Necklaces and Lyndon words in colexicographic and binary reflected Gray code order

Joe Sawada Aaron Williams Dennis Wong

September 25, 2017

Abstract

We present the first efficient algorithm to list necklaces, Lyndon words, or pseudo-necklaces of length n in colexicographic order. The algorithm has two interesting properties. First, it can be applied to construct a de Bruijn sequence of order n in $O(1)$-time per bit. Second, it can easily be modified to efficiently list necklaces, Lyndon words, or pseudo-necklaces of length n in binary reflected Gray code order.

1 Introduction

Two fundamental pursuits in the area of discrete algorithms are (1) to discover efficient algorithms to exhaustively generate basic combinatorial objects and (2) to construct single instances of more complex combinatorial objects. Often, these pursuits are related. For instance, consider the problem of constructing a single binary *de Bruijn sequence* of order n, which is a binary sequence of length 2^n that when considered cyclicly contains each binary string of length n as a substring. Perhaps the most well-known construction of such a sequence requires the exhaustive generation of necklaces [5, 6], where a *necklace* is the lexicographically smallest string in equivalence class of strings under rotation. Specifically, by concatenating the longest aperiodic (primitive) prefixes of the necklaces of length n in lexicographic order, we obtain a de Bruijn sequence of order n. For example, the lexicographic ordering of the 14 binary necklaces of length 6, with their longest aperiodic prefixes in bold, is as follows:

000000, **000001**, **000011**, **000101**, **000111**, **001**001, **001011**,
001101, **001111**, **01**0101, **010111**, **011**011, **011111**, **1**11111.

The corresponding de Bruijn sequence of order 6 with length $2^6 = 64$ is the concatenation of the bolded strings above:

0000001000011000101000111001001011001101001111010101110110111111.

Amazingly, it was only recently discovered by Dragon et al. [4] that a de Bruijn sequence (coined as the "Grandmama") can also be constructed by applying the same concatenation scheme to necklaces, but in colexicographic (colex) order. A conjecture as to why this result had not been discovered earlier is also provided in [4]. However, unlike the lexicographic ordering of necklaces, which can be generated in $O(1)$-amortized time [5, 9], there was previously no known efficient algorithm to list necklaces in colex order. This is stated as an open problem in [4]. We extend this problem to additionally consider Lyndon words and pseudo-necklaces which are defined in Section 2.1.

Problem #1: Find an efficient algorithm to list necklaces, Lyndon words, or pseudo-necklaces of length n in colex order.

In addition to generating combinatorial objects efficiently, it is often useful for the ordering to have special properties. For instance, the binary reflected Gray code (BRGC) [7], that lists all binary strings of length n such that successive strings differ by a single bit, has many known applications [8]. Rather surprisingly, Vajnovszki [14] proved that when only the necklaces or Lyndon words are considered in this ordering, they appear as a *2-Gray code*: successive strings differ in at most 2 bit positions. Again, no known efficient algorithm to generate these objects in BRGC order was previously known and it was stated as an open problem in [14].

Problem #2: Find an efficient algorithm to list necklaces, Lyndon words, or pseudo-necklaces of length n in BRGC order.

For comparison, we list the necklaces of length 6 in three different orders (formally defined in the next section) in the following table:

Necklaces of length 6

Lex order	Colex Order	BRGC order
000000	000000	000000
000001	000001	000011
000011	001001	011011
000101	000101	001011
000111	010101	001111
001001	001101	111111
001011	000011	011111
001101	001011	010111
001111	011011	000111
010101	000111	000101
010111	010111	010101
011011	001111	001101
011111	011111	001001
111111	111111	000001

In addition to these three orderings, other necklace orderings considered include a Gray code by Vajnovszki and Weston [13, 16], a cool-lex Gray code by Sawada and Williams [12], a fixed-density Gray code by Wang and Savage [15], and a conjectured Gray code by Degni and Drisko [3]. Savage and Wang [15] also proved the non-existence of 1-Gray codes (where successive strings differ by a single bit) for necklaces of even length using a simple counting argument.

In this paper, we solve both open problems, providing algorithms that:

1. Exhaustively generate all necklaces, Lyndon words, or pseudo-necklaces in colex order in $O(1)$-amortized time per string.

2. Exhaustively generate all necklaces, Lyndon words, or pseudo-necklaces in BRGC order in $O(1)$-amortized time per string.

The first result can be immediately applied to generate the "Grandmama" de Bruijn sequence in $O(1)$-time per bit. Pseudo-necklaces were first used in [12] and are formally defined in the next section.

The remainder of this paper is outlined as follows. In Section 2 we present the necessary definitions and background. In Section 3 we present our algorithms to efficiently list necklaces, Lyndon words and pseudo-necklaces in either colex or BRGC order. In Section 4 we present the additional minor details required to construct the "Grandmama" de Bruijn sequence. In Section 5 we prove that pseudo-necklaces of length n, like necklaces and Lyndon words, form a 2-Gray code when considered in BRGC order.

2 Background and Notation

All strings considered in this paper are binary. Our algorithms use a run-length representation for binary strings using a series of *blocks* which are maximal substrings of the form 0^*1^*. Each block B_i can be represented by two integers (s_i, t_i) corresponding to the number of 0s and 1s respectively. For example, the string $\alpha = 000110100011001$ can be represented by $B_4B_3B_2B_1 = (3,2)(1,1)(3,2)(2,1)$. Maintaining this *block representation* is critical to the efficiency of the 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 our discussion, 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

We say a string α is *periodic* if $\alpha = \beta^t$ for some string β and $t \geq 2$. If a string is not periodic, it is said to be *aperiodic* (or *primitive*). 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 $\alpha = a_1a_2\cdots a_n = B_bB_{b-1}\cdots B_1$ is a *pseudo-necklace* if $B_b \leq B_i$ for all $1 \leq i < b$. Pseudo-necklaces were first defined in [12] and they are used as a stepping stone in our algorithms for necklaces and Lyndon words. We will use the following notation to denote these objects:

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

Note that $\mathbf{L}(n) \subseteq \mathbf{N}(n) \subseteq \mathbf{P}(n)$. Enumeration formulae for necklaces and Lyndon words are well known. For instance, see [2]. However it is an open problem to find a simple enumeration formula for pseudo-necklaces. The following lemma follows from the definition of a pseudo-necklace.

Lemma 2.1 *Let $\gamma = a_{j+1}a_{j+2}\cdots a_n$ be the suffix of some length n pseudo-necklace, where $1 \leq j < n$. Let $B_bB_{b-1}\cdots B_1 = (s_b,t_b)(s_{b-1},t_{b-1})\cdots(s_1,t_1)$ be the block representation of γ with a smallest block at index r. Then 1γ is the suffix of some pseudo-necklace of length n if and only if*

- *$s_b = 0$ and $(j-1, t_b+1) \leq B_r$ or*
- *$s_b > 0$ and $(j-1,1) \leq B_r$.*

Moreover if $0^{j-1}1\gamma$ is not a pseudo-necklace then $0^j\gamma$ is the unique pseudo-necklace of length n and suffix γ.

Proof. Clearly $\alpha = 0^{j-1}1\gamma$ is the length n string with suffix 1γ that has the smallest first block B'. If α is a pseudo-necklace, then clearly the bulleted condition holds and vice-versa. If it is not a pseudo-necklace, then $B' > B_r$. Clearly, the first block in any other string $a_1a_2\cdots a_j\gamma$ with at least one 1 in its length j prefix must greater than B'. Thus any other such string will also not be a pseudo-necklace. Therefore if α is not a pseudo-necklace, then since γ is the suffix of some length n pseudo-necklace, that necklace must uniquely be $0^j\gamma$. □

2.2 Colex and BRGC order

The colex ordering used in this paper is defined only on strings with the same length. Thus, in this special case, we can define *colex order* to be lexicographic order on strings when they are read from right to left. A general definition of colex order can be found in [8].

The following definitions and algorithms may seem a bit tedious; however, it will allow us to simplify our algorithm discussion later for necklaces and Lyndon words.

Let $COLEX(n)$ denote the listing of binary strings of length n in colex order. Let the notation $\mathcal{L} \cdot x$ denote the listing $\mathcal{L}$ with the character x appended to the end of each string. Then $COLEX(n)$ can be defined recursively as follows:

$$COLEX(n) = \begin{cases} 0, 1 & \text{if } n = 1 \\ COLEX(n-1) \cdot 0,\ COLEX(n-1) \cdot 1 & \text{if } n > 1. \end{cases}$$

Let $BRGC(n)$ denote the listing of binary strings of length n in BRGC order. Let $\overline{BRGC}(n)$ denote the listing $BRGC(n)$ in reverse order. Then $BRGC(n)$ can be defined recursively as follows:

$$BRGC(n) = \begin{cases} 0, 1 & \text{if } n = 1 \\ BRGC(n-1) \cdot 0,\ \overline{BRGC}(n-1) \cdot 1 & \text{if } n > 1. \end{cases}$$

This definition of BRGC order is the same as the one used by Vajnovszki [14] as elaborated in [11]. When the strings are read from right-to-left, we obtain the classic definition of the BRGC [7].

These recurrences can easily be turned into the respective recursive algorithms GEN shown in Algorithm 1. In these functions, $\alpha = a_1 a_2 \cdots a_n$ is the binary string being generated, where $\gamma = a_{j+1} a_{j+2} \cdots a_n$ is the current suffix already generated. $B_b B_{b-1} \cdots B_1$ denotes the block representation of γ recalling that $B_i = (s_i, t_i)$; this representation is required later for the efficient generation of necklaces and Lyndon words. Each function GEN makes one call each to the functions ADDZERO and ADDONE, which update the data structures and make recursive calls to GEN for the ordering being generated. The function VISIT() processes, or outputs, the current string α. Note, the parameter rev is used only by the BRGC ordering to determine whether or not the listing should be produced in the reverse order.

Algorithm 1 The function GEN to list binary strings in colex order (left) and BRGC order (right).

```
1: function GEN(j, b, rev)
2:     if j = 0 then VISIT( )
3:     else
4:         ADDZERO(j, b, −)
5:         ADDONE(j, b, −)
```

```
1: function GEN(j, b, rev)
2:     if j = 0 then VISIT( )
3:     else
4:         if rev then
5:             ADDONE(j, b, FALSE)
6:             ADDZERO(j, b, TRUE)
7:         else
8:             ADDZERO(j, b, FALSE)
9:             ADDONE(j, b, TRUE)
```

The functions ADDZERO and ADDONE sets a_j to 0 and 1 respectively, update the block representation accordingly, and make the appropriate recursive call. These functions are illustrated in Algorithm 2. If $a_j \leftarrow 1$ and $s_b > 0$, then the number of blocks in $a_j a_{j+1} \cdots a_n$ is $b + 1$ and a new block is created. Otherwise the number of blocks remains the same. Since the data structures $a_1 a_2 \cdots a_n$ and $B_b B_{b-1} \cdots B_1$ are maintained globally, the data structures need to be restored after the recursive calls. The initial call for both orderings is GEN$(n, 1, 0)$.

Algorithm 2 Functions to assign a value to a_j, update the block representation $B_b B_{b-1} \cdots B_1$ accordingly, and make the appropriate recursive call.

```
1: function AddZero(j, b, rev)
2:     a_j ← 0
3:     s_b ← s_b + 1
4:     Gen(j−1, b, rev)
5:     s_b ← s_b − 1
```

```
1: function AddOne(j, b, rev)
2:     a_j ← 1
3:     if s_b = 0 then
4:         t_b ← t_b + 1
5:         Gen(j−1, b, rev)
6:         t_b ← t_b − 1
7:     else
8:         B_{b+1} ← (0, 1)
9:         Gen(j−1, b + 1, rev)
```

Since each recursive call to GEN (for either colex or BRGC order) requires a constant amount of work and the number of nodes in the recursive computation tree contains $2^{n+1} - 1$ nodes (it is a complete binary tree), we obtain the following well-known remark.

Remark 2.2 *The algorithms* GEN *for colex order and BRGC order to list all binary strings of length n run in $O(1)$-amortized time per string.*

A *k-Gray code* for a set of strings $\mathbf{S}$ is an ordered list for $\mathbf{S}$ such that the Hamming distance between any two consecutive words in the list is at most k (if it exists). Such an ordering is said to be *cyclic* if the Hamming distance between the first and last strings is also at most k. In [14], Vajnovszki proved that if necklaces or Lyndon words of length n are listed as they appear in $BRGC(n)$, then they form a cyclic 2-Gray code.

Theorem 2.3 *[14] The BRGC ordering induces a cyclic 2-Gray code on $\mathbf{N}(n)$ and $\mathbf{L}(n)$.*

We extend this result to pseudo-necklaces.

Theorem 2.4 *The BRGC ordering induces a cyclic 2-Gray code on $\mathbf{P}(n)$.*

The proof of this Theorem is presented later in Section 5. The BRGC does not induce a 1-Gray code on $\mathbf{P}(n)$ since the first two pseudo-necklaces in this order are 0^n and $0^{n-2}11$ for $n \geq 2$.

3 Necklaces, Lyndon words and pseudo-necklaces in colex and BRGC order

In this section we modify the colex and BRGC algorithms for binary strings to generate necklaces, Lyndon words and pseudo-necklaces. A naïve approach to generate necklaces in these orders is simply to test if each string is a necklace. Such a test can be done in $O(n)$ time using standard techniques [1]. Since the number of length n binary strings is $O(n)$ times the number of necklaces, this will result in an $O(n^2)$-amortized time algorithm to generate each necklace. A similar analysis can be applied to Lyndon words. In the following sections we describe details that lead to $O(1)$-amortized time algorithms. We begin with modifications required to generate pseudo-necklaces efficiently.

3.1 Pseudo-necklaces

By making three relatively minor changes to the algorithms presented in Section 2.2, we can efficiently generate pseudo-necklaces in colex and BRGC order.

1. Initialize $a_1 a_2 \cdots a_n$ to 0^n and maintain the prefix $a_1 a_2 \cdots a_j = 0^j$ at the start of each recursive call to GEN.

2. Add a new integer parameter r, corresponding to the index of a smallest block in $\gamma = B_b B_{b-1} \cdots B_1$, as the last parameter in the previously defined functions: GEN, ADDZERO, and ADDONE.

3. Define a new function EXTEND, based on Lemma 2.1, to test in $O(1)$-time whether 1γ is the suffix of some pseudo-necklace of length n.

These modifications can be implemented as follows. (1.) To maintain the prefix 0^j at the start of each recursive call, a_j must be restored to 0 at the end of the function ADDONE. Maintenance of this prefix also means we no longer need to set $a_j \leftarrow 0$ in ADDZERO. (2.) The parameter r, initially set to 1, is updated only when a_j is set to 0 and the updated block B_b is less than B_r. In this case r gets updated to b. (3.) The function EXTEND, which requires the new parameter r, is given in Algorithm 3. It follows directly from Lemma 2.1 except for the first two **if** statements which handle the special cases required for the pseudo-necklaces 0^n and 1^n, respectively. To apply this function, instead of testing if $j = 0$ at the start of each function GEN, it tests whether or not the function EXTEND returns TRUE. If it is FALSE, then the only prefix of γ that will produce a pseudo-necklace is 0^j (from the proof of Lemma 2.1). Since this prefix is already initialized to 0^j, the string can be visited immediately, where the block representation s_b must be incremented by j.

Pseudocode that applies these changes is given in Algorithm 4 and Algorithm 5.

Algorithm 3 A function that returns if $1a_{j+1}a_{j+2}\cdots a_n$ is the suffix of some length n pseudo-necklace.

```
1: function EXTEND(j, b, r)
2:     if j < n and a_n = 0 then return FALSE
3:     if j > 0 and s_r = 0 then return TRUE
4:     if s_b = 0 and (j − 1, t_b + 1) ≤ B_r then return TRUE
5:     if s_b > 0 and (j − 1, 1) ≤ B_r then return TRUE
6:     return FALSE
```

Algorithm 4 Updated functions ADDZERO and ADDONE for pseudo-necklaces.

```
1: function ADDZERO(j, b, rev, r)
2:     s_b ← s_b + 1
3:     if B_b < B_r then GEN(j−1, b, rev, b)
4:     else GEN(j−1, b, rev, r)
5:     s_b ← s_b − 1
```

```
1: function ADDONE(j, b, rev, r)
2:     a_j ← 1
3:     if s_b = 0 then
4:         t_b ← t_b + 1
5:         GEN(j−1, b, rev, r)
6:         t_b ← t_b − 1
7:     else
8:         B_{b+1} ← (0, 1)
9:         GEN(j−1, b + 1, rev, r)
10:    a_j ← 0
```

Algorithm 5 The function GEN to list pseudo-necklaces in colex order (left) and BRGC order (right).

```
1: function GEN(j, b, rev, r)
2:     if not EXTEND(j, b, r) then
3:         s_b ← s_b + j
4:         VISIT(b, r)
5:         s_b ← s_b − j
6:     else
7:         ADDZERO(j, b, −, r)
8:         ADDONE(j, b, −, r)
```

```
1: function GEN(j, b, rev, r)
2:     if not EXTEND(j, b, r) then
3:         s_b ← s_b + j
4:         VISIT(b, r)
5:         s_b ← s_b − j
6:     else
7:         if rev then
8:             ADDONE(j, b, FALSE, r)
9:             ADDZERO(j, b, TRUE, r)
10:        else
11:            ADDZERO(j, b, FALSE, r)
12:            ADDONE(j, b, TRUE, r)
```

The function VISIT(b, r) can be implemented to print out the string $a_1a_2 \cdots a_n$ and/or the block representation $B_bB_{b-1} \cdots B_1$. The parameter r is used in the next section when we consider necklaces and Lyndon words. The initial call is GEN$(n, 1, 0, 1)$ for both orders with $B_1 = (s_1, t_1)$ initialized to $(0, 0)$. Since each call to GEN either generates a pseudo-necklace or makes two recursive calls, and each recursive call requires $O(1)$-time, we obtain the following theorem.

Theorem 3.1 *Pseudo-necklaces of length n can be generated in $O(1)$-amortized time and $O(n)$ space in either colex order or BRGC order.*

3.2 Necklaces and Lyndon words

To efficiently generate necklaces and Lyndon words, we apply similar techniques to the ones used in [12] to efficiently generate necklaces and Lyndon words with fixed-density (the number of 1s is fixed) in colex and cool-lex order. The main idea is to strengthen the definition of r that will allow for a more efficient test to determine whether or not a pseudo-necklace is a necklace or Lyndon word.

Recall for pseudo-necklaces that $\gamma = B_bB_{b-1} \cdots B_1 = a_{j+1}a_{j+2} \cdots a_n$ and the parameter r denotes an index such that B_r is a lexicographically smallest block in γ. For necklaces and Lyndon words we strengthen the definition of r to be $suf(\gamma)$ which is defined as follows:

$suf(\gamma) =$ the index r such that $B_rB_{r-1} \cdots B_1$ is the lexicographically smallest suffix of $\gamma = B_bB_{b-1} \cdots B_1$.

Observe that B_r is still a lexicographically smallest block of γ. As with pseudo-necklaces, this parameter will only be updated by the function ADDZERO. After s_b is incremented (a_j is already set to 0), the value for r can be updated using the function SUF(b, r) given in Algorithm 6. This function returns $suf(0\gamma)$ given the parameter $r = suf(\gamma)$. The function is straightforward except for one optimization noted in [12] that is relevant to the analysis[1]: inside the **for** loop if $b - i = r$, then $B_bB_{b-1} \cdots B_1$ is of the form $\beta\beta\delta$ for some non-empty strings β and δ. Moreover, from the definition of r, it must be that $\beta < \delta$ and hence $B_bB_{b-1} \cdots B_1 < B_rB_{r-1} \cdots B_1$.

Algorithm 6 Computing $suf(B_bB_{b-1} \cdots B_1)$ where $0\gamma = B_bB_{b-1} \cdots B_1 = a_ja_{j+1} \cdots a_n$ and $r = suf(a_{j+1}a_{j+2} \cdots a_n)$.

```
1: function SUF(b, r) returns int
2:     for i from 0 to r − 1 do
3:         if b − i = r then return b
4:         if B_{b−i} > B_{r−i} then return r
5:         if B_{b−i} < B_{r−i} then return b
6:     return r
```

If $\alpha = B_bB_{b-1} \cdots B_1$ and $suf(\alpha) = b$ then α is a Lyndon word by definition, and hence a necklace. If $suf(\alpha) < b$ then following lemma is equivalent to [12, Lemma 1] (where they use a slightly different definition of $suf(\alpha)$). It can be used to optimize the test to determine if α is a necklace or Lyndon word.

Lemma 3.2 *Let $\alpha = B_bB_{b-1} \cdots B_1$ represent a binary string where $r = suf(\alpha)$ and $r < b$. Then,*
- ▷ *α is a necklace if and only if $\alpha \leq B_rB_{r-1} \cdots B_1B_bB_{b-1} \cdots B_{r+1}$ and*
- ▷ *α is a Lyndon word if and only if $\alpha < B_rB_{r-1} \cdots B_1B_bB_{b-1} \cdots B_{r+1}$.*

The function TESTNECKLACE shown in Algorithm 7 applies this lemma to return the longest Lyndon prefix of $\alpha = B_bB_{b-1} \cdots B_1$ (when $suf(\alpha) < b$) if it is a necklace or 0 otherwise. This function is equivalent to the one presented in [12]. Applying this function, we need only make two modifications to the pseudo-necklace algorithm presented in the previous subsection:

[1]The inequalities in the second and third **if** statement shown in Figure 5 of [12] are incorrect and should be switched.

Algorithm 7 If $B_b B_{b-1} \cdots B_1$ is a necklace where $r = suf(\alpha) < b$, return the length of its longest Lyndon prefix; otherwise return 0.

```
1: function TESTNECKLACE(b, r) returns int
2:     if b = 1 then return 1
3:     p ← 0
4:     for i from 0 to b − 1 do
5:         if r − i ≤ 0 then r ← r + b
6:         if B_{b−i} < B_{r−i} then return 0
7:         if B_{b−i} > B_{r−i} then return n
8:         if r < b then p ← p + s_{r−i} + t_{r−i}
9:     return p
```

1. Modify the function ADDZERO from Algorithm 4 to update the parameter r appropriately: replace lines 3-4 with the single call GEN($j-1, b, rev$, SUF(b, r)).

2. In the function VISIT, apply TESTNECKLACE when $r < b$ to determine if the pseudo-necklace is a necklace or Lyndon word. If the function returns a value greater than 0, then it is a necklace; if it returns n, it is a Lyndon word. If $r = b$ the pseudo-necklace is a Lyndon word (as mentioned earlier).

By applying these changes we can generate only the pseudo-necklaces that are either necklaces or Lyndon words, while respecting the colex or BRGC ordering. A complete C program to generate necklaces, Lyndon words, or pseudo-necklaces in either colex order or BRGC order is given in the Appendix.

3.2.1 Analysis

Let the *density* of a binary string denote the number of occurrences of 1. Let $\mathbf{N}(n, d)$ denote the set of necklaces of length n and density d. Let $\mathbf{P}(n, d)$ denote the set of pseudo-necklaces of length n and density d. The following results are proved in [12, Section 3.4].

1. $|\mathbf{P}(n, d)| \leq c|\mathbf{N}(n, d)|$, for some constant c.

2. The number of comparisons required by TESTNECKLACE over all pseudo-necklaces in $\mathbf{P}(n, d)$ is bounded above by $c|\mathbf{N}(n, d)|$, for some constant c.

3. The number of comparisons required by SUF over all pseudo-necklaces in $\mathbf{P}(n, d)$ is bounded above by $c|\mathbf{N}(n, d)|$, for some constant c.

Taking these results and summing over all $0 \leq d \leq n$, we obtain the following:

(a) $|\mathbf{P}(n)| \leq c|\mathbf{N}(n)|$, for some constant c.

(b) The amount of computation required by TESTNECKLACE over all pseudo-necklaces in $\mathbf{P}(n)$ is bounded above by $c|\mathbf{N}(n)|$, for some constant c.

(c) The amount of computation required by SUF over all pseudo-necklaces in $\mathbf{P}(n)$ is bounded above by $c|\mathbf{N}(n)|$, for some constant c.

We now apply these new results to analyze the algorithms to generate necklaces in colex order or BRGC order. As standard for generation algorithms, the time required to output a string is not part of the analysis. Observe that each call to GEN requires only a constant amount of computation, not including any calls to SUF or TESTNECKLACE. For the moment, ignore the work required by these two functions. Also, note that each call to GEN will either spawn two recursive calls or visit a pseudo-necklace of length n. This

means every node in the recursive computation tree either has two children, or is a leaf corresponding to a pseudo-necklace. Thus, the total work done for the recursive algorithm, not counting the calls made to SUF and TESTNECKLACE, is proportional to $|\mathbf{P}(n)|$ which by (a) is bounded by a constant times $|\mathbf{N}(n)|$. Now consider the work done by all calls to TESTNECKLACE. Since this function is called exactly once for each pseudo-necklace of length n, from (b) this work will also be bounded by some constant times $|\mathbf{N}(n)|$. Finally, consider all calls made to SUF. Observe that each call to SUF is applied to a unique binary string β of length up to n which corresponds to a node in the recursive computation tree. From our earlier discussion, the number of such calls is bounded by a constant times $|\mathbf{N}(n)|$. If β is not a pseudo-necklace, then the work done by its call to SUF is constant (one iteration of the **for** loop). Otherwise from (c), the total amount of work done by all pseudo-necklaces of length up to n is bounded by some constant times $\sum_{j=1}^{n} |\mathbf{N}(j)|$. This expression, in turn, is bounded by a constant times $|\mathbf{N}(n)|$ [9]. Finally, since $|\mathbf{N}(n)| < c|\mathbf{L}(n)|$ for $n > 1$ and a constant c [10], we obtain the following result.

Theorem 3.3 *Necklaces and Lyndon words of length n can be generated in $O(1)$-amortized time and $O(n)$ space in either colex order or BRGC order.*

4 Application: De Bruijn sequence construction

Let $\mathcal{DB}(n)$ denote the "Grandmama" de Bruijn sequence (defined in [4]) that can be constructed by concatenating the longest aperiodic prefixes of the length n necklaces listed in colex order. For example, by considering the colex listing of necklaces for $n = 6$ given in Section 1 we obtain

$$\mathcal{DB}(6) = 000000100100010101001101000011001011011000111010111001111011111 1.$$

To efficiently generate $\mathcal{DB}(n)$, we can use our algorithm to generate necklaces in colex order and directly apply the value returned from TESTNECKLACE to determine the longest aperiodic prefix of a given necklace. C code to generate $\mathcal{DB}(n)$ is given in the Appendix.

Corollary 4.1 *The de Bruijn sequence $\mathcal{DB}(n)$ can be generated in $O(1)$-amortized time per bit and $O(n)$-space.*

5 Proof of Theorem 2.4

Given a binary string α, let $w(\alpha)$ denote the number of 1s in α. The following result for BRGC order is proved by Vajnovzski in [14]:

Lemma 5.1 *Let $\alpha = a_1 a_2 \cdots a_n$ and $\beta = b_1 b_2 \cdots b_n$ be binary strings such that $\alpha \neq \beta$. Let r be the rightmost position in which α and β differ. Then α comes before β (denoted by $\alpha \prec \beta$) in BRGC order if and only if $w(a_r a_{r+1} \cdots a_n)$ is even.*

Let $flip(\alpha, i)$ be the string obtained by complementing the i-th bit in α, and let $flip(\alpha, i, j)$ be the string obtained by complementing the i-th and j-th bits in α. The following remark follows from the definition of pseudo-necklace.

Remark 5.2 *If $\alpha = a_1 a_2 \cdots a_n$ is a pseudo-necklace, then*
- ▷ *$flip(\alpha, i)$ is a pseudo-necklace if a_i is the leftmost 1 in α,*
- ▷ *$flip(\alpha, i, j)$ is a pseudo-necklace if a_i is the leftmost 1 in α and $a_j = 0$ with $i < j$.*

We now prove Theorem 2.4.

Proof. Let $\alpha = a_1 a_2 \cdots a_n$ be a pseudo-necklace that is not last in BRGC order. Let $\beta = b_1 b_2 \cdots b_n$ be the pseudo-necklace that appears immediately after α BRGC order. Suppose that α and β differ in at least three positions. Let i be the leftmost position such that $a_i = 1$ or $b_i = 1$, and let r be the rightmost position where they differ. Thus $i + 1 < r$. We consider two cases based on possible values for a_i:

> **Case 1**: $a_i = 1$. If $w(\alpha)$ is even, let $\gamma = flip(\alpha, i)$; otherwise, let $\gamma = flip(\alpha, i, i+1)$. Thus by Remark 5.2, γ is a pseudo-necklace (note that if $a_{i+1} = 1$ then $flip(\alpha, i, i+1) = \alpha$). Let ℓ be the rightmost index such that α and $\gamma = q_1 q_2 \cdots q_n$ differ. By the definition of γ, $w(a_\ell a_{\ell+1} \cdots a_n)$ is even, and thus $\alpha \prec \gamma$ by Lemma 5.1. Also observe that $\ell < r$ and that $q_r q_{r+1} \cdots q_n = a_r a_{r+1} \cdots a_n$. Thus, r will also be the rightmost bit such that β and γ differ. By Lemma 5.1, since $\alpha \prec \beta$, $w(a_r a_{r+1} \cdots a_n)$ is even, and thus also by Lemma 5.1, $\gamma \prec \beta$. Thus $\alpha \prec \gamma \prec \beta$, a contradiction.
>
> **Case 2**: $a_i = 0$. If $w(\beta)$ is odd, let $\gamma = flip(\beta, i)$; otherwise, let $\gamma = flip(\beta, i, i+1)$. By the definition of i, b_i is the leftmost 1 in β. Therefore by Remark 5.2, γ is a pseudo-necklace. Let ℓ be the rightmost index such that β and $\gamma = q_1 q_2 \cdots q_n$ differ. By the definition of γ, $w(q_\ell q_{\ell+1} \cdots q_n)$ is even, and thus $\gamma \prec \beta$ by Lemma 5.1. Also observe that $\ell < r$ and that $q_r q_{r+1} \cdots q_n = b_r b_{r+1} \cdots b_n$. Thus, r will also be the rightmost bit such that α and γ differ. By Lemma 5.1, since $\alpha \prec \beta$, $w(a_r a_{r+1} \cdots a_n)$ is even, and thus also by Lemma 5.1, $\alpha \prec \gamma$. Thus $\alpha \prec \gamma \prec \beta$, a contradiction.

Since each case results in a contradiction, α and β differ in at most two positions and hence the BRGC ordering induces a 2-Gray code on $\mathbf{P}(n)$. Finally, to show the cyclic property, note that the first and last strings in BRGC order are 0^n and $0^{n-1}1$ respectively [8]. Since these strings are also pseudo-necklaces, they are also the first and last pseudo-necklaces in BRGC order and they differ in one bit position. □

6 Acknowledgement

The authors would like to commend one of the reviewers who provided valuable feedback on the paper. The research of Joe Sawada is supported by the *Natural Sciences and Engineering Research Council of Canada* (NSERC) grant RGPIN 400673-2012. The research of Dennis Wong is supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the "ICT Consilience Creative Program" (IITP-2015-R0346-15-1007) supervised by the IITP (Institute for Information & communications Technology Promotion).

References

[1] K. S. Booth. Lexicographically least circular substrings. *Inform. Process. Lett.*, 10(4/5):240–242, 1980.

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

[3] C. Degni and A. Drisko. Gray-ordered binary necklaces. *Electron. J. Combin.*, 14(1):23 pages, 2007.

[4] P. B. Dragon, O. I. Hernandez, and A. Williams. *LATIN 2016: Theoretical Informatics: 12th Latin American Symposium, Ensenada, Mexico, April 11-15, 2016, Proceedings*, chapter The Grandmama de Bruijn Sequence for Binary Strings, pages 347–361. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[5] H. Fredricksen and I. J. Kessler. An algorithm for generating necklaces of beads in two colors. *Discrete Math.*, 61(2):181 – 188, 1986.

[6] H. Fredricksen and J. Maiorana. Necklaces of beads in k colors and k-ary de Bruijn sequences. *Discrete Math.*, 23:207–210, 1978.

[7] F. Gray. Pulse code communication. *U.S. Patent 2,632,058*, 1953.

[8] F. Ruskey. *Combinatorial Generation*. Working version (1i) edition, 1996.

[9] F. Ruskey, C. Savage, and T. M. Y. Wang. Generating necklaces. *J. Algorithms*, 13:414–430, 1992.

[10] F. Ruskey and J. Sawada. An efficient algorithm for generating necklaces with fixed density. *SIAM J. Comput.*, 29(2):671–684, 1999.

[11] A. Sabri and V. Vajnovszki. Two Reflected Gray Code based orders on some restricted growth sequences. *ArXiv e-prints*, June 2013.

[12] J. Sawada and A. Williams. A Gray code for fixed-density necklaces and Lyndon words in constant amortized time. *Theor. Comput. Sci.*, 502:46–54, 2013.

[13] V. Vajnovszki. Gray code order for Lyndon words. *Discrete Math. Theor. Comput. Sci.*, 9(2):145–151, 2007.

[14] V. Vajnovszki. More restrictive Gray codes for necklaces and Lyndon words. *Inform. Process. Lett.*, 106(3):96–99, 2008.

[15] T. M. Wang and C. D. Savage. A Gray code for necklaces of fixed density. *SIAM J. Discrete Math*, 9:654–673, 1997.

[16] M. Weston and V. Vajnovszki. Gray codes for necklaces and Lyndon words of arbitrary base. *Pure Math. Appl. (PU.M.A.)*, 17(1-2):175–182, 2006.

Appendix - C code

```
//-----------------------------------------------------------------------------
// NECKLACES, LYNDON WORDS, PSEUDO-NECKLACES in COLEX or BRGC ORDER IN O(1)-AMORTIZED TIME
//-----------------------------------------------------------------------------
#include <stdio.h>
#define MAX 100

int type,N,total=0;
int a[MAX], S[MAX], T[MAX], NECK=0, LYN=0, DB=0, BRGC=0;

void Gen(int j, int b, int r, int rev);

//-----------------------------------------------------------------------------
// RETURNS 0 IF NOT A NECKLACE, OTHERWISE RETURNS THE LENGTH OF THE LONGEST LYNDON PREFIX
//-----------------------------------------------------------------------------
int TestNecklace(int b, int r) {
    int i, p=0;

    if (b == 1) return 1;
    for (i=0; i<b; i++) {
        if (r-i <= 0) r += b;
        if (r < b || b == 1) p += S[r-i] + T[r-i];
        if (S[b-i] < S[r-i] || (S[b-i] == S[r-i] && T[b-i] > T[r-i])) return 0;
        if (S[b-i] > S[r-i] || (S[b-i] == S[r-i] && T[b-i] < T[r-i])) return N;
    }
    return(p);
}
//-----------------------------------------------------------------
// RETURNS b IF CURRENT SUFFIX STARTING AT BLOCK b IS SMALLER THAN
// THE ONE STARTING FROM BLOCK r;  OTHERWISE RETURNS r
//-----------------------------------------------------------------
int Suf(int b, int r) {

    for (int i=0; i<r; i++) {
        if (b-i == r) return b;
        if (S[b-i] < S[r-i] || (S[b-i] == S[r-i] && T[b-i] > T[r-i])) return r;
        if (S[b-i] > S[r-i] || (S[b-i] == S[r-i] && T[b-i] < T[r-i])) return b;
    }
    return r;
}
//-------------------------------------------------
// VISIT EACH PSEUDO-NECKLACE
//-------------------------------------------------
void Visit(int b, int r) {
    int i,p=N;

    // TEST IF PSEUDO-NECKLACE IS A NECKLACE/LYNDON WORD
    if (S[b] == S[r] && T[b] == T[r] && b > r) {
        p = TestNecklace(b,r);
        if (NECK && p == 0) return;
        if (LYN  && p != N) return;
    }
    if (LYN && (a[1] == 1 || a[N] == 0)) return;

    // OUTPUT THE STRING AND BLOCKS
    if (DB) {
        if (a[1] == 1 || a[N] == 0) p = 1;
        for (i=1; i<=p; i++) printf("%d", a[i]);
    }
    else {
        for (i=1; i<=N; i++) printf("%d", a[i]);
        printf("   ");
        for (i=b; i>0; i--) printf(" (%d %d)", S[i], T[i]);
        printf("\n");
        total++;
    }
}
```

```
//-----------------------------------------------------------------------
// RETURN TRUE IF 1a[j+1..n] IS THE SUFFIX OF SOME LENGTH N PSEUDO-NECKLACE
//-----------------------------------------------------------------------
int Extend(int j, int b, int r) {

    if (j < N && a[N] == 0) return 0;    // Case 0^n
    if (j > 0 && S[r] == 0) return 1;    // Case 1^n
    if (S[b] == 0 && (j-1 > S[r] || (j-1 == S[r] && T[b]+1 <= T[r]))) return 1;
    if (S[b] > 0  && (j-1 > S[r] || (j-1 == S[r] && 1 <= T[r]))) return 1;
    return 0;
}
//-------------------------------------------------
void AddZero(int j, int b, int rev, int r) {

    S[b] = S[b]+1;
    Gen(j-1, b, rev, Suf(b,r));
    S[b] = S[b]-1;
}//-------------------------------------------------
void AddOne(int j, int b, int rev, int r) {

    a[j] = 1;
    if (S[b] == 0) {
        T[b] = T[b]+1;
        Gen(j-1,b,rev,r);
        T[b] = T[b]-1;
    }
    else {
        S[b+1] = 0;  T[b+1] = 1;
        Gen(j-1,b+1,rev,r);
    }
    a[j] = 0;
}
//-----------------------------------------------------------------------------------------
// LIST NECKLACES (LYNDON WORDS/PSEUDO-NECKLACES) IN COLEX OR BRGC ORDER.
// CURRENT SUFFIX IS a[j+1..n] WITH BLOCK REPRESENTATION B[b..1] = (S[b],T[b]).. (S[1],T[1])
// THE SMALLEST SUFFIX STARTS AT BLOCK r.  rev INDICATES WHETHER TO REVERSE LISTING.
//-----------------------------------------------------------------------------------------
void Gen(int j, int b, int rev, int r) {

    if (!Extend(j, b, r)) {
        S[b] = S[b] + j;
        Visit(b,r);
        S[b] = S[b] - j;
    }
    else if (BRGC && rev) {
        AddOne(j,b,0,r);
        AddZero(j,b,1,r);
    }
    else {
        AddZero(j,b,0,r);
        AddOne(j,b,1,r);
    }
}
//-------------------------------------------------
void Input() {

    printf("\n  Colex Order                BRGC Order\n");
    printf("  ----------------------    -------------------\n");
    printf("  1.  Necklaces             5. Necklaces \n");
    printf("  2.  Lyndon words          6. Lyndon words\n");
    printf("  3.  Pseudo-necklaces      7. Pseudo-necklaces\n");
    printf("  4.  De Bruijn sequence  \n");

    printf("\nENTER option: ");    scanf("%d", &type);

    if (type == 1 || type == 5) NECK = 1;
    if (type == 2 || type == 6) LYN = 1;
    if (type == 4) DB = NECK =  1;
```

```
    if (type >=5) BRGC = 1;

    printf("ENTER length n: ");    scanf("%d", &N);
    printf("\n");
}
//-------------------------------------------------
int main() {

    Input();
    S[1] = 0;  T[1] = 0;
    Gen(N,1,0,1);
    if (!DB) printf("Total = %d\n", total);
    printf("\n");
}
```