JOURNAL OF ALGORITHMS 4, 363–381 (1983)

Factorizing Words over an Ordered Alphabet

JEAN PIERRE DUVAL

Laboratoire d'Informatique, Faculté des Sciences, Université de Rouen B.P. 67, 76130 Mont Saint Aignan, France

Received May 15, 1982

An efficient algorithm to obtain a factorization of words over an ordered alphabet known as Lyndon factorization is presented. Applications of this algorithm are given to the computation of the least suffix of a word and the least circular shift of a word.

INTRODUCTION

A number of algorithms deal with linear arrays of symbols, also called strings or words. This contains for example string-matching algorithms and, more generally, the algorithms related to the theory of automata.

A special case occurs when the set of symbols is an ordered set. This leads of course to the vast domain of sorting algorithms. The purpose of this paper is to show how a computation of a special decomposition of a string allows the simultaneous solution of several special sorting problems such as finding the lexicographically least circular substring, finding the minimum suffix of a string. The first problem has recently been studied by K. S. Booth [2] and by V. Siloach [8]. The second one is usually treated by the method of the so-called "position tree" (see [1], for instance).

The decomposition of words that we study here is the following: any word w can be written uniquely

$$w = w_1 w_2 \dots w_n,$$

where

(1) the sequence $w_1, w_2, \dots, w_n$ is nonincreasing for lexicographic order,

(2) each w_i is strictly less than any of its proper circular shift.

This decomposition is due to Chen, Fox, and Lyndon [3] and it has been introduced for the purpose of computing a basis of the free Lie algebras. It is a particular case of a factorization of free monoids (see [7]).

0196-6774/83 $3.00
Copyright © 1983 by Academic Press, Inc.
All rights of reproduction in any form reserved.

In the first section, we introduce the necessary background to handle Lyndon words. Some of the material is of common knowledge to the specialists of this field. Some other is not standard; such is the characterization of the left factors of Lyndon words which is used as a basis for the algorithms presented further. A complete study of this type of property appears in [5].

The second section presents the algorithms of factorization of a word into Lyndon words. Two *variants* of this algorithm are described. They differ by the use in the second of an auxiliary table which plays a role which is similar to the "failure function" used in the Knuth–Morris–Pratt algorithm [6]. A preliminary version of these algorithms has been presented in [4].

The third section presents three applications of the factorization algorithm: the first one is the computation of the *minimum* suffix of a word; the second, of the *maximum* suffix. The third one is the computation of the least circular shift of a word. This problem has been already considered in [2] and also in [8].

I. Factorizing into Lyndon Words

We consider a fixed alphabet A which is a totally ordered set. We denote by A^* the free monoid over the set A. A^+ is A^* minus the empty word. The free monoid A^* is ordered by lexicographic order. Therefore, for $u, v \in A^*$ one has

$$u < v$$

iff either u is a prefix of v and $u \neq v$, or

$$u = ras, \qquad v = rbt$$

with $r, s, t \in A^*$, $a, b \in A$, and $a < b$.

The following property is an immediate consequence of the above definition:

Let $u, v \in A^*$ with $u < v$. If u is not a prefix of v, for $g, h \in A^*$ one has $ug < vh$.

By definition, a *Lyndon word* is a word $w \in A^+$ such that whenever $w = uv$ with u, v nonempty, then

$$uv < vu.$$

Note that $uv \neq vu$. For example, $w = ababb$ is a Lyndon word but $abbab$ is not since $ababb < abbab$. For $u = v = ab$, one has $uv = vu$; thus $abab$ is not a Lyndon word. For $k \geqslant 2$ and $w \in A^*$, w^k is not a Lyndon word.

We denote by L the set of Lyndon words. Since a single letter is trivially a Lyndon word one has

$$A \subset L.$$

PROPOSITION 1.1. *A Lyndon word has no proper factor which is both a prefix and a suffix of the word.*

Proof. Let $w \in A^+$ and u be a proper factor which is both a prefix and a suffix of w. The word w has the form $w = uv'$ and $w = v''u$ for some $v', v'' \in A^+$.

Suppose that $w \in L$. According to the definition we have that $w < v'u$ and $w < uv''$. Then $v''u < v'u$ and $uv' < uv''$. Thus $v'' < v'$ and $v' < v''$, this is impossible. Then $w \notin L$. □

PROPOSITION 1.2. *A word $w \in A^+$ is in L iff for each proper suffix v of w one has $w < v$.*

Proof. *If*: for $w = uv$ with u, v nonempty, if $uv < v$ then $uv < vu$. *Only if*: let $w \in L$ and v be a proper suffix of $w = uv$. We have then $uv < vu$. By Proposition 1.1, v is not a prefix of uv. Then $uv < v$. □

As a consequence we have:

$$\text{Let } w = a'ua'' \in L \text{ with } a', a'' \in A, u \in A^*. \text{ Then } w < a'' \text{ and } a' < a''. \tag{1.1}$$

PROPOSITION 1.3. *Let $u, v \in L$: $uv \in L$ iff $u < v$.*

Proof. *Only if*: by Proposition 1.2, if $uv \in L$, $u < uv < v$. *If*: let s be a proper suffix of uv. Then s has the form either $s = u'v$ with u' a proper suffix of u, either $s = v$, or $s = v'$ with v' a proper suffix of v.

For u' a proper suffix of $u \in L$. $u < u'$ and u' is not a prefix of u. Then, $uv < u'v$.

If u is not a prefix of v, since $u < v$ one has $uv < v$.

If u is a prefix of v, v has the form $v = uv''$ with v'' a proper suffix of v. Therefore, $v < v''$ and $uv < uv''$; $uv < v$.

Then, we have that $uv < v$.

For v' a proper suffix of v, $v < v'$. Then $uv < v < v'$.

Therefore, in both cases we have that $uv < s$. By Proposition 1.2, it follows that $uv \in L$. □

The following consequence is not difficult. It stands without proof.

COROLLARY 1.4. *Let $u, v \in L$ and $u < v$. For $k, k' \geqslant 1$, $u^k v^{k'} \in L$.*

It is a particular case of the above property that:

$$\text{For } w \in L, a \in A, k \geqslant 1 \text{ and } w < a, \text{ one has } w^k a \in L. \tag{1.2}$$

PROPOSITION 1.5. *Let $u \in A^*$, $v \in A^+$ with $uv \in L$. If $a \in A$ and $v < a$, $ua \in L$.*

Proof. We assume that u is a nonempty word. Let s be a proper suffix of ua. It has the form $s = u'a$, where u' is a suffix of u and $u' \neq u$. Then, $u'v$ is a proper suffix of uv. Since $uv \in L$, we have that $uv < u'v$. Since $v < a$, we have that $uv < u'a$. Then, u is not a prefix of $u'a$, thus $ua < u'a$. Therefore, by Proposition 1.2, ua is in L. □

Remark. In Proposition 1.5, we may have $v' \in L$ with $v < v'$, instead of $a \in A$ with $v < a$, then $uv' \in L$.

Our interest turns to a characterization of the nonempty prefixes of Lyndon words. We denote by P the set of these prefixes

$$P = \{w | w \in A^+ \text{ and } wA^* \cap L \neq \varnothing\}.$$

Let c be the maximum letter in A if there is one (that is the case if A is finite). We denote

$$P' = P \cup \{c^k | k \geqslant 2\}.$$

If there is no maximum letter in A, $P' = P$.

One has

$$L \subset P.$$

LEMMA 1.6. *Let $w = (uav')^k u$ with $u, v' \in A^*$, $a \in A$, $k \geqslant 1$ and $uav' \in L$. The following propositions hold:*

(1) *For $a' \in A$ and $a < a'$, $wa' \in L$.*

(2) *For $a' = a$, $wa' \in P' - L$.*

(3) *For $a' \in A$ and $a > a'$, $wa' \notin P'$.*

Proof. Let $a' \in A$ with $a < a'$. We have that $uav' \in L$ and $av' < a'$. Then, by Proposition 1.5, $ua' \in L$. Since $uav' \in L$, $ua' \in L$ and $uav' < ua'$, and according to Corollary 1.4, we have that $(uav')^k ua' \in L$. This proves (1).

The word ua is both a proper prefix and a suffix of $(uav')^k ua$. Then $(uav')^k ua$ is not in L, $wa \notin L$. If uav' is reduced to the maximum letter c of A, we have that u, v are the empty word and $a = c$; therefore, $wa = c^{k+1}$ and $wa \in P'$. If uav' is not reduced to the maximum letter c (which is true if there is no maximum in A), according to 1.1 we may find a letter a'' such

that $uav' < a''$; therefore, by (1.2), $(uav')^{k+1}a'' \in L$; then wa is a prefix of $(uav')^{k+1}a''$, $wa \in P'$. This proves (2).

Let $a' \in A$ with $a > a'$. For $h \in A^*$, we have that $(uav')^k ua'h < ua'h$. Therefore, $wa'h$ is not in L. Then, wa' is not a prefix of some Lyndon word. Since a' cannot be the maximum in A, it follows that $wa' \notin P'$. This proves (3). □

We denote by S the set of strict sesquipowers of the words in L, that is,

$$S = \{(uv)^k u | u \in A^*, v \in A^+, k \geqslant 1 \text{ and } uv \in L\}.$$

PROPOSITION 1.7. $P' = S$.

Proof. Let $w \in S$, w has the form $w = (uav')^k u$ with $u \in A^*$, $v' \in A^*$, $a \in A$, $k \geqslant 1$ and $uav' \in L$. By Lemma 1.6, $wa \in P'$. Therefore, w is a prefix of a word in P'. Then $w \in P'$. This proves $S \subset P'$.

Let $v \in L$; we have $v = (uv)^1 u$, where u is the empty word. Therefore, $v \in S$. This proves $L \subset S$.

Then, we have to prove that $P' - L \subset S$. This is more interesting.

The proof is by induction on the length of the word.

A word of length 1 is reduced to a single letter. Since $A \subset L \subset S$, it is in S.

Assume that each word of $P' - L$ with length less than or equal to $n - 1$ is in S.

Let $w \in P' - L$ with length n. w has the form $w = w_1 a$ with $a \in A$. Then w_1 has length $n - 1$ and w_1 is in P'. According to the induction hypothesis w_1 is in S. Then w_1 has the form $w_1 = (u_1 a_1 v_1')^{k_1} u_1$ with $u_1, v_1' \in A^*$, $a_1 A$, $k > 1$ and $u_1 a_1 v_1' \in L$. Then $(u_1 a_1 v_1')^{k_1} u_1 a$ is in $P' - L$, and according to Lemma 1.6, we have that $a_1 = a$. Therefore, we have that w has the form $(uv)^k u$, where u, v, k has the respective value, either $u = u_1 a_1$, $v = v_1'$, $k = k_1$ if v_1' is nonempty, or u is the empty word $v = u_1 a_1 v_1'$, $k = k_1 + 1$ if v_1' is empty. In both cases we have that $uv \in L$, $v \in A^+$ and $k \geqslant 1$. Therefore $w \in S$.

Now then, we have $P' \subset S$. □

Therefore, to recognize if a word is a possible prefix of a Lyndon word we have to recognize if it is in S. Lemma 1.6 gives us a method to do this by reading the word from left to right. This the basic idea used in Section II.

The proof of the following result can be found in [7].

THEOREM 1.8 (CHEN, FOX, LYNDON). *Any word $w \in A^*$ admits a unique factorization.*

$$w = w_1 w_2 \ldots w_m \tag{1.3}$$

such that each w_i $(1 \leqslant i \leqslant m)$ *is a Lyndon word and*

$$w_1 \geqslant w_2 \geqslant \ldots \geqslant w_m. \tag{1.4}$$

The decomposition (1.3) will be called in the sequel the *standard factorization* of the word w. We denote it by

$$\mathrm{CFL}(w) = w_1 w_2 \ldots w_m.$$

It is not difficult to see that such a factorization exists. In fact, in any factorization of w into Lyndon words which does not satisfies (1.2) we can group two consecutive factors u, v with $u < v$ (Proposition 1.3) and obtain a factorization with fewer factors. The process may be trivially started since the letters are Lyndon words.

The unicity of the standard factorization is a consequence of the following proposition, either by property (1), (2), or (3).

PROPOSITION 1.9. *Let* $w \in A^+$ *and* $\mathrm{CFL}(w) = w_1 w_2 \ldots w_m$. *The following properties hold*:

(1) w_m *is the minimum suffix of* w.

(2) w_m *is the longest suffix of* w *which is in* L.

(3) w_1 *is the longest prefix of* w *which is in* L.

Proof. Let s be a suffix of w. It has the form

$$s = w_i' w_{i+1} \ldots w_m,$$

where w_i' is a nonempty suffix of w_i and $i \leqslant m$.

Since $w_i \in L$, we have that $w_i' \geqslant w_i$. Then $w_i' \geqslant w_i \geqslant \ldots \geqslant w_m$. Therefore w_m is the minimum suffix of w. This proves (1).

Assume that s is strictly longer than w_m. Then $i < n$ and $w_i' < s$. Therefore, $w_m < s$ and s is not in L. This proves (2).

Let p be a prefix of w. It has the form

$$p = w_1 \ldots w_j',$$

where w_j' is a nonempty prefix of w_j. If p is strictly longer than w_1 we have that $j > 1$. Therefore, $w_j' \leqslant w_j \ldots \leqslant w_1$ and $w_1 < p$. Then p is not in L. This proves (3). □

PROPOSITION 1.10. *Let* $w \in A^+$ *and* w_1 *be the longest prefix of* $w = w_1 w'$ *which is in* L. *We have*

$$\mathrm{CFL}(w) = w_1 \mathrm{CFL}(w').$$

Proof. Let $\mathrm{CFL}(w') = w_2 \ldots w_n$. Since $w_1 w_2 \notin L$, it follows from Pro-

position 1.2 that $w_1 \geqslant w_2$. Therefore $\mathrm{CFL}(w) = w_1\mathrm{CFL}(w')$. □

PROPOSITION 1.11. *Let* $w = (uav')^q ua'h$ *with* $u, v', h \in A^*$, $a, a' \in A$, $q \geqslant 1$ *and* $uav' \in L$. *Let*

$$w_1 = w_2 = \cdots = w_q = uav'.$$

We have that

$$\mathrm{CFL}\big((uav')^q u\big) = w_1 w_2 \ldots w_q \,\mathrm{CFL}(u)$$

and, for $a > a'$

$$\mathrm{CFL}(w) = w_1 w_2 \ldots w_q \,\mathrm{CFL}(ua'h) \qquad \textit{whatever } h \in A^*.$$

Proof. Since a Lyndon word has no proper prefix and suffix, the longest prefix of $(uav')^q u$ which is in L is a prefix of uav'. Therefore it is uav'. According to Proposition 1.10, we have that

$$\mathrm{CFL}\big((uav')^q u\big) = w_1\,\mathrm{CFL}\big((uav')^{q-1} u\big)$$

and, by induction for $q \geqslant 2$,

$$\mathrm{CFL}\big((uav')^q u\big) = w_1 w_2 \ldots w_q \,\mathrm{CFL}(u).$$

Assume $a > a'$, according to Lemma 1.6, $(uav')^q ua'h \notin P'$ (whatever $h \in A^*$). Therefore, the longest prefix of w which is in L is a prefix of $(uav')^q u$. The same arguments as above lead us to

$$\mathrm{CFL}(w) = w_1 w_2 \ldots w_q \,\mathrm{CFL}(ua'h). \;\square$$

Now then, Proposition 1.11 and Lemma 1.6 give a method for factorizing a word into Lyndon words by reading it from left to right. This is done in Section II. But, before we do it, let us see the problem of minimum suffix. It is obviously related; according to Proposition 1.9 the minimum suffix of a word is the last term of the factorization.

We denote by $\mathrm{minsuf}(w)$ the minimum nonempty suffix of $w \in A^+$.

PROPOSITION 1.12. *Let* $uv \in L$ *with* $u, v \in A^+$. *Then,* $\mathrm{minsuf}(u)$ *is a prefix of* u.

Proof. Let $s = \mathrm{minsuf}(u)$. We have that $s \leqslant u$. If $s \neq u$ and s is not a prefix of u, then $sv < uv$. Since $uv \in L$, we have that $uv \leqslant sv$. Therefore, s is a prefix of u. □

PROPOSITION 1.13. *Let* $w = (uav')^q ua'h$ *with* $u, v', h \in A^*$, $a, a \in A$, $q \geqslant 1$ *and* $uav' \in L$. *Let* sa *be the minimum suffix of* ua. *The following*

propositions holds:

(1) *If* $a < a'$, $\text{minsuf}((uav')^q ua') = (uav'^q)ua'$.

(2) *If* $a = a'$, $\text{minsuf}((uav')^q ua') = \text{minsuf}(ua)$.

(3) *If* $a > a'$, $\text{minsuf}((uav')^q ua') = \text{minsuf}(sh)$ *whatever* $h \in A^*$.

Proof. For $a < a'$, it follows from Lemma 1.6 that $(uav')^q ua' \in L$. This proves (1).

For $a = a'$, according to Proposition 1.11, the last term in the factorization of $(uav')^q ua$ is the last in the factorization of ua. Therefore, $\text{minsuf}((uav')^q ua) = \text{minsuf}(ua)$. This proves (2).

For $a > a'$, according to Proposition 1.11, the last term in the standard factorization of w is the last term in the factorization of $ua'h$. Therefore $\text{minsuf}((uav')^q ua'h) = \text{minsuf}(ua'h)$. Let s' be a suffix of u longer than s. We have that $sa < s'a$. Since sa is shorter than s', we have that $sa < s'a'h$. Therefore $sa'h < sa < s'a'h$. Then $s'a'h$ is not the minimum suffix of $ua'h$. Now then $\text{minsuf}(ua'h)$ is a suffix of $sa'h$. This proves (3). □

II. Factorization Algorithm

We present an algorithm to compute the standard factorization of a word. It is in time linear in the length of the word. In fact there are two algorithms. The first one (Algorithm 2.1) has the interesting property that it uses only three variables for a complete computation; it requires no more than $2n$ comparisons between two letters, where n is the length of the word. The second one (Algorithm 2.2) is slightly faster in that sense that it requires no more than $3n/2$ comparisons but it uses an auxiliary storage of size $n/2$.

The basic idea is the same for both of these algorithms. It consists in reading the word from left to right. The variable j is the index of the current input letter, and the variable i is such that when we read the letter a_j one has

$$a_1 \dots a_{i-1} = a_{j-i+1} \dots a_{j-1} \qquad \text{and} \qquad a_1 \dots a_{j-i} \in L. \tag{2.1}$$

Initial values are $i = 1$ and $j = 2$. For $a_i < a_j$ (that is, case 1) or $a_i = a_j$ (that is, case 2), the variable i is computed in order that (2.1) still holds for the next value $j + 1$ of j. Then, let i' and j' be the respective values of i and j at the first time we find $a_i > a_j$ (or $j = n + 1$). We have that $a_1 \dots a_{j'-i'}$ is the first factor in the standard factorization of $a_1 \dots a_n$. The other factors are obtained by factorization $a_{j'-i'+1} \dots a_n$. The variable k is introduced ($k = j' - i'$) in order that the remaining suffix is $a_{k+1} \dots a_n$. The prefix $a_1 \dots a_k$ is no more considered.

At this step the corresponding property for (2.1) is

$$a_{k+1} \dots a_{i-1} = a_{k+j-i+1} \dots a_{j-1} \quad \text{and} \quad a_{k+1} \dots a_{j-i} \in L. \tag{2.1'}$$

And, for $k < i'$, (2.1') holds for $i = i'$ and $j = j'$.

If we have that $k \geqslant i$, the process may restart at the beginning of the remaining suffix $a_{k+1} \dots a_n$ by taking $i = k + 1$ and $j = k + 2$. That is the method for Algorithm 2.1. But this implies a rescanning from a_{k+1} to $a_{j'}$. Such a rescanning is avoided in Algorithm 2.2 by introducing a table f, such that if $k \geqslant i'$ we have that (2.1') holds for the values $i = k + f[j' - k]$ and $j = j'$.

Then, Algorithm 2.1 is as follows:

Algorithm 2.1

Input: a word string $a_1 \dots a_n$ of letters over A.

Output: the sequence FACT $= (k_1, k_2, \dots, k_m)$ such that, for

$$w_1 = a_1 \dots a_{k_1}, w_2 = a_{k_1+1} \dots a_{k_2}, \dots, w_m = a_{k_{m-1}+1} \dots a_m$$

one has

$$\mathrm{CFL}(a_1 \dots a_n) = w_1 w_2 \dots w_m. \tag{2.2}$$

The method is described as follows:

```
{FACT: sequence of integers}:
begin FACT: = the empty sequence; k: = 0
      while k < n do begin
                  i: = k + 1; j: = k + 2;
      99: case "compare a_i: : a_j" of
                  1 {a_i < a_j}: (i: = k + 1; j: = j + 1; goto 99)
                  2 {a_i = a_j}: (i: = i + 1; j: = j + 1; goto 99)
                  3 {a_i > a_j or j = n + 1}: (repeat k: = k + (j - i);
                                                  add k in FACT
                                              until k ⩾ i)
          endcase
      endwhile
end
```

Let us prove the correctness of the above program.

Let k', i', and j' be the respective values of k, i, and j at the first time we enter the while loop with $k \neq 0$. Thus, i' and j' are the values of i and j at the first time we have that $a_i > a_j$ or $j = n + 1$.

Since $k = 0$ from $j = 2$ to this step, we may ignore the variable k in a first analysis. And show that (2.1) holds from $j = 2$ to j'.

For $i = 1$ and $j = 2$, $a_1 \dots a_{i-j}$ and $a_{j-i+1} \dots a_{j-1}$ are the empty word and $a_1 \dots a_{j-i}$ is reduced to one letter a_1. Therefore, (2.1) holds.

Then assume that (2.1) holds at step j. The comparison is between the letters a_i and a_j.

$$\textit{For } a_i = a_j, \text{ (2.1) } \textit{holds with the new respective values } i + 1 \textit{ and } j + 1 \textit{ for } i \textit{ and } j. \tag{2.3}$$

This is quite clear. It is the justification for case 2 statement in the program.

For the current values i and j we define $q(i, j)$, $r(i, j)$, $u(i, j)$, $a(i, j)$, and $v'(i, j)$ as follows:

$$q = \left\lfloor \frac{j-1}{j-i} \right\rfloor, \qquad r = \operatorname{mod}(j-1, j-i), \qquad u = a_1 \dots a_r,$$
$$a = a_{r+1}, \qquad \text{and} \qquad v' = a_{r+2} \dots a_{j-i}. \tag{2.4}$$

Since (2.1) holds we have that

$$uav' = a_1 \dots a_{j-i} \in L; \qquad a = a_i \qquad \text{and} \qquad a_1 \dots a_{j-1} = (uav')^q u. \tag{2.5}$$

Then, we have that

$$\textit{For } a_i < a_j, \text{ (2.1) } \textit{holds with the new respective values } 1 \textit{ and } j + 1 \textit{ for } i \textit{ and } j. \tag{2.6}$$

Proof. We have that $uav' \in L$ and $a < a_j$. According to Lemma 1.6, we have that $(uav')^q u a_j \in L$. Therefore, by (2.5), $a_1 \dots a_j \in L$. And (2.1) holds for $i_1 = 1$ and $j_1 = j + 1$. This proves (2.6). □

This is the justification for case 1 statement in the program.

Therefore, according to (2.3) and (2.6), (2.1) holds from $j = 2$ to j'. We define, with $q = q(i', j')$,

$$k_1 = (j' - i'), \qquad k_2 = 2(j' - i'), \dots, k_q = q(j' - i')$$

and

$$w_1 = a_1 \dots a_{k_1}, \qquad w_2 = a_{k_1+1} \dots a_{k_2}, \dots, w_q = a_{k_{q-1}+1} \dots a_{k_q}. \tag{2.7}$$

Note that, according to (2.1), we have that $w_1 = w_2 = \cdots = w_q$.

Then, we have that $a_{i'} > a_{j'}$ (or $j' = n + 1$) and

$$\mathrm{CFL}(a_1 \dots a_n) = w_1 w_2 \dots w_q \,\mathrm{CFL}\big(a_{k_q+1} \dots a_n\big). \tag{2.8}$$

Proof. We assume the notation (2.4) for $i = i'$ and $j = j'$. Then we have that $a_1 \dots a_n = (uav')^q u a_{j'} \dots a_n$.

Since $a > a_{j'}$ (or $j' = n + 1$) and $uav' \in L$, it follows from Proposition 1.11 that $\mathrm{CFL}(a_1 \dots a_n) = w_1 w_2 \dots w_q \,\mathrm{CFL}(u a_{j'} \dots a_n)$.

Since $k_q = q(j' - i')$, $u a_{j'} \dots a_n = a_{k_q+1} \dots a_n$.

This proves (2.8). □

Now then, we are in case 3 statement with $i' = j'$. The *repeat* statement consists to add in FACT the successive $k_1, k_2, \dots, k_q$ as defined in (2.7). When the *repeat* statement ends the variable k has the value $k' = k_q$. Then, we enter the while loop with $k = k'$; it is the first time that $k \neq 0$ when entering the while loop.

At this step the situation is as follows:

$$\left.\begin{array}{l}\text{We have in FACT the } k_1, k_2, \dots, k_q \text{ corresponding to}\\ \text{the first } q \text{ factors in (2.2). The remaining factors are}\\ \text{those of the standard factorization of } a_{k'+1} \dots a_n.\\ a_{k'+1} \dots a_{j'-1} = a_1 \dots a_r, \text{ with } r = j' - 1 - k'.\end{array}\right\} \tag{2.9}$$

Then, $a_1 \dots a_{k'}$ is no more considered, and the process continues for $a_{k'+1} \dots a_n$. The corresponding property to (2.1) is now (with $k = k'$)

$$a_{k'+1} \dots a_{i-1} = a_{k+j-i+1} \dots a_{j-1} \qquad \text{and} \qquad a_{k+1} \dots a_{k+j-i} \in L. \tag{2.1'}$$

THEOREM 2.1. *Algorithm* 2.1 *computes the standard factorization of* $a_1 \dots a_n$ *as denoted by* (2.2). *It requires no more than* $2n$ *comparisons between two letters and uses only three variables.*

Proof. Let k', i', and j' be the respective values of k, i, and j at the first time we enter the while loop with $k \neq 0$. It follows from the above discussion that we have in FACT the values $k_1, k_2, \dots, k_q$ corresponding to the first q factors as defined in (2.2). Remaining factors may be obtained by factorizing $a_{k'+1} \dots a_n$.

Note that at this step the number of times we have "compare $a_i :: a_j$" is $j' - 1 : j' - 2$ with $j = 2$ to $j' - 1$ and $a_i = a_j$ or $a_i < a_j$, one more for the last comparison $a_{i'} > a_{j'}$ or $j' = n + 1$. According to $k' = q(j' - i')$ with

$q = \lfloor (j' - 1)/(j' - i') \rfloor$ we have that $j' - 1 < 2k'$. Therefore, no more than $2k'$ comparisons have been used at this step.

Now then, if $k' = n$, $a_{k'+1} \dots a_n$ is the empty word and the process ends after no more than $2n$ comparisons.

If $k' < n$, the process continues with $i = k' + 1$ and $j = k' + 2$. This is just the same as for an initial word $a_{k'+1} \dots a_n$. Therefore, by induction on the length of the words, we may assume that it computes the standard factorization of $a_{k'+1} \dots a_n$. And it requires no more than $2(n - k')$ comparisons between two letters.

Thus, when the process ends, FACT is correct according to (2.2) and it requires no more than $2n$ comparisons between two letters.

The auxiliary variables are k, i, and j. □

Let us consider now Algorithm 2.2 described below.

Algorithm 2.2.

Input: A word string $a_1 \dots a_n$ of letters over A.
Output: The sequence FACT as defined by (2.2).
{FACT: sequence of integers; f: array $[2 \dots \lfloor n + 1/2 \rfloor]$ of integers}
begin: FACT: = the empty sequence; k: = 0; j: = 2, $f[2]$: = 1;
while $k < n$ **do begin**
i: = $k + f[j - k]$;
99: **case** "compare a_i::a_j" **of**
1 $\{a_i < a_j\}$: (i: = $k + 1$; j: = $j + 1$; *if* $j \leqslant (n + 1)$ **div** 2 **then** $f[j - h]$: = $i - k$; **goto** 99)
2 $\{a_i = a_j\}$: (i: = $i + 1$; j: = $j + 1$; *if* $j \leqslant (n + 1)$ **div** 2 **then** $f[j - k]$: = $i - k$; **goto** 99)
3 $\{a_i > a_j$ **or** $j = n + 1\}$: (**repeat** k: = $k + (j - i)$; **add** k *in FACT*
until $k \geqslant i$; **if** $k = j - 1$ **then** j: = $j + 1$)
endcase
endwhile
end {"if $j \leqslant (n + 1)$ **div** 2 **then**" is optional}

Let k', i', and j' be the respective values of k, i, and j at the first time we enter the while loop with $k \neq 0$ in Algorithm 2.2. The process from the starting point to this step is just the same as for Algorithm 2.1, except that we have affected some values to f. Therefore the above discussion for Algorithm 2.1 is still correct at this step.

Then the process differs. In Algorithm 2.1 we restart with $i = k' + 1$ and $j = k' + 2$. Then we rescan the word from $a_{k'+1}$ to $a_{j'}$. And we have that $a_{k'+1} \dots a_{j'-1} = a_1 \dots a_r$ according to (2.9).

Let us define the table f by

$$\textit{For } j = 2 \textit{ to } j', (2.1) \textit{ holds for } i = f[j]. \tag{2.11}$$

Note that f *is uniquely defined.*

Proof. Suppose that for some j, we have that (2.1) holds for $i = i_1$ and for $i = i_2 (i_1 < i_2)$.

Then from (2.1) we have that $a_{j-i_2+1} \dots a_{j-i_1} = a_1 \dots a_{i_2-i_1}$.

Thus $a_1 \dots a_{i_2-i_1}$ is both a prefix and a suffix of $a_1 \dots a_{j-i_1}$. Therefore, $a_1 \dots a_{j-i_1}$ is not in L contradicting (2.1) for $i = i_1$. □

Therefore, for $j = 2$ to j', $f[j]$ is the corresponding value of i when processing Algorithm 2.1. This is the justification for "$f[j-k] := i-k$" in the case 1 and case 2 statements of Algorithm 2.2 (note that $k = 0$).

Since $a_{k'+1} \dots a_{j'-1} = a_1 \dots a_r$, and f satisfies (2.11) at step k', i', j' it is true that

$$\textit{For } j = k' + 2 \textit{ to } j', (2.1') \textit{ holds for } i = k' + f[j-k']. \tag{2.11'}$$

Then, we have to process the suffix $a_{k'+1} \dots a_n$ and according to (2.11') we may continue with $i = k' + f[j' - k']$ and $j = j'$ as the new values for i and j. The table f is still correct for $j = k' + 2$ to j'. This is the method in Algorithm 2.2. And there is no more rescanning of $a_{k'+1} \dots a_{j'-1}$.

Note that the conditional "*if* $j - 1 \leqslant (n+1)$ *div* 2 *then*" introduces a restriction on the definition of f. But, according to the definition of $k' = q(j' - i')$ and $r = j' - 1 - k'$ as in (2.4) we have that $j' - k' = r + 1 \leqslant j'/2 \leqslant (n+1)/2$. Thus, the table f is never used in the range $](n+1)/2, n]$.

Note that if it occurs that $k' = j' - 1$, $a_{k'+1} \dots a_{j'-1}$ is an empty word and the process restarts with $j = k' + 2$ as the new value for j. This is the justification for "*if* $k = j - 1$ *then* $j := j + 1$" in the case 3 statement.

THEOREM 2.2. *Algorithm* 2.2 *computes the standard factorization of* $a_1 \dots a_n$. *It requires no more than* $3n/2$ *comparisons between two letters and an auxiliary storage of size* $n/2$.

Proof. It follows from the above discussion that the algorithm is correct.

We may count the number of comparisons by charging them either to a letter or to a factorization factor.

If $a_i = a_j$ or $a_i < a_j$, charge the letter a_j. Then j is increased by one and a_j is no more charge.

If $a_1 > a_j$ or $j = n + 1$. A new factor is obtained. If the new value k is $j - 1$, then charge the letter a_j; in this case j is increased by one and a_j is no more charge. If the new value k is such that $k < j - 1$, then charge the new factor; it has length $j - i$ and, since $k < j - 1$, $j - i$ does not divide $j - 1$; thus $j - i \geqslant 2$.

The total charge is: n for the letters $a_2, \ldots, a_n$, "a_{n+1}," and, since the factors of length 1 are never charged, no more than $n/2$ for the factors. Then $3n/2$. □

III. Applications

A first application concerns the computation of the minimum suffix of a word. Since the minimum suffix of a word is the last factor in the standard factorization of this word (Proposition 1.9), Algorithm 2.1 may be used to compute it, using only three indices and no more than $2n$ comparisons between two letters. Algorithm 2.2 may be used too.

But we can do better and compute for a given word $a_1 \ldots a_n$ the table M defined by

$$\textit{For } 1 \leqslant j \leqslant n,\ a_{M(j)+1} \ldots a_j \textit{ is the minimum suffix of } a_1 \ldots a_j. \quad (3.1)$$

This is possible by a slight improvement of either Algorithm 2.1 or 2.2; Algorithm 3.1, described below, includes the use of the table M not only for the result but also to have a faster move of variable k.

Algorithm 3.1

Input: a word string $a_1 \ldots a_n$ of letters over A.
Output: the table $M[1 \ldots n]$ according to (3.1).
{MINSUF: array $[1 \ldots n]$ of integers; $f[2 \ldots n]$: array of integers}
begin: $k := 0$; $j := 2$; $M[1] := 0$; $f[2] := 1$;
while $j \leqslant n$ **do begin**
$i := k + f[j - k]$;
99: **case** "compare $a_i :: a_j$" *of*
1 $\{a_i < a_j\}$: ($M[j] := k$; $i := k + 1$; $j := j + 1$; $f[j - k] := i - k$; **goto** 99)
2 $\{a_i = a_j\}$: ($M[j] := M[i] + j - i$; $i := i + 1$; $j := j + 1$; $f[j - k] := i - k$; **goto** 99)
3 $\{a_i > a_j\}$: ($k := M[i] + j - i$; **if** $k = j - 1$ **then begin** $M[j] := k$; $j := j + 1$ **end**)

endcase
endwhile
end

Let i' and j' be the respective values of i and j at the first time we find $a_i > a_j$. From $j = 2$ to j', Algorithm 3.1 is just the same as Algorithm 2.2.

Thus we have that (2.1) holds and with the notation (2.4).

$$uav' = a_1 \dots a_{j-1} \in L, \qquad a = a_1, \qquad \text{and} \qquad a_1 \dots a_{j-1} = (uav')^q u.$$

For $a_i < a_j$ $(2 \leqslant j \leqslant j')$, according to Proposition 1.13 we have that $\text{minsuf}(a_1 \dots a_j) = a_1 \dots a_j$.

Therefore $M(j) = 0$ ($M(j) = k$ for $k \neq 0$). This is the justification for "$M(j) := k$" in case 1 statement.

For $a_i = a_j$ $(2 \leqslant j < j')$, according to Proposition 1.13 we have that $\text{minsuf}(uav')^q ua_j) = \text{minsuf}(ua_j)$ and this is $\text{minsuf}((uav')^{q-1} ua_j)$. Therefore,

$$\text{minsuf}(a_1 \dots a_j) = \text{minsuf}(a_1 \dots a_i).$$

Thus $a_{M(j)+1} \dots a_j = a_{M(i)+1} \dots a_i$. Then $M(j) = M(i) + j - i$. This is the justification for the case 2 statement.

Now then, for j' we have that $a_{i'} > a_{j'}$. Let sa be the minimum suffix of ua. Note that according to Proposition 1.13 it is the minimum suffix of $(uav')^{q-1} ua$ which is $a_1 \dots a_{i'}$. According to Proposition 1.13 we have that $\text{minsuf}(a_1 \dots a_{j'} h) = \text{minsuf}(sa_{j'} h)$ whatever $h \in A^*$. Let $k' = M(i') + j' - i'$. Since $a_{M(i')+1} \dots a_{i'-1} = a_{M(i')+j'-i'+1} \dots a_{j'-1} = s$, we have that

$$\text{minsuf}(a_1 \dots a_{j'} h) = \text{minsuf}(a_{k'+1} \dots a_{j'} h) \text{ whatever } h \in A^*.$$

This is the justification for "$k := M(i) + j - i$" in case 3 statement. But what about the f corresponding to $a_{k'+1} \dots a_{j'}$ for the new value k' of k?

According to Proposition 1.12, since $uav' \in L$ we have that sa is a prefix of ua. Therefore $a_{k'+1} \dots a_{j'-1} = a_1 \dots a_{j'-1-k}$. Then the table f is correct for $a_{k'+1} \dots a_{j'-1}$, when we have it relative to the basis index k. That is truly what is done. Therefore, the process may continue and same arguments as above still holds.

The case $k' = j' - 1$, is a particular one since we have that $a_{k'+1} \dots a_{j'} = a_{j'}$. In such case the word reduced to a single letter is its own minimum suffix, and we process the next j. This is the justification for "**if** $k = j - 1$ **then begin** $M(j) := k$; $j := j + 1$ **end**" in case 3 statement.

THEOREM 3.1. *Algorithm* 3.1 *computes for the word* $a_1 \dots a_n$ *the table* $M[1 \dots n]$ *such that*

$$a_{M(j)+1} \dots a_j = \text{minsuf}(a_1 \dots a_j) \qquad (\textit{for } 1 \leqslant j \leqslant n).$$

It requires no more than $3n/2$ *comparisons between two letters.*

Proof. Correctness follows from the above discussion. The number of comparisons is less than or equal to the number of comparisons for Algorithm 2.2. Then, it is no more than $3n/2$. □

The second application concerns the computation of the maximum suffix of a word. This problem is not completely symmetrical to the preceding one as it is shown below.

The natural way to have a connection between maximum and minimum is to consider the converse order $\mathcal{R}$ over the alphabet A which is defined by

$$\text{for } a, b \in A,\ b\mathcal{R}a \qquad \text{iff } a < b.$$

It induces on A^* the lexicographic converse order denoted by $\mathcal{R}$. Note that for $u, v \in A^*$ we may have $u < v$ and $u\mathcal{R}v$. For example, $u = aba$ and $v = abab$ we have that $u < v$ and $u\mathcal{R}v$. It is not difficult to see that

$$\text{For } u, v \in A^*,\ u < v \text{ and } u\mathcal{R}v \text{ iff } u \text{ is a prefix of } v.$$

Thus, a word which is its own maximum suffix with respect to the order $\mathcal{R}$, is not necessarily a Lyndon word with respect to the direct order $<$.

We denote by P'' the set of words which are there own maximum suffix with respect to the converse order $\mathcal{R}$.

$$\text{If } uv \in P'' \text{ with } u \in A^+, \text{ then } u \in P''.$$

Proof. Let u' be a proper suffix of u. Since $uv \in P''$ we have that $u'v\mathcal{R}uv$. Therefore $u'\mathcal{R}uv$. Since u' is shorter than u, it follows $u'\mathcal{R}u$. Then u is its own maximum suffix and $u \in P''$. □

$$\text{If } u \in L \text{ then } u \in P''.$$

Proof. Let u' be a proper suffix of u. Since $u \in L$, we have that $u < u'$ and u' is not a prefix of u. Then, $u'\mathcal{R}u$. Therefore u is its own maximum suffix with respect to $\mathcal{R}$. □

The set P of prefixes of Lyndon words is defined in Section I. And P' is $P' = P \cup \{c^k | k \geqslant 2\}$ if there is a maximum letter c in A; otherwise $P' = P$.

We have that $P' \subset P''$.

Proof. Since, $L \subset P''$ and each prefix of a word of P'' is in P'', we have that $P \subset P''$. It is clear that for $a \in A$ and $k \geqslant 2$, we have that $a^k \in P''$. Then $P' \subset P''$. □

PROPOSITION 3.2. $P' = P''$.

Proof. Let $w \in P''$ and w' be the longest prefix of w which is in P'. According to Proposition 1.7, we have that w' has the form $w' = (uav')^q u$ with $u, v' \in A^*$, $a \in A$, $q \geqslant 1$, and $uav' \in L$.

Suppose that $w \neq w'$. Let $a' \in A$, $h \in A^*$, be such that $w = w'a'h$. According to Lemma 1.6, since $w'a' \notin P'$ we have that $a' < a$. Then $a\mathcal{R}a'$ and $(uav)^q ua'h\mathcal{R}ua'h$. Therefore w is not its own maximum suffix with respect to the converse order, contradicting $w \in P''$.

Therefore $w = w'$, and $w \in P'$. Then $P'' \subset P'$.
Since $P' \subset P''$, we have that $P' = P''$. □

We have a complete characterization of P'' as prefixes of Lyndon words. It is quite clear that the maximum suffix of a word is the longest suffix of this word which is its own maximum suffix. Therefore the problem of finding the maximum suffix of a word with respect to the converse order $\mathfrak{R}$ is just the same as the problem of finding the longest suffix which is in P'.

Let $a_1 \dots a_n$ be a word of length n. We define the table RMAX[$1 \dots n$] by for $1 \leqslant j \leqslant n$, $a_{\mathrm{RMAX}[j]+1} \dots a_j$ is the maximum of $a_1 \dots a_j$ with respect to the converse order $\mathfrak{R}$.

According to $P'' = P'$, the following holds:

For $1 \leqslant j \leqslant n$, $a_{\mathrm{RMAX}[j]+1} \dots a_j$ *is the longest suffix of* $a_1 \dots a_j$ *which is in* P'.

Then, let us see that RMAX[j] is implicitly computed when factorizing $a_1 \dots a_n$ or determining minimum suffixes.

The improvement of either Algorithm 2.2 or Algorithm 3.1 is as follows:

add "RMAX[j]: = k" *at the beginning of case* 1 *and case* 2 *statements, and, add in case* 3 *statement* "*if* $k = j - 1$ *then* RMAX[j]: = k".

The justification for such an improvement is as follows:

Let k_3', i' and j' be the respective values of k, i, j at the first time we enter the while loop with $k \neq 0$ in algorithm 3.1. Let k_2', i', and j' be the corresponding values for Algorithm 2.2. The values i' and j' are the same in Algorithm 2.2 as in 3.1, and $k_2' \leqslant k_3'$. The process is the same for the two algorithms until this point.

For $j = 2$ to $j' - 1$, we have that (2.1) holds. Therefore, for $j = 2$ to $j' - 1$, if $a_i = a_j$ or $a_i < a_j$, $a_1 \dots a_j \in P'$ and RMAX[j] = 0. This is the justification for "RMAX[j]: = k" in case 1 and case 2 statement, since $k = 0$ in these steps.

Then, we find $a_{i'} > a_{j'}$ and k takes the new value k_3' for Algorithm 3.1 or k_2' for Algorithm 2.2. Since the process at this step consists to consider now $a_{k'+1} \dots a_n$ instead of $a_1 \dots a_n$, with $k' = k_3$ or k_2, $k' \leqslant k_3$, the justification of correctness is as follows:

For $j' \leqslant j \leqslant n$, *the longest suffix of* $a_1 \dots a_j$ *which is in* P', *is a suffix of* $a_{k_3'+1} \dots a_j$.

Proof. At step j', $a_1 \dots a_{j'-1}$ has the form $(uav')^q u$ with $u, v' \in A^*$, $a \in A$, $q \geqslant 1$ and $uav' \in L$. And we have that $a = a_i$, and $a_{i'} > a_{j'}$.

Let sa be the minimum suffix of ua. According to Proposition 1.12, sa is the minimum suffix of $(uav')^q ua$. Let ta be a suffix of $(uav')^q ua$ longer than

sa. We have that $sa < ta$ and $sa \leqslant t$. Since $a' < a$ we have that $sa' < t$ and sa' is not a prefix of t. Therefore, $sa'h < ta'h$ whatever h, and $sa'h$ is not a prefix of $ta'h$. Thus $ta'h \mathfrak{R} sa'h$. Then, the maximum suffix of $(uav')^q ua'h$ with respect to the converse order $\mathfrak{R}$ is a suffix of $sa'h$.

According to the preceding analysis for minimum suffix. We have that $s = a_{M[i']+1} \dots a_{i'-1}$ and $sa' a_{j'+1} \dots a_n = a_{M[i']+j'-i'+1} \dots a_n$, that is, $a_{k'_3+1} \dots a_n$.

Therefore, for $j' \leqslant j \leqslant n$, the maximum suffix of $a_1 \dots a_j$ with respect to the converse order $\mathfrak{R}$ is a suffix of $a_{k'_3+1} \dots a_n$. □

Then, if $j' = k'_3 + 1$, we have $a_{k'_3+1} \dots a_j$, and $\mathrm{RMAX}[j'] = k'_3$. And the process continues with $j = k'_3 + 2$. Otherwise the process continues with $k = k'_3$ and $j = j'$. This is the justification for the case 3 statement.

The third application concerns least circular shift. The conjugacy class of a word w is defined by

$$\bar{w} = \{uv | u \in A^*, v \in A^* \text{ and } vu = w\}.$$

The least circular shift of w is the minimum in the conjugacy class of w. It is a classical property that the least circular shift of a word has the form

$$w = u^q \qquad \text{with } u \in L \text{ and } q \geqslant 1.$$

It is quite clear that each circular shift of a word w is a factor of ww. Therefore, to recognize it, we may search in ww a factor of length $|w|$ which is a power of a Lyndon word. Then, the least circular shift u^q of w appears when factorizing ww into Lyndon words.

PROPOSITION. *Let $a_1 \dots a_n$ be a word. When factorizing $a_1 \dots a_n a_1 \dots a_n$ into Lyndon words by either algorithm* 2.1 *or algorithm* 2.2, *the least circular shift is $a_{k+1} \dots a_{j-1}$ when for the first time we have that $j - 1 - k = n$ and $j - i$ divides n.*

Proof. For simplification of notations we assume that $a_1 \dots a_n$ is a primitive word (i.e., is not a power of another word). Then, the least circular shift $w' = vu$ of $a_1 \dots a_n = uv$, is a Lyndon word. We have that $a_1 \dots a_n a_1 \dots a_n = uw'v$. Since u is a suffix of w', the last factor when factorizing u is a suffix of the Lyndon word w'; thus it is greater than w'. Since v is a prefix of w', the first factor when factorizing u is less than w'. Therefore $\mathrm{FACT}(a_1 \dots a_n a_1 \dots a_n) = \mathrm{FACT}(u).w'.\mathrm{FACT}(v)$. Then w' is a factor when factorizing $a_1 \dots a_n a_1 \dots a_n$ and it has length u. Thus the first time we find $j - 1 - k = n$ and $j - i = n$ we have that $a_{k+1} \dots a_{j-1}$ is a Lyndon word of length n, that is w'. If $a_1 \dots a_n$ is a power of another word, its least circular shift has the form $w' = w_1^q$ with $w \in L$ and $q > 1$. Then, arguments similar to those above hold and lead to the condition $j - 1 - k$

$= n$ and $j - i$ divides n. Then $a_{k+1} \dots a_{j-1} = (a_{k+1} \dots a_{j-i})^q$, with $n = q(j - i)$, is the least circular shift of $a_1 \dots a_n$. □

REFERENCES

1. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, "The Design and Analysis of Computer Algorithm," pp. 329–334, Addison–Wesley, Reading, Mass., 1974.
2. K. S. BOOTH, Lexicographically least circular substring, *Inform. Process. Lett.* **10**, (4–5) (1980), 240–242.
3. K. T. CHEN, R. H. FOX, AND R. C. LYNDON, Free differential calculus, IV, *Ann. of Math.* **68** (1958), 81–95.
4. J. P. DUVAL, Algorithme de factorisation d'un mot en mots de Lyndon, in "Actes du premier colloque AFCET-SMF de Mathématiques appliquées," tome 2, pp. 15–26, 1978.
5. J. P. DUVAL, Mots de Lyndon et périodicité, *RAIRO Inform. Théor.* **14**, (2) (1980), 181–191.
6. D. E. KNUTH, J. H. MORRIS, AND V. R. PRATT, Fast pattern matching in strings, *SIAM J. Comput.* **6** (2) (1977), 323–350.
7. LOTHAIRE, "Combinatorics on Words," Addison–Wesley, Reading, Mass., 1982.
8. V. SILOACH, Fast canonization of circular strings, *J. Algorithms* **2** (1981), 107–121.