Alternative Algorithms for Lyndon Factorization*

Sukhpal Singh Ghuman[1], Emanuele Giaquinta[2], and Jorma Tarhio[1]

[1] Department of Computer Science and Engineering, Aalto University
P.O.B. 15400, FI-00076 Aalto, Finland
{Sukhpal.Ghuman,Jorma.Tarhio}@aalto.fi

[2] Department of Computer Science, P.O.B. 68, FI-00014 University of Helsinki, Finland
Emanuele.Giaquinta@cs.helsinki.fi

Abstract. We present two variations of Duval's algorithm for computing the Lyndon factorization of a word. The first algorithm is designed for the case of small alphabets and is able to skip a significant portion of the characters of the string, for strings containing runs of the smallest character in the alphabet. Experimental results show that it is faster than Duval's original algorithm, more than ten times in the case of long DNA strings. The second algorithm computes, given a run-length encoded string R of length ρ, the Lyndon factorization of R in $O(\rho)$ time and constant space.

1 Introduction

Given two strings w and w', w' is a rotation of w if $w = uv$ and $w' = vu$, for some strings u and v. A string is a Lyndon word if it is lexicographically smaller than all its proper rotations. Every string has a unique factorization in Lyndon words such that the corresponding sequence of factors is nonincreasing with respect to lexicographical order. This factorization was introduced by Chen, Fox and Lyndon [2]. Duval's classical algorithm [3] computes the factorization in linear time and constant space. The Lyndon factorization is a key ingredient in a recent method for sorting the suffixes of a text [8], which is a fundamental step in the construction of the Burrows-Wheeler transform and of the suffix array, as well as in the bijective variant of the Burrows-Wheeler transform [4] [6]. The Burrows-Wheeler transform is an invertible transformation of a string, based on the sorting of its rotations, while the suffix array is a lexicographically sorted array of the suffixes of a string. They are the basis for important data compression methods and text indexes. Although Duval's algorithm runs in linear time and is thus efficient, it can still be useful to further improve the time for the computation of the Lyndon factorization in the cases where the string is either huge or compressible and given in a compressed form.

Various alternative algorithms for the Lyndon factorization have been proposed in the last twenty years. Apostolico and Crochemore presented a parallel algorithm [1], while Roh *et al.* described an external memory algorithm [10]. Recently, I *et al.* showed how to compute the Lyndon factorization of a string given in grammar-compressed form and in Lempel-Ziv 78 encoding [5].

In this paper, we present two variations of Duval's algorithm. The first variation is designed for the case of small alphabets like the DNA alphabet {a, c, g, t}. If the string contains runs of the smallest character, the algorithm is able to skip a significant portion of the characters of the string. In our experiments, the new algorithm is more than ten times faster than the original one for long DNA strings.

* Supported by the Academy of Finland (grant 134287).

Sukhpal Singh Ghuman, Emanuele Giaquinta, Jorma Tarhio: Alternative Algorithms for Lyndon Factorization, pp. 169–178.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

The second variation is for strings compressed with run-length encoding. The run-length encoding of a string is a simple encoding where each maximal consecutive sequence of the same symbol is encoded as a pair consisting of the symbol plus the length of the sequence. Given a run-length encoded string R of length ρ, our algorithm computes the Lyndon factorization of R in $O(\rho)$ time and uses constant space. It is thus preferable to Duval's algorithm in the cases in which the strings are stored or maintained in run-length encoding.

2 Basic definitions

Let Σ be a finite ordered alphabet of symbols and let Σ^* be the set of words (strings) over Σ ordered by lexicographic order. The empty word ε is a word of length 0. Let also Σ^+ be equal to $\Sigma^* \setminus \{\varepsilon\}$. Given a word w, we denote with $|w|$ the length of w and with $w[i]$ the i-th symbol of w, for $0 \leq i < |w|$. The concatenation of two words u and v is denoted by uv. Given two words u and v, v is a substring of u if there are indices $0 \leq i, j < |u|$ such that $v = u[i] \cdots u[j]$. If $i = 0$ ($j = |u| - 1$) then v is a prefix (suffix) of u. We denote by $u[i..j]$ the substring of u starting at position i and ending at position j. For $i > j$ $u[i..j] = \varepsilon$. We denote by u^k the concatenation of k u's, for $u \in \Sigma^+$ and $k \geq 1$. The longest border of a word w, denoted with $\beta(w)$, is the longest proper prefix of w which is also a suffix of w. Let $lcp(w, w')$ denote the length of the longest common prefix of words w and w'. We write $w < w'$ if either $lcp(w, w') = |w| < |w'|$, i.e., if w is a proper prefix of w', or if $w[lcp(w, w')] < w'[lcp(w, w')]$. For any $0 \leq i < |w|$, $\text{ROT}(w, i) = w[i..|w| - 1]w[0..i - 1]$ is a rotation of w. A Lyndon word is a word w such that $w < \text{ROT}(w, i)$, for $1 \leq i < |w|$. Given a Lyndon word w, the following properties hold:

1. $|\beta(w)| = 0$;
2. either $|w| = 1$ or $w[0] < w[|w| - 1]$.

Both properties imply that no word a^k, for $a \in \Sigma$, $k \geq 2$, is a Lyndon word. The following result is due to Chen, Fox and Lyndon [7]:

Theorem 1. *Any word w admits a unique factorization $CFL(w) = w_1, w_2, \ldots, w_m$, such that w_i is a Lyndon word, for $1 \leq i \leq m$, and $w_1 \geq w_2 \geq \cdots \geq w_m$.*

The run-length encoding (RLE) of a word w, denoted by $\text{RLE}(w)$, is a sequence of pairs (runs) $\langle (c_1, l_1), (c_2, l_2,), \ldots, (c_\rho, l_\rho) \rangle$ such that $c_i \in \Sigma$, $l_i \geq 1$, $c_i \neq c_{i+1}$ for $1 \leq i < r$, and $w = c_1^{l_1} c_2^{l_2} \cdots c_\rho^{l_\rho}$. The interval of positions in w of the factor w_i in the Lyndon factorization of w is $[a_i, b_i]$, where $a_i = \sum_{j=1}^{i-1} |w_j|$, $b_i = \sum_{j=1}^{i} |w_j| - 1$. Similarly, the interval of positions in w of the run (c_i, l_i) is $[a_i^{rle}, b_i^{rle}]$ where $a_i^{rle} = \sum_{j=1}^{i-1} l_j$, $b_i^{rle} = \sum_{j=1}^{i} l_j - 1$.

3 Duval's algorithm

In this section we briefly describe Duval's algorithm for the computation of the Lyndon factorization of a word. Let L be the set of Lyndon words and let

$$P = \{w \mid w \in \Sigma^+ \text{ and } w\Sigma^* \cap L \neq \emptyset\},$$

be the set of nonempty prefixes of Lyndon words. Let also $P' = P \cup \{c^k \mid k \geq 2\}$, where c is the maximum symbol in Σ. Duval's algorithm is based on the following Lemmas, proved in [3]:

```
LF-Duval(w)
k ← 0
while k < |w| do
        i ← k + 1
        j ← k + 2
        while TRUE do
            if j = |w| + 1 or w[j − 1] < w[i − 1] then
                while k < i do
                    output(w[k..k + j − i])
                    k ← k + j − i
                break
            else
                if w[j − 1] > w[i − 1] then
                    i ← k + 1
                else
                    i ← i + 1
                j ← j + 1
```

Figure 1. Duval's algorithm to compute the Lyndon factorization of a string.

Lemma 2. *Let $w \in \Sigma^+$ and w_1 be the longest prefix of $w = w_1 w'$ which is in L. We have $CFL(w) = w_1 CFL(w')$.*

Lemma 3. $P' = \{(uv)^k u \mid u \in \Sigma^*, v \in \Sigma^+, k \geq 1$ *and* $uv \in L\}$.

Lemma 4. *Let $w = (uav')^k u$, with $u, v' \in \Sigma^*$, $a \in \Sigma$, $k \geq 1$ and $uav' \in L$. The following propositions hold:*

1. *For $a' \in \Sigma$ and $a > a'$, $wa' \notin P'$;*
2. *For $a' \in \Sigma$ and $a < a'$, $wa' \in L$;*
3. *For $a' = a$, $wa' \in P' \setminus L$.*

Lemma 2 states that the computation of the Lyndon factorization of a word w can be carried out by computing the longest prefix w_1 of $w = w_1 w'$ which is a Lyndon word and then recursively restarting the process from w'. Lemma 3 states that the nonempty prefixes of Lyndon words are all of the form $(uv)^k u$, where $u \in \Sigma^*, v \in \Sigma^+, k \geq 1$ and $uv \in L$. By the first property of Lyndon words, the longest prefix of $(uv)^k u$ which is in L is uv. Hence, if we know that $w = (uv)^k uav'$, $(uv)^k u \in P'$ but $(uv)^k ua \notin P'$, then by Lemma 2 and by induction we have $CFL(w) = w_1 w_2 \cdots w_k\, CFL(uav')$, where $w_1 = w_2 = \cdots = w_k = uv$. Finally, Lemma 4 explains how to compute, given a word $w \in P'$ and a symbol $a \in \Sigma$, whether $wa \in P'$, and thus makes it possible to compute the factorization using a left to right parsing. Note that, given a word $w \in P'$ with $|\beta(w)| = i$, we have $w[0..|w| - i - 1] \in L$ and $w = (w[0..|w| - i - 1])^q w[0..r - 1]$ with $q = \lfloor \frac{|w|}{|w|-i} \rfloor$ and $r = |w| \mod (|w| - i)$. For example, if $w = abbabbab$, we have $|w| = 8$, $|\beta(w)| = 5$, $q = 2$, $r = 2$ and $w = (abb)^2 ab$. The code of Duval's algorithm is shown in Figure 1.

The following is an alternative formulation of Duval's algorithm by I *et al.* [5]:

Lemma 5. *Let $j > 0$ be any position of a string w such that $w < w[i..|w| - 1]$ for any $0 < i \leq j$ and $lcp(w, w[j..|w| - 1]) \geq 1$. Then, $w < w[k..|w| - 1]$ also holds for any $j < k \leq j + lcp(w, w[j..|w| - 1])$.*

```
LF-SKIP(w)
e ← |w| − 1
while e ≥ 0 and w[e] = c̄ do
    e ← e − 1
l ← |w| − 1 − e
w ← w[0..e]
s ← min Occ_{c̄c̄}(w) ∪ {|w|}
LF-DUVAL(w[0..s − 1])
r ← 0
while s < |w| do
    w ← w[s..|w| − 1]
    while w[r] = c̄ do
        r ← r + 1
    s ← |w|
    k ← 1
    P ← {c̄^r c | c ≤ w[r]}
    j ← 0
    for i ∈ Occ_P(w) : i > j do
        h ← lcp(w, w[i..|w| − 1])
        if h = |w| − i or w[i + h] < w[h] then
            s ← i
            k ← 1 + ⌊h/s⌋
            break
        j ← i + h
    for i ← 1 to k do
        output(w[0..s − 1])
    s ← s × k
for i ← 1 to l do
    output(c̄)
```

Figure 2. The algorithm to compute the Lyndon factorization that can potentially skip symbols.

Lemma 6. *Let w be a string with $CFL(w) = w_1, w_2, \ldots, w_m$. It holds that $|w_1| = \min\{j \mid w[j..|w|-1] < w\}$ and $w_1 = w_2 = \cdots = w_k = w[0..|w_1|-1]$, where $k = 1 + \lfloor lcp(w, w[|w_1|..|w|-1])/|w_1| \rfloor$.*

Based on these Lemmas, Duval's algorithm can be implemented by initializing $j \leftarrow 1$ and executing the following steps until w becomes ε: 1) compute $h \leftarrow lcp(w, w[j..|w|-1])$. 2) if $j+h < |w|$ and $w[h] < w[j+h]$ set $j \leftarrow j+h+1$; otherwise output $w[0..j-1]$ k times and set $w \leftarrow w[jk..|w|-1]$, where $k = 1 + \lfloor h/j \rfloor$, and set $j \leftarrow 1$.

4 Improved algorithm for small alphabets

Let w be a word over an alphabet Σ with $CFL(w) = w_1, w_2, \ldots, w_m$ and let $\bar{c}$ be the smallest symbol in Σ. Suppose that there exists $k \geq 2, i \geq 1$ such that $\bar{c}^k$ is a prefix of w_i. If the last symbol of w is not $\bar{c}$, then by Theorem 1 and by the properties of Lyndon words, $\bar{c}^k$ is a prefix of each of $w_{i+1}, w_{i+1}, \ldots, w_m$. This property can be exploited to devise an algorithm for Lyndon factorization that can potentially skip symbols. Our algorithm is based on the alternative formulation of Duval's algorithm by I *et al.*. Given a set of strings $\mathcal{P}$, let $Occ_{\mathcal{P}}(w)$ be the set of all (starting) positions in w corresponding to occurrences of the strings in $\mathcal{P}$. We start with the following Lemmas:

Lemma 7. *Let w be a word and let $s = \max\{i \mid w[i] > \bar{c}\} \cup \{-1\}$. Then, $CFL(w) = CFL(w[0..s])CFL(\bar{c}^{(|w|-1-s)})$.*

Proof. If $s = -1$ or $s = |w| - 1$ the Lemma plainly holds. Otherwise, Let w_i be the factor in $CFL(w)$ such that $s \in [a_i, b_i]$. To prove the claim we have to show that $b_i = s$. Suppose by contradiction that $s < b_i$, which implies $|w_i| \geq 2$. Then, $w_i[|w_i| - 1] = \bar{c}$, which contradicts the second property of Lyndon words. □

Lemma 8. *Let w be a word such that $\bar{c}\bar{c}$ occurs in it and let $s = \min Occ_{\{\bar{c}\bar{c}\}}(w)$. Then, we have $CFL(w) = CFL(w[0..s-1])CFL(w[s..|w|-1])$.*

Proof. Let w_i be the factor in $CFL(w)$ such that $s \in [a_i, b_i]$. To prove the claim we have to show that $a_i = s$. Suppose by contradiction that $s > a_i$, which implies $|w_i| \geq 2$. If $s = b_i$ then $w_i[|w_i|-1] = \bar{c}$, which contradicts the second property of Lyndon words. Otherwise, since w_i is a Lyndon word it must hold that $w_i < \text{ROT}(w_i, s - a_i)$. This implies at least that $w_i[0] = w_i[1] = \bar{c}$, which contradicts the hypothesis that s is the smallest element in $Occ_{\{\bar{c}\bar{c}\}}(w)$. □

Lemma 9. *Let w be a word such that $w[0] = w[1] = \bar{c}$ and $w[|w| - 1] \neq \bar{c}$. Let r be the smallest position in w such that $w[r] \neq \bar{c}$. Note that $w[0..r-1] = \bar{c}^r$. Let also $\mathcal{P} = \{\bar{c}^r c \mid c \leq w[r]\}$. Then we have*

$$b_1 = \min\{s \in Occ_{\mathcal{P}}(w) \mid w[s..|w|-1] < w\} \cup \{|w|\} - 1\,,$$

where b_1 is the ending position of factor w_1.

Proof. By Lemma 6 we have that $b_1 = \min\{s \mid w[s..|w|-1] < w\} - 1$. Since $w[0..r-1] = \bar{c}^r$ and $|w| \geq r+1$, for any string v such that $v < w$ we must have that either $v[0..r] \in \mathcal{P}$, if $|v| \geq r+1$, or $v = \bar{c}^{|v|}$ otherwise. Since $w[|w|-1] \neq \bar{c}$, the only position s that satisfies $w[s..|w|-1] = \bar{c}^{|w|-s}$ is $|w|$, corresponding to the empty word. Hence,

$$\{s \mid w[s..|w|-1] < w\} = \{s \in Occ_{\mathcal{P}}(w) \mid w[s..|w|-1] < w\} \cup \{|w|\}$$

□

Based on these Lemmas, we can devise a faster factorization algorithm for words containing runs of $\bar{c}$. The key idea is that, using Lemma 9, it is possible to skip symbols in the computation of b_1, if a suitable string matching algorithm is used to compute $Occ_{\mathcal{P}}(w)$. W.l.o.g. we assume that the last symbol of w is different from $\bar{c}$. In the general case, by Lemma 7, we can reduce the factorization of w to the one of its longest prefix with last symbol different from $\bar{c}$, as the remaining suffix is a concatenation of $\bar{c}$ symbols, whose factorization is a sequence of factors equal to $\bar{c}$. Suppose that $\bar{c}\bar{c}$ occurs in w. By Lemma 8 we can split the factorization of w in $CFL(u)$ and $CFL(v)$ where $uv = w$ and $|u| = \min Occ_{\{\bar{c}\bar{c}\}}(w)$. The factorization of $CFL(u)$ can be computed using Duval's original algorithm.

Concerning v, let $r = \min\{i \mid v[i] \neq \bar{c}\}$. By definition $v[0] = v[1] = \bar{c}$ and $v[|v|-1] \neq \bar{c}$, and we can apply Lemma 9 on v to find the ending position s of the first factor in $CFL(v)$. To this end, we have to find the position $\min\{i \in Occ_{\mathcal{P}}(v) \mid v[i..|v|-1] < v\}$, where $\mathcal{P} = \{\bar{c}^r c \mid c \leq v[r]\}$. For this purpose, we can use any algorithm for multiple string matching to iteratively compute $Occ_{\mathcal{P}}(v)$ until either a position i is found that satisfies $v[i..|v|-1] < v$ or we reach the end of the string. Let $h = lcp(v, v[i..|v|-1])$, for a given $i \in Occ_{\mathcal{P}}(v)$. Observe that $h \geq r$ and, if $v < v[i..|v|-1]$, then, by Lemma 5,

```
LF-RLE(R)
k ← 0
while k < |R| do
    (m, q) ← LF-RLE-NEXT(R, k)
    for i ← 1 to q do
        output (k, k + m − 1)
        k ← k + m
```

```
LF-RLE-NEXT(R = ⟨(c_1, l_1), ..., (c_ρ, l_ρ)⟩, k)
i ← k
j ← k + 1
while TRUE do
    if i > k and l_{j−1} < l_{i−1} then
        z ← 1
    else z ← 0
    s ← i − z
    if j = |R| or c_j < c_s or
        (c_j = c_s and l_j > l_s and c_j < c_{s+1}) then
        return (j − i, ⌊(j − k − z)/(j − i)⌋)
    else
        if c_j > c_s or l_j > l_s then
            i ← k
        else
            i ← i + 1
        j ← j + 1
```

Figure 3. The algorithm to compute the Lyndon factorization of a run-length encoded string.

we do not need to verify the positions $i' \in Occ_{\mathcal{P}}(v)$ such that $i' \leq i + h$. Given that all the patterns in $\mathcal{P}$ differ in the last symbol only, we can express $\mathcal{P}$ more succinctly using a character class for the last symbol and match this pattern using a string matching algorithm that supports character classes, such as the algorithms based on bit-parallelism. In this respect, SBNDM2 [11], a variation of the BNDM algorithm [9] is an ideal choice, as it is sublinear on average. Instead of $\mathcal{P}$, it is naturally possible to search for $\bar{c}^r$, but that solution is slower in practice for small alphabets. Note that the same algorithm can also be used to compute $\min Occ_{\bar{c}\bar{c}}(w)$ in the first phase.

Let $h = lcp(v, v[s..|v|-1])$ and $k = 1 + \lfloor h/s \rfloor$. Based on Lemma 6, the algorithm then outputs $v[0..s-1]$ k times and iteratively applies the above method on $v' = v[sk..|v|-1]$. It is not hard to verify that, if $v' \neq \varepsilon$, then $|v'| \geq r+1$, $v'[0..r-1] = \bar{c}$ and $v'[|v'|-1] \neq \bar{c}$, and so Lemma 9 can be used on v'. The code of the algorithm is shown in Figure 2. The computation of the value $r' = \min\{i \mid v'[i] \neq \bar{c}\}$ for v' takes advantage of the fact that $v'[0..r-1] = \bar{c}$, so as to avoid useless comparisons. If the the total time spent for the iteration over the sets $Occ_{\mathcal{P}}(v)$ is $O(|w|)$, the full algorithm has also linear time complexity in the worst case. To see why, it is enough to observe that the positions i for which the algorithm verifies if $v[i..|v|-1] < v$ are a subset of the positions verified by the original algorithm.

5 Computing the Lyndon factorization of a run-length encoded string

In this section we present an algorithm to compute the Lyndon factorization of a string given in RLE form. The algorithm is based on Duval's original algorithm and on a combinatorial property between the Lyndon factorization of a string and its RLE, and has $O(\rho)$-time and $O(1)$-space complexity, where ρ is the length of the RLE. We start with the following Lemma:

Lemma 10. *Let w be a word over Σ and let $w_1, w_2, \ldots, w_m$ be its Lyndon factorization. For any $1 \leq i \leq |\text{RLE}(w)|$, let $1 \leq j, k \leq m$, $j \leq k$, such that $a_i^{rle} \in [a_j, b_j]$ and $b_i^{rle} \in [a_k, b_k]$. Then, either $j = k$ or $|w_j| = |w_k| = 1$.*

Proof. Suppose by contradiction that $j < k$ and either $|w_j| > 1$ or $|w_k| > 1$. By definition of j, k, we have $w_j \geq w_k$. Moreover, since both $[a_j, b_j]$ and $[a_k, b_k]$ overlap with $[a_i^{rle}, b_i^{rle}]$, we also have $w_j[|w_j| - 1] = w_k[0]$. If $|w_j| > 1$, then, by definition of w_j, we have $w_j[0] < w_j[|w_j| - 1] = w_k[0]$. Instead, if $|w_k| > 1$ and $|w_j| = 1$, we have that w_j is a prefix of w_k. Hence, in both cases we obtain $w_j < w_k$, which is a contradiction. □

The consequence of this Lemma is that a run of length l in the RLE is either contained in *one* factor of the Lyndon factorization, or it corresponds to l unit-length factors. Formally:

Corollary 11. *Let w be a word over Σ and let $w_1, w_2, \ldots, w_m$ be its Lyndon factorization. Then, for any $1 \leq i \leq |\text{RLE}(w)|$, either there exists w_j such that $[a_i^{rle}, b_i^{rle}]$ is contained in $[a_j, b_j]$ or there exist l_i factors $w_j, w_{j+1}, \ldots, w_{j+l_i-1}$ such that $|w_{j+k}| = 1$ and $a_{j+k} \in [a_i^{rle}, b_i^{rle}]$, for $0 \leq k < l_i$.*

This property can be exploited to obtain an algorithm for the Lyndon factorization that runs in $O(\rho)$ time. First, we introduce the following definition:

Definition 12. *A word w is a LR word if it is either a Lyndon word or it is equal to a^k, for some $a \in \Sigma$, $k \geq 2$. The LR factorization of a word w is the factorization in LR words obtained from the Lyndon factorization of w by merging in a single factor the maximal sequences of unit-length factors with the same symbol.*

For example, the LR factorization of *cctgccaa* is $\langle cctg, cc, aa \rangle$. Observe that this factorization is a (reversible) encoding of the Lyndon factorization. Moreover, in this encoding it holds that each run in the RLE is contained in one factor and thus the size of the LR factorization is $O(\rho)$. Let L' be the set of LR words. We now present the algorithm LF-RLE-NEXT(R, k) which computes, given an RLE sequence R and an integer k, the longest LR word in R starting at position k. Analogously to Duval's algorithm, it reads the RLE sequence from left to right maintaining two integers, j and ℓ, which satisfy the following invariant:

$$\begin{aligned} &c_k^{l_k} \cdots c_{j-1}^{l_{j-1}} \in P'; \\ &\ell = \begin{cases} |\text{RLE}(\beta(c_k^{l_k} \cdots c_{j-1}^{l_{j-1}}))| & \text{if } j - k > 1, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{1}$$

The integer j, initialized to $k + 1$, is the index of the next run to read and is incremented at each iteration until either $j = |R|$ or $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}} \notin P'$. The integer ℓ, initialized to 0, is the length in runs of the longest border of $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}}$, if $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}}$ spans at least two runs, and equal to 0 otherwise. For example, in the case of the word ab^2ab^2ab we have $\beta(ab^2ab^2ab) = ab^2ab$ and $\ell = 4$. Let $i = k + \ell$. In general, if $\ell > 0$, we have

$$\begin{aligned} &l_{j-1} \leq l_{i-1}, l_k \leq l_{j-\ell}, \\ &\beta(c_k^{l_k} \cdots c_{j-1}^{l_{j-1}}) = c_k^{l_k} c_{k+1}^{l_{k+1}} \cdots c_{i-2}^{l_{i-2}} c_{i-1}^{l_{j-1}} = c_{j-\ell}^{l_k} c_{j-\ell+1}^{l_{j-\ell+1}} \cdots c_{j-2}^{l_{j-2}} c_{j-1}^{l_{j-1}}. \end{aligned}$$

Note that the longest border may not fully cover the last (first) run of the corresponding prefix (suffix). Such the case is for example for the word ab^2a^2b. However, since $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}} \in P'$ it must hold that $l_{j-\ell} = l_k$, i.e., the first run of the suffix is fully covered. Let

$$z = \begin{cases} 1 & \text{if } \ell > 0 \wedge l_{j-1} < l_{i-1}, \\ 0 & \textit{otherwise.} \end{cases}$$

Informally, the integer z is equal to 1 if the longest border of $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}}$ does not fully cover the run (c_{i-1}, l_{i-1}). By 1 we have that $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}}$ can be written as $(uv)^q u$, where

$$\begin{aligned} &q = \lfloor \tfrac{j-k-z}{j-i} \rfloor, r = z + (j-k-z) \mod (j-i), \\ &u = c_{j-r}^{l_{j-r}} \cdots c_{j-1}^{l_{j-1}}, uv = c_k^{l_k} \cdots c_{j-\ell-1}^{l_{j-\ell-1}} = c_{i-r}^{l_{i-r}} \cdots c_{j-r-1}^{l_{j-r-1}}, \\ &uv \in L' \end{aligned}$$

For example, in the case of the word ab^2ab^2ab, for $k = 0$, we have $j = 6, i = 4, q = 2, r = 2$. The algorithm is based on the following Lemma:

Lemma 13. *Let j, ℓ be such that invariant 1 holds and let $s = i - z$. Then, we have the following cases:*

1. *If $c_j < c_s$ then $c_k^{l_k} \cdots c_j^{l_j} \notin P'$;*
2. *If $c_j > c_s$ then $c_k^{l_k} \cdots c_j^{l_j} \in L'$ and 1 holds for $j+1$, $\ell' = 0$;*

Moreover, if $z = 0$, we also have:

3. *If $c_j = c_i$ and $l_j \leq l_i$, then $c_k^{l_k} \cdots c_j^{l_j} \in P'$ and 1 holds for $j+1$, $\ell' = \ell + 1$;*
4. *If $c_j = c_i$ and $l_j > l_i$, either $c_j < c_{i+1}$ and $c_k^{l_k} \cdots c_j^{l_j} \notin P'$ or $c_j > c_{i+1}$, $c_k^{l_k} \cdots c_j^{l_j} \in L'$ and 1 holds for $j+1$, $\ell' = 0$.*

Proof. The idea is the following: we apply Lemma 4 with the word $(uv)^q u$ as defined above and symbol c_j. Observe that c_j is compared with symbol $v[0]$, which is equal to $c_{k+r-1} = c_{i-1}$ if $z = 1$ and to $c_{k+r} = c_i$ otherwise.

First note that, if $z = 1$, $c_j \neq c_{i-1}$, since otherwise we would have $c_{j-1} = c_{i-1} = c_j$. In the first three cases, we obtain the first, second and third proposition of Lemma 4, respectively, for the word $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}} c_j$. Independently of the derived proposition, it is easy to verify that the same proposition also holds for $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}} c_j^m$, $m \leq l_j$. Consider now the fourth case. By a similar reasoning, we have that the third proposition of Lemma 4 holds for $c_k^{l_k} \cdots c_j^{l_i}$. If we then apply Lemma 4 to $c_k^{l_k} \cdots c_j^{l_i}$ and c_j, c_j is compared to c_{i+1} and we must have $c_j \neq c_{i+1}$ as otherwise $c_i = c_j = c_{i+1}$. Hence, either the first (if $c_j < c_{i+1}$) or the second (if $c_j > c_{i+1}$) proposition of Lemma 4 must hold for the word $c_k^{l_k} \cdots c_j^{l_i+1}$. □

We prove by induction that invariant 1 is maintained. At the beginning, the variables j and ℓ are initialized to $k+1$ and 0, respectively, so the base case trivially holds. Suppose that the invariant holds for j, ℓ. Then, by Lemma 13, either $c_k^{l_k} \cdots c_j^{l_j} \notin P'$ or it follows that the invariant also holds for $j+1, \ell'$, where ℓ' is equal to $\ell + 1$, if $z = 0$, $c_j = c_i$ and $l_j \leq l_i$, and to 0 otherwise. When $c_k^{l_k} \cdots c_j^{l_j} \notin P'$ the algorithm returns the pair $(j - i, q)$, i.e., the length of uv and the corresponding exponent. Based on

Lemma 2, the factorization of R can then be computed by iteratively calling LF-RLE-NEXT. When a given call to LF-RLE-NEXT returns, the factorization algorithm outputs the q factors uv starting at positions k, $k+(j-i)$, ..., $k+(q-1)(j-i)$ and restarts the factorization at position $k+q(j-i)$. The code of the algorithm is shown in Figure 3. We now prove that the algorithm runs in $O(\rho)$ time. First, observe that, by definition of LR factorization, the for loop at line 4 is executed $O(\rho)$ times. Suppose that the number of iterations of the while loop at line 2 is n and let $k_1, k_2, \ldots, k_{n+1}$ be the corresponding values of k, with $k_1 = 0$ and $k_{n+1} = |R|$. We now show that the s-th call to LF-RLE-NEXT performs less than $2(k_{s+1} - k_s)$ iterations, which will yield $O(\rho)$ number of iterations in total. This analysis is analogous to the one used by Duval. Suppose that i', j' and z' are the values of i, j and z at the end of the s-th call to LF-RLE-NEXT. The number of iterations performed during this call is equal to $j' - k_s$. We have $k_{s+1} = k_s + q(j' - i')$, where $q = \lfloor \frac{j'-k_s-z}{j-i'} \rfloor$, which implies $j' - k_s < 2(k_{s+1} - k_s) + 1$, since, for any positive integers x, y, $x < 2\lfloor x/y \rfloor y$ holds.

6 Experiments with LF-Skip

The experiments were run on MacBook Pro with the 2.4 GHz Intel Core 2 Duo processor and 2 GB memory. Programs were written in the C programming language and compiled with the gcc compiler (4.8.2) using the -O3 optimization level.

We tested the LF-skip algorithm and Duval's algorithm with various texts. With the protein sequence of the Saccharomyces cerevisiae genome (3 MB), LF-skip gave a speed-up of 3.5 times over Duval's algorithm. Table 1 shows the speed-ups for random texts of 5 MB with various alphabets sizes. With longer texts, speed-ups were larger. For example, the speed-up for the 50 MB DNA text (without newlines) from the Pizza&Chili Corpus[1] was 14.6 times.

$\|\Sigma\|$	Speed-up
2	9.0
3	7.7
4	7.2
5	6.1
6	4.8
8	4.3
10	3.5
12	3.4
15	2.4
20	2.5
25	2.2
30	1.9

Table 1. Speed-up of LF-skip with various alphabet sizes in a random text.

We made also some tests with texts of natural language. Because runs are very short in natural language, the benefit of LF-skip is marginal. We even tried alphabet transformations in order to vary the smallest character of the text, but that did not help.

[1] http://pizzachili.dcc.uchile.cl/

7 Conclusions

In this paper we have presented two variations of Duval's algorithm for computing the Lyndon factorization of a string. The first algorithm was designed for the case of small alphabets and is able to skip a significant portion of the characters of the string for strings containing runs of the smallest character in the alphabet. Experimental results show that the algorithm is considerably faster than Duval's original algorithm. The second algorithm is for strings compressed with run-length encoding and computes the Lyndon factorization of a run-length encoded string of length ρ in $O(\rho)$ time and constant space.

References

1. A. Apostolico and M. Crochemore: *Fast parallel Lyndon factorization with applications*. Mathematical Systems Theory, 28(2) 1995, pp. 89–108.
2. K. T. Chen, R. H. Fox, and R. C. Lyndon: *Free differential calculus. IV. The quotient groups of the lower central series*. Annals of Mathematics, 68(1) 1958, pp. 81–95.
3. J.-P. Duval: *Factorizing words over an ordered alphabet*. J. Algorithms, 4(4) 1983, pp. 363–381.
4. J. Y. Gil and D. A. Scott: *A bijective string sorting transform*. CoRR, abs/1201.3077 2012.
5. T. I, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda: *Faster lyndon factorization algorithms for SLP and LZ78 compressed text*, in SPIRE, O. Kurland, M. Lewenstein, and E. Porat, eds., vol. 8214 of Lecture Notes in Computer Science, Springer, 2013, pp. 174–185.
6. M. Kufleitner: *On bijective variants of the Burrows-Wheeler transform*, in Proceedings of the Prague Stringology Conference 2009, J. Holub and J. Žďárek, eds., 2009, pp. 65–79.
7. M. Lothaire: *Combinatorics on Words*, Cambridge Mathematical Library, Cambridge University Press, 1997.
8. S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino: *Sorting suffixes of a text via its Lyndon factorization*, in Proceedings of the Prague Stringology Conference 2013, J. Holub and J. Žďárek, eds., 2013, pp. 119–127.
9. G. Navarro and M. Raffinot: *Fast and flexible string matching by combining bit-parallelism and suffix automata*. ACM Journal of Experimental Algorithmics, 5 2000.
10. K. Roh, M. Crochemore, C. S. Iliopoulos, and K. Park: *External memory algorithms for string problems*. Fundam. Inform., 84(1) 2008, pp. 17–32.
11. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching*. Inf. Process. Lett., 110(4) 2010, pp. 148–152.