Computing Longest Previous Factor in linear time and applications
Maxime Crochemore, Lucian Ilie

To cite this version:

HAL Id: hal-00619691
https://hal-upec-upem.archives-ouvertes.fr/hal-00619691
Submitted on 13 Feb 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Computing Longest Previous Factor in linear time and applications *

MAXIME CROCHEMORE ⋆ ⋆ ⋆ and LUCIAN ILIE ⋆ ⋆ ⋆ ⋆ ,†

1 Department of Computer Science, King’s College London, London WC2R 2LS, UK, e-mail: maxime.crochemore@kcl.ac.uk
Institut Gaspard-Monge, Université Paris-Est, F-77454 Marne-la-Vallée Cedex 2
2 Department of Computer Science, University of Western Ontario, N6A 5B7, London, Ontario, CANADA, e-mail: ilie@csd.uwo.ca

October 24, 2007

Abstract. We give two optimal linear-time algorithms for computing the Longest Previous Factor (LPF) array corresponding to a string w. For any position i in w, LPF[i] gives the length of the longest factor of w starting at position i that occurs previously in w. Several properties and applications of LPF are investigated. They include computing the Lempel-Ziv factorization of a string and detecting all repetitions (runs) in a string in linear time independently of the integer alphabet size.

Key words: algorithms design, strings, suffix array, longest common prefix, longest previous factor, Lempel–Ziv factorization, repetitions, runs

MSC: 68W05, 68W40, 68R15

1 Introduction

Given a string w, we introduce the Longest Previous Factor (LPF) array defined as follows. For any position i in w, LPF[i] gives the length of the longest factor of w starting at position i that occurs previously in w. Formally, if w[i] denotes the ith letter of w and w[i..j] is the factor w[i]w[i + 1]...w[j], then

LPF[i] = max( {ℓ | w[i..i + ℓ - 1] is a factor of w[0..i + ℓ - 2]} ∪ {0} ) .

We give two linear-time (optimal) algorithms for computing LPF using suffix arrays. The first uses no additional information whereas the second uses the longest common prefix array which is often part of the suffix array data structure. Previously such algorithms involved computing the suffix trees, which are more complex and take a lot of space. Also, a logarithmic factor of the size of the alphabet often appears in the complexity. Our algorithms use suffix arrays, are much simpler, and their complexity is alphabet independent.

⋆ This work has been presented at the AutoMathA’07 conference, see [5].
⋆⋆ Research supported in part by CNRS.
⋆ ⋆ ⋆ Research supported in part by NSERC.
† Corresponding author
One important application is computing the Lempel–Ziv factorization [14]. Recently [1] gave a suffix-array-based algorithm for computing Lempel–Ziv factorization. However, their algorithm is essentially a simulation of the suffix tree using the suffix array. The description in [1] is very brief but it seems that their approach can be used to achieve similar goals with ours, nevertheless in a significantly more complicated way.

Simultaneously and independently of our work, [2] gave an algorithm that is similar with our second one. Our first algorithm is more general and our approach for the second gives a clearer explanation as well as more insight into the structure of LPF.

### 2 Suffix arrays

We recall in this section briefly the notions of suffix array and longest common prefix. Consider a string \( w = w[0 \ldots n-1] \) of length \( n \) over an alphabet \( A \) that is an integer interval of size no more than \( n^c \), for some constant \( c \). The suffix of \( w \) starting at position \( i \) is denoted by \( \text{suf}_i = w[i..n-1] \), for \( 0 \leq i \leq n-1 \). The suffix array of \( w \), [16], denoted \( \text{SA} \), gives the suffixes of \( w \) sorted ascendingly in lexicographical order, that is, \( \text{suf}_{\text{SA}[0]} < \text{suf}_{\text{SA}[1]} < \ldots < \text{suf}_{\text{SA}[n-1]} \). The suffix array of the string \( abbaabbaaabab \) is shown in the second column of Fig. 1.

Often the suffix array is used in combination with another array, the Longest Common Prefix (LCP) which gives the length of the longest common prefix between consecutive suffixes of \( \text{SA} \), that is, \( \text{LCP}[i] \) is the length of the longest common prefix of \( \text{suf}_{\text{SA}[i]} \) and \( \text{suf}_{\text{SA}[i-1]} \); see the third column of Fig. 1 for an example.

<table>
<thead>
<tr>
<th>( i )</th>
<th>( \text{SA}[i] )</th>
<th>( \text{LCP}[i] )</th>
<th>( \text{suf}_{\text{SA}[i]} )</th>
<th>prev_( \text{SA}[i] )</th>
<th>prev_( \text{LCP}[i] )</th>
<th>prev_( \text{SA}[i] )</th>
<th>LPF_( \text{SA}[i] )</th>
<th>PrevOcc_( \text{SA}[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>0</td>
<td>aaabab</td>
<td>-1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>2</td>
<td>aabab</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>aabbaaabab</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>1</td>
<td>ab</td>
<td>3</td>
<td>10</td>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>2</td>
<td>abab</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>2</td>
<td>abbaabbaaabab</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>3</td>
<td>abbaaabab</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>0</td>
<td>b</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>1</td>
<td>baaabab</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>3</td>
<td>baabab</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>2</td>
<td>bab</td>
<td>2</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>1</td>
<td>bbabaab</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>4</td>
<td>bbaabbaaabab</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>2</td>
<td>bbbaabab</td>
<td>1</td>
<td>-1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 1.** The arrays \( \text{SA} \), \( \text{LCP} \), and LPF for the string \( abbaabbaaabab \).
3 A direct algorithm

We give first a direct algorithm for computing LPF from the suffix array. We compute also, for each \(i\), a position \(\text{PrevOcc}[i] < i\) where the longest previous factor at \(i\) occurs.\(^1\) (If LPF\([i]\) = 0, then \(\text{PrevOcc}[i] = -1\).) Both arrays for our example are shown in the last two columns in Fig. 1.

The idea of the algorithm is as follows. For any position \(i\), the longest factor starting at \(i\) that occurs also to the left of \(i\) in \(w\) is the longest common prefix between the suffix \(\text{suf}_i\) and the suffixes starting to the left of \(i\) in \(w\); that is, \(\text{suf}_j, 0 \leq j \leq i - 1\). However, given \(\text{SA}\), we need only consider those which are closest to \(\text{suf}_i\) in \(\text{SA}\). We shall therefore compute, for each \(i\), the closest positions in \(\text{SA}\) that are smaller than \(i\); in most cases there will be two such positions, one before and one after \(i\) in \(\text{SA}\). Denote them by \(\text{prev}_{\text{<}}[i]\) and \(\text{prev}_{\text{>}}[i]\), respectively. If one of them does not exists, then we assign the value \(-1\); see columns 5 and 6 in Fig. 1. Rephrasing the above, LPF\([i]\) is obtained as the length of the longest common prefix between \(\text{suf}_i\) and either \(\text{suf}_{\text{prev}_{\text{<}}[i]}\) or \(\text{suf}_{\text{prev}_{\text{>}}[i]}\), whichever is longer.

After \(\text{prev}_{\text{<}}\) and \(\text{prev}_{\text{>}}\) are found, LPF is computed for all values of \(i\) in increasing order, using the property that LPF\([i]\) \(\geq\) LPF\([i - 1]\) \(- 1\). Thus, we already know that \(w[i \ldots i + \text{LPF}[i - 1] - 2]\) occurred to the left of \(i\) and need only try to extend it. A problem appears because we do not know whether we should compare \(\text{suf}_i\) to \(\text{suf}_{\text{prev}_{\text{<}}[i]}\) or \(\text{suf}_{\text{prev}_{\text{>}}[i]}\). We shall therefore compute two arrays, LPF\([0 \ldots n - 1]\) and LPF\([0 \ldots n - 1]\); they have the same meaning as LPF except that they consider only positions corresponding to suffixes lexicographically smaller, resp. larger, than \(\text{suf}_i\). Formally, we have

\[
\text{LPF}_{\text{<}}[i] = \max\{\{j \mid \exists k < i, w[i \ldots i + j - 1] = w[k \ldots k + j - 1] \text{ and } \text{suf}_k < \text{suf}_i\} \cup \{0\}\}
\]

and

\[
\text{LPF}_{\text{>}}[i] = \max\{\{j \mid \exists k < i, w[i \ldots i + j - 1] = w[k \ldots k + j - 1] \text{ and } \text{suf}_k > \text{suf}_i\}\}
\]

\(\text{LPF}_{\text{<}}\) and \(\text{LPF}_{\text{>}}\) are defined identically except for the last condition which becomes \(\text{suf}_k > \text{suf}_i\). Consider one of them, say \(\text{LPF}_{\text{<}}\). We still have that \(\text{LPF}_{\text{<}}[i] \geq \text{LPF}_{\text{<}}[i - 1] - 1\). This is because, if \(w[i - 1] = w[k - 1]\), then the order between \(\text{suf}_i\) and \(\text{suf}_k\) is the same as between \(\text{suf}_{i - 1}\) and \(\text{suf}_{k - 1}\). That means, we already know that \(\text{suf}_i\) and \(\text{suf}_{\text{prev}_{\text{<}}[i]}\) have a common prefix of length \(\text{LPF}_{\text{<}}[i - 1] - 1\) and we check only the following letters. This explains why our algorithm runs in \(O(n)\) time. Finally, LPF\([i]\) is the maximum between LPF\([i]\) and LPF\([i]\). The algorithm is given in Fig. 2.

Computing the arrays \(\text{prev}_{\text{<}}\) and \(\text{prev}_{\text{>}}\) is a matter of manipulating data structures. We can construct a doubly linked list with the elements of \(\text{SA}\) and two extra elements at the beginning and at the end with value \(-1\), for the case \(\text{prev}_{\text{<}}[i]\) or \(\text{prev}_{\text{>}}[i]\) do not exist. The values are computed in decreasing order of \(i\), from \(n\) to 0, and each \(i\) is removed once the values for \(i\) have been computed. When \(i\) is considered, it is the largest left in the list and therefore the one before and the one after in the list will give the values of \(\text{prev}_{\text{<}}[i]\) and \(\text{prev}_{\text{>}}[i]\). In fact pointers can be avoided, and arrays used instead, because after the doubly linked list is created only deletions are performed. The details are omitted.

\(^1\) Note that a suffix-tree-based algorithm would compute the leftmost such position in the string whereas our algorithm might produce a different one. For instance, in our example, \(\text{PrevOcc}[12] = 10\) but the left most occurrence of \(\text{ab}\) starts at 0.
of adjacent edges are labelled 0 and 1, corresponding to the longest common prefixes that are highest in the graph (“peaks”) are considered first. For vertex 13, the two solid edges only.

\[ j \text{ and ordinate maximum of the two gives } LPF \]

The vertices are arranged in the left-to-right order corresponding to their order in arrays can be computed in time and space \( O(n) \) using the fact that the suffix array of a string of length \( n \) requires less memory space. The idea is similar to the one above. Assuming we know the longest common prefixes between \( suf_i \) and either \( suf_{prev<}[i] \) or \( suf_{prev>}[i] \), it is enough to take the maximum of the two.

\[ \text{LPF} \]

\[ \text{LPF}[0] \leftarrow \text{LPF}_{<}[0] \leftarrow \text{LPF}_{>}[0] \leftarrow 0 \]

\[ \text{for } i \text{ from 1 to } n-1 \text{ do} \]

\[ j \leftarrow \max(\text{LPF}_{<}[i-1] - 1, 0); k \leftarrow \max(\text{LPF}_{>}[i-1] - 1, 0) \]

\[ \text{else while } (u[j + j] = u[\text{prev}_<[i] + j]) \text{ do } j \leftarrow j + 1 \]

\[ \text{if } (\text{prev}_>[i] = -1) \text{ then } \text{LPF}_<[i] \leftarrow 0 \]

\[ \text{else while } (u[i + k] = u[\text{prev}_>[i] + k]) \text{ do } k \leftarrow k + 1 \]

\[ \text{LPF}_{>}[i] \leftarrow k \]

\[ \text{LPF}[i] \leftarrow \max(\text{LPF}_{<}[i], \text{LPF}_{>}[i]) \]

\[ \text{else if } (\text{LPF}[i] = 0) \text{ then } \text{PrevOcc}[i] \leftarrow -1 \]

\[ \text{else if } (\text{LPF}_{<}[i] > \text{LPF}_{>}[i]) \text{ then } \text{PrevOcc}[i] \leftarrow \text{prev}_>[i] \]

\[ \text{else } \text{PrevOcc}[i] \leftarrow \text{prev}_<[i] \]

\[ \text{return LPF and PrevOcc} \]

FIG. 2. Algorithm for computing LPF directly from the SA via arrays \( prev_< \) and \( prev_> \).

It should be clear from the above discussion that the arrays \( prev_< \) and \( prev_> \) can be computed in time \( O(n) \). The algorithm \( \text{COMPUTE}\text{-}LPF \) uses \( O(n) \) space. Using the fact that the suffix array of a string of length \( n \) over an integer alphabet can be computed in \( O(n) \) time by any of the algorithms in [7, 9, 11, 12], we obtain:

**Theorem 1.** Given a string of length \( n \) over an integer alphabet, the LPF and PrevOcc arrays can be computed in time and space \( O(n) \).

4 An algorithm using LCP

Our second algorithm for computing LPF uses the LCP array. Its advantage over the previous algorithm is that it processes the suffix array in one pass and requires less memory space. The idea is similar to the one above. Assuming we know the longest common prefixes between \( suf_i \) and either \( suf_{prev<}[i] \) or \( suf_{prev>}[i] \), it is enough to take the maximum of the two.

Using this idea, we give a space-efficient algorithm for computing LPF. For a better understanding, it is useful to arrange the SA and LCP arrays in a graph. The vertices are labelled by the SA values and the edges by the LCP values. The vertices are arranged in the left-to-right order corresponding to their order in SA and are placed at a height corresponding to their starting position in the string. In other words, if \( SA[i] = j \), then the vertex labelled \( j \) is plotted with abscissa \( i \) and ordinate \( j \). An example if shown by the graph in Fig. 3(i) consisting of the solid edges only.

Now consider the vertices in decreasing order of their labels, that is, vertices that are highest in the graph (“peaks”) are considered first. For vertex 13, the two adjacent edges are labelled 0 and 1, corresponding to the longest common prefixes of \( suf_{13} \) with \( suf_{prev<}[13] = suf_4 \) and \( suf_{prev>}[13] = suf_7 \), respectively. Therefore, the maximum of the two gives \( LPF[13] = 1 \). On the other hand, the minimum of the
two gives the longest common prefix of \( \text{suf}_4 \) and \( \text{suf}_7 \); we remove the vertex 13 and add a (dotted) edge between 4 and 7, labelled 0. Notice that \( \text{prev}_<[7] = 4 \) and \( \text{prev}_>[7] = 2 \), so this property is an invariant of the graph. Next we consider the vertex 12 and so on. Fig. 3(ii) shows the graph after having considered the vertices 13, 12, and 11.

It is clear that we need not consider the vertices in this order. For instance, we can compute \( \text{LPF}[6] \) right away. Precisely, any vertex which is a “peak” in our graph can have its \( \text{LPF} \) value computed. In the algorithm in Fig. 4 we consider the vertices in the order they appear in the \( \text{SA} \) (that is, left to right in the graph) and use a stack to store unprocessed vertices. The vertices in the stack will be in increasing order from bottom to top. Whenever a vertex with a smaller label than the one on top of the stack is encountered, the top of the stack is processed and popped. For instance, vertex 4 has to wait for 13 and 7 to be processed. In order to process all vertices uniformly, we assume a last (virtual) vertex with label \( n \) and height \( -1 \). The stack contains pairs of the form \((x.\text{len}, x.\text{pos})\), where \( x.\text{pos} \) is a position (in \( \text{SA} \)) and \( x.\text{len} \) stores the longest common prefix between \( \text{suf}_{x.\text{pos}} \) and the suffix corresponding to the node right below \( x \) in the stack (or 0 if none). For instance, when the vertex 7 is considered, 13 is processed and removed from the stack. The top of the stack becomes 4 and 7 is then pushed on top of it: \( \text{top}(\mathcal{S}).\text{len} \leftarrow 0 \) and \( \text{top}(\mathcal{S}).\text{pos} \leftarrow 7 \).

The correctness of the algorithm follows from the above discussion. It runs in \( O(n) \) time because each element of \( \text{SA} \) is pushed only once on to the stack. Also, \cite{10} gives a very simple linear time algorithm to compute the LCP array.
ComputeLPF using LCP(w, SA, LCP)

1. SA[n] ← −1; LCP[n] ← 0
2. push((0, SA[0]), S)
3. for i from 1 to n do
4. lcp ← LCP[i]
5. while (S̸= ∅) and (SA[i] < TOP(∅).pos)
7. v ← TOP(∅)
8. POP(∅)
9. if (LPF[v.pos] = 0) then PrevOcc[v.pos] ← −1
10. else if (v.len > lcp) then PrevOcc[v.pos] ← TOP(∅).pos
11. else PrevOcc[v.pos] ← SA[i]
12. if (i < n) then push((lcp, SA[i]), S)
13. return LPF

Fig. 4. Algorithm for computing LPF using LCP.

The spaced used by the stack is at most n pairs of integers, which is reached for the string a^{n−1}b. However, the expected size of the stack is much less.

There is an interesting consequence of the above discussion, namely that the array LPF is a permutation of LCP. This is shown by analyzing the graph in Fig. 3(i). The labels of the solid edges form the LCP array. When removing a “peak” from the graph, the maximum of the two labels of the adjacent edges becomes an LPF value whereas the minimum becomes the label of the newly formed (dotted) edge. Each value will eventually become an LPF value which proves the statement.

Proposition 1. LPF is a permutation of LCP.

Remark 1. Note that [1] suggests a bottom-up computation of the LPF array on the lcp-interval tree (isomorphic with the suffix tree) and therefore leads to a similar result with ours. However, our approach is much simpler.

5 Application 1: computing the Lempel–Ziv factorization

The Lempel–Ziv factorization of w [14] is the decomposition w = u_0u_1⋯u_k, where each u_i (except possibly u_k) is the longest prefix of u_iu_{i+1}⋯u_k that has another occurrence to the left in w or a single letter in case this prefix is empty.

For our example the Lempel–Ziv factorization is a.b.a.abb.baa.ab.ab.

The Lempel–Ziv factorization is a basic and powerful technique for text compression [17]. It has many variants used in gzip or PKzip software, and more generally in dictionary compression methods.

The Lempel–Ziv factorization is easily computed from LPF. The algorithm is shown in Fig. 5. For the example text abbaabbbaaabab in Fig. 1, the algorithm outputs LZ = [0, 1, 2, 3, 4, 7, 10, 12].
Theorem 2. The Lempel–Ziv factorization of a string of length \( n \) over an integer alphabet can be computed in \( O(n) \) time.

The experimental results of [2] may be explained by the fact that suffix-tree-based algorithms computing the Lempel–Ziv factorization usually have complexity \( O(\log(|A|)n) \), where \(|A|\) is the cardinality of the alphabet.

Also, note that suffix trees allow online computation of the Lempel–Ziv factorization. However, this comes with the extra \( \log(|A|) \) factor. It remains open whether true linear-time online computation is possible.

6 Application 2: computing runs in linear time

Repetitions are a fundamental topic in stringology and appear in many applications such as text algorithms, data compression, or analysis of biological sequences. The simplest repetition is a square \( ww \), where \( w \) is any string. A general repetition has the form \( w^e \), for any rational exponent \( e \geq 2 \) such that \( e|w| \) is an integer; e.g., \( (aabab)^7 = aababaabaa \). Particularly important turned out to be maximal repetitions [15] or \( \text{runs} \). A run is an occurrence of a repetition that cannot be extended. As an example, the string \( aababaabba \) contains the runs \( aa \) at positions 0 and 5, \( ababa \), and \( bb \). Runs allow the encoding of all repetitions in linear space [13].

An element of the Lempel–Ziv factorization carries information already processed by any online algorithms computing repetitions. Therefore it is not surprising that the Lempel–Ziv factorization plays a central part in the algorithm of [13] as well as in all efficient computations of repetitions in strings. Their running time is then \( O(n \log(|A|)) \). Using a suffix array and the algorithms described in previous sections leads to linear-time computations on integer alphabets. This applies to: testing square freeness of a string [3], computing all leftmost maximal periodicities [15], computing all runs [13], computing of all local periods of a string [6], and computing all primitively-rooted squares occurring in a string [8]. In particular, we get:

Theorem 3. The runs of a string of length \( n \) over an integer alphabet can be computed in \( O(n) \) time.
7 Acknowledgements

We warmly thank Bill Smyth and Simon Puglisi for interesting discussions related to the present subject during the “String Masters” meeting held in Hamilton in July 2007.

Also, we would like to thank the anonymous referees for comments that helped improving the presentation.

References