1989

Optimal Canonization of all Substrings of a String

A. Apostolico

M. Crochemore

Report Number:

89-903

OPTIMAL CANONIZATION OF ALL SUBSTRINGS OF A STRING

A. Apostolico
M. Crochemore

CSD-TR-903
August 1989
(Revised April 1990)

OPTIMAL CANONIZATION OF ALL SUBSTRINGS OF A STRING

A. Apostolico [1] *and M. Crochemore* [2]

Purdue University CS TR 903
(August 1989)
(Revised April 1990)

ABSTRACT

Any word can be decomposed uniquely into lexicographically nonincreasing factors each one of which is a *Lyndon word*. This paper addresses the relationship between the Lyndon decomposition of a word x and a canonical rotation of x, i.e., a rotation w of x that is lexicographically smallest among all rotations of x. The main combinatorial result is a characterization of the Lyndon factor of x with which w must start. As an application, faster on-line algorithms for finding the canonical rotation(s) of x are developed by nontrivial extension of known Lyndon factorization strategies. Unlike their predecessors, the new algorithms lend themselves to incremental variants that compute, in linear time, the canonical rotations of all prefixes of x. The fastest such variant represents the main algorithmic contribution of the paper. It performs within the same $3|x|$ character-comparisons bound as that of the fastest previous on-line algorithms for the canonization of a single string. This leads to the canonization of all substrings of a string in optimal quadratic time, within less than $3|x|^2$ character comparisons and using linear auxiliary space.

Key words and phrases: combinatorics on words, Lyndon factorization, pattern matching algorithms, failure functions, lexicographic order, canonization of circular strings.

1) Dept. of Computer Science, Purdue University, West Lafayette, 47907 Indiana (USA) - Dipartimento di Matematica Pura e Applicata, U. of L'Aquila (Italy). Work by this author was supported in part by the French and Italian Ministries of Education, by the British Research Council Grant SERC-E76797, by the NSF Grant CCR-8900305, by the Library of Medicine Grant NIH R01 LM05118, by the AFOSR Grant 90-0107, and by the NATO Grant CRG900293.

2) L.I.T.P., University of Paris 7, 2-Place Jussieu, 75251 Paris, France. Work by this author was supported in part by PRC 'Mathématiques et Informatique' and by NATO Grant CRG900293.

1. INTRODUCTION.

An important factorization of free monoids [Lo] was introduced in [CFL] by Chen, Fox and Lyndon for computing a basis of the free Lie algebras. According to this factorization (known as the *Lyndon* factorization), any word can be written in a unique way as a concatenation of lexicographically non increasing factors, with the additional property that each factor is lexicographically least among its circular shifts. Two efficient methods for producing the factorization of an input word x of n symbols were proposed in [Du]. (The reader is encouraged to familiarize from the start with the first one of these methods, which is reported at the beginning of Section 3.) Both methods work on-line, i.e., they parse the input string into its factors while scanning it from left to right, but their respective bounds in terms of numbers of character comparisons depend on the amount of auxiliary storage needed. Specifically, word x is decomposed in a number of character comparisons bounded by $2n$ with constant auxiliary space, or, alternatively, in $(3/2)n$ comparisons with $n/2$ auxiliary memory locations. This speed-up is obtained by incorporating in the algorithm the computation of a table that locally resembles the failure functions used in string searching algorithms (see, e.g., [AHU], ch. 9; [KMP]).

In different contexts, the problem was studied of computing, for a given string x, the circular shift of x that is lexicographically least among all such shifts. This problem and the related one of checking the equivalence of two circular strings find many applications, e.g., in computing the single function coarsest partition [PTB], in checking polygon similarity [AK], in isomorphism tests for special classes of graphs [BL], and in molecular sequence comparisons [KS]. An algorithm requiring $3n$ comparisons and auxiliary space linear in n was presented in [Bo]. This algorithm too represents an extension of the computation of the failure function for x, and the auxiliary space needed is precisely that used to allocate the values of such function. The algorithm is also on-line, so that it can start with the character comparisons while the input string x is being read. It is intriguing that Booth's canonization algorithm gains all the information needed for the Lyndon factorization of the input, but it does not need to use it. A canonization algorithm faster than Booth's was subsequently developed by Shiloach [Sh]. This algorithm is remarkable in at least two respects. First, it works within a number of character comparisons bounded by $n + d/2$, where d is the displacement of the smallest starting position of a least circular shift with respect to the first position of x. Second, it requires only constant auxiliary space. Shiloach's algorithm is more complex than the algorithm in [Bo], and it cannot operate on-line, since it can start with its comparisons only after having learned the length of the input string and having acquired the middle character of x.

Some natural questions are prompted by the fact that, by definition, a Lyndon word is the lexicographically least rotation of itself. Thus, it is natural to ask how much extra information is needed in order to determine the lexicographically least rotation of a word given the Lyndon factorization of that word. Answering this question is not easy. In fact, even the partial answers that we give in Section 3 require some nontrivial combinatorial properties such as those derived in Section 2. A related question is whether an on-line algorithm that acquires information by processing the input string from left to right could approach or even match the outstanding performance of the algorithm in [Sh]. Questions like this are usually appropriate in the realm of algorithmic design, since the efficiency of an algorithm depends sometimes critically on the global information which is available to that algorithm.

As pointed out in [Du], any algorithm computing the Lyndon factorization of x can be used to find the least circular shift of x. This is done by running that algorithm on the string xx and performing some constant-time extra checks. Thus, simple extensions of the on-line algorithms in [Du] yield the least circular shift of x in $4n$ or $3n$ character comparisons, depending on whether or not linear auxiliary space is allowed. This is not better than the bound of [Bo], but it suggests that with $3n$ comparisons one can accumulate more information than that needed to find a lexicographically least circular shift. In this paper, we study in depth the relation between the Lyndon factorization of a word and the lexicographically least circular shift(s) of that word. As mentioned, this study leads to establish several combinatorial properties, which are presented in Section 2. Based on the results of this section, we show in Section 3 that a simple extension of the algorithms in [Du] enables to find the least lexicographic rotations of a string x with at most f additional character comparisons, where $f = \min[d, n/2]$. As a by-product, we also get on-line algorithms that find the least lexicographic rotation of x in a total number of character comparisons bounded by $2n$ or $1.5n + f$, depending on whether constant or $n/2$ auxiliary memory locations are used, respectively. The first bound improves on the $3n$ comparisons of [Bo], but unlike the latter it does not use linear auxiliary space. Each bound is the smallest known in its category.

The algorithms of Section 3 lend themselves to *incremental* variants, that are presented in Section 4. We show there that, if linear auxiliary space is allowed, then the computation of the least rotations of all prefixes of a string can be carried out in overall linear time. Such a performance seems not achievable through any of the previously known canonization strategies. Moreover, we show that the least rotations of all prefixes of a string can be cumulatively computed within the same bounds ($3n$ character comparisons and linear auxiliary space) that are required of the previously fastest on-line canonization algorithm [Bo] in order to find the least rotation of just one string. Straightforward extensions of these developments lead then to an optimal O(n^2) algorithm for the canonization of all substrings of a string of n characters, while the adaptation of any of the previous canonization algorithms requires time O(n^3). Our fastest algorithm for this problem performs less than $3|x|^2$ character comparisons, thus achieving an amortized complexity of 3 character comparisons per substring, and it uses linear auxiliary space.

2. LYNDON WORDS AND LEAST ROTATIONS

Let Σ be a finite alphabet totally ordered by the relation <, and let Σ^+ (resp. Σ^*) be the free semigroup (resp. monoid) generated by Σ. The total order < is extended in its corresponding lexicographic order on Σ^+, as follows: for any pair of words $x, y \in \Sigma^+$, $x < y$ iff either $y \in x \Sigma^+$ or:

$$x = ras, y = rbt, \text{ with } a<b; \quad a,b \in \Sigma, \; r, s, t \in \Sigma^*.$$

Fact 1. For v not in $u\Sigma^*$, and for any $w, z \in \Sigma^*$, $u < v$ implies $uw < vz$.

Given a word $x = s_1s_2...s_n$ in Σ^+, the i-th rotation of x (i=1,2,...,n) is the word $w = s_is_{i+1}...s_ns_1s_2...s_{i-1}$. A *least lexicographic rotation LR(x)* of x is a rotation of x that is lexicographically smallest among all rotations of x. That is, for $u \in \Sigma^*$, $v \in \Sigma^+$ we have $LR(x) = vu$ if $x = uv$ and for any pair $u', v' \in \Sigma^*$, $x = u'v'$ implies $vu \leq v'u'$. Since all rotations of x have equal length, then for any two such rotations w and w', $w \neq w'$ implies that w and w' differ in at least one symbol. An *LR* uv of x is completely identified by its position $|u|$ in x. We call $|u|$ a *least starting position (LSP)*. In the following, we shall be concerned with finding the *LSP*'s of string x. The following observation is easy to check (cf. also [Sh]).

Fact 2. String x has q *LSP* 's if and only if x can be written as $x = v^q$ for some word $v \in \Sigma^+$.

A word $x \in \Sigma^+$ is a *Lyndon word* iff x is smaller than any of its nonempty suffixes. For instance, on the alphabet $\{a, b\}$, *aaab*, *abbb*, *aabab* and *aababaabb* are Lyndon words. By the definition of lexicographic order, one gets then immediately that if x is a Lyndon word, then no nonempty proper suffix of x can be also a prefix of x. A word with this property is called *border-*

free. A word x is said to be *primitive* if setting $x = w^k$ implies $k = 1$. An immediate consequence of the preceding statement is then that any Lyndon word is also a primitive word.

Lyndon Theorem. Any word $x \in \Sigma^+$ can be written in a unique way as nonincreasing product of Lyndon words: $x = l_1 l_2 ... l_k$, $l_1 \geq l_2 \geq \ldots \geq l_k$ Moreover, l_k is the lexicographically smallest suffix of x.

The sequence $(l_1, l_2, \ldots, l_k)$ of Lyndon words such that $x = l_1 l_2 ... l_k$ and $l_1 \geq l_2 \geq \ldots \geq l_k$ is called the Lyndon decomposition of x. The following properties motivate our interest in Lyndon words.

Lemma 1. Let m be an *LSP* for x. Then m is also the position in x of some factor in the Lyndon decomposition of x.

Proof. Assume the contrary, i.e., that an *LSP* of x coincides with some position m of x that falls within some l_i. Let v be the suffix of l_i starting at position m. By the definition of a Lyndon word and since v is a nonempty proper suffix of x, one has $l_i < v$. Moreover, v cannot be a prefix of l_i, since l_i is border-free. Thus, Fact 1 shows that v cannot be a prefix of *LR(x)* and this leads to a contradiction.•••

A consequence of Lemma 1 is that, if l is a Lyndon word, then $LR(l) = l$. In fact, Lyndon words can be defined alternatively as primitive words that coincide with their respective least lexicographic rotations (see, e.g., [Lo]).

Lemma 2. If $x = l^e$, with $e \geq 1$ and l a Lyndon word, then $LR(x) = x$, and there are precisely e *LSP*'s for x, namely: $0, |l|, 2|l|, \ldots, (e-1)|l|$.

Proof. A straightforward consequence of Fact 2 and Lemma 1.•••

From now on, we concentrate on the cases where the conditions of Lemma 2 are not met, i.e., we assume $x = l_1 .. l_k$ with $k \geq 2$ and $l_1 \neq l_k$.

We introduce the notions of *prev* and *rest* of a factor in the Lyndon decomposition of the word x. These notions are used in the next lemmas to characterize the least rotation(s) of x. Let l be a factor of the Lyndon decomposition of x. Let i and j be respectively the smallest and the largest integers such that $l_i = l_{i+1} = \ldots = l_{j-1} = l_j = l$. Then $prev(l) = l_1 \ldots l_{i-1}$ and $rest(l) = l_{j+1} \ldots l_k$. One gets then that, for any factor l of the Lyndon decomposition of x, $x = prev(l)l^e rest(l)$ where e (≥ 1) is the number of occurrences of l in the decomposition.

Lemma 3. Let l be a factor occurring e (≥ 1) times in the Lyndon decomposition of x. If $l = rest(l)prev(l)$ then $LR(x) = l^{e+1}$, and there are precisely $e+1$ LSP's for x , namely: $|prev(l)|$, $|prev(l)| + |l|$, $|prev(l)| + 2|l|, \ldots, |prev(l)| + e|l|$.

Proof. Since $l = rest(l)prev(l)$, then $LR(x)$, which is also $LR(l^e rest(l)prev(l))$, is equal to $LR(l^{e+1})$. Thus, Lemma 2 gives the conclusion.•••

As an example, let $x = babaabbabaabbabaab = (babaab)^3$. We have $l_1 = b$, $l_2 = ab$, $l_3 = l_4 = aabbab$, $l_5 = aab$, $rest(l_3) = aab$, $prev(l_3) = bab$ and $LR(x) = (aabbab)^3$.

Lemma 4. Let l be a factor occurring e (≥ 1) times in the Lyndon decomposition of x. If $l \neq rest(l)prev(l)$ then $LR(x) < l^c rest(l)prev(l)l^{e-c}$ for $0<c<e$.

Proof. First note that $rest(l)prev(l) \neq l^g$ for $g \geq 1$. This follows from the assumption $l \neq rest(l)prev(l)$ in case $g = 1$. When $g > 1$, setting $rest(l)prev(l) = l^g$ implies that either l is a prefix of $rest(l)$ or l is a suffix of $prev(l)$. But this contradicts the definitions of $rest(l)$ and $prev(l)$.

Let g (≥ 0) be the largest integer such that $rest(l)prev(l) = l^g w$. So, the word w is nonempty and l is not a prefix of it. We will now consider two cases according to whether w is prefix of l or not.

Assume that w is a prefix of l. Then $l = ww'$ for some nonempty word w'. Since w is nonempty and l is a Lyndon word, we get $l < w'$. Then $rest(l)prev(l)l = l^g wl < l^g ww' = l^{g+1}$. Fact 1 applies and gives $rest(l)prev(l)l^e < l^{g+1}l^{c-1}wl^{e-c} = l^c rest(l)prev(l)l^{e-c}$ for $0<c<e$. This achieves the proof of the first case.

Consider now the second case, when w is not a prefix of l. We then have $w < l$ or $l < w$ where in both cases no word is a prefix of the other so that Fact 1 applies. First, if $w < l$, we get $rest(l)prev(l) = l^g w < l^{g+1}$ which gives, by Fact 1, $rest(l)prev(l)l^e < l^{g+1}l^{c-1}wl^{e-c} = l^c rest(l)prev(l)l^{e-c}$ for $0<c<e$. Secondly, if $l < w$, we get $l^{g+1} < l^g w = rest(l)prev(l)$ which gives, by Fact 1, $l rest(l)prev(l) = l^{g+1}w < rest(l)prev(l)$. Thus $l^e rest(l)prev(l) < l^{e-1} rest(l)prev(l) \ldots < l^c rest(l)prev(l)$ for $0<c<e$. Applying again Fact 1 gives $l^e rest(l)prev(l) < l^c rest(l)prev(l)l^{e-c}$. This achieves the proof of the second case.

In both cases we get $LR(x) < l^c rest(l)prev(l)l^{e-c}$ for $0<c<e$ as claimed.•••

The next lemma gives a necessary condition in order for a Lyndon factor of x to be also a prefix of $LR(x)$.

Lemma 5. If $x = prev(l)l^e$ and $prev(l)$ is non-empty, then $LR(x)$ is of the form vl^eu with u, v in Σ^* and $prev(l) = uv$.

Proof. By definition, $prev(l)$ cannot be equal to l. The claim is then an immediate consequence of Lemma 4. •••

Lemma 6. Let l be a factor occurring e (≥ 1) times in the Lyndon decomposition of x. If $LR(x) = l^e rest(l)prev(l)$, then $rest(l)$ is a prefix of l.

Proof. Assume $rest(l)$ is not a prefix of l. Since l cannot be a prefix of $rest(l)$, then we can find u, v, $w \in \Sigma^*$ and a, $b \in \Sigma$, such that $l = ubv$ and $rest(l) = uaw$. By the Lyndon theorem we have $a < b$. Thus $LR(x) \leq rest(l)prev(l)l^e < l^e rest(l)prev(l)$, a contradiction with the hypothesis.•••

Lemma 7. Let l be a factor occurring e (≥ 1) times in the Lyndon decomposition of x. If $rest(l)$ is a prefix of l and l is a proper prefix of $rest(l)prev(l)$, then $LR(x)$ is of the form $vl^e rest(l)u$ with u, v in Σ^* and $prev(l) = uv$.

Proof. We know from Lemma 4 that the rotations of x of the form $l^c rest(l)prev(l)l^{e-c}$ with $0<c<e$ are greater than $LR(x)$. Thus, we only have to prove that no LSP falls at the beginning of $rest(l)$ or within $rest(l)$.

We also know from the proof of Lemma 4 that $rest(l)prev(l) \neq l^g$ for $g \geq 1$. So, if g (≥ 1) is the largest integer such that $rest(l)prev(l) = l^g u$, the word u is nonempty and none of its prefix is l.

Let $l_1 \ldots l_i \ldots l_j \ldots l_k$ be the Lyndon decomposition of x, with $l = l_i = \ldots = l_j$, $prev(l) = l_1 \ldots l_{i-1}$ and $rest(l) = l_{j+1} \ldots l_k$. Note that, since l is a proper prefix of $rest(l)prev(l)$ and l is strictly longer than $rest(l)$, then $prev(l)$ cannot be empty. Therefore, we have $i > 1$. Let $w' \in \Sigma^*$, $w \in \Sigma^+$ and p be such that $l^g = rest(l)l_1 \ldots l_{p-1}w'$ and $l_p = w'w$. We have $1 \leq p < i$ and then $l < l_p \leq w$. Moreover, by our choice of g, l is not a prefix of w. From $l < w$, we get $l^{g+1} < l^g w$, which, by using Fact 1 and arguments in the proof of Lemma 4, leads to $l^e rest(l)prev(l) < rest(l)prev(l)l^e$. Thus $LR(x) < rest(l)prev(l)l^e$.

Finally, we show that $LR(x)$ cannot be of the form $vprev(l)l^e u$ with $rest(l) = uv$ and u nonempty. In fact, in this situation, $vprev(l)l^e u$ starts by a nonempty proper suffix of l. Applying again Fact 1 to l and its suffix leads to $l^e rest(l)prev(l) < vprev(l)l^e u$ and thus to $LR(x) < vprev(l)l^e u$.•••

For example, let $x = babaabbaabbaab$. Then $l_1 = b$, $l_2 = ab$, $l_3 = l_4 = aabb$, $l_5 = aab$. With $l = l_4$ we have $prev(l) = bab$, $rest(l) = aab$ and $LR(x) = aabb\ aabb\ aab\ b\ ab$.

Lemma 8. Let l be a factor occurring e (≥ 1) times in the Lyndon decomposition of x. If $rest(l)$ is non-empty and $rest(l)prev(l)$ is a proper prefix of l, then $LR(x) < l^e rest(l)prev(l)$.

Proof. Let w be such that $l = rest(l)prev(l)w$. The word w is non-empty and $l < w$ with l not a prefix of w. Then $rest(l)prev(l)l < rest(l)prev(l)w$ and, by Fact 1, $rest(l)prev(l)l^e < rest(l)prev(l)wl^{e-1}rest(l)prev(l) = l^e rest(l)prev(l)$, whence the claim follows.•••

As an example, let $x = babaabbabbaab$. Then $l_1 = b$, $l_2 = ab$, $l_3 = aabbabb$ and $l_4 = aab$. We see that $LR(x) = aab\ b\ ab\ aabbabb$.

Lemma 9. Let l be a factor occurring e (≥ 1) times in the Lyndon decomposition of x. Assume that ub is a prefix of $prev(l)$ and $rest(l)ua$ is a prefix of l with u in Σ^*, a,b in Σ, and $a \neq b$. Then, if $a < b$, $LR(x)$ is of the form $vl^e rest(l)u$ with u, v in Σ^* and $prev(l) = uv$. If $a > b$, $LR(x) < l^e rest(l)prev(l)$.

Proof. When $b < a$, we have $rest(l)prev(l) < l$ which, by Fact 1, gives $rest(l)prev(l)l^e < l^e rest(l)prev(l)$. Thus $LR(x) < l^e rest(l)prev(l)$.

Assume now that $a < b$. Let r be any proper suffix of $rest(l)$. Let r' be such that $rest(l) = r'r$. For some word t, $ruat$ is a proper suffix of l and, then, $l < ruat < rub$. Thus, $l^e rest(l)prev(l) < rprev(l)l^e r'$. This inequality, together with Lemmas 1 and 4, yields the conclusion.•••

As an example, let $x = babaababaababaab$. Then $l_1 = b$, $l_2 = ab$, $l_3 = l_4 = aabab$, $l_5 = aab$, and $LR(x) = l_3^2 rest(l_3)prev(l_3) = aabab\ aabab\ aab\ b\ ab$. For $y = babaabbbaabbbaab$ we get $l_1 = b$, $l_2 = ab$, $l_3 = l_4 = aabbb$, $l_5 = aab$. Then $LR(y) = rest(l_3)prev(l_3)l_3^2 = aab\ b\ ab\ aabbb\ aabbb$.

Let l be one of the factors in the Lyndon decomposition of x. We say that l is a *special factor of* x if and only if $rest(l)$ is a prefix of l and, in addition, one of the following conditions is satisfied:

rest(l) is empty;

l is a prefix of *rest(l)prev(l)*; or

$l <$ *rest(l)prev(l)* but l is not a prefix of *rest(l)prev(l)*.

Observe that, for any word x, the Lyndon decomposition $l_1 l_2 \ldots l_k$ of x has at least one special factor, namely, l_k. The preceding lemmas support the following Theorem.

Theorem 1. Let $l_1 l_2 \ldots l_k$ be the Lyndon factorization of a non-empty word x. Let t be the smallest index such that l_t is a special factor of x. Then $LR(x)$ is $l_t \ldots l_k l_1 \ldots l_{t-1}$, and $|prev(l_t)|$ is an *LSP* for x.

Proof. We know from Lemma 1 and Fact 2 that $LR(x) = l_r \ldots l_k l_1 \ldots l_{r-1}$ for one or more values of r in $\{1, 2, \ldots, k\}$. Thus, we only need to show that r can be t.

The minimality of t implies that $prev(l_t) = l_1 \ldots l_{t-1}$. Since l_t is a special factor, then $rest(l_t)$ is a prefix of l_t. If both $rest(l_t)$ and $prev(l_t)$ are empty, the conclusion follows from Lemma 2. If $l_t = rest(l_t)prev(l_t)$, the conclusion follows from Lemma 3. If l_t satisfies one of the other conditions in the definition of a special factor, then Lemmas 5, 7 or 9 assert that $LR(x) = v l_t \ldots l_k u$ with $uv = prev(l_t)$. Thus, it remains to prove that, in this case, v is empty.

Applying again Lemma 1, v is of the form $l_r \ldots l_{t-1}$ with r in $(1, 2, \ldots, t)$ (if $r=t$, v is assumed to be empty). Suppose $r < t$. By definition, l_r is not special. This means that either $rest(l_r)$ is not a prefix of l_r or none of the three conditions above is met. If $rest(l_r)$ is not a prefix of l_r, Lemma 6 shows that $LR(x) < l_r \ldots l_k l_1 \ldots l_{r-1}$. In the other situations, Lemma 8 or Lemma 9 yield the same conclusion. Thus, v is empty and $LR(x) = l_t \ldots l_k l_1 \ldots l_{t-1}$. This also proves that $|prev(l_t)|$ is a minimal *LSP* for x .•••

As an example, let $x = caabaabbaabaacaabaabbaabaa$. The Lyndon decomposition of x is $l_1 = c$, $l_2 = aabaabbaabaac$, $l_3 = aabaabb$, $l_4 = aab$ and $l_5 = l_6 = a$. The factors l_2, l_3, l_4 and l_5 are special. We have $LR(x) = aabaabbaabaac\ aabaabb\ aab\ a\ a\ c$. In this example x is a square and has 2 *LSP*'s.

3. ALGORITHMS THAT USE CONSTANT AUXILIARY SPACE

In this Section, we restrict ourselves to a model of computation where only constant auxiliary space is available, and we use the combinatorics of the preceding Section to retrieve an *LSP* of x from its Lyndon decomposition, through a small number of extra character comparisons. As mentioned, the use of Lyndon decompositions in the search for *LSP*'s was first introduced in [Du], where the *LSP*'s are computed with constant auxiliary space in at most $4n$ character comparisons. The approach of this section leads to an algorithm that produces the *LSP*'s of x from scratch in $2n$ comparisons, i.e., within the same number of character comparisons needed to carry out the Lyndon decomposition. In the realm of on-line algorithms, this is faster than the previously known ones. We start by reporting below, for convenience of the reader, the first one of the two algorithms presented in [Du] for decomposing a string x in its Lyndon factors. Note that, in this original formulation of the algorithm, cases 1 and 2 implicitly assume "and $j \leq n$" as part of the condition.

```
Procedure L [Du]
            Input: A string $x = s_1 s_2 ... s_n$ of symbols over an alphabet $\Sigma$.
            Output: The sequence $FACT = (m[1], m[2], ...., m[k])$ such that
                $l_1 = s_1 s_2 ... s_{m[1]}$ ; $l_2 = s_{m[1]+1} ... s_{m[2]}$ ; ...; $l_k = s_{m[k-1]+1} ... s_n$
begin $FACT$ := the empty sequence; $m := 0$
            while $m < n$ do begin
                $i := m+1$; $j := m+2$;
                99: case "compare $s_i :: s_j$" of
                    1: $(s_i < s_j)$: $i := m+1$; $j := j+1$; goto 99
                    2: $(s_i = s_j)$: $i := i+1$; $j := j+1$; goto 99
                    3: $(s_i > s_j$ or $j = n+1)$: repeat $m := m + (j-i)$; append $m$ to $FACT$
                                                until $m \geq i$
                endcase
            endwhile
end
```

The structural simplicity of Procedure L rests on subtle combinatorial properties. We refer to [Du] for the details, and limit our discussion to the operation of the procedure on the example string $x = babaabbabaabbabaab$. The first time the **while** loop is entered, it immediately results in an instance of case 3. The procedure sets $l_1 = b$, and re-enters the **while** loop with $m = 1$. The second iteration compares s_2 with s_3 and s_4, in succession, which results in case 1 and 3, respectively. The procedure identifies $l_2 = ab$, and re-enters the **while** loop with $m = 3$. The third iteration lasts until the condition $j = n+1$ (end of the string) is met, since no intervening instance of case 3 stops it in between. Through the **repeat** cycle, the procedure sets $l_3 = l_4 = aabbab$. The final iteration finds finally $l_5 = aab$. The nontrivial invariant conditions exploited by the procedure are that, at the beginning of each iteration, the factorization of $s_1 s_2 ... s_m$ has been correctly computed and, moreover, such a factorization is a prefix of the factorization of x. Along these lines, it is possible to establish the following theorem.

Theorem 2 [Du]. Procedure L computes the Lyndon factorization of a word x of length n in O(n) time, with a number of character comparisons bounded by $2n$ and constant auxiliary space.

As mentioned, a faster variant of Procedure L is possible. Such a variant performs no more than $1.5n$ character comparisons, but it needs $n/2$ auxiliary storage locations. The reader is referred to [Du] for the details. Some rearrangements in the body of Procedure L lead to the code presented below. The procedure so modified will be called Procedure LR. As is easy to check, removal from Procedure LR of the statement identified with an asterisk leads to a code that is perfectly equivalent to that of the original procedure L. The role of statement (*) is that of recording in a list *SP?* all possible candidates for a leftmost *LSP* of x. By Theorem 1, such candidates coincide with the positions of prospective special factors, and thus they correspond to all values of m in correspondence with which, during execution of either L or LR, the index j reaches the value n+1. For later use, the recording of statement (*) is not limited to the value m. Rather, the value of the index i at the time of recording is also saved. Clearly, statement (*) does not increase the number of character comparisons of the procedure, nor does it affect its time complexity.

Once Procedure L is available, it is not difficult to devise a procedure that, given a string x and the queue *SP?*, detects the position m of the earliest special factor in the Lyndon decomposition of x. Theorem 1 ensures then that such an m is also an *LSP* for x. Our procedure is called LSP and is given below in a slightly redundant but self-explanatory form.

```
Procedure LR
begin FACT := SP? := the empty sequence; $m := 0$; $i := 1$; $j := 2$;
        while $m < n$ do begin
            case "compare $s_i :: s_j$" of
                1: ($s_i < s_j$ and $j \le n$): $i := m+1$; $j := j+1$;
                2: ($s_i = s_j$ and $j \le n$): $i := i+1$; $j := j+1$;
                3: ($s_i > s_j$ or $j = n+1$):
                    begin
                (*)     if ($j = n+1$) then append pair ($m$ , $i$ ) to SP?
                        repeat $m := m + (j-i)$; append $m$ to FACT
                        until $m \ge i$
                        $i := m+1$; $j := m+2$
                    end
            endcase
        endwhile
end
```

Procedure LSP

```
        Input: A string $x = s_1 s_2 \ldots s_n$ of symbols over an alphabet $\Sigma$; the queue $SP?$.
        Output: an $LSP$ of $x$.
begin
special := false;
while special = false do begin
        $(m,i)$ := next($SP?$); $r := m$;
        $p := n + 1 - i$;                       { $p$ is the period of $s_{m+1} \ldots s_n = l \ldots l\ rest(l)$ ; $p = |l|$ }
        repeat $r := r + p$ until $(r \geq i)$;      { at the outset, $r$ is the first position of $rest(l)$ }
        if $(r = n)$ then special := true;        { Lemmas 2 and 5, case $rest(l)$ empty }
          else
            begin
            $j := 1$;
            while $(i \leq r)$ and $(j \leq m)$ and $(x[i] = x[j])$ do
              begin $i := i + 1$; $j := j + 1$ endwhile;
            if $(i = r + 1)$ then special := true;
                                          { Lemmas 3 and 7, case $l$ prefix of $rest(l)prev(l)$ }
            else if $(j = m + 1)$ then                                  { $i \leq r$ }
                  special := false          { Lemma 8, case $rest(l)prev(l)$ prefix of $l$ }
              else if $(x[i] < x[j])$ then special := true;             { Lemma 9 }
                     else special := false;                             { Lemma 9 }
            end
        endwhile
        output ($LSP = m$)
end
```

We leave it for the interested reader to show that, with minor additions, Procedure LSP can be made to output also the length of the smallest period of x whenever x has more than one *LSP*. This information is sufficient for the subsequent task of generating all *LSP*'s of x. The correctness of LSP is readily established by simple inspection of its code and accompanying captions. From now on, we concentrate on the assessment of the time complexity of the procedure.

Lemma 10. Procedure LSP performs at most $d = LSP(x)$ character comparisons.

Proof. We prove the claim by induction on the iterations of the outermost **while** loop of LSP. The claim clearly holds if the condition $r = n$ (i.e., *rest(l)* is empty) is detected the first time that **while** is entered, since no character comparison is involved before that test. Assuming now $r < n$, this prompts the execution of the inner **while** loop, which performs at most m character comparisons. At this point, we distinguish two cases, as follows.

Case 1: The statements following the inner **while** result in setting variable *special* to the value true. Then LSP terminates with $LSP = m$, whence the claimed bound follows.

encaps: Variable *special* is set to false. Then $LSP > m$, and we can charge the character comparisons made so far to the first m positions of x. Let l be the Lyndon factor occurring at position m in x. Since m was a candidate in *SP?*, then $rest(l)$ is a prefix of l. Since Procedure LSP entered the inner **while** loop, then $|rest(l)| < |l|$. Let (m', i') be the next candidate in the queue *SP?*, and let l' be the corresponding Lyndon factor. Since l' is a prefix of $rest(l)$, then $|l'| \leq |rest(l)| < |l|$. Thus, prior to testing m', the number of character comparisons performed by the procedure does not exceed $m' - |l|$. By the structure of LSP, testing m' requires no more than $|l|$ comparisons, and there are enough characters of l to undertake the associated charges.

The above argument is easily iterated through the candidates in *SP?*, which establishes the claim. •••

Lemma 11. Procedure LSP performs less than $n/2$ character comparisons.

Proof. Let (m_k, i_k) be the k-th element in the queue *SP?*. Let $l_{(k)}$ be the Lyndon factor at position m_k. Let g_k be the length of $rest(l_{(k)})$ and h_k be the number of character comparisons performed by Procedure LSP in order to test (m_k, i_k).

We certainly have $g_1+h_1 \leq n/2$, since $rest(l_{(1)})$ is a prefix of $l_{(1)}$. Setting $x = w\ rest(l_{(1)})$, we observe, in addition, that the characters compared by the procedure fall pairwise within disjoint sets of positions of the word w.

For every other pair (m_k, i_k), one may note that Procedure LSP deals with Lyndon factors confined into the suffix of length g_{k-1} of x. This implies $g_k+h_k < g_{k-1}$. (In fact, one can see that the tighter inequality $2g_k+h_k \leq g_{k-1}$ holds for $k > 1$.)
Adding up all these inequalities for $k = 1, 2$, etc. leads to $\Sigma h_k \leq n/2$, which completes the proof.•••

As an example, let $x = abaabbaabaacaabaabbaabaaca$. Its Lyndon factorization is $(abaabbaabaac, aabaabbaabaac, a)$. Procedure LSP takes exactly $12=|x|/2-1=d$ character comparisons.

Lemma 12. Procedure LSP runs in $O(n)$ time and uses constant auxiliary space.

Proof. The bound on the additional space used is trivial. All the operations inside the outer **while** loop other than those involved in the inner **while** or **repeat** take constant time. Since the number of candidates in *SP?* is bounded by n, then the total cost of these operations is $O(n)$. By Lemma 10, the total cost of all the executions of the inner **while** loop is $O(d)$. Thus, we only need to examine the total cost charged by the executions of the **repeat**. Observe that each execution of the **repeat** of LSP can be put in one-to-one correspondence with a corresponding execution of the **repeat** cycle of either Procedure L or LR. The claim then follows from Theorem 2. •••

The following Theorem summarizes these results.

Theorem 3. Let $m_1, m_2, ..., m_t$ be the positions of x , in increasing order, of all factors in the Lyndon decomposition of x which admit of their respective rests as their prefix. Let d be the smallest *LSP* of x. Then the *LSP*'s of x can be found in at most $f = \min[d, n/2]$ character comparisons, $O(n)$ time and constant auxiliary space.

Theorems 1 and 2 yield an overall bound of $2n+f$ for the cascaded procedures LR and LSP. If we are interested only in the *LSP*'s of x, however, then the execution of LR can be stopped as soon as the first special factor is detected. It turns out that this policy has the effect of fully absorbing the character comparisons needed by LSP within the $2n$ bound of LR. To be more precise, let SPECIAL(m,i) be a function that tests whether m is an *LSP* for x. Function SPECIAL can be extracted trivially from the body of Procedure LSP. Let now LR' be the procedure obtained from LR by substituting the statement "**if** ($j = n+1$) **then** append pair (m , i) to *SP?* " with the statement "**if** ($j = n+1$) **and** SPECIAL(m,i) **then stop** {$LSP = m$}".

Theorem 4. Procedure LR' finds an *LSP* of input string x in at most $2n$ character comparisons, using constant auxiliary space.

Proof. Let m_1 be the first value of m which is handed by LR' to SPECIAL for testing. We prove first that, immediately prior to this test, the total number of character comparisons performed by the procedure is bounded by $n + m_1$. Immediately prior to this test, index j has reached the value $n+1$ for the first time. It is not difficult to check (or cf. [Du]) that the total number of character comparisons performed by LR' (or, equivalently, by L or LR) up to the moment that m_1 was added to the list *FACT* is bounded above by $2m_1$. Immediately after appending m_1 to *FACT*, Procedure LR' sets the index j to the value $m_1 + 2$. Since no Lyndon factor was added to *FACT* while j moved from $m_1 + 2$ to $n + 1$, then no instance of case 3 occurred during this time. Thus, while j moved from $m_1 + 2$ to $n + 1$, only cases 1 and 2 were handled by the procedure. Observe that each one of these cases involves precisely one character comparison and one unit advancement of j, and j is never backed up by the procedure. We charge each comparison to the position of x identified by the current value of j, so that each position of x in the range $[m + 2, n + 1]$ is now charged exactly once. In conclusion, the total number of comparisons performed by the procedure while j

moved from $m_1 + 2$ to $n+1$ is $(n + 1) - (m_1 + 2) + 1 = n - m_1$. This shows that the overall number of character comparisons performed by LR' up to the moment that index j reaches the value $n + 1$ for the first time is bounded by $n + m_1$.

Let now $l_{(1)}$ be the Lyndon factor at position m_1. Let g_1 be the length of $rest(l_{(1)})$ and h_1 be the number of character comparisons performed by Function SPECIAL in order to test m_1. Recall that, as a consequence of Theorem 1, $h_1 \leq |l_{(1)}| - g_1$. We may thus charge these h_1 comparisons to the last $|l_{(1)}| - h_1$ positions of $l_{(1)}$. By this, the positions of x occupied by the last $|l_{(1)}| - h_1$ characters of $l_{(1)}$ have been charged at most twice, i.e., once through the sweeping of j from $m_1 + 2$ to $n + 1$ and once while performing the h_1 comparisons of SPECIAL. If now the test of m_1 succeeds, this clearly proves the claim. If it does not, then this implies that $rest(l_{(1)})$ is not empty, and that, prior to resuming with any character comparisons, the procedure will append the position m_2 of $rest(l_{(1)})$ to *FACT*. This implies that the character comparisons will resume with $j = m_2 + 2$. Observe at this point that each position of $rest(l_{(1)})$ has been charged only once, but the same holds for the first g_1 positions of $l_{(1)}$. Letting those g_1 positions of $l_{(1)}$ undertake the charge of the corresponding positions of $rest(l_{(1)})$ leads again to the assertion that, immediately after m_2 has been added to *FACT*, the total number of character comparisons performed by the procedure is bounded by $2m_2$. Since $rest(l_{(1)})$ is a prefix of $l_{(1)}$, and no instance of case 3 occurred while j moved from $m + 2$ to $n+1$, then no instance of case 3 can occur while j moves from $m_2 + 2$ towards $n+1$. Hence, j will reach again $n+1$, which makes m_2 precisely the next candidate to be tested by SPECIAL. This enables to iterate the above argument, which leads to establish the claim. •••

4. USING LINEAR AUXILIARY SPACE

In this Section, we relax the constraint on the auxiliary space. Although our next algorithms will use a modest number of additional memory locations (from $n/2$ to n), such a resource seems crucial to their performance.

It is instructive to revisit the results of the previous section under the assumption that the second Lyndon factorization algorithm of [Du] is used in the place of Procedure L. That algorithm requires $n/2$ auxiliary locations, but its bound on the total number of character comparisons is $1.5n$. The bound implied by Theorem 3 becomes, correspondingly, $1.5n+f$. An alternate analysis, which we leave for an exercise, leads to $n + \min[n, 1.5d]$. Both bounds are not better than $2n$ in the worst case. This is partly due to the fact that resort to linear auxiliary space does not affect the charges (linear in n or in d) of Lemmas 10 and 11. It also seems to suggest that the computation of the *LSP*'s of all prefixes of the input string inherently requires quadratic time. It turns out that, with linear auxiliary storage, linear time suffices. The auxiliary space is needed to store a table similar to the *next* function [KMP] of x. The interested reader shall find that, if such a table was given at no expense in advance, then an algorithm for the *LSP*'s of x developed from the second factorization in [Du] would match the $n + d/2$ bound of [Sh]. Throughout most of the rest of this Section, we shall be concerned with the proof of the following Theorem.

Theorem 5. Given a string x *of* n symbols, the *LSP*'s of all prefixes of x can be produced in optimal O(n) time and linear space.

Theorem 5 is an easy consequence of the discussion and lemmas that follow. The basic criterion subtending the Theorem can be derived by purely combinatorial arguments. However, it is more convenient for us to reason in terms of the procedures of the previous Section, since the correctness of such procedures encapsulates the needed combinatorial properties in a succinct way.

Our technique is illustrated in terms of the constant space procedures of Section 3, but similar constructions hold for the variant that uses linear auxiliary space. To start with the description of this technique, we need to introduce the notion of a run.

With the execution of Procedure L (or equivalently, LR or LR') on some input string x, we associate a unique parse of the string 12...n of positions of x into consecutive *x-runs*, as follows. An x-run is identified by giving the ordered pair [*left*, *right*] of its *endpoints*. Let [$left_1$, $right_1$], [$left_2$, $right_2$], ..., [$left_d$, $right_d$] be the x-runs, with left endpoints in increasing order. Then $left_k$, $1 \leq k \leq d$, is the value that the variable i gets assigned through the opening line (i.e., $i := m+1$; $j := m+2$) of the k-th iteration of the **while** loop of Procedure L. We also have $right_d = n$ and $right_k = left_{k+1} - 1$ for $k < d$. Observe that the **while** loop is re-entered only following an instance of Case 3. An alternative definition of $right_k$ ($k = 1,2,...d$-1) is that $right_k$ is the value assigned to the variable m as the final result of the management the k-th instance of Case 3 during execution of Procedure L. If [*left*, *right*] is an x-run, then the *x-shadow* corresponding to that run is the set of positions of x in the interval [*left*, *reach*], where *reach* +1 is the largest value attained by variable j while variable i lies inside that run. While the collection of all x-runs defines a partition of the positions of x, the collection of all x-shadows represents a covering, since the shadows of two consecutive runs may possibly overlap.

As an example, the runs that the procedure produces on the string x = *caabaabbaabaacaabaabbaabaacaabaabbaabaa* are: [1, 1] (*a*), [2, 27] (*aabaabbaabaacaabaabbaabaac*), [28, 34] (*aabaabb*), [35, 37] (*aab*), [38,39] (*aa*). The corresponding shadows are, in succession: [1, 1], [2, 39], [28, 39], [35, 39], [38, 39].

Let now $x_p = s_1 s_2 \ldots s_p$ be the p-th prefix of x, $p = 1, 2, \ldots n$. The following facts are easy consequences of the structure and correctness of Procedure L.

FACT 3. If $[left_k, reach_k]$ is an x-shadow, then $[left_k, \min[p, reach_k]]$ is an x_p-shadow for any $p \geq left_k$.

FACT 4. Assume that, for some $k \leq d$ and $p > left_k$, x_p is given as input to Procedure L. Then the opening line of the k-th iteration of the **while** loop will set $i := left_k$ and $j := left_k+1$. Moreover, during the k-th iteration, variable j will move uniformly and in unit increments from $left_k$ to $1 + \min[p, reach_k]$. Finally, variable i will have values in $[left_k, \min[p, right_k]]$ only during the k-th iteration.

From the above Facts we get, in particular, that for any value of k, $left_k$ -1 is the position of a factor in the Lyndon decomposition of x_p for every $p \geq left_k$. With reference to some fixed x_p, let now $[left, reach]$ be some x-shadow for which $left \leq p \leq reach$, and let $[left, right]$ be the x-run starting at *left*. For every value of j in $[left + 1, reach]$, let $con(j)$ be the value of i such that $con(j)$ is in $[left, right]$ and $s_{con(j)}$ is compared with s_j by the procedure. This definition of $con(j)$ is unambiguous, because of Facts 3 and 4.

Lemma 13. Let $w = s_{left}\, s_{left+1} \ldots s_p$ Then one of the following cases holds. Case 1: $p = left$ or $s_{con\,(p)} < s_p$: then the Lyndon factor of x_p at position[1] $left$ - 1 is precisely w. Case 2: $s_{con\,(p)} = s_p$. Then setting $h = p\text{-}con(p)$ and $u = s_{left}\, s_{left+1} \ldots s_{left+h}$, we have that $w = (u)^k u'$ for some $k > 0$, u is a factor in the Lyndon decomposition of x and u' is a nonempty prefix of u.

Proof. It follows from Facts 3 and 4 that, letting Procedure L run on input x_p, would produce the x_p-shadow $[left, p]$. That either Case 1 or Case 2 above applies is a consequence of the fact that no instance of the Case 3 of the procedure may occur while the j variable scans the interval $[left+1, p]$. The claim descends then from the correctness of the procedure as applied to the input string x_p (cf. the possible actions taken by the procedure following the comparison of the claim). •••

[1]Recall that if $x = vwy$, then the *position* of w in x is $|v|$.

Let now *first*(*left*, p) be the minimum m such that $m + 1 \geq left$ and m is the position in x_p of a special factor in the Lyndon decomposition of x_p. We have *first*(*left*, *left*) = *left*-1, since [*left*, *left*] is an x_{left}-shadow and the single character s_{left} is a Lyndon word.

Lemma 14. If *first*(*left*, p) > *left* - 1, then *first*(*left*, p) = *first*(*left*, p-*con*(p)) +p-con(p).

Proof. We know from Lemma 13 that either $w = s_{left}\, s_{left+1} \ldots s_p$ is a Lyndon word and hence also the last factor in the Lyndon decomposition of x_p, or else the Lyndon decomposition of w has the form $(u)^k\, u'$ where u is a Lyndon word, $|u| = p - con(p)$ and $u' = rest(u)$ is a proper prefix of u. In the first case, w meets one of the conditions for being a special factor in the Lyndon decomposition of x_p, namely, that *rest*(w) is empty. Thus *first*(*left*, p) = *left* - 1, which contradicts the assumption *first*(*left*, p) > *left* -1. Thus, it must be that $s_{left}\, s_{left+1} \ldots s_p$ has the form $(u)^k\, u'$, with u a Lyndon word, $|u| = p - con(p)$ and $u' = rest(u)$ a nonempty prefix of u. Now u cannot be a special factor, otherwise we would have again *first*(*left*,p) = *left* -1. Thus $|u'| < |u|$, and $|u'|$ may or may not be a Lyndon word. We now discuss the two corresponding cases.

Assume first that u' is a Lyndon word. Then $(u)^k\, u'$ represents the last k+ 1 factors in the Lyndon decomposition of x_p. Since u is not a special factor and u' meets the condition: *rest*(u') empty, we have that *first*(*left*, p) = *left* -1 + $k|u|$. Now, Facts 3 and 4 ensure that *left* is the left endpoint of a run in the Lyndon factorization of the prefix $x_{(p-|u|)}$. Clearly, the last k factors in such a factorization are in the form $(u)^{k-1}u'$. By Theorem 1, the conditions for u to be a special factor depend only on the three words u, u' and *prev*(u). Therefore, if u is not a special factor in x_p, then u is not a special factor in $x_{(p-|u|)}$. Since u' is a Lyndon word, then *first*(*left*, p-*con*(p)) = *left* + (k-1)$|u|$ = *first*(*left*, p) - (p-*con*(p)). Thus, the claim holds in this case.

If u' is not a Lyndon word, then the Lyndon factorization of u' is in the form vgv' for some integer k and nonempty words v and v' with v' a prefix of v. We also have *first*(*left*, p) ≥ *left* -1 + $k|u|$, and *first*(*left*, p-*con*(p)) ≥ *left* -1 + (k-1)$|u|$. If now v' is a Lyndon word, then applying to v' the argument previously applied to u' yields the claim. Otherwise *first*(*left*, p) ≥ *left* -1 $k|u|$ + $g|v|$, and we can argue as above that *first*(*left*, p-*con*(p)) ≥ *left* -1 (k-1)$|u|$ + $g|v|$. Iteration of this argument yields the claim. •••

Our next ingredient for the linear-time canonization of all prefixes of x is represented by a precomputed table that enables to know, in constant time, the result of the lexicographic comparison between x and an arbitrary suffix of x. Clearly, such a table supports, in constant time, any necessary test between some *prev(l)* and the corresponding segment of l, without resort to the procedure LSP. We call such a table *compare*, and define it formally as follows. For every position i of x, we have that *compare*(i) = ">" iff $s_1 s_2 \ldots s_{n-i} > s_{i+1} \ldots s_n$, *compare*$(i)$ = "=" iff $s_1 s_2 \ldots s_{n-i} = s_{i+1} \ldots s_n$, and finally *compare*(i) = "<" iff $s_1 s_2 \ldots s_{n-i} < s_{i+1} \ldots s_n$. The precomputation of *compare* can be based on that of a table such as the function *next* of [KMP]. Recall that *next*(i) is defined as the largest j less than i such that $s_i \neq s_j$ and, moreover, $s_1 s_2 \ldots s_{j-1} = s_{i-j+1} \ldots s_{i-1}$. The construction of *next* that is given in [KMP] takes $2n$ character comparisons for a string of length n. It is not difficult to check, however, that if $s_1 = s_2$ then that bound becomes $1.5n$. Such an improved bound can also be achieved in the cases where $s_1 \neq s_2$: informally, the key to this improvement is the observation that, once it is known that $s_1 \neq s_2$, then during the consecutive alignments of the string with itself that are considered in the computation of *next* one does not need to compare s_1 until an occurrence of s_2 has been found. The computation of *compare* can be carried out within the same control structure of the computation of *next*, and within the same number of character comparisons. In fact, as soon as the procedure for the computation of *next* finds that *next*$(i) = j$, then we can decide the value of *compare*$(i-j)$ simply based on the result of the comparison between s_i and s_j. Observe that, since *compare* can take one of only three values, irrespective of n, then its space occupancy does not affect the bound of n on the auxiliary storage needed.

Proof of Theorem 5. Clearly, the position of the earliest special factor in the decomposition of x_p is the minimum value attained by *first* (k, p) over all x_p-shadows of the form $[k, p]$. The facts and lemmas of this Section show that we do not need to compute all such shadows explicitly, since each one of them is implicit in the structure of some corresponding x-shadow. Now, we can regard the application of Procedure L to input string x as a stream of consecutive managements of individual x-shadows. Besides its normal operation, the procedure can compute an n-location table *special*, initially filled with some integer larger than n. For every p in $[1, n]$, *special*(p) will report at the end an *LSP* for x_p. At the beginning, the procedure sets *special*$(1) = 0$. While j describes an x-shadow of left endpoint m+1, the procedure computes *first*$(m+1, j)$. As already seen, *first*$(m+1, m+1) = m$. By Lemma 14, for $j = p > m + 1$ we only need to test the condition *first*$(m+1, p) = m$. The table *compare* supports this test in constant time. The invariant condition is clearly that *first*$(m+1, p - \text{con}(p))$ is available at this point, since j moved uniformly from m+2 to p. Thus, the procedure can compute *first*$(m+1, p)$ in constant time (and actually without performing character comparisons). The procedure can now set *special(p)* to the minimum between *first*(m+1, p) and the old value of *special(p)*, and proceed to the next value of p. Since only constant time statements were added, this upgrade of Procedure L still takes linear time. •••

As noted, the upgraded procedure of Theorem 5 does not perform additional character comparisons. Combining the $2n$ upper bound of Procedure L with the $1.5n$ needed to compute *compare*, we get a total bound of $3.5n$ character comparisons for this upgrade. It is easy to show that the shadow covering of x would not change if we used the faster, $1.5n$ character comparisons Lyndon decomposition algorithm. This leads to a variant that computes the LSP's of all prefixes of x within the same $3n$ character-comparisons bound of the previously fastest on-line algorithms for the canonization of a single string. Straightforward iteration of either variant through all suffixes of x yields our final claim.

Theorem 5. Given a string x of length n, the *LSP*'s of all substrings of x can be produced in optimal $O(n^2)$ time and linear space.

Acknowledgment: We gratefully acknowledge the careful review and constructive remarks by one of the referees, which led to several improvements in the presentation of our ideas.

References.

[AHU] AHO, A.V., J.E. HOPCROFT AND J.D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., (1974).

[AT] AKL, A.G. AND G.T. TOUSSAINT, **An improved Algorithmic Check for Polygon Similarity**, *Information Processing Letters* 7, 127-128 (1978).

[BL] BOOTH, K.S. AND G.S. LUEKER, **Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms**, *Journal of Computer and System Sciences* **13**, 335-379 (1976).

[Bo] BOOTH, K.S., **Lexicographically Least Circular Substrings**, *Information Processing Letters* **10**, 240-242, (1980).

[CFL] CHEN, K.T., R.H. FOX AND R.C. LYNDON, **Free Differential Calculus**, IV, *Ann. of Math.* **68**, 81-95 (1958).

[Du] DUVAL, J.P., **Factorizing Words over an Ordered Alphabet**, *J. of Algorithms* **4**, 363-381 (1983).

[KMP] KNUTH, D.E., J.H. MORRIS AND V.R. PRATT, **Fast Pattern Matching in Strings,** *SIAM Journal on Computing* **6**, 322-350 (1977).

[KS] KRUSKAL, J.B. AND D. SANKOFF (eds.), *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, Mass., (1985).

[Lo] LOTHAIRE, M., *Combinatorics on Words*, Addison-Wesley, Reading, Mass., (1982).

[PTB] PAIGE, R., R. TARJAN AND R. BONIC, **A Linear Time Solution to the Single Function Coarsest Partition Problem,** *Theoretical Computer Science* **40**, 67-84 (1985).

[Sh] SHILOACH, Y., **Fast Canonization of Circular Strings,** *J. of Algorithms* **2**, 107-121, (1981).