [18] or a Suffix Automaton [2] of the string, not only run in time $O(n \log |A|)$ (n is the length of the string and $|A|$ is the size of the alphabet) but also the data structures they use are more space-expensive than the Suffix Array which is used for computing the Longest Previous Factor array.

A slight variant of string parsing, whose relation with the LZ77 factorization is analyzed 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 factorization can usually be skipped because it was done already on one of its previous occurrences. A typical application of this idea resides in algorithms to compute repetitions in strings (see [2, 16, 15]). For example, the algorithm in [15] reports all maximal repetitions (called runs) occurring in a string of length n in $O(n \log |A|)$ time. It runs in linear time if a Suffix Array is used instead of a Suffix Tree [4]. Indeed, this seems to be the only technique that leads to linear-time algorithms, independently of the alphabet size, for this type of question.

The 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 [17], the structure can be built in linear-time by different methods [11, 13, 14, 9] for sorting the suffixes of the string, possibly adding the method of [12] to compute the Longest Common Prefix array. The result holds if the string is drawn from an integer alphabet, that is, if the alphabet of the string 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 in [4], however an array with the same definition appears already in [8], as part of the concept of a Quasi Suffix Array (their π array is the Longest Previous

Factor). In [8], the authors presented a direct computation running in $O(n \log n)$ average time. A naive computation of the Longest Previous Factor array, either on the string 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.

The first linear-time algorithm for computing the Longest Previous Factor array appears in [4] where the application to computing the LZ77 factorization (in linear time) is given as well. An improved version, running on-line on the Suffix Array of the string and requiring only $O(\sqrt{n})$ extra memory space, is shown in [6].

In this article we intensively use the Suffix Array of the string 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.

We improve on the previous results and show that the computation of the Longest Previous Factor array of a string 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. An algorithm similar to the one of [4] is described in Section 3. 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 Longest Previous Factor array. Finally, the Longest Previous Factor array can be obtained as a permutation of the Longest Common Prefix array and we give in Section 6 a linear-time algorithm for that. Alternatively, this an be viewed as a sorting network transforming the Longest Common Prefix array into the Longest Previous Factor array. We conclude by discussing a couple of important research directions.

2   Basic definitions

We consider a string y of length n over an alphabet A. The ith letter of y is denoted by $y[i]$ and $y[i \mathinner{.\,.} j] = y[i]y[i+1] \cdots y[j]$. If, for a string x and two positions i and j in y we have $x = y[i \mathinner{.\,.} j]$, then we say that x occurs at position i in y. We index y's letters from 0 to $n-1$, that is, $y = y[0 \mathinner{.\,.} n-1]$. The suffix of y that starts at position i is $\mathsf{suf}_i(y) = y[i \mathinner{.\,.} n-1]$. The length of y is denoted by $|y| = n$.

In the *LZ77 factorization*, a string y is decomposed into factors, called phrases, $u_0, u_1, \ldots, u_k$, such that

(i) $y = u_0 u_1 \cdots u_k$,

(ii) $u_0 = y[0]$ and,

(iii) for $i \geq 1$, u_i is either a new letter or, otherwise, the longest prefix of $u_i u_{i+1} \cdots u_k$ occurring both at position $|u_0 u_1 \cdots u_{i-1}|$ and before it (that is, to its left) in y; see Example 1.

The *Longest Previous Factor* array is defined by $\mathsf{LPF}[0] = 0$ and, for $1 \leq i \leq n-1$, by

$$\mathsf{LPF}[i] = \max\{k \mid y[i \mathinner{.\,.} i+k-1] = y[j \mathinner{.\,.} j+k-1], \text{ for some } 0 \leq j < i\} \ .$$

Given the LPF array, the LZ77 factorization is easily computed by the following algorithm (see [4]):

$\mathrm{LZ77}(y, \mathsf{LPF})$

```
1   u_0 ← y[0]
2   k ← 0
3   while ((ℓ ← |u_0 u_1 ··· u_k|) < n) do
4         u_{k+1} ← y[ℓ .. ℓ + max{1, LPF[ℓ]} − 1]
5         k ← k + 1
6   return u_0, u_1, ..., u_k
```

Example 1. For the string $y = \mathtt{abaababababbabbb}$, the LZ77 factorization is

$$y = \mathtt{a.b.a.aba.bab.babb.b} \ .$$

The LPF array corresponding to y is given in the third row of Table 1; the other two arrays, prev and next, will be defined in Section 3.

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
$\mathsf{LPF}[i]$	0	0	1	3	2	4	3	2	1	4	3	2	2	1
$\mathsf{prev}[i]$	−1	−1	1	2	3	4	5	1	7	8	9	10	9	12
$\mathsf{next}[i]$	1	14	7	7	7	7	7	14	14	14	12	12	14	14

Table 1. The LPF, prev, and next arrays for the string $y = \mathtt{abaababababbabbb}$.

The Suffix Array of the string y is a data structure used for indexing its content. It comprises two arrays that we denote SA and LCP (from Longest Common Prefix) and that are defined as follows.

The array SA stores the list of positions of y associated with the sorted list of its suffixes in increasing lexicographic order. That is, the array is such that

$$\mathsf{suf}_{\mathsf{SA}[0]}(y) < \mathsf{suf}_{\mathsf{SA}[1]}(y) < \cdots < \mathsf{suf}_{\mathsf{SA}[n-1]}(y) \ .$$

Thus, SA is indexed by the ranks of the suffixes in their sorted list.

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

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

We shall use sometimes the array ISA, the inverse of SA, that provides the rank of a position.

Example 2. For our running example $y = \mathtt{abaabababbabbb}$, the SA and LCP arrays are shown in Table 2.

rank	SA[r]	suf$_{\mathsf{SA}[r]}(y)$	LCP[r]	LPF[SA[r]]
0	2	aababababbabbb	0	1
1	0	abaabababbabbb	1	0
2	3	ababbabbbb	3	3
3	5	ababbabbb	4	4
4	7	abbabbb	2	2
5	10	abbb	3	3
6	13	b	0	1
7	1	baabababbabbb	1	0
8	4	babababbb	2	2
9	6	babbabbb	3	3
10	9	babbb	4	4
11	12	bb	1	2
12	8	bbabbb	2	1
13	11	bbb	2	2

Table 2. The SA and LCP arrays for the string $y = \mathtt{abaabababbabbb}$. The last column gives the *lpf* array indexed by ranks of suffixes instead of positions.

It is natural to define the LPF array indexed on *positions* in the string and the SA and LCP arrays indexes on *ranks* rather than positions. We show in the last column of Table 2 the LPF array indexed also by rank, that is, $\mathsf{LPF}[\mathsf{SA}[\cdot]]$. This duality, of indexing by position or by rank, will appear a number of times throughout the paper.

The computation of the Suffix Array can be done in time $O(n \log n)$ in the comparison model [17] (see [3, 7, 10]). If the string is over an alphabet of integers in the range $[0, n^c]$ for some constant c, then the Suffix Array can be built in time $O(n)$ [11, 13, 14, 9] (see also [3]). An elegant algorithm for building the LCP array in linear time is given in [12].

2.1   Graphic representation

The Suffix Array of the string y has a nice graphic representation that helps understand the algorithms computing the LPF array.

Figure 1 shows the Suffix Array representation for the string $y = $ abaababababbabbb. The abscissa axis refers to ranks of suffixes and the ordinate axis refers to their positions. The sorted list of suffixes is plotted by their positions, and consecutive ranks are linked by an edge whose label is the associated LCP value.
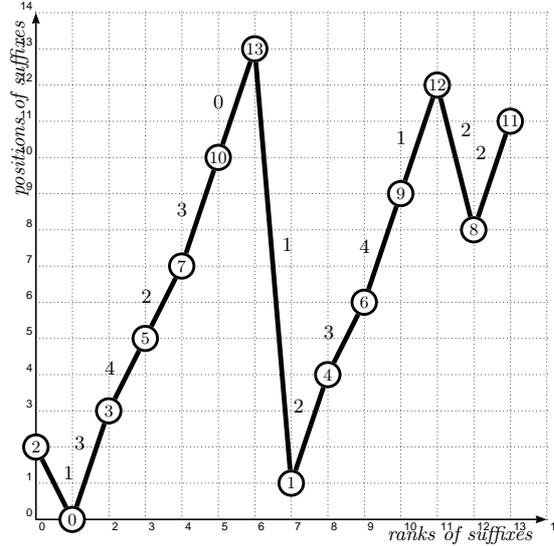
Fig. 1. Graph of the permutation of positions of abaababababbabbb (Example 2) in the lexicographic order of the corresponding suffixes. Labels of edges are Longest Common Prefix lengths between consecutive suffixes.

3   Computing LPF using SA and LCP

The LCP array satisfies the following simple property, a consequence of the lexicographic ordering of suffixes in the SA: the length of the lcp value

between two positions at ranks r and t, $r < t$, that is, $|\mathsf{lcp}(\mathsf{SA}[r], \mathsf{SA}[t])|$ is the minimal value in $\mathsf{LCP}[r+1\mathbin{\ldotp\ldotp}t]$.

For a rank r, let us define $\mathsf{prev}[r]$ as the largest rank $s < r$, for which $\mathsf{SA}[s] < \mathsf{SA}[r]$, if it exists, and as -1 otherwise. Let us also define the dual notion, $\mathsf{next}[r]$, as the smallest rank $t > r$ for which $\mathsf{SA}[t] < \mathsf{SA}[r]$, if it exists, and as n otherwise. The two arrays prev and next for our running example are shown in Table 1.

The above property of the LCP array implies that to compute $\mathsf{LPF}[\mathsf{SA}[r]]$ there is no need to look at ranks smaller than $\mathsf{prev}[r]$ or greater than $\mathsf{next}[r]$, that is,

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

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

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

Also, the minimum of the two values is the length of the lcp between positions at ranks $r-1$ and $r+1$, if defined, that is, $|\mathsf{LCP}(\mathsf{SA}[r-1], \mathsf{SA}[r+1])|$. This gives the idea underlying the algorithm LPF-SIMPLE, which is similar to the one of [4].

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

```
 1   LCP-store ← LCP
 2   LCP[n] ← 0
 3   ISA ← inverse SA
 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 ← ISA[i]
 9        LPF[i] ← max{LCP[r], LCP[next[r]]}
10        LCP[next[r]] ← min{LCP[r], LCP[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
```

The algorithm computes the two arrays prev and next at the same time as computing the LPF array. Steps 9 and 10 implement the earlier discussion and it is enough to explain why the prev and next arrays are correctly computed. The initialization in steps 4-6 gives the correct values for the peaks only. However, in the **for** cycle at step 7, the peaks are considered in decreasing order of their height (that is, position) and hence the updates in steps 11-14 will provide the correct values.

For instance, the algorithm starts with the highest peak in the graph. The LPF value can already be computed for this position. In Figure 1, the highest peak has position 13 and rank 6, that is, $\mathsf{SA}[6] = 13$. Neighboring positions are lower and therefore $\mathsf{prev}[6] = 5$ and $\mathsf{next}[6] = 7$ are correct. Using the above formula for LPF, we have $\mathsf{LPF}[13] = \max\{0, 1\} = 1$. Once this value is set, we remove the node from the graph by adding an edge between its neighbors, 10 and 1, labeled by the minimum of the two LCP values, $\min\{0, 1\} = 0$. The values $\mathsf{next}[5] = 7$ and $\mathsf{prev}[7] = 5$ are also set, the former being already the correct value (due to the peak position 10 at rank 5).

The algorithm LPF-SIMPLE runs in linear time but requires the extra arrays prev, next and ISA in addition to its input and output. Since the LCP array is modified during the algorithm, it can be stored (step 1) in case it is needed later.

Figure 2 illustrates several steps of the algorithm when the input is our running example abaabababbabbb.

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

3.1   An alternative algorithm

The array next can alternatively be pre-computed by the following procedure NEXT whose linear-time behavior is an interesting exercise left to the reader. Only the code in black is needed for the computation of next.

When all code (black and gray) is used, the procedure computes also the array $\mathsf{LPF}_>$ (defined in [4]) that is similar with LPF but accounts for larger ranks only. It is defined, for $r = 0, \ldots, n - 1$, by

$$\mathsf{LPF}_>[r] = |\mathsf{lcp}(\mathsf{SA}[r], \mathsf{SA}[\mathsf{next}[r]])|.$$
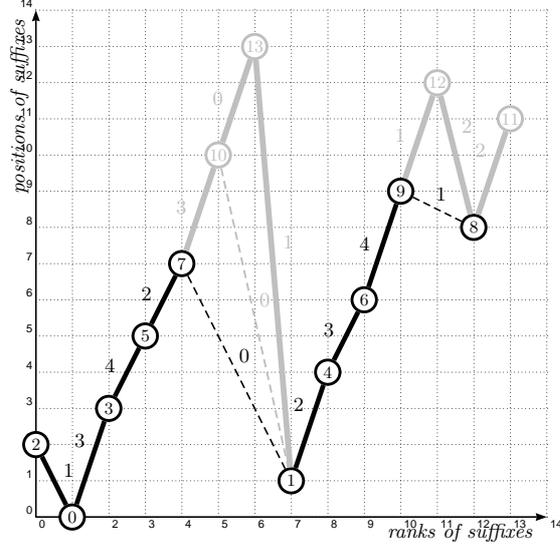
Fig. 2. Illustration of the run of the algorithm LPF-SIMPLE on the string `abaabababbabbbb` (see Figure 1) after processing positions 13, 12, 11, 10. Gray positions and gray edges are no longer considered. The next step is to process position 9 at rank 10. According to the weights of edges pending from 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 $\mathsf{LPF}[8]$.

$\mathrm{NEXT}(\mathsf{SA}, \mathsf{LCP}, n)$

```
 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 SA[t] > SA[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 array is done symmetrically. Also, the array $\mathsf{LPF}_<$ is symmetrically defined and computed. When both arrays $\mathsf{LPF}_<$ and $\mathsf{LPF}_>$ are available, the computation of the LPF array is done using the following equality:

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

4   On-line computation

Techniques of the previous sections to compute the LPF array 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 string of length n. This algorithm is similar with the one of [6] with some differences in view of our time-space optimal algorithm in the next section.

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 $\mathsf{SA}[r]$ 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 $\mathsf{LPF}[\mathsf{SA}[r]] = \mathsf{LCP}[r]$. 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 $\mathsf{SA}[n] = -1$ and $\mathsf{LCP}[n] = 0$. But we prefer the present design that is compatible with the algorithm of the next section.

LPF-ON-LINE$(\mathsf{SA}, \mathsf{LCP}, n)$

```
 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 SA[r] < SA[t] then
 7                  LPF[SA[t]] ← max{t-lcp, r-lcp}
 8                  r-lcp ← min{t-lcp, r-lcp}
 9                  POP(S)
10             elseif (SA[r] > SA[t]) and (r-lcp ≤ t-lcp) then
11                  LPF[SA[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[SA[t]] ← t-lcp
18        POP(S)
19   return LPF
```

The extra memory space used by the algorithm LPF-ON-LINE to compute the LPF array of a string 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 string of length n. It is proved in [6] that this quantity is $O(\sqrt{n})$.

For most values of n there are plenty of strings 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 $\{a, b\}$ (with $a < b$) giving the maximal size stack. This word is $aabab^2 \cdots ab^{k-1}$ and the maximal stack size is k.

The maximal stack sizes for strings of lengths 4 to 22 are given in Table 3.

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

Table 3. Maximum stack size for various string lengths.

Proposition 2. *The algorithm* LPF-ON-LINE *computes the LPF array of a string of length n from its Suffix Array in linear time and $O(\sqrt{n})$ space. It requires less than $2\sqrt{2n} + O(1)$ integer cells in addition to its input and output.*

5   Time-space optimal algorithm

In this section we show that the computation of the LPF array of a string can be implemented with only constant memory space in addition to the SA, LCP, and LPF arrays. 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 array for a sufficiently large part of the string. The rest of the computation for the remaining positions is done in a more time-expensive manner but for a small part of the string. This preserves the linear running time of the whole computation.

LPF-OPTIMAL($\mathsf{SA}, \mathsf{LCP}, n$)

    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 array
    4   LPF-ON-LINE($\mathsf{SA}, \mathsf{LCP}, K$)
    5   LPF-NAIVE($\mathsf{SA}, \mathsf{LCP}, K, n$)
    6   LPF-ANCHORED($\mathsf{SA}, \mathsf{LCP}, K, n$)
    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 array. Access to the array is done via the SA array. Doing so, the stack is treated like a continuous space $\mathsf{LPF}[\mathsf{SA}[K \mathinner{..} n - 1]]$. 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 3. Table 4 shows the content of the LPF array at two stages of the run of the procedure LPF-ON-LINE on Example 2: (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) $\mathsf{LPF}[\mathsf{SA}[r]]$	1		3	4					5	3	4	2	1	0
(ii) $\mathsf{LPF}[\mathsf{SA}[r]]$	1	0	3	4	2	3	1				8	2	7	0

Table 4. The content of the LPF table during the procedure LPF-ON-LINE.

The row (i) shows that LPF values of positions 2, 3, 5 at respective ranks 0, 2, 3 have already been computed. The part $\mathsf{LPF}[\mathsf{SA}[8 \mathinner{..} 13]]$ of the array stores the content of the stack: $((1, 0), (4, 2), (5, 3))$.

The 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.

5.1   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.

LPF-NAIVE$(\mathsf{SA}, \mathsf{LCP}, K, n)$

```
 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 SA[s] > SA[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 SA[t] > SA[r] do
13             t ← t + 1
14             right ← min{right, LCP[t]}
15        LPF[SA[r]] ← max{left, right}
16   return LPF
```

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 $K = n - 2\sqrt{2n}$.

5.2   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 $\mathsf{LPF}[\mathsf{SA}[r]]$ might not be achieved. The same phenomenon happens for a rank in the second part, whose associated prev rank is smaller than K.

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

LPF-ANCHORED$(\mathsf{SA}, \mathsf{LCP}, K, n)$

```
 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 SA[s] < SA[t] then
 7             LPF[SA[t]] ← max{LPF[SA[t]], ℓ}
 8             t ← t + 1
 9             ℓ ← min{ℓ, LCP[t]}
10        else LPF[SA[s]] ← max{LPF[SA[s]], ℓ}
11             ℓ ← min{ℓ, LCP[s]}
12             s ← s − 1
13   return LPF
```

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 array of a string of length n on an integer alphabet can be built from the read-write Suffix Array and Longest Common Prefix array in time $O(n)$ (independently of the alphabet size) with a constant amount of extra memory space.*

The algorithm LPF-OPTIMAL uses the Suffix Array of the input string in a read-only manner but does not use the LPF array in a write-only manner. If this last condition is to be satisfied, the question remains of whether there exists a linear-time LPF array 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.

Proposition 3. *The Longest Previous Factor array of a string of length n on an integer alphabet can be built from its read-only Suffix Array and Longest Common Prefix 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   Computing LPF by permuting LCP

It has been noticed in [4] that the content of the LPF array is the same as that of the LCP array up to some permutation. Then the following questions arise:

– Can this permutation be computed efficiently?
– Does it depend on the string, its Suffix Array or maybe just its Longest Common Prefix array?

It turns out that, for fixed SA and LCP arrays, it does not depend on the actual string. 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 array. This observation leads to the algorithm LPF-by-permuting-LCP, producing LPF by permuting the elements of LCP.

In the algorithm LPF-by-permuting-LCP below we assume that the array next has been pre-computed by the procedure NEXT from Section 3.1 (in which the gray instructions, related to $LPF_>$, are removed, as well as the parameter LCP).

The array next has been defined as the next rank with a smaller suffix. It helps defining another array, nextPos, that behaves similarly but positions instead of ranks. More precisely, it computes, for each position i, the next closest position nextPos[i] in the Suffix Array that is smaller than i, that is,

$$\mathsf{nextPos}[\mathsf{SA}[r]] = \mathsf{SA}[\min\{t \mid t > r \text{ and } \mathsf{SA}[t] < \mathsf{SA}[r]\}] = \mathsf{SA}[\mathsf{next}[r]] \ .$$

Equivalently, if we set the position $i = \mathsf{SA}[r]$, then

$$\mathsf{nextPos}[i] = \mathsf{SA}[\mathsf{next}[\mathsf{ISA}[i]]] \ .$$

Initially, the LPF array is a copy of the LCP array permuted according to the SA array. The algorithm permutes the elements of the array to get the correct LPF array. This involves a modification of the "peaks" idea used previously. If we look again at Figure 1, the highest peak (position

13) has the two adjacent edges labeled 0 and 1. The higher of the two values becomes LPF[13] whereas the smaller one is the length of the lcp between suffixes at ranks 5 and 7. Another way to do this is by permuting the labels 0 and 1 such that each goes to the right place, that is, 1 is on the edge to the left of 13 (and will remain there as the correct LPF value) and 0 goes on the edge to the left of 1, as the new LCP value.
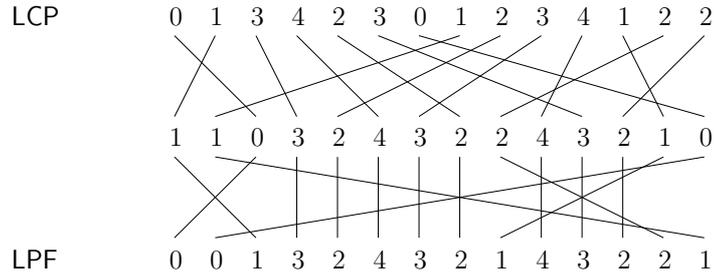
LPF-by-permuting-LCP(SA, LCP, n)

```
 1   SA[n] ← −1
 2   π ← SA
 3   for r ← 0 to n − 1 do
 4        LPF[SA[r]] ← LCP[r]
 5        nextPos[SA[r]] ← SA[next[r]]
 6   for i ← n − 1 downto 0 do
 7        if (nextPos[i] ≥ 0) and (LPF[i] < LPF[nextPos[i]]) then
 8             Exchange(LPF[i], LPF[nextPos[i]])
 9             π ← π ∘ (LPF[i], LPF[nextPos[i]])
10   return LPF, π
```

Note that the algorithm LPF-by-permuting-LCP computes also explicitly the permutation π to transform the LCP array into the LPF array, that is, the permutation that satisfies $\mathsf{LPF}[\pi[i]] = \mathsf{LCP}[i]$.

The final permutation is obtained by composing the SA (which is a permutation) with the transpositions produced in step 8. For our example, this process is shown in Figure 3. The actual final permutation is

$$\begin{pmatrix} 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13 \\ 0\ 2\ 3\ 5\ 7\ 10\ 1\ 13\ 4\ 6\ 9\ \ 8\ 12\ 11 \end{pmatrix}.$$

Fig. 3. The permutation of LCP that gives LPF, as computed by the algorithm LPF-by-permuting-LCP.

The algorithm LPF-by-permuting-LCP uses only one integer array in addition to its input and output since the `next` array it uses is substituted for the `nextPos` array. We proved the following result:

Theorem 2. *The algorithm* LPF-by-permuting-LCP *computes the* LPF *array of a string of length n from its Suffix Array in linear time and space. It requires $n + O(1)$ integer cells in addition to its input and output.*

Note that the elements of LPF are exchanged (lines 7–9) only if they are not in increasing order. Which elements are compared, depends on values in `nextPos`, and this in turn depends only on the Suffix Array. So, for a given Suffix Array, one can construct a sorting network implementing lines 6–9 of the algorithm LPF-by-permuting-LCP. Hence, the following proposition holds:

Proposition 4. *For a given Suffix Array of a string, there exists a sorting network processing a sequence of n numbers in such a way that it transforms the* LCP *array into the* LPF *array. Moreover, the shape of the sorting network depends only on the Suffix Array, but not on its* LCP *array nor on the actual string.*

7   Conclusion

A number of interesting problems remain open. First, it is easy to compute the LZ77 factorization using the LPF array. However, no way of doing the opposite computation is known. The LZ77 factorization contains less information and therefore, it may not be of great use in constructing the LPF array.

Second, our algorithm LPF-optimal is time-space optimal assuming the SA and LCP arrays are used for computing the LPF array. It is of great interest to find fast algorithms that compute the LPF array without such information or, more generally, without using any string indexes.

Acknowledgements

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, C. Iliopoulos, M. Kubica, W. Rytter, and T. Waleń, LPF computation revisited, in: J. Fiala, J. Kratochvíl, and M. Miller, eds., *Proc. of IWOCA 2009*, Lecture Notes in Comput. Sci. **5874**, Springer, Heidelberg, 2009, 158–169.

6. 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.

7. M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific Publishing, Hong-Kong, 2002. 310 pages.

8. 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.

9. 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.

10. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, UK, 1997. 534 pages.

11. 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.

12. 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.

13. 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.

14. 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.

15. R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *FOCS*, pages 596–604, 1999.

16. M. G. Main. Detecting leftmost maximal periodicities. *Discret. Appl. Math.*, 25:145–153, 1989.

17. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

18. M. Rodeh, V. R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *J. ACM*, 28(1):16–24, 1981.

19. J. Storer and T. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

20. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, New York, 1994.
21. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
22. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.