Generating Unlabeled Necklaces and Irreducible Polynomials over GF(2)

Kevin Cattell* Frank Ruskey† Joe Sawada‡
C. Robert Miers§ Micaela Serra¶

Department of Computer Science,
University of Victoria, Victoria, B.C., CANADA

Abstract

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. Generalization is made to the case where no substring 0^t occurs, for fixed t. All these algorithms have optimal running times in the sense that their running times are proportional to the number of necklaces produced. As an application, we describe the implementation of a fast algorithm for listing all degree n irreducible and primitive polynomials over GF(2).

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 group $\mathbb{C}_n$ 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

*Research supported by NSERC. e-mail: kcattell@csr.uvic.ca
†Research supported by NSERC. e-mail: fruskey@csr.uvic.ca
‡e-mail: jsawada@csr.uvic.ca
§Dept. Mathematics. Research supported by NSERC. e-mail: crmiers@math.uvic.ca
¶Research supported by NSERC. e-mail: mserra@csr.uvic.ca

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, Savage, and Wang [20]. Closely related algorithms for generating Lyndon words (aperiodic necklaces) were developed by Duval [3] and shown to be efficient by Berstel and Pocchiola [2]. 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 contrast to the case where our three non-trivial actions are considered in isolation, the composition of more than two of the actions gives rise to equivalence classes for which no efficient generation algorithms were previously known, unless k is fixed. For example, composing (b) and (c) results in the dihedral group, with the resulting equivalence classes often called *bracelets*. No more efficient algorithm is known than simply listing all necklaces and rejecting those that are larger than their reversals (i.e., are not the representative of a bracelet). Proskurowski, Ruskey, and Smith [18] 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.

The contributions of this paper are three fold:

1. The presentation of a new efficient recursive framework for necklace generation. A recursive CAT algorithm for generating all necklaces.

2. An $O(n)$ algorithm for recognizing binary unlabeled necklaces; i.e., the composition of (c) and (d) for $k = 2$. CAT algorithms for generating unlabeled binary necklaces and necklaces with no substring 0^t, all within the framework developed in item 1.

3. The development of the fastest known algorithm for generating all irreducible and/or primitive polynomials over GF(2). This algorithm has been used to suggest several enumerative results about irreducible polynomials; these results are proven elsewhere. This algorithm makes use of the unlabeled necklace algorithm to cut its computation time in half from what it would be if we were using the ordinary necklace algorithm.

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://sue.csc.uvic.ca/~cos` in the sections on generating necklaces and irreducible polynomials. To use COS the user chooses a type of combinatorial object and supplies input parameters and output options. COS then returns to the client over the WWW a list of all such objects. Implementations of our algorithms in C may also be downloaded from the COS site and thousands have done so. The pages on necklaces and irreducible polynomials are regularly visited.

1.1 Background and Definitions

Defining the alphabet $\Sigma_k = \{0, 1, \ldots, k-1\}$, 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 $\mathbf{N}_k(n)$.

$$\mathbf{N}_k(n) \stackrel{\text{def}}{=} \{\alpha \in \Sigma_k^n \mid \alpha \leq \beta \text{ for all } \alpha \sim \beta\} \tag{1}$$

For example $\mathbf{N}_2(4) = \{0000, 0001, 0011, 0101, 0111, 1111\}$. The cardinality of $\mathbf{N}_k(n)$ is denoted $N_k(n)$.

In an *unlabeled necklace* we don't care about which color a bead has, only about whether two beads have different colors. As 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 $\hat{\mathbf{N}}_k(n)$ of all k-ary length n unlabeled necklaces is defined as follows.

$$\hat{\mathbf{N}}_k(n) \stackrel{\text{def}}{=} \{\alpha \in \mathbf{N}_k(n) \mid \alpha \leq \pi(\alpha) \text{ for all } \pi \in \mathbb{S}_k\}. \tag{2}$$

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 non-empty and $k > 0$. An aperiodic necklace is called a *Lyndon word*. The set of all k-ary Lyndon words of length n is denoted $\mathbf{L}_k(n)$.

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

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 *pre-necklace* if it is the prefix of some necklace. The set of all k-ary pre-necklaces of length n is denoted $\mathbf{P}_k(n)$.

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

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 pre-necklaces of length at most n. These numbers will prove useful in analyzing the algorithms developed below. Formally,

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

We denote unlabeled pre-necklaces $\hat{\mathbf{P}}_k(n)$, with cardinality $\hat{P}_k(n)$.

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

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

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

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

PROOF: The equations for $L_k(n)$ and $N_k(n)$ are well-known; a proof of the former may be found in Graham, Knuth, Patashnik (pg. 141) [11] and of the latter in Lothaire (pg. 9) [16]. The equations for $\hat{N}_k(n)$ and $\hat{L}_k(n)$ are from Gilbert and Riordan [8]. The equation for $P_k(n)$ follows from Lemma 5 and the equation for $\hat{P}_k(n)$ follows from Lemma 7. □

In the case where $k = 2$ and the substring 00 is forbidden, formulae analagous to $N_2(n)$, $L_2(n)$ and $P_2(n)$ may be derived. Replace $2^{n/d}$ by $F_{n/d-1} + F_{n/d+1}$, where F_n is the n-th Fibonacci number.

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). In the next section, we will use the recursive methodology to generate all unlabeled necklaces and it is used in [21] to generate all necklaces of fixed density by a CAT algorithm. The algorithm is based on a new theorem, Theorem 2 below, that tells us how to construct a length n pre-necklace from a length $n-1$ pre-necklace.

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,

$$lyn(a_1 a_2 \cdots a_n) \stackrel{\text{def}}{=} \max\{1 \le p \le n \mid a_1 a_2 \cdots a_p \in \mathbf{L}_k(p)\}. \tag{7}$$

For example, $lyn(001010110010) = 8$. 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 (lex minimal rotation) of a string. The theorem specifies exact conditions, in terms of n and lyn, under which a character can be appended to a pre-necklace and still remain a pre-necklace.

THEOREM 2 *Let $\alpha = a_1 a_2 \cdots a_{n-1} \in \mathbf{P}_k(n-1)$ and $p = lyn(\alpha)$. The string $\alpha b \in \mathbf{P}_k(n)$ if and only if $a_{n-p} \le b \le k-1$. Furthermore,*

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

We will prove Theorem 2 later in this section after building up some preliminary results. The following characterizations of Lyndon words and necklaces are well-known (e.g., Lothaire [16], pg. 64) and are clearly equivalent to the definitions given earlier.

$$\alpha = xy \in \mathbf{L} \text{ if and only if } xy < yx, \text{ for all non-empty } x, y. \tag{8}$$

$$\alpha = xy \in \mathbf{N} \text{ if and only if } xy \leq yx, \text{ for all non-empty } x, y. \tag{9}$$

The following two lemmas are proven in [20] (Lemma 1, pg. 419; Lemma 2, pg. 420); Corollary 1 follows from Lemma 2.

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

Lemma 2 *If $\alpha b \in \mathbf{P}_k(n)$ is a pre-necklace, where $b < k-1$, then $\alpha(b+1) \in \mathbf{L}_k(n)$.*

Corollary 1 *If $\alpha b \in \mathbf{P}_k(n)$ is a pre-necklace, where $b < k-1$, then $\alpha j \in \mathbf{L}_k(n)$ for $b < j < k$.*

Lemma 3 *Let $\alpha = a_1 \cdots a_n$ be a pre-necklace. If x is a substring of α with length k, then $x \geq a_1 \cdots a_k$.*

Proof: Directly from the definitions of a necklace and a pre-necklace. □

Lemma 4 *If $\alpha \in \mathbf{L}$ and $\alpha = \beta\gamma$ for some β and γ with γ non-empty, then for any $t \geq 1$*

(a) $\alpha^t\beta \in \mathbf{P}$, and

(b) $\alpha^t\beta \in \mathbf{N}$ if and only if $|\beta| = 0$.

Proof: (a) Since α is a necklace, both α^t and α^{t+1} are necklaces by the Lemma 1. Thus $\alpha^t\beta$ is a pre-necklace and is a necklace if $\beta = \varepsilon$. If $\beta \neq \varepsilon$ then $\alpha = \beta\gamma < \gamma\beta$, since $\alpha \in \mathbf{L}$. Therefore,

$$\alpha^t\beta = (\beta\gamma)^t\beta = \beta(\gamma\beta)^t > \beta(\beta\gamma)^t = \beta\alpha^t,$$

so that $\alpha^t\beta$ is not a necklace. □

Lemma 5 *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 $j = p+1, \ldots, n$.*

Proof: If $a_{j-p} = a_j$ for $j = p+1, \ldots, n$, then $\alpha = \beta^t\delta$ for some $t \geq 1$ where $\beta = a_1 a_2 \cdots a_p \in \mathbf{L}$ and δ is a prefix of β. Thus $\alpha \in \mathbf{P}$ by Lemma 4.

Conversely, assume that $\alpha \in \mathbf{P}$, and let j be the smallest index $j > p$ such that $a_{j-p} \neq a_j$. By the definition of j, the string α has the form $\beta^t\delta a_j \cdots a_n$ for some $t \geq 1$ where $\beta = a_1 a_2 \cdots a_p \in \mathbf{L}$ and δ is a proper prefix of β. For some γ, $\beta = \delta\gamma$ where the first symbol of γ is a_{j-p}. By definition $\beta^t\delta$ is a pre-necklace since $\beta^t\delta\gamma = \beta^{t+1}$ is a necklace by Lemma 1. We will derive separate contradictions, depending whether $a_{j-p} < a_j$ or $a_{j-p} > a_j$.

If $a_j < a_{j-p}$, then $\delta a_j < a_1 \cdots a_i$ where $|\delta a_j| = i$. This statement contradicts Lemma 3 where $x = \delta a_j$.

```
procedure gen ( n, p : integer );
local j : integer;
begin
      if n > M then PrintIt( p )
      else begin
            a_n := a_{n-p};    gen( n+1, p );
            for j ∈ {a_{n-p}+1, ..., k-2, k-1} do begin
                  a_n := j;    gen( n+1, n );
            end;
      end;
end {of gen};
```

Figure 1: Recursive algorithm for generating necklaces (note: $a_0 = 0$.)

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

Table 1: Different objects output by different versions of PrintIt.

If $a_j > a_{j-p}$ then by Corollary 1 the string $\beta^t \delta a_j$ is a Lyndon word, contradicting the fact that $p = lyn(\alpha)$ is less than j. □

The formula (6) for $P_k(n)$ follows from Lemma 5. Now we can prove Theorem 2.

Proof of Theorem 2: Let $\alpha = a_1 a_2 \cdots a_{n-1} \in \mathbf{P}_k(n-1)$ and $p = lyn(\alpha)$. Then $\alpha = \beta^t \gamma$ where β is the Lyndon word $a_1 a_2 \cdots a_p$ and $t > 0$.

If $b = a_{n-p}$, then αb is a pre-necklace by Lemma 5. By the definition of lyn, it follows that either $lyn(\alpha b) = lyn(\alpha)$ or $lyn(\alpha b) = n$. If $lyn(\alpha b) = n$, then αb must be a Lyndon word. But αb cannot be a Lyndon word by (8), taking $x = \beta^t$ and $y = \gamma b$.

If $b > a_{n-p}$, then αb is a Lyndon word according to Lemma 2.

Finally, let $b < a_{n-p}$. If αb is a pre-necklace, then there is some string τ such that $\beta^t \gamma b \tau$ is a necklace. But then, taking $x = \beta^t$ and $y = \gamma b \tau$, we have $yx < xy$ in contradiction to (9). Thus αb is not a necklace if $b < a_{n-p}$. □

The algorithm gen(n, p) of Figure 1 follows directly from Theorem 2, and the parameters n and p have the same meanings they had in the theorem. We assume that $a_0 = 0$. The initial call is gen(1,1); it will generate all strings in $\mathbf{N}_k(M)$. 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. Various types of objects may be produced, depending on PrintIt(p), as shown in Table 1. A more complicated PrintIt call can be used to produce all "complemented cycling registers" as defined in Golomb [10], also in constant amortized time (see section 2.2 below).

We now show that algorithm gen is very efficient; it has the CAT property.

THEOREM 3 *Algorithm* gen *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 pre-necklaces of length at most n, as expressed in (3).

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

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

We have

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

The equality in (11) is valid because every pre-necklace is obtained as a prefix of β^*, where β is some Lyndon word. Hence

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

Thus

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

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) is constant; the necklace Algorithms 1 is CAT. □

2.1 Generating restricted classes of necklaces

The recursive form of algorithm gen makes it easy to add restrictions to the kinds of necklaces generated and still maintain the CAT property. We describe below two such restrictions.

To restrict the number of 0's in the string to lie under some bound z, simply add another parameter to gen that keeps track of the number of 0's used so far. So long as z is not too small, the resulting algorithm is CAT.

To restrict the necklaces to those that do not contain the substring 0^t, simply iterate, for $s = 1, 2, \ldots, t$, through the process of calling gen$(s+1, s)$, after initializing $a[1..s] = 0^{s-1}1$. The resulting algorithm is CAT by an argument that parallels the proof of Theorem 3. However, the asymptotic of [20] cannot be used, it must be extended to Fibonacci-like numbers.

2.2 Generating equivalence classes of the complemented cycling register

Golomb [10], considers the complemented cycling register over the set of all binary strings of length n. The complemented cycling register (CCR) transforms input as shown below.

$$b_1 b_2 \cdots b_n \to b_2 \cdots b_n \bar{b}_1$$

Repeated application of this operation eventually results in the initial string. 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.

As an example we show a cycle of the CCR for $n = 6$

$$\begin{array}{l} 010101 \rightarrow 101011 \rightarrow 010110 \rightarrow 101101 \rightarrow 011010 \rightarrow 110101 \rightarrow \\ 101010 \rightarrow 010100 \rightarrow 101001 \rightarrow 010010 \rightarrow 100101 \rightarrow 001010 \rightarrow \end{array}$$

The strings $000000, 100001, 001001, 011011$ also give rise to distinct cycles containing 12 elements and the size of the cycle containing 110011 is 4.

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

$$\begin{array}{l} (b_1 \oplus b_2)(b_2 \oplus b_3) \cdots (b_{n-1} \oplus b_n)(b_n \oplus \bar{b}_1) \\ (b_2 \oplus b_3) \cdots (b_{n-1} \oplus b_n)(b_n \oplus \bar{b}_1)(\bar{b}_1 \oplus \bar{b}_2) \end{array}$$

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. This implies that the 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. Thus, by making minor modifications to gen we can generate all distinct cycles of the CCR for given n.

First we add an additional parameter d to keep track of the number of 1's in the string. If the resulting string has an even number of 1's then it is rejected. We also maintain another array v which holds a representative for a cycle of the CCR (not necessarily the lexicographic representative). We use the following equation at each recursive call to find the next bit in the representative for the cycle

$$v_t = a_{t-1} \oplus v_{t-1}$$

The resulting algorithm is CAT since it is proportional to the running time for generating all odd density necklaces.

3 Generating Unlabeled Binary Necklaces

In this section we develop a CAT algorithm to generate all 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 $\hat{N}_k(n), \hat{L}_k(n)$ and $\hat{P}_k(n)$. The first two enumerations were given by Gilbert and Riordan [8]. The enumeration for $\hat{P}_k(n)$ follows from Lemma 7.

LEMMA 6 *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 1 α^t is a necklace and therefore by definition α^t is an unlabeled necklace for $t \geq 1$. □

LEMMA 7 *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 6, β^{t+1} is an unlabeled necklace and thus $\alpha \in \hat{\mathbf{P}}$.

Conversely, assume that $\alpha \in \hat{\mathbf{P}}$. If $p \neq lyn(\alpha)$ then there must exist a Lyndon prefix of α with length greater than p that is not in $\hat{\mathbf{L}}$. This contradicts the assumption that $\alpha \in \hat{\mathbf{P}}$. If $p = lyn(\alpha)$ then by Lemma 5 $a_{j-p} = a_j$ for $j = p+1, \ldots, n$. □

To generate unlabeled binary necklaces we need to add a check to ensure that the string we are currently generating is not greater than the complement of each of its rotations. To perform this check, we embed a one pass verification algorithm into the same recursive scheme used to generate necklaces.

A naïve method for testing whether a length n necklace is its unlabeled representative is to compare the necklace with all of its complemented rotations, taking worst-case time $\Theta(n^2)$. However, by proving a few properties of binary strings and necklaces we can perform this test with a one pass scan of the necklace in time $O(n)$.

LEMMA 8 *If* $x = x_1 \cdots x_k$ *and* $y = y_1 \cdots y_k$ *are two binary strings with equal length and* $x \leq y$*, then* $\overline{x} \geq \overline{y}$.

PROOF: If $x = y$ then $\overline{x} = \overline{y}$. If $x < y$ then there exists an i such that $x_i < y_i$. Let $\hat{i}$ be the smallest of such values. Thus $x_1 \cdots x_{\hat{i}-1} = y_1 \cdots y_{\hat{i}-1}$, $x_{\hat{i}} = 0$ and $y_{\hat{i}} = 1$. Now $\overline{x_1 \cdots x_{\hat{i}-1}} = \overline{y_1 \cdots y_{\hat{i}-1}}$ and $\overline{x_{\hat{i}}} = 1$ and $\overline{y_{\hat{i}}} = 0$. Therefore $\overline{x} \geq \overline{y}$. □

LEMMA 9 *If* x *and* y *are two strings with equal length and* $x < y$ *then for all strings* z, z' *such that* $|z| = |z'|$, $xz < yz'$.

LEMMA 10 *Let* $\alpha = a_1 \cdots a_n \in \mathbf{N}_2(n)$*. Let* $y = a_1 \cdots a_k$*. If* $a_{k+1} = 0$ *and* $y = xy'$ *where* $0 < |y'| = m$ *then* $y'1 > a_1 \cdots a_{m+1}$.

PROOF: Since $y0$ is a substring of α, by Lemma 3 , $y'0 \geq a_1 \cdots a_{m+1}$. Now since $y'1 > y'0$ we get the result $y'1 > a_1 \cdots a_{m+1}$. □

LEMMA 11 *Let* $\alpha = a_1 \cdots a_n \in \mathbf{N}_2(n)$ *and* $x = x_1 \cdots x_n$ *be a binary string. If* $x > \alpha$ *such that* $x_1 \cdots x_k = a_1 \cdots a_k$ *and* $x_{k+1} = 1$ *and* $a_{k+1} = 0$ *then* $x_i \cdots x_n x_1 \cdots x_{i-1} > \alpha$ *for all* $i \leq k+1$.

PROOF: Let $y = x_1 \cdots x_k = a_1 \cdots a_k$. Since $a_{k+1} = 0$ then by Lemma 10 all strings $x_i \cdots x_k 1 > a_1 \cdots a_{k-i+2}$ for $i \leq k$. Thus by Lemma 9 all strings $x_i \cdots x_n x_1 \cdots x_{i-1} > \alpha$ for $i \leq k$. For the string $x_{k+1} \cdots x_n x_1 \cdots x_k$ we know that $x_{k+1} = 1$. By the definition of a necklace a_1 must be 0 since $a_{k+1} = 0$. Therefore $x_{k+1} \cdots x_n x_1 \cdots x_k > \alpha$ by Lemma 9. □

```
function IsUnlabeled ( a_1 ··· a_M : necklace) : boolean;
local c, n : integer;
begin
      c := 0;
      for n := 1 to M do
            if ‾a_n > a_{n-c} then c := n;
            else if ‾a_n < a_{n-c} then return(false);
      return(true);
end;
```

Figure 2: $O(n)$ verification algorithm for unlabeled binary necklaces.

For binary string $\alpha = a_1 a_2 \cdots a_n$, let $\mathbf{Rot}_k(\alpha)$ denote the set of all complemented rotations $\overline{a_i \cdots a_n a_1 \cdots a_{i-1}}$ of α, where $1 \le i \le k$. Note that $\hat{\mathbf{N}}_2(n)$ consists exactly of those necklaces α that satisfy $r \ge \alpha$ for all $r \in \mathbf{Rot}_n(\alpha)$.

Theorem 4 *Let $\alpha = a_1 \cdots a_n \in \mathbf{N}_2(n)$. If there is a k such that, for every $r \in \mathbf{Rot}_k(\alpha)$, $r \ge \alpha$ 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. Thus we must show that $r \ge \alpha$ for all $r \in \mathbf{Rot}_n(\alpha)$. We are given that all $r \in \mathbf{Rot}_k(\alpha)$ satisfy this condition so we must only show that $\overline{a_{j+1} \cdots a_n a_1 \cdots a_j} \ge \alpha$ for $k \le j < n$.

Since $\overline{a_{k+1} \cdots a_n} = a_1 \cdots a_{n-k}$, taking $x = \overline{a_{j+1} \cdots a_n}$ in Lemma 3 yields either

Case #1: $\overline{a_{j+1} \cdots a_n} = a_1 \cdots a_{n-j}$, or
Case #2: $\overline{a_{j+1} \cdots a_n} > a_1 \cdots a_{n-j}$.

In Case #1 we start by showing that $\overline{a_{n-j+1} \cdots a_n} \ge a_1 \cdots a_j$ by considering two subcases: $n - j + 1 \le k$ and $n - j + 1 > k$.

If $n - j + 1 \le k$, then $r = a_{n-j+1} \cdots a_n a_1 \cdots a_{n-j} \in \mathbf{Rot}_k(\alpha)$. Thus $r \ge \alpha$ which implies $\overline{a_{n-j+1} \cdots a_n} \ge a_1 \cdots a_j$.

If $n - j + 1 > k$, then $\overline{a_{n-j+1} \cdots a_n}$ is a substring of $\overline{a_{j+1} \cdots a_n}$ and therefore is a substring of $a_1 \cdots a_{n-j}$. By Lemma 8 $\overline{a_{n-j+1} \cdots a_n} \ge a_1 \cdots a_j$.

Thus we have $\overline{a_{n-j+1} \cdots a_n} \ge a_1 \cdots a_j$. Now by Lemma 8 $\overline{a_1 \cdots a_j} \ge a_{n-j+1} \cdots a_n$ which gives us the result $\overline{a_{j+1} \cdots a_n a_1 \cdots a_j} \ge \alpha$.

In Case #2 we can apply Lemma 9.

Therefore, we have shown that $r \ge \alpha$ for all $r \in \mathbf{Rot}_n(\alpha)$ and thus $\alpha \in \hat{\mathbf{N}}_2(n)$. □

Corollary 2 *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, by Lemma 9, $\overline{a_i \cdots a_n a_1 \cdots a_{i-1}} > \alpha$. Now if there exists a smallest i such that $\overline{a_i \cdots a_n} = a_1 \cdots a_{n-i+1}$ then, by Theorem 4, 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 value c for which

$$\overline{a_{c+1} \cdots a_n} = a_1 \cdots a_{n-c}, \tag{13}$$

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 representing extreme values for com. First, we give a lemma useful in proving the theorem.

LEMMA 12 *A binary string $\alpha = a_1 \cdots a_n$ is an unlabeled pre-necklace if and only if α is a pre-necklace and $\overline{a_i \cdots a_n} \geq a_1 \cdots a_{n-i+1}$ for all $1 \leq i \leq n$.*

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

To prove the converse we let $p = lyn(\alpha)$. If $n \ mod \ p = 0$ then α is a necklace. By Corollary 2, α is also an unlabeled necklace and thus by definition α is an unlabeled pre-necklace. Otherwise if $n \ mod \ p \neq 0$ 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 2) 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 Lemma 8 we have $\overline{a_1 \cdots a_p} \geq a_i \cdots a_p a_1 \cdots a_{i-1}$. Therefore for $1 \leq i \leq \lfloor n/p \rfloor * p$ we have $\overline{a_i \cdots a_m} \geq a_1 \cdots a_{m-i+1}$. Again by noticing that $a_1 \cdots a_p$ is an unlabeled necklace we have that $\overline{a_i \cdots a_p} \geq a_1 \cdots a_{p-i+1}$. Thus we have $\overline{a_i \cdots a_m} \geq a_1 \cdots a_{m-i+1}$ for $\lfloor n/p \rfloor * p < i \leq m$. Now since β is a necklace we can use Corollary 2 to observe that β is an unlabeled necklace and thus by definition α is an unlabeled pre-necklace. □

THEOREM 5 *Let $\alpha = a_1 a_2 \cdots a_{n-1} \in \hat{\mathbf{P}}_2(n-1)$ and $c = com(\alpha)$. The string $\alpha b \in \hat{\mathbf{P}}_2(n)$ if and only if (i) $\alpha b \in \mathbf{P}_2(n)$ and (ii) $a_{n-c} = 0$ or $b = \bar{a}_{n-c}$. Furthermore*

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

PROOF: Assume that $\alpha b \in \hat{\mathbf{P}}_2(n)$. By the previous lemma $\alpha b \in \mathbf{P}_2(n)$. 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 pre-necklace. Therefore either $a_{n-c} = 0$ 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 must 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 the previous lemma. 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 by Lemma 9 $\overline{a_i \cdots a_n} > a_1 \cdots a_{n-i+1}$ 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 3 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 (13) holds. Note that the case $a_{n-c} = b = 1$ cannot occur by the discussion in the first paragraph of the proof. □

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

Figure 3: Recursive algorithm for generating unlabeled binary necklaces.

Given a necklace $\alpha = a_1 a_2 \cdots a_n \in \mathbf{N}_2(n)$, Theorem 5 suggests a simple one-pass algorithm to check whether $\alpha \in \hat{\mathbf{N}}_2(n)$, or not. We apply the theorem to each prefix of α, maintaining c along the way. Pseudo-code for our algorithm may be found in Figure 2.

To generate unlabeled necklaces, we modify the necklace generation algorithm by introducing the parameter c as described in the previous theorem. Psuedocode for this algorithm is given in Figure 3. The initial call is genU$(2, 1, 1)$ and we assume $a_0 = a_1 = 0$. Unlabeled necklaces, Lyndon words, and pre-necklaces can all be produced by using Table 1 as before.

Observe that the computation tree of genU is a subtree of the computation tree of gen 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 6 *Algorithm genU is a CAT algorithm.*

It remains an interesting challenge to extend these ideas to generate unlabeled necklaces over non-binary alphabets; there seems to be no obvious way to extend Theorem 5.

4 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 non-zero polynomial p is said to be *irreducible* over GF(2) if it has no non-trivial factorization $p = p_1 p_2$. Moreover, an irreducible polynomial is also *primitive* if it has a root α such that α generates the non-zero elements of GF(2^n) (that is, $\{\alpha, \alpha^2, \alpha^3, \ldots\} = \mathrm{GF}(2^n) \backslash \{0\}$). Moreover, if p has one such root, then all of its roots are generators. An equivalent definition of primitivity is that $x^k \bmod p \neq 1$ for all divisors k of $2^n - 1$ except $2^n - 1$ itself.

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. We explain the correspondence below.

Both primitive and irreducible polynomials have various applications, where the two most important are in coding theory and in linear finite state machines used in digital circuit testing applications. For the former, such polynomials are the generator of cyclic codes, and the particular cycle structure of the polynomial chosen gives different performances. For the latter, the characteristic polynomial of a matrix, which represents a linear operator, also determines the behaviour of the associated linear finite state machine. Primitive polynomials induce a *maximal length* cycle structure, that is, a linear finite state machine (or the code words from a cyclic code) will traverse the entire space of possible non-zero *states* (for the LFSM) or non-zero *codewords* (for cyclic codes) before returning to the initial starting point, in one single cycle. Irreducible polynomials induce a set of disjoint subcycles. Both these properties can be very useful in their domain of application [1, 14].

Let q be a primitive polynomial of degree n and α one of its roots. For each $1 \leq b \leq 2^n - 1$, and setting $\lambda = \alpha^b$, there is an equivalence class $\{\lambda, \lambda^2, \lambda^4, \ldots, \lambda^{2^{m-1}}\}$ 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 [19] 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 $m = n$.

Noting that since successive powers of two amount to taking circular shifts of b, all irreducible polynomials of degree n may be generated by generating all Lyndon words of length n. As each Lyndon word is generated it is converted into an integer b and then the corresponding polynomial $p(x)$ is computed.

For example, consider the length 5 aperiodic necklace 00101, so that $b = 5$. The equivalence class of rotations is $\{00101, 01010, 10100, 01001, 10010\}$. The elements of the class are the binary representations of the numbers $\{5, 10, 20, 9, 18\}$.

Let α be a root of the (arbitrarily chosen) primitive degree 5 polynomial $q(x) = x^5 + x^2 + 1$, and let $\lambda = \alpha^b = \alpha^5$. Then

$$\begin{aligned} p(x) &= (x+\lambda)(x+\lambda^2)(x+\lambda^4)(x+\lambda^8)(x+\lambda^{16}) \\ &= (x+\alpha^5)(x+\alpha^{10})(x+\alpha^{20})(x+\alpha^9)(x+\alpha^{18}) \\ &= (x+\alpha^2+1)(x+\alpha^4+1)(x+\alpha^3+\alpha^2)(x+\alpha^4+\alpha^3+\alpha)(x+\alpha+1) \\ &= x^5+x^4+x^2+x+1. \end{aligned}$$

Thus the irreducible polynomial corresponding to $b = 5$ is $x^5 + x^4 + x^2 + x + 1$. Note that p depends on the choice of the primitive polynomial q.

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

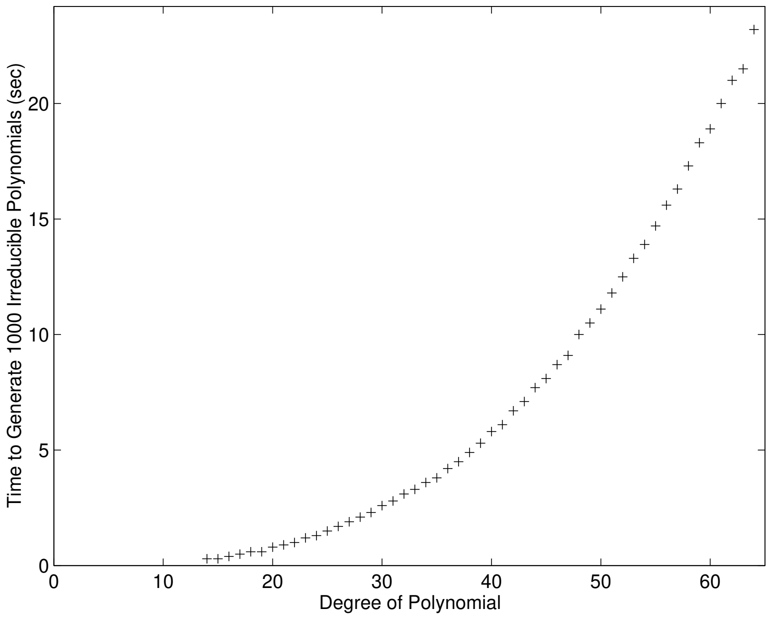

Figure 4: Time to Generate 1000 Irreducible Polynomials

Lyndon words, but only the unlabeled Lyndon words. This saves (asymptotically) a factor of two in the computation time.

To check whether an irreducible polynomial is primitive we simply compute $\gcd(b, 2^n - 1)$. An irreducible polynomial is primitive if and only if the greatest common divisor is one.

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. For each $n = 14, 15, \ldots, 64$, we have generated 1000 irreducible polynomials using our program. The running times in seconds are shown in Figure 4. The program `poly.c`, available on COS, was compiled using `gcc -O4` 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 [22], should be used. The algorithm 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 lead 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 non-zero odd terms is $\hat{L}_k(n)$. As another example, the number of irreducible polynomials with trace 1 (coefficient of x^{n-1}) is $\hat{L}_k(n)$ (Table 2 lists all degree 6 irreducible polynomials along with

Table 2: Degree 6 irreducible polynomials

Polynomial	Trace
x^6+x+1	0
x^6+x^3+1	0
$x^6+x^4+x^2+x+1$	0
$x^6+x^4+x^3+x+1$	0
x^6+x^5+1	1
$x^6+x^5+x^2+x+1$	1
$x^6+x^5+x^3+x^2+1$	1
$x^6+x^5+x^4+x+1$	1
$x^6+x^5+x^4+x^2+1$	1

their trace).

References

[1] P.H. Bardell, W.H. McAnney and J. Savir, *Built-In Test for VLSI - Pseudorandom Techniques*, John Wiley & Sons, 1987.

[2] J. Berstel and M. Pocchiola, Average cost of Duval's algorithm for generating Lyndon words, Theoretical Computer Science, 132 (1994) 415-425.

[3] J-P. Duval, Génération d'une section des classes de conjugaison et arbre des mots de Lyndon de longueur bornée, Theoretical Computer Science, 60 (1988) 255-283.

[4] M.C. Er, A Fast algorithm for generating set partitions, The Computer Journal, 31 (1988) 283-284.

[5] H. Fredricksen and I.J. Kessler, An algorithm for generating necklaces of beads in two colors, Discrete Mathematics, 61 (1986) 181-188.

[6] H. Fredricksen and I.J. Maiorana, Necklaces of beads in k colors and k-ary de Bruijn sequences, Discrete Mathematics, 23 (1978) 207-210.

[7] J. von zur Gathen and J. Gerhard, Arithemetic 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. Mathematics, 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, No. 4 (1967) 358-370.

[10] S.W. Golomb, *Shift Register Sequences*, Holden-Day, Inc., 1967.

[11] R.L. Graham, D.E. Knuth and O. Patashnik *Concrete Mathematics*, Addison-Wesley, 1989.

[12] R.A. Kaye, A Gray code for set partitions, Information Processing Letters, 5 (1976) 171-173.

[13] D.E. Knuth, *Axioms and Hulls*, Lecture Notes in Computer Science #606, Springer-Verlag, 1992.

[14] R. Lidl and H. Niederreiter, *Introduction to finite fields and their applications*, Cambridge University Press, 1994.

[15] H. Lüneburg, *Tools and Fundamental Constructions of Combinatorial Mathematics*, B.I. Wissenschaftsverlag, 1989.

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

[17] R.L. Miller, Necklaces, symmetries, and self-reciprocal polynomials, Discrete Math., 22 (1978) 25-33.

[18] 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 Mathematics, 11 (1998) 94-109.

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

[20] F. Ruskey, C.D. Savage, and T. Wang, Generating necklaces, J. Algorithms, 13 (1992) 414-430.

[21] J. Sawada and F. Ruskey, *An efficient algorithm for listing fixed density necklaces*, manuscript, 1998.

[22] J. von zur Gathen and J. Gerhard, Arithmetic and factorization of polynomials over F_2, Proc. ISSAC 96, ACM press (1996), 1-9.

A Cycles of the CCR Program

```
#include <stdio.h>

int a[50];
int v[50];
int n;

void Print(int p, int d) {

    int j;

    if ((n%p != 0) || (d%2 == 0)) return;

    for(j=1; j<=n; j++)
        printf("%d",v[j]);
    printf("\n");

}

void Gen(int t, int p, int d) {

    int j;

    if(t>n) Print(p,d);
    else {
        v[t] = (a[t-1] + v[t-1])%2;
        a[t] = a[t-p];
        if (a[t] == 0) {
            Gen(t+1,p,d);
            a[t] = 1;
            Gen(t+1,t,d+1);
        }
        else Gen(t+1,p,d+1);
    }
}

void main() {

    a[0] = 0;
    v[0] = 0;
    printf("enter n: ");
    scanf("%d", &n);
    Gen(1,1,0);
}
```