An Efficient Algorithm for Generating Necklaces with Fixed Density

Joe Sawada[*] Frank Ruskey[†]

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

Abstract

A k-ary necklace is an equivalence class of k-ary strings under rotation. A necklace of fixed density is a necklace where the number of zeroes is fixed. We present a fast, simple, recursive algorithm for generating (i.e., listing) fixed density k-ary necklaces or aperiodic necklaces. The algorithm is optimal in the sense that it runs in time proportional to the number of necklaces produced.

1 Introduction

There are many reasons to develop algorithms for producing lists of basic combinatorial objects. First, the algorithms are truely useful and find many applications in diverse areas such as hardware and software testing, non-parametric statistics, and combinatorial chemistry. Secondly, the development of these algorithms can lead to mathematical discoveries about the objects themselves, either experimentally, or through insights gained in the development of the algorithms.

The primary performance goal in an algorithm for listing a combinatorial family is an algorithm whose running time is proportional to the number of objects produced. The amount of output required to print the objects is not the correct measure: the correct measure is the amount of data structure change that the objects undergo, since typical applications only need to process that part of successive object that changes. In this paper an *efficient* algorithm is one that uses only a constant amount of computation per object, in an amortized sense. Such algorithms are also said to be CAT, for Constant Amortized Time.

Necklaces are a fundamental type of combinatorial object, arising naturally, for example, in the construction of single-track Gray codes, in the enumeration of irreducible polynomials over finite fields, and in the theory of free Lie algebras. Efficient algorithms for exhaustively generating necklaces were first developed by Fredricksen and Kessler [4] and Fredricksen and Maiorana [5], although they did not prove that they were efficient. They were proven to be efficient by Ruskey, Savage, and Wang [7]. Closely related algorithms for generating Lyndon words (aperiodic necklaces) were developed by Duval [3] and shown to be efficient by Berstel and Pocchiola [1]. Subsequently, a recursive algorithm was developed that was more flexible and easier to analyze than the earlier algorithms, which were all iterative [2]. In many applications not all necklaces are required, but rather only those of fixed density (the number of zeroes is fixed). Previous to this paper, no efficient generation algorithm for fixed density necklaces was known.

[*]Research supported by NSERC. e-mail: `jsawada@csr.uvic.ca`
[†]Research supported by NSERC. e-mail: `fruskey@csr.uvic.ca`

Previous fixed density necklace algorithms had running times of $O(n \cdot N(n,d))$ (Wang and Savage [8]) and $O(N(n))$ (Fredricksen and Kessler [4]), 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, but our results apply over a general alphabet. We take a new approach by first modifying Ruskey's recursive algorithm for generating necklaces [2] and then optimizing it for the fixed density case. Recursive algorithms have several advantages over their iterative counterparts. They are generally simpler and easier to analyze. They are are more suitable to conversion to a backtracking algorithms, since subtrees are easily pruned from the computation tree. In fact, we have used just such a backtracking to discover new minimal difference covers (sets of numbers achieving all possible differences, mod n).

In the following section we will give some definitions related to necklaces and then in Section 3 we will introduce a fast algorithm for generating fixed density k-ary necklaces. In Section 4 we analyze the algorithm, proving the algorithm is CAT for any density.

2 Background and Definitions

A k-ary *necklace* is an equivalence class of k-ary strings under rotation. We identify each necklace with the lexicographically least representative in its equivalence class. The set of all k-ary necklaces with length n is denoted $\mathbf{N}_k(n)$. For example $\mathbf{N}_2(4) = \{0000,0001,0011,0101,0111,1111\}$. The cardinality of $\mathbf{N}_k(n)$ is denoted $N_k(n)$.

An important class of necklaces are those that are aperiodic. An aperiodic necklace is called a *Lyndon word*. Let $\mathbf{L}_k(n)$ denote the set of all k-ary Lyndon words with length n. For example $\mathbf{L}_2(4) = \{0001,0011,0111\}$. The cardinality of $\mathbf{L}_k(n)$ is denoted $L_k(n)$.

A string α is a *pre-necklace* if it is a prefix of some necklace. The set of all k-ary pre-necklaces with length n is denoted $\mathbf{P}_k(n)$. For example $\mathbf{P}_2(4) = \mathbf{N}_2(4) \cup \{0010,0110\}$. The cardinality of $\mathbf{P}_k(n)$ is $P_k(n)$.

We denote fixed density necklaces, Lyndon words and pre-necklaces in a similar manner by adding the additional parameter d to represent the number of non-zero characters in the strings. We refer to the number d as the density of the string. Thus the set of k-ary necklaces with density d 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, we introduce the set $\mathbf{P}'_k(n,d)$ which contains the elements of $\mathbf{P}_k(n,d)$ whose last character is non-zero. Its cardinality is denoted $P'_k(n,d)$.

To count fixed density necklaces we let $N(n_0, n_1, \cdots n_{k-1})$ denote the number of necklaces composed of n_i occurrences of the symbol i, for $i = 0, 1, \ldots, k$-1. Let the density of the necklace $d = n_1 + \cdots + n_{k-1}$ and $n_0 = n - d$. It is known from Gilbert and Riordan [6] that

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

To get the number of fixed density necklaces with length n and density d, we sum over all possible

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

Figure 1: The recursive necklace algorithm.

values of $n_1, n_2, \ldots, n_{k-1}$

$$N_k(n, d) = \sum_{n_1+\cdots+n_{k-1}=d} N(n_0, n_1, \ldots, n_{k-1})$$

The number of fixed density Lyndon words are counted by a similar formula.

3 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 $\mathsf{Gen}(t, p)$ (Figure 1) [2]. We then optimize this new necklace algorithm for the fixed density case by making a few key observations about fixed density necklaces.

To begin we give a brief overview of $\mathsf{Gen}(t, p)$. 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 $\mathsf{Gen}(1,1)$ 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 $\mathsf{Gen}(t, p)$, 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 $\mathsf{Gen}(t, p)$ makes one recursive call for each valid value for the next character in the string, 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 $\mathsf{PrintIt}(p)$ allow us to differentiate between these various objects as shown in Table 1.

The computation tree for $\mathsf{Gen}(t, p)$ 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. By comparing the number of nodes in the computation tree to the number of objects generated it was shown that this algorithm is CAT [2].

3.1 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

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

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

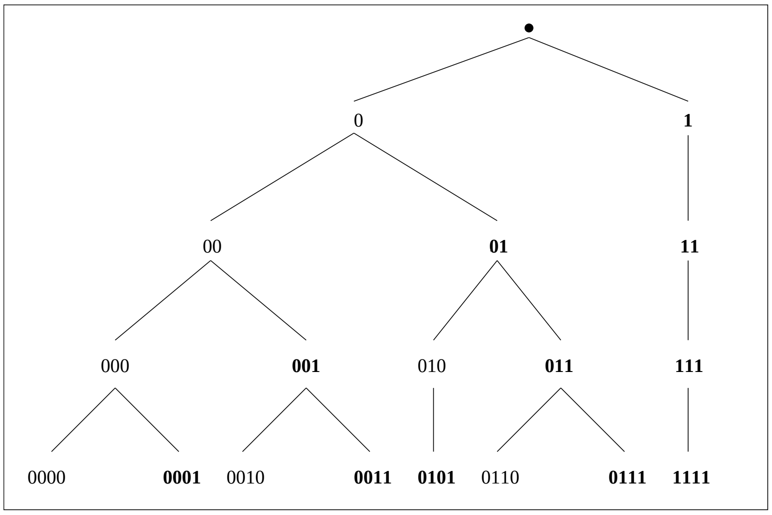

Figure 2: Computation Tree for $N_2(4)$ from $\mathsf{Gen}(t,p)$

the computation tree by compressing all of the pre-necklaces whose last character is 0. Looking at Figure 2, we want to generate only the nodes in bold. This results in the modified computation tree shown in Figure 3. 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 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 ith element of the array a represents the position of the ith non-zero character, and the ith element of the array b represents the value of the ith 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. (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

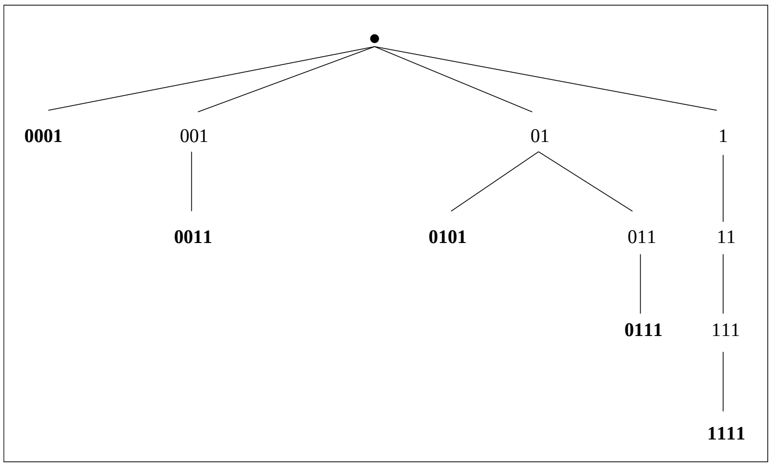

Figure 3: Computation Tree for $N_2(4)$ from $\mathsf{Gen2}(t,p)$

so that the new string still has the pre-necklace property. We compute this maximal position for the next character using the following expression

$$\lfloor (t+1)/p \rfloor a_p + a_{(t+1) \bmod p}.$$

The minimal value for this position is b_p if $(t+1) \bmod p = 0$ and $b_{(t+1) \bmod p}$ otherwise. By the properties of pre-necklaces all larger values at the maximal position are also valid [7]. 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 $\mathsf{PrintIt}(p)$ function to print out either the Lyndon words or necklaces in a similar manner to the original algorithm $\mathsf{Gen}(t,p)$.

This modified algorithm, $\mathsf{Gen2}(t,p)$, for generating necklaces is given in Figure 4. Each initial branch of the computation tree is a result of a separate call to $\mathsf{Gen2}(t,p)$, 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 Gen2 and must be generated separately. The nodes of the resulting computation tree for $\mathsf{Gen2}(t,p)$ 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.

3.2 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 ith 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

```
procedure Gen2 ( t, p : integer );
local i, j, max : integer;
begin
    if a_t ≥ n then PrintIt( p )
    else begin
        max = (t + 1)/p * a_p + a_{(t+1) mod p};
        if max ≤ n then begin
            a_{t+1} := max;
            if (t + 1) mod p = 0 then b_{t+1} := b_p;
            else b_{t+1} := b_{(t+1) mod p};
            Gen2 ( t + 1, p );
        end else begin
            max := n;   a_{t+1} := n;   b_{t+1} := 1;
            Gen2 ( t + 1, t + 1 );
        end;
        for i ∈ {b_{t+1} + 1, ..., k − 2, k − 1} do begin
            b_{t+1} := i;
            Gen2 ( 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;
                Gen2 ( t + 1, t + 1 );
    end; end; end;
end {of Gen2};
```

Figure 4: Modified recursive necklace algorithm

any resulting string with length n will have density less than d. Also, because the last non-zero character must be in the nth 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 nth 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 nth 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 nth position will result in a string that is a Lyndon word [7]. 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 with the following optimizations:

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

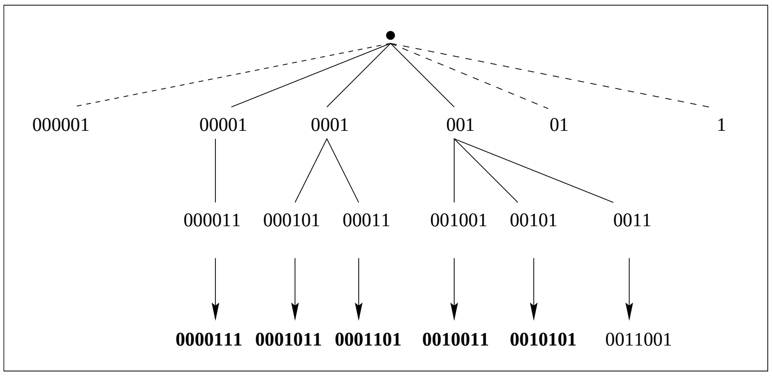

Figure 5: Computation Tree (solid edges only) for $N_2(7,3)$ from $\mathsf{GenFix}(t,p)$

2. The ith 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 nth position in $\mathsf{PrintIt}(p)$ function.

The computation tree for generating $N_2(7,3)$ is given in Figure 5. The dotted lines indicate the initial branches we do not need to follow by modification 1. The arrows indicate the strings produced by adding the final character to the nth position. The bold strings indicate the actual necklaces produced by the $\mathsf{PrintIt}(p)$ 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 6. In the binary case we make use of the fact that we can generate binary necklaces with density $d > n/2$ 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.

4 Analysis of Algorithm

In this section we show that $\mathsf{GenFix}(t,p)$ 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 $k-1$ 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

```
procedure GenFix ( 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 := ((t + 1)/p) ∗ a_p + a_{(t+1) mod p};
        if max ≤ tail then begin
            a_{t+1} := max;
            if (t + 1) mod p = 0 then b_{t+1} := b_p;
            else b_{t+1} =: b_{(t+1) mod p};
            GenFix( t + 1, p );
            for i ∈ {b_{t+1} + 1, . . . , k − 2, k − 1} do begin
                b_{t+1} := i;
                GenFix( 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;
                GenFix( t + 1, t + 1 );
    end; end; end;
end {of GenFix};
```

Figure 6: Fixed density necklace algorithm

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

Recall that we generate binary fixed-density necklaces with density greater than $n/2$ by generating $N(n, n-d)$ 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 1 $P'_k(n, d) \leq N_k(n, d) + L_k(n, d)$

PROOF: Partition $\mathbf{P}'_k(n, d)$ into two classes: necklaces and non-necklaces. We show the existence of an injective mapping (proof omitted) from the non-necklaces to $L_k(n, d)$. □

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 2 *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 3 *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 (1) and noting that the remaining terms are all non-negative we have

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

There exists an injective mapping from the periodic necklaces to Lyndon words of the same length and density (proof omitted), implying the upper bound $N_k(n,d) \leq 2L_k(n,d)$. □

We now use the previous lemmas and some basic binomial coefficient identities to get a simple upper bound on the size of the computation tree:

$$\begin{aligned}
CompTree_k(n,d) &\leq 3\sum_{i=1}^{d-1}\sum_{j=1}^{n-d+i} L_k(j,i) \\
&\leq 3\sum_{i=1}^{d-1}\sum_{j=1}^{n-d+i} \frac{1}{j}\binom{j}{i}(k-1)^i \\
&= 3\sum_{i=1}^{d-2}\frac{1}{i}\binom{n-d+i}{i}(k-1)^i + \frac{3}{d-1}\binom{n-1}{d-1}(k-1)^{d-1} \qquad (2)
\end{aligned}$$

$$< \frac{6}{d-1}\binom{n-1}{d-1}(k-1)^{d-1} + \frac{3}{d-1}\binom{n-1}{d-1}(k-1)^{d-1} \quad (3)$$

$$= \frac{9}{d-1}\binom{n-1}{d-1}(k-1)^{d-1}$$

The simplification between equations (2) and (3) is a result of the following lemma. We omit the proof by induction on d.

LEMMA 4 *For either* $\{1 < d < n$ *and* $k > 2\}$ *or* $\{1 < d \leq n/2$ *and* $k = 2\}$ *the following inequality is valid:*

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

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 3 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 1 *Algorithm GenFix for generating fixed density k-ary necklaces is CAT.*

References

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

[2] K. Cattell, F. Ruskey, J. Sawada, C.R. Miers, M. Serra, Generating Unlabeled Necklaces and Irreducible Polynomials over GF(2), manuscript, 1998.

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

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

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

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

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

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