SIAM J. COMPUT.
Vol. 31, No. 1, pp. 259–268

GENERATING BRACELETS IN CONSTANT AMORTIZED TIME*

JOE SAWADA†

Abstract. A bracelet is the lexicographically smallest element in an equivalence class of strings under string rotation and reversal. We present a fast, simple, recursive algorithm for generating (i.e., listing) k-ary bracelets. Using simple bounding techniques, we prove that the algorithm is optimal in the sense that the running time is proportional to the number of bracelets produced. This is an improvement by a factor of n (where n is the length of the bracelets being generated) over the fastest, previously known algorithm to generate bracelets.

Key words. bracelet, necklace, CAT algorithm, generate, forbidden substring

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

PII. S0097539700377037

1. Introduction.

The rapid growth in the fields of combinatorial chemistry and computational biology is resulting in an increased demand for efficient algorithms which produce exhaustive lists of combinatorial objects [1]. Dan Gusfield (see [9, p. xv]) claims that "significant contributions to computational biology might be made by extending or adapting [string] algorithms from computer science, even when the original algorithm has no clear utility in biology." In particular, correspondences between DNA sequences and restricted classes of circular strings are described in [3].

Within the mathematical sciences, researchers are constantly trying to find patterns hidden in the structure of combinatorial objects. The growing trend of using computers and algorithms to produce lists of such objects is allowing researchers to obtain more information about the objects themselves. Often, this will lead to a more thorough understanding of an object which may lead to new and interesting discoveries. In some cases, algorithms which produce exhaustive lists can be used to prove the existence of a related object [12].

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

Strings with equivalence under rotation is one of the most fundamental types of combinatorial objects. Such objects, more commonly known as necklaces, arise naturally in many areas including knot theory, color printing, DNA sequencing, and the theory of free Lie algebras. Algorithms for generating necklaces and Lyndon words (aperiodic necklaces) were first developed by Fredricksen and Kessler [6] and Fredricksen and Maiorana [7]. These algorithms were proven to be CAT by Ruskey, Savage, and Wang [11].

Many applications, however, do not require all necklaces, but instead only those satisfying a particular restriction. A recursive necklace generation algorithm outlined in [2] has led to several algorithms which efficiently generate restricted classes of

*Received by the editors August 24, 2000; accepted for publication (in revised form) January 23, 2001; published electronically July 25, 2001. This research was supported by NSERC and partial support of Czech grant GAČR 201/99/0242 and ITI under project LN-00A 056.
http://www.siam.org/journals/sicomp/31-1/37703.html

†Department of Computer Science, University of Sydney, Sydney, Australia (sawada@cs.usyd.edu.au).

necklaces including binary unlabeled necklaces [2], fixed density necklaces [12], and necklaces with forbidden substrings [13].

Another restricted class of necklaces are bracelets. More specifically, bracelets are necklaces with equivalence under string reversal. Lists of bracelets are shown to have application in the calibration of color printers by Emmel [5]. However, the problem of efficiently generating these lists has remained open for some time. Previously, the fastest known algorithm to generate bracelets was a modification of Savage and Wang's necklace algorithm [11] by Lisonek [10]. This algorithm has running time $O(n \cdot B_k(n))$ (where $B_k(n)$ denotes the number of k-ary bracelets of length n), which is the same as the second algorithm outlined in the beginning of section 3.

The problem of efficiently generating bracelets is answered in this paper with the development of a bracelet generation algorithm that runs in constant amortized time. We begin with some background and definitions of the relevant objects in section 2. In section 3, we outline our bracelet generation algorithm. In section 4, we discuss strings with no 0^i substring (forbidden substrings). These strings are then used when we analyze our bracelet generation algorithm in section 5.

2. Background. We define a *necklace* to be the lexicographically smallest element of an equivalence class of k-ary strings under rotation. The set of all necklaces of length n is denoted $\mathbf{N}_k(n)$. The cardinality of $\mathbf{N}_k(n)$ is denoted $N_k(n)$. An aperiodic necklace is called a *Lyndon word*. The set of all k-ary Lyndon words of length n is denoted $\mathbf{L}_k(n)$ and has cardinality $L_k(n)$. A word α is called a *prenecklace* if it is the prefix of some necklace. The set of all k-ary prenecklaces of length n is denoted $\mathbf{P}_k(n)$. The cardinality of $\mathbf{P}_k(n)$ is denoted $P_k(n)$.

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

Necklaces, Lyndon words, and prenecklaces can all be generated using the recursive necklace generation algorithm $\mathsf{GenNecklaces}(t, p)$ shown in Figure 2.1. It is important to have a solid understanding of this algorithm because it will be the basis for the bracelet generation algorithm developed in the following section. The basic idea behind the algorithm is to generate all length n prenecklaces, and then perform an appropriate test in the function $\mathsf{PrintIt}(p)$ to obtain the desired object. If necklaces are required, then the prenecklace is printed only if p divides n; if Lyndon words are required, then the prenecklace is printed if $n = p$. If $\alpha = a_1 \cdots a_{t-1}$ is a prenecklace with its longest Lyndon prefix having length p, then a length t prenecklace can be obtained by appending any value greater than or equal to a_{t-p} to α. The initial call is $\mathsf{GenNecklaces}(1,1)$ and a_0 is initialized to 0.

The following theorem provides enumeration formulas for necklaces, Lyndon words, prenecklaces, and bracelets.

THEOREM 2.1. *The following formulas are valid for all $n \geq 1$, $k \geq 1$:*

$$L_k(n) = \frac{1}{n} \sum_{d|n} \mu(d) k^{n/d}, \tag{2.1}$$

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

FIG. 2.1. *The recursive necklace algorithm.*

$$N_k(n) = \frac{1}{n}\sum_{d|n}\phi(d)k^{n/d}, \tag{2.2}$$

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

$$B_k(n) = \begin{cases} \frac{1}{2}(N_k(n) + \frac{k+1}{2}k^{n/2}), & n \textit{ even}, \\ \\ \frac{1}{2}(N_k(n) + k^{(n+1)/2}), & n \textit{ odd}. \end{cases} \tag{2.4}$$

Proof. The equations for $L_k(n)$, $N_k(n)$, and $B_k(n)$ are proved by Gilbert and Riordan in [8]. The equation for $P_k(n)$ is proved in [2]. $\square$

In the analysis of our bracelet algorithm it will be useful to look at another way to count prenecklaces. Let $P^0_k(n)$ count all k-ary prenecklaces of length n that begin with 0. Notice that the number of k-ary prenecklaces of length n beginning with 1 is equal to $P^0_{k-1}(n)$. Similarly the number of k-ary prenecklaces of length n beginning with 2 is $P^0_{k-2}(n)$. This observation leads to the following equation:

$$P_k(n) = \sum_{j=1}^{k} P^0_j(n). \tag{2.5}$$

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

A more sophisticated approach will use a necklace finding algorithm, which determines the necklace of a length n string in $O(n)$ time. Such an algorithm is easily derived from Duval's algorithm for factoring a string into Lyndon words [4] or from Theorem 2.1 in [2]. Using this technique, we need only compare the generated necklace with the necklace of its reversal. This approach yields a much better running time of $O(n \cdot B_k(n))$ to generate bracelets; however, it is still far from being CAT.

In the quest to find a faster algorithm to generate bracelets, we return to the original idea of comparing a generated necklace to every rotation of it reversal. We start by making a simple observation.

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

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

Following the above approach, we still need to perform additional testing for the reversals starting at position j where $a_1 \cdots a_j = a_j \cdots a_1$. The number of such reversals could be as many as $n/2$. The following theorem addresses this issue.

THEOREM 3.1. *If $a_1 \cdots a_n$ is a necklace where $a_1 \cdots a_q = a_q \cdots a_1$ and there exists an r in $\{q+1, \ldots, n\}$ such that $a_1 \cdots a_r = a_r \cdots a_1$ and $a_{r+1} \cdots a_n \leq a_n \cdots a_{r+1}$, then $a_{q+1} \cdots a_n \leq a_n \cdots a_{q+1}$.*

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

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

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

The following is a summary of the modifications required to transform GenNecklaces(t, p) into an algorithm which generates bracelets. Notice that each modification requires only a constant amount of computation per character generated except the addition of the function CheckRev(t, i).

- Add the parameter u to maintain the value of i from Observation 1: the number of consecutive equivalent characters at the start of the prenecklace (i.e., the prenecklace starts with a^u).
- Add the parameter v to maintain the number of consecutive a's at the end of the prenecklace, where $a = a_1$.
- Add the function CheckRev(t, i) to compare the prenecklace to its reversal (when $u = v$). If the prenecklace is greater than its reversal, it returns -1; if the prenecklace is less than its reversal, it returns 0; otherwise, the prenecklace is the same as its reversal and 1 is returned.
- Add the parameter r to maintain the length of the longest prenecklace equal to its reversal (i.e., the largest value r for which $a_1 \cdots a_r = a_r \cdots a_1$).
- Add a test to each character in a position greater than $(n-r)/2 + r$ which

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

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

FIG. 3.1. *Bracelet generation algorithm.*

will determine whether or not $a_r \cdots a_n$ is greater than its reversal. This will involve the additional parameter RS to hold intermediate boolean values indicating whether or not the reversal is smaller (RS).

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

The resulting algorithm GenBracelets(t, p, r, u, v, RS) is shown in Figure 3.1. The initial call is GenBracelets(1,1,0,0,0,FALSE). To illustrate this algorithm we trace the parameters as the string 0010023003 gets generated:

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

In the following section we give several counting results for strings with no 0^i substring. These results will then be applied when we analyze the algorithm, showing that it runs in constant amortized time.

4. Forbidden substrings. We denote the set of all k-ary strings of length n with no 0^i substring by $\mathbf{I}_k(n,i)$. The cardinality of this set, denoted $I_k(n,i)$, is given by the following recurrence relation:

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

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

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

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

Lemma 4.1. *If $k, i \ge 2$, then*

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

Proof. The base cases when $n \le i$ are trivial. If $n > i$, then we induct on n:

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

Lemma 4.2. *If $n > 2$ and $k, i \ge 2$, then*

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

Proof.

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

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

THEOREM 4.3. *If $n > 2$ and $k, i \geq 2$, then*

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

Proof.

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

5. Analysis of the algorithm. In this section we show that the algorithm GenBracelets for generating bracelets is CAT. We analyze the algorithm by looking at the computation tree and determining the amount of computation done at each node. To get a bound on the size of the bracelet computation tree, we observe the following bounds obtained from (2.1) and (2.2) along with Lemma 4.4 from [12]:

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

Now using (2.4) we get the following bounds on the number of bracelets:

$$\frac{k^n}{2n} \leq B_k(n) \leq 2\frac{k^n}{n}. \tag{5.1}$$

Since the necklace algorithm GenNecklaces is CAT [2], the size of its computation tree is less than ck^n/n for some constant c. This bound is also true for GenBracelets since its computation tree is smaller than that of GenNecklaces. However, unlike the necklace computation tree, the bracelet computation tree has some nodes that require more than a constant amount of computation. From our algorithm, these nodes are the ones that make a call to CheckRev. Thus, to prove the bracelet generation algorithm GenBracelets is CAT, we must show that the computation performed by all calls to CheckRev is bounded by some constant times the total number of bracelets generated. The task of analyzing this extra computation is divided into the following four subsections.

5.1. Identifying the prenecklaces. From the algorithm, each node that makes a call to CheckRev is a prenecklace of the form a^i or $a^i\gamma a^i$ where the nonempty string γ begins and ends with a character lexicographically greater than a. Note that the length of such prenecklaces is at most $n-1$. Each call to CheckRev results in computation proportional to $(t-2i)/2$, where t is the length of the prenecklace. Since any prenecklace of the form a^i requires only constant computation in CheckRev (because $t=i$), we can restrict our attention to prenecklaces of the form $a^i\gamma a^i$. To simplify this task we consider only the prenecklaces beginning with 0, later using (2.5) to account for the remaining prenecklaces. We also ignore the fact that many of these prenecklaces are never generated by the algorithm (i.e., the prenecklace 002100300 is never generated since the prenecklace 002100 is terminal).

The next series of observations are crucial to the success of the analysis. Notice that the number of prenecklaces of the form $0^i\gamma 0^i$ is less than or equal to the number of prenecklaces of the form $0^i\gamma$. We now group these prenecklaces together according to length. Such strings will have length of at least 2, but not greater than $n-2$. Define the set of all k-ary prenecklaces of length n beginning with 0, ending with a nonzero character, and with no 0^i substring to be $\mathbf{P}'_k(n,i)$. Equivalently, the set $\mathbf{P}'_k(n,i)$ contains all prenecklaces with length n of the form $0^j\gamma$ for $1 \le j < i$. The cardinality of this set is denoted as $P'_k(n,i)$. If we let $E_k(n)$ denote the extra computation that results from all calls made to CheckRev by prenecklaces beginning with 0 (while generating $\mathbf{B}_k(n)$), then we obtain the following bound:

$$E_k(n) \le \sum_{i=2}^{n-2} \frac{n-i}{2} P'_k(n-i,i). \tag{5.2}$$

5.2. Bounding the restricted prenecklaces. In this subsection we find an upper bound for $P'_k(n,i)$ first using restricted Lyndon words, and then in terms of strings with forbidden substrings. Because every prenecklace is obtained as a prefix of a β^* where β is some Lyndon word, we arrive at the formula given in (2.3):

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

If we let $L_k(j,i)$ denote the number of Lyndon words of length j with no 0^i substring, then we obtain the following upper bound for $P'_k(n,i)$:

$$P'_k(n,i) \le \sum_{j=1}^{n} L_k(j,i). \tag{5.3}$$

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

LEMMA 5.1. *If $n \ge 1$ and $i \ge 1$, then*

$$L_k(n,i) \le \frac{1}{n} I_k(n,i).$$

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

Using the previous lemma and Theorem 4.3 ($n > 2$) we can simplify the upper bound in (5.3). Note that the latter bound is also satisfied when $n = 2$.

$$\begin{aligned} P'_k(n,i) &\le \sum_{j=1}^{n} \frac{1}{j} I_k(j,i) \\ &\le \frac{8}{n} I_k(n,i). \end{aligned}$$

5.3. Converting back to prenecklaces. Using the bound discovered in the previous subsection, we can now substitute back into (5.2) and simplify:

$$\begin{aligned} E_k(n) &\le \sum_{i=2}^{n-2} \frac{n-i}{2} P'_k(n-i,i) \\ &\le 4 \sum_{i=2}^{n-2} I_k(n-i,i). \end{aligned}$$

We now use a clever trick to bound this sum in terms of prenecklaces. Observe that we can insert 0^i1 at the front of each string in $\mathbf{I}_k(n-i,i)$ to obtain a new a set of strings of length $n+1$. Notice that each new string is a unique prenecklace regardless of the parameter i. Thus the number of strings in the union of the sets $\mathbf{I}_k(n-i,i)$ for $i = 2, \ldots, n-1$ is less than $P_k(n+1)$. We can divide this total by $k-1$, since we could have arbitrarily chosen any of $k-1$ characters to insert after 0^i. Thus

$$\begin{aligned} E_k(n) &\le \frac{4}{k-1} P_k(n+1) \\ &\le \frac{4k}{k-1} P_k(n) \\ &\le 8 \sum_{j=1}^{n} L_k(j) \\ &\le 8 \sum_{j=1}^{n} \frac{k^j}{j} \\ &\le 24 \frac{k^n}{n}. \end{aligned} \tag{5.4}$$

The simplification found in (5.4) is valid for $k \ge 2$ and can easily be proved by induction.

5.4. Accounting for all prenecklaces. Because the bound on $E_k(n)$ is only for prenecklaces beginning with 0, we use (2.5) to get an upper bound on the extra computation performed by all prenecklaces. Note that $E_1(n) = 0$.

$$\begin{aligned} ExtraWork &\le \sum_{j=2}^{k} E_j(n) \\ &\le \frac{24}{n} \sum_{j=2}^{k} j^n \\ &\le 48 \frac{k^n}{n}. \end{aligned}$$

From (5.1), the total number of bracelets generated is bounded below by $k^n/2n$. Thus, the running time of the algorithm GenBracelets is proportional to the number of bracelets generated, which proves the following theorem.

THEOREM 5.2. *The k-ary bracelet generation algorithm GenBracelets is CAT.*

Experimentally, the constant is less than 8 where we compare the number of calls to GenBracelets plus the number of iterations of the for loop in CheckRev to the number of bracelets generated.

Acknowledgments. The author would like to thank Frank Ruskey for many helpful discussions in all aspects of this paper, as well as the anonymous referee who suggested a simpler proof for Theorem 3.1.

REFERENCES

[1] L. Batton, C. Bohun, A. Bona, K. Cheng, T. Doman, J. Drew, R. Edwards, S. Kutay, C. Laflamme, D. McCrea, W. Myrvold, F. Ruskey, J. Sawada, P. van den Driessche, J. Vander Kloet, and K. Wood, *Classification of chemical compound pharmacophore structures*, in Proceedings of the Third PIMS Industrial Problem Solving Workshop, C. Bose, ed., 1999, pp. 83–93.

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

[3] W. Chen and J. Louck, *Necklaces, MSS sequences, and DNA sequences*, Adv. Appl. Math., 18 (1997), pp. 18–32.

[4] J.-P. Duval, *Factoring words over an ordered alphabet*, J. Algorithms, 4 (1983), pp. 363–381.

[5] P. Emmel and R. Hersch, *Exploring ink spreading*, in Proceedings of the 8th IS & T/SID Color Imaging Conference: Color Science and Engineering, Scottsdale, AZ, 2000, pp. 335–341.

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

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

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

[9] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, Cambridge, UK, 1997.

[10] P. Lisonek, *Computer-Assisted Studies in Algebraic Combinatorics*, dissertation, Johannes Kepler University, Linz, Austria, 1994.

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

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

[13] F. Ruskey and J. Sawada, *Generating necklaces and strings with forbidden substrings*, in Computing and Combinatorics, Lecture Notes in Comput. Sci. 1858, D.-Z. Du, P. Eades, V. Estivill-Castro, X. Lin, and A. Sharma, eds., Springer-Verlag, New York, 2000, pp. 330–339.