INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Fast Algorithms to Generate Restricted Classes of Strings Under Rotation

by

Joseph James Sawada
B.Sc., University of Victoria, 1996

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

Dr. Frank Ruskey, Supervisor (Department of Computer Science)

Dr. Micaela Serra, Departmental Member (Department of Computer Science)

Dr. Valerie King, Departmental Member (Department of Computer Science)

Dr. Bob Miers, Outside Member (Department of Mathematics)

Dr. Carla Savage, External Examiner (Department of Computer Science, North Carolina State University)

Supervisor: Dr. Frank Ruskey

ABSTRACT

A necklace is a representative of an equivalence class of k-ary strings under rotation. Efficient algorithms for generating (ie., listing) necklaces have been known for some time. Many applications, however, require a restricted class of necklaces for which no efficient generation algorithm previously existed. This dissertation addresses this problem by developing fast algorithms to generate the following restricted classes of necklaces: (a) unlabeled necklaces, (b) fixed density necklaces, (c) necklaces where the number of each alphabet symbol is fixed, (d) chord diagrams, (e) necklaces which avoid a particular Lyndon substring, and (f) bracelets. An analysis for each algorithm (a), (b), (e), and (f) shows that the amount of computation is proportional to the number of strings produced. Experimental results give a strong indication that the algorithms for (c) and (d) also achieve this time bound. In addition, a new derivation of the known formula for counting chord diagrams is presented, along with a linear time algorithm to generate a basis for the n-th homogeneous component of the free Lie algebra.

Examiners:

Dr. Frank Ruskey, Supervisor (Department of Computer Science)

Dr. Micaela Serra, Departmental Member (Department of Computer Science)

Dr. Valerie King, Departmental Member (Department of Computer Science)

Dr. Bob Miers, Outside Member (Department of Mathematics)

Dr. Carla Savage, External Examiner (Department of Computer Science, North Carolina State University)

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank Frank Ruskey for all that he has taught me, for his guidance, for his financial support, and for being a good guy. I cannot imagine a better student-supervisor relationship.

I would also like to thank my committee and external examiner, along with the entire computer science department (especially the office staff) here at the University of Victoria for their support.

My research has been supported in part by Natural Sciences and Engineering Research Council (NSERC).

For my parents

Chapter 1

Introduction

The rapid growth in the fields of combinatorial chemistry and computational biology is resulting in an increased demand for efficient algorithms which produce exhaustive lists of combinatorial objects [3]. Dan Gusfield (see [14], pg. xv) claims that "significant contributions to computational biology might be made by extending or adapting [string] algorithms from computer science, even when the original algorithm has no clear utility in biology." In particular, correspondences between DNA sequences and restricted classes of circular strings are described in [7].

Within the mathematical sciences, researchers are constantly trying to find patterns hidden in the structure of combinatorial objects. The growing trend of using computers and algorithms to produce lists of such objects is allowing researchers to obtain more information about the objects themselves. Often, this will lead to a more thorough understanding of an object which may lead to new and interesting discoveries. In some cases, algorithms which produce exhaustive lists can be used to prove the existence of a related object. For example, the fixed density necklace algorithm outlined in Chapter 4 is used to prove the existence of a (131,13) difference cover (see Chapter 4.5).

An important consideration for any algorithm is its running time. For generation algorithms, the ultimate performance goal is an algorithm with computation proportional to the number of objects generated (where the computation reflects the total amount of change to the data structures, and not the time required to print out the

object). Such algorithms are said to be CAT, for Constant Amortized Time.

Strings with equivalence under rotation are one of the most fundamental types of combinatorial object. Such objects, more commonly known as *necklaces*, arise naturally in many areas including knot theory, color printing, DNA sequencing and the theory of free Lie algebras. Algorithms for generating necklaces and Lyndon words (aperiodic necklaces) were first developed by Fredricksen and Kessler [10] and Fredricksen and Maiorana [11]. These algorithms were proven to be CAT by Ruskey, Savage and Wang [21]. A recursive version of these algorithms is outlined in [20].

Many applications do not require all necklaces, but instead only those satisfying a particular restriction. Previous to this dissertation, no efficient algorithm was known to generate any of the following restricted classes of necklaces:

- unlabeled necklaces,
- fixed density necklaces,
- necklaces where the number of each alphabet symbol is fixed,
- chord diagrams,
- necklaces with forbidden substrings, or
- bracelets.

A brief description of each object follows.

Unlabeled necklaces are necklaces with equivalence under permutation of the alphabet symbols. In the binary case, they have application in the generation of irreducible polynomials over GF(2) [6]. No algorithm has been previously published to generate unlabeled necklaces, even in the binary case.

Fixed density necklaces are necklaces where the number of non-zero characters, or the density, is fixed. Previous fixed density necklace algorithms have running times of $O(n \cdot N(n,d))$ (Wang and Savage [28]) and $O(N(n))$ (Fredricksen and Kessler [10]), where $N(n,d)$ denotes the number of necklaces with length n and density d and $N(n)$ denotes the number of necklaces with length n. Wang and Savage base their algorithm on finding a Hamilton cycle in a graph related to a tree of necklaces. The main feature of their algorithm is that it also generates the strings in Gray code order.

The basis of Fredricksen and Kessler's algorithm is a mapping of lexicographically ordered compositions to necklaces. Both algorithms consider only binary necklaces.

Another restricted class of necklaces are those where the number of each alphabet symbol is fixed. Such strings are prominent in cyclic arrangements [4], and also arise in the generation of polygons - where each alphabet symbol represents a direction: north, east, south, and west. In the binary case, these strings are equivalent to fixed density necklaces. In the general case, no fast generation algorithm was previously known.

Chord diagrams can be represented by length $2n$ unlabeled necklaces over an alphabet of size n, where there are exactly two occurrences of each alphabet symbol. Chord diagrams have application in the context of Vassiliev invariants, which in turn have application in knot theory [2]. A related object called a linearized chord diagram is studied by Stoimenow in [24], and braided chord diagrams are discussed by Birman and Trapp in [5]. Much attention has been placed on the enumeration of chord diagrams, but no generation algorithms have been previously published.

Necklaces with forbidden substrings are another restricted class of necklaces. If the forbidden substring is 0^r, then a simple modification to the recursive necklace algorithm will yield a fast algorithm to generate necklaces. For any other forbidden substring, no efficient algorithm was previously known.

The final restricted class of necklaces mentioned are bracelets. *Bracelets* are necklaces with equivalence under string reversal. They arise in several areas including color printing [27]. The problem of efficiently generating bracelets has been considered by several researchers, but has remained an open problem for some time. For example, Lisonek [17] modified Savage and Wang's necklace algorithm [21] to generate bracelets. This algorithm has running time $O(n \cdot B_k(n))$, where $B_k(n)$ denotes the number of k-ary bracelets of length n. No previously known algorithm has achieved a lower time bound.

This dissertation presents fast generation algorithms for each of the preceding objects. Chapter 2 provides a background on mathematical notation as well as a background on necklaces, Lyndon words and pre-necklaces. A recursive algorithm for

generating these objects is outlined in detail.

In Chapter 3, an efficient algorithm for generating binary unlabeled necklaces is presented, along with a proof showing the algorithm is CAT.

In Chapter 4, a modified version of the recursive necklace generation algorithm is outlined. Then, using this modified algorithm, an efficient algorithm for generating k-ary fixed density necklaces is presented, along with a proof showing the algorithm is CAT. As an application, the algorithm is used to generate difference covers.

In Chapter 5, two algorithms are outlined for generating necklaces where the number of each alphabet symbol is fixed. The first algorithm is simple, but does not appear to run in constant amortized time. The second algorithm is more complex, but experimental evidence indicates that it is CAT. The time bound is proved in a special case.

In Chapter 6, simple counting techniques are used to derive the already known formula for enumerating chord diagrams. In addition, two algorithms are developed for generating chord diagrams. Experimental results indicate that the latter of the two algorithms presented runs in constant amortized time.

Chapter 7 combines algorithms for generating strings and necklaces with a real-time pattern matching algorithm. The combination of these algorithms results in CAT algorithms for generating both strings and circular strings with forbidden substrings, and necklaces with forbidden Lyndon substrings. The analysis proves that the number of strings with forbidden substring f is proportional to the number of circular strings with forbidden substring f.

In Chapter 8, an efficient algorithm for generating k-ary bracelets is presented along with a proof showing the algorithm runs in constant amortized time. There is also a brief discussion of strings with no 0^t substring.

Lyndon words of length n can be used to form a basis for the n-th homogeneous component of the free Lie algebra. Chapter 9 outlines a linear time algorithm for generating a special bracketing of the Lyndon words which corresponds to this basis.

Chapter 10 summarizes the research contributions made in this dissertation, and outlines avenues of future research.

Chapter 2

Background

This chapter gives a background of necklaces, Lyndon words, and pre-necklaces along with an overview of a recursive necklace generation algorithm. First, there is a discussion of mathematical notation and an important lemma.

2.1 Phi and mu

Two number theoretic functions appear frequently throughout this dissertation. The *Euler totient* function on a positive integer n, denoted $\phi(n)$, is the number of integers in the set $\{0, 1, \ldots, n-1\}$ that are relatively prime to n. The *Möbius* function $\mu(n)$ of a positive integer n is defined by the following formula:

$$\mu(n) = \begin{cases} (-1)^r & \text{if } n \text{ is the product of } r \text{ distinct prime numbers.} \\ 0 & \text{otherwise.} \end{cases}$$

The following lemma, known as the Möbius inversion principle, can be used to enumerate many of the aperiodic objects discussed in this dissertation.

LEMMA 1 *If f and g are functions on the positive integers then*

$$g(n) = \sum_{d|n} f(d) \qquad \textit{if and only if} \qquad f(n) = \sum_{d|n} \mu(d) g(\frac{n}{d}).$$

A proof for this lemma may be found in [13].

2.2 Burnside's lemma

One of the most useful tools for enumerating combinatorial objects with equivalence under some group action is Burnside's Lemma.

BURNSIDE'S LEMMA *If a group G acts on a set S and $Fix(g) = \{s \in S | g(s) = s\}$, then the number of equivalence classes is given by*

$$\frac{1}{|G|} \sum_{g \in G} |Fix(g)|.$$

This lemma is used to enumerate *all* of the objects discussed in this dissertation that have equivalence under some group action. In particular, this lemma is used along with simple counting techniques to derive an enumeration formula for chord diagrams.

2.3 Necklaces, Lyndon words, and pre-necklaces

The fundamental object behind each of the algorithms described in this dissertation is the necklace. A *necklace* is the canonical representative of an equivalence class of k-ary strings under rotation. Unless otherwise stated, the canonical representative is the lexicographically smallest element in the equivalence class. As an example, the set of all binary necklaces of length 4 is {0000,0001,0011,0101,0111,1111}. The set of all k-ary necklaces with length n is denoted by $\mathbf{N}_k(n)$. The cardinality of this set is denoted by $N_k(n)$.

Using the lexicographically smallest element as the canonical representative, an aperiodic necklace is called a *Lyndon word*. The set of all k-ary Lyndon words with length n is denoted by $\mathbf{L}_k(n)$ and has cardinality $L_k(n)$. The term *periodic necklace* is reserved for all necklaces that are not aperiodic.

A word is called a *pre-necklace* if it is the prefix of some necklace. The set of all k-ary pre-necklaces with length n is denoted by $\mathbf{P}_k(n)$ and has cardinality $P_k(n)$.

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

$$N_k(n) = \frac{1}{n}\sum_{d|n} \phi(d) k^{n/d}. \tag{2.1}$$

$$L_k(n) = \frac{1}{n}\sum_{d|n} \mu(d) k^{n/d}. \tag{2.2}$$

$$P_k(n) = \sum_{i=1}^{n} L_k(i). \tag{2.3}$$

PROOF: The equations for $L_k(n)$ and $N_k(n)$ are verified by Gilbert and Riordan [12]. The equation for $P_k(n)$ follows from Lemma 6 (stated later in this section). □

We now state several lemmas about necklaces, Lyndon words, and pre-necklaces. The following two lemmas are proved in [21].

LEMMA 2 *If α is a necklace, then α^t is a necklace for $t \geq 1$.*

LEMMA 3 *If $\alpha b \in \mathbf{P}_k(n)$ is a pre-necklace, where $b < k-1$, then $\alpha(b+1) \in \mathbf{L}_k(n)$.*

The next lemma can be proved directly from the definitions of a necklace and a pre-necklace. Inequalities between words are always with respect to lexicographic order.

LEMMA 4 *Let $\alpha = a_1 \cdots a_n$ be a pre-necklace. If x is a substring of α with length k, then $x \geq a_1 \cdots a_k$.*

Reutenauer [19] gives a useful lemma about Lyndon words.

LEMMA 5 *If α and β are Lyndon words with $\alpha < \beta$, then $\alpha\beta$ is a Lyndon word.*

The following lemma from Ruskey [20] characterizes pre-necklaces. It uses the function *lyn* on strings which is the length of the longest Lyndon prefix of the string. Formally, if $a_1 a_2 \cdots a_n$ is a k-ary string then

$$lyn(a_1 a_2 \cdots a_n) = max\{1 \leq p \leq n | a_1 a_2 \cdots a_p \in \mathbf{L}_k(p)\}.$$

```
procedure Necklace ( t, p : integer ):
local j : integer:
begin
    if t > n then PrintIt( p )
    else begin
        a_t := a_{t-p}:  Necklace( t + 1, p ):
        for j ∈ {a_{t-p} + 1, ..., k - 2, k - 1} do begin
            a_t := j:  Necklace( t + 1, t ):
end: end: end:
```

Figure 2.1: The recursive necklace algorithm

LEMMA 6 *Let k-ary string $\alpha = a_1 \cdots a_n$ and $p = lyn(\alpha)$. Then $\alpha \in \mathbf{P}_k(n)$ if and only if $a_{j-p} = a_j$ for $j = p+1, \ldots, n$.*

The following, very important theorem, is proved in [20]. It leads to the recursive necklace generation algorithm described in the next subsection. It can also be used to develop a linear time algorithm (equivalent to Duval's algorithm [9]) for factoring a string into Lyndon words. This algorithm, in turn, yields a linear time algorithm for finding the necklace of an arbitrary string [20].

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 $a_{n-p} \leq b \leq k-1$. Furthermore,*

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

2.3.1 A recursive necklace generation algorithm

The recursive necklace generation algorithm **Necklace**(t, p), shown in Figure 2.1, follows directly from Theorem 2. The general approach of this algorithm is to generate all length n pre-necklaces. The pre-necklace being generated is stored in the array a; one position for each character. We assume that $a_0 = 0$. The initial call is **Necklace**(1,1) and each recursive call appends a character to the pre-necklace to get a new

pre-necklace. At the beginning of each recursive call to Necklace(t, p), the length of the pre-necklace being generated is $t-1$ and the length of the longest Lyndon prefix is p. As long as the length of the current pre-necklace is less than n, each call to Necklace(t, p) makes one recursive call for each value from a_{t-p} to $k-1$, updating the values of both t and p in the process. This algorithm can generate necklaces, Lyndon words or pre-necklaces of length n in lexicographic order by specifying which object we want to generate. The function PrintIt(p) allow us to differentiate between these various objects as shown in Table 2.1.

Sequence type	PrintIt(p)
Pre-necklaces ($\mathbf{P}_k(n)$)	println($a_1 \cdots a_n$)
Lyndon words ($\mathbf{L}_k(n)$)	**if** $p = n$ **then** println($a_1 \cdots a_n$)
Necklaces ($\mathbf{N}_k(n)$)	**if** $n \bmod p = 0$ **then** println($a_1 \cdots a_n$)

Table 2.1: Different objects output by different versions of PrintIt

The computation tree for Necklace(t, p) consists of all pre-necklaces of length less than or equal to n. As an example, we show a computation tree for $N_2(4)$ in Figure 2.2. By comparing the number of nodes in the computation tree to the number of objects generated, it was shown that this algorithm is CAT [20].

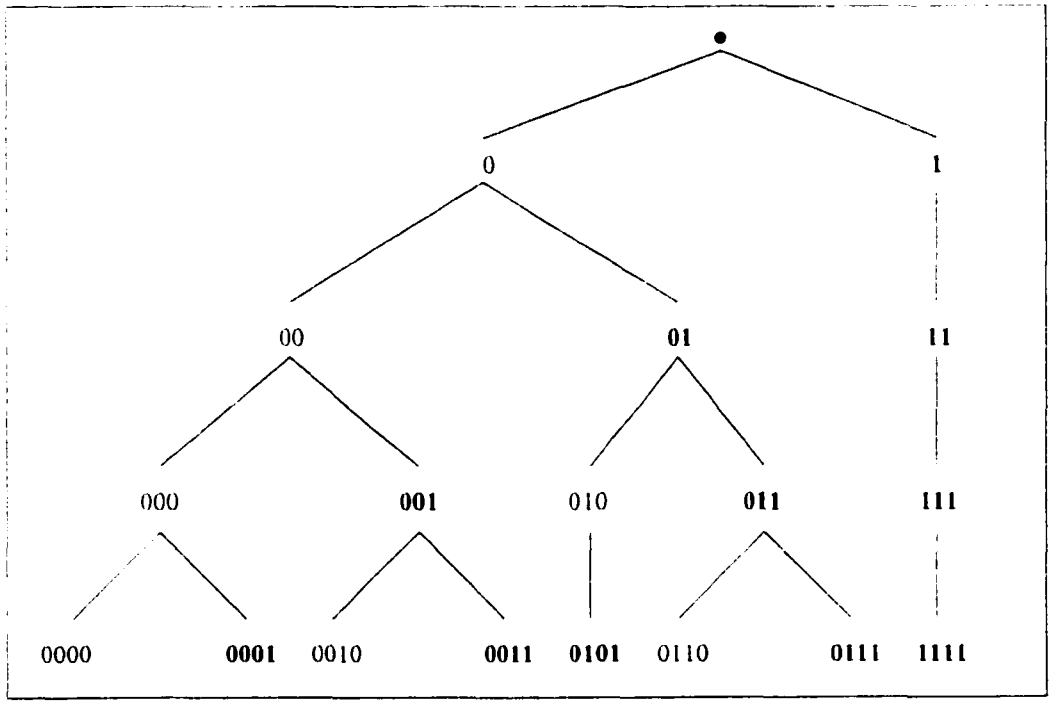

Figure 2.2: Computation tree for $N_2(4)$ from Necklace(t, p)

Chapter 3

Unlabeled Necklaces

This chapter outlines a CAT algorithm for generating binary unlabeled necklaces.

3.1 Background

An *unlabeled necklace* is the canonical representative (lexicographically smallest element) of an equivalence class of strings under rotation and permutation of its alphabet symbols. From this definition, the necklaces 0001 and 0111 are in the same equivalence class since one can be transformed into the other by permuting the symbols 0 and 1. The set of all k-ary unlabeled necklaces with length n is denoted $\widehat{\mathbf{N}}_k(n)$ with cardinality $\widehat{N}_k(n)$. For example, $\widehat{\mathbf{N}}_2(4) = \{0000, 0001, 0011, 0101\}$. An *unlabeled Lyndon word* is an aperiodic unlabeled necklace. The set of all k-ary length n unlabeled Lyndon words is denoted $\widehat{\mathbf{L}}_k(n)$ with cardinality $\widehat{L}_k(n)$. A word is called an *unlabeled pre-necklace* if it the prefix of some unlabeled necklace. The set of all k-ary unlabeled pre-necklaces is denoted by $\widehat{\mathbf{P}}_k(n)$, and has cardinality $\widehat{P}_k(n)$.

The following theorem gives enumeration formulas for these objects in the binary case. General formulas for $\widehat{N}_k(n)$ and $\widehat{L}_k(n)$ also exist and can be found in [12].

THEOREM 3 *The following formulae are valid for all* $n \geq 1$, $k \geq 1$:

$$\widehat{N}_2(n) = N_2(n) - \frac{1}{2n} \sum_{\text{odd } d|n} \phi(d) 2^{n/d}, \tag{3.1}$$

$$\widehat{L}_2(n) = \frac{1}{2n} \sum_{\text{odd } d|n} \mu(d) 2^{n/d}. \tag{3.2}$$

$$\widehat{P}_2(n) = \sum_{i=1}^{n} \widehat{L}_2(i). \tag{3.3}$$

PROOF: The equations for $\widehat{N}_2(n)$ and $\widehat{L}_2(n)$ are from Gilbert and Riordan [12]. The equation for $\widehat{P}_2(n)$ follows from Lemma 8 (stated in the next section). □

3.2 Generating binary unlabeled necklaces

This section focuses on generating binary unlabeled necklaces. It is an open problem to generate unlabeled necklaces over a general alphabet of size k.

Binary unlabeled necklaces can be generated by generating all binary necklaces and then performing a test on each necklace to determine whether or not it is the unlabeled representative. In the binary case, a necklace and its complement are in the same equivalence class. Therefore, this test can be performed by taking the complement of the generated necklace and using a necklace finding algorithm to find its corresponding necklace. Such an algorithm runs in linear time (see Chapter 2.3). The resulting necklace is then compared to the original: if the original is not larger, then it is an unlabeled necklace. This algorithm yields an overall running time of $O(n \cdot \mathbf{N}(n))$, which is far from efficient.

To improve upon this naïve algorithm, the linear time test required at the end of the necklace generation must be eliminated. The remainder of this section is used to prove Theorem 5. This theorem for unlabeled pre-necklaces is analogous to Theorem 2 for pre-necklaces. It suggests the addition of another parameter c to the routine **Necklace**(t, p) which replaces the need for the linear time test. Several lemmas are needed before we state and prove this theorem. The following two lemmas are analogous to Lemma 2 and Lemma 6 for necklaces.

LEMMA 7 *If* $\alpha = a_1 \cdots a_n \in \widehat{\mathbf{N}}$, *then* $\alpha^t \in \widehat{\mathbf{N}}$ *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 α^t is a necklace and therefore by definition α^t is an unlabeled necklace for $t \geq 1$. □

LEMMA 8 *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 7, 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 $\pi(a_1 \ldots a_q) < a_1 \ldots a_q$. This contradicts the assumption that α is an unlabeled pre-necklace, 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 6, $a_{j-p} = a_j$ for $j = p+1, \ldots, n$. □

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 $1 \leq i \leq k$. The set $\hat{\mathbf{N}}_2(n)$ consists exactly of those necklaces α that satisfy $r \geq \alpha$ for all $r \in \mathbf{Rot}_n(\alpha)$. Observe that

$$\text{if } |x| = |y|, \text{ then } x \leq y \text{ if and only if } \bar{x} \geq \bar{y}. \tag{3.1}$$

THEOREM 4 *Let $\alpha = a_1 \cdots a_n \in \mathbf{N}_2(n)$. If there is a k such that, for every $r \in \mathbf{Rot}_k(\alpha)$, $r \geq \alpha$ 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, it must be shown that $r \geq \alpha$ for all $r \in \mathbf{Rot}_n(\alpha)$. It is given that all $r \in \mathbf{Rot}_k(\alpha)$ satisfy this condition so it must only be shown that $\overline{a_{j+1} \cdots a_n a_1 \cdots a_j} \geq \alpha$ for $k \leq j < n$.

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 4 yields either $\overline{a_{j+1} \cdots a_n} > a_1 \cdots a_{n-j}$, or $\overline{a_{j+1} \cdots a_n} = a_1 \cdots a_{n-j}$. In the former case the result

is trivial. In the latter case consider $\overline{a_{n-j+1}\cdots a_n}$ and look at two subcases. If $n-j+1 \leq k$, then $r = \overline{a_{n-j+1}\cdots a_n a_1 \cdots a_{n-j}} \in \mathbf{Rot}_k(\alpha)$. Thus $r \geq \alpha$ which implies $\overline{a_{n-j+1}\cdots a_n} \geq a_1 \cdots a_j$. If $n-j+1 > k$, then $\overline{a_{n-j+1}\cdots a_n}$ is a substring of $\overline{a_{k+1}\cdots a_n}$ and is therefore a substring of the pre-necklace $a_1 \cdots a_{n-j}$. By Lemma 4, $\overline{a_{n-j+1}\cdots a_n} \geq a_1 \cdots a_j$. Thus in both subcases $\overline{a_{n-j+1}\cdots a_n} \geq a_1 \cdots a_j$. Now using (3.1), we have $\overline{a_1 \cdots a_j} \geq a_{n-j+1}\cdots a_n$ which implies that $\overline{a_{j+1}\cdots a_n a_1 \cdots a_j} \geq \alpha$ for $k \leq j < n$. □

COROLLARY 1 *Let $\alpha = a_1 \cdots a_n \in \mathbf{N}_2(n)$. If $\overline{a_i \cdots a_n} \geq a_1 \cdots a_{n-i+1}$ for all $1 \leq i \leq n$ 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 Theorem 4, α is an unlabeled necklace. Otherwise, α is an unlabeled necklace by definition. □

Define $com(a_1 \cdots a_n)$ to be the smallest positive value c for which

$$\overline{a_{c+1}\cdots a_n} = a_1 \cdots a_{n-c}. \tag{3.5}$$

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. One final lemma is given before stating the main theorem.

LEMMA 9 *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 $\widehat{\mathbf{P}}(n)$ 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, let $p = lyn(\alpha)$. If $n \ mod \ p = 0$ then α is a necklace. By Corollary 1, 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 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 (3.4) 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 1 shows 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 $b = a_{n-c}$. 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 Lemma 9 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 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 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 9. 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 4 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 clearly $com(\alpha b) = c$. If $b = a_{n-c} = 0$, then $com(\alpha b) = n$ since there is no value of c for which (3.5) 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 unlabeled pre-necklaces (and thus unlabeled necklaces)

```
Procedure Unlabeled ( 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:  Unlabeled( t + 1, p, t ):
            end:
            a_t := 1:
            if a_{t-p} = 1 then Unlabeled( t + 1, p, c ):
            else Unlabeled( t + 1, t, c ):
        end else begin
            a_t := 0:  Unlabeled( t + 1, p, c ):
end: end: end:
```

Figure 3.1: Unlabeled binary necklace algorithm

can be generated by introducing the additional parameter c to the pre-necklace generation algorithm Necklace(t, p). Pseudocode for this algorithm is given in Figure 3.1. The initial call is Unlabeled$(2, 1, 1)$, first initializing $a_0 = a_1 = 0$. Unlabeled necklaces, Lyndon words, and pre-necklaces can all be produced by using Table 2.1 as before.

3.3 Analysis

Observe that the computation tree of Unlabeled(t, p, c) is a subtree of the computation tree of Necklace(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 6 *Algorithm Unlabeled(t, p, c) for generating binary unlabeled necklaces is CAT.*

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.

Chapter 4

Fixed Density Necklaces

This chapter develops a CAT algorithm for generating fixed density necklaces. An additional algorithm is presented for the binary case. As an application, an algorithm is outlined to generate difference covers.

4.1 Background

The *density* of a string is defined to be the number of non-zero characters in the string. Thus, a length n string with density d contains exactly $n-d$ zeros. The set of k-ary necklaces with density d and length n is represented by $\mathbf{N}_k(n,d)$ and has cardinality $N_k(n,d)$. For example $\mathbf{N}_3(4,2) = \{0011,0012,0021,0022,0101,0102,0202\}$. Similarly, the set of fixed density Lyndon words is represented by $\mathbf{L}_k(n,d)$ with cardinality $L_k(n,d)$. The set of fixed density pre-necklaces is denoted by $\mathbf{P}_k(n,d)$ and has cardinality $P_k(n,d)$. In addition to these familiar terms we introduce the set $\mathbf{P}'_k(n,d)$ which is the elements of $\mathbf{P}_k(n,d)$ whose last character is non-zero. Its cardinality is denoted $P'_k(n,d)$.

Fixed density necklaces can be counted using the object discussed in the following chapter: necklaces where the number of each alphabet symbol is fixed. Let $\mathbf{N}(n_0,n_1,\ldots,n_{k-1})$ denote the set of necklaces composed of n_i occurrences of the symbol i, for $i = 0,1,\ldots,k-1$. The cardinality of this set is denoted $N(n_0,n_1,\ldots,n_{k-1})$. Similarly, the set of all Lyndon words composed of n_i occurrences of the symbol i is

denoted by $\mathbf{L}(n_0, n_1, \ldots, n_{k-1})$, with cardinality $L(n_0, n_1, \ldots, n_{k-1})$. The two enumeration formulas stated in the following theorem are known from Gilbert and Riordan [12].

THEOREM 7 *The following formulae are valid for all* $n_i \geq 1$, $k \geq 1$:

$$N(n_0, n_1, \ldots, n_{k-1}) = \frac{1}{n} \sum_{j \mid gcd(n_0, n_1, \ldots, n_{k-1})} \phi(j) \frac{(n/j)!}{(n_0/j)! \cdots (n_{k-1}/j)!} \tag{4.1}$$

$$L(n_0, n_1, \ldots, n_{k-1}) = \frac{1}{n} \sum_{j \mid gcd(n_0, n_1, \ldots, n_{k-1})} \mu(j) \frac{(n/j)!}{(n_0/j)! \cdots (n_{k-1}/j)!} \tag{4.2}$$

Let the density of the necklace $d = n_1 + \cdots + n_{k-1}$ and $n_0 = n - d$. To get the number of fixed density necklaces (and Lyndon words) with length n and density d, we sum over all possible values of $n_1, n_2, \ldots, n_{k-1}$ to obtain:

$$N_k(n, d) = \sum_{n_1 + \cdots + n_{k-1} = d} N(n - d, n_1, \ldots, n_{k-1}) \tag{4.3}$$

$$L_k(n, d) = \sum_{n_1 + \cdots + n_{k-1} = d} L(n - d, n_1, \ldots, n_{k-1}) \tag{4.4}$$

In the binary case these expressions simplify as follows

$$N_2(n, d) = \frac{1}{n} \sum_{j \mid gcd(n, d)} \phi(j) \binom{n/j}{d/j}$$

$$L_2(n, d) = \frac{1}{n} \sum_{j \mid gcd(n, d)} \mu(j) \binom{n/j}{d/j}$$

Currently, it is not known how to count fixed density pre-necklaces.

4.2 Generating fixed density necklaces

We use a two step approach to develop a fast algorithm for generating fixed density necklaces. First we create a new necklace algorithm based on the recursive necklace generation algorithm **Necklace**(t, p) presented in Chapter 2.3.1. We then optimize this

new necklace algorithm for the fixed density case by making a few key observations about fixed density necklaces.

4.2.1 Modified necklace algorithm

For every necklace of positive density, the last character of the string must be non-zero. Thus, if we are concerned only with generating necklaces or Lyndon words we can reduce the size of the computation tree by compressing all of the pre-necklaces whose last character is 0. Looking at Figure 2.2, we want to generate only the nodes in bold. This results in the modified computation tree shown in Figure 4.1. Notice that at each successive level in this tree we are incrementing the density of the pre-necklace rather than the length. To generate this modified tree we create a recursive routine based on the original necklace algorithm in Figure 2.1; however, rather than determining the valid values for the next position in the string, we need to determine both the valid positions and the values for the next non-zero character.

To make this change we use the array a to hold the positions of the non-zero characters and maintain another array b to indicate the values of the non-zero characters. The i-th element of the array a represents the position of the i-th non-zero character, and the i-th element of the array b represents the value of the i-th non-zero character. Thus if we generate a necklace with length 7 with $a = [3,4,5,7]$ and $b = [1,3,2,1]$, the corresponding necklace is 0013201. It is important to note that we can also maintain the original necklace structure by performing some extra constant time operations. Note that in the binary case, the second array b is not necessary since all non-zero characters must be 1. We use the parameter t to indicate the current density of the string. The length of the current string is a_t. Since all Lyndon prefixes end in a non-zero character, we let a_p indicate the length of the longest Lyndon prefix. Using these two parameters, we can compute all valid positions and values for the next non-zero character.

To determine the valid positions and values for the next non-zero character and to maintain the lexicographic ordering we compute the maximum position and the minimum value for that position so that the new string still has the pre-necklace

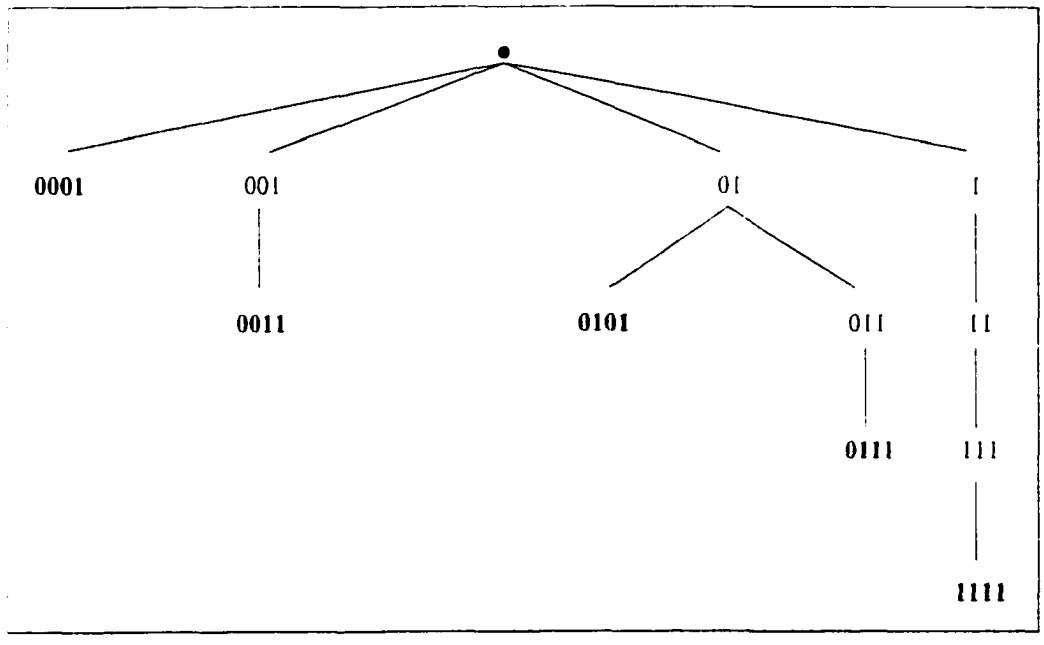

Figure 4.1: Computation tree for $N_2(4)$ from ModifiedNeck(t, p)

property. The maximum position for the next character is given by $a_{(t+1-p)} + a_p$. The minimal value for this position is $b_{(t+1-p)}$. By the properties of pre-necklaces all larger values at the maximal position are also valid [21]. Also, all positions before the maximum position and greater than the position of the last assigned non-zero character (a_t) can hold all values ranging from 1 to $k-1$. (Note that since we want to generate all necklaces with length n, we restrict the position to be less than or equal to n.) For each of these valid combinations of position and value, we lexicographically assign the position to a_{t+1} and the value to b_{t+1}, followed by a recursive call updating both t and p. Finally, if the position of the last non-zero element is greater than or equal to n, we call the PrintIt(p) function to print out either the Lyndon words or necklaces in a similar manner to the original algorithm Necklace(t, p).

This modified algorithm, ModifiedNeck(t, p), for generating necklaces is given in Figure 4.2. Each initial branch of the computation tree is a result of a separate call to ModifiedNeck(t, p), each call specifying a different combination for the position and value of the first non-zero character. Note that the 0 string is not generated by this algorithm and must be generated separately. The nodes of the resulting computation

```
procedure ModifiedNeck ( t, p : integer ):
local i, j, max : integer;
begin
    if a_t ≥ n then PrintIt( p )
    else begin
        max = a_(t+1-p);
        if max ≤ n then begin
            a_{t+1} := max;
            b_{t+1} := b_(t+1-p);
            ModifiedNeck(t + 1, p );
        end else begin
            max := n;   a_{t+1} := n;   b_{t+1} := 1;
            ModifiedNeck( t + 1, t + 1 );
        end;
        for i ∈ {b_{t+1} + 1, ..., k − 2, k − 1} do begin
            b_{t+1} := i;
            ModifiedNeck( t + 1, t + 1 );
        end;
        for j ∈ {max − 1, max − 2, ..., a_t + 1} do begin
            a_{t+1} := j;
            for i ∈ {1, ..., k − 2, k − 1} do begin
                b_{t+1} := i;
                ModifiedNeck( t + 1, t + 1 );
    end; end; end;
end;
```

Figure 4.2: Modified necklace algorithm

tree for ModifiedNeck(t, p) are all pre-necklaces with length less than or equal to n whose last character is non-zero.

Observe that we are not restricted to generating the necklaces in lexicographic order. Many orders are possible by re-ordering the order of the recursive calls.

4.2.2 Fixed density necklace algorithm

We now optimize our modified algorithm for the fixed density case by making several observations. First, we restrict the position of the first non-zero character depending on the density. In particular, there are no necklaces with density d that can have the first non-zero character in a position after $n - d + 1$ or before $\lfloor (n-1)/d + 1 \rfloor$. Also,

if we are generating a string with length n and density d and have just placed the i-th non-zero character then the $(i+1)$-st non-zero character must come before the position $n-(d-i)+2$. If we place the next character at or after this position then any resulting string with length n will have density less than d. Also, because the last non-zero character must be in the n-th position, we stop the string generation after placing the $(d-1)$-st non-zero character. Thus, the strings generated by following this last restriction are strings with length less than n and density $d-1$. By following this approach, we may generate up to $k-1$ strings for each call to PrintIt(p) since we can place up to $k-1$ characters in the n-th position. However, it is not always the case that we will generate all $k-1$ strings or even any strings with each call to PrintIt(p). Thus we add an additional constant time test to see which values can be placed in the n-th position. This test is similar to the test for finding the maximal valid position and minimum value for the next non-zero character as outlined in the previous sub-section. Once a minimum value is determined (if there is one at all), we perform the usual tests to determine if the string is a necklace or a Lyndon word. All larger values for the n-th position will result in a string that is a Lyndon word [21]. Thus the overall work done in the PrintIt(p) function to determine the valid strings remains constant for each string generated.

In summary, we use our modified necklace algorithm outlined in Figure 4.2 with the following optimizations:

1. The first non-zero character must be between $n-d+1$ and $(n-1)/d+1$ inclusive.

2. The i-th non-zero character must be placed at or before the $(n-d+i)$-th position.

3. Stop generating when we have assigned $d-1$ non-zero characters.

4. Determine valid values for n-th position in PrintIt(p) function.

The computation tree for generating $N_2(7,3)$ is given in Figure 4.3. The dotted lines indicate the initial branches we do not need to follow by modification 1. The

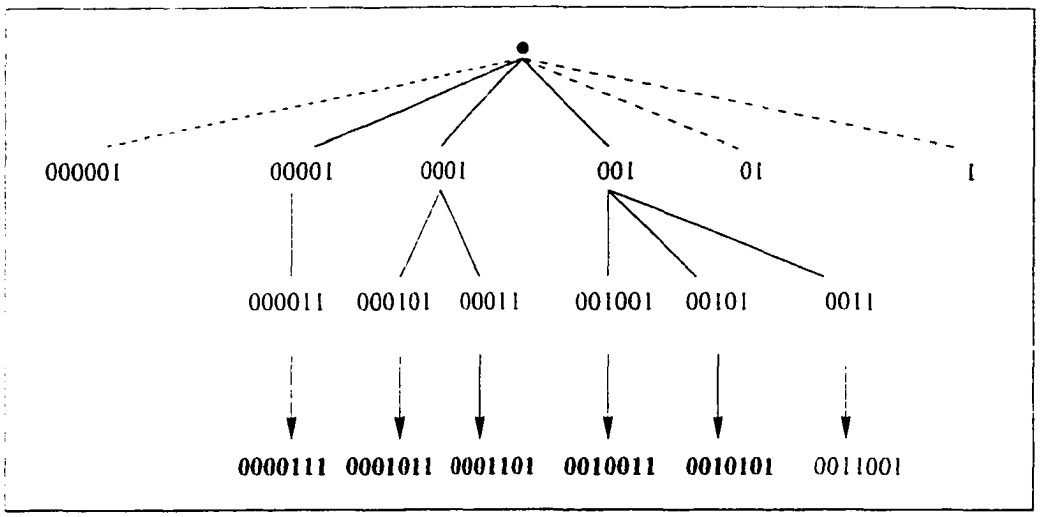

Figure 4.3: Computation tree (solid edges only) for $N_2(7,3)$ from FixedDensity(t,p)

arrows indicate the strings produced by adding the final character to the n-th position. The bold strings indicate the actual necklaces produced by the PrintIt(p) function. The remaining string (0011001) is rejected since it is not a necklace.

The algorithm for generating fixed density necklaces and Lyndon words in lexicographic order is given in Figure 4.4. Binary necklaces with $d > n/2$ can be generated by complementing the output from generating necklaces with density $n - d$. In this case, however, the strings generated are not in lexicographic order and are not necessarily the lexicographic representatives for their respective equivalence classes. To generate fixed density pre-necklaces, we generate $N(n+1, d+1)$ and print out only the first n characters making sure we do not print the same string twice.

4.3 Analysis

In this section we show that FixedDensity(t,p) is CAT. We start the analysis by analyzing several trivial cases. When the desired density of the string is n, the computation tree and strings produced are equivalent to the generation of $N_{k-1}(n)$ which we already know is CAT. When the density is 0 we simply generate the 0 string, and when $d = 1$ we generate the $k-1$ strings where the last bit ranges from 1 to

```
procedure FixedDensity ( t, p : integer ):
local i, j, max, tail : integer;
begin
    if t ≥ d − 1 then PrintIt(p);
    else begin
        tail := n − (d − t) + 1;
        max := a_(t+1−p);
        if max ≤ tail then begin
            a_{t+1} := max;
            b_{t+1} := b_(t+1−p);
            FixedDensity( t + 1, p );
            for i ∈ {b_{t+1} + 1, ..., k − 2, k − 1} do begin
                b_{t+1} := i;
                FixedDensity( t + 1, t + 1 );
            end;
            tail := max − 1;
        end;
        for j ∈ {tail, tail − 1, ..., a_t + 1} do begin
            a_{t+1} := j;
            for i ∈ {1, ..., k − 2, k − 1} do begin
                b_{t+1} := i;
                FixedDensity( t + 1, t + 1 );
    end; end; end;
end;
```

Figure 4.1: Fixed density necklace algorithm

$k-1$ and the rest of the string is all 0's. In each case where the density is greater than 0 the resulting strings are generated in constant amortized time.

For the non-trivial cases we examine the number of nodes in the computation tree, noting that the amount of work to generate each node is constant. When $1 < d < n$, the nodes in the computation tree consist only of pre-necklaces that end in a non-zero bit with density i ranging from 1 to $d-1$ and length ranging from $(n-1)/d + i$ to $n - d + i$. Recall that $\mathbf{P}'_k(n,d)$ is the set of pre-necklaces with length n and density d where the last bit is non-zero. Thus, the size of the computation tree for our fixed density algorithm ($1 < d < n$) is bounded by the expression

$$CompTree_k(n,d) \leq \sum_{i=1}^{d-1} \sum_{j=\frac{n-1}{d}+i}^{n-d+i} P'_k(j,i).$$

Recall that we generate binary fixed density necklaces with density greater than $n/2$ by generating $\mathbf{N}(n, n-d)$ and complementing the output. Therefore in the case where $k = 2$ (and only in this case) we have the restriction that d is less than or equal to $n/2$.

To prove that our algorithm is efficient we will show that the ratio between the size of the computation tree and the number of strings produced is bounded by a constant. Since there does not appear to be a simple explicit formula for $P'_k(n,d)$ our approach will be to derive an upper bound in terms of $N_k(n,d)$ and $L_k(n,d)$.

LEMMA 10 *The following inequality is valid for all $n \geq 1$, $k \geq 1$, and $0 \leq d \leq n$:*

$$P'_k(n,d) \leq N_k(n,d) + L_k(n,d).$$

PROOF: We partition $\mathbf{P}'_k(n,d)$ into two categories: necklaces and non-necklaces. Let the elements of $\mathbf{P}'_k(n,d)$ that are not necklaces be $\mathbf{Q}'_k(n,d)$.

We show that $Q'_k(n,d) \leq L_k(n,d)$ by providing an injective mapping of $\mathbf{Q}'_k(n,d)$ to $\mathbf{L}_k(n,d)$. By Lemma 6 each element of the set $\mathbf{Q}'_k(n,d)$ must have the form: $\alpha = (a_1 \cdots a_p)^j a_1 \cdots a_m$ where $p = lyn(\alpha)$, $j \geq 1$ and $0 < m < p$. Let n_i be the number of occurrences of the symbol i in $a_1 \cdots a_m$ and define the string $\gamma = 0^{n_0} 1^{n_1} \cdots (k-1)^{n_{k-1}}$. We define a function f on the set $\mathbf{Q}'_k(n,d)$ as follows:

$$f(\alpha) = \gamma (a_1 \cdots a_p)^j.$$

For example, $f((0021013)^7 0021013) = 0001123(0021013)^7$. This mapping preserves both length and density. Since γ and $a_1 \cdots a_p$ are both Lyndon words and $\gamma < a_1 \cdots a_p$, it follows from repeated use of Lemma 5 that $f(\alpha) \in \mathbf{L}_k(n,d)$.

To show that f is injective consider two unique elements of $\mathbf{Q}'_k(n,d)$: $\alpha =$

$(a_1 \cdots a_p)^s a_1 \cdots a_i$ and $\beta = (b_1 \cdots b_q)^t b_1 \cdots b_j$. If $i = j$ then $f(\alpha) \neq f(\beta)$ since $a_1 \cdots a_p$ and $b_1 \cdots b_j$ are both Lyndon words and $a_1 \cdots a_p \neq b_1 \cdots b_q$. Otherwise assume that $i < j$. Since a_i and b_j are both non-zero, the i-th element of $f(\alpha)$ is non-zero and the j-th element of $f(\beta)$ is non-zero. Now if the i-th element of $f(\beta)$ is non-zero then the $(i+1)$-st element must also be non-zero if $f(\alpha) = f(\beta)$. However the $(i+1)$-st element of $f(\alpha) = a_1$ which is 0. Thus $f(\alpha) \neq f(\beta)$ for unique $\alpha, \beta \in \mathbf{Q}'_k(n,d)$. Thus f is an injection from $\mathbf{Q}'_k(n,d)$ to $\mathbf{L}_k(n,d)$.

Now since there exists an injective mapping from $\mathbf{Q}'_k(n,d)$ to $\mathbf{L}_k(n,d)$ we have $Q'_k(n,d) \leq L_k(n,d)$. From earlier discussion we know that $P'_k(n,d) = N_k(n,d) + Q'_k(n,d)$ and thus $P'_k(n,d) \leq N_k(n,d) + L_k(n,d)$. □

We observe in the binary case that by taking each element from $\mathbf{P}_2(n,d)$ and adding a 1 to the end of the string we get the set $\mathbf{P}'_2(n+1,d+1)$. Thus from the previous Lemma we also get an upper bound on $P_2(n,d)$.

COROLLARY 2 *The following inequality is valid for all $n \geq 1$ and $0 \leq d \leq n$:*

$$P_2(n,d) \leq N_2(n+1,d+1) + L_2(n+1,d+1).$$

We can now bound our computation tree as the sum of fixed density necklaces and fixed density Lyndon words:

$$CompTree_k(n,d) \leq \sum_{i=1}^{d-1} \sum_{j=\frac{n-1}{d}+i}^{n-d+i} N_k(j,i) + L_k(j,i).$$

However, by plugging the formulas for fixed density necklaces and Lyndon words into the above expression we end up with a complicated quadruple sum. Therefore we will prove two lemmas which give simple bounds for fixed density Lyndon words and necklaces.

LEMMA 11 *The following inequality is valid for all* $0 \leq d \leq n$:

$$L_k(n,d) \leq \frac{1}{n}\binom{n}{d}(k-1)^d.$$

PROOF: Each element of $\mathbf{L}_k(n,d)$ is a representative of an equivalence class of k-ary strings, each with n elements. If we add up the elements from each equivalence class we will get $nL_k(n,d)$ unique strings each of length n and density d. The expression $\binom{n}{d}(k-1)^d$ counts the total number of k-ary strings with length n and density d. Therefore $L_k(n,d) \leq \frac{1}{n}\binom{n}{d}(k-1)^d$. □

A similar bound for $N_k(n,d)$ is more difficult to obtain. Here we bound $N_k(n,d)$ by $L_k(n,d)$.

LEMMA 12 *The following inequality is valid for all* $0 < d < n$:

$$\frac{1}{n}\binom{n}{d}(k-1)^d \leq N_k(n,d) \leq 2L_k(n,d).$$

PROOF: By considering case when $j = 1$ in equation (4.1) and noting that the remaining terms are all non-negative we have

$$\begin{aligned} N_k(n,d) &\geq \frac{1}{n} \sum_{n_1+\cdots+n_{k-1}=d} \frac{n!}{(n_0!)(n_1!)\cdots(n_{k-1}!)} \\ &= \frac{1}{n}\binom{n}{d} \sum_{n_1+\cdots+n_{k-1}=d} \frac{d!}{(n_1!)\cdots(n_{k-1}!)} \\ &= \frac{1}{n}\binom{n}{d}(k-1)^d. \end{aligned}$$

The final equality is a result of the basic multinomial expansion.

To show that $N_k(n,d) \leq 2L_k(n,d)$, we provide an injective mapping of the periodic necklaces to Lyndon words. If α is a periodic necklace then $\alpha = (a_1 \cdots a_p)^j$ where $p = lyn(\alpha)$ and $j > 1$. Since $d < n$ we know that $a_1 = 0$. We define a function g on

all periodic necklaces with length n and density d as follows:

$$g(\alpha) = 0(a_1 \cdots a_p)^{j-1} a_2 \cdots a_p.$$

This function simply moves the bit $a_{p(j-1)+1} = a_1 = 0$ to the front of the string. This operation preserves both length and density. Since $(a_1 \cdots a_p)^{j-1} a_2 \cdots a_p$ is a Lyndon word, by Lemma 5 $g(\alpha)$ is a Lyndon word.

To show that g is an injection we consider two unique periodic necklaces: $\alpha = (a_1 \cdots a_p)^t$ and $\beta = (b_1 \cdots b_q)^s$. If $p = q$ and $g(\alpha) = g(\beta)$ then $a_1 \cdots a_p = b_1 \cdots b_q$ contradicting the fact that $\alpha \neq \beta$. If $p \neq q$, then assume that $p < q$. This implies that $t > j > 1$. Now comparing the characters in positions $2, 3, \ldots, q+1$ of $g(\alpha)$ and $g(\beta)$ we observe that if $g(\alpha) = g(\beta)$ then $b_1 \cdots b_q = (a_1 \cdots a_p)^t a_1 .. a_s$ for some $t \geq 1$ and $1 \leq s \leq p$. However since $a_1 \cdots a_p$ is a Lyndon word then $(a_1 \cdots a_p)^t a_1 \cdots a_s$ is periodic if $s = p$ and is not a necklace if $s < p$. This contradicts the fact that $b_1 \cdots b_q$ is a Lyndon word. Thus $g(\alpha) \neq g(\beta)$ for unique periodic necklaces α and β. Therefore g is an injective mapping from the periodic necklaces to Lyndon words.

Since there exists an injective mapping from the periodic strings of $\mathbf{N}_k(n,d)$ to $\mathbf{L}_k(n,d)$ we get the result $N_k(n,d) \leq 2L_k(n,d)$. □

Using the previous lemmas we can simplify our upper bound on the size of the computation tree:

$$\begin{aligned}
CompTree_k(n,d) &= \sum_{i=1}^{d-1} \sum_{j=\frac{n-1}{d}+i}^{n-d+i} P'_k(j,i) \\
&\leq \sum_{i=1}^{d-1} \sum_{j=\frac{n-1}{d}+i}^{n-d+i} N_k(j,i) + L_k(j,i) \\
&\leq 3 \sum_{i=1}^{d-1} \sum_{j=1}^{n-d+i} L_k(j,i) \\
&\leq 3 \sum_{i=1}^{d-1} \sum_{j=1}^{n-d+i} \frac{1}{j} \binom{j}{i} (k-1)^i
\end{aligned}$$

$$= 3\sum_{i=1}^{d-1}\frac{1}{i}(k-1)^i\sum_{j=1}^{n-d+i}\binom{j-1}{i-1}$$

$$= 3\sum_{i=1}^{d-1}\frac{1}{i}\binom{n-d+i}{i}(k-1)^i. \qquad (4.5)$$

To get the last two equalities we used some basic binomial coefficient identities.

To simplify this bound for the computation tree even more, we inductively prove yet another upper bound for the remaining sum in equation (4.5). We first prove an upper bound for the case when $k > 2$ and $1 < d < n$. We then provide a similar proof for the case when $k = 2$. In the latter case we take advantage of the fact that we can generate binary necklaces with $d > n/2$ by generating necklaces with density $n - d$ and then complementing the output of each generated necklace to get all necklaces with density d. Once again, this is the only situation where the strings are not generated in lexicographic order. Thus when $k = 2$, we only consider the case when $1 < d \leq n/2$.

LEMMA 13 *For $2 \leq d < n$ and $k > 2$:*

$$\sum_{i=1}^{d-2}\frac{1}{i}\binom{n-d+i}{i}(k-1)^i < \frac{2}{d-1}\binom{n-1}{d-1}(k-1)^{d-1}.$$

PROOF: We prove the lemma by induction on d. Let

$$S_k(n,d) = \sum_{i=1}^{d-2}\frac{1}{i}\binom{n-d+i}{i}(k-1)^i.$$

Basis: $d = 2$ or 3, $n \geq 3$. Observe that this covers all cases for $n = 3, 4$:

$$d = 2 \quad S_k(n,2) = 0 < 2(n-1)(k-1)$$
$$d = 3 \quad S_k(n,3) = (n-2)(k-1) < \binom{n-1}{2}(k-1)^2$$

Assume: $S_k(n,d) < \frac{2}{d-1}\binom{n-1}{d-1}(k-1)^{d-1}$ for $1 < d < n-1$, $k > 2$, and $n \geq 5$. Consider

$S_k(n, d+1)$:

$$\begin{aligned}
S_k(n,d+1) &= \sum_{i=1}^{d-1} \frac{1}{i}\binom{n-d-1+i}{i}(k-1)^i \\
&= \sum_{i=1}^{d-2} \frac{1}{i}\binom{(n-1)-d+i}{i}(k-1)^i + \frac{1}{d-1}\binom{n-2}{d-1}(k-1)^{d-1} \\
&< \frac{2}{d-1}\binom{(n-1)-1}{d-1}(k-1)^{d-1} + \frac{1}{d-1}\binom{n-2}{d-1}(k-1)^{d-1} \\
&= \frac{3}{d-1}\binom{n-2}{d-1}(k-1)^{d-1} \\
&= \frac{3d}{(d-1)(n-1)}\binom{n-1}{d}(k-1)^{d-1} \\
&\leq \frac{2}{d}\binom{n-1}{d}(k-1)^d.
\end{aligned}$$

To show that the last inequality is correct we prove that $\frac{3d}{(d-1)(n-1)} \leq \frac{2}{d}(k-1)$ for $n \geq 5$. By multiplying both sides by $\frac{d}{k-1}$ we get $\frac{3d^2}{(d-1)(n-1)(k-1)} \leq 2$. The LHS of this inequality is maximized when we maximize $d = n-2$ and minimize $k = 3$. By substituting these values and rearranging we get:

$$\begin{aligned}
3(n-2)(n-2) &\leq 4(n-1)(n-3) \\
0 &\leq 4(n^2-4n+3) - 3(n^2-4n+1) \\
0 &\leq n(n-4).
\end{aligned}$$

This equality is true for $n \geq 4$. □

LEMMA 14 *For $2 \leq d \leq n/2$ and $k = 2$:*

$$\sum_{i=1}^{d-2} \frac{1}{i}\binom{n-d+i}{i}(k-1)^i < \frac{2}{d-1}\binom{n-1}{d-1}(k-1)^{d-1}.$$

PROOF: We prove the lemma by induction on d. Let

$$S_k(n,d) = \sum_{i=1}^{d-2} \frac{1}{i}\binom{n-d+i}{i}(k-1)^i.$$

Basis: $d = 2$ or 3, $n \geq 3$. Observe that this covers all cases for $n = 3, 4, 5, 6, 7$:

$$\begin{aligned} d = 2 \qquad & S_k(n,2) = 0 < 2(n-1)(k-1) \\ d = 3 \qquad & S_k(n,3) = (n-2)(k-1) < \binom{n-1}{2}(k-1)^2 \end{aligned}$$

Assume: $S_k(n,d) < \frac{2}{d-1}\binom{n-1}{d-1}(k-1)^{d-1}$ for $1 < d < n/2$ and $n \geq 5$. From the proof of the previous lemma we know:

$$\begin{aligned} S_k(n,d+1) &< \frac{3d}{(d-1)(n-1)}\binom{n-1}{d}(k-1)^{d-1} \\ &\leq \frac{2}{d}\binom{n-1}{d}(k-1)^d. \end{aligned}$$

To show that the last inequality is correct we prove that $\frac{3d}{(d-1)(n-1)} \leq \frac{2}{d}(k-1)$ for $n \geq 8$. By substituting in the value 2 for k and multiplying both sides by d we get $\frac{3d^2}{(d-1)(n-1)} \leq 2$. The LHS of this inequality is maximized when we maximize $d = \frac{n}{2} - 1$. By substituting this value for d and rearranging the terms we get:

$$\begin{aligned} 3(\frac{n}{2}-1)^2 &\leq 2(\frac{n}{2}-2)(n-1) \\ 0 &\leq 2(\frac{n}{2}-2)(n-1) - 3(\frac{n}{2}-1)^2 \\ 0 &\leq 2(\frac{n^2}{2} - \frac{5n}{2} + 2) - 3(\frac{n^2}{4} - n + 1) \\ 0 &\leq \frac{n^2}{4} - 2n + 1. \end{aligned}$$

By solving this quadratic we see that the inequality holds for $n \geq 8$. □

We now use the previous lemmas to get a simple upper bound on the size of the

computation tree:

$$\begin{aligned} CompTree_k(n,d) &\leq 3\sum_{i=1}^{d-1}\frac{1}{i}\binom{n-d+i}{i}(k-1)^i \\ &= 3\sum_{i=1}^{d-2}\frac{1}{i}\binom{n-d+i}{i}(k-1)^i + \frac{3}{d-1}\binom{n-1}{d-1}(k-1)^{d-1} \\ &< \frac{6}{d-1}\binom{n-1}{d-1}(k-1)^{d-1} + \frac{3}{d-1}\binom{n-1}{d-1}(k-1)^{d-1} \\ &= \frac{9}{d-1}\binom{n-1}{d-1}(k-1)^{d-1}. \end{aligned}$$

Recall that our goal is to prove that the ratio of nodes in the computation tree to the number of strings produced is bounded by a constant. From Lemma 12 we have a lower bound on the number of strings produced:

$$N_k(n,d) > \frac{1}{n}\binom{n}{d}(k-1)^d = \frac{1}{d}\binom{n-1}{d-1}(k-1)^d.$$

Thus the ratio of our computation tree to necklaces produced is:

$$\frac{CompTree_k(n,d)}{N_k(n,d)} < 9\frac{d}{(d-1)(k-1)} \leq 18.$$

Experimentally, this constant is less than 3.

THEOREM 8 *Algorithm* **FixedDensity**(t,p) *for generating fixed density k-ary necklaces is CAT.*

4.4 Another algorithm

This chapter has already developed a CAT algorithm for generating fixed density necklaces. In all cases except when $k = 2$ and $d > n/2$, the algorithm **FixedDensity**(t,p) generates the necklaces in lexicographic order where the lexicographically smallest string is the representative of each equivalence class.

This section addresses this exception by developing an algorithm to generate binary fixed density necklaces (and Lyndon words) for $d > n/2$, in lexicographic order.

where each necklace is the lexicographically smallest string in its equivalence class. The approach taken is similar to the approach taken in the algorithm FixedDensity(t, p), except instead of generating pre-necklaces that end with non-zero characters, we generate pre-necklaces that end with the zero character.

Let q be the desired number of 0's in the necklaces (the density is $n - q$). To generate pre-necklaces of length n with exactly q zeros, we maintain the positions of the 0's as well as the difference between each consecutive zero. The string $a = a_0 a_1 \cdots a_{q-1}$ maintains the position of the 0's and $gaps = g_1 g_2 \cdots g_q$ maintains the gaps between consecutive zeros. Formally, the value $g_i = a_i - a_{i-1}$ for $i = 1, \ldots, q$ and $g_q = n + 1 - a_{q-1}$. For each value t, the string $a_0 \cdots a_{t-1}$ represents a pre-necklace. For example, the string $a = 12569$ represents the pre-necklace 001100110. The corresponding $gaps$ string is 13132 where $n = 10$.

Using these data structures, the string a represents a pre-necklace only if the string $gaps$ is a pre-necklace. Thus, if we have a pre-necklace represented by $a_0 \cdots a_{t-1}$, we can use the string $gaps$ to determine the minimum valid position for the next 0. The value $min = a_{t-1} + g_{t-p}$, where p is the length of the longest Lyndon prefix of $g_1 \cdots g_{t-1}$, represents this minimum position. Since we are restricting the number of zeros in the string, we can also determine a maximum valid position for the next zero. The value $max = n - g_1(q - t)$ represents this maximum position unless $p = 1$, in which case the value must be incremented by one. Since max depends on g_1, it is only valid for $t > 1$. However, since a_0 must be 1, the maximum value for a_1 is $n/q + 1$. The case when $q = 1$ must be handled separately to generate the single necklace 01^{n-1}.

Pseudocode for the algorithm (when $q > 1$) just described is given in Figure 4.5. The value a_0 is initialized to 1 and the function FixedZero(2,1) must be called for each possible value for a_1 ($g_1 = a_1 - 1$), where a_1 ranges from 2 to $n/q + 1$. The function PrintIt(p) is similar to before except the value n is replaced with q.

When the value $q \leq n/2$ (ie., $d \geq n/2$), experimental evidence indicates that this algorithm is CAT. This leads to the following conjecture.

```
procedure FixedZero ( t, p : integer ):
local j, min, max : integer;
begin
    if t = q then begin
        g_t := n + 1 - a_{t-1};
        if g_t > g_{t-p} then PrintIt(d);
        else if g_t = g_{t-p} then PrintIt(p);
    end else begin
        min := a_{t-1} + g_{t-p};
        max := n - g_1(q - t);
        if p = 1 then max := max + 1;
        for j ∈ {min, min + 1, ..., max} do begin
            a_t := j;
            g_t := j - a_{t-1};
            if j = min then FixedZero(t + 1, p);
            else FixedZero(t + 1, t);
end; end; end;
```

Figure 4.5: Algorithm for generating necklaces with q zeros

CONJECTURE 1 *Algorithm FixedZero(t, p) for generating necklaces where the number of zeros is less than $n/2$ is CAT.*

This algorithm has application in the following chapter when we look at necklaces where the number of each alphabet symbol is fixed.

4.5 An application

As an application, we embed our fixed density necklace algorithm FixedDensity(t, p) into a program which generates difference covers. A set $D = \{a_1, \cdots, a_d\}$, $1 < a_i < n$, is called a (n, d) difference cover if for every $t \not\equiv 0 \bmod n$ there exists an ordered pair (a_i, a_j) in D such that $a_i - a_j = t \bmod n$. For example, the set $\{1,2,3,6\}$ is a $(10, 4)$ difference cover. A (n, d) difference cover is minimal if a $(n, d-1)$ difference cover does not exist. Another algorithm for generating difference covers is presented in [29].

To generate all difference covers (n, d) we generate all fixed density necklaces $\mathbf{N}_2(n, d)$ where the position of each one in the necklace represents a number in the

set D. To determine whether the necklace represents a difference cover, we keep track of information about each ordered pair. This additional work takes at worst case $O(d)$ time for every node in the computation tree. Thus the overall running time for generating all the (n,d) difference covers is $O(d \cdot N_2(n,d))$.

In practice, it is useful to know whether or not a (n,d) difference cover exists. When n gets large the search space may become infeasible to work with; however, if we have some intuition about what the first few numbers may be in the set D, we can customize our algorithm to drastically reduce the search space. Using this strategy we were able to prove the existence of a (131,13) difference cover, namely

$$\{1, 8, 27, 33, 34, 44, 57, 71, 73, 79, 88, 91\}.$$

Chapter 5

Fixing the Number of each Alphabet Symbol

This chapter discusses two algorithms for generating necklaces where the number of each alphabet symbol is fixed. The first algorithm is simple, but does not appear to run in constant amortized time. The second algorithm uses a new canonical form to represent each equivalence class. Experimental evidence indicates that it is CAT. This time bound is proved in a special case.

5.1 Background

The last chapter on fixed density necklaces introduced the notion of necklaces and Lyndon words where the number of each alphabet symbol is fixed. Recall that $\mathbf{N}(n_0, n_1, \ldots, n_{k-1})$ denotes the set of necklaces composed of n_i occurrences of the symbol i, for $i = 0, 1, \ldots, k-1$. The cardinality of this set is denoted $N(n_0, n_1, \ldots, n_{k-1})$. Similarly, the set of all Lyndon words composed of n_i occurrences of the symbol i is denoted by $\mathbf{L}(n_0, n_1, \ldots, n_{k-1})$, with cardinality $L(n_0, n_1, \ldots, n_{k-1})$. Enumeration formulas for these objects was given in Theorem 7.

Notice that when $k = 2$, the set $\mathbf{N}(n_0, n_1)$ is equivalent to $\mathbf{N}_2(n, d)$ where $n = n_0 + n_1$ and $d = n_1$.

```
procedure SimpleFixedAlph ( t, p : integer );
local j : integer;
begin
    if a_{t-1} = -1 then return;
    if t > n then PrintIt( p )
    else for j ∈ {a_{t-p}, ..., k-2, k-1} do begin
            a_t := j;
            n_j := n_j - 1;
            if j = a_{t-p} then SimpleFixedAlph( t+1, p );
            else SimpleFixedAlph( t+1, t );
            n_j := n_j + 1;
end; end; end;
```

Figure 5.1: A simple algorithm to generate $\mathbf{N}(n_0, n_1, \ldots, n_{k-1})$

n where $k = n/2$ and $n_i = 2$	Number of necklaces	Ratio of work done to necklaces generated
4	2	8.0
6	16	10.4
8	318	11.5
10	11352	12.7
12	623760	14.1
14	18648960	15.7

Table 5.1: Experimental results for SimpleFixeAlph(t, p)

5.2 A simple algorithm

Necklaces where the number of each alphabet symbol is fixed can be generated in lexicographic order by making two simple modifications to Necklace(t, p). First, the value n_i is decremented when the character i is added to the pre-necklace being generated. Second, if the character i is appended when $n_i = 0$, then the recursion is terminated at that node. Pseudocode for such an algorithm is shown in Figure 5.1.

This algorithm is very simple; however, experimental results indicate that the algorithm in not CAT. In particular, Table 5.1 shows the increasing ratios of the work done compared to the number of necklaces generated where each of the n_i equals 2. The work done is computed by counting the number of recursive calls made.

α	β
12023103	37162458
20231031	26581347
02310312	15472836
23103120	48361725
31031202	37256814
10312023	26145738
03120231	15384627
31202310	48273516

Table 5.2: An equivalence class illustrating the β strings

5.3 A fast algorithm

In the previous section, a simple algorithm was developed to generate necklaces where the number of each alphabet symbol is fixed. However, because the algorithm did not appear to run in constant amortized time, we consider another algorithm. This algorithm uses another canonical form to represent the equivalence classes.

5.3.1 A new canonical form

Until now, it has been assumed that the representative of each equivalence class of strings under rotation was the lexicographically smallest element. In this subsection, we introduce a new canonical form to represent the necklaces.

Let $\alpha = a_1 a_2 \cdots a_n$ be a k-ary string in an equivalence class of strings under rotation. Let pos_i be the increasing sequence composed of the positions (or indexes) for all occurrences of the value i in α. Now let the string $\beta = pos_0 pos_1 \cdots pos_{k-1}$. We define a new canonical representative of each equivalence class of strings under rotation to be the string α that yields the lexicographically smallest string β.

As an example, consider the equivalence class containing the string $\alpha = 12023103$. Table 5.2 illustrates each rotation of α along with its corresponding β string. From the figure, the representative of this equivalence class is 03120231.

Before we develop a generation algorithm using this canonical form, we first outline a linear time verification algorithm for determining whether or not a string α is in

canonical form.

5.3.2 A verification algorithm

A naïve method for determining if α is in canonical form is to compare its β string with the β string of all other strings in its equivalence class. Such an algorithm would take worst case time $O(n^2)$. The following algorithm runs in linear time.

By the definition of the canonical form, we see that the positions of the minimum value in the string α are the most critical. If v^* is the minimum value, then we consider the string $pos_{v^*} = q_1 q_2 \cdots q_t$, where there are t occurrences of the value v^* in α. In order for α to be in canonical form then q_1 must equal 1. If q_1 had any other value, then there would exist a rotation of α such that $q_1 = 1$. This would yield a smaller pos_{v^*} string, and thus a smaller β string. Now consider the modified string $pos'_{v^*} = r_1 r_2 \cdots r_t$, where $r_i = q_{i+1} - q_i$ for $i = 1, 2, \ldots, t-1$ and $r_t = n - q_t + 1$. If the string pos'_{v^*} is a necklace (using lex-minimal string as representative), then the original string pos_{v^*} will be the lexicographically smallest string when compared to all other pos_{v^*} strings resulting from other strings in α's equivalence class. Furthermore, if pos'_{v^*} is a Lyndon word, then α will be the unique string in its equivalence class to yield the string pos_{v^*}, and thus it is in canonical form. If pos'_{v^*} is not a necklace, then we can find a rotation of the string α such that a smaller string pos_{v^*} can be obtained, implying that α is not in canonical form. As an example to the above strategy, consider the string $\alpha = 363546378$. Since the minimum value is 3, we get $pos_3 = 137$ and $pos'_3 = 243$. Because pos'_3 is a Lyndon word, α is in canonical form.

Using the strategy just described, we can determine whether or not a string α is in canonical form unless the string pos'_{v^*} is a periodic necklace. If pos'_{v^*} has length t and period p, and setting $p' = np/t$, then the rotations of the string α starting at positions $p'+1, 2p'+1, \ldots, n-p'+1$ will all yield the same string pos_{v^*}. In this case, we must continue examining α's corresponding β string. We update the value v^* to the next smallest value found in α, and focus on the new string pos_{v^*}. Observe that we can no longer employ the same strategy as before, since the starting points for the rotations of α have been restricted. Of these remaining strings, for α

to be in canonical form, it must have the lexicographically smallest string pos_v. To determine this, we modify the string pos_v in the following manner. First, the values $p'+1, 2p'+1, \ldots, n-p'+1, n+1$ are inserted into pos_v so the string is still in sorted order. Then each value j is replaced with $(j-1) \bmod p'$. Finally, we replace all 0's (which were originally the values $p'+1, 2p'+1, \ldots, n+1$) with p'. As before, if the resulting string, denoted pos'_v, is a Lyndon word, then α is in canonical form. If pos'_v is a periodic necklace with period p and length t, then we repeat this procedure with the next v, updating p' to np/t. If pos'_v is not a necklace, then α is not in canonical form. Following this procedure, if v ever exceeds $k-1$, then the the string α is in canonical form and has period equal to the last updated value for p'.

As an example to the above verification strategy, consider the following string with length 12: $\alpha = 021342021432$. We want to determine if α is in canonical form. First we consider $pos_0 = 1\ 7$ and $pos'_0 = 6\ 6$. Since pos'_0 is a periodic necklace we must consider $pos_1 = 3\ 9$, with $p' = 6$. To modify pos_1, we insert the value 7 and 13 to get the string 3 7 9 13. Next, we replace each value j with $(j-1) \bmod 6$ to get 2 0 2 0. Finally, we replace the 0's with 6 to get the new string $pos'_1 = 2\ 6\ 2\ 6$. Since this is a periodic necklace, we must repeat this procedure for the string pos_2 updating $p' = 6$. We now consider $pos_2 = 2\ 6\ 8\ 12$, and perform the modifications to get $pos'_2 = 1\ 5\ 6\ 1\ 5\ 6$. Again we have a periodic necklace and again p' gets updated to 6. Now we must consider $pos_3 = 4\ 11$. In this case $pos'_3 = 3\ 6\ 4\ 6$. Since this is a Lyndon word, the string α is in canonical form.

In the worst case, this verification algorithm must analyze each string pos'_v for $v = 0, 1, \ldots, k-1$. We can determine if pos'_v is a necklace or a Lyndon word in linear time (see Chapter 2.3). Therefore, an upper bound for the running time of the algorithm is proportional to $\sum_{v=0}^{k-1} |pos'_v|$. Observe that length of each string pos'_v is at most $|pos_v| + |pos_{v-1}|$. Thus, since $\sum_{v=0}^{k-1} |pos_v| = n$, the verification algorithm runs in time $O(n)$.

5.3.3 A generation algorithm

Using the observations made in the verification algorithm, we outline a new algorithm for generating necklaces where the number of each alphabet symbol is fixed. Pseudocode is not provided for this algorithm.

The general approach of the algorithm is to fill the string a with the value v starting with the smallest value v^* so that the corresponding pos'_v is a necklace (with lex-minimal as representative). If pos'_v is a Lyndon word, then a will be in canonical form, no matter how the remaining values are placed. If pos'_v is a periodic necklace, then the value v is incremented and the next string pos'_v is generated. This approach suggests three different subroutines: one to generate pos'_{v^*}; one to generate pos'_v for $v > v^*$; and one to arbitrarily place the remaining values.

The first problem of generating the string pos'_{v^*}, where the smallest value $v^* = 0$, is equivalent to generating binary necklaces (fixed density for this application). In this case, instead of printing the resulting necklace pos'_0, we continue generation of the string a. If the necklace pos'_0 is a Lyndon word then we call the routine to arbitrarily place the remaining values. Otherwise we call the routine to generate pos'_v for $v > v^*$.

The task of generating pos'_v for $v > 0$ presents a new and intersting problem. It is equivalent to generating binary strings (fixed density for this application) of length m with equivalence under rotations of length r. Thus, the strings can be thought of as being divided into m/r components, each of length r. An algorithm has been developed to generate these strings, however, a formal explanation is omitted due to the complexity of the algorithm.

Once we generate a string pos'_v that is a Lyndon word, we can arbitrarily place the remaining values into the remaining positions. This problem is equivalent to generating permutations of a multiset. A CAT algorithm for this task is outlined in [22]. It requires that all available positions are pre-assigned the largest value $k - 1$, and a linked list must be used to maintain the values v such that $n_v > 0$.

Each of the preceding sub-routines have been combined to form an algorithm which generates necklaces where the number of each alphabet symbol is fixed. Notice

that this approach can also be used to generate unrestricted necklaces by removing the fixed density restrictions on the first two routines. The algorithm has been implemented and experimental evidence indicates that it runs in constant amortized time. Details of the algorithm have been omitted.

In general, no analysis has been attempted for the developed algorithm. However, consider the special case where there exists an n_i relatively prime to n. Such a value will always exist if n is prime. In this case, since the labeling of the symbols is arbitrary, swap n_i with n_0. Now, the first routine to generate pos'_0 will always generate a Lyndon word. This means that no calls to the second routine to generate pos'_t for $t > 0$ will be required. Now since the Lyndon words pos'_0 can be generated in constant amortized time using FixedDensity(t, p) and because the placement of the remaining values can also be done in constant amortized time, the result is a CAT algorithm for generating necklaces where the number of each alphabet symbol is fixed. Note, that since the algorithm FixedDensity(t, p) complements the output for $n_0 < n/2$, the generated necklaces may not be in the canonical form described. However, if the algorithm FixedZero(t, p) is used when $n_0 < n/2$, then the canonical form will be preserved. The latter algorithm however, is only conjectured to run in constant amortized time.

Chapter 6

Chord Diagrams

This chapter outlines two algorithms for generating chord diagrams. Experimental results indicate that the latter of the two algorithms runs in constant amortized time. In addition, the previously known formula for enumerating chord diagrams is derived using simple counting techniques.

6.1 Background

An *undirected graph* G consists of a set $V(G)$ of vertices and a set $E(G)$ of edges where each edge e in $E(G)$ is associated with an unordered pair (u, v) of vertices from $V(G)$. If (u, v) is an edge in $E(G)$, then we say: u and v are *endpoints* of e; the vertices u and v are *adjacent*; and e is *incident* to u and v. A *chord diagram* is an undirected graph with $2n$ vertices and n edges (chords) where the vertices are embedded on an oriented (counter-clockwise) circle and each vertex is adjacent to exactly one other vertex (ie. each vertex is the endpoint of exactly one edge). Figure 6.1 illustrates a chord diagram with 4 chords. Chord diagrams are isomorphic under rotation of the vertices embedded on the circle.

Recently, there have been several papers published that discuss chord diagrams. A primary reason for this is because chord diagrams were shown to have application in the context of Vassiliev invariants, which in turn have application in knot theory [2]. A related object called a linearized chord diagram is studied by Stoimenow in

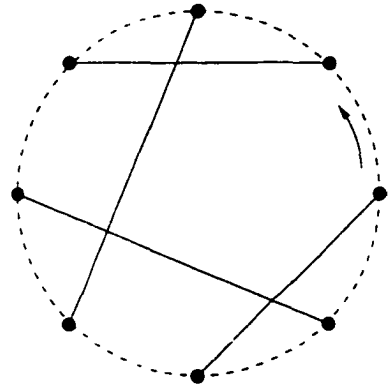

Figure 6.1: Chord diagram with 4 chords

[24], and braided chord diagrams are discussed by Birman and Trapp in [5].

Three independent papers by Li and Sun [16], Cori and Marcus [8], and Stoimenow [25], have derived enumeration formulas for the number of of non-isomorphic chord diagrams. In each of these papers, the exact formula is the main result; however, in each case the derivation of the formula uses relatively complex methods. Cori and Marcus use Burnside's lemma along with liftings of quasi-diagrams, Li and Sun introduce a new object called a generalized m-configuration, and Stoimenow uses Burnside's lemma along with two new objects: linerized chord diagrams and generalized linearized chord diagrams. As a secondary result in this chapter, we derive an exact formula for the number of non-isomorphic chord diagrams with n chords using simple counting techniques.

6.2 Enumerating non-isomorphic chord diagrams

The bifactorial of an integer n, denoted $n!!$, is defined by the following formula:

$$n!! = \begin{cases} \prod_{j=0}^{\lfloor \frac{n-1}{2} \rfloor} (n-2j) & \text{if } n > 0. \\ 1 & \text{if } n = 0 \text{ or } n = -1. \\ 0 & \text{if } n \leq -2. \end{cases}$$

Using this notation, it is easy to see that the number of chord diagrams with n chords is $(2n-1)!!$.

The set of all chord diagrams with n chords is partitioned into equivalence classes by the cyclic group $\mathbb{C}_{2n}$. Two chord diagrams are isomorphic if one can be obtained by some rotation of the other. If we let σ denote a single rotation, then the group elements of $\mathbb{C}_{2n}$ are σ^j for $j = 1, 2, \ldots, 2n$. To count the number of non-isomorphic chord diagrams with n chords, denoted $C'(n)$, we apply Burnside's lemma:

$$C'(n) = \frac{1}{2n} \sum_{j=1}^{2n} |Fix(\sigma^j)|.$$

The number of chord diagrams fixed by σ^j depends only on the order of σ^j. In other words, if two group elements σ^j and σ^k have the same order, then the set of chord diagrams fixed by each group element will be the same. The number of elements of $\mathbb{C}_{2n}$ with order p (where $p|2n$) is $\phi(p)$. Thus, if we let $T(2n, p)$ denote the number of chord diagrams with n chords fixed by a group element of order p (namely σ^j) then

$$C'(n) = \frac{1}{2n} \sum_{pq=2n} \phi(p) T(2n, p).$$

We now derive a formula for $T(2n, p)$ by deriving recurrence equations for two cases: p odd and p even. We start by labeling the vertices on a chord diagram from 1 to $2n$ in counter-clockwise order around the circle. With this labeling, we define the *length* of a chord starting from vertex i and ending at vertex j to be $(j - i) \bmod 2n$. We now consider the chords starting at vertices $q, 2q, \ldots, pq$, where $pq = 2n$. If a chord diagram is fixed by σ^j, then the length of the chords starting at these positions must be the same. If p is odd then there are $2n - p$ possible lengths for the chords, since it is impossible for two vertices in the list $q, 2q, \ldots, pq$ to be joined together (see Figure 6.2(a)). If we now ignore these chords, we are reduced to the problem of counting $T(2n - 2p, p)$. We arrive at the following recurrence relation if p is odd:

$$T(2n, p) = (2n - p) T(2n - 2p, p).$$

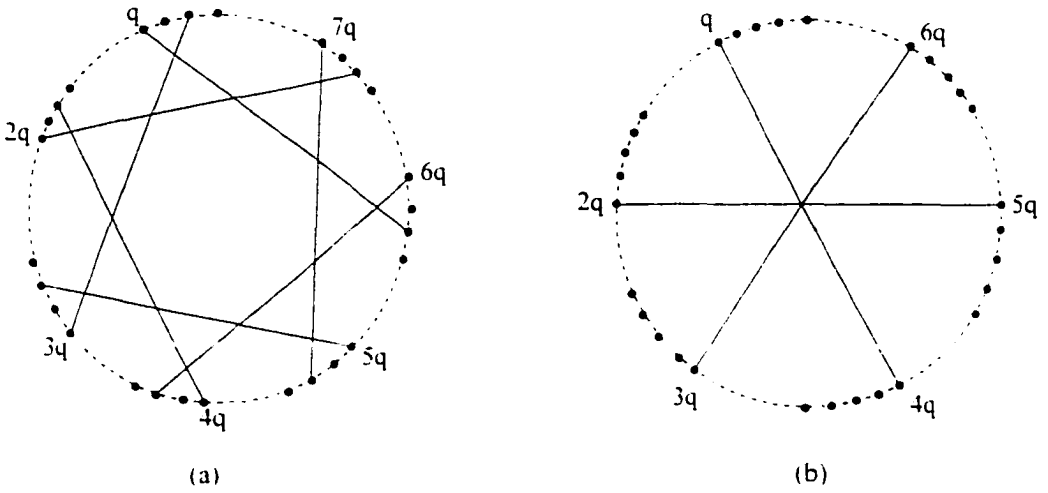

Figure 6.2: (a) One of the $2n - p$ possible lengths for the chords starting at $q, 2q, \ldots, pq$. (b) For p even, there is only one choice for the endpoint landing back in the list $q, 2q, \ldots, pq$

In the base case, $T(0, p) = 1$.

If p is even, then there is the additional possibility that the chords have length n (see Figure 6.2(b)). Thus each vertex in the list $q, 2q, \ldots, pq$ can be joined to exactly one other vertex in the list or to any of the other $2n - p$ vertices as before. We obtain the following recurrence relation if p is even:

$$T(2n, p) = (2n - p)T(2n - 2p, p) + T(2n - p, p).$$

In the base cases, $T(p, p) = T(0, p) = 1$.

Solving the two recurrence equations yields the following exact formula:

$$T(2n, p) = \begin{cases} (q-1)!! \cdot p^{q/2} & \text{if } p \text{ odd,} \\ \sum_{j=0}^{\lfloor \frac{q}{2} \rfloor} \binom{q}{2j} \cdot (2j-1)!! \cdot p^j & \text{if } p \text{ even.} \end{cases}$$

Proof by induction on q will verify the correctness of the above result.

6.3 Representing chord diagrams

There are numerous ways to represent chord diagrams. Several objects equivalent to our definition of chord diagrams have been studied by other authors, including polygons where the sides are identified pairwise [26], and one-vertex maps [15]. We choose to ignore these other representations and focus on new string representations.

One of the most natural ways to represent a chord diagram is to first assign each chord a unique value from 1 to n, and then label the vertices with the value of their incident chord. If we arbitrarily pick a starting vertex s, then we obtain a string representation by recording the vertex values starting at vertex s and moving counter-clockwise (by convention) around the circle. In this manner, any string with length $2n$ containing exactly two occurrences of the values 1 through n can be used to represent a chord diagram. An example of this string representation is shown in Figure 6.3(a). Such string representations have equivalence under string rotation and permutation of the alphabet symbols 1 through n. Thus, there may be up to $2n(n!)$ strings in each equivalence class. These strings are equivalent to unlabeled necklaces where the number of each alphabet symbol is 2. Earlier, we developed an algorithm for generating binary unlabeled necklaces, but there is no known fast solution for generating unlabeled necklaces in the general case.

Because no efficient generation algorithm currently exists using this first string representation, we consider another approach. First, we assign each vertex a position from 1 to $2n$ by starting at a vertex s and moving counter-clockwise around the circle. We then label each vertex i with the length of the chord starting at vertex i. Recall that the length of a chord starting at vertex i and ending at vertex j is $(j-i) \bmod 2n$. In this case, the label given to each vertex is independent of the starting vertex s; however, there is a dependency between each pair of vertices joined by a chord; if two vertices are adjacent then the sum of their values equals $2n$. This is easily seen by the definition of length. Again, we obtain the string representation by recording the vertex values starting at vertex s and moving counter-clockwise around the circle. In this second string representation, we no longer have equivalence under permutation

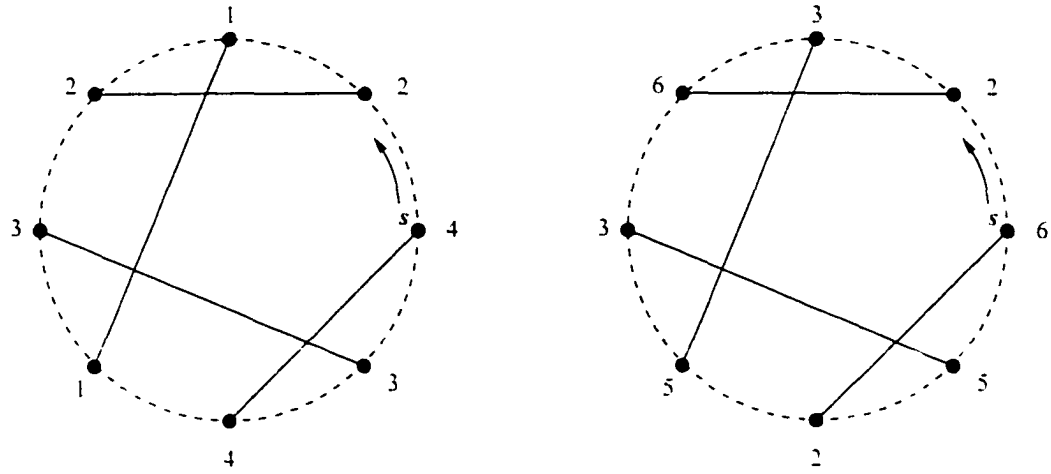

Figure 6.3: Two string representations: (a) label chords then vertices (b) label vertices by chord length

of the alphabet symbols, and the number of each alphabet symbol is no longer fixed; however, the size of the alphabet has increased to $2n-1$ from n. An example of this string representation is given in Figure 6.3(b).

In each of the following two sections, we present an algorithm for generating non-isomorphic chord diagrams. Both algorithms use the second of two string representations outlined in this section; however, each algorithm uses a different representative for each equivalence class. Both algorithms also make use of the necklace generation algorithm Necklace(t, p).

6.4 A simple algorithm

In this section we develop a simple algorithm to list all non-isomorphic chord diagrams with n chords. To represent the chord diagrams, we use the string representation that labeled the vertices according to their chord lengths as described in the previous section. Since we wish to generate only non-isomorphic chord diagrams, we must list exactly one string per equivalence class. Using the lexicographically smallest string as the representative of each equivalence class, we arrive at a problem equivalent to generating length $2n$ necklaces on an alphabet of size $2n-1$, with the added restriction

that each necklace corresponds to a valid chord diagram.

Recall that when generating necklaces, we build up a pre-necklace one character at a time. Applying this to chord diagrams, we instead add one chord or two characters at a time. Thus, if we are adding the value j to the t-th position of the string, then we must also add the value $2n - j$ to the $(t + j)$-th position (due to the nature of the string representation for chord diagrams that we are using). This adds the condition that $t + j \leq 2n$. In addition, we must ensure that we have not already assigned values to positions t and $t + j$ in the pre-necklace. If we have already assigned a value to the t-th position, then we continue generation with position $t + 1$ only if the string $a_1 \ldots a_t$ is a valid pre-necklace. Adding these simple modifications to Necklace(t, p), we ensure that each necklace generated corresponds to a valid chord diagram. The resulting algorithm for generating non-isomorphic chord diagrams in lexicographic order, SimpleChords(t, p), is shown in Figure 6.4. The initial call is SimpleChords$(1, 1)$ and a_0 is initially set to 1. Aperiodic chord diagrams can be generated by replacing the test $2n \bmod p = 0$ with $2n = p$, as was the case with necklaces.

6.4.1 Analysis

Recall that the goal is to develop a generation algorithm which runs in constant amortized time. To see if a mathematical analysis is worthwhile, we first gather some experimental results. To analyze the performance of this algorithm, we compare the amount of work done to the number of chord diagrams generated. Since each recursive call is the result of a constant amount of work, we simply count the number of recursive calls made. This approach is the same as counting the number of nodes in the computation tree. The resulting ratios of work done compared to chord diagrams generated are shown in Table 6.1. Notice that the ratios are steadily increasing as the number of chords increases. This is a strong indication that the algorithm is *not* CAT. For this reason, we attempt no mathematical analysis and focus on developing a faster algorithm.

```
procedure SimpleChords ( t, p : integer ):
local j : integer:
begin
    if t > 2n then
        if 2n mod p = 0 then Print( )
    else begin
        j := a_{t-p}:
        if a_t = 0 and t + j ≤ 2n then begin
            if a_{t+j} = 0 then begin
                a_t := j:   a_{t+j} := 2n - j:
                SimpleChords( t + 1, p ):
                a_{t+j} := 0:
            end:
            for j ∈ {a_{t-p} + 1, ..., 2n - t} do begin
                if a_{t+j} = 0 then begin
                    a_t := j:   a_{t+j} := 2n - j:
                    SimpleChords( t + 1, t ):
                    a_{t+j} := 0:
                end:
            end:
            a_t := 0:
        end:
        else if a_t = a_{t-p} then SimpleChords( t + 1, p ):
        else if a_t > a_{t-p} then SimpleChords( t + 1, t ):
    end:
end:
```

Figure 6.4: A simple algorithm for generating non-isomorphic chord diagrams with n chords

Number of chords n	Non-isomorphic chord diagrams	Ratio of work done to chord diagrams generated
4	18	10.6
5	105	12.8
6	902	14.1
7	9749	15.3
8	127072	16.3
9	1915951	17.3
10	32743182	18.3
11	624999093	19.2
12	13176573910	20.1

Table 6.1: Experimental results for SimpleChords(t, p)

6.5 A fast algorithm

In this section we develop a fast algorithm for generating non-isomorphic chord diagrams. In this algorithm we use the same string representation for chord diagrams, but this time we use the canonical form described in Section 5.3.1. Recall that the verification described in Section 5.3.2 was used as a basis to outline an algorithm for generating necklaces where the number of each alphabet symbol is fixed. We will use the same ideas and notation here to generate chord diagrams.

The general idea behind this new generation algorithm for chord diagrams is to fill the length $2n$ string α with the value v starting with the smallest value v^*, so that the corresponding string pos'_v is a necklace (with lex-minimal as representative). If pos'_v is a Lyndon word, then we fill the remaining positions of α with values greater than v, making sure that the string represents a valid chord diagram. Otherwise, if pos'_v is a periodic necklace, then we must repeat the procedure for $v+1$.

Observe that the only chord diagram with minimum value n is $\alpha = n^{2n}$. Since, we can generate this string separately, we let the minimum value v^* range from 1 to $n-1$. We divide the algorithm outlined above into three separate recursive routines: the first routine GenPos$(s, t, p, v, last)$ generates the necklaces pos'_{v^*}; the second routine GenPos2$(s, t, p, p', v, part)$ generates the necklaces pos'_v for all $v > v^*$; and the third routine GenRest(s, t, v), fills the remaining positions with values that are greater than v. The algorithm FastChords(n), shown in Figure 6.5, uses these routines to generate all non-isomorphic chord diagrams with n chords. To simplify the notation, the i-th element of pos'_v is denoted $pos'_v(i)$. The chord diagram generated is stored in $\alpha = a_0 \cdots a_{2n-1}$.

Due to the dependencies of the string representation, whenever one of these routines adds the value v to position s, it must also assign the value $2n - v$ to position $(s+v) \bmod 2n$. A global linked list is used to keep track of the available positions in α, in increasing order. The variable *head*, is the value of the first available position, and the value $2n$ represents the end of the list. If the list is implemented using an array with next and previous pointers, then the functions Add(s), Remove(s), and

```
procedure FastChords (n):
local i, v* : integer:
begin
    InitList():
    for v* ∈ {1,2,...,n − 1} do pos_v*(0) := 0:
    for v* ∈ {1,2,...,n − 1} do begin
        a_0 := v*;   a_1 := 2n − v*:
        Remove(0): Remove(v*):
        Gen(head, 1, 1, v*, 0):
        GenRest(head, head.next, v*):
        Add(v*): Add(0):
    end:
    for i ∈ {0,1,2,...,2n − 1} do a_i := n:
    Print():
end;
```

Figure 6.5: A fast algorithm for generating non-isomorphic chord diagrams with n chords

Avail(s) can be implemented in constant time. The routine InitList() initializes the list to contain every position from 0 to $2n-1$.

The various details of each recursive function are described in the following subsections.

6.5.1 GenPos($s, t, p, v, last$)

We generate all necklaces pos'_v using the fundamental necklace properties from the previous necklace algorithms. In addition to the usual parameters t and p, we also maintain the position s (in α) and the value v we wish to insert at position s. We also keep track of the position of the last inserted value in the variable $last$. Aside from the usual necklace properties, we can insert the value v into position s as long as the position $(v+s) \bmod 2n$ is available.

At each step the string $pos'_v = q_1 q_2 \cdots q_{t-1}$ is a pre-necklace. The smallest valid position such that $q_1 q_2 \cdots q_t$ is also a pre-necklace is equal to $last + q_{t-p}$ and is stored in the variable min. However, from the discussion in the verification algorithm (see Chapter 5.3.2), the string we desire has the value $2n - last$ appended to pos'_v. Thus,

```
procedure GenPos ( s,t,p,v,last: integer ):
local next,e,min,p2 : integer;
begin
    min := last + pos'_v(t − p);
    if min = 2n and t mod p = 0 then begin
        if t = n then Print( );
        else GenPos2(head, 1, 1, 2n · p/t, v + 1, 0);
    end;
    else if min < 2n and s < 2n then begin
        e := (s + v) mod 2n;
        if s ≥ min and Avail(e) then begin
            next := s.next;
            if next = e then next := e.next;
            a_s := v;   a_e := 2n − v;
            Remove(s);   Remove(e);
            pos'_v(t) := s − last;
            p2 := p;
            if s ≠ min then p2 := t;
            GenPos(next, t + 1, p2, v, s);
            if s + pos'_v(t + 1 − p2) < 2n then GenRest(head, head.next, v);
            Add(e);   Add(s);
        end;
    GenPos(s.next, t, p, v, last);
end; end;
```

Figure 6.6: GenPos($s, t, p, v, last$)

if $min < 2n$, then the string with the appended value is a Lyndon word. If $min = 2n$ and $t \bmod p = 0$ then the modified string is a periodic necklace. Pseudocode for the resulting procedure GenPos($s, t, p, v, last$) is shown in Figure 6.6.

6.5.2 GenPos2($s, t, p, p', v, part$)

To generate the strings pos'_v for values $v > v^*$ we again use the basic necklace properties. This time the string being generated has a slightly different construction. In this case, we must maintain the values p' (as described in the verification algorithm), v, and another value *part* which is used when we add the additional values to pos'_v (as described in the verification algorithm). Unlike the verification algorithm, however,

we do not place all the additional values at once. Instead, we place them incrementally as we pass the positions equal to the value that is to be inserted. Thus, the first time the value s is greater than p', we add the value p' to the string pos'_v. The variable min is used in a similar manner as with the previous method GenPos($s, t, p, v, last$).

The one special case that must be considered, is the case when $v = n$. In this case we do not allow the starting vertex s to be greater than n because the resulting string is equivalent to placing the value in position $n - s$. Pseudocode for the resulting routine GenPos2($s, t, p, p', v, part$) is shown in Figure 6.7.

6.5.3 GenRest(s, e, v)

The routine GenRest(s, e, v) is a simple recursive procedure that fills the remaining available positions in a with values greater than v. It starts with the first available position s, and for each available position e, attempts to assign the value $e - s$ to position s and the value $2n - e + s$ to position e. Such assignments are valid as long as the values are greater than v. Once all the positions are filled, the string is printed. Pseudocode for this routine is shown in Figure 6.8.

6.5.4 Analysis

As with the previous algorithm, we obtain experimental results for the amount of work done compared to the number of chord diagrams generated. Again, since the work done for each recursive call is constant, we count the amount of work done by summing the number of recursive calls. The resulting ratios are shown in Table 6.2. Notice that the ratios are decreasing (after $n = 5$) as the number of chords increases. This gives a very strong indication that the algorithm runs in constant amortized time.

CONJECTURE 2 *Algorithm FastChords(n) for generating non-isomorphic chord diagrams is CAT.*

A mathematical proof for the conjecture is very difficult because there seems no easy way to count, or bound, the number of nodes at each level of the computation tree.

```
procedure GenPos2 ( s, t, p, p', v, part: integer ):
local next, e, min: integer:
begin
    min := pos'_v(t - p);
    if v = n and s > n then begin
        if head = 2n then Print( ):
    end:
    else if t = 1 and s > p' then begin
        if v < n then GenPos2(head, 1, 1, p', v + 1, 0):
    end:
    else if s > p'(part + 1) then begin
        pos'_v(t) := p':
        if min = p' then GenPos2(s, t + 1, p, p', v, part + 1):
        else GenPos2(s, t + 1, t, p', v, part + 1):
    end:
    else if s = 2n then begin
        if head = 2n then Print( ):
        else if min ≠ p' then GenRest(head, head.next, v):
        else if t mod p = 0 then GenPos2(head, 1, 1, 2n - p/t, v + 1, 0):
    end:
    else begin
        e := (s + v) mod 2n:
        if s mod p' ≥ min and Avail(e) then begin
            next := s.next:
            if next = e then next := e.next:
            a_s := v;   a_e := 2n - v;
            Remove(s):   Remove(e);
            pos'_v(t) := s mod p':
            if s mod p' = min then GenPos2(next, t + 1, p, p', v, part):
            else GenPos2(next, t + 1, t, p', v, part):
            Add(e):   Add(s):
        end:
        GenPos2(s.next, t, p, p', v, part):
end: end:
```

Figure 6.7: GenPos2($s, t, p, p', v, part$)

```
procedure GenRest ( s, e, v : integer );
begin
    if s = 2n then Print();
    else if e ≠ 2n then begin
        if e − s > v and 2n − e + s > v then begin
            a_s := e − s;   a_v := 2n − a_s;
            Remove(s);  Remove(e);
            GenRest(head.head.next.v);
            Add(e);   Add(s);
        end;
        if 2n − e.next + s > v then GenRest(s,e.next,v);
end; end;
```

Figure 6.8: GenRest(s, e, v)

Number of chords n	Non-isomorphic chord diagrams	Ratio of work done to chord diagrams generated
4	18	13.4
5	105	14.0
6	902	12.4
7	9749	10.9
8	127072	9.8
9	1915951	9.4
10	32743182	8.9
11	624999093	8.5
12	13176573910	8.2

Table 6.2: Experimental results for FastChords(n)

Chapter 7

Forbidden Substrings

The problem of generating strings with forbidden substrings is naturally related to the classic pattern matching problem, which takes as input a pattern P of length m and a text T of length n, and finds all occurrences of the pattern in the text. Several algorithms perform this task in linear time, $O(n+m)$, including the Boyer-Moore algorithm, the Knuth-Morris-Pratt (KMP) algorithm and an automata-based algorithm (which requires non-linear initialization) [14].

The Boyer-Moore algorithm is not suitable for our purposes since it does not operate in real-time. This means that if we are generating a string (piece of text) one character at a time, then the Boyer-Moore algorithm will not perform its checks in constant time per character. On the other hand, the automata-based algorithm operates in real-time and the KMP algorithm can be adapted to do so [14].

Using the automata-based pattern matching algorithm, this chapter develops CAT algorithms to generate:

(a) all k-ary strings of length n that have no substring equal to f,

(b) all k-ary circular strings of length n that have no substring equal to f, and

(c) all k-ary necklaces of length n that have no substring equal to f, where f is a Lyndon word.

Each algorithm generates the string from left-to-right, applying the pattern match-

ing as each character is generated. In the unrestricted case, the generation algorithm is straight forward, however, when the pattern is taken circularly, the algorithm and its analysis become more complicated. In the necklace case, the algorithm Necklace(t, p) is used.

In the following section we provide background and definitions for these objects along with a brief description of the automata-based pattern matching algorithm. In Section 7.2, we outline the details of each algorithm. We analyze the algorithms, proving that they run in constant amortized time, in Section 7.3.

7.1 Background

We denote the set of all k-ary strings of length n with no substring equal to f by $\mathbf{I}_k(n, f)$. The cardinality of this set is $I_k(n, f)$. For the remainder of this chapter we will assume that the forbidden string f has length m. Clearly if $m > n$, then $I_k(n, f) = k^n$, and if $m = n$ then $I_k(n, f) = k^n - 1$. If $m < n$, then an exact formula will depend on the forbidden substring f, but can be computed using the transfer matrix method [23]. In Section 7.3 we derive several bounds on the value of $I_k(n, f)$.

We denote the set of all k-ary circular strings of length n with no substring equal to f by $\mathbf{C}_k(n, f)$. The cardinality of this set is $C_k(n, f)$. In this case, we allow the forbidden string to make multiple passes around the circular string. Thus, if a string α is in $\mathbf{I}_k(n, f)$ and $m > n$, then it is still possible for the string α to contain the forbidden string f. For example, if $\alpha = 0110$ and $f = 11001100$, then α is *not* in the set $\mathbf{C}_k(4, f)$. We prove that $C_k(n, f)$ is proportional to $I_k(n, f)$ in Section 7.3.2.

The set of all k-ary necklaces of length n with no substring equal to f is denoted $\mathbf{N}_k(n, f)$ and has cardinality $N_k(n, f)$. The set of all k-ary Lyndon words of length n with no substring equal to f is denoted $\mathbf{L}_k(n, f)$ and has cardinality $L_k(n, f)$. For $N_k(n)$ and $L_k(n)$ the strings are considered to be circular when avoiding f. The set of all k-ary pre-necklaces of length n with no substring equal to f is denoted $\mathbf{P}_k(n, f)$ and has cardinality $P_k(n, f)$. A standard application of Burnside's Lemma will yield

the following formula for $N_k(n, f)$:

$$N_k(n, f) = \frac{1}{n} \sum_{d|n} \phi(d) C_k(n/d, f). \tag{7.1}$$

By applying Möbius inversion we obtain a formula for $L_k(n, f)$:

$$L_k(n, f) = \frac{1}{n} \sum_{d|n} \mu(d) C_k(n/d, f). \tag{7.2}$$

7.1.1 The automata-based string matching algorithm

One of the best tools for pattern recognition problems is the finite automaton. If $f = f_1 f_2 \cdots f_m$ is the pattern we are trying to find in a string α, then a deterministic finite automaton can be created to process the string α one character at a time, in constant time per character. In other words, we can process the string α in *real-time*. The preprocessing steps required to set up such an automaton can be done in time $O(km)$, where k denotes the size of the alphabet (see [1], pg. 331).

The automaton has $m + 1$ states, which we take to be the integers $0, 1, \ldots, m$. The state represents the length of the current match. Suppose we have processed t characters in the string $\alpha = a_1 a_2 \cdots a_n$ and the current state is s. The transition function $\delta(s, j)$ is defined so that if $j = a_{t+1}$ matches f_{s+1}, then $\delta(s, j) = s + 1$. Otherwise, $\delta(s, j)$ is the largest state q such that $f_1 \cdots f_q = a_{t-q+2} \cdots a_{t+1}$. If the automaton reaches state m, the only accepting state, then the string f has been found in α. The transition function is efficiently created using an auxiliary function *fail*. The failure function is defined for $1 \leq i \leq m$ such that *fail*(i) is the length of the longest proper suffix of $f_1 \cdots f_i$ equal to a prefix of f. If there is no such suffix, then *fail*$(i) = 0$. This fail function is the same as the fail function in the classic KMP algorithm.

7.2 Algorithms

In this section we develop an algorithm to generate necklaces avoiding a particular Lyndon substring f. We start by looking at the simpler problem of generating k-ary strings with forbidden substrings and then consider circular strings, focusing on how to handle the wraparound.

If n is the length of the strings being generated and m is the length of the forbidden substring f, then the following algorithms apply for $2 < m \leq n$. In the cases where $m = 1$ or 2, trivial algorithms can be developed. For circular strings and necklaces, if $m > n$, then the forbidden substring can be truncated to a length n string, as long as it repeats in a circular manner after the n-th character.

7.2.1 Generating k-ary strings with forbidden substrings

A naïve algorithm to generate all strings in $\mathbf{I}_k(n, f)$ will generate all k-ary strings of length n, and then upon generation of each string, perform a linear time test to determine whether or not it contains the forbidden substring. A simple and efficient approach for generating strings is to construct a length n string by taking a string of length $n-1$ and appending each of the k characters in the alphabet to the end of the string. This strategy suggests a simple recursive scheme, requiring one parameter for the length of the current string. Since this recursive algorithm runs in constant amortized time, the naïve algorithm will take linear time per string generated.

A more advanced algorithm will embed a real-time automata-based string matching algorithm into the string generation algorithm. Since an automata-based string matching algorithm takes constant time to process each character, we can generate each new character in constant time. We store the string being generated in $a = a_1 a_2 \cdots a_n$ and at each step, we maintain two parameters: t and s. The parameter t represents the next position in the string to be filled, and the parameter s represents the state in the finite automata produced for the string f. Recall that each state s is an integer value that represents the length of the current match. Thus if we begin a recursive call with parameters t and s, then $f_1 \cdots f_s = a_{t-s} \cdots a_{t-1}$.

```
procedure ForbString ( t, s : integer ):
local j, q : integer:
begin
    if t > n then Print()
    else begin
        for j ∈ {0, 1, ..., k − 1} do begin
            a_t := j;
            q := δ(s, j);
            if q ≠ m then ForbString(t + 1, q);
end: end: end:
```

Figure 7.1: An algorithm for generating k-ary strings with no substring equal to f

We continue generating the current string as long as $s \neq m$. When $t > n$, we print out the string using the function Print(). Pseudocode for this algorithm is shown in Figure 7.1. The transition function $\delta(s, j)$ is used to update the state s as described in Section 7.1.1. The initial call is ForbString(1,0).

Following this approach, each node in the computation tree will correspond to a unique string in $\mathbf{I}_k(j, f)$ where j ranges from 1 to n. Since the amount of computation at each node is constant, the total computation is proportional to

$$CompTree_k(n, f) = \sum_{j=1}^{n} I_k(j, f).$$

In Section 7.3.1 we prove that this sum is proportional to $I_k(n, f)$, which proves the following theorem. (Recall that the preprocessing required to set up the automata for f takes time $O(km)$. This amount is negligible compared to the size of the computation tree.)

THEOREM 9 *Algorithm ForbString(t, s) for generating k-ary strings of length n with no substring equal to f is CAT.*

7.2.2 Generating k-ary circular strings with forbidden substrings

We now focus on the more complicated problem of generating circular strings with forbidden substring f. To solve this problem we use the previous algorithm, but now we must also check that the wraparound of the string does not yield the forbidden substring. More precisely, if $\alpha = a_1 a_2 \cdots a_n$ is in $\mathbf{I}_k(n, f)$ then the additional substrings we must implicitly test against the forbidden string f are $a_{n-m+1+i} \cdots a_n a_1 \cdots a_i$ for $i = 1, 2, \ldots, m-1$. To perform these additional tests, we could continue the pattern matching algorithm by appending the first $m-1$ characters to the end of the string. This will result in $m-1$ additional checks for each generated string, yielding an algorithm that runs in time $O(m I_k(n, f))$. This approach can be tweaked to yield a CAT algorithm for circular strings, but leads to difficulties in the analysis when applied in the necklace context.

If we wish to use the algorithm ForbString(t, s), we need another way to test the substrings starting in the last $m-1$ positions of α. We accomplish this feat by maintaining a new boolean data structure $match(i, t)$ and dividing the work into two separate steps. In the first step we compare the substring $a_1 \cdots a_t$ against the last i characters in f. If they match, then the boolean value $match(i, i)$ is set to TRUE; otherwise it is set to FALSE. In the second step, we check to see if $a_{n-m+1+i} \cdots a_n$ matches the first $m-i$ characters in f. If they match and $match(i, i)$ is TRUE, then we reject the string. If there is no match for all $1 \leq i \leq m-1$, then α is in $\mathbf{C}_k(n, f)$.

To execute the first step, we start by initializing $match(i, 0)$ to TRUE for all i from 1 to $m-1$. We define $match(i, t)$ for $1 \leq i \leq m-1$ and $i \leq t \leq m-1$ to be TRUE if $match(i, t-1)$ is TRUE and $a_t = f_{m-i+t}$. Otherwise $match(i, t)$ is FALSE. This definition implies that $match(i, i)$ will be TRUE if $a_1 \cdots a_i$ matches the last i characters of f. Pseudocode for a routine that sets these values for each t is shown in Figure 7.2. The procedure SetMatch(t) is called after the t-th character in the string α has been assigned for $t < m$. Thus, if $t < m$, we must perform additional work proportional to $m - t$ for all strings in $\mathbf{I}_k(t, f)$. We will prove later that this extra

```
procedure SetMatch ( t : integer ):
local i : integer;
begin
    for i ∈ {t, t+1, ..., m-1} do begin
        if match(i, t-1) and f_{m-t+1} = a_t then match(i, t) := TRUE;
        else match(i, t) := FALSE;
end; end;
```

Figure 7.2: Procedure used to set the values of $match(i, t)$

```
function CheckSuffix ( s : integer ) returns boolean;
begin
    while s > 0 do begin
        if match(m-s, m-s) then return(FALSE);
        else s := fail(s);
    end;
    return(TRUE);
end;
```

Figure 7.3: Function used to test the wraparound of circular strings

work will not affect the asymptotic running time of the algorithm.

To execute the second step, we observe that after the n-th character has been generated ($t = n + 1$), the string $a_{n-s+1} \cdots a_n$ is the longest suffix of α to match a prefix of f. Using the array *fail*, as described in Section 7.1.1, we can find all other suffixes that match a prefix of f in constant time per suffix. Then, for each suffix with length j found equal to a prefix of f, we check $match(m-j, m-j)$. If $match(m-j, m-j)$ is TRUE, then α is not in $\mathbf{C}_k(n, f)$. Note that $I_k(n-j, f)$ is an upper bound on the number of strings where $a_{n-j+1} \cdots a_n$ matches a prefix of f. Thus, for each $1 \leq j \leq m-1$ the extra work done is proportional to $I_k(n-j, f)$. Pseudocode for the tests required by this second step is shown in Figure 7.3. The function **CheckSuffix**(s) takes as input the parameter s which represents the length of the longest suffix of α to match a prefix of f. It returns TRUE if α is in $\mathbf{C}_k(n, f)$ and FALSE otherwise.

Following this approach, we can generate all circular strings with forbidden substrings by adding the routines **SetMatch**(t) and **CheckSuffix**(s) to **ForbString**(t, s).

```
procedure ForbCircular ( t, s : integer ):
local j, q : integer:
begin
    if t > n then begin
        if CheckSuffix(s) then Print( ):
    end else begin
        for j ∈ {0, 1, ..., k − 1} do begin
            a_t := j:
            q := δ(s, j):
            if t < m then SetMatch(t):
            if q ≠ m then ForbCircular(t + 1, q):
end: end: end:
```

Figure 7.4: An algorithm for generating k-ary circular strings with no substring equal to f

Pseudocode for the resulting algorithm is shown in Figure 7.4. The initial call is ForbCircular(1,0).

Observe that the size of the computation tree will be the same as before; however, in this case, the computation at each node is not always constant. The extra work performed at these nodes is bounded by

$$ExtraWork_k(n, f) \leq \sum_{j=1}^{m-1}(m-j)I_k(j, f) + \sum_{j=1}^{m-1} I_k(n-j, f).$$

The first sum represents the work done by SetMatch(t) and the second the work done by CheckSuffix(s). In Section 7.3.2 we show that this extra work is proportional to $I_k(n, f)$. In addition, we also prove that $C_k(n, f)$ is proportional to $I_k(n, f)$. These results prove the following theorem.

THEOREM 10 *Algorithm ForbCircular(t, s) for generating k-ary circular strings of length n with no substring equal to f is CAT.*

7.2.3 Generating k-ary necklaces with forbidden substrings

Using the ideas from the previous two algorithms, we now outline an algorithm to generate necklaces with forbidden substrings. First, we embed the real-time automata

```
procedure ForbNecklace ( t, p, s : integer ):
local j, q : integer:
begin
    if t > n then begin
        if n mod p = 0 and CheckSuffix(s) then Print( ):
    end else begin
        a_t := a_{t-p}:
        q := δ(s, a_t):
        if t < m then SetMatch(t):
        if q ≠ m then ForbNecklace(t + 1, p, q):
        for j ∈ {a_{t-p} + 1, ..., k - 1} do begin
            a_t := j:
            q := δ(s, j):
            if t < m then SetMatch(t):
            if q ≠ m then ForbNecklace(t + 1, t, q):
end: end: end:
```

Figure 7.5: An algorithm for generating k-ary necklaces with no substring equal to f

based string matching algorithm into the necklace generation algorithm Necklace(t, p). Then, since we must also test the wraparound for necklaces, we add the same tests as outlined in the circular string algorithm. Applying these two simple steps will yield an algorithm for necklace generation with no substring equal to f. Pseudocode for such an algorithm is shown in Figure 7.5. Lyndon words can be generated by replacing the test "n mod $p = 0$" with the test "$n = p$." The initial call is ForbNecklace(1,1,0).

To analyze the running time of this algorithm, we again must show that the number of necklaces generated, $N_k(n, f)$, is proportional to the amount of computation done. In this case the size of the computation tree is

$$NeckCompTree_k(n, f) = \sum_{j=1}^{n} P_k(j, f).$$

However, as with the circular string case, not all nodes perform a constant amount of work. The extra work performed by these nodes is bounded by

$$NeckExtraWork_k(n, f) \leq \sum_{j=1}^{m-1} (m - j) P_k(j, f) + \sum_{j=1}^{m-1} P_k(n - j, f).$$

Note that this expression is the same as the extra work in the circular string case, except we have replaced $I_k(n, f)$ with $P_k(n, f)$.

In Section 7.3.3 we show that when f is a Lyndon word, $NeckCompTree_k(n, f)$ and $NeckExtraWork_k(n, f)$ are both proportional to $\frac{1}{n} I_k(n, f)$. In addition, we also prove that $N_k(n, f)$ is proportional to $\frac{1}{n} I_k(n, f)$. These results prove the following theorem.

THEOREM 11 *Algorithm* **ForbNecklace**(t, p, s) *for generating k-ary necklaces of length n with no substring equal to f is CAT, so long as f is a Lyndon word.*

7.3 Analysis of the Algorithms

This section provides an asymptotic analysis for each of the three generation algorithms developed in this chapter.

7.3.1 Strings

In the string generation algorithm, the total computation is proportional to $CompTree_k(n, f)$ which is given by the following expression:

$$CompTree_k(n, f) = \sum_{j=1}^{n} I_k(j, f).$$

To show that this sum is proportional to the number of strings generated, $I_k(n, f)$, we first prove three lemmas. Recall that we assume $k > 1$.

LEMMA 15 *If string f has length $m \geq 1$ and $n \geq m$, then*

$$I_k(n, f) \geq k I_k(n-1, f) - I_k(n-m, f).$$

PROOF: Let $\alpha = a_1 \cdots a_{n-1}$ be a string in the set $\mathbf{I}_k(n-1, f)$. If we append a character a_n to α, then the string $a_1 \cdots a_n$ is in $\mathbf{I}_k(n, f)$ as long as $f \neq a_{n-m+1} \cdots a_n$.

The number of strings where $f = a_{n-m+1} \cdots a_n$ is at most $I_k(n-m, f)$. □

LEMMA 16 *If string f has length $m > 2$ then*

$$I_k(n, f) \geq I_k(n-1, f) + I_k(n-2, f).$$

PROOF: We prove this lemma by induction on n using the previous lemma. In the base cases, where $n \leq m$, the result is trivial.

$$\begin{aligned} I_k(n, f) &\geq kI_k(n-1, f) - I_k(n-m, f) \\ &\geq 2I_k(n-1, f) - I_k(n-3, f) \\ &\geq I_k(n-1, f) + I_k(n-2, f) + I_k(n-3, f) - I_k(n-3, f) \\ &\geq I_k(n-1, f) + I_k(n-2, f). \end{aligned}$$

□

LEMMA 17 *If string f has length $m > 2$ then*

$$I_k(n, f) \geq \frac{3}{2} I_k(n-1, f).$$

PROOF: If $n \leq m$, the proof is trivial. Otherwise, we use Lemma 15 followed by Lemma 16 to prove the inequality:

$$\begin{aligned} I_k(n, f) &\geq 2I_k(n-1, f) - I_k(n-3, f) \\ &\geq \frac{3}{2} I_k(n-1, f) + \frac{1}{2} I_k(n-1, f) - I_k(n-3, f) \\ &\geq \frac{3}{2} I_k(n-1, f) + \frac{1}{2}(I_k(n-2, f) + I_k(n-3, f)) - I_k(n-3, f) \\ &\geq \frac{3}{2} I_k(n-1, f) + I_k(n-3, f) - I_k(n-3, f) \\ &\geq \frac{3}{2} I_k(n-1, f). \end{aligned}$$

□

The following lemma proves that the total work done by the algorithm is proportional to the total number of strings generated, thus proving Theorem 9.

LEMMA 18 *If string f has length $m > 2$ then*

$$\sum_{j=1}^{n} I_k(j, f) \leq 3I_k(n, f).$$

PROOF: Using Lemma 17 and induction, we prove the result. In the base case when $n \leq m$, the proof is trivial. Otherwise,

$$\begin{aligned}
\sum_{j=1}^{n} I_k(j, f) &\leq I_k(n, f) + \sum_{j=1}^{n-1} I_k(j, f) \\
&\leq I_k(n, f) + 3I_k(n-1, f) \\
&\leq I_k(n, f) + 2I_k(n, f) \\
&\leq 3I_k(n, f).
\end{aligned}$$

□

7.3.2 Circular strings

In the circular string algorithm, the size of the computation tree is the same as the previous algorithm, where it was shown to be proportional to $I_k(n, f)$. In this case, however, there is some extra work required to test the wrap-around of the string. Recall that this extra work is proportional to $ExtraWork_k(n, f)$ which is bounded as follows:

$$\begin{aligned}
ExtraWork_k(n, f) &\leq \sum_{j=1}^{m-1} (m-j) I_k(j, f) + \sum_{j=1}^{m-1} I_k(n-j, f) \\
&\leq \sum_{j=1}^{n} \sum_{t=1}^{j} I_k(t, f) + \sum_{j=1}^{n} I_k(j, f).
\end{aligned}$$

We now use Lemma 18 to simplify the above bound.

$$\begin{aligned} ExtraWork_k(n,f) &\leq 3\sum_{j=1}^{n} I_k(j,f) + \sum_{j=1}^{n} I_k(j,f) \\ &\leq 12 I_k(n,f). \end{aligned}$$

We have now shown that the total work done by the circular string algorithm is proportional to $I_k(n,f)$. Since the total number of strings generated is $C'_k(n,f)$, we must show that $C'_k(n,f)$ is proportional to $I_k(n,f)$.

THEOREM 12 *If the forbidden substring f has length $m > 2$, then*

$$C'_k(n,f) \geq \frac{1}{3} I_k(n,f).$$

PROOF: A string in the set $\mathbf{I}_k(n,f)$ is *not* in the set $\mathbf{C}_k(n,f)$ if it is of the form $a_1 a_2 \cdots a_n$ where $a_i \cdots a_n a_1 \cdots a_{m-(n-i+1)} = f$ for some $i > n-m+1$. For each fixed i, the number of such elements is at most $I_k(n-m,f)$. Since there are $m-1$ choices for i, we deduce:

$$C'_k(n,f) \geq I_k(n,f) - (m-1) I_k(n-m,f). \tag{7.3}$$

Notice that Lemma 17 implies that the expression $(m-1)I_k(n-m,f)$ is maximized when $m = 3$. Thus, we find an upper bound for this expression when $m = 3$ using both Lemma 16 and Lemma 17.

$$\begin{aligned} (m-1) I_k(n-m,f) &\leq 2 I_k(n-3,f) \\ &\leq I_k(n-2,f) + I_k(n-3,f) \\ &\leq I_k(n-1,f) \\ &\leq \frac{2}{3} I_k(n,f). \end{aligned}$$

Substituting this bound back into (7.3) gives the desired result. □

The previous work in this subsection along with Theorem 12 proves Theorem 10.

7.3.3 Necklaces

To prove Theorem 11, we must show that the computation tree along with the extra work done by the necklace generation algorithm is proportional to $N_k(n, f)$. Recall that the computation tree is given by:

$$NeckCompTree_k(n, f) = \sum_{j=1}^{n} P_k(j, f).$$

To simplify this expression we need three additional lemmas; the bound of the first lemma does not hold if f is not a Lyndon word.

LEMMA 19 *If Lyndon word f has length $m > 2$, then*

$$P_k(n, f) \leq \sum_{j=1}^{n} L_k(j, f). \tag{7.4}$$

PROOF: Let $\alpha = a_1 \cdots a_j$ be a Lyndon word and let f also be a Lyndon word. Since α and f are both Lyndon words, it is impossible for f to "wrap around" α when α is considered to be circular. This observation along with equation (2.3), proves the lemma. □

LEMMA 20 *If string f has length $m > 2$ then*

$$L_k(n, f) \leq \frac{1}{n} C_k(n, f).$$

PROOF: Each string counted by $L_k(n, f)$ is a representative of an equivalence class of circular strings, each class having n elements. Summing the strings from all equivalence classes, we get $nL_k(n, f)$ unique k-ary circular strings of length n with no substring equal to f. Since the expression $C_k(n, f)$ counts all k-ary circular strings with length n and no substring equal to f, we get $L_k(n, f) \leq \frac{1}{n} C_k(n, f)$. □

LEMMA 21 *If string f has length $m > 2$ then*

$$\sum_{j=1}^{n} \frac{1}{j} I_k(j, f) \leq \frac{12}{n} I_k(n, f).$$

PROOF: This proof uses Lemma 17 and Lemma 18:

$$\begin{aligned}\sum_{j=1}^{n}\frac{1}{j}I_k(j,f) &\le 2\sum_{j=\lceil n/2\rceil}^{n}\frac{1}{j}I_k(j,f)\\ &\le \frac{4}{n}\sum_{j=\lceil n/2\rceil}^{n}I_k(j,f)\\ &\le \frac{12}{n}I_k(n,f).\end{aligned}$$

□

Applying the previous three lemmas, we show that the computation tree is proportional to $\frac{1}{n}I_k(n,f)$.

$$\begin{aligned}NeckCompTree_k(n,f) &= \sum_{j=1}^{n}P_k(j,f) \le \sum_{j=1}^{n}\sum_{t=1}^{j}L_k(t,f)\\ &\le \sum_{j=1}^{n}\sum_{t=1}^{j}\frac{1}{t}C_k(t,f) \le \sum_{j=1}^{n}\sum_{t=1}^{j}\frac{1}{t}I_k(t,f)\\ &\le \sum_{j=1}^{n}\frac{12}{j}I_k(j,f) \le \frac{144}{n}I_k(n,f).\end{aligned}$$

This inequality is now used to show that the extra work done by the necklace algorithm is also proportional to $\frac{1}{n}I_k(n,f)$. Recall that the extra work is given by:

$$\begin{aligned}NeckExtraWork_k(n,f) &\le \sum_{j=1}^{m-1}(m-j)P_k(j,f)+\sum_{j=1}^{m-1}P_k(n-j,f)\\ &\le \sum_{j=1}^{n}\sum_{t=1}^{j}P_k(t,f)+\sum_{j=1}^{n}P_k(j,f).\end{aligned}$$

Further simplification of this bound follows from the bound on $NeckCompTree_k(n,f)$ along with Lemma 21.

$$\begin{aligned}NeckExtraWork_k(n,f) &\le 144\sum_{j=1}^{n}\frac{1}{j}I_k(j,f)+\frac{144}{n}I_k(n,f)\\ &\le \frac{12^3+12^2}{n}I_k(n,f).\end{aligned}$$

We have now shown that the total computation performed by the necklace generation algorithm is proportional to $\frac{1}{n}I_k(n,f)$. From equation (7.1), $N_k(n,f) \geq \frac{1}{n}C_k(n,f)$, and since $C_k(n,f) \geq \frac{1}{3}I_k(n,f)$, Theorem 11 is proved.

This analysis shows that necklaces with forbidden Lyndon substrings can be generated in constant amortized time. When the forbidden substring is not restricted to be a Lyndon word, no analysis is performed. Some strings, such as $f = 100$, will not exhibit this time bound; however, it remains an open problem to determine exactly which set of substrings will not yield a CAT algorithm.

Chapter 8

Bracelets

This chapter develops an algorithm to generate k-ary bracelets. An analysis proves that the algorithm runs in constant amortized time.

8.1 Background

A *bracelet* is the lexicographically smallest element of an equivalence class of k-ary strings under string rotation and reversal (or a necklace that is also lexicographically minimal among the circular rotations of its reversal). The set of all k-ary bracelets is denoted $\mathbf{B}_k(n)$ and has cardinality $B_k(n)$. In each equivalence class associated with a given bracelet, there exists at most two necklaces: the bracelet itself and the necklace corresponding to the reversal of the bracelet (in some cases the two may be the same). For example, the equivalence class that contains the bracelet 00112012 also contains the necklace 00210211 (see Figure 8.1).

The following theorem for enumerating bracelets is proved by Gilbert and Riordan using Burnside's lemma [12].

THEOREM 13 *The following formula is valid for all $n \geq 1$, $k \geq 1$:*

$$B_k(n) = \begin{cases} \frac{1}{2}(N_k(n) + \frac{k+1}{2}k^{n/2}) & n \text{ even.} \\ \\ \frac{1}{2}(N_k(n) + k^{(n+1)/2}) & n \text{ odd.} \end{cases} \tag{8.1}$$

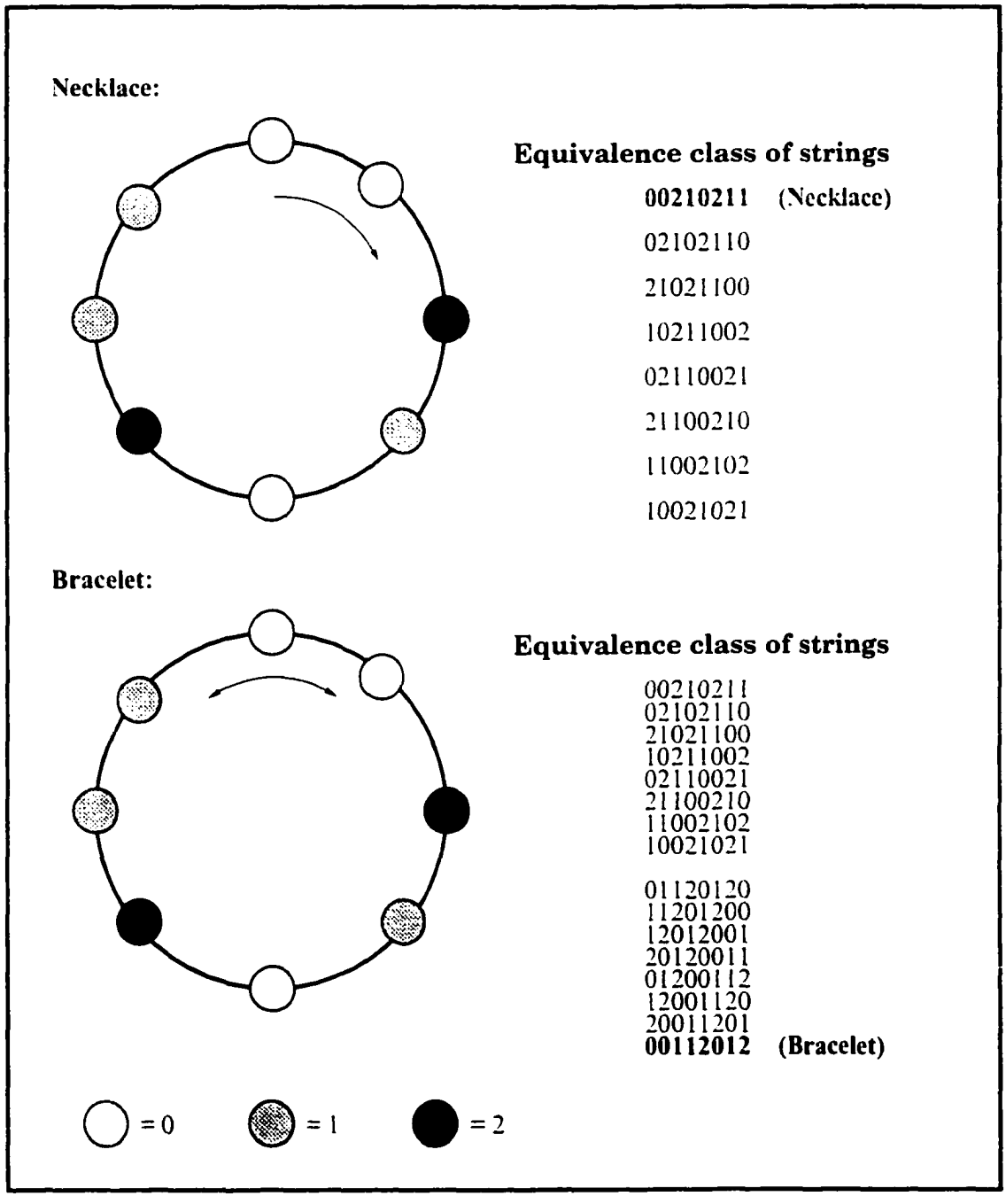

Figure 8.1: Necklaces and bracelets

Notice from the formula that the the number of bracelets is proportional to the number of necklaces since $B_k(n) > \frac{1}{2}N_k(n)$.

In the analysis of our algorithm, it is useful to look at another way to count pre-necklaces. Let $P_k^0(n)$ count all k-ary pre-necklaces of length n that begin with 0. Notice that the number of k-ary pre-necklaces of length n beginning with 1 is equivalent to $P_{k-1}^0(n)$. Similarly the number of k-ary pre-necklaces of length n beginning with 2 is equivalent to $P_{k-2}^0(n)$. This observation leads to the following equation:

$$P_k(n) = \sum_{j=1}^{k} P_j^0(n). \tag{8.2}$$

8.2 Generating Bracelets

In this section we outline a fast algorithm to generate bracelets. Since when $k = 1$, the only bracelet is 0^n, we assume $k \geq 2$. A naïve algorithm for generating bracelets is to generate all k-ary necklaces of length n and then test each necklace against all rotations of its reversal. If no reversed rotation is less than the generated necklace, then the necklace is a bracelet. Since there are n rotations and each test takes $O(n)$ time, this naïve approach will give us an overall running time of $O(n^2 \cdot B_k(n))$ to generate all k-ary bracelets of length n.

A more sophisticated approach will use a necklace finding algorithm, which determines the necklace of a length n string in $O(n)$ time (see Chapter 2.3). Using this technique, we need only compare the generated necklace with the necklace of its reversal. This approach yields a much better running time of $O(n \cdot B_k(n))$ to generate bracelets; however, it is still far from efficient.

To develop an efficient bracelet generation algorithm, we modify Necklace(t, p) by making two important observations. The observations are based on the original idea of comparing a necklace to every rotation of its reversal.

OBSERVATION 1 *If a necklace α is of the form $a^i a_{i+1} \cdots a_n$ for some character $a \neq a_{i+1}$, then we need only test the reversed rotations that also begin with a^i.*

Taking this observation into account, we are making a large improvement on the number of reversed rotations we must check. For example, for the necklace 0010023003 we need only check the three reversed rotations that begin with 00: 0030032001, 0010030032 and 0032001003. To test each reversal we could wait until the entire necklace has been generated, but this will take $O(n)$ time per reversal and we will see no improvement over the naïve algorithms. Instead, if a character is generated in position j that satisfies the condition stated in Observation 1, we immediately compare the pre-necklace $a_1 \cdots a_j$ with its reversal $a_j \cdots a_1$. This comparison will yield one of three outcomes. If $a_1 \cdots a_j > a_j \cdots a_1$ then we terminate the generation from this node, since appending characters to the end of these strings will not affect their relative ordering. If $a_1 \cdots a_j < a_j \cdots a_1$, then no additional testing is required for this reversal. However, if $a_1 \cdots a_j = a_j \cdots a_1$, then more testing must be done on the tail of the strings which has yet to be generated.

Following the above approach, we still need to perform additional testing for the reversals starting at position j where $a_1 \cdots a_j = a_j \cdots a_1$. This leads to the second key observation which we state as a theorem.

THEOREM 14 *If $a_1 \cdots a_n$ is a necklace where $a_1 \cdots a_q = a_q \cdots a_1$ and there exists an r in $\{q+1, \ldots, n\}$ such that $a_1 \cdots a_r = a_r \cdots a_1$ and $a_{r+1} \cdots a_n \leq a_n \cdots a_{r+1}$, then $a_{q+1} \cdots a_n \leq a_n \cdots a_{q+1}$.*

PROOF: Let $P_q = a_1 \cdots a_q$, $P_r = a_1 \cdots a_r$, $x = a_{q+1} \cdots a_r$ and $y = a_{r+1} \cdots a_n$. Let $\hat{x}$ and $\hat{y}$ denote the reversals of x and y respectively. Since P_r and P_q are palindromes $P_r = P_q x = \hat{x} P_q$. Thus, $\alpha = P_q xy = \hat{x} P_q y$. But since α is a necklace, $\alpha = \hat{x} P_q y \leq P_q y \hat{x}$. Thus, since $y \leq \hat{y}$, $xy \leq y\hat{x} \leq \hat{y}\hat{x}$ as required. □

This observation implies that we need only perform extra testing on the reversal starting at the largest position r such that $a_1 \cdots a_r = a_r \cdots a_1$. This extra testing is the comparison of $a_{r+1} \cdots a_n$ to $a_n \cdots a_{r+1}$. If $a_{r+1} \cdots a_n > a_n \cdots a_{r+1}$ then the generated string is *not* a bracelet. This test can be performed in constant time per character for each character generated after position $(n - r)/2 + r$.

Finally, we note that if $a_1 = a_n$ then the only strings that are bracelets must be of the form a^n for some character a.

The following is a summary of the modifications required to transform Necklace(t, p) into an algorithm which generates bracelets:

- Add the parameter u to maintain the value of i from Observation 1: the number of consecutive equivalent characters at the start of the pre-necklace (ie. the pre-necklace starts with a^u).

- Add the parameter v to maintain the number of consecutive a's at end the end of the pre-necklace, where $a = a_1$.

- Add the function CheckRev(t, i) to compare the pre-necklace to its reversal (when $u = v$). If the pre-necklace is greater than its reversal it returns -1; if the pre-necklace is less than its reversal it returns 0; otherwise, the pre-necklace is the same as its reversal and 1 is returned.

- Add the parameter r to maintain the length of the longest pre-necklace equal to its reversal (ie. the largest value r for which $a_1 \cdots a_r = a_r \cdots a_1$).

- Add a test to each character in a position greater than $(n - r)/2 + r$ which will determine whether or not $a_r \cdots a_n$ is greater than its reversal. This will involve the additional parameter RS to hold intermediate boolean values indicating whether or not the *R*eversal is *S*maller.

- Reject the string if $a_1 = a_n$ and the string is not equal to a^n (ie. $t = n$ and $u \neq n$).

An algorithm that is the result of the above modifications is shown in Figure 8.2. The initial call is Bracelet(1,1,0,0,0,FALSE). To illustrate this algorithm we trace the parameters as the string 0010023003 gets generated:

```
function CheckRev( t, i: integer ) returns integer:
local j: integer:
begin
    for j from i + 1 to (t + 1)/2 do begin
        if a_j < a_{t-j+1} then return 0:
        if a_j > a_{t-j+1} then return -1:
    end:
    return 1:
end:

procedure Bracelet( t, p, r, u, v: integer; RS: boolean ):
local rev, i: integer:
begin
    if t - 1 > (n - r)/2 + r then begin
        if a_{t-1} > a_{n-t+2+r} then RS := FALSE:
        else if a_{t-1} < a_{n-t+2+r} then RS := TRUE:
    end:
    if t > n then begin
        if RS = FALSE and n mod p = 0 then PrintIt( ):
    end
    else begin
        a_t := a_{t-p};
        if a_t = a_1 then v := v + 1:
        else v := 0;
        if u = t - 1 and a_{t-1} = a_1 then u := u + 1:
        if t = n and u ≠ n and a_n = a_1 then begin end;
        else if u = v then begin
            rev := CheckRev( t, u ):
            if rev = 0 then Bracelet( t + 1, p, r, u, v, RS ):
            if rev = 1 then Bracelet( t + 1, p, t, u, v, FALSE ):
        end:
        else Bracelet(t + 1, p, r, u, v, RS ):
        if u = t then u := u - 1:
        for j ∈ {a_{t-p} + 1, ..., k - 1} do begin
            a_t := j:
            if t = 1 then Bracelet( t + 1, t, r, 1, 1, RS )
            else Bracelet( t + 1, t, r, u, 0, RS ):
end: end: end:
```

Figure 8.2: Bracelet generation algorithm

α	-	0	0	1	0	0	2	3	0	0	3
t	1	2	3	4	5	6	7	8	9	10	11
p	1	1	1	3	3	3	6	7	7	7	10
r	0	1	2	2	2	5	5	5	5	5	5
a	0	1	2	2	2	2	2	2	2	2	2
e	0	1	2	0	1	2	0	0	1	2	0
RS	F	F	F	F	F	F	F	F	F	F	T

In the following section we give several counting results for strings with no 0^i substring. These results will then be applied in the analysis of this algorithm.

8.3 Forbidden substrings

Recall from the previous chapter that $\mathbf{I}_k(n, f)$ denotes the set of all k-ary strings of length n with no substring equal to f. In this section we investigate a special case of these strings where $f = 0^i$. In this special case we can derive a new recurrence to enumerate these strings:

$$I_k(n, 0^i) = \begin{cases} k^n & \text{if } 0 \leq n < i. \\ (k-1)\sum_{j=1}^{i} I_k(n-j, 0^i) & \text{if } n \geq i. \end{cases}$$

It is easy to verify the correctness of this formula. If $n < i$ then the set $\mathbf{I}_k(n, 0^i)$ will contain all k-ary strings. Otherwise, we categorize the strings in $\mathbf{I}_k(n, 0^i)$ by the number of consecutive 0's found at the tail of each string. Since there are $k-1$ choices for the character appearing before this string of 0's, we arrive at the given recurrence relation.

We obtain another recurrence equation by considering a string $\alpha = a_1 \cdots a_{n-1}$ in the set $\mathbf{I}_k(n-1, 0^i)$. If we append a character a_n to α, then the string $a_1 \cdots a_n$ is in $\mathbf{I}_k(n, 0^i)$ as long as $a_{n-i+1} \cdots a_n \neq 0^i$. The number of strings where $a_{n-i+1} \cdots a_n = 0^i$

is exactly equal to $I_k(n-i,0^i)$. Thus we arrive at a second recurrence equation:

$$I_k(n,0^i) = \begin{cases} k^n & \text{if } 0 \le n < i. \\ kI_k(n-1,0^i) - (k-1)I_k(n-i-1,0^i) & \text{if } n \ge i. \end{cases}$$

LEMMA 22 *If $k, i \ge 2$ then*

$$I_k(n,0^i) \ge \sum_{j=1}^{n-2} I_k(j,0^i).$$

PROOF: The base cases when $n \le i$ is trivial. If $n > i$ then we induct on n:

$$\begin{aligned} I_k(n,0^i) &\ge I_k(n-1,0^i) + I_k(n-2,0^i) \\ &\ge \sum_{j=1}^{n-3} I_k(j,0^i) + I_k(n-2,0^i) \\ &= \sum_{j=1}^{n-2} I_k(j,0^i). \end{aligned}$$

□

LEMMA 23 *If $n > 2$ and $k, i \ge 2$ then*

$$\frac{I_k(n,0^i)}{n} \ge \frac{I_k(n-1,0^i)}{n-1}.$$

PROOF:

$$\begin{aligned} (n-1)I_k(n,0^i) &= k(n-1)I_k(n-1,0^i) - (k-1)(n-1)I_k(n-i-1,0^i) \\ &= nI_k(n-1,0^i) + (kn-n-k)I_k(n-1,0^i) \\ &\qquad - (k-1)(n-1)I_k(n-i-1,0^i) \\ &\ge nI_k(n-1,0^i) + 2(kn-n-k)I_k(n-3,0^i) \\ &\qquad - (kn-n-k+1)I_k(n-3,0^i) \\ &\ge nI_k(n-1,0^i) + (kn-n-k-1)I_k(n-3,0^i) \end{aligned}$$

$$\geq \quad n I_k(n-1,0^t).$$

□

We now prove a theorem that will be used in the analysis of our bracelet generation algorithm. The proof of the theorem uses the previous two lemmas.

THEOREM 15 *If $n > 2$ and $k, t \geq 2$ then*

$$\sum_{j=1}^{n} \frac{1}{j} I_k(j,0^t) \leq \frac{8}{n} I_k(n,0^t).$$

PROOF:

$$\begin{aligned}
\sum_{j=1}^{n} \frac{1}{j} I_k(j,0^t) &\leq 2 \sum_{j=\lceil n/2 \rceil}^{n} \frac{1}{j} I_k(j,0^t) \\
&\leq \frac{2}{n} I_k(n,0^t) + \frac{2}{n-1} I_k(n-1,0^t) + 2 \sum_{j=\lceil n/2 \rceil}^{n-2} \frac{1}{j} I_k(j,0^t) \\
&\leq \frac{4}{n} I_k(n,0^t) + \frac{4}{n} \sum_{j=\lceil n/2 \rceil}^{n-2} I_k(j,0^t) \\
&\leq \frac{8}{n} I_k(n,0^t).
\end{aligned}$$

□

8.4 Analysis

In this section we show that the algorithm **Bracelet**(t,p,r,u,v,RS) for generating bracelets is CAT. We analyze the algorithm by looking at the computation tree and determining the amount of work done at each node. To get a bound on the size of the bracelet computation tree, we observe the following bounds obtained from equations (2.2) and (2.1):

$$L_k(n) \leq \frac{k^n}{n} \leq N_k(n) \leq 2\frac{k^n}{n}.$$

Now using equation (8.1) we get the following bounds on the number of bracelets:

$$\frac{k^n}{2n} \leq B_k(n) \leq 2\frac{k^n}{n}. \tag{8.3}$$

Since the necklace algorithm Necklace(t, p) is CAT, the size of its computation tree is less than ck^n/n for some constant c. This bound is also true for Bracelet(t, p, r, u, v, RS) since its computation tree is smaller than that of Necklace(t, p). However, unlike the necklace computation tree, the bracelet computation tree has nodes that require more than a constant amount of work. From our algorithm, these nodes are the ones that make a call to CheckRev(t, i). Thus, to prove the bracelet generation algorithm Bracelet(t, p, r, u, v, RS) is CAT, we must show that the work performed by all calls to CheckRev(t, i) is bounded by some constant times the total number of bracelets generated.

From the algorithm, each node that makes a call to CheckRev(t, i) is a pre-necklace of the form a^t or $a^t\gamma a^t$ where $1 \leq i < n/2$ and the non-empty string γ begins and ends with a character lexicographically greater than a. The length of such pre-necklaces is strictly less than n. Each call to CheckRev(t, i) results in work proportional to $(t - 2i)/2$, where t is the length of the pre-necklace. Since any pre-necklace of the form a^t requires no extra work, we concentrate on finding a bound for the number of pre-necklaces of the form $a^t\gamma a^t$.

To simplify this task we consider only the pre-necklaces beginning with 0, using equation (8.2) to account for the remaining pre-necklaces. We also ignore the fact that some of these pre-necklaces are never generated by the algorithm. Now observe that the number of pre-necklaces of the form $0^t\gamma 0^t$ is less than or equal to the number of pre-necklaces of the form $0^t\gamma$. We define the set of all k-ary pre-necklaces of length n beginning with 0, ending with a non-zero character, and with no 0^t substring, to be $\mathbf{P}'_k(n, 0^t)$. Equivalently, the set $\mathbf{P}'_k(n, 0^t)$ contains all pre-necklaces with length n of the form $0^j\gamma$ for $1 \leq j < i$. The cardinality of this set is denoted $P'_k(n, 0^t)$.

If we let $E_k(n)$ denote the extra work that results from all calls made to CheckRev(t, i) by pre-necklaces beginning with 0 (while generating $\mathbf{B}_k(n)$), then we obtain

the following bound:

$$E_k(n) \leq \sum_{i=2}^{n-1} \frac{n-i}{2} P'_k(n-i, 0^t). \tag{8.4}$$

There is no known formula to count $P'_k(n, 0^t)$ so we derive an upper bound. Because every pre-necklace is obtained as a prefix of a β^* where β is some Lyndon word, we arrive at the formula given in equation (2.3):

$$P_k(n) = \sum_{j=1}^{n} L_k(j).$$

Recall that $L_k(j, 0^t)$ denotes the number of Lyndon words of length j with no 0^t substring. The following upper bound is obtained for $P'_k(n, 0^t)$:

$$P'_k(n, 0^t) \leq \sum_{i=1}^{n} L_k(j, 0^t). \tag{8.5}$$

Now recall that the number of k-ary strings of length n with no 0^t substring is denoted by $I_k(n, 0^t)$. Using these strings we obtain an upper bound for $L_k(n, 0^t)$.

LEMMA 24 *The following inequality is valid for $n \geq 1$, $k \geq 1$ and $t \geq 1$:*

$$L_k(n, 0^t) \leq \frac{1}{n} I_k(n, 0^t).$$

PROOF: Each string counted by $L_k(n, 0^t)$ is a representative of an equivalence class of strings each with n elements. If we add up the elements from each equivalence class we get $nL_k(n, 0^t)$ unique strings each of length n with no 0^t substring. The expression $I_k(n, 0^t)$ counts the total number of strings with length n and no 0^t substring. Therefore $L_k(n, 0^t) \leq \frac{1}{n} I_k(n, 0^t)$. □

Now using the previous lemma and Theorem 15 (assuming $n > k$) we can simplify

the upper bound in (8.5):

$$\begin{aligned} P'_k(n,0^t) &\leq \sum_{j=1}^{n} \frac{1}{j} I_k(j,0^t) \\ &\leq \frac{8}{n} I_k(n,0^t). \end{aligned}$$

We now substitute this result into (8.4) and simplify:

$$\begin{aligned} E_k(n) &\leq \sum_{i=2}^{n-1} \frac{n-i}{2} P'_k(n-i,0^t) \\ &\leq 4 \sum_{i=2}^{n-1} I_k(n-i,0^t). \end{aligned}$$

Observe that we can insert 0^t1 at the front of each string in $\mathbf{I}_k(n-i,0^t)$, creating a set of strings of length $n+1$. Notice that each new string is a unique pre-necklace regardless of the parameter i. Thus the number of strings in the union of the sets $\mathbf{I}_k(n-i,0^t)$ for $i = 2,\ldots,n-1$ is less than $P_k(n+1)$. We can divide this total by $k-1$, since we could have arbitrarily chosen any of $k-1$ characters to insert after 0^t. Thus:

$$\begin{aligned} E_k(n) &\leq \frac{1}{k-1} P_k(n+1) \\ &= \frac{1}{k-1} \sum_{j=1}^{n+1} L_k(j) \\ &\leq \frac{4}{k-1} \sum_{j=1}^{n+1} \frac{k^j}{j} \\ &\leq \frac{12}{k-1} \frac{k^{n+1}}{n+1} \\ &\leq 24 \frac{k^n}{n}. \end{aligned} \tag{8.6}$$

The simplification found in equation (8.6) is valid for $k \geq 2$ and can easily be proved by induction. Because the bound on $E_k(n)$ is only for pre-necklaces beginning with 0, we use equation (8.2) to get an upper bound on the extra work performed by all

pre-necklaces. Note that $E_1(n) = 0$.

$$ExtraWork \leq \sum_{j=2}^{k} E_j(n) \leq \frac{24}{n} \sum_{j=2}^{k} j^n \leq 48 \frac{k^n}{n}.$$

From (8.3), the total number of bracelets generated is bounded below by $k^n/2n$. Thus, the running time of the algorithm **Bracelet**(t, p, r, u, v, RS) is proportional to the number of bracelets generated, which proves the following theorem.

THEOREM 16 *Algorithm* **Bracelet**(t, p, r, u, v, RS) *for generating k-ary bracelets is CAT.*

Experimentally, the constant is less than 8 (where we count the number of calls to **Bracelet**(t, p, r, u, v, RS) plus the number of iterations of the **for** loop in **Check-Rev**(t, i)).

Chapter 9

Lyndon Brackets

In this chapter an algorithm is developed to generate a special bracketing of length n Lyndon words. These strings are known to form a basis for the n-th homogeneous component of the free Lie algebra.

9.1 Background

Recall that $\mathbf{L}_k(n)$ denotes the set of all length n Lyndon words over an alphabet of size k. Let $\mathbf{L}_k$ denote the set of all Lyndon words over an alphabet of size k. The *standard factorization* of a Lyndon word w ($|w| > 1$), denoted $\sigma(w)$, is the pair (l, m), $l, m \in \mathbf{L}_k$ such that $w = lm$ where m has maximal length and l is non-empty. By Proposition 5.1.3 in Lothaire [18] such a factorization exists. Using this standard factorization, the Lyndon words can be recursively mapped into a special bracketing using the following function:

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

If w is a Lyndon word, then $\gamma(w)$ is called the *Lyndon bracket* of w. The set of all length n Lyndon brackets is demonstrated to be a basis for the n-th homogeneous component of the free Lie algebra in [18]. The bracketing that results when γ is applied to $\mathbf{L}_2(6)$ is illustrated in Figure 9.1.

[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]

Figure 9.1: The Lyndon brackets of $\mathbf{L}_2(6)$

9.2 Generating Lyndon brackets

This section addresses the problem of generating the brackets for $\mathbf{L}_k(n)$. The Lyndon words can be generated in constant amortized time using the algorithm Necklace(t, p), but the problem of generating these words with their respective bracketing previously had no fast solution.

A naïve approach to generating these strings is to generate a Lyndon word of length n and then test each proper right factor (starting with maximal length) until a Lyndon word is found. This process is repeated recursively for each of the two factors. Verifying whether or not each factor is a Lyndon word takes linear time (see Chapter 2.3). In the worst case, this test will have to be performed for each proper right factor. Since this must be done recursively, the total running time to generate each basis element will be $O(n^3)$.

A significant improvement can be made to this naïve algorithm from the following observation. If the longest proper right factor is known for each Lyndon factor, then the bracketing for each Lyndon word can be determined in linear time using the recursive function PrintBracket$(start, end)$ displayed in Figure 9.2. In this function, the Lyndon word being generated $a_1 \ldots a_n$ is stored in the array a. If $a_i \ldots a_j$ is a Lyndon word then the value of $split(i, j)$ is the starting position of its longest proper right factor. Observe that $split(i, j)$ is defined only for $i < j$. As an example the $split(i, j)$ values for the Lyndon word 001001011 are displayed in Figure 9.3. In this figure the value i represents the row number and the value j represents the column

```
procedure PrintBracket ( start, end : integer );
begin
    if start = end then print(a_start);
    else begin
        print(" [ ");
        PrintBracket( start, split(start, end) − 1 );
        print(" . ");
        PrintBracket( split(start, end), end );
        print(" ] ");
end; end;
```

Figure 9.2: A function to print the brackets of a Lyndon word

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

Figure 9.3: The values $split(i, j)$ for the Lyndon word 001001011

number, where each value ranges from 1 to n. The entry $split(1, n)$ determines the starting point of the longest proper right factor in the original Lyndon word, and thus the standard factorization of 001001011 is (001.001011).

Observe that the value $split(i, j)$, where $i < j$, can be defined recursively by the following:

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

Thus, the key to obtaining each value $split(i, j)$ is to determine all the Lyndon factors embedded in the Lyndon word $a_1 \cdots a_n$. This can be done by modifying the algorithm

```
procedure LyndonBracket ( t : integer ):
local i, j : integer;  q : array of integer;
begin
    if t > n then begin
        if n = p[1] then begin PrintBracket( 1, n ); println(); 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;
```

Figure 9.4: An algorithm for generating Lyndon brackets

Necklace(t, p) to generate Lyndon words. In this algorithm, the parameter p maintains the length of the longest Lyndon prefix of the pre-necklace $a_1 \cdots a_{t-1}$. However, to obtain all the Lyndon factors, the longest Lyndon prefix must be maintained for all strings $a_i \cdots a_{t-1}$ where $1 \leq i < t$. If this value is stored in p_i, then effectively, the parameter p is replaced with the global string of values $p_1 \cdots p_n$. If the string starting at a_i is not a pre-necklace, then p_i is assigned the value 0; otherwise, it maintains the length of the longest Lyndon prefix starting at a_i. Note that the value p_1 replaces the old parameter p. Using the values $p_1 \cdots p_t$, the values for $split(i, t)$ can be determined using its 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 Figure 9.4 is the result of applying these modifications to the algorithm Necklace(t, p). The initial call is LyndonBracket(1) and the values p_i are initialized to 1.

9.3 Analysis

Updating the values for $p_1 \cdots p_t$ and $split(i, t)$ where $1 \leq i < t$ takes linear time. The function PrintBracket($start, end$) also takes linear time. Thus, since the the algorithm Necklace(t, p) for generating Lyndon words is CAT, the algorithm LyndonBracket(t) for generating Lyndon brackets will take linear amortized time per string generated.

Chapter 10

Conclusion

This chapter summarizes the primary contributions of this dissertation, and details avenues for future research.

10.1 Contributions

This dissertation has considered the problem of generating restricted classes of strings under rotation. In particular, fast new algorithms have been developed to generate:

- binary unlabeled necklaces,
- fixed density necklaces,
- necklaces where the number of each alphabet symbol is fixed,
- chord diagrams,
- necklaces and strings with forbidden substrings,
- bracelets, and
- Lyndon brackets.

An analysis proves the generation algorithms for binary unlabeled necklaces, fixed density necklaces, necklaces and strings with forbidden substrings, and bracelets are

CAT. Experimental results indicate the same time bound for the algorithms to generate necklaces where the number of each alphabet symbol is fixed and chord diagrams. The algorithm for generating the Lyndon brackets (a basis for the n-th homogeneous component of the free Lie algebra) runs in $O(n)$ amortized time per basis element.

The two most significant results are the algorithms for generating fixed density necklaces and bracelets that run in constant amortized time. Previous work on these problems failed to attain this time bound: for example see [10], [17], and [28]. The result for fixed density necklaces has been published in the *Siam Journal on Computing*, Vol. 29 No. 2.

10.2 Future work

This dissertation leaves several open problems relating to the generation of restricted classes of necklaces.

- Develop an efficient algorithm to generate unlabeled necklaces over any sized alphabet.
- Prove or disprove that the algorithm FixedZero(t, p) presented in Chapter 4.1 is CAT when the number of zeros is less than $n/2$.
- Develop a CAT algorithm to generate necklaces where the number of alphabet symbols is fixed using the outline presented in Chapter 5.3.
- Prove or disprove that the algorithm FastChords(n) presented in Chapter 6.5 is CAT.
- Determine the set of substrings for which the algorithm ForbNecklace(t, p, s) is not CAT, and develop an efficient algorithm for these substrings.

Bibliography

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley , 1974.

[2] D. Bar-Natan. On the Vassiliev knot invariants. Topology 34 (1995) 423-472.

[3] L. Batton, C. Bohun, A. Bona, K. Cheng, T. Doman, J. Drew, R. Edwards, S. Kutay, C. Laflamme, D. McCrea, W. Myrvold, F. Ruskey, J. Sawada, P. van den Driessche, J. Vander Kloet and K. Wood. Classification of chemical compound pharmacophore structures. PIMS Third Industrial Problem Solving Workshop (to appear), 1999.

[4] D. Beckwith. Equivalence classes and cyclic arrangements. The American Mathematical Monthly, Vol. 105 No. 8 (1998) 774-775.

[5] J. Birman and R. Trapp. Braided chord diagrams. Journal of Knot Theory and its Ramifications. Vol. 7 No. 1 (1998) 1-22.

[6] K. Cattell, F. Ruskey, J. Sawada, C.R. Miers, M. Serra, A fast algorithm to generate unlabeled necklaces and irreducible polynomials over GF(2), to appear in Journal of Algorithms.

[7] W.Chen ad J. Louck, Necklaces, MSS sequences and DNA sequences, Advances in Applied Mathematics, 18 (1997) 18-32.

[8] R. Cori and M. Marcus. Counting non-isomorphic chord diagrams. Theoretical Computer Science. 204 (1998) 55-73.

[9] J-P. Duval. Factoring words over an ordered alphabet. Journal of Algorithms. 4 (1983). 363-381.

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

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

[12] E.N. Gilbert and J. Riordan. Symmetry types of periodic sequences. Illinois J. Mathematics. 5 (1961) 657-665.

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

[14] Dan Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.

[15] J. Harer and D. Zagier, The Euler characteristic of the moduli space of curves, Inventiones Mathematicae, 85 (1986) 457–485.

[16] B. Li and H. Sun, Exact number of chord diagrams and an estimation of the number of spine diagrams of order n, Chinese Science Bulletin, Vol. 42 No. 9 (1997), 705-720.

[17] P. Lisonek, *Computer-assisted Studies in Algebraic Combinatorics*, Dissertation 1994.

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

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

[20] F. Ruskey, *Combinatorial Generation*, manuscript, 1995.

[21] F. Ruskey, C.D. Savage, and T. Wang, Generating necklaces, Journal of Algorithms, 13 (1992) 414–430.

[22] F. Ruskey and D. van Baronaigien, Fast recursive algorithms for generating combinatorial objects, Congressus Numerantium, 41 (1984), 53-62.

[23] R.P. Stanley, *Enumerative Combinatorics, Volume I*, Wadsworth & Brooks/Cole, 1986.

[24] A. Stoimenow, Enumeration of chord diagrams and an upper bound for Vassiliev invariants, Journal of Knot Theory and its Ramifications, Vol. 7, No. 1 (1998), 93-114.

[25] A. Stoimenow, On the number of chord diagrams, manuscript, http://www.informatik.hu-berlin.de/~stoimenow.

[26] T. Walsh and A. Lehman, Counting rooted maps by genus I,II, Journal of Combinatorial Theory (B), 13 (1972), 192-218, 122-141.

[27] S. Wang, C Hains, Method of calibrating a digital printer using component test patches and the Yule-Nielsen equation, United States Patent Number 5,748,330 (1998).

[28] T.M.Y Wang and C.D. Savage, A Gray code for necklaces of fixed density, SIAM Journal of Discrete Math, 9 (1996) 654-673.

[29] D. Wiedermann, Cyclic difference covers through 133, Congressus Numerantium, 90 (1992), 181-185.