A Gray code for fixed-density necklaces and Lyndon words in constant amortized time

J. Sawada* A. Williams†

September 13, 2010

Abstract

This paper develops a constant amortized time algorithm to produce the cyclic cool-lex Gray code for fixed-density binary necklaces, Lyndon words, and pseudo-necklaces. It is the first Gray code for these objects that achieves this time bound. The algorithm is applied: (i) to develop a constant amortized time cyclic Gray code for necklaces, Lyndon words, and pseudo-necklaces ordered by density and (ii) to obtain a fixed-density de Bruijn sequence using constant time per n bits on average. In addition to Gray code order, the algorithms can be easily modified to output the strings in co-lex order.

Keywords: Lyndon word, necklace, pseudo-necklace, Gray code, cool-lex, fixed-density, de Bruijn sequence

1 Introduction

Combinatorial generation is the study of efficient algorithms for exhaustively generating every instance of a specific combinatorial object. The research area is fundamental to computer science, as evidenced by Knuth's devotion of over 400 pages to the subject in the upcoming volume of *The Art of Computer Programming* [6, 7, 8]. One of the most important aspects of combinatorial generation is to find orderings of the objects so that only a constant amount of change is required to go from one instance to the next. Such orderings are called *Gray codes*.

Fast and simple algorithms for generating necklaces and Lyndon words have been known for some time [3, 4, 9]. However, an open problem for many years was whether or not there existed a Gray code to list such objects. For *fixed-density* binary necklaces – those where the number of 1s is fixed – Gray codes were discovered separately by Wang and Savage [17] and Ueda [14]. Both algorithms were constructed by finding a Hamilton path in a graph whose vertices correspond to the necklaces of length n and density d. Their algorithms did not use lexicographically minimal representations and thus did not apply to Lyndon words, and unfortunately the lists produced by these algorithms could not be concatenated together to find a Gray code for all binary necklaces; they were not cyclic. Additionally, these algorithms did not obtain the optimal constant amortized time (CAT) implementations. Such an algorithm does exist, however, for lexicographic

*Computing and Information Science, University of Guelph, Canada. Research supported by NSERC. E-mail: jsawada@uoguelph.ca

†Computer Science, University of Victoria, Canada. Research supported by NSERC.E-mail: haron@uvic.ca

order [10]. In 2006 the problem of finding a Gray code for necklaces was finally answered by Vajnovszki [15] for a binary alphabet and then generalized for alphabets of arbitrary size by Vajnovszki and Weston [18]. Additionally, conjectures have been made by Degni and Drisko regarding necklace Gray codes using non-traditional representatives [1]. However, there remained three interesting open problems: (i) to find a Gray code for fixed-density Lyndon words, (ii) to find a Gray code for fixed-density necklaces using the lexicographically minimal rotation as the representative, and (iii) to find a Gray code for necklaces and Lyndon words that is ordered by density. All three of these problems were answered simultaneously in [11] based on the cool-lex framework that we re-visit in Section 2.2. The main result of this paper is to provide CAT implementations for these Gray codes thus providing:

- ▷ the first CAT Gray code algorithm for fixed-density necklaces and Lyndon words,
- ▷ the first CAT Gray code algorithm for necklaces and Lyndon words ordered by density,
- ▷ an algorithm to construct a fixed-density de Bruijn sequence in constant time per n bits on average.

As an intermediate step we develop a CAT algorithm to generate fixed-density *pseudo-necklaces* which we define in the next section.

As mentioned earlier, the cool-lex framework is applied to efficiently generate the Gray codes. This framework has also been used to produce over a dozen other CAT algorithms for other fixed-density objects by implementing a single "oracle" function for each object [13]. For many of the objects, a constant time oracle is fairly straightforward; however, for necklaces, an efficient oracle is not a trivial matter.

The rest of the paper is outlined as follows. In Section 2, we provide a background on necklaces and Lyndon words and introduce the notion of pseudo-necklaces. We also outline the cool-lex Gray code for any fixed-density bubble language specified in [11, 13] and describe how the output can be altered to obtain co-lex order. In Section 3, we develop and prove our main result: a CAT algorithm for the cool-lex Gray code for fixed density necklaces and Lyndon words. Then in Section 4 we describe how to apply our main result (i) to list all necklaces and Lyndon words in Gray code order by density in constant amortized time and (ii) to construct a fixed-density de Bruijn sequence using constant time per n bits on average. We conclude in Section 5 with a summary and a few open problems.

2 Background

We begin by defining a compact representation for binary strings using a series of *blocks* which are maximal substrings of the form 0^*1^*. Each block B_i is composed of two integers (s_i, t_i) representing the number of 0s and 1s respectively. For example, the string $\alpha = 00011010100011001$ can be represented by $B_5B_4B_3B_2B_1 = (3,2)(1,1)(1,1)(3,2)(2,1)$. Maintaining this block representation will be critical to the efficiency of our algorithms in this paper.

A binary string $\alpha = a_1a_2\cdots a_m$ is said to be *lexicographically smaller* than $\beta = b_1b_2\cdots b_n$, written $\alpha < \beta$, if one of the following holds:

(1) $m < n$ and $a_1a_2\cdots a_m = b_1b_2\cdots b_m$ or
(2) there exists $1 \leq i < m$ such that $a_1a_2\cdots a_i = b_1b_2\cdots b_i$ and $a_{i+1} < b_{i+1}$.

To simplify the discussion later, we write $B_i < B_j$ if $0^{s_i}1^{t_i} < 0^{s_j}1^{t_j}$ in the lexicographic order just defined.

2.1 Necklaces, Lyndon words, and Pseudo-necklaces

A *necklace* is defined to be the lexicographically smallest string in an equivalence class of strings under rotation. A *Lyndon word* is an aperiodic necklace. A string $\alpha = a_1 \cdots a_n = B_c \cdots B_1$ is a *pseudo-necklace* if $B_c \leq B_i$ for all $1 \leq i < c$. This is the first time that pseudo-necklaces have been defined and they will be used as a stepping stone in our algorithms for necklaces and Lyndon words. For any binary string α we say that the *density* of the string is the number of 1s it contains and we denote this number by $den(\alpha)$. In this paper we are concerned with binary necklaces, Lyndon words, and pseudo-necklaces of *fixed-density*.

We will use the following notation to denote these objects:

- $\mathbf{N}(n, d)$: the set of binary necklaces of length n and density d,
- $\mathbf{L}(n, d)$: the set of binary Lyndon words of length n and density d,
- $\mathbf{P}(n, d)$: the set of binary pseudo-necklaces of length n and density d.

Note that $\mathbf{L}(n, d) \subseteq \mathbf{N}(n, d) \subseteq \mathbf{P}(n, d)$. To further illustrate these objects we provide a few examples:

- ▷ 001001 is a necklace but not a Lyndon word,
- ▷ 00101001 is a pseudo-necklace but not a necklace,
- ▷ $01001 = (1, 1)(2, 1)$ is not a pseudo-necklace since $(1, 1) > (2, 1)$,
- ▷ $0010110 = (2, 1)(1, 2)(1, 0)$ is not a pseudo-necklace since $(2, 1) > (1, 0)$.

The number of fixed-density necklaces and Lyndon words, denoted $N(n, d)$ and $L(n, d)$ respectively, can be deduced using Burnside's Lemma and Möbius inversion as described in [5] and [10]. The formulae are as follows:

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

$$L(n, d) = \frac{1}{n} \sum_{j \backslash gcd(n,d)} \mu(j) \binom{n/j}{d/j}.$$

Euler's totient function $\phi(j)$ denotes the number of positive integers less than or equal to j that are relatively prime to j. The Möbius function $\mu(j)$ evaluates to $(-1)^t$ if j is the product of t distinct primes, and 0 otherwise. It remains an open problem to find a simple enumeration formula for fixed-density pseudo-necklaces.

2.2 Cool-lex Gray code for bubble languages

A fixed-density language $\mathcal{L}$ is said to be a *first-10 bubble language* if it has the following property: if $\alpha \in \mathcal{L}$ then by swapping the first 10 (if it exists) to 01 yields another string in $\mathcal{L}$. The following recurrence from [11] can be used to produce a listing of the strings in any first-10 bubble language $\mathcal{L}$ composed of binary strings with length n and density d in Gray code order:

$$\mathbf{C}(s, t, \gamma) = \begin{cases} \mathbf{C}(s-1, 1, 01^{t-1}\gamma), \mathbf{C}(s-1, 2, 01^{t-2}\gamma), \ldots, \mathbf{C}(s-1, t-j, 01^{j}\gamma), 0^s 1^t \gamma & \text{if } s > 0 \\ 1^t \gamma & \text{if } s = 0 \end{cases}$$

where j is the smallest non-negative integer such that $0^{s-1}1^{t-j}01^j\gamma \in \mathcal{L}$. In this recurrence γ represents a fixed suffix and each recursive term prepends a string of the form 10^i to γ. To be precise, $\mathbf{C}(d, n-d, \epsilon)$ will produce the Gray code for $\mathcal{L}$. Since fixed-density necklaces (and Lyndon words) are proved to be first-10

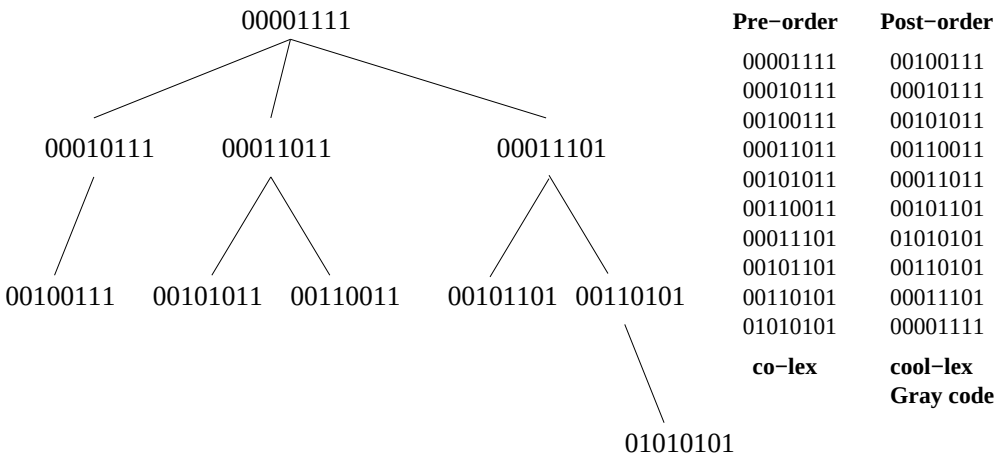

Figure 1: Computation tree for $\mathbf{P}(8,4) = \mathbf{N}(8,4)$ illustrating both co-lex and cool-lex order.

bubble languages in [11], they can be generated in Gray code order using this recurrence. A similar proof will show that pseudo-necklaces also form a first-10 bubble languages.

Observe that if we alter the first line of the recurrence so the last term $0^s1^t\gamma$ is moved to the front, then the resulting order is co-lex order. Because of this similarity, this specific Gray code ordering has been called a *cool-lex* ordering. As an illustration consider the computation tree in Figure 1 for $\mathbf{C}(4,4,\epsilon)$ where $\mathcal{L} = \mathbf{P}(8,4) = \mathbf{N}(8,4)$. Each node in the computation tree $\alpha = 0^s1^t\gamma$ corresponds to the string that gets output directly from a recursive call to $\mathbf{C}(s,t,\gamma)$. By traversing the tree in post-order, we obtain the cool-lex Gray code. If the tree is traversed in pre-order, we obtain co-lex order.

Pseudocode for a recursive algorithm Gen(s,t) based on the recurrence is given in Figure 2. The function Oracle(s,t) is specific to the bubble language $\mathcal{L}$ being generated and it returns the value j from the recurrence for the current string $\alpha = 0^s1^t\gamma$. Observe that each iteration of the **for** loop updates α by performing a single swap of the bits in position s and $s+t-i$. This operation effectively prepends 01^i to γ. Also included are the procedures UpdateBlock(s,t,i) and RestoreBlock(s,t,i) required to maintain the run-length block representation. The details of these procedures are discussed in [13] as they were required for the oracles of other bubble languages. The variable c indicates the current number of blocks in α, and at the start of each recursive call the suffix γ from the recurrence will be composed of the final $c-1$ blocks: $B_{c-1}B_{c-2}\cdots B_1$. The initial call is Gen$(n-d,d)$ where the data structures are initialized as follows: $\alpha := 0^{n-d}1^d$, $c := 1$, and $B_1 := (n-d,d)$.

The function Visit() can be used to print out the strings in full form α or in block form $B_c\cdots B_1$. Additionally, the difference between successive strings can be output as a sequence of shifts (shifting one bit to the left in a specified substring) or as a sequence of 1 or 2 swaps [11].

Since every recursive call visits a string in $\mathcal{L}$, we obtain the following theorem:

THEOREM 1. *[11] If the total amount of computation required by all calls to Oracle$(\mathcal{L})$ is proportional to the number of strings in a fixed-density first-10 bubble language $\mathcal{L}$, then the algorithm Gen(s,t) will generate all strings in $\mathcal{L}$ in constant amortized time.*

Our challenge is to efficiently implement an oracle for fixed-density necklaces and Lyndon words. In [13], a $\Theta(n)$ oracle is provided which leads to a $\Theta(n)$-amortized time algorithm. In the next section, we will discuss

```
procedure RestoreBlock(int s, int t, int i)

    if i = 0 and (c > 2 or B_1 ≠ (1,0)) then
        s_{c-1} := s_{c-1} - 1
        s_c := s
    else
        B_{c-1} := (s,t)
        c := c - 1
end.

procedure Gen(int s, int t)
int i, j
    if s > 0 and t > 0 then
        j := Oracle(s,t)
        for i := t - 1 downto j do
            UpdateBlock(s,t,i)
            Swap(a_s, a_{s+t-i})
            Gen(s - 1, t - i)
            Swap(a_s, a_{s+t-i})
            RestoreBlock(s,t,i)
    Visit()
end.
```

```
procedure UpdateBlock(int s, int t, int i)

    if i = 0 and c > 1 then
        s_{c-1} := s_{c-1} + 1
        s_c := s - 1
    else
        B_c := (1,i)
        B_{c+1} := (s - 1, t - i)
        c := c + 1
end.
```

Figure 2: A simple recursive algorithm to list all strings in any first-10 bubble language $\mathcal{L}$ in cool-lex Gray code order. Included are the routines required to efficiently maintain the block encoding.

an oracle that leads to a CAT algorithm.

3 A CAT algorithm for fixed-density necklaces and Lyndon words

In this section we present a CAT algorithm for fixed-density pseudo-necklaces, necklaces and Lyndon words. First, we provide a constant time oracle for fixed-density pseudo-necklaces which immediately results in a CAT algorithm for these strings. Then we present a new algorithm to test if the block representation of a string is a necklace. Combining these two results we obtain an algorithm to generate fixed-density necklaces or Lyndon words in cool-lex Gray code order. A non-trivial analysis proves that the algorithm is CAT.

3.1 An oracle for pseudo-necklaces

Recall that an oracle for pseudo-necklaces must return the smallest non-negative value j such that $\beta = 0^{s-1}1^{t-j}01^j\gamma$ is a pseudo-necklace given that $\alpha = 0^s1^t\gamma$ is a pseudo-necklace. In order to make the oracle efficient, we maintain the block representation $B_c \cdots B_1$ for α and also maintain an extra parameter r within the recursion such that B_r is the lexicographically smallest block in γ. This extra parameter can be trivially maintained in constant time.

Consider the special case when $s = 1$. Since all pseudo-necklaces with density $d < n$ must begin with 0 the oracle returns t. To make sure that β does not end with 0 when $d > 0$, we also consider a special case when $c = 1$ where γ is empty. Clearly $j > 0$ since all pseudo-necklaces with $d > 0$ end with 1. If $s > 2$ then

```
function PseudoOracle( int s, int t , int r ) returns int

    if s = 1 then return t
    if c = 1 then
        if s = 2 then return ⌊(n−1)/2⌋
        return 1
    if s − 1 > s_r + 1 then return 0
    if s − 1 = s_r + 1 then
        if (s − 1, t) ≤ (s_{c−1} + 1, t_{c−1}) then return 0
        return 1
    if s − 1 = s_r then
        if s = 2 then return max(t − t_r, ⌊(t+1)/2⌋)
        if (s − 1, t) ≤ (s_{c−1} + 1, t_{c−1}) then return max(t − t_r, 0)
        return max(t − t_r, 1)
    if s − 1 = s_r − 1 then return t
end.
```

Figure 3: Oracle for fixed-density pseudo-necklaces.

the oracle returns 1. Otherwise $s = 2$ (since we already handled the case when $s = 1$) and the oracle returns $\lfloor \frac{n-1}{2} \rfloor$.

In the remaining cases the oracle must satisfy two conditions (i) $(s-1, t-j) \leq (s_r, t_r)$ and (ii) $(s-1, t-j)$ must be less than or equal to the second block of β. If $j > 0$, the second block of β is $(1, j)$; otherwise $j = 0$ and the second block is $(s_{c-1} + 1, t_{c-1})$. We focus on four possible values for $s - 1$ relative to s_r recalling that $s_{c-1} \leq s_r$.

Case 1 $s-1 > s_r+1$: The oracle returns 0.

Case 2 $s-1 = s_r+1$: If $(s-1, t) \leq (s_{c-1} + 1, t_{c-1})$ then the oracle returns 0; otherwise it returns 1.

Case 3 $s-1 = s_r$: To satisfy the first condition $t - j \leq t_r$ and thus $j \geq t - t_r$. To satisfy the second condition we consider two sub-cases. If $s = 2$ then $j \geq \lfloor \frac{t+1}{2} \rfloor$. Thus the oracle returns $\mathsf{max}(t - t_r, \frac{t+1}{2} \rfloor)$. If $s > 2$ and $(s-1, t) \leq (s_{c-1} + 1, t_{c-1})$ then the oracle returns $\mathsf{max}(t - t_r, 0)$; otherwise it returns $\mathsf{max}(t - t_r, 1)$.

Case 4 $s-1 = s_r-1$: Since γ contains the substring 0^s, β must start with 0^s to be a pseudo-necklace. Thus $j = t$.

These cases are summarized in the pseudocode function $\mathsf{Oracle}(s, t, r)$ shown in Figure 3.

COROLLARY 1. *Fixed-density pseudo-necklaces can be generated in cool-lex Gray code order or co-lex order in constant amortized time.*

3.2 Testing if a string is a necklace or Lyndon word

The fastest known method for testing whether or not a string $\alpha = a_1 \cdots a_n$ is a necklace runs in $\Theta(n)$ time and is based on Duval's algorithm [2]. Using the block representation, a string α represented by $B_c \cdots B_1$ will be a necklace if it is less than or equal to each of its rotations $\alpha_j = B_j \cdots B_1 B_c \cdots B_{j+1}$ for $j = 1$ to $c - 1$. With this representation, Duval's algorithm can be applied to test the string in $\Theta(c)$ time.

```
function TestNecklace(int r) returns int
int i, p = 0
    if d = 0 or d = n then return 1
    for i := 0 to c − 1 do
        if r − i ≤ 0 then r := r + c
        if B_{c−i} < B_{r−i} then return 0
        if B_{c−i} > B_{r−1} then return n
        if r < c then p := p + s_{r−i} + t_{r−i}
    return p
end.
```

Figure 4: If $B_c \cdots B_1$ is a necklace, this function returns the length of its longest Lyndon prefix; otherwise it returns 0.

In this subsection we describe a new method that uses an extra piece of information. Let $suf(\alpha)$ denote the index r such that $B_r \cdots B_1$ is the lexicographically smallest suffix of $\gamma = B_{c-1} \cdots B_1$. Note that B_r is the lexicographically smallest block of γ and hence can be used in the pseudo-necklace oracle. Given $suf(\alpha)$, the following lemma can be used to optimize the test to determine if α is a necklace or Lyndon word.

LEMMA 1. *Let $\alpha = B_c \cdots B_1$ represent a binary string where $r = suf(\alpha)$. Then,*

- ▷ *α is a necklace $\Leftrightarrow \alpha \leq \alpha_r$ and*
- ▷ *α is a Lyndon word $\Leftrightarrow \alpha < \alpha_r$.*

PROOF: If α is a necklace (Lyndon word) then by definition $\alpha \leq \alpha_r$ ($\alpha < \alpha_r$). For the other direction, observe from the definition of r that $\beta = B_r \cdots B_1$ is a Lyndon word. Suppose $\alpha = \alpha_r$. Then $\alpha = \beta^m$ since β is a Lyndon word and thus α is a periodic necklace. Now suppose $\alpha < \alpha_r$. If the first r blocks of α are less than $B_r \cdots B_1$ then α must be the lexicographically smallest suffix of α and hence a Lyndon word. Otherwise, if the first r blocks of α are equal to $B_r \cdots B_1$, then there must exist another substring of r blocks in α that is less than $B_r \cdots B_1$ for $\alpha < \alpha_r$ which contradicts the definition of $r = suf(\alpha)$. □

As an example of how we apply this Lemma, consider the string $\alpha = 00011010100011001$ with $c = 5$ blocks. For this string $r = suf(\alpha) = 2$ since B_2B_1 is the lexicographically smallest suffix of $B_4 \cdots B_1 = 010100011001$. To test if this string is a necklace we first compare $B_c = B_5$ with $B_r = B_2$. Since they are the same we compare the next blocks $B_{c-1} = B_4 = (1, 1) = 01$ and $B_{r-1} = B_1 = (2, 1) = 001$. But since B_4 is greater than B_1 we can conclude after 2 block comparisons that the original string is not a necklace.

Following the example, the simple function TestNecklace(r) shown in Figure 4 can be used to test whether or not a string $\alpha = B_c \cdots B_1$ is a necklace. If it is a necklace, the function returns the length of its longest Lyndon prefix. If the necklace is aperiodic (i.e., it is a Lyndon word), this value is n; otherwise it will correspond to the length of $B_r \cdots B_1$ except for the special cases when $d = 0$ ($\alpha = 0^n$) or $d = n$ ($\alpha = 1^n$) in which case the length of the longest Lyndon prefix is 1. If the string is not a necklace, the function returns 0. Observe that in many cases, this tester requires far fewer than c block comparisons.

Since the oracle we construct for fixed-density necklaces and Lyndon words will apply this necklace tester, we must maintain the parameter $r = suf(\alpha)$ within the main recursive function Gen(s, t) as α gets updated. The only way that r will change is if $B_{c-1} \cdots B_1 < B_r \cdots B_1$ in which case we update r to $c - 1$. This test can be performed by calling the function TestSuffix(r) outlined in Figure 5. The function takes in the current value r and returns TRUE if the suffix starting at B_{c-1} is smaller than the suffix starting at B_r. The function is straightforward except for one optimization: inside the **for** loop if $c - 1 - i = r$ then the string $B_{c-1} \cdots B_1$

```
function TestSuffix (int r) returns boolean
int i
    for i := 0 to r − 1 do
        if c − 1 − i = r then return TRUE
        if B_{c−1−i} < B_{r−i} then return FALSE
        if B_{c−1−i} > B_{r−1} then return TRUE
    return FALSE
end.
```

Figure 5: This function returns TRUE if and only if $B_{c-1} \cdots B_1 < B_r \cdots B_1$.

being tested must be of the form $\beta\beta\delta$ for non-empty strings β and δ. From the definition of r we have $\beta < \delta$ and hence $\gamma < \beta\delta$. Thus the function returns TRUE.

Using this function to maintain the parameter r, we update the recursive call to Gen(s, t) with the following:

```
if TestSuffix(r) then Gen(s, t, c − 1)
else Gen(s, t, r)
```

The initial call becomes Gen$(n - d, d, 1)$. Again, observe that this value of r also satisfies the requirement for the pseudo-necklace oracle since B_r will be a smallest block in $B_{c-1} \cdots B_1$.

3.3 An oracle for fixed-density necklaces and Lyndon words

Before we present an oracle for fixed-density necklaces and Lyndon words we first prove the following straightforward lemma.

LEMMA 2. *If $\beta = 0^{s-1}1^{t-j}01^j\gamma$ is a pseudo-necklace where $t > j$ and γ is either empty or begins with 0, then $\beta' = 0^{s-1}1^{t-(j+1)}01^{j+1}\gamma$ is a Lyndon word.*

PROOF: Since β is a pseudo-necklace every block in β will be greater than or equal to its first block. This means that the first block of β', which is smaller than the first block of β, is clearly smaller than every block in $01^{j+1}\gamma$. Thus β' is a Lyndon word. □

Consider a string $\alpha = 0^s1^t\gamma$ where $r = suf(\alpha)$. If j is the value returned by PseudoOracle(s, t, r) then we can use the necklace tester TestNecklace(r) to determine whether or not the string $\beta = 0^{s-1}1^{t-j}01^j\gamma$ is a necklace. If β is a necklace then the fixed-density necklace oracle also returns j; otherwise, by the previous lemma it will return $j + 1$. Note that in order to apply the necklace tester on β, we temporarily convert α to β by calling UpdateBlock(s, t, j). This means that we also have to update r by calling TestSuffix. Pseudocode for this oracle is given in Figure 6. The oracle is easily modified for Lyndon words by replacing the condition $p > 0$ with $p = n$ in the final **if** statement.

In the following subsection we will prove that by using this oracle, the cool-lex Gray algorithm for fixed-density necklaces and Lyndon words runs in constant amortized time.

```
function NecklaceOracle( int s, int t , int r ) returns int
int j, p
    j := PseudoOracle(s, t, r)
    UpdateBlock(s, t, j)
    if TestSuffix(r) then p := TestNecklace(c − 1)
    else p := TestNecklace(r)
    RestoreBlock(s, t, j)
    if p > 0 then return j
    return j + 1
end.
```

Figure 6: Oracle for fixed-density necklaces.

3.4 Analysis

The number of recursive calls to Gen(s, t, r) for any bubble language is equal to the number of strings generated (visited). Also note that except for the functions TestSuffix(r) and TestNecklace(r) the work done by each recursive call is constant. Thus, to prove that our algorithm for generating fixed-density necklaces or Lyndon words is CAT, we need only show that the total work done by calls to these two functions is proportional to $N(n, d)$.

Notice that there are two calls to the function TestSuffix(r) for each recursive call to Gen(s, t, r): one used to update r and one that is called from the oracle. In each case, the string tested will be unique which means each string (pseudo-necklace) will be tested at most twice. Similarly, each call to TestNecklace(r) gets called on a unique pseudo-necklace. Now observe that each of these functions has one simple **for** loop. We can account the first and last iteration of each for loop as a constant amount of work proportional to $|\mathbf{P}(n, d)|$ which is proportional to $N(n, d)$ (this follows from the necklace oracle which considers each pseudo-necklace). Thus it remains to account for the remaining **for** loop iterations which will be comparing two equal blocks of the string α being tested.

We consider each function TestSuffix(r) and TestNecklace(r) separately and assume that α and β are two different strings that get tested where each A_i and B_i denotes a block of the form 0^+1^+:

- $\alpha = A_c \cdots A_1$ with $r = suf(\alpha)$,
- $\beta = B_{c'} \cdots B_1$ with $r' = suf(\beta)$.

Considering these strings as being circular, assume $A_0 = A_c$ and $A_{-1} = A_{c-1}$ etc. We also use the notation A_x^- to denote the string A_x with one 0 removed. Note that A_x^- is not a block when A_x contains only one 0 (unless $x = c$).

Analyzing: TestSuffix(r)

For this function the equal blocks being compared for a string α are A_{c-1-i} and A_{r-i} where $0 < i < r-1$ and $c-1-i > r$. For such a comparison to be made for a given i, it must be that $A_{c-1} \cdots A_{c-1-i} = A_r \cdots A_{r-i}$. Since $i > 0$ this implies $A_c = A_r$. Consider the following mapping f:

$$f(\alpha, i) = 0A_c \cdots A_{c-i}A^-_{c-i-1}A_{c-i-2} \cdots A_1.$$

Clearly this mapping preserves length and density. Given the constraints outlined earlier for α and i, we also have $f(\alpha, i) \in \mathbf{N}(n, d)$ since the leading block will have the most 0s over all blocks. By proving that for

all valid α and i the mapping $f(\alpha, i)$ is 1-1, we prove that the number of equal block comparisons under consideration is at most $N(n, d)$:

PROOF: Suppose that $f(\alpha, i) = f(\beta, j)$ where $\alpha \neq \beta$. If $r = r'$ then we must have $A_r \cdots A_1 = B_r \cdots B_1$ since these blocks are unaffected by f by the restrictions on r, r'. This implies that $A_{c-1} \cdots A_{c-i-1} = B_{c'-1} \cdots B_{c'-i-1}$ where WLOG $i \geq j$. If $i = j$ then it must be that $\alpha = \beta$, a contradiction. If $i > j$ then clearly $f(\alpha, i) \neq f(\beta, j)$. Now, WLOG assume $r > r'$. Because of the restriction $c - i - 1 > r$, the substring $A_r \cdots A_1$ remains unchanged in $f(\alpha, i)$. This means that β must also have at least r blocks. By definitions of r and r' we must have $A_r \cdots A_1 < B_r \cdots B_1$. But by the nature of f the string $B_r \cdots B_1$ will either stay the same or become lexicographically larger in $f(\beta, j)$. But this means that $f(\alpha, i) \neq f(\beta, j)$, a contradiction.

Since it is obvious that $f(\alpha, i) \neq f(\alpha, j)$ if $i \neq j$, we have just proved that f is 1-1. □

Since the number of equal block comparisons in this case is at most $N(n, d)$, the total number of comparisons required by all calls to the function TestSuffix(r) is bounded by a constant times $N(n, d)$.

Analyzing: TestNecklace(r)

For this function the equal blocks being compared for a string α are A_{c-i} and A_{r-i} where $0 < i < c - 1$. For such a comparison to be made for a given i, it must be that $A_c \cdots A_{c-i} = A_r \cdots A_{r-i}$. For this case we make two simplifying assumptions: (1) $c > 3$ (otherwise c is considered constant) and (2) $i < c/2$ which will account for at least half of the equal block comparisons under consideration.

Next we partition the comparisons into two groups: those where $c - i > r$ and those where $c - i \leq r$. If $c - i > r$, then we can again use the mapping f and the identical proof from the previous case to map each comparison to a unique necklace in $\mathbf{N}(n, d)$. Thus this first sub-case accounts for at most $N(n, d)$ comparisons. The more difficult sub-case is when $c - i \leq r$. Since $A_c \cdots A_{c-i} = A_r \cdots A_{r-i}$, we have:

$$A_c \cdots A_{r-i} = (A_c \cdots A_{r+1})^{k+1} A_c \cdots A_{c-m}$$

where $k = \lfloor \frac{i+1}{c-r} \rfloor$ and $m = i - (c - r)k$. Thus the blocks $A_c \cdots A_{r+1}$ completely determine the blocks $A_r \cdots A_{r-i}$. Now consider the mapping g for each valid α and i (it is well defined because of our earlier restrictions on c and i):

$$g(\alpha, i) = 0A_c \cdots A_{r+1} 0A_r \cdots A_{c-i} A^-_{c-i-1} A^-_{c-i-2} A_{c-i-3} \cdots A_1.$$

Clearly, $g(\alpha, i)$ maintains the length and density. We also have $g(\alpha, i) \in \mathbf{N}(n, d)$ since the blocks $0A_c$ and $0A_r$ will have the most 0s over all blocks and the $c - r - 1$ blocks after A_c are the same as the $c - r - 1$ blocks after A_r. Finally, by proving that for all valid α and i the mapping $g(\alpha, i)$ is 1-1 we prove that the number of equal block comparisons in this sub-case is at most $N(n, d)$:

PROOF: Suppose that $g(\alpha, i) = g(\beta, j)$ where $\alpha \neq \beta$. Notice that $A_c \cdots A_{r+1} = B_{c'} \cdots B_{r'+1}$ since the blocks $0A_r$ and $0B_{r'}$ must line up in $g(\alpha, i)$ and $g(\beta, j)$. But as mentioned earlier, these substrings completely define $A_c \cdots A_{r-i}$ and $B_c \cdots B_{r'-j}$. Thus WLOG if $i < j$ then $A_{c-i-1} = B_{c'-i-1}$. But this contradicts $g(\alpha, i) = g(\beta, j)$. Otherwise if $i = j$ we have $A_c \cdots A_{r-i} = B_{c'} \cdots B_{r'-i}$. Thus $A_{c-i-1} = B_{c'-i-1}$ which means that $A_{c-i-2} = B_{c'-i-2}$ for $g(\alpha, i) = g(\beta, j)$. But since $\alpha \neq \beta$ we must have $A_{c-i-3} \cdots A_1 \neq B_{c'-i-3} \cdots B_1$ which contradicts $g(\alpha, i) = g(\beta, j)$. □

Since each sub-case requires at most $N(n,d)$ comparisons given our assumptions, the overall number of comparisons required by the function TestNecklace(r) is bounded by a constant times $N(n,d)$.

As a result of this analysis we obtain the following theorem:

THEOREM 2. *Fixed-density necklaces can be generated in cool-lex Gray code order or co-lex order in constant amortized time.*

Since $L(n,d)$ is proportional to $N(n,d)$ for $0 < d < n$ we obtain the following corollary:

COROLLARY 2. *Fixed-density Lyndon words can be generated in cool-lex Gray code order or co-lex order in constant amortized time.*

4 Applications

We can apply the CAT algorithms presented in the previous section (i) to generate a Gray code for all pseudo-necklaces, necklaces, or Lyndon words ordered by density in constant amortized time and (ii) to produce a fixed-density de Bruijn sequence in constant time for every n bits.

4.1 Necklaces and Lyndon words in Gray code order

In order to generate all pseudo-necklaces, necklaces, or Lyndon words in Gray code order by density, we can simply append together the lists obtained from the cool-lex fixed-density algorithm [11]. Also notice that if we consider the even densities in increasing order followed by the odd densities in decreasing order, then we will obtain a Gray code for necklaces or Lyndon words that is *cyclic*. The interfaces between the densities will differ by the flipping of at most 4 bits. Sample output illustrating the interfacing between densities is given in Table 1 for $n = 6$.

The following corollaries are a direct result of Theorem 1.

COROLLARY 3. *Binary necklaces or Lyndon words of length n in can be listed in a Gray code order by density in constant amortized time.*

COROLLARY 4. *Binary necklaces or Lyndon words of length n can be listed in a cyclic Gray code order in constant amortized time.*

4.2 Short hand de Bruijn sequences for fixed-density strings

As an interesting application, the fixed-density necklace cool-lex Gray code can be applied in the first explicit construction of fixed-density de Bruijn sequences. This application is briefly illustrated below; details can be found in [12]. Consider the following string

$$00011100110101001011. \tag{1}$$

Its successive substrings of length five are given below:

$$00011, 00111, 01110, 11100, 11001, \ldots, 01011, 10110, 01100, 11000, 10001. \tag{2}$$

Density	Necklaces	Lyndon words
$d=0$	0 0 0 0 0 0	
$d=1$	0 0 0 0 0 **1**	0 0 0 0 0 1
$d=2$	0 0 **1** 0 0 1	
	0 0 0 1 0 1	0 0 0 **1** 0 1
	0 0 0 0 1 1	0 0 0 0 1 1
$d=3$	0 0 **1** 0 1 1	0 0 **1** 0 1 1
	0 1 0 1 0 1	
	0 0 1 1 0 1	0 0 1 1 0 1
	0 0 0 1 1 1	0 0 0 1 1 1
$d=4$	0 **1** 0 1 1 1	0 **1** 0 1 1 1
	0 1 1 0 1 1	
	0 0 1 1 1 1	0 0 1 1 1 1
$d=5$	0 **1** 1 1 1 1	0 **1** 1 1 1 1
$d=6$	**1** 1 1 1 1 1	

(a)

Density	Necklaces	Lyndon words
$d=0$	0 0 0 0 0 0	
$d=2$	0 0 **1** 0 0 1	
	0 0 0 1 0 1	0 0 0 **1** 0 1
	0 0 0 0 1 1	0 0 0 0 1 1
$d=4$	0 **1** 0 1 1 1	0 **1** 0 1 1 1
	0 1 1 0 1 1	
	0 0 1 1 1 1	0 0 1 1 1 1
$d=6$	**1** 1 1 1 1 1	
$d=5$	0 **1** 1 1 1 1	0 **1** 1 1 1 1
$d=3$	0 0 **1** 0 1 1	0 0 **1** 0 1 1
	0 1 0 1 0 1	
	0 0 1 1 0 1	0 0 1 1 0 1
	0 0 0 1 1 1	0 0 0 1 1 1
$d=1$	0 0 0 0 0 **1**	0 0 0 0 0 1

(b)

Table 1: Sample Gray code listings for necklaces and Lyndon words for $n = 6$: (a) ordered by density (b) cyclic.

Interestingly, the $\binom{6}{3} = 20$ strings in (2) are all distinct. Furthermore, they represent all fixed-density binary strings with $n = 6$ and $d = 3$ except that the last (redundant) bit is omitted. This *fixed-density de Bruijn sequence* in (1) was obtained by concatenating the necklaces of length 6 and density 3 produced by the cool-lex Gray code (see Table 1) in reverse order and reducing the periodic strings to their longest Lyndon prefixes. Thus we concatenate the strings: 000111, 001101, 01, 001011.

Using our algorithm to generate fixed-density necklaces in cool-lex order, we can generate the reversal of such a de Bruijn sequence by modifying the Visit() function to:

- ▷ accept the parameter $r = suf(\alpha)$,
- ▷ determine p, the length of the longest Lyndon prefix of the necklace by calling TestNecklace(r) and
- ▷ output $a_p a_{p-1} \cdots a_1$.

Recall that the work required by all calls to TestNecklace(r) has been proved to be proportional to $N(n, d)$. Since the total length of a fixed-density de Bruijn sequence for a given n and d is proportional to $n \cdot N(n, d)$ we obtain the following theorem.

THEOREM 3. *Fixed-density de Bruijn sequences can be generated in constant time for each n bits on average.*

5 Summary and open problems

In this paper we optimize the recursive algorithm to list fixed-density necklaces and Lyndon words in the cool-lex Gray code order described in [11]. The result is a CAT algorithm that can be applied to (i) generate all necklaces and Lyndon words in Gray code order in constant amortized time and (ii) to generate fixed-density de Bruijn sequences in constant time per n bits on average. Along the way we introduce a new combinatorial object called a pseudo-necklace and provide a CAT algorithm to generate them as well. A unified C implementation for each algorithm discussed in this paper is available at `http://www.socs.uoguelph.ca/~sawada/programs.html`.

There remains several interesting avenues for future research:

- ▷ Is there a loop-free algorithm to generate fixed-density necklaces?
- ▷ Can the cool-lex order of fixed-density necklaces and Lyndon words be efficiently ranked/unranked?
- ▷ Can a variation of cool-lex be used to find a Gray code for fixed-density bracelets or unlabeled necklaces?
- ▷ Are there interesting applications for pseudo-necklaces? Are there simple enumeration formulae for them?

6 Acknowledgements

Thanks to Frank Ruskey for many helpful discussions and valuable comments.

References

[1] C. Degni and A. Drisko, Gray-ordered binary necklaces, Electron. J. Combin., Vol. 14 No. 1 (2007) Research Paper 7, 23 pp. (electronic).

[2] J.P. Duval, Factorizing words over an ordered alphabet, J. Algorithms Vol. 4 No. 4 (1983) 363-381.

[3] H. Fredricksen and I. J. Kessler, An algorithm for generating necklaces of beads in two colors, Discrete Math., Vol. 61 No. 2-3 (1986) 181-188.

[4] H. Fredricksen and J. Maiorana, Necklaces of beads in k colors and k-ary de Bruijn sequences, Discrete Math., Vol. 23 No. 3 (1978) 207-210.

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

[6] D. E. Knuth, The Art of Computer Programming, Volume 4: Generating All Tuples and Permutations, Fascicle 2, Addison-Wesley, February 2005, 150 pages.

[7] D. E. Knuth, The Art of Computer Programming, Volume 4: Generating all Combinations and Partitions, Fascicle 3, Addison-Wesley, July 2005, 150 pages.

[8] D. E. Knuth, The Art of Computer Programming, Volume 4: Generating All Trees; History of Combinationatorial Generation,, Fascicle 4, Addison-Wesley, February 2006, 150 pages.

[9] F. Ruskey, C. Savage, and T. Wang, Generating Necklaces, Journal of Algorithms, 13 (1992) 414430.

[10] F. Ruskey and J. Sawada, An efficient algorithm for generating necklaces of fixed-density, SIAM J. Comput., 29 (1999) 671684.

[11] F. Ruskey, J. Sawada, and A. Williams, Binary bubble languages and cool-lex Gray codes, submitted 2010.

[12] F. Ruskey, J. Sawada, and A. Williams, Fixed-density de Bruijn sequences, submitted 2010.

[13] J. Sawada, and A. Williams, Efficient oracles for generating binary bubble languages, submitted 2010.

[14] T. Ueda, Gray codes for necklaces, Discrete Math., Vol. 219 No. 1-3 (2000) 235-248.

[15] V. Vajnovszki, Gray code order for Lyndon words, Discrete Math. Theor. Comput. Sci., Vol. 9 No. 2 (2007) 145-151.

[16] V. Vajnovszki, More restrictive Gray codes for necklaces and Lyndon words, Inform. Process. Lett. Vol. 106 No. 3 (2008), 96-99.

[17] T. Wang and C. Savage, A Gray code for necklaces of fixed-density, SIAM J. Discrete Math., Vol. 9 No. 4 (1996) 654-673.

[18] M. Weston and V. Vajnovszki, Gray codes for necklaces and Lyndon words of arbitrary base, Pure Math. Appl. (PU.M.A.), Vol. 17 No. 1-2 (2006),175-182.