Journal of Algorithms **37,** 267–282 (2000)
doi:10.1006/jagm.2000.1108, available online at http://www.idealibrary.com on IDEAL®

Fast Algorithms to Generate Necklaces, Unlabeled Necklaces, and Irreducible Polynomials over GF(2)[1]

Kevin Cattell

HP Research Labs, Santa Rosa, California
E-mail: kevin_cattell@hp.com

Frank Ruskey, Joe Sawada, and Micaela Serra

Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada
E-mail: fruskey@csr.uvic.ca; sawada@csr.uvic.ca, mserra@csr.uvic.ca

and

C. Robert Miers

Department of Mathematics and Statistics, University of Victoria, Victoria, British Columbia, Canada
E-mail: crmiers@math.uvic.ca

Received August 22, 1998

Many applications call for exhaustive lists of strings subject to various constraints, such as inequivalence under group actions. A k-ary necklace is an equivalence class of k-ary strings under rotation (the cyclic group). A k-ary unlabeled necklace is an equivalence class of k-ary strings under rotation and permutation of alphabet symbols. We present new, fast, simple, recursive algorithms for generating (i.e., listing) all necklaces and binary unlabeled necklaces. These algorithms have optimal running times in the sense that their running times are proportional to the number of necklaces produced. The algorithm for generating necklaces can be used as the basis for efficiently generating many other equivalence classes of strings under rotation and has been applied to generating bracelets, fixed density necklaces, and chord diagrams. As another application, we describe the implementation of a fast algorithm for listing all degree n irreducible and primitive polynomials over GF(2). © 2000 Academic Press

[1] Research supported by NSERC.

1. INTRODUCTION

Four of the most natural group actions on strings over a fixed alphabet are: (a) leaving the string unchanged, (b) reversing the string, (c) rotating the string, and (d) permuting the symbols of the string by a permutation of the alphabet symbols. The four groups giving rise to these actions are (a) $\mathbb{Z}_1$, (b) $\mathbb{Z}_2$ acting on the indices (reversal), (c) the cyclic $\mathbb{C}_n$ acting on the indices, and (d) the symmetric group $\mathbb{S}_k$ acting on the alphabet, assuming the alphabet consists of k symbols.

Each group action, or composition of group actions, partitions the set of k-ary strings into equivalence classes, namely the orbits of the action. Many applications call for exhaustive lists of representatives of these equivalence classes. To generate such equivalence classes, it is natural to choose as representative the lexicographically smallest string. With this representation, efficient algorithms are known for generating the equivalence classes of each of the actions (a), (b), (c), and (d). By "efficient" we mean that the amount of computation used in generating the objects is proportional to the number of objects generated. Such algorithms are said to be CAT, for constant amortized time.

For (a) we are simply counting in base k which is known to be efficient for $k \geq 2$. For (b) efficient algorithms are easily developed. In case (c) the equivalence classes are usually called *necklaces*. Efficient algorithms for generating necklaces were developed by Fredricksen and Kessler [5] and Fredricksen and Maiorana [6]; these algorithms were proven to be efficient by Ruskey *et al* [20]. Closely related algorithms for generating Lyndon words (aperiodic necklaces) were developed by Duval [3] and shown to be efficient by Berstel and Pocchiola [1]. In case (d) the representative strings are usually called *restricted growth functions* and efficient algorithms for generating them have been developed by Er [4], Kaye [12], and others.

In addition to these four natural group actions considered in isolation, interesting equivalence classes also emerge by composing two or more of the group actions. For example, composing (b) and (c) results in the dihedral groups, with the resulting equivalence classes often called *bracelets*. Only recently (using the framework outlined in this paper) Sawada [23] developed an algorithm to generate k-ary bracelets in constant amortized time. Proskurowski *et al.* [17] show that the orbits of the composition of (b) and (d) can be generated in amortized $O(\sqrt{k})$ time, which is CAT if k is fixed. It remains an interesting challenge to develop efficient algorithms for the other compositions.

In this paper we present a new recursive framework for necklace generation. We then use this framework to develop a CAT algorithm to generate equivalence classes of the complemented cycling register (which are in one-to-one correspondence with vortex-free tournaments). We also use this framework to develop a CAT algorithm for generating unlabeled binary necklaces (and Lyndon words), i.e., the composition of (c) and (d) for $k = 2$. This algorithm was used in the implementation of a fast algorithm for listing all degree n irreducible polynomials over GF(2). The framework has also been used to efficiently generate other classes of strings as summarized below. Undoubtedly, other applications of the framework will also be discovered.

- Necklaces: Done in this paper, CAT algorithm.
- Vortex-free tournaments: Done in this paper, CAT algorithm.
- Binary unlabeled necklaces: Done in this paper, CAT algorithm.
- Irreducible polynomials over GF(2): Done in this paper.
- Fixed density necklaces: Done in Ruskey and Sawada [21], CAT algorithm.
- Bracelets: Done in Sawada [23], a CAT algorithm.
- Chord diagrams (i.e., $2n$ points on an oriented circle embedded in the plane joined pairwise by chords): to appear.
- Necklaces with forbidden substrings: Done in Ruskey and Sawada [22], CAT algorithm if the forbidden substring is a Lyndon word.
- A basis for the nth homogeneous component of the free Lie algebra: to appear.

The combinatorial objects which we are listing are fundamental and there is considerable interest in having efficient algorithms to generate them. This interest comes from mathematicians, computer scientists, electrical engineers, and scientists in other disciplines. Our algorithms have all been implemented and are used in the Combinatorial Object Server (COS) at http://www.theory.csc.uvic.ca/~cos in the sections on Necklaces and Irreducible Polynomials over GF(2).

1.1. *Background and Definitions*

Defining the alphabet $\Sigma_k \overset{\text{def}}{=} \{0, 1, \ldots, k-1\}$, the set Σ_k^n consists of all k-ary strings of length n. Define an equivalence relation $\sim$ on Σ_k^* by $\alpha \sim \beta$ if and only if there exist $u, v \in \Sigma_k^+$ such that $\alpha = uv$ and $\beta = vu$. The equivalence classes of $\sim$ are called *necklaces* and we identify each necklace with the lexicographically least representative in its equivalence class. The set of all necklaces of length n over a k-ary alphabet is denoted $\mathbf{N}_k(n)$.

$$\mathbf{N}_k(n) \overset{\text{def}}{=} \{\alpha \in \Sigma_k^n | \alpha \leq \beta \text{ for all } \alpha \sim \beta\} \qquad (1)$$

For example $\mathbf{N}_2(4) = \{0000, 0001, 0011, 0101, 0111, 1111\}$. The cardinality of $\mathbf{N}_k(n)$ is denoted $N_k(n)$. For a string $\alpha = a_1 a_2 \cdots a_n$, let $\sigma(\alpha)$ denote a left shift of α by one position: $\sigma(a_1 a_2 \cdots a_n) = a_2 \cdots a_n a_1$.

In an *unlabeled necklace* we do not care about which color a bead has, but only about whether two beads have different colors. Unlabeled necklaces 0001 and 0111 are identical since one can be transformed into the other by interchanging 0 and 1 at every position of the string. Formally, the set $\mathbf{N}_k(n)$ of all k-ary length n unlabeled necklaces is defined as follows:

$$\hat{\mathbf{N}}_k(n) \overset{\text{def}}{=} \left\{\alpha \in \mathbf{N}_k(n) \mid \alpha \le \sigma^i(\pi(\alpha)) \ \forall \pi \in \mathbb{S}_k \text{ and } 0 < i < n\right\}. \quad (2)$$

For example, $\hat{\mathbf{N}}_2(4) = \{0000, 0001, 0011, 0101\}$. As before,, we denote $\hat{N}_k(n) = |\hat{\mathbf{N}}_k(n)|$.

A string α is *periodic* if $\alpha = \beta^k$ where β is nonempty and $k > 0$. An aperiodic necklace is called a *Lyndon word*. The set of all k-ary Lyndon words of length n is denoted $\mathbf{L}_k(n)$.

$$\mathbf{L}_k(n) \overset{\text{def}}{=} \left\{\alpha \in \mathbf{N}_k(n) \mid \alpha \text{ is aperiodic}\right\}$$

For example, $\mathbf{L}_2(4) = \{0001, 0011, 0111\}$. The cardinality of $\mathbf{L}_k(n)$ is denoted $L_k(n)$.

An *unlabeled Lyndon word* is an aperiodic unlabeled necklace. The set of all k-ary length n unlabeled Lyndon words is denoted $\hat{\mathbf{L}}_k(n)$ with cardinality $\hat{L}_k(n)$. For example, $\hat{\mathbf{L}}_2(4) = \{0001, 0011\}$.

A word α is called a *prenecklace* if it is the prefix of some necklace. The set of all k-ary prenecklaces of length n is denoted $\mathbf{P}_k(n)$.

$$\mathbf{P}_k(n) \overset{\text{def}}{=} \left\{\alpha \in \Sigma_k^n \mid \alpha\beta \in \mathbf{N}_k(n+m) \text{ for some } m \ge 0 \text{ and } \beta \in \Sigma_k^m\right\}$$

For example $\mathbf{P}_2(4) = \mathbf{N}_2(4) \cup \{0010, 0110\}$. The cardinality of $\mathbf{P}_k(n)$ is denoted $P_k(n)$. Let $W_k(n)$ denote the number of prenecklaces of length at most n. These numbers will prove useful in analyzing the algorithms developed below. Formally,

$$W_k(n) \overset{\text{def}}{=} 1 + \sum_{i=1}^{n} P_k(i). \quad (3)$$

We denote unlabeled prenecklaces $\hat{\mathbf{P}}_k(n)$ with cardinality $\hat{P}_k(n)$.

$$\hat{\mathbf{P}}_k(n) \overset{\text{def}}{=} \left\{\alpha \in \Sigma_k^n \mid \alpha\beta \in \hat{\mathbf{N}}_k(n+m) \text{ for some } m \ge 0 \text{ and } \beta \in \Sigma_k^m\right\}$$

THEOREM 1.1. *The following formulae are valid for all $n \geq 1$, $k \geq 1$:*

$$L_k(n) = \frac{1}{n}\sum_{d|n} \mu(d) k^{n/d}, \quad \hat{L}_2(n) = \frac{1}{2n}\sum_{\text{odd } d|n} \mu(d) 2^{n/d}, \tag{4}$$

$$N_k(n) = \frac{1}{n}\sum_{d|n} \phi(d) k^{n/d}, \quad \hat{N}_2(n) = N_2(n) - \frac{1}{2n}\sum_{\text{odd } d|n} \phi(d) 2^{n/d}, \tag{5}$$

$$P_k(n) = \sum_{i=1}^{n} L_k(i), \quad \hat{P}_2(n) = \sum_{i=1}^{n} \hat{L}_2(i). \tag{6}$$

Proof. The equations for $L_k(n)$ and $N_k(n)$ are well known; a proof of the former may be found in Graham *et al.* [11, p. 141] and of the latter in Lothaire [16, p. 9]. The equations for $\hat{N}_2(n)$ and $\hat{L}_2(n)$ are from Gilbert and Riordan [8]. The equation for $P_k(n)$ follows from Lemma 2.3 and the equation for $\hat{P}_2(n)$ follows from Lemma 3.2. ▌

2. GENERATING NECKLACES

In this section we describe a recursive algorithm for generating necklaces and Lyndon words. Compared with previous algorithms, our contribution is a recursive formulation that leads to simpler proofs and analysis. It is also more amenable to modification (for example, existing algorithms only generate in lexicographic order, whereas ours can generate in many different orders). The algorithm is based on a result, Theorem 2.1, that tells us how to construct a length n prenecklace from a length $n-1$ prenecklace. In the next section, we will use this recursive methodology to generate all unlabeled necklaces. This methodology has already been used, in other papers, to generate necklaces with fixed density [21] and bracelets [23], both by CAT algorithms.

The following characterizations of Lyndon words and necklaces are well known (e.g., Lothaire [16, p. 64]) and are clearly equivalent to the definitions given earlier.

$$\alpha = xy \in \mathbf{L} \text{ if and only if } xy < yx, \text{ for all nonempty } x, y. \tag{7}$$

$$\alpha = xy \in \mathbf{N} \text{ if and only if } xy \leq yx, \text{ for all nonempty } x, y. \tag{8}$$

We now need to state a couple of technical lemmas. The following lemma is proven in [20, Lemma 1, p. 419].

LEMMA 2.1. *If $\alpha \in \mathbf{N}$, then $\alpha^t \in \mathbf{N}$ for $t \geq 1$.*

The lemma below follows directly from (8) and the definition of a prenecklace.

LEMMA 2.2. *Let $\alpha = a_1 \cdots a_n$ be a prenecklace. If x is a substring of α with length k, then $x \geq a_1 \cdots a_k$.*

The following lemma follows from Lemma 7.14 of [18] and the ensuing discussion.

LEMMA 2.3. *Let $\alpha = a_1 a_2 \cdots a_n$ be a string and $p = lyn(\alpha)$. Then $\alpha \in \mathbf{P}$ if and only if $a_{j-p} = a_j$ for $j = p+1, \dots, n$.*

For $\alpha \in \Sigma_k^*$, let $lyn(\alpha)$ be the length of the longest prefix of α that is a Lyndon word. This function is well defined since $\Sigma_k \subseteq \mathbf{L}$. More formally,

$$lyn(a_1 a_2 \cdots a_n) \overset{\text{def}}{=} \max\{1 \leq p \leq n \mid a_1 a_2 \cdots a_p \in \mathbf{L}_k(p)\}. \tag{9}$$

For example, $lyn(001010010) = 5$. Note that if $\alpha = a_1 a_2 \cdots a_n$ is a Lyndon word, then $lyn(\alpha) = n$.

The following theorem is very useful. We are tempted to call it the "fundamental theorem of necklaces"! It leads not only to the necklace generation algorithm, but also to algorithms for producing the Lyndon factorization of a word and for determining the necklace (lexicographically minimal rotation) of a string, both in time linear in the length of the string. The theorem specifies exact conditions, in terms of n and lyn, under which a character can be appended to a prenecklace and still remain a prenecklace. In many ways, the theorem is implicit in the work of Fredricksen [5, 6] and Duval [2, 3] but not explicitly stated there. Reutenauer [18] has a very similar statement on page 164.

THEOREM 2.1. *Let $\alpha = a_1 a_2 \cdots a_{n-1}$ be a string in $\mathbf{P}_k(n-1)$ and let $p = lyn(\alpha)$. The string αb in $\mathbf{P}_k(n)$ if and only if $a_{n-p} \leq b \leq k-1$. Furthermore,*

$$lyn(\alpha b) = \begin{cases} p & \text{if } b = a_{n-p} \\ n & \text{if } a_{n-p} < b \leq k-1. \end{cases}$$

ALGORITHM 2.1.

```
procedure gen($t, p$ : integer);
local $j$ : integer;
begin
      if $t > n$ then PrintIt($p$)
      else begin
            $a_t := a_{t-p}$; gen($t+1, p$);
            for $j \in \{a_{t-p}+1, \ldots, k-2, k-1\}$ do begin
                $a_t := j$; gen($t+1, t$);
            end;
        end
    end;
```

The algorithm gen(t, p) of Algorithm 2.1 follows directly from Theorem 2.1. The parameter p has the same meaning it had in the theorem, and the parameter t replaces n. In general, if $\alpha = a_1 \cdots a_{t-1}$ is a prenecklace with $lyn(\alpha) = p$, then a call to gen(t, p) will generate all length n prenecklaces with prefix α. To generate all length n prenecklaces assign $a_0 = 0$ and make the initial call gen$(1, 1)$. If the symbols selected in the for loop are selected in increasing order, then the listing is lexicographic; in opposite order the listing is in reverse lexicographic order. Necklaces and Lyndon words can be produced by adding a test to the function PrintIt(p) as described in Table I.

We now show that algorithm gen(t, p) is efficient; it has the CAT property.

THEOREM 2.2. *Algorithm* gen(t, p) *is a CAT algorithm.*

Proof. Call the number of nodes in the computation tree $W_k(n)$. From the structure of the algorithm, $W_k(n)$ is equal to the number of prenecklaces of length at most n, as expressed in (3).

From the expressions (4) and (5) we obtain the following bounds.

$$\frac{1}{n}\left(k^n - (n-1)k^{n/2}\right) \le L_k(n) \le \frac{k^n}{n} \le N_k(n) \le \frac{1}{n}\left(k^n + (n-1)k^{n/2}\right) \tag{10}$$

TABLE I
Different Objects Output by Different Versions of PrintIt

Sequence type	PrintIt(p)
Prenecklaces ($\mathbf{P}_k(n)$)	Println($a[1..n]$)
Lyndon words ($\mathbf{L}_k(n)$)	**if** $p = n$ **then** Println($a[1..n]$)
Necklaces ($\mathbf{N}_k(n)$)	**if** n mod $p = 0$ **then** Println($a[1..n]$)
De Bruijn sequence	**if** n mod $p = 0$ **then** Print($a[1..p]$)

We have from (6)

$$P_k(n) = \sum_{i=1}^{n} L_k(i) \le \sum_{i=1}^{n} \frac{1}{i} k^i. \tag{11}$$

Hence

$$W_k(n) - 1 = \sum_{j=1}^{n} P_k(i) \le \sum_{j=1}^{n} \sum_{i=1}^{j} \frac{1}{i} k^i.$$

Thus

$$\frac{W_k(n) - 1}{N_k(n)} \le \frac{n}{k^n} \sum_{j=1}^{n} \sum_{i=1}^{j} \frac{1}{i} k^i.$$

In [20] it is shown that this last expression converges to $(k/(k-1))^2$ as $n \to \infty$. Thus the asymptotic running time per necklace (or per Lyndon word, noting that $L_k(n)$ is $\Theta(N_k(n))$ by (10)) is constant; the necklace algorithm in Algorithm 2.1 is CAT no matter what type of object is being generated. ▮

2.1. *Application: Generating Equivalence Classes of the Complemented Cycling Register*

Golomb [10] considers the complemented cycling register (CCR) over the set of all binary strings of length n. The CCR transforms a given input string $\mathbf{b} = b_1 b_2 \cdots b_n$ into an output string $C(\mathbf{b})$ as shown below.

$$C(b_1 b_2 \cdots b_n) = b_2 \cdots b_n \overline{b}_1.$$

Repeated application of the operation C eventually results in the initial string; the sequence $\mathbf{b}, C(\mathbf{b}), C(C(\mathbf{b}))), \dots$ is called a *cycle*. If we let $CCR(n)$ denote the number of distinct cycles of the CCR taken over all binary strings of length n then

$$CCR(n) = \frac{1}{2n} \sum_{\text{odd } d|n} \phi(d) 2^{n/d}. \tag{12}$$

It is shown by Knuth [13] that the right side of (12) is equal to the number of vortex-free tournaments on n vertices. This number is also the same as the number of odd density binary necklaces with length n.

Now consider the mapping f that sends two consecutive members of a cycle to their exclusive-or. For example, $f(010101, 101011) = 111110$. Applied to other members of the cycle a circular rotation of 111110 results.

This is true in general, as we show below.

$$\mathbf{b} \oplus C(\mathbf{b}) = (b_1 \oplus b_2)(b_2 \oplus b_3) \cdots (b_{n-1} \oplus b_n)\big(b_n \oplus \bar{b}_1\big)$$

$$C(\mathbf{b}) \oplus C(C(\mathbf{b})) = (b_2 \oplus b_3) \cdots (b_{n-1} \oplus b_n)\big(b_n \oplus \bar{b}_1\big)\big(\bar{b}_1 \oplus \bar{b}_2\big)$$

Since $\bar{x} \oplus \bar{y} = x \oplus y$, these strings are shifts of one another. Furthermore, they must contain an odd number of 1's, since $b_1 \oplus b_2 \oplus b_2 \oplus b_3 \oplus \cdots \oplus b_{n-1} \oplus b_n \oplus b_n \oplus \bar{b}_1 = b_1 \oplus \bar{b}_1 = 1$. The process is reversible, so long as a value for b_1 is specified. Thus there is a natural correspondence between binary necklaces with an odd number of 1's and distinct cycles of the CCR on binary strings of length n. By making small modifications to gen(t, p) we can generate a unique representative from each distinct cycle of the CCR, for a given n, as described below.

First add an additional parameter d to gen(t, p) which keeps track of the number of 1's in the string. If the resulting string of length N has an even number of 1's then it is rejected. We also maintain another array $\mathbf{b}$ which holds a representative for a cycle of the CCR (not necessarily the lexicographically least representative). We use the equation $b_t = a_{t-1} \oplus b_{t-1}$ at each recursive call to find the next bit in the representative for the cycle; this adds only a constant amount of computation to each node of the computation tree.

The resulting algorithm is CAT since its running time is proportional to the running time for generating all odd density necklaces, since the number of odd density necklaces is asymptotically half the total number of necklaces.

3. GENERATING UNLABELLED BINARY NECKLACES

In this section we develop a CAT algorithm to generate all binary unlabeled necklaces. Recall that an unlabeled necklace is an equivalence class of necklaces under permutation of alphabet symbols. In the binary case, a necklace and its complement are in the same equivalence class. As representative we choose the lexicographically smallest string in the equivalence class, giving rise to the set $\hat{\mathbf{N}}_2(n)$.

In Section 1 we gave explicit expressions to count $\hat{N}_2(n)$, $\hat{L}_2(n)$, and $\hat{P}_2(n)$. The following two lemmas are analogous to Lemmas 2.1 and 2.3 for necklaces.

LEMMA 3.1. *If* $\alpha = a_1 \cdots a_n \in \hat{\mathbf{N}}$, *then* $\alpha^t \in \hat{\mathbf{N}}$ *for* $t \geq 1$.

Proof. Let $\beta = b_1 \cdots b_n$ be equivalent to α under permutation of its symbols. Then by definition of an unlabeled necklace, β must be greater than or equal to α and thus β^t must be greater than or equal to α^t. From Lemma 2.1 α^t is a necklace and therefore by definition α^t is an unlabeled necklace for $t \geq 1$. ▮

LEMMA 3.2. *Let $\alpha = a_1 \cdots a_n$ be a string and let p equal the length of the longest unlabeled Lyndon prefix of α. Then $\alpha \in \hat{\mathbf{P}}$ if and only if $a_{j-p} = a_j$ for $j = p + 1, \ldots, n$.*

Proof. If $a_{j-p} = a_j$ for $j = p + 1, \ldots, n$, then $\alpha = \beta^t\delta$ for some $t \geq 1$ where $\beta = a_1 \cdots a_p \in \hat{\mathbf{L}}$ and δ is a prefix of β. By Lemma 3.1, β^{t+1} is an unlabeled necklace and thus $\alpha \in \hat{\mathbf{P}}$. Conversely, assume that $\alpha \in \hat{\mathbf{P}}$. If $lyn(\alpha) = q$ and $p \neq q$ then there must exist a Lyndon prefix of α with length greater than p that is not in $\hat{\mathbf{L}}$. This implies that there exists a permutation π of the alphabet symbols such that $\pi(a_1, \ldots, a_q) < a_1 \cdots a_q$. This contradicts the assumption that α is an unlabeled prenecklace, since no matter what we append to the string α, it can never be an unlabeled necklace. Now since we must have $p = lyn(\alpha)$, by Lemma 2.3 $a_{j-p} = a_j$ for $j = p + 1, \ldots, n$. ▮

To generate unlabeled necklaces, we could simply generate all necklaces and perform a test for each necklace to determine whether or not it is the unlabeled representative. To perform this test, we take the complement of the generated necklace and use a necklace finding algorithm to find its corresponding necklace. Such an algorithm (which runs in time $O(n)$) can easily be derived from Duval's algorithm for factoring a string into Lyndon words [2] or from Theorem 2.1. We then compare the resulting necklace to the original; if the original is not larger, then it is an unlabeled necklace. This algorithm yields an overall running time of $O(n\mathbf{N}(n))$, which is far from efficient.

In order to improve upon this naïve algorithm to generate unlabeled necklaces we must improve upon the linear time test required at the end of the necklace generation. In the remainder of this section we build up to a theorem, Theorem 3.1, which suggests the addition of another parameter to the necklace algorithm gen(t, p). The constant time maintenance of this parameter at each node of the computation tree eliminates the need for the linear time test, thus yielding a CAT algorithm to generate unlabeled necklaces.

Before we state our main theorem, we first introduce some additional notation and state some useful lemmas. Note that $\hat{\mathbf{N}}_2(n)$ consists exactly of those necklaces α that satisfy $\alpha \leq \overline{\sigma^i(\alpha)}$ for all $0 < i < n$. Observe that

$$\text{if } |x| = |y|, \text{ then } x \leq y \text{ if and only if } \bar{x} \geq \bar{y}. \tag{13}$$

LEMMA 3.3. *Let* $\alpha = a_1 \cdots a_n \in \mathbf{N}_2(n)$. *If there is a* k *such that, for every* $0 < i \le k$, *we have* $\alpha \le \overline{\sigma^i(\alpha)}$ *and* $\overline{a_{k+1} \cdots a_n} = a_1 \cdots a_{n-k}$, *then* $\alpha \in \hat{\mathbf{N}}_2(n)$.

Proof. By the definition of an unlabeled necklace, a necklace is its unlabeled representative if and only if it is less than or equal to each of its complemented rotations. We are given that $\alpha \le \overline{\sigma^i(\alpha)}$ for $0 < i \le k$ so we need only show that $\alpha \le \overline{\sigma^i(\alpha)}$ for $k < i < n$.

Since $\overline{a_{k+1} \cdots a_n} = a_1 \cdots a_{n-k}$, taking $x = \overline{a_{i+1} \cdots a_n}$ in Lemma 2.2 yields either $\overline{a_{i+1} \cdots a_n} > a_1 \cdots a_{n-i}$ or $\overline{a_{i+1} \cdots a_n} = a_1 \cdots a_{n-i}$. In the former case the result is trivial. In the latter case we consider $\overline{a_{n-i+1} \cdots a_n}$ and look at two subcases. If $n - i + 1 \le k$, then $\overline{\sigma^{n-i+1}(\alpha)} \ge \alpha$ which implies $\overline{a_{n-i+1} \cdots a_n} \ge a_1 \cdots a_1$. If $n - i + 1 > k$, then $\overline{a_{n-i+1} \cdots a_n}$ is a substring of $\overline{a_{k+1} \cdots a_n}$ and is therefore a substring of the prenecklace $a_1 \cdots a_{n-i}$. By Lemma 2.2 we have $\overline{a_{n-i+1} \cdots a_n} \ge a_1 \cdots a_i$. Thus in both subcases $\overline{a_{n-i+1} \cdots a_n} \ge a_1 \cdots a_i$. Now using (13) we have $\overline{a_1 \cdots a_i} \ge a_{n-i+1} \cdots a_n$ which gives us the result $\overline{a_{i+1} \cdots a_n a_1 \cdots a_i} = \overline{\sigma^i(\alpha)} \ge \alpha$ for $k < i < n$. ▮

COROLLARY 3.1. *Let* $\alpha = a_1 \cdots a_n \in \mathbf{N}_2(n)$. *If* $\overline{a_i \cdots a_n} \ge a_1 \cdots a_{n-i+1}$ *for all* $1 \le i \le n$ *then* α *is an unlabeled necklace.*

Proof. If $\overline{a_i \cdots a_n} > a_1 \cdots a_{n-i+1}$, then $\overline{a_i \cdots a_n a_1 \cdots a_{i-1}} > \alpha$. If there exists a smallest i such that $\overline{a_i \cdots a_n} = a_1 \cdots a_{n-i+1}$ then, by Lemma 3.3, α is an unlabeled necklace. Otherwise, α is an unlabeled necklace by definition. ▮

We need one final definition before presenting the main theorem of this section. Define $com(a_1 \cdots a_n)$ to be the smallest positive value c, for which

$$\overline{a_{c+1} \cdots a_n} = a_1 \cdots a_{n-c}, \tag{14}$$

or n if no such value of c exists. For example, $com(000111000111) = 3$, $com((01)^m)) = 1$, and $com(0^n) = n$; these last two examples represent extreme values for com. First, we give a lemma useful in proving the theorem.

LEMMA 3.4. *A binary string* $\alpha = a_1 \cdots a_n$ *is an unlabeled prenecklace if and only if* α *is a prenecklace and* $\overline{a_i \cdots a_n} \ge a_1 \cdots a_{n-i+1}$ *for all* $1 \le i \le n$.

Proof. Assume that α is an unlabeled prenecklace. Since $\hat{\mathbf{P}}(n)$ is a subset of $P(n)$ then α is a prenecklace. By definition of an unlabeled prenecklace there exists an unlabeled necklace γ such that $\gamma = \alpha\delta$ for some string δ. Thus by the definition of an unlabeled necklace $\overline{a_i \cdots a_n} \ge a_1 \cdots a_{n-i+1}$ for all $1 \le i \le n$.

To prove the converse we let $p = lyn(\alpha)$. If $n \bmod p = 0$ then α is a necklace. By Corollary 3.1, α is also an unlabeled necklace and thus by definition α is an unlabeled prenecklace. Otherwise, if $n \bmod p \neq 0$ then we construct a string $\beta = (a_1 \cdots a_p)^{\lceil n/p \rceil}$ of length m by extending α. By observing that $a_1 \cdots a_p$ is an unlabeled necklace (using the fact that $a_1 \cdots a_p$ is a necklace and Corollary 3.1) we get $a_1 \cdots a_p \leq \overline{a_i \cdots a_p a_1 \cdots a_{i-1}}$ for all $1 \leq i \leq p$. Thus by (13) we have $\overline{a_1 \cdots a_p} \geq a_i \cdots a_p a_1 \cdots a_{i-1}$. Therefore for $1 \leq i \leq p\lfloor n/p \rfloor$ we have $\overline{a_i \cdots a_m} \geq a_1 \cdots a_{m-i+1}$. Again since $a_1 \cdots a_p$ is an unlabeled necklace, $\overline{a_i \cdots a_p} \geq a_1 \cdots a_{p-i+1}$. Thus we have $\overline{a_i \cdots a_m} \geq a_1 \cdots a_{m-i+1}$ for $p\lfloor n/p \rfloor < i \leq m$. Now since β is a necklace Corollary 3.1 shows that β is an unlabeled necklace, and thus by definition α is an unlabeled prenecklace. ▌

THEOREM 3.1. *Let* $\alpha = a_1 a_2 \cdots a_{n-1} \in \hat{\mathbf{P}}_2(n-1)$ *and* $c = com(\alpha)$. *The string* $ab \in \hat{\mathbf{P}}_2(n)$ *if and only if* (i) $ab \in \mathbf{P}_2(n)$ *and* (ii) $a_{n-c} = 0$ *or* $b = \bar{a}_{n-c}$. *Furthemore*

$$com(\alpha b) = \begin{cases} n & \text{if } b = a_{n-c} = 0 \\ c & \text{if } b = \overline{a_{n-c}}. \end{cases}$$

Proof. Assume that $\alpha b \in \hat{\mathbf{P}}_2(n)$. By Lemma 3.4 we also have $\alpha b \in \mathbf{P}_2(n)$. If $a_{n-c} = b = 1$ then the string $a_1 \cdots a_{n-c} > \overline{a_{c+1} \cdots a_n}$, a contradiction to the definition of an unlabeled prenecklace. Therefore either a_{n-c} or $b = 0$. If $a_{n-c} = 1$ and $b = 0$ then $b = \overline{a_{n-c}}$. Thus either $a_{n-c} = 0$ or $b = \overline{a_{n-c}}$.

To prove the converse we need only show that $\overline{a_i \cdots a_n} \geq a_1 \cdots a_{n-i+1}$ for all $1 \leq i \leq n$ by Lemma 3.4. Because $\alpha \in \hat{\mathbf{P}}_2(n-1)$ and $c = com(\alpha)$ we observe that $\overline{a_i \cdots a_{n-1}} > a_1 \cdots a_{n-i}$ for all $1 \leq i \leq c$. Thus we clearly also have $\overline{a_i \cdots a_n} > a_1 \cdots a_{n-i+1}$ for $1 \leq i \leq c$. If $\overline{a_{n-c}} = b$ then $\overline{a_{c+1} \cdots a_n} = a_1 \cdots a_{n-c}$ by definition of $com(\alpha)$. If $a_{n-c} = b = 0$ then we have by a similar argument that $\overline{a_{c+1} \cdots a_n} > a_1 \cdots a_{n-c}$. Now applying Lemma 2.2 for either case we get $\overline{a_i \cdots a_n} \geq a_1 \cdots a_{n-i+1}$ for $c + 1 \leq i \leq n$. Therefore $\overline{a_i \cdots a_n} \geq a_1 \cdots a_{n-i+1}$ for all $1 \leq i \leq n$ and thus $\alpha b \in \hat{\mathbf{P}}_2(n)$.

Furthermore if $\overline{a_{n-c}} = b$, then we clearly have $com(\alpha b) = c$ and if $b = a_{n-c} = 0$, then by our convention $com(\alpha b) = n$ since there is no value of c for which (14) holds. Note that the case $a_{n-c} = b = 1$ cannot occur by the discussion in the first paragraph of the proof. ▌

This theorem implies that we can generate all unlabeled prenecklaces (and thus unlabeled necklaces) by introducing the additional parameter c to the prenecklace generation algorithm gen(t, p). Pseudocode for this algorithm is given in Algorithm 3.1. The initial call is genU$(2, 1, 1)$, first initializing $a_0 = a_1 = 0$. Unlabeled necklaces, Lyndon words, and prenecklaces can all be produced by using Table I as before. In general, if $\alpha = a_1 \cdots a_{t-1}$ is an unlabeled prenecklace with $lyn(\alpha) = p$ and $com(\alpha) = c$, then a call to genU(t, p, c) will generate all length n unlabeled prenecklaces with prefix α.

ALGORITHM 3.1.

```
Procedure genU ($t, p, c$: integer );
begin
    if $t > n$ then PrintIt($p$);
    else begin
        if $a_{t-c} = 0$ then begin
            if $a_{t-p} = 0$ then begin
                $a_t := 0$; genU($t + 1, p, t$);
            end;
            $a_t := 1$;
            if $a_{t-p} = 1$ then genU($t + 1, p, c$);
            else genU($t + 1, t, c$);
        end else begin
            $a_t := 0$; genU($t + 1, p, c$);
        end;
    end;
end;
```

Observe that the computation tree of genU(t, p, c) is a subtree of the computation tree of gen(t, p) and that only constant computation is performed at each node of the tree. Furthermore, the number of unlabeled binary necklaces is at least half the number of labeled binary necklaces. These observations prove the following theorem.

THEOREM 3.2. *Algorithm* genU(t, p, c) *is a CAT algorithm.*

It remains an interesting challenge to extend these ideas to generate unlabeled necklaces over nonbinary alphabets; there seems to be no obvious way to extend Theorem 3.1.

3.1. *Application: Generating All Irreducible Polynomials over GF*(2)

In the finite field GF(2), we denote by GF(2)$[x]$ the set of all polynomials over GF(2) in indeterminate x. A nonzero polynomial p is said to be *irreducible* over GF(2) if it has no nontrivial factorization $p = p_1 p_2$. An irreducible polynomial is also *primitive* if it has a root α such that α generates the nonzero elements of GF(2^n) (that is, $\{\alpha, \alpha^2, \alpha^3, \dots\} = \mathrm{GF}(2^n) \setminus \{0\}$). If p has one such root, then all of its roots are generators.

The number of degree n irreducible polynomials over GF(2) is $L_2(n)$, which is the same as the number of binary Lyndon words of length n. There is a natural correspondence between the two sets which we use to generate all irreducible and primitive polynomials of given degree. In the following paragraph we explain the correspondence.

Let q be a primitive polynomial of degree n and α one of its roots. For each $\lambda = \alpha^b$, where b ranges from 1 to $2^n - 1$, there is an equivalence class $\{\lambda, \lambda^2, \lambda^4, \dots, \lambda^{2^{m-1}}\}$ of size m, where $m \mid n$. These equivalence classes are in one-to-one correspondence with irreducible polynomials whose degree divides n by forming the polynomial $p(x) = (x + \lambda)(x + \lambda^2) \cdots (1 + \lambda^{2^{m-1}})$. This correspondence is explained in Golomb [9] (and see Lüneberg [15] or Reutenauer [18] for another correspondence via normal bases), but we have not before seen it exploited in an algorithmic context. The irreducible polynomials of degree n are those for which $m = n$.

Since successive powers of two amount to taking circular shifts of b, all irreducible polynomials of degree n may be generated by feeding in a primitive root α and generating all Lyndon words of length n. As each Lyndon word is generated it is converted into an integer b and then the corresponding irreducible polynomial $p(x)$ is computed. An irreducible polynomial is primitive if and only if b and $2^n - 1$ are relatively prime.

The *reciprocal* of a degree n polynomial $f(x)$ is the polynomial $x^n f(1/x)$. The reciprocal of an irreducible polynomial is irreducible, and the reciprocal of a primitive polynomial is primitive. Under the correspondence mentioned above reciprocal polynomials correspond to those generated by the complement of the number used to produce b. Thus we do not generate all Lyndon words, but only the unlabeled Lyndon words. This saves (asymptotically) a factor of two in the computation time.

Our algorithm was implemented in *C* with each polynomial stored in a single machine word in the obvious manner. The intermediate calculations involve polynomials over GF(2^n), which are stored as length n arrays of words. In one day on a middle-of-the-road workstation we can generate all 134,215,680 irreducible polynomials of degree 32, noting along the way the 67,108,864 that are primitive. Of course, for large values of n it becomes infeasible to generate all irreducible polynomials, but the algorithm can still be used to generate as many as are wanted. The program poly.c, available on COS, was compiled using gcc -04 and run on a Sun Microsystems Ultra 1 machine.

Theoretically, the computation of each irreducible polynomial takes $O(n^2)$ polynomial multiplications. Our multiplication routine is the naïve algorithm implemented on machine words. For larger n faster algorithms, such as those discussed in von zur Gathen [24], should be used. The algorithm (along with the other algorithms outlined in this paper) is well suited to parallelization using a system such as Brendan McKay's autoson, since there is a natural tree structure to the recursive algorithm which can be used to assign different subtrees to different processors.

We used the algorithm to compute various statistics of irreducible polynomials and these have led to several results and conjectures about the sizes of various subclasses of irreducible polynomials. For example, the number of irreducible polynomials with an odd number of nonzero odd terms is $\hat{L}_k(n)$. As another example, the number of irreducible polynomials with trace 1 (coefficient of x^{n-1}) is $\hat{L}_k(n)$.

ACKNOWLEDGMENTS

We thank the referees for their helpful comments. Algorithm 2.1 and Theorem 2.1 were discovered in 1993/94 by the second author as he was preparing [19]. It was one of the referees that pointed out that the essence of Theorem 2.1 is contained in Reutenauer [18].

REFERENCES

1. J. Berstel and M. Pocchiola, Average cost of Duval's algorithm for generating Lyndon words, *Theoret. Comput. Sci.* **132** (1994), 415–425.
2. J.-P. Duval, Factoring words over an ordered alphabet, *J. Algorithms* **4** (1983), 363–381.
3. J.-P. Duval, Génération d'une section des classes de conjugaison et arbre des mots de Lyndon de longueur bornée, *Theoret. Comput. Sci.* **60** (1988), 255–283.
4. M. C. Er, A fast algorithm for generating set partitions, *Comput. J.* **31** (1988), 283–284.
5. H. Fredricksen and I. J. Kessler, An algorithm for generating necklaces of beads in two colors, *Discrete Math.* **61** (1986), 181–188.
6. H. Fredricksen and I. J. Maiorana, Necklaces of beads in k colors and k-ary de Bruijn sequences, *Discrete Math.* **23** (1978), 207–210.
7. J. von zur Gathen and J. Gerhard, "Arithmetic and Factorization of Polynomials over F_2," Technical report tr-rsfb-96-018, University of Paderborn, Germany, 1996.
8. E. N. Gilbert and J. Riordan, Symmetry types of periodic sequences, *Illinois J. Math.* **5** (1961), 657–665.
9. S. W. Golomb, "Irreducible Polynomials, Synchronization Codes, Primitive Necklaces, and the Cyclotomic Algebra," Univ. of North Carolina Monograph Series in Probability and Statistics, Vol. 4, pp. 358–370, 1967.
10. S. W. Golomb, "Shift Register Sequences," Holden-Day, Oakland, 1967.
11. R. L. Graham, D. E. Knuth, and O. Patashnik, "Concrete Mathematics," Addison-Wesley, Reading, MA, 1989.
12. R. A. Kaye, A Gray code for set partitions, *Inform. Process. Lett.* **5** (1976), 171–173.
13. D. E. Knuth, "Axioms and Hulls," Lecture Notes in Compu ter Science, Vol. 606, Springer-Verlag, Berlin/New York, 1992.
14. R. Lidl and H. Niederreiter, "Introduction to Finite Fields and Their Applications," Cambridge Univ. Press, Cambridge, UK, 1994.

15. H. Lüneburg, "Tools and Fundamental Constructions of Combinatorial Mathematics," Wissenschafts, 1989.
16. M. Lothaire, "Combinatorics on Words," Cambridge Univ. Press, Cambridge, UK, 1983.
17. A. Proskurowski, F. Ruskey, and M. Smith, Analysis of algorithms for listing equivalence classes of k-ary strings induced by simple group actions, *SIAM J. Discrete Math.* **11** (1998), 94–109.
18. C. Reutenauer, "Free Lie Algebras," Clarendon Press, Oxford, 1993.
19. F. Ruskey, Combinatorial generation, manuscript, course notes for CSC 528, Univ. Victoria, 1995.
20. F. Ruskey, C. D. Savage, and T. Wang, Generating necklaces, *J. Algorithms* **13** (1992), 414–430.
21. F. Ruskey and J. Sawada, An efficient algorithm for generating necklaces with fixed density, *SIAM J. Comput.* **29** (1999), 671–684.
22. F. Ruskey and J. Sawada, Generating necklaces and strings with forbidden substrings, *in* "Proc. 6th Annual International Combinatorics and Computing Conference (CO-COON), Sydney, Australia, July 2000," Lecture Notes in Computer Science, to appear, Springer-Verlag, Berlin/New York.
23. J. Sawada, Generating bracelets in constant amortized time, manuscript, 1999.
24. J. von zur Gathen and J. Gerhard, Arithmetic and factorization of polynomials over F_2, *in* "Proc. ISSAC 96," pp. 1–9, Assoc. Comput. Mach, New York, 1996.