Elsevier Editorial System(tm) for Journal of Discrete Algorithms
Manuscript Draft

Manuscript Number: JDA-10-50R1

Title: Efficient algorithms for three variants of the LPF table

Article Type: Stringology, Bioinformatics and Algorith

Keywords: longest previous reverse factor; longest previous non-overlapping reverse factor; longest previous non-overlapping factor; longest previous factor; palindrome; runs; suffix array; text compression

Corresponding Author: Mr. Marcin Kubica, Ph.D.

Corresponding Author's Institution: University of Warsaw

First Author: Maxime Crochemore, Professor

Order of Authors: Maxime Crochemore, Professor; Costas S Iliopoulos, Professor; Marcin Kubica, Ph.D.; Wojciech Rytter, Professor; Tomasz Walen, Ph.D.

Abstract: The concept of a longest previous factor (LPF) is inherent to Ziv-Lempel factorization of strings in text compression, as well as in statistics of repetitions and symmetries.
It is expressed in the form of a table --- LPF[i] is the maximum length of a factor starting at position i, that also appears earlier in the given text.
We show how to compute efficiently three new tables storing different variants of previous factors (past segments) of a string.
The longest previous non-overlapping factor, for a given position i, is the longest factor starting at i which has an exact copy occurring entirely before, while the longest previous non-overlapping reverse factor for a given position i is the longest factor starting at i, such that its reverse copy occurs entirely before.
In both problems the previous copies of the factors are required to occur within the prefix ending at position i-1.
The longest previous (possibly overlapping) reverse factor is the longest factor starting at i, such that its reverse copy starts before i.

These problems have not been explicitly considered before, but they have several applications and they are natural extensions of the longest previous factor problem, which has been extensively studied.

Moreover, the newly introduced tables store additional information on the structure of the string, helpful to improve, for example, gapped palindrome detection and text compression using reverse factors.

Response to Reviewers: We agree with all the reviewers' comments (with one small exception, see detailed response).
Many thanks for a careful reviews.

Detailed Response to Reviewers

All the remarks have been taken into account and applied.
The only exception is the suggestion of Reviewer#1,
to change symbols LPnF, LPrF, LPnrF to LPFn, LPFr, LPFnr.
Since it is a matter of taste, we would rather leave them in the original form.

Efficient algorithms for three variants of the LPF table☆

Maxime Crochemore[a,c], Costas S. Iliopoulos[a,e], Marcin Kubica[b],
Wojciech Rytter[b,d], Tomasz Waleń[b]

[a] *King's College London, London WC2R 2LS, UK*
[b] *Institute of Informatics, University of Warsaw, Warsaw, Poland*
[c] *Université Paris-Est, France*
[d] *Dept. of Math. and Informatics, Copernicus University, Toruń, Poland*
[e] *Digital Ecosystems & Business Intelligence Institute, Curtin University of Technology, Perth WA 6845, Australia*

Abstract

The concept of a longest previous factor (LPF) is inherent to Ziv-Lempel factorization of strings in text compression, as well as in statistics of repetitions and symmetries. It is expressed in the form of a table — $\mathsf{LPF}[i]$ is the maximum length of a factor starting at position i, that also appears earlier in the given text. We show how to compute efficiently three new tables storing different variants of previous factors (past segments) of a string. The longest previous non-overlapping factor, for a given position i, is the longest factor starting at i which has an exact copy occurring entirely before, while the longest previous non-overlapping reverse factor for a given position i is the longest factor starting at i, such that its reverse copy occurs entirely before. In both problems the previous copies of the factors are required to occur within the prefix ending at position $i-1$. The longest previous (possibly overlapping) reverse factor is the longest factor starting at i, such that its reverse copy starts before i.

These problems have not been explicitly considered before, but they have several applications and they are natural extensions of the longest previous factor problem, which has been extensively studied. Moreover, the newly introduced tables store additional information on the structure of the string, helpful to improve, for example, gapped palindrome detection and text compression using reverse factors.

Keywords: longest previous reverse factor, longest previous non-overlapping reverse factor, longest previous non-overlapping factor, longest previous factor, palindrome, runs, suffix array, text compression

---

☆Research supported in part by the Royal Society, UK.

Email addresses: `maxime.crochemore@kcl.ac.uk` (Maxime Crochemore), `csi@dcs.kcl.ac.uk` (Costas S. Iliopoulos), `kubica@mimuw.edu.pl` (Marcin Kubica), `rytter@mimuw.edu.pl` (Wojciech Rytter), `walen@mimuw.edu.pl` (Tomasz Waleń)

1. Introduction

In this paper we describe new algorithmic results which exploit the power of suffix arrays [6, 16, 18, 19, 24, 25]. Three new useful tables related to the table of the longest previous factors (the `LPF` table, see [7, 8, 9, 12]) are computed in linear time, additionally using the power of data structures for Range Minimum Queries (RMQ, in short) [2, 10]. The `LPF` table, for a given position i, contains the maximum length of a factor starting at position i, whose exact copy starts before position i. We assume throughout the paper, that we have an integer alphabet, sortable in linear time. This assumption implies we can compute the suffix array in linear time, with constant coefficient independent of the alphabet size.

The first problem is to compute efficiently, for a given string y, the table of the longest previous non-overlapping reverse factors (the `LPnrF` table), that stores at each index i the maximal length of factors (substrings), that both start at position i in y and occur in reverse entirely before position i. This concept is close to the table of the longest previous factors (the `LPF` table), for which the previous occurrence is not reverse. The latter table extends the Ziv-Lempel factorization of a text [28] intensively used for text compression (known as LZ77 method, see [1]). It turns out, that both problems are related to each other, and together they can be applied to compress sequences containing repeated, possibly reversed fragments.

Another problem is to compute the table of longest previous reverse factors (the `LPrF` table). In the sense of the definition, this problem resembles the problem of computing the `LPF` table very much. However, if we consider positions of the corresponding characters, it turns out they are not as related, as the problems of computing the `LPnrF` and `LPF` tables. Also, it does not have such natural applications in compression. However, it can be useful when extracting symmetries, e.g. in detection of gapped and ordinary palindromes.

The third problem is to compute the table of longest previous non-overlapping factors (the `LPnF` table). In the sense of the definition, the `LPnF` table differs very slightly from the `LPF` table (because the latter allows overlaps between the considered occurrences while the former does not), but the `LPF` table is a permutation of the longest common prefix array (`LCP` array) [17], while `LPnF` usually is not, and the algorithms for `LPnF` differ much from those for `LPF`. However, the `LPnF` table can be useful when computing repetitions.

The `LPnrF` table generalises a factorization of strings used by Kolpakov and Kucherov [21] to extract certain types of palindromes in molecular sequences. These palindromes are of the form uvw where v is a short string and w is the complemented reverse of u (complement consists in exchanging letters `A` and `U`, as well as `C` and `G`, the Watson-Crick pairs of nucleotides). These palindromes play an important role in RNA secondary structure prediction because they signal potential hair-pin loops in RNA folding (see [3]).

An additional motivation for considering the `LPnrF` table is text compression. Indeed, it may be used, in connection with the `LPF` table, to improve the Ziv-Lempel factorization (the basis of several popular compression software) by considering occurrences of reverse factors as well as usual factors. The feature has already been implemented in [14] but without LPnrF and LPF tables, and our algorithm provides a more efficient technique to compress DNA sequences under the scheme.

We design algorithms computing the LPnrF, LPrF and LPnF tables. They are computed, using two pre-computed read-only arrays (SUF and LCP) composing the suffix array, in linear time on any integer alphabet.

As far as we know, the LPnrF table of a string has never been considered before. Our source of inspiration was the notion of LPF table and the optimal methods for computing it in [4, 8]. It is shown there that the LPF table can be derived from the Suffix Array of the input string both in linear time and with only a constant amount of additional space.

The second problem, computation of the LPnF table of non-overlapping previous factors, emerged from a version of Ziv-Lempel factorization. An alternative algorithm solving this problem was given in [27]. The factorization it leads to plays an important role in string algorithms because the work done on an element of the factorization is skipped since already done on one of its previous occurrences. A typical application of this idea is to compute repetitions in strings (see [5, 20, 22]). It happens that the algorithm for the LPnF table computation is a simple adaptation of the algorithm for LPnrF. It may be surprising, because in one case we deal with exact copies of factors and in the second with reverse copies.

The problem of computing the LPrF table has been included for the sake of completeness — this way we cover all possible combinations of previous factors: reversed or not, and overlapping or not. The LPrF table, when compared to LPnrF, has no known applications, yet.

In this article we show that the computation of the LPnrF, LPrF and LPnF tables of a string can be done in linear time from its Suffix Array. So, we get the same running time as the algorithm described in [21] for the corresponding factorization although our algorithm produces more information stored in the table and ready to be used.

In addition to the Suffix Array of the input string, the algorithm makes use of a data structure for constant time RMQs, and the Manacher's algorithm to recognize palindromes [23]. The question of whether there exists a direct linear-time algorithm, for integer alphabets, not using all these sophisticated techniques (that is RMQ, Suffix Array or suffix tree) exists remains open. Its solution would open an exciting path of novel techniques for text processing.

2. Preliminaries

Let us consider a string $y = y[0\,..\,n-1]$ of length n. By y^R we denote the reverse of y, that is $y^R = y[n-1]y[n-2]\dots y[0]$. The LPF table (see [7, 8, 9, 12]), and the three other tables we consider, LPnrF, LPrF and LPnF, are

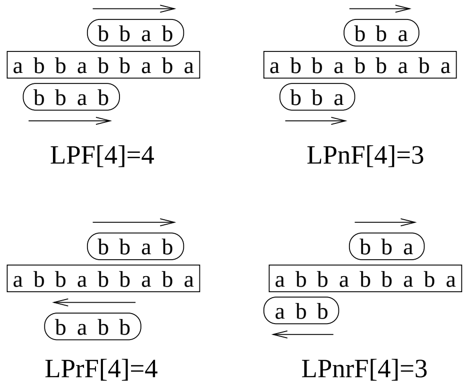


Figure 1: Illustration of $\mathsf{LPF}[4]$, $\mathsf{LPnF}[4]$, LPrF and $\mathsf{LPnrF}[4]$ for the string `abbabbaba`.

defined (for $0 \leq i < n$) as follows (see Figure 2):

$$\begin{aligned}
\mathsf{LPF}[i] &= \max\{j : \exists_{0 \leq k < i} : y[k \mathinner{..} k+j-1] = y[i \mathinner{..} i+j-1]\} \\
\mathsf{LPnrF}[i] &= \max\{j : \exists_{0 \leq k \leq i-j} : y[k \mathinner{..} k+j-1]^R = y[i \mathinner{..} i+j-1]\} \\
\mathsf{LPrF}[i] &= \max\{j : \exists_{0 \leq k < i} : y[k \mathinner{..} k+j-1]^R = y[i \mathinner{..} i+j-1]\} \\
\mathsf{LPnF}[i] &= \max\{j : \exists_{0 \leq k \leq i-j} : y[k \mathinner{..} k+j-1] = y[i \mathinner{..} i+j-1]\}
\end{aligned}$$

It can be noted that in the definition of the LPF and LPrF tables the occurrences of $y[k \mathinner{..} k+j-1]$ and $y[i \mathinner{..} i+j-1]$ may overlap, while it is not the case with the other tables above. For example, the string $y =$ `abbabbaba` has the following tables:

position i	0	1	2	3	4	5	6	7	8
$y[i]$	`a`	`b`	`b`	`a`	`b`	`b`	`a`	`b`	`a`
$\mathsf{LPF}[i]$	0	0	1	5	4	3	2	2	1
$\mathsf{LPnrF}[i]$	0	0	2	1	3	3	2	2	1
$\mathsf{LPrF}[i]$	0	6	5	5	4	3	2	2	1
$\mathsf{LPnF}[i]$	0	0	1	3	3	3	2	2	1

We start the computation of these arrays with computation of the Suffix Array [6, 16, 18, 24, 25] for the text y. It is a data structure used for indexing the text. It comprises three tables denoted by SUF, RANK and LCP, and is defined as follows. The SUF array stores the list of positions in y sorted according to the increasing lexicographic order of suffixes starting at these positions. That is, the SUF table is such that:

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

Thus, indices of SUF are ranks of the respective suffixes in the increasing lexicographic order. The RANK array is the inverse of the SUF array, that is:

$$\mathsf{SUF}[\mathsf{RANK}[i]] = i \quad \text{and} \quad \mathsf{RANK}[\mathsf{SUF}[r]] = r$$

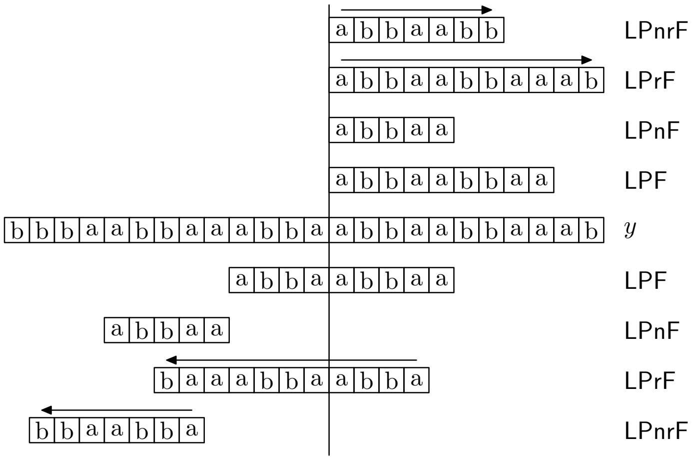


Figure 2: Comparison of LPF, LPnF, LPrF and LPnrF tables; it shows differences between LPF and LPnF, and between LPrF and LPnrF.

The LCP [17] array is indexed by the ranks of the suffixes, and stores the lengths of the longest common prefixes of consecutive suffixes in SUF. Let us denote by $\mathsf{lcp}(i, j)$ the length of the longest common prefix of $y[i \mathinner{..} n-1]$ and $y[j \mathinner{..} n-1]$ (for $0 \le i, j < n$). Then, we set $\mathsf{LCP}[0] = 0$ and, for $0 < r < n$, we have:

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

For example, the Suffix Array of the text $y =$ `abbabbaba` is:

i	$s[i]$	RANK[i]
0	a	3
1	b	8
2	b	6
3	a	2
4	b	7
5	b	5
6	a	1
7	b	4
8	a	0

rank r	SUF[r]	LCP[r]	suf(SUF[r])
0	8	0	a
1	6	1	aba
2	3	2	abbaba
3	0	5	abbabbaba
4	7	0	ba
5	5	2	baba
6	2	3	babbaba
7	4	1	bbaba
8	1	4	bbabbaba

The Suffix Array can be built in time $O(n)$ (see [6] [6, 16, 18, 19, 25]).

In the algorithms presented in this paper we use the Minimum (Maximum) Range Query data-structure (RMQ, in short). Let us assume, that we are given an array $A[0 \mathinner{..} n-1]$ of numbers. This array is preprocessed to answer the following form of queries: given an interval $[\ell \mathinner{..} r]$ (for $0 \le \ell \le r < n$), find the minimum (maximum) value $A[k]$ for $\ell \le k \le r$.

The RMQ problem has received much attention in the literature. Bender and Farach-Colton [2] presented an algorithm with $O(n)$ preprocessing complexity and $O(1)$ query time, using $O(n \log n)$-bits of space. The same result was previously achieved in [13], albeit with a more complex data structure. Sadakane [26], and recently Fischer and Heun [11] presented a succinct data structures, which achieve the same time complexity using only $O(n)$ bits of space.

3. The technique of alternating search

At the heart of our algorithms for computing the LPnrF and LPnF tables, there is a special search in a given interval of the table SUF for a position k (the best candidate) which gives the next value of the table (LPnrF or LPnF). This search is composed of two simple alternating functions, so we call it here the *alternating search*.

Assume we have an integer function $\mathit{Val}(k)$ which is non-increasing for $k \geq i$. Our goal is to find any position k in the given range $[i\,..\,j]$, which maximises $\mathit{Val}(k)$ and satisfies some given property $\mathit{Candidate}(k)$ (we call values satisfying $\mathit{Candidate}(k)$ simply *candidates*). We assume, that $\mathit{Val}(k)$ and $\mathit{Candidate}(k)$ can be computed in $O(1)$ time. Let us also assume, that the following two functions are computable in $O(1)$ time:

- $\mathit{FirstMin}(i, j)$ — returns the first position k in $[i\,..\,j]$ with the minimum value of $\mathit{Val}(k)$,
- $\mathit{NextCand}(i, j)$ — returns any candidate k from $[i\,..\,j)$ if there are any, otherwise it returns some arbitrary value not satisfying $\mathit{Candidate}(k)$.

Without loss of generality, we can assume that j is a candidate — otherwise, we can narrow our search to the range $[i\,..\,\mathit{NextCand}(i, j)]$. Please, observe, that:

$$\mathit{Val}(k) > \mathit{Val}(j) \text{ for } i \leq k < \mathit{FirstMin}(i, j)$$

Hence, if $\mathit{FirstMin}(i, j) > i$ and $\mathit{NextCand}(i, \mathit{FirstMin}(i, j))$ is a candidate, then we can narrow our search to the interval $[i\,..\,\mathit{NextCand}(i, \mathit{FirstMin}(i, j))]$. Otherwise, j is the position we are looking for.

Consequently, we can iterate *FirstMin* and $\mathit{NextCand}(i, k)$ queries, increasing with each step the value of $\mathit{Val}(j)$ by at least one unit. This observation is crucial for the complexity analysis of our algorithms.

```
Algorithm 1: Alternating-Search($i, j$)
    $k$ := initial candidate in the range $[i\,..\,j]$, satisfying Candidate;
    while Candidate($k$) do
        $j := k$ ;
        $k := \mathit{NextCand}(i, \mathit{FirstMin}(i, j))$;
    return $j$;
```

Lemma 1. *Let $k = $ Alternating-Search(i, j). The execution time of Alternating-Search(i, j) is $O(\mathit{Val}(k) - \mathit{Val}(j) + 1)$.*

Proof. Observe, that each iteration of the while loop, except the last one, increases $\mathit{Val}(k)$ by at least one. The last iteration assigns the value of k to j, which is then returned as a result. Hence, the number of iterations performed by the while loop is not greater than $\mathit{Val}(k) - \mathit{Val}(j) + 1$. Each iteration requires $O(1)$ time, what concludes the proof. ☐

In the following sections, we apply the Alternating-Search algorithm to compute the LPnrF and LPnF tables. Our strategy is to design the algorithm in which, in each invocation of the Alternating-Search algorithm, the initial value of $\mathit{Val}(k)$ is smaller than the previously computed element of the LPnrF/LPnF table by at most 1. In other words, we start with a reasonably good candidate, and the cost of a single invocation of the Alternating-Search algorithm can be charged to the difference between two consecutive values. The linear time follows from a simple amortisation argument. The details are in the following sections.

4. Computation of the LPnrF table

This section presents how to calculate the LPnrF table, for a given string y of size n, in $O(n)$ time. First, let us create a string $x = y\#y^R$ of size $N = 2n+1$ (where $\#$ is a character not appearing in y). For the sake of simplicity, we set that $y[n] = \#$ and $y[-1] = x[-1] = x[N]$ are defined and smaller than any character in $x[0\,..\,N-1]$.

Let SUF be the suffix array related to x, RANK be the inverse of SUF (that is $\mathsf{SUF}[\mathsf{RANK}[i]] = i$, for $0 \le i < N$), and LCP be the longest common prefix table related to x. Let i and j, $0 \le i, j < N$ be two different positions in x, and let $i' = \mathsf{RANK}[i]$ and $j' = \mathsf{RANK}[j]$. Observe, that:

$$\begin{aligned} \mathsf{lcp}(i,j) &= \min\{\mathsf{LCP}[\min(i',j')+1\,..\,\max(i',j')]\} \\ \mathsf{LPnrF}[i] &= \max\{\mathsf{lcp}(i,j) : j \ge N-i\} \end{aligned}$$

Let us define two auxiliary arrays: $\mathsf{LPnrF}_{>}$ and $\mathsf{LPnrF}_{<}$, which are variants of the LPnrF array restricted to the case, where the first mismatch character in the reversed suffix is greater (smaller) than the corresponding character in the suffix. More formally, using x:

$$\begin{aligned} \mathsf{LPnrF}_{>}[i] &= \max\left\{ \begin{aligned} j : \exists_{j-1\le k<i} : y[k-j+1\,..\,k]^R = y[i\,..\,i+j-1] \\ \text{and}\ \ y[k-j] > y[i+j] \end{aligned} \right\} \\ \mathsf{LPnrF}_{<}[i] &= \max\left\{ \begin{aligned} j : \exists_{j-1\le k<i} : y[k-j+1\,..\,k]^R = y[i\,..\,i+j-1] \\ \text{and}\ \ y[k-j] < y[i+j] \end{aligned} \right\} \end{aligned}$$

or equivalently, using x:

$$\begin{aligned} \mathsf{LPnrF}_{>}[i] &= \max\left\{ \begin{aligned} j : \exists_{N-i\le k\le N-j} : x[k\,..\,k+j-1] = x[i\,..\,i+j-1] \\ \text{and}\ \ x[k+j] > x[i+j] \end{aligned} \right\} \\ \mathsf{LPnrF}_{<}[i] &= \max\left\{ \begin{aligned} j : \exists_{N-i\le k\le N-j} : x[k\,..\,k+j-1] = x[i\,..\,i+j-1] \\ \text{and}\ \ x[k+j] < x[i+j] \end{aligned} \right\} \end{aligned}$$

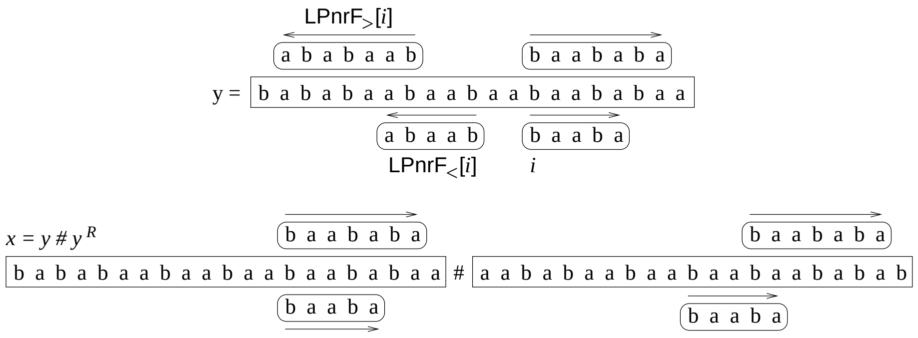


Figure 3: Examples of $\mathsf{LPnrF}_{>}$ and $\mathsf{LPnrF}_{<}$ values, in the text y and in $x = y\#y^R$.

The following lemma, formulates an important property of the LPnrF array, which is extensively used in the presented algorithm.

Lemma 2. *For $0 < i < n$, we have $\mathit{LPnrF}_{>}[i] \geq \mathit{LPnrF}_{>}[i-1] - 1$ and $\mathit{LPnrF}_{<}[i] \geq \mathit{LPnrF}_{<}[i-1] - 1$.*

Proof. Without loss of generality, we can limit the proof to the first property. Let $\mathsf{LPnrF}_{>}[i-1] = j$. So, there exists some $k < i-1$, such that:

$$y[k-j+1\mathinner{.\,.}k]^R = y[i-1\mathinner{.\,.}i+j-2] \quad \text{and} \quad y[k-j] > y[i+j-1]$$

Omitting the first character, we obtain:

$$y[k-j+1\mathinner{.\,.}k-1]^R = y[i\mathinner{.\,.}i+j-2] \quad \text{and} \quad y[k-j] > y[i+j-1]$$

and hence $\mathsf{LPnrF}_{>}[i] \geq j-1 = \mathsf{LPnrF}_{>}[i-1] - 1$. □

In the algorithm computing the LPnrF array, we use two data structures for RMQ queries. They are used to answer, in constant time, two types of queries:

- $\mathsf{FirstMinPos}(p, q, \mathsf{LCP})$ returns the first (from the left) position in the range $[p\mathinner{.\,.}q]$ with minimum value of LCP,
- $\mathsf{MaxValue}(p, q, \mathsf{SUF})$ returns the maximal value from $\mathsf{SUF}[p\mathinner{.\,.}q]$.

Lemma 3. *The $\mathit{MaxValue}(p, q, \mathit{SUF})$ and $\mathit{FirstMinPos}(p, q, \mathit{LCP})$ queries require $O(n)$ preprocessing time, and then can be answered in constant time.*

Proof. Clearly, the SUF and LCP arrays can be constructed in $O(n)$ time (see [6]). The $\mathsf{MaxValue}(p, q, \mathsf{SUF})$ and $\mathsf{FirstMinPos}(p, q, \mathsf{LCP})$ queries are applied to the sequence of $O(n)$ length. Hence they require $O(n)$ preprocessing time and then can be answered using Range Minimum Queries in constant time (see [10]). Note that, in the FirstMinPos query we need slightly modified range queries, that return the first (from the left) minimal value, but the algorithms solving RMQ problem can be modified to accommodate this fact. □

Algorithm 2: Compute-LPrF$_>$

```
initialization: $\mathsf{LPnrF}_>[0] := 0$; $k_0 := 0$ ;
for $i = 1$ to $n - 1$ do
    $r_i := \mathsf{RANK}(i)$ { start Alternating Search } ;
    $k := \mathit{InitialCandidate}(k_{i-1}, \mathsf{LPnrF}_>[i-1])$ ;
    while $k \geq N - i$ do
        $k_i := k$ ;
        $r_k := \mathsf{RANK}(k)$ ;
        $r'_k := \mathsf{FirstMinPos}(r_i + 1,\ r_k,\ \mathsf{LCP})$ ;
        $\mathsf{LPnrF}_>[i] := \mathsf{LCP}[r'_k]$ ;
        if $r_i + 1 < r'_k$ then
            $k := \mathsf{MaxValue}(r_i + 1,\ r'_k - 1,\ \mathsf{SUF})$
        else break;
return $\mathsf{LPnrF}_>$;
```

Function `InitialCandidate`(k, l)

```
if $l > 0$ then
    return $k + 1$
else
    return $N$;
```

Algorithm 2 computes the $\mathsf{LPnrF}_>$ array from left to right. In each iteration it also computes the value k_i, which is the position of the substring (in the second half of x), that maximizes $\mathsf{LPnrF}_>[i]$. Namely, if $\mathsf{LPnrF}_>[i] = j$, then:

$$y[i\mathinner{.\,.}i+j-1] = x[k_i\mathinner{.\,.}k_i+j-1] = y[N-k_i-j+1\mathinner{.\,.}N-k_i]^R$$

Lemma 4. *Algorithm 2 works in $O(n)$ time.*

Proof. We prove this lemma using amortized cost analysis. The amortization function equals $\mathsf{LPnrF}_>[i]$. Initially we have $\mathsf{LPnrF}_>[0] = 0$.
Observe, that the body of the for loop is an instance of the Algorithm 1, with:

$$\begin{aligned} \mathit{Val}(k) &= \mathsf{lcp}(i,k) \\ \mathit{Candidate}(k) &\equiv k \geq N - i \\ \mathit{FirstMin}(i,k) &= \mathsf{FirstMinPos}(\mathsf{RANK}[i]+1, \mathsf{RANK}[k], \mathsf{LCP}) \\ \mathit{NextCand}(i,j) &= \mathsf{MaxValue}(\mathsf{RANK}[i]+1, j-1, \mathsf{SUF}) \end{aligned}$$

Hence, by Lemmata 1 and 2, each iteration of the **for** loop takes $O(\mathsf{LPnrF}_>[i] - \mathsf{LPnrF}_>[i-1] + 2)$ time, and the overall time complexity of Algorithm 2 is $O(n + \mathsf{LPnrF}[n-1] - \mathsf{LPnrF}[0]) = O(n)$.
The correctness of the algorithm follows from the fact that (for each i) the body of the while loop is executed at least once (as a consequence of Lemma 2). ☐

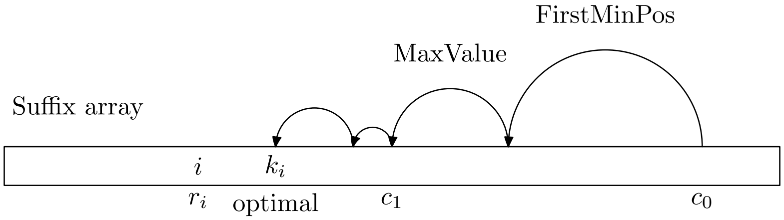


Figure 4: Iterations of the while loop of Algorithm 2.

Theorem 1. *The LPnrF array can be computed in $O(n)$ time. For (polynomially bounded) integer alphabets the complexity does not depend on the size of the alphabet.*

Proof. The table $\mathsf{LPnrF}_<$ can be computed using similar approach in $O(n)$ time. Then, $\mathsf{LPnrF}[i] = \max(\mathsf{LPnrF}_<[i], \mathsf{LPnrF}_>[i])$. □

5. Computation of the LPrF table

This section presents how to calculate the LPrF table, for a given string y of length n, in $O(n)$ time. We will show, how to reduce it to a new problem of the longest previous *overlapping* reverse factor. This new problem is to compute a LPorF table, defined as follows:

$$\mathsf{LPorF}[i] = \max\{j : j = 0 \text{ or } \exists_{i-j<k<i} : y[k \mathinner{..} k+j-1]^R = y[i \mathinner{..} i+j-1]\}$$

Let us consider the longest previous reversed factor of $y[i \mathinner{..} n-1]$ for some $i = 0, \ldots, n-1$. There are two possible cases: either it occurs not overlapping position i, or it overlaps it. In the first case, its length equals $\mathsf{LPnrF}[i]$, and in the latter one it equals $\mathsf{LPorF}[i]$. Hence:

$$\mathsf{LPrF}[i] = \max(\mathsf{LPnrF}[i], \mathsf{LPorF}[i])$$

We have already shown how to compute the LPnrF table in $O(n)$ time. Now, we will show how to compute the LPorF table in the same time complexity.

Let i be a position in y, $0 \le i < n$, and let $j = \mathsf{LPorF}[i] > 0$. Since $\mathsf{LPorF}[i]$ cannot be equal 1, we have $\mathsf{LPorF}[i] \ge 2$. Let us consider an overlapping reversed occurrence of $y[i \mathinner{..} i+j-1]$ and let k be its starting position. We have $i - j < k < i$ and:

$$y[k \mathinner{..} k+j-1]^R = y[i \mathinner{..} i+j-1]$$

Note, that:

$$y[i \mathinner{..} k+j-1] = y[i \mathinner{..} k+j-1]^R$$

and:

$$y[k+j \mathinner{..} i+j-1] = y[k \mathinner{..} i-1]^R$$

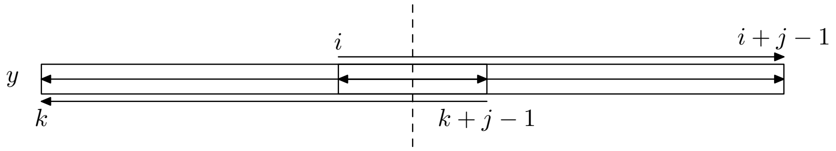


Figure 5: Previous overlapping reversed factor and related palindrome.

Hence:

$$y[k\,..\,i+j-1] = y[k\,..\,i+j-1]^R$$

That is, $y[k\,..\,i{+}j{-}1]$ is a palindrome (see Fig. 5). The center of this palindrome is at $\frac{k+i+j-1}{2}$, where halves denote positions between characters.

The reverse implication is also valid. Let $y[b\,..\,e]$ be a palindrome, where $0 \le b < e < n$. The center of the palindrome is at $\frac{b+e}{2}$. For any such integer i, that $b < i \le \frac{b+e}{2}$, we have: $y[i\,..\,e] = y[b\,..\,b{+}e{-}i]^R$. Hence, $\mathsf{LPorF}[i] \ge e{-}i{+}1$. Moreover, taking into account all such palindromes, we obtain:

$$\mathsf{LPorF}[i] \quad = \quad \max\left\{ e-i+1 : b < i \le \frac{b+e}{2} \;\text{ and }\; y[b\,..\,e] = y[b\,..\,e]^R \right\} \quad (1)$$

Information about all the palindromes in y can be obtained in $O(n)$ time using Manacher's algorithm [23]. The output from this algorithm has a form of a table $D[0\,..\,2(n-1)]$, such that $D[c]$ is the maximum length of a palindrome with a center at position $\frac{c}{2}$ (where halves denote positions between characters). More formally, the maximal palindrome with a center at position $\frac{c}{2}$ is:

$$y\left[\frac{c-D[c]}{2}\,..\,\frac{c+D[c]}{2}\right]$$

Having computed array D, we can reformulate equation 1, as:

$$\begin{aligned} \mathsf{LPorF}[i] \quad &= \quad \max\left\{ \frac{c+D[c]}{2} - i + 1 : \frac{c-D[c]}{2} < i \le \frac{c}{2} \right\} = \\ &= \quad \max\left\{ \frac{c+D[c]}{2} : c - D[c] < 2i \le c \right\} - i + 1 \end{aligned}$$

Array D can be processed from right to left, and each of the above maxima can be computed in a constant amortized time. With each index i, two new elements, $D[2i]$ and $D[2i+1]$, should be considered. On the other hand, all such values $D[c]$ considered in the previous step, for which $c - D[c] = 2i$, can be discarded in further computations. Moreover, we can use the following two observations to further limit the number of values $D[c]$ needed to compute $\mathsf{LPorF}[i]$.

Lemma 5. *Let c_1 and c_2 be two such indices, that $0 \le c_1 < c_2 \le 2(n-1)$ and $c_1 - D[c_1] \ge c_2 - D[c_2]$, then $D[c_1]$ does not influence the computation of the LPorF array.*

Proof. If i is such an index, that $c_1 - D[c_1] < 2i \leq c_1$, then also $c_2 - D[c_2] < 2i \leq c_2$. Moreover, $D[c_2] > D[c_1]$ and hence $\frac{c_2+D[c_2]}{2} > \frac{c_1+D[c_1]}{2}$. □

Lemma 6. *Let c_1 and c_2 be two such indices, that $0 \leq c_1 < c_2 \leq 2(n-1)$ and $c_1 + D[c_1] \geq c_2 + D[c_2]$, then $D[c_2]$ does not influence the values of $\mathsf{LPorF}[i]$, for $i \leq \frac{c_1}{2}$.*

Proof. If i is such an index, that $2i \leq c_1$. Even if $2i > c_2 - D[c_2]$, then $\frac{c_1+D[c_1]}{2} \geq \frac{c_2+D[c_2]}{2}$. □

As an immediate consequence of Lemmata 5 and 6, we obtain the following fact:

Lemma 7. *When computing $\mathsf{LPorF}[o\,..\,i]$, instead of considering all the values $D[2i\,..\,2(n-1)]$, one can limit considerations to $D[c_1]$, $D[c_2]$, …, $D[c_m]$, where $c_1, c_2, \ldots, c_m$ is the maximal sequence satisfying the following properties:*

- $i \leq c_1 < c_2 < \cdots < c_m$,
- $c_1 - D[c_1] < c_2 - D[c_2] < \cdots < c_m - D[c_m] < 2i$,
- $c_1 + D[c_1] < c_2 + D[c_2] < \cdots < c_m + D[c_m]$.

Due to Lemma 7, we can use a two-sided queue to store all relevant indices $c_1, c_2, \ldots, c_m$. Moreover, if the queue is empty, then$\mathsf{LPorF}[i] = 0$, and otherwise:

$$\mathsf{LPorF}[i] = \frac{c_m + D[c_m]}{2} - i + 1$$

Algorithm 4 exploits the above observations, calculating the LPorF array.

Algorithm 4: Compute-LPorF

```
initialization: $q :=$ empty ;
for $i = n-1$ downto 0 do
    Insert($q, 2i+1$) ;
    Insert($q, 2i$) ;
    $\mathsf{LPorF}[i] = GetMax(q)$ ;
return LPorF;
```

Function `Insert` (q, c)

```
if empty($q$) or $c - D[c] < q.first - D[q.first]$ then
    while not empty($q$) and $c + D[c] \geq q.first + D[q.first]$ do
        remove_first($q$);
    insert_first($q, c$);
```

Total number of elements inserted into queue q does not exceed $2n-1$. Since each element can be removed only once, the amortized running time of *Insert* and *GetMax* functions is constant. Hence, the total running time of Algorithm 4 is $O(n)$. As a consequence, we obtain the following theorem:

Function `GetMax`(q, i)

```
while not $empty(q)$ and $q.last - D[q.last] \geq 2i$ do
    $remove\_last(q)$ ;
if $empty(q)$ then
    return $0$
else
    return $(q.last + D[q.last])/2 - i + 1$
```

Theorem 2. *The* LPrF *array can be computed in* $O(n)$ *time.*

6. Longest previous non-overlapping factor

This section presents how to calculate the LPnF table in $O(n)$ time. First, let us investigate the values of the LPnF array. For the sake of simplicity, we set that $y[n]$ is defined and smaller than any character in $y[0\,..\,n-1]$. For each value $j = \mathsf{LPnF}[i]$, let us have a look at the characters following the respective factors of length j. Let $0 \leq k < i$ be such that $y[k\,..\,k+j-1] = y[i\,..\,i+j-1]$. There are two possible reasons, why these factors cannot be extended:

- either the following characters do not match (that is, $y[k+j] \neq y[i+j]$), or
- they match, but if the factors are extended, then they would overlap (that is, $y[k+j] = y[i+j]$ and $k + j = i$).

We divide the LPnF problem into two subproblems, and (for $0 \leq i < n$) define:

$$\begin{aligned}
\mathsf{LPnF}^M[i] &= \max\left\{\begin{array}{r} j : \exists_{k<j} : y[k\,..\,k+j-1] = y[i\,..\,i+j-1], \\ y[k+j] \neq y[i+j] \text{ and } k+j \leq i \end{array}\right\} \\
\mathsf{LPnF}^O[i] &= \max\{j : \exists_{k<j} : y[k\,..\,k+j-1] = y[i\,..\,i+j-1] \text{ and } k+j = i\}
\end{aligned}$$

It is easy to see that $\mathsf{LPnF}[i] = \max\{\mathsf{LPnF}^M[i], \mathsf{LPnF}^O[i]\}$. The $\mathsf{LPnF}^O[i]$ is, in fact, the maximum radius of a square that has its center between positions $i-1$ and i. Such array can be easily computed in linear time from runs, using approach proposed in [20].

We have to show how to compute the LPnF^M array. Following the same scheme we have used for the LPnrF problem, we reduce this problem to the computation of two tables, namely $\mathsf{LPnF}^M_>$ and $\mathsf{LPnF}^M_<$, defined as LPnF^M with the restriction that the mismatch character in the previous factor $y[k+j]$ is greater (smaller) than $y[i+j]$. More formally:

$$\begin{aligned}
\mathsf{LPnF}^M_>[i] &= \max\left\{\begin{array}{r} j : \exists_{0\leq k\leq i-j} : y[k\,..\,k+j-1] = y[i\,..\,i+j-1] \\ \text{and } \; y[k+j] > y[i+j] \end{array}\right\} \\
\mathsf{LPnF}^M_<[i] &= \max\left\{\begin{array}{r} j : \exists_{0\leq k\leq i-j} : y[k\,..\,k+j-1] = y[i\,..\,i+j-1] \\ \text{and } \; y[k+j] < y[i+j] \end{array}\right\}
\end{aligned}$$

Clearly, $\mathsf{LPnF}^M[i] = \max(\mathsf{LPnF}^M_>[i], \mathsf{LPnF}^M_<[i])$. Without loss of generality, we can limit our considerations to computation of $\mathsf{LPnF}^M_>$. Just like LPnrF, the $\mathsf{LPnF}^M_>$ array has the property, that for any i, $1 < i \leq n$, $\mathsf{LPnF}^M_>[i] \geq \mathsf{LPnF}^M_>[i-1] - 1$.

Lemma 8. *For $0 < i < n$, we have $\mathsf{LPnF}^M_>[i] \geq \mathsf{LPnF}^M_>[i-1] - 1$.*

Proof. Let $\mathsf{LPnF}^M_>[i-1] = j$. So, there exists some $0 \leq k \leq i - j - 1$, such that:

$$y[k \mathinner{.\,.} k+j-1] = y[i-1 \mathinner{.\,.} i+j-2] \quad \text{and} \quad y[k+j] > y[i+j-1]$$

If we omit the first characters, then we obtain:

$$y[k+1 \mathinner{.\,.} k+j-1] = y[i \mathinner{.\,.} i+j-2] \quad \text{and} \quad y[k+j] > y[i+j-1]$$

and hence $\mathsf{LPnF}^M_>[i] \geq j - 1 = \mathsf{LPnF}^M_>[i-1] - 1$. □

Algorithm 7: Compute-$\mathsf{LPnF}_>$

```
initialization: $\mathsf{LPnF}^M_>[0] := 0$; $k_0 = 0$ ;
for $i = 1$ to $n - 1$ do
    $r_i := \mathsf{RANK}[i]$ ;
    $(k, l) = \mathit{InitialCandidate}(k_{i-1}, \mathsf{LPnF}^M_>[i-1])$ ;
    while $l = 0$ or $k + l \leq i$ do
        $k_i = k$;
        $r_k := \mathsf{RANK}[k]$ ;
        $r'_k := \mathsf{FirstMinPos}(r_i + 1, r_k, \mathsf{LCP})$ ;
        $\mathsf{LPnF}^M_>[i] := l$ ;
        if $[r_i + 1 \leq r'_k - 1] \neq \emptyset$ then
            $k := \mathsf{MinValue}(r_i + 1,\ r'_k - 1,\ \mathsf{SUF})$ ;
            $l := \mathsf{lcp}(r_i, \mathsf{RANK}[k])$ ;
        else break;
return $\mathsf{LPnF}^M_>$;
```

Function `InitialCandidate`(k, l)

```
if $l > 0$ then
    return $(k + 1, l - 1)$
else
    return $(n, 0)$;
```

In the algorithm computing the $\mathsf{LPnF}^M_>$ array, we use two data structures for RMQ queries. They are applied to answer, in constant time, two types of queries:

- $\mathsf{FirstMinPos}(p, q, \mathsf{LCP})$ returns the first (from the left) position in the range $[p \mathinner{.\,.} q]$ with minimum value of LCP,

- $\mathsf{MinValue}(p, q, \mathsf{SUF})$ returns the minimal value from $\mathsf{SUF}[p\,..\,q]$.

Lemma 9. *Algorithm 7 works in $O(n)$ time.*

Proof. We prove this lemma using amortized cost analysis. The amortization function equals $\mathsf{LPnF}^M_>[i]$. Initially we have $\mathsf{LPnF}^M_>[0] = 0$. Please observe, that the body of the for loop is an instance of the Algorithm 1, with:

$$\begin{aligned} \mathit{Val}(k) &= \mathsf{lcp}(i, k) \\ \mathit{Candidate}(k) &\equiv k + l \le i \text{ or } l = 0 \\ \mathit{FirstMin}(i, k) &= \mathsf{FirstMinPos}(\mathsf{RANK}[i] + 1, \mathsf{RANK}[k], \mathsf{LCP}) \\ \mathit{NextCand}(i, j) &= \mathsf{MinValue}(\mathsf{RANK}[i] + 1, j - 1, \mathsf{SUF}) \end{aligned}$$

Hence, by Lemmata 1 and 8, each iteration of the for loop takes $O(\mathsf{LPnF}^M_>[i] - \mathsf{LPnF}^M_>[i-1] + 2)$ time, and the overall time complexity of Algorithm 7 is $O(n + \mathsf{LPnF}^M_>[n-1] - \mathsf{LPnF}^M_>[0]) = O(n)$.

The correctness of the algorithm follows from the fact that (for each i) the body of the while loop is executed at least once (as a consequence of 8). ☐

Theorem 3. *The LPnF array can be computed in $O(n)$ time (without using the suffix trees). For (polynomially bounded) integer alphabets the complexity does not depend on the size of the alphabet.*

Proof. The table $\mathsf{LPnF}^M_<$ can be computed using similar approach in $O(n)$ time. As already mentioned, the LPnF^O array can also be computed in $O(n)$ time. Then, $\mathsf{LPnF}[i] = \max(\mathsf{LPnF}^M_<[i], \mathsf{LPnF}^M_>[i], \mathsf{LPnF}^O[i])$. ☐

7. Applications to text compression

Several text compression algorithms and many related software are based on factorizations of input text in which each element is a factor of the text occurring at a previous position possibly extended by one character (see [1] for variants of the scheme). We assume, to simplify the description, that the current element occurs before as it is done in LZ77 parsing [28].

Algorithm 9: AbstractSemiGreedyfactorization(w)

```
$i = 1$; $j = 0$; $n = |w|$ ;
while $i \le n$ do
    $j = j + 1$ ;
    if $w[i]$ doesn't appear in $w[1\,..\,(i-1)]$ then $f_j = w[i]$;
    else
        $f_j = u$ such that $uv$ is the longest prefix of $w[i\,..\,n]$ for which $u$
        appears before position $i$ and $v$ appears before position $i + |u|$.
    $i = i + |f_j|$ ;
return $(f_1 \ldots f_j)$
```

An improvement on the scheme, called optimal parsing, has been proposed in [15]. It optimizes the parsing by utilizing a semi-greedy algorithm. The algorithm reduces the number of elements of the factorization. Algorithm 9 is an abstract semi-greedy algorithm for computing factorization of the word w. At a given step, instead of choosing the longest factor starting at position i and occurring before, which is the greedy technique, the algorithm chooses the factor whose next factor goes to the furthest position. The semi-greedy scheme is simple to implement with the LPF table. We should also note, that LPnrF array can be used to construct reverse Lempel-Ziv factorization described in [21] in $O(n)$ time, while in [21] authors present $O(n \log |\Sigma|)$ algorithm.

Combining reverse and non-reverse types of factorization is a mere application of the LPF (or LPnF) and LPnrF tables as shown in Algorithm 10. We get the next statement as a conclusion of the section.

Theorem 4. *The optimal parsing using factors and reverse factors can be computed in linear time independently of the alphabet size.*

Algorithm 10: LinearTimeSemiGreedyfactorization(w)

```
$i = 1$; $j = 0$; $n = |w|$ ;
compute LPF and LPnrF arrays for word $w$ ;
let $\text{MAXF}[i] = \max\{\mathsf{LPF}[i], \mathsf{LPnrF}[i]\}$ ;
let $\text{MAXF}^+[i] = \text{MAXF}[i] + i$ ;
prepare $\text{MAXF}^+$ for range maximum queries ;
while $i \leq n$ do
    $j = j + 1$ ;
    if $w[i]$ doesn't appear in $w[1 .. (i-1)]$ then $f_j = w[i]$;
    else
        let $k = \text{MAXF}[i]$ ;
        find $i \leq q < i + k$ such that $\text{MAXF}^+[q]$ is maximal ;
        $f_j = w[i .. q]$ ;
return $(f_1 .. f_j)$
```

References

[1] Bell, T. C., Clearly, J. G., Witten, I. H., 1990. Text Compression. Prentice Hall Inc., New Jersey.

[2] Bender, M. A., Farach-Colton, M., 2000. The LCA problem revisited. In: Gonnet, G. H., Panario, D., Viola, A. (Eds.), Latin American Theoretical INformatics (LATIN). Vol. 1776 of Lecture Notes in Computer Science. Springer, pp. 88–94.

[3] Böckenhauer, H.-J., Bongartz, D., 2007. Algorithmic Aspects of Bioinformatics. Springer, Berlin.

[4] Chen, G., Puglisi, S. J., Smyth, W. F., 2008. Lempel-ziv factorization using less time & space. Mathematics in Computer Science 1 (4), 605–623.

[5] Crochemore, M., 1986. Transducers and repetitions. Theoretical Computer Science 45 (1), 63–86.

[6] Crochemore, M., Hancart, C., Lecroq, T., 2007. Algorithms on Strings. Cambridge University Press, Cambridge, UK.

[7] Crochemore, M., Ilie, L., 2008. Computing Longest Previous Factor in linear time and applications. Information Processing Letters 106 (2), 75–80.

[8] Crochemore, M., Ilie, L., Iliopoulos, C., Kubica, M., Rytter, W., Waleń, T., 2009. LPF computation revisited. In: Fiala, J., Kratochvl, J., Miller, M. (Eds.), International Workshop on Combinatorial Algorithms. Vol. 5874 of Lecture Notes in Computer Science. Springer, Berlin, pp. 158–169.

[9] Crochemore, M., Ilie, L., Smyth, W. F., 2008. A simple algorithm for computing the Lempel-Ziv factorization. In: Storer, J. A., Marcellin, M. W. (Eds.), 18th Data Compression Conference. IEEE Computer Society, Los Alamitos, CA, pp. 482–488.

[10] Fischer, J., Heun, V., 2006. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In: Lewenstein, M., Valiente, G. (Eds.), Proc. Symposium on Combinatorial Pattern Matching (CPM). Vol. 4009 of Lecture Notes in Computer Science. Springer, pp. 36–48.

[11] Fischer, J., Heun, V., 2007. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: Chen, B., Paterson, M., Zhang, G. (Eds.), Proceedings of the International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE'07). Vol. 4614 of Lecture Notes in Computer Science. Springer-Verlag, Hangzhou, China, April 7–9, 2007, pp. 459–470.

[12] Franek, F., Holub, J., Smyth, W. F., Xiao, X., 2003. Computing quasi suffix arrays. Journal of Automata, Languages and Combinatorics 8 (4), 593–606.

[13] Gabow, H., Bentley, J., Tarjan, R., 1984. Scaling and related techniques for geometry problems. In: Symposium on the Theory of Computing (STOC). pp. 135–143.

[14] Grumbach, S., Tahi, F., 1993. Compression of DNA sequences. In: Data Compression Conference. pp. 340–350.

[15] Hartman, A., Rodeh, M., 1985. Optimal parsing of strings. In: Apostolico, A., Galil, Z. (Eds.), Combinatorial algorithms on words. Vol. 12 of Computer and System Sciences. Springer, Berlin, pp. 155–167.

[16] Kärkkäinen, J., Sanders, P., 2003. Simple linear work suffix array construction. In: Baeten, J. C. M., Lenstra, J. K., Parrow, J., Woeginger, G. J. (Eds.), ICALP. Vol. 2719 of Lecture Notes in Computer Science. Springer, pp. 943–955.

[17] Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K., 2001. Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Proc. Symposium on Combinatorial Pattern Matching (CPM). Vol. 2089 of Lecture Notes in Computer Science. Springer, pp. 181–192.

[18] Kim, D. K., Sim, J. S., Park, H., Park, K., 2003. Linear-time construction of suffix arrays. In: Proc. Symposium on Combinatorial Pattern Matching (CPM). Vol. 2676 of Lecture Notes in Computer Science. Springer, pp. 186–199.

[19] Ko, P., Aluru, S., 2003. Space efficient linear time construction of suffix arrays. In: Proc. Symposium on Combinatorial Pattern Matching (CPM). Vol. 2676 of Lecture Notes in Computer Science. Springer, pp. 200–210.

[20] Kolpakov, R. M., Kucherov, G., 1999. Finding maximal repetitions in a word in linear time. In: Proc. Symposium on Foundations of Computer Science (FOCS). pp. 596–604.

[21] Kolpakov, R. M., Kucherov, G., 2008. Searching for gapped palindromes. In: Ferragina, P., Landau, G. M. (Eds.), Combinatorial Pattern Matching, 19th Annual Symposium, Pisa, Italy, June 18-20, 2008. Vol. 5029 of Lecture Notes in Computer Science. Springer, Berlin, pp. 18–30.

[22] Main, M. G., 1989. Detecting leftmost maximal periodicities. Discret. Appl. Math. 25, 145–153.

[23] Manacher, G. K., 1975. A new linear-time “on-line” algorithm for finding the smallest initial palindrome of a string. J. ACM 22 (3), 346–351.

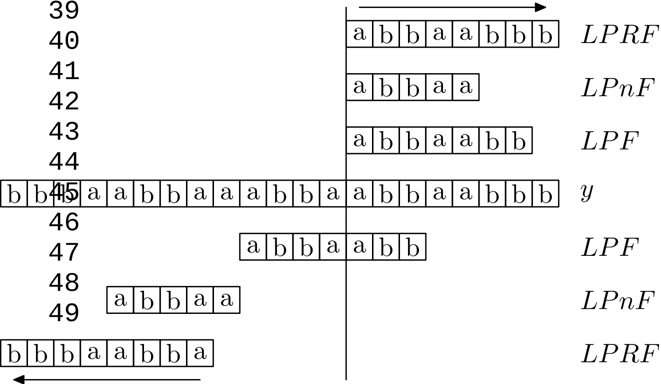
[24] Manber, U., Myers, E. W., 1993. Suffix arrays: A new method for on-line string searches. SIAM J. Comput. 22 (5), 935–948.

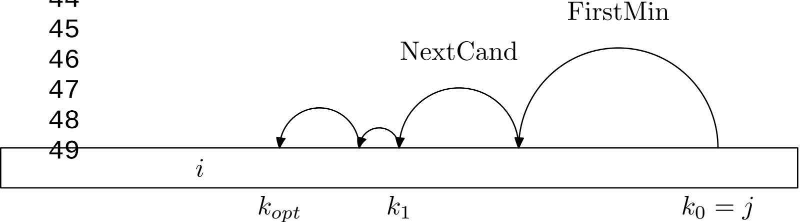
[25] Nong, G., Zhang, S., Chan, W. H., 2009. Linear time suffix array construction using D-critical substrings. In: Kucherov, G., Ukkonen, E. (Eds.), Proc. Symposium on Combinatorial Pattern Matching (CPM). Vol. 5577 of Lecture Notes in Computer Science. Springer, pp. 54–67.

[26] Sadakane, K., 2007. Succinct data structures for flexible text retrieval systems. Journal of Discrete Algorithms 5 (1), 12–22.

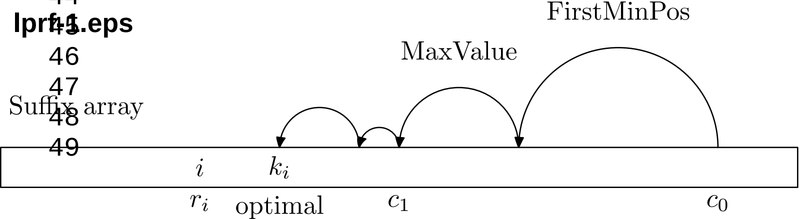
[27] Tischler, G., 2009. Personal communication.

[28] Ziv, J., Lempel, A., 1977. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, 337–343.

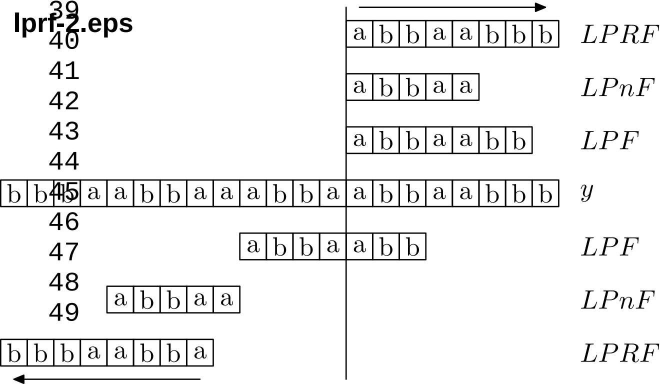

abbaabbb
LPRF
abbaa
LPnF
abbaabb
LPF
bbbaabbaaabbaabbaabbb
y
abbaabb
LPF
abbaa
LPnF
bbbaabba
LPRF


FirstMin
NextCand
i
k_{opt}
k_1
$k_0 = j$

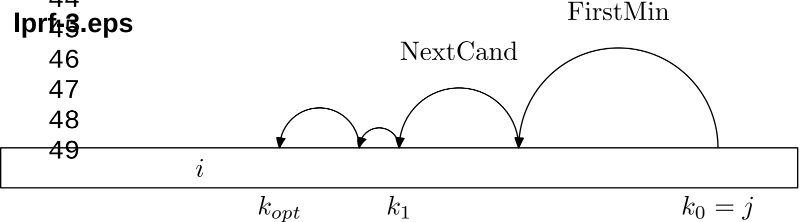
lprf-1.eps



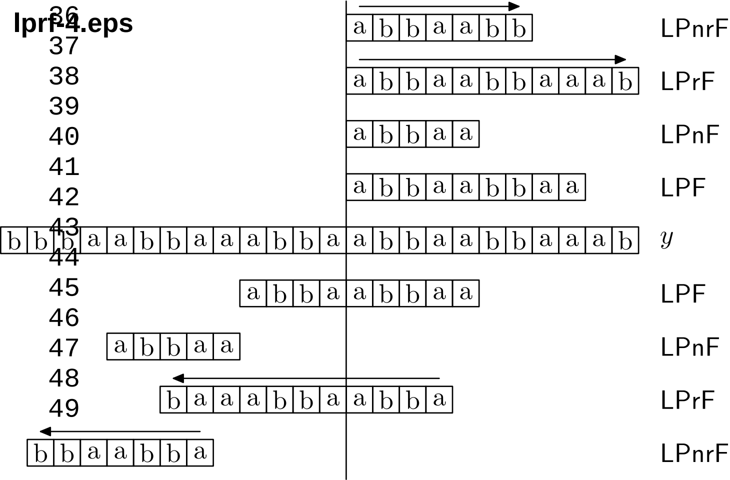
lprf-2.eps


a b b a a b b b
LPRF
a b b a a
LPnF
a b b a a b b
LPF
b b b a a b b a a a b b a a b b a a b b b
y
a b b a a b b
LPF
a b b a a
LPnF
b b b a a b b a
LPRF

lprf-3.eps



lprf-4.eps


a b b a a b b
LPnrF
a b b a a b b a a a b
LPrF
a b b a a
LPnF
a b b a a b b a a
LPF
b b b a a b b a a a b b a a b b a a b b a a a b
y
a b b a a b b a a
LPF
a b b a a
LPnF
b a a a b b a a b b a
LPrF
b b a a b b a
LPnrF

lprf-5.eps