Available at WWW.MATHEMATICSWEB.ORG POWERED BY SCIENCE @ DIRECT®

Journal of Algorithms 46 (2003) 21–26

Journal of Algorithms

www.elsevier.com/locate/jalgor

Generating Lyndon brackets. An addendum to: Fast algorithms to generate necklaces, unlabeled necklaces and irreducible polynomials over GF(2)

J. Sawada [a,*,1] and F. Ruskey [b,2]

[a] *Basser Department of Computer Science, Madsen Bldg F09, University of Sydney, NSW 2006, Australia*

[b] *Department of Computer Science, University of Victoria, Canada*

Received 3 April 2001

Abstract

It is well known that the Lyndon words of length n can be used to construct a basis for the nth homogeneous component of the free Lie algebra. We develop an algorithm that uses a dynamic programming table to efficiently generate the standard bracketing for all Lyndon words of length n, thus constructing a basis for the nth homogeneous component of the free Lie algebra. The algorithm runs in linear amortized time; i.e., $O(n)$ time per basis element. For a single Lyndon word, the table (and thus the standard bracketing) can be computed in time $O(n^2)$.
© 2003 Elsevier Science (USA). All rights reserved.

Keywords: Lyndon word; Free Lie algebra; Lyndon bracket; Basis; nth homogeneous component; Generate; Dynamic programming

* Corresponding author.
E-mail address: sawada@cs.usyd.edu.au (J. Sawada).

[1] Research supported by NSERC and partial support of Czech Grant GAČR 201/99/0242 and ITI under project LN-00A 056.

[2] Research supported by NSERC.

1. Introduction

In the paper “Fast algorithms for generating necklaces, unlabeled necklaces and irreducible polynomials over GF(2)” Cattell et al. [2] outline a new recursive framework for generating necklaces and Lyndon words. In this addendum, we use the framework to generate Lyndon brackets. This introduction will re-iterate some basic definitions; however, for a complete discussion including examples and a background, [2, Sections 1 and 2] should be consulted.

A *necklace* is the lexicographically smallest element in an equivalence class of strings under rotation. A *prenecklace* is a prefix of some necklace. A *Lyndon word* is an aperiodic necklace. The *standard factorization* of a Lyndon word w ($|w| > 1$), denoted $\sigma(w)$, is the pair of Lyndon words (l, m) such that $w = lm$ where m has maximal length and l is non-empty. According to Lothaire [4, Proposition 5.1.3] such a factorization exists and furthermore, if m is the proper right Lyndon factor of maximal length, then l is also a Lyndon word. Using this standard factorization, the Lyndon words can be recursively mapped into their standard bracketing using the following function:

$$\gamma(w) = \begin{cases} w & \text{if } |w| = 1, \\ [\gamma(l), \gamma(m)] & \text{otherwise, where } \sigma(w) = (l, m). \end{cases}$$

Let $\mathbf{L}_k(n)$ denote the set of k-ary Lyndon words of length n. If w is in $\mathbf{L}_k(n)$, then $\gamma(w)$ is called the *Lyndon bracket* of w. We define the length of $\gamma(w)$ to be n. As an example, the bracketings that occur when γ is applied to $\mathbf{L}_2(6)$ is illustrated in Fig. 1.

Lothaire [4] and Reutenauer [6] both demonstrate that the set of standard bracketings of all Lyndon words in $\mathbf{L}_k(n)$ is a basis for the nth homogeneous component of the free Lie algebra over an alphabet of size k. Munthe-Kaas and Owren [5] discuss applications in numerical algorithms which use computations in free Lie algebras. Using the Lyndon words, several finely homogeneous computations (the number of occurrences of each alphabet symbol is fixed) in free Lie algebras are discussed by Andary [1], including the computation of a (non-standard) left-bracketing. However, we know of no algorithms for producing the standard Lyndon bracketing. A naïve implementation of the definitions gives an algorithm with running time $O(n^3)$ per bracketing; we improve this to $O(n)$ per bracketing by integrating the computation of a dynamic programming bracketing table into a fast algorithm for generating Lyndon words. For a single Lyndon word, we show that the table (and thus the standard bracketing) can be computed in time $O(n^2)$.

```
[0,[0,[0,[0,[0,1]]]]]
[0,[0,[0,[[0,1],1]]]]
[0,[[0,[0,1]],[0,1]]]
[0,[0,[[[0,1],1],1]]]
[0,[[0,1],[[0,1],1]]]
[[0,[[0,1],1]],[0,1]]
[0,[[[[0,1],1],1],1]]
[[0,1],[[[0,1],1],1]]
[[[[[0,1],1],1],1],1]
```

Fig. 1. The Lyndon brackets of $\mathbf{L}_2(6)$.

The algorithm of this paper has been incorporated into the "Combinatorial Object Server" at `www.theory.cs.uvic.ca/~cos` under the "necklace" section.

2. Generating Lyndon brackets

In this section we develop an algorithm for generating the length n Lyndon brackets over an alphabet of size k (or equivalently, a basis for the nth homogeneous component of the free Lie algebra). The Lyndon words can be generated in constant amortized time (where the computation reflects the total amount of change to the data structures, and not the time required to print out the object) using Algorithm 2.1 from [2], but the problem of generating these words with their respective bracketing previously had no fast solution.

The fundamental problem of computing the Lyndon brackets is to find the standard factorization of an arbitrary Lyndon word. In [3], Duval describes a linear-time algorithm for factoring a word $w = \alpha_1\alpha_2\cdots\alpha_m$ into is unique Lyndon factors such that each α_i is a Lyndon word and $\alpha_1 \geqslant \alpha_2 \geqslant \cdots \geqslant \alpha_m$. Applying this algorithm, a word w is a Lyndon word if $m = 1$. However, because of the restrictions on the factoring, this algorithm does not help us further factor a Lyndon word. Thus, we must look at another approach for finding the standard factorization of a Lyndon word.

A naïve approach to finding the Lyndon bracket of $w \in \mathbf{L}_k(n)$ is to first test each proper right factor m (starting with maximal length) until a Lyndon word is found. Recall that if $w = lm$ where m is the longest proper right factor of w that is Lyndon, then the standard factorization of w is $\sigma(w) = (l, m)$. By repeating this process recursively for each of the two factors we arrive at the Lyndon bracket of w (where the stopping condition is when a Lyndon factor of length 1 is reached). Duval's algorithm can be used to test whether each proper right factor is a Lyndon word in linear time. In the worst case, this test will have to be performed for each proper right factor which implies that to find the initial standard factorization takes time $O(n^2)$. Since this must be done recursively for each right factor, the worst case running time to generate the Lyndon bracket for w is $O(n^3)$.

A significant improvement can be made to this naïve algorithm from the following observation. Given a Lyndon word $w = a_1a_2\cdots a_n$, if we know ahead of time what the standard factorization is for each Lyndon subword of w, then determining the Lyndon bracket of w can be done in linear time. If $a_i \ldots a_j$ is a subword of w where $i < j$ then we define $\mathit{split}(i, j)$ to be the starting position of its longest proper right Lyndon factor. Using this notation, the recursive function PrintBracket(ℓ, r) displayed in Fig. 2 will print out the Lyndon bracket of w. As an example the $\mathit{split}(i, j)$ values for the Lyndon word 001001011 are displayed in Fig. 3. In this figure the value i represents the row number and the value j represents the column number, where each value ranges from 1 to n. The entry $\mathit{split}(1, n)$ determines the starting point of the longest proper right Lyndon factor in the original Lyndon word. Thus the standard factorization of 001001011 is (001, 001011).

Observe that the value $\mathit{split}(i, j)$, where $i < j$, can be defined recursively by the following recurrence relation:

$$\mathit{split}(i, j) = \begin{cases} i + 1 & \text{if } a_{i+1}\cdots a_j \text{ is a Lyndon word,} \\ \mathit{split}(i + 1, j) & \text{otherwise.} \end{cases}$$

```
procedure PrintBracket ($\ell, r : \mathbb{N}$);
begin
    if $\ell = r$ then print($a_\ell$);
    else begin
        print(" [ ");
        PrintBracket($\ell$, $split(\ell, r) - 1$);
        print(" , ");
        PrintBracket($split(\ell, r)$, $r$);
        print(" ] ");
    end;
end;
```

Fig. 2. A function to print the brackets of a Lyndon word.

$$\begin{bmatrix} - & 2 & 2 & 4 & 5 & 4 & 7 & 4 & 4 \\ - & - & 3 & 4 & 5 & 4 & 7 & 4 & 4 \\ - & - & - & 4 & 5 & 4 & 7 & 4 & 4 \\ - & - & - & - & 5 & 5 & 7 & 7 & 5 \\ - & - & - & - & - & 6 & 7 & 7 & 7 \\ - & - & - & - & - & - & 7 & 7 & 7 \\ - & - & - & - & - & - & - & 8 & 9 \\ - & - & - & - & - & - & - & - & 9 \\ - & - & - & - & - & - & - & - & - \end{bmatrix}$$

Fig. 3. The values $split(i, j)$ for the Lyndon word 001001011.

Thus, the key to obtaining each value $split(i, j)$ is to determine all the Lyndon subwords embedded in the Lyndon word $w = a_1a_2 \cdots a_n$. This can be done by modifying the algorithm for generating Lyndon words outlined in Algorithm 2.1 of [2]. In this algorithm, the parameter p maintains the length of the longest Lyndon prefix of the prenecklace $a_1 \cdots a_{t-1}$. However, to find all the Lyndon subwords, we must maintain the length of the longest Lyndon prefix for all strings $a_i \cdots a_{t-1}$ where $1 \leqslant i < t$. By storing this value for each i in p_i, then effectively we can replace the parameter p with the global array of values $p_1 \cdots p_n$. Note that the value p_1 maintains the value of the old parameter p.

In our new algorithm, each value p_i is updated in a similar manner as the original parameter p. Thus when $a_t = a_{t-p_i}$, the value p_i remains unchanged. Similarly, if $a_t > a_{t-p_i}$ then $a_i \cdots a_t$ is a Lyndon word and hence p_i is updated to $t - i + 1$ (the length). The only exceptional case occurs when $a_t < a_{t-p_i}$. In this case the word $a_i \cdots a_t$ is not a prenecklace so we assign the value 0 to p_i. Note that any extension of this string will never be a prenecklace, and since $a_{t-p_i} = a_t$, the value of p_i will remain unchanged at 0.

Now using the values $p_1 \cdots p_t$, we can determine the values for $split(i, t)$ using the associated recurrence; the string $a_{i+1} \cdots a_t$ is a Lyndon word when $p_{i+1} = t - i$. The function LyndonBracket(t) shown in Fig. 4 is the result of applying these modifications to Algorithm 2.1 of [2]. Note that the condition $n = p_1$ is observed before we print the bracket. In the original algorithm, this test was hidden in the Printit(p) function. The initial call to the algorithm is LyndonBracket(1) and the values p_i are initialized to 1. Because the array p is maintained globally, we copy the original values of p to a local array q. We then restore the original values of p for each iteration of the outer loop.

```
procedure LyndonBracket ($t$ : $\mathbb{N}$);
local $i, j$ : $\mathbb{N}$; $q$ : array of $\mathbb{N}$;
begin
    if $t > n$ then begin
        if $n = p_1$ then begin
            PrintBracket($1, n$);
            newline;
        end;
    end;
    else begin
        $q := p$;
        for $j$ from $a_{t-p_1}$ to $k - 1$ do begin
            $a_t := j$;
            for $i$ from $1$ to $t - 1$ do begin
                if $a_t < a_{t-p_i}$ then $p_i := 0$;
                if $a_t > a_{t-p_i}$ then $p_i := t - i + 1$;
            end;
            for $i$ from $t - 1$ downto $1$ do begin
                if $p_{i+1} = t - i$ then $split(i, t) := i + 1$;
                else $split(i, t) := split(i + 1, t)$;
            end;
            LyndonBracket( $t + 1$ );
            $p := q$;
        end;
    end;
end;
```

Fig. 4. An algorithm for generating Lyndon brackets.

2.1. Analysis of the algorithm

Prior to each recursive call to LyndonBracket(t) a linear amount of work is done. This linear work is reflected in the two loops to update the values p_i and $split(i, t)$ along with the array copy of q to p. Since Algorithm 2.1 of [2] runs in constant amortized time, this means that the computation of the $split(i, j)$ values is done in linear time per Lyndon word generated. Now since the function PrintBracket(ℓ, r) also takes linear time, we obtain the following theorem.

Theorem 1. *The algorithm* LyndonBracket(t) *for generating all k-ary Lyndon brackets of length n runs in $O(n)$ amortized time.*

A small modification of this algorithm can be used to determine the Lyndon bracket for a given Lyndon word $w = a_1a_2 \cdots a_n$ in $O(n^2)$ time. This is done by removing the outer **for** loop, the assignment to a_t, and the array copies $p := q$ and $q := p$. Once this is done, n recursive calls are made, each taking linear time. This dynamic programming approach will yield an improvement by a factor of n over the naïve algorithm.

Acknowledgments

The authors thank Bob Miers for teaching us about free Lie algebras. We thank the referees for many helpful comments.

References

[1] P. Andary, Finely homogeneous computations in free Lie algebras, Discrete Math. Theor. Comput. Sci. 1 (1997) 101–114.

[2] K. Cattell, F. Ruskey, J. Sawada, M. Serra, C.R. Miers, Fast algorithms to generate necklaces, unlabeled necklaces and irreducible polynomials over GF(2), J. Algorithms 37 (2) (2000) 267–282.

[3] J.-P. Duval, Factorizing words over an ordered alphabet, J. Algorithms 4 (1983) 363–381.

[4] M. Lothaire, Combinatorics on Words, Cambridge University Press, 1983.

[5] H. Munthe-Kaas, B. Owren, Computations in a free Lie algebra, Philos. Trans. Roy. Soc. London Ser. A 357 (1999) 957–981.

[6] C. Reutenauer, Free Lie Algebras, Clarendon Press, Oxford, 1993.