SIAM J. COMPUT.
Vol. 29, No. 2, pp. 671–684

AN EFFICIENT ALGORITHM FOR GENERATING NECKLACES WITH FIXED DENSITY*

FRANK RUSKEY† AND JOE SAWADA†

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 zeros 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.

Key words. necklaces, Lyndon words, fixed density, CAT algorithm, generate, difference covers

AMS subject classifications. 05-04, 68R05, 68R15

PII. S0097539798344112

1. Introduction.

There are many reasons to develop algorithms for producing lists of basic combinatorial objects. First, the algorithms are truly useful and find many applications in diverse areas such as hardware and software testing, nonparametric statistics, and combinatorial chemistry. Second, 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. 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 constant amortized time (CAT) algorithms.

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 [8]. 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 zeros is fixed). Previous to this paper, no efficient generation algorithm for fixed-density necklaces was known.

Previous fixed-density necklace algorithms had running times of $O(n \cdot N(n,d))$ (Wang and Savage [9]) 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

*Received by the editors August 31, 1998; accepted for publication (in revised form) December 23, 1998; published electronically November 23, 1999. This research was supported by the NSERC.
http://www.siam.org/journals/sicomp/29-2/34411.html

† Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada (fruskey@csr.uvic.ca, jsawada@csr.uvic.ca).

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 more suitable to conversion to 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. 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. In section 5 we conclude by outlining an application and some future work.

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 *prenecklace* if it is a prefix of some necklace. The set of all k-ary prenecklaces 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 prenecklaces in a similar manner by adding the additional parameter d to represent the number of nonzero 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 prenecklaces is denoted by $\mathbf{P}_k(n, d)$ and has cardinality $P_k(n, d)$. In addition to these familiar terms we introduce the set $\mathbf{P}'_k(n, d)$, which is the elements of $\mathbf{P}_k(n, d)$ whose last character is nonzero. Its cardinality is denoted $P'_k(n, d)$.

To count fixed-density necklaces we let $N(n_0, n_1, \ldots, n_{k-1})$ 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

$$N(n_0, n_1, \ldots, n_{k-1}) = \frac{1}{n} \sum_{j \backslash gcd(n_0, \ldots, n_{k-1})} \phi(j) \frac{(n/j)!}{(n_0/j)! \cdots (n_{k-1}/j)!}. \tag{2.1}$$

To get the number of fixed-density necklaces with length n and density d, we sum over all possible values of $n_1, n_2, \ldots, n_{k-1}$:

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

The number of fixed-density Lyndon words is defined similarly:

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

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

In the binary case these expressions simplify as follows:

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

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

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

In the following section we will introduce a CAT algorithm to generate fixed-density necklaces. When analyzing the performance of our algorithm we make use of the following lemmas about prenecklaces and Lyndon words.

Cattell et al. [2] give a lemma that characterizes prenecklaces by making use of a function *lyn* on strings, which is the length of the longest Lyndon prefix of the string:

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

LEMMA 2.1. *Let k-ary string* $\alpha = a_1 \cdots a_n$ *and* $p = lyn(\alpha)$. *Then* $\alpha \in \mathbf{P}_k(n)$ *if and only if* $a_{j-p} = a_j$ *for* $j = p + 1, \ldots, n$.

Reutenauer [7] gives a useful lemma about Lyndon words. Inequalities between words are always with respect to lexicographic order.

LEMMA 2.2. *If* α *and* β *are Lyndon words with* $\alpha < \beta$, *then* $\alpha\beta$ *is a Lyndon word.*

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 3.1) [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 prenecklaces. The prenecklace being generated is stored in the array a with 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 prenecklace to get a new prenecklace. At the beginning of each recursive call to $\mathsf{Gen}(t, p)$, the length of the prenecklace being generated is $t - 1$ and the length of the longest Lyndon prefix is p. As long as the length of the current prenecklace 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 prenecklaces of length n in lexicographic order by specifying which object we want to generate. The function $\mathsf{PrintIt}(p)$ allows us to differentiate between these various objects as shown in Figure 3.2.

The computation tree for $\mathsf{Gen}(t, p)$ consists of all prenecklaces of length less than or equal to n. As an example, we show a computation tree for $N_2(4)$ in Figure 3.3. 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].

```
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};
```

FIG. 3.1. *The recursive necklace algorithm.*

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

FIG. 3.2. *Different objects output by different versions of* PrintIt(p).

3.1. Modified necklace algorithm. For every necklace of positive density, the last character of the string must be nonzero. Thus, if we are concerned only with generating necklaces or Lyndon words we can reduce the size of the computation tree by compressing all the prenecklaces whose last character is 0. Looking at Figure 3.3, we want to generate only the nodes in bold. This results in the modified computation tree shown in Figure 3.4. Notice that at each successive level in this tree we are incrementing the density of the prenecklace rather than the length. To generate this modified tree we create a recursive routine based on the original necklace algorithm in Figure 3.1; 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 nonzero character.

To make this change we use the array a to hold the positions of the nonzero characters and maintain another array b to indicate the values of the nonzero characters. The ith element of the array a represents the position of the ith nonzero character, and the ith element of the array b represents the value of the ith nonzero 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 nonzero 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 nonzero 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 nonzero character.

To determine the valid positions and values for the next nonzero character and to maintain the lexicographic ordering we compute the maximum position and the minimum value for that position so that the new string still has the prenecklace property. We compute this maximal position for the next character using the following

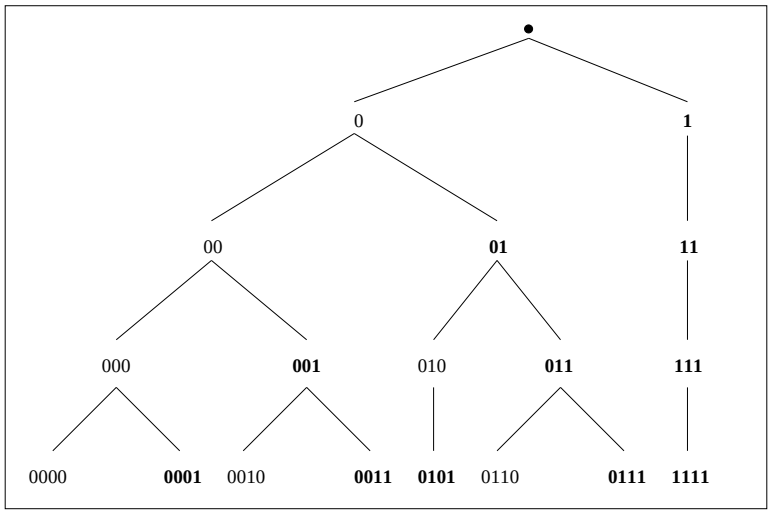

FIG. 3.3. *Computation tree for* $N_2(4)$ *from* $\mathsf{Gen}(t, p)$.

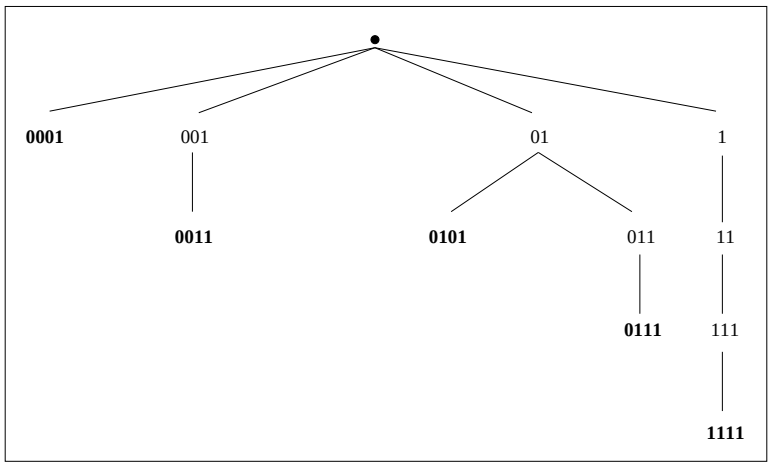

FIG. 3.4. *Computation tree for* $N_2(4)$ *from* $\mathsf{Gen2}(t, p)$.

expression:

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

The minimal value for this position is b_{t+1-p}. By the properties of prenecklaces all larger values at the maximal position are also valid [8]. Also, all positions before the maximum position and greater than the position of the last assigned nonzero 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 nonzero 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 $\mathsf{Gen}(t, p)$.

```
procedure Gen2 ( t, p : integer );
local i, j, max : integer;
begin
    if a_t ≥ n then PrintIt( p )
    else begin
        max = a_{t+1-p} + a_p;
        if max ≤ n then begin
            a_{t+1} := max;
            b_{t+1} := b_{t+1-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};
```

FIG. 3.5. *Modified recursive necklace algorithm.*

This modified algorithm, Gen2(t, p), for generating necklaces is given in Figure 3.5. Each initial branch of the computation tree is a result of a separate call to Gen2(t, p), each call specifying a different combination for the position and value of the first nonzero character. Note that the zero string is not generated by Gen2(t,p) and must be generated separately. The nodes of the resulting computation tree for Gen2(t, p) are all prenecklaces with length less than or equal to n whose last character is nonzero.

A complete C program for this modified necklace algorithm is available from the authors. A simplified program for the binary case is also available. Observe that we are not restricted to generating the necklaces in lexicographic order. Many orders are possible by reordering 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 nonzero character depending on the density. In particular, there are no necklaces with density d that can have the first nonzero 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 nonzero character, then the $(i+1)$st nonzero character must come before the position $n - (d - i) + 2$. If we place the next character at or after this position, then any resulting string with length n will have density less than d. Also, because the last nonzero character must be in the nth position, we stop the string generation after placing the $(d-1)$st nonzero character.

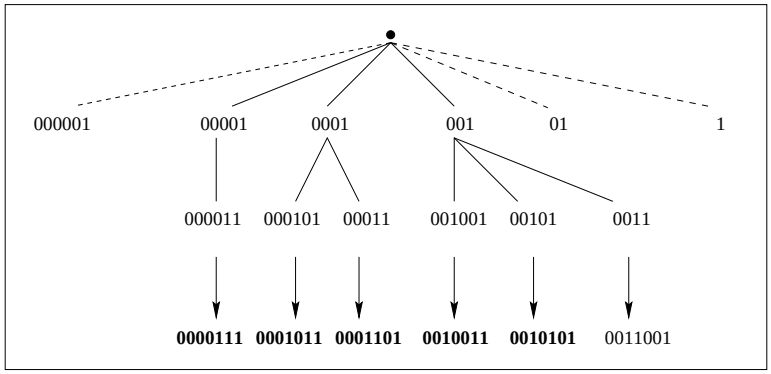

FIG. 3.6. *Computation tree (solid edges only) for* $N_2(7,3)$ *from* GenFix(t,p).

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 nonzero character as outlined in the previous subsection. 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 [8]. 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 3.5 with the following optimizations:

1. The first nonzero character must be between $n-d+1$ and $(n-1)/d+1$ inclusive.
2. The ith nonzero character must be placed at or before the $(n-d+i)$th position.
3. Stop generating when we have assigned $d-1$ nonzero characters.
4. Determine valid values for the nth position in the PrintIt(p) function.

The computation tree for generating $N_2(7,3)$ is given in Figure 3.6. 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 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 3.7. To generate fixed-density prenecklaces, we generate $N(n+1,d+1)$ and print out only the first n characters, making sure we do not print the same string twice. A complete C program for this fixed-density necklace algorithm is available from the authors. A simplified program for the binary case is also available. In the latter program 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.

```
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 := a_{t+1−p} + a_p;
        if max ≤ tail then begin
            a_{t+1} := max;
            b_{t+1} := b_{t+1−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};
```

FIG. 3.7. *Fixed-density necklace algorithm.*

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 zero we simply generate the zero 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 zeros. In each case where the density is greater than zero the resulting strings are generated in CAT.

For the nontrivial 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 prenecklaces that end in a nonzero 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 prenecklaces with length n and density d, where the last bit is nonzero. Thus, the size of the computation tree for our fixed-density algorithm $(1 < d < n)$ is bounded by the expression

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

Recall that we generate binary fixed-density necklaces with density greater than $n/2$ by generating $N(n, n - d)$ 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 4.1. $P'_k(n,d) \leq N_k(n,d) + L_k(n,d)$.

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

We show that $Q'_k(n,d) \leq L_k(n,d)$ by providing an injective mapping of $\mathbf{Q}'_k(n,d)$ to $\mathbf{L}_k(n,d)$. By Lemma 2.1 each element of the set $\mathbf{Q}'_k(n,d)$ must have the form $\alpha = (a_1 \cdots a_p)^j a_1 \cdots a_m$, where $p = lyn(\alpha)$, $j \geq 1$, and $0 < m < p$. Let n_i be the number of occurrences of the symbol i in $a_1 \cdots a_m$ and define the string $\gamma = 0^{n_0} 1^{n_1} \cdots (k-1)^{n_{k-1}}$. We define a function f on the set $\mathbf{Q}'_k(n,d)$ as follows:

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

For example, $f((002101303)^7 0021013) = 0001123(002101303)^7$. This mapping preserves both length and density. Since γ and $a_1 \cdots a_p$ are both Lyndon words and $\gamma < a_1 \cdots a_p$, it follows from repeated use of Lemma 2.2 that $f(\alpha) \in \mathbf{L}_k(n,d)$.

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

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

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

COROLLARY 4.2. $P_2(n,d) \leq N_2(n+1,d+1) + L_2(n+1,d+1)$.

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

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

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

LEMMA 4.3. *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 4.4. *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 the case when $j = 1$ in (2.1) and noting that the remaining terms are all nonnegative, we have

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

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

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

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

This function simply moves the bit $a_{p(j-1)+1} = a_1 = 0$ to the front of the string. This operation preserves both length and density. Since $(a_1 \cdots a_p)^{j-1} a_2 \cdots a_p$ is a Lyndon word, by Lemma 2.2 $g(\alpha)$ is a Lyndon word.

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

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

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

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

$$\begin{aligned}
&\leq 3\sum_{i=1}^{d-1}\sum_{j=1}^{n-d+i} L_k(j,i)\\
&\leq 3\sum_{i=1}^{d-1}\sum_{j=1}^{n-d+i} \frac{1}{j}\binom{j}{i}(k-1)^i\\
&= 3\sum_{i=1}^{d-1}\frac{1}{i}(k-1)^i\sum_{j=1}^{n-d+i}\binom{j-1}{i-1}\\
&= 3\sum_{i=1}^{d-1}\frac{1}{i}\binom{n-d+i}{i}(k-1)^i. \qquad (4.1)
\end{aligned}$$

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

To simplify this bound for the computation tree even more, we inductively prove yet another upper bound for the remaining sum in (4.1). We first prove an upper bound for the case when $k > 2$ and $1 < d < n$. We then provide a similar proof for the case when $k = 2$. In the latter case we take advantage of the fact that we can generate binary necklaces with $d > n/2$ by generating necklaces with density $n-d$ and then complementing the output of each generated necklace to get all necklaces with density d. Once again, this is the only situation where the strings are not generated in lexicographic order. Thus when $k = 2$, we only consider the case when $1 < d \leq n/2$.

LEMMA 4.5. *For $2 \leq d < n$ and $k > 2$,*

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

Proof. We prove the lemma by induction on d. Let

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

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

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

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

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

$$= \frac{3d}{(d-1)(n-1)}\binom{n-1}{d}(k-1)^{d-1}$$
$$\leq \frac{2}{d}\binom{n-1}{d}(k-1)^{d}.$$

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

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

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

LEMMA 4.6. *For $2 \leq d \leq n/2$ and $k = 2$,*

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

Proof. We prove the lemma by induction on d. Let

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

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

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

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

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

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

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

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

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

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

Recall that our goal is to prove that the ratio of nodes in the computation tree to the number of strings produced is bounded by a constant. From Lemma 4.4 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 4.7. *Algorithm* GenFix *for generating fixed-density k-ary necklaces is CAT.*

5. Future work and an application. In this paper we have presented a CAT algorithm for generating fixed-density k-ary necklaces. This algorithm is used when we want to generate necklaces where the number of zeros is fixed; however, if we want all necklaces where the number of occurrences for every character is fixed, then our algorithm works only for the binary case. An efficient algorithm for the k-ary case would be very interesting, but currently does not exist. Another open problem is to count the number of fixed-density prenecklaces; the number of fixed-density necklaces and Lyndon words is known and was given in this paper.

5.1. Generating difference covers. As an application, we embed our fixed-density necklace algorithm into a program that generates difference covers. A set $D = \{a_1, \ldots, a_k\}$, $1 < a_i < n$, is called an (n,k) difference cover if for every $d \neq 0 \bmod n$ there exists an ordered pair (a_i, a_j) in D such that $a_i - a_j = d \bmod n$. For example, the set {1, 2, 3, 6} is a $(10,4)$ difference cover. An (n,k) difference cover is minimal if an $(n,k-1)$ difference cover does not exist.

To generate all difference covers (n,k) we generate all fixed-density necklaces $\mathbf{N}_2(n,k)$ where the position of each one in the necklace represents a number in the set D. To determine whether the necklace represents a difference cover, we keep track of information about each ordered pair. This additional work takes at worst case $O(k)$ time for every node in the computation tree. Thus the overall running time for generating all the (n,k) difference covers is $O(kN_2(n,k))$.

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

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

A complete C program for generating difference covers with equivalence under rotation is available from the authors.

REFERENCES

[1] J. Berstel and M. Pocchiola, *Average cost of Duval's algorithm for generating Lyndon words*, Theoret. Comput. Sci., 132 (1994), pp. 415–425.

[2] K. Cattell, F. Ruskey, J. Sawada, C. R. Miers, and M. Serra, *Fast Algorithms to Generate 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*, Theoret. Comput. Sci., 60 (1988), pp. 255–283.

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

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

[6] E. N. Gilbert and J. Riordan, *Symmetry types of periodic sequences*, Illinois J. Math., 5 (1961), pp. 657–665.

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

[8] F. Ruskey, C. D. Savage, and T. Wang, *Generating necklaces*, J. Algorithms, 13 (1992), pp. 414–430.

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