Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT®

ELSEVIER

Theoretical Computer Science 301 (2003) 477–489

Theoretical Computer Science

www.elsevier.com/locate/tcs

Note

A fast algorithm to generate necklaces with fixed content

Joe Sawada[1]

Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ont. Canada M5S 1A4

Received 2 December 2001; received in revised form 22 September 2002; accepted 9 January 2003
Communicated by A. Del Lungo

Abstract

We develop a fast algorithm for listing all necklaces with fixed content. By fixed content, we mean the number of occurrences of each alphabet symbol is fixed. Initially, we construct a simple but inefficient algorithm by making some basic modifications to a recursive necklace generation algorithm. We then improve it by using two classic combinatorial optimization techniques. An analysis using straight forward bounding techniques is used to prove that the algorithm runs in constant amortized time.
© 2003 Elsevier Science B.V. All rights reserved.

Keywords: Necklace; Lyndon word; Fixed content; CAT algorithm; Exhaustive generation; Combinatorial optimization

1. Introduction

The development of algorithms to list all non-isomorphic occurrences of some combinatorial object is a fundamental pursuit within the realm of theoretical computer science. Such algorithms find application in many diverse areas including: hardware and software testing, combinatorial chemistry, coding theory, and computational biology. In addition, such lists are often studied with the hope of gaining a more thorough understanding of a particular object.

When developing such algorithms, the ultimate performance goal is for the amount of computation to be proportional to the number of objects generated. Such algorithms are said to be CAT for constant amortized time. When analyzing such algorithms, the

[1] Research supported by NSERC.
E-mail address: jsawada@cs.toronto.edu (J. Sawada).

0304-3975/03/$ - see front matter © 2003 Elsevier Science B.V. All rights reserved.
doi:10.1016/S0304-3975(03)00049-5

correct measure of computation is the total amount of data structure change and not the time required to print the objects. This is because typical applications only process the part of the object that has undergone some change.

The object of focus in this paper is the necklace (for a thorough background and history on necklaces, consult [2]). A *necklace* is the lexicographically smallest element in an equivalence class of strings under rotation. An important restricted class of binary necklaces is the one where the number of zeroes is fixed. Such necklaces are said to have *fixed density*. The generation of fixed density necklaces was first investigated in [3]. Subsequently, a Gray code was developed in [7], and a CAT algorithm was developed in [5].

A more general scenario (posed as an open problem in [5]) is to consider k-ary necklaces where the number of occurrences of every alphabet symbol is fixed. Such necklaces are said to have *fixed content*. Chord diagrams are shown to be a restricted class of such necklaces in [6]. Necklaces with fixed content can also be used as a basis for obtaining all necklaces where the *sum* of the alphabet symbols is fixed. Until now, no efficient algorithm was known for generating necklaces with fixed content.

The primary result in this paper is the development of an elegant and efficient algorithm for generating k-ary necklaces with fixed content. We start by presenting a simple but inefficient generation algorithm based on a recursive necklace generation algorithm. Then, by applying only basic combinatorial optimization techniques, we modify the simple algorithm. An analysis which also uses basic combinatorial techniques proves that the resulting algorithm achieves the ultimate performance goal of running in constant amortized time. In the final section, we remark that these results also hold for the generation of Lyndon words with fixed content.

2. Background

Recall that a *necklace* is the lexicographically smallest element of an equivalence class of k-ary strings under rotation, where the alphabet (w.l.o.g.) is $\{0, 1, 2, \ldots, k-1\}$. An aperiodic necklace is called a *Lyndon word*. A *prenecklace* is a prefix of a necklace. The following table illustrates these objects for $n=4$ and $k=2$. Note that 0110 is a prenecklace since it is a prefix of the necklace 011011.

Prenecklace	Necklace	Lyndon word
0000	0000	
0001	0001	0001
0010		
0011	0011	0011
0101	0101	
0110		
0111	0111	0111
1111	1111	

Table 1
Notation of the necklace related objects

Object with fixed content	Set	Cardinality
Necklaces	$\mathbf{N}_k(n_0, n_1, \ldots, n_{k-1})$	$N_k(n_0, n_1, \ldots, n_{k-1})$
Lyndon words	$\mathbf{L}_k(n_0, n_1, \ldots, n_{k-1})$	$L_k(n_0, n_1, \ldots, n_{k-1})$
Prenecklaces	$\mathbf{P}_k(n_0, n_1, \ldots, n_{k-1})$	$P_k(n_0, n_1, \ldots, n_{k-1})$
Strings	$\mathbf{S}_k(n_0, n_1, \ldots, n_{k-1})$	$S_k(n_0, n_1, \ldots, n_{k-1})$

The following theorem from [2], uses the function lyn, which when applied to a string α returns the length of its longest prefix that is a Lyndon word. For example, $lyn(0110012) = 3$, $lyn(0001) = 4$, and $lyn(2100221) = 1$. In this theorem, the set of all k-ary prenecklaces of length n is denoted by $\mathbf{P}_k(n)$.

Theorem 1 (Fundamental theorem of necklaces). *Let $\alpha = a_1 a_2 \cdots a_{n-1} \in \mathbf{P}_k(n-1)$ and $p = lyn(\alpha)$. The string $\alpha b \in \mathbf{P}_k(n)$ if and only if $a_{n-p} \leqslant b \leqslant k-1$. Furthermore,*

$$lyn(\alpha b) = \begin{cases} p & \text{if } b = a_{n-p}, \\ n & \text{if } a_{n-p} < b \leqslant k-1. \end{cases}$$

Using this theorem, it is a easy to develop a recursive algorithm for generating prenecklaces [2]. This method of object generation is related to the general ECO method described in [1]. If α is a prenecklace with $p = lyn(\alpha)$, then α is a necklace if and only if $n \bmod p = 0$ and α is a Lyndon word if and only if $n = p$. Hence, this theorem can also be applied to generate both necklaces and Lyndon words.

The following corollary to this theorem is frequently used in the analysis of our algorithm.

Corollary 1. *If $\alpha = a_1 a_2 \cdots a_n$ is a Lyndon word, then αb is a prenecklace for all $a_1 \leqslant b \leqslant k-1$.*

We say that a string object has *fixed content* if the number of occurrences of each alphabet symbol is fixed. Table 1 shows the symbols used to represent each relevant set of objects along with the symbols used to enumerate them. Each object is assumed to be over an alphabet of size k and to have n_i occurrences of the symbol i for $0 \leqslant i \leqslant k-1$. For example, the set of all k-ary necklaces with n_i occurrences of each symbol i is denoted $\mathbf{N}_k(n_0, n_1, \ldots, n_{k-1})$ and has cardinality $N_k(n_0, n_1, \ldots, n_{k-1})$.

As an example, we show the sets of necklaces, aperiodic necklaces, and prenecklaces where $k = 2$, $n_0 = 3$, and $n_1 = 3$.

$$\mathbf{N}_2(3,3) = \{000111, 001011, 001101, 010101\},$$

$$\mathbf{L}_2(3,3) = \{000111, 001011, 001101\},$$

$$\mathbf{P}_2(3,3) = \mathbf{N}_2(3,3) \cup \{001110, 010110\}.$$

It is easy to enumerate the number of strings with fixed content and enumeration formulae for necklaces and Lyndon words with fixed content are derived in [4]. In the following formulae, it is assumed that each $n_i \geqslant 1$ and $k \geqslant 1$.

$$S(n_0, n_1, \ldots, n_{k-1}) = \frac{n!}{n_0! \cdots n_{k-1}!}, \tag{1}$$

$$N(n_0, n_1, \ldots, n_{k-1}) = \frac{1}{n} \sum_{j \mid gcd(n_0, n_1, \ldots, n_{k-1})} \phi(j) \frac{(n/j)!}{(n_0/j)! \cdots (n_{k-1}/j)!}, \tag{2}$$

$$L(n_0, n_1, \ldots, n_{k-1}) = \frac{1}{n} \sum_{j \mid gcd(n_0, n_1, \ldots, n_{k-1})} \mu(j) \frac{(n/j)!}{(n_0/j)! \cdots (n_{k-1}/j)!}. \tag{3}$$

The Euler totient function on a positive integer n, denoted $\phi(n)$, is the number of integers in the set $\{0, 1, \ldots, n-1\}$ that are relatively prime to n. The Möbius function of a positive integer n, denoted $\mu(n)$, is $(-1)^r$ if n is the product of r distinct primes and 0 otherwise.

No enumeration formula is known for prenecklaces with fixed content.

3. Algorithm

In this section, we develop a fast algorithm for generating necklaces with fixed content. First, we construct a simple algorithm by making some basic modifications to the recursive necklace algorithm [2] derived from Theorem 1. We then improve this algorithm using two classic combinatorial optimization techniques. The final algorithm is proved to run in constant amortized time in the following section. In these algorithms it is assumed that $n = n_0 + n_1 + \cdots + n_{k-1}$ where $n_i > 0$ for $0 \leqslant i \leqslant k-1$.

3.1. *A simple algorithm*

In this subsection we develop a simple algorithm to generate the necklaces $\mathbf{N}(n_0, n_1, \ldots, n_{k-1})$ by making two small modifications to the recursive necklace algorithm obtained from Theorem 1 [2]. The basic idea behind the recursive necklace algorithm is to repeatedly extend prenecklaces until they have length n. Those prenecklaces that are not necklaces are rejected. The first modification is to decrement the value n_i when a character i is added to the prenecklace being generated. Secondly, we do not attempt to append the character i if $n_i = 0$. Pseudocode for such an algorithm is shown in Fig. 1. If the lines labeled (1), (2), (3) and (4) are removed from the pseudocode, then we are left with the original necklace generation algorithm. Since we assume that $n_0 > 0$, the first character in any necklace must be 0. Thus a_1 is initialized to 0 and we decrement n_0 before making the initial call of SimpleFixedContent(2,1). To illustrate the algorithm, we show the computation tree for $\mathbf{N}(2, 1, 3)$ in Fig. 2.

Although this algorithm is very simple, experimental results indicate that the algorithm is not CAT. There are two reasons why this algorithm is not efficient. First, each iteration of the **for** loop does not necessarily generate a recursive call. In the worst

```
    procedure SimpleFixedContent ( $t$, $p$ : integer );
    local $j$ : integer;
    begin
        if $t > n$ then begin
            if $n \bmod p = 0$ then Print();
        end;
        else begin
            for $j \in \{a_{t-p}, \ldots, k-2, k-1\}$ do begin
(1)             if $n_j \neq 0$ then begin
                    $a_t := j$;
(2)                 $n_j := n_j - 1$;
                    if $j = a_{t-p}$ then SimpleFixedContent( $t+1$, $p$ );
                    else SimpleFixedContent( $t+1$, $t$ );
(3)                 $n_j := n_j + 1$;
(4)             end;
            end;
    end; end;
```

Fig. 1. A simple algorithm to generate $\mathbf{N}(n_0, n_1, \ldots, n_{k-1})$.

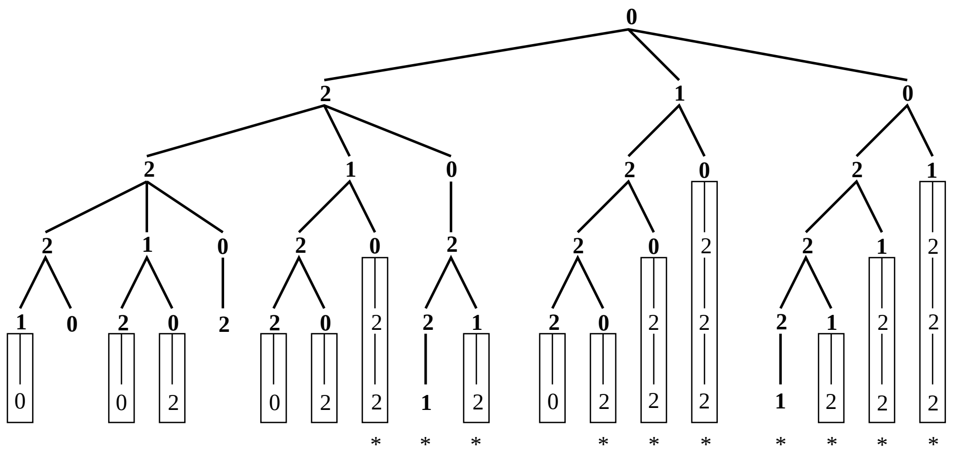

Fig. 2. The computation tree for $\mathbf{N}(2,1,3)$ using SimpleFixedContent(t, p). Each node represents a prenecklace that is obtained by tracing a path from the root to the node. The nodes with a * below them represent the necklaces in $\mathbf{N}(2,1,3)$. If we ignore the vertices inside the rectangles, which are the 0 and 2 chains, then we obtain the computation tree for $\mathbf{N}(2,1,3)$ using FastFixedContent(t, p, s).

case it is possible to do a linear amount of work, O(k), to generate only a constant number of recursive calls. Second, many prenecklaces end with a *chain*, or strings of consecutive characters that have the same value. For example, given the content $n_0 = 2, n_1 = 1, n_2 = 3$, the necklace 010222 contains a chain of length 3. This problem gives rise to many nodes in the computation tree with only one child. We attack these problems separately in the next two subsections.

```
procedure ListFixedContent ( $t$, $p$ : integer );
local $j$: integer;
begin
    if $t > n$ then begin
        if $n \bmod p = 0$ then Print();
    end;
    else begin
        $j := head$;
        while $j \geqslant a_{t-p}$ do begin
            $a_t := j$;
            $n_j := n_j - 1$;
            if $n_j = 0$ then Remove($j$);

            if $j = a_{t-p}$ then ListFixedContent($t+1, p$);
            else ListFixedContent($t+1, t$);

            if $n_j = 0$ then Add($j$);
            $n_j := n_j + 1$;
            $j :=$ Next($j$);
end; end; end;
```

Fig. 3. An algorithm to generate $\mathbf{N}(n_0, n_1, \ldots, n_{k-1})$ using an ordered list.

3.2. Modified algorithm using an ordered list

To keep track of which characters are available to be appended to the current prenecklace, we introduce a global ordered list. Initially this list is composed of the values 0 through $k-1$. As before, each time a value j gets appended to a prenecklace by the assignment $a_t := j$, the value n_j is decremented. If the updated value for n_j is 0, then the value gets removed from the list. After each recursive call, the list must be restored to its initial state. Thus, if the updated value is $n_j = 0$, we must add it back to the list.

If we order this list from smallest to largest, we can replace the **for** loop with a **while** loop which traverses the list starting with the smallest available value j such that $j \geqslant a_{t-p}$. This will have the advantage of skipping over all the values i where $n_i = 0$. However, we still have the problem of finding the smallest available value j such that $j \geqslant a_{t-p}$. Such a search will take time $O(k)$ in the worst case. Thus, we are still faced with the possibility of performing a linear amount of work to make a constant number of recursive calls.

To solve this problem, we order the list from largest to smallest and generate the strings in *reverse* lexicographic order. The **while** loop now traverses the list from the largest value, which is the *head* of the list, until a value less than a_{t-p} is reached. This approach will yield a constant amount of work per recursive call.

Pseudocode which applies this ordered list to generate the necklaces $\mathbf{N}(n_0, n_1, \ldots, n_{k-1})$ is shown in Fig. 3. Again, we initialize a_1 to 0 and decrement n_0, but now we must also remove 0 from the list if the updated value of $n_0 = 0$. The initial call is ListFixedContent(2,1).

The operation Remove(j) can be implemented to remove the value j from the ordered list in constant time. The operation Add(j) adds the value j back to the ordered list. This function can also be implemented in constant time since it is only called (effectively) right after a call to Remove(j). The variable *head* represents the first and largest value in the list and the operation Next(j) will return the next value less than j in the list, or -1 if there is no next element. A list with these operations can be implemented using an array where each cell has next and previous pointers.

3.3. Chain removal

With the implementation of the global ordered list to maintain the available characters, every recursive call is the result of a constant amount of work. Now we focus on the problem of removing chains. This problem can be attacked using a classic strategy of initializing each character in the string $a_1 \cdots a_n$ to the maximum value $k-1$. Thus, before attempting to assign a value to a_t, we first check to see if the only remaining characters are $k-1$. This will be the case if the number of remaining positions $n-t+1$ equals the value n_{k-1} (Note: this value reflects the number of characters $k-1$ that remain to be placed in the prenecklace being generated—*not* the total number of occurrences of the character). In this case we can stop the recursion because the remaining positions $a_t \cdots a_n$ have already been pre-assigned the value $k-1$.

Using such a strategy, we still need to perform a test which determines whether or not the resulting length n prenecklace is a necklace. To perform this test in constant time we need to know the number of consecutive values equal to $k-1$ starting at position a_{t-p}. If this data is stored in r_{t-p} then by applying Theorem 1, the prenecklace will be a necklace if $n_{k-1} > r_{t-p}$ or if $n_{k-1} = r_{t-p}$ and $n \bmod p = 0$.

Example. Consider the prenecklace 022210 where $k=3$, $t=7$, $p=5$, $r_{t-p}=3$, and $n_{k-1}=n-t+1$. If $n=8$ then the resulting prenecklace 02221022 is not a necklace since $n_{k-1}=2$. If $n=9$ then the prenecklace 022210222 is also not a necklace since $n_{k-1}=3=r_{t-p}$ but $n \bmod p \neq 0$. However if $n=10$, the prenecklace 0222102222 is a necklace since $n_{k-1}=4>r_{t-p}$.

To maintain the array of values $r_1 \cdots r_n$ we require the use of another parameter s which stores the starting point of the current run of values $k-1$. If the current length of such a run is 0, then the parameter s is assigned the value t. For example, consider the prenecklace 002333 where $t=7$ (the position for the next character to be placed in the string). If $k-1=3$ then $s=4$. If $k-1=4$, then $s=t=7$. Before each recursive call the value $t-s$ gets assigned to r_s. Note that this value can be changed as the length of the run of $k-1$'s increases. Also observe that r_j does not get an assignment if $a_{j-1}=k-1$. This information, however is not required since when we want to consider r_{t-p}, it must be the case that $a_{t-1} \neq k-1$ and hence $a_{t-p-1} \neq k-1$.

Note that this strategy does not remove the chains for values other than $k-1$. However, by observing that no necklace (except 0^n) ends with 0, we can also eliminate chains ending with 0 by stopping the recursion if $n_0 = n-t+1$. To ensure that the

```
procedure FastFixedContent ( $t$, $p$, $s$ : integer );
local $j$, $s'$ : integer;
begin
    if $n_{k-1} = n - t + 1$ then begin
        if $n_{k-1} = r_{t-p}$ and $n \bmod p = 0$ then Print();
        else if $n_{k-1} > r_{t-p}$ then Print();
    end;
    else if $n_0 \neq n - t + 1$ then begin
        $j := head$;
        $s' := s$;
        while $j \geqslant a_{t-p}$ do begin
            $r_s := t - s$;
            $a_t := j$;
            $n_j := n_j - 1$;
            if $n_j = 0$ then Remove($j$);
            if $j \neq k - 1$ then $s' := t + 1$;

            if $j = a_{t-p}$ then FastFixedContent($t + 1, p, s'$);
            else FastFixedContent($t + 1, t, s'$);

            if $n_j = 0$ then Add($j$);
            $n_j := n_j + 1$;
            $j :=$ Next($j$);
        end;
        $a_t := k - 1$;
end; end;
```

Fig. 4. A fast algorithm to generate $\mathbf{N}(n_0, n_1, \ldots, n_{k-1})$.

worst chaining problem occurs with the value $k - 1$, the content should be ordered so that the value $k - 1$ has the maximum number of occurrences.

Recall that for each of these algorithms the value $n = n_0 + n_1 + \cdots + n_{k-1}$, where each value n_i for $0 \leqslant i \leqslant k-1$ initially represents the number of occurrences of the character i in the generated necklaces. Pseudocode that removes the chains of $k - 1$'s and 0's from the computation tree is shown in Fig. 4. The values a_i for $2 \leqslant i \leqslant n$ are initialized to $k - 1$ and the values r_i for $1 \leqslant i \leqslant n$ are initialized to 0. Before we make the initial call of FastFixedContent(2,1,2), we perform the same intializations as required by the previous algorithm ListFixedContent(t, p): we set a_1 to 0, decrement n_0, and remove 0 from the list of available characters if the updated value of $n_0 = 0$. Observe that in the case $n_{k-1} = n - t + 1 = 0$, the prenecklace is a necklace if $n \bmod p = 0$ as before. Note that this necklace still gets printed by this algorithm since $a_{t-p} = 0$ and thus $r_{t-p} = 0 = n_{k-1}$.

To compare the computation tree generated by this algorithm for $\mathbf{N}(2, 1, 3)$ to the computation tree generated by simple algorithm SimpleFixedContent(t, p), see Fig. 2. In this figure, the computation tree for FastFixedContent(t, p, s) is the one obtained by removing the 0 and 2 chains that appear inside the rectangles of the computation tree for SimpleFixedContent(t, p).

4. Analysis

In this section, we analyze the algorithm FastFixedContent(t, p, s) for generating the set of necklaces $\mathbf{N}(n_0, n_1, \ldots, n_{k-1})$. It is assumed that the character with the maximum number of occurrences is $k-1$ implying that $n_i \leqslant n_{k-1}$ for all $0 \leqslant i < k-1$.

The approach this algorithm takes is to recursively generate prenecklaces $\mathbf{P}(m_0, m_1, \ldots, m_{k-1})$ where $m_i \leqslant n_i$ for all $0 \leqslant i \leqslant k-1$. Because of the optimization that removes $k-1$ chains from the end of the prenecklaces, the algorithm does not generate prenecklaces in $\mathbf{P}(n_0, \ldots, n_{k-2}, m_{k-1})$ ending with the character $k-1$ where $m_{k-1} < n_{k-1}$. Similarly, because of the optimization which removes 0 chains from the end of the prenecklaces, the algorithm does not generate prenecklaces in $\mathbf{P}(m_0, n_1, \ldots, n_{k-1})$ ending with the character 0 where $m_0 < n_0$. From this discussion, we let $\mathbf{P}(t)$ denote the set of prenecklaces with length t generated by this algorithm.

Since the algorithm is recursive, we can view each generated prenecklace as a node in a computation tree. If we let $CompTree(n_0, n_1, \ldots, n_{k-1})$ denote the size of the computation tree then:

$$CompTree(n_0, n_1, \ldots, n_{k-1}) = \sum_{j=1}^{n} |\mathbf{P}(j)|.$$

Due to the implementation of the ordered list, each prenecklace, or node, in the computation tree is a result of a constant amount of work. Thus, to prove that the algorithm runs in constant amortized time, we must show that $CompTree(n_0, n_1, \ldots, n_{k-1})$ is less than some constant times the total number of necklaces generated. Our goal in the remainder of this section will be to prove that the following expression holds for some constant c:

$$CompTree(n_0, n_1, \ldots, n_{k-1}) \leqslant cN(n_0, n_1, \ldots, n_{k-1}).$$

If $\alpha = a_1 a_2 \cdots a_t$ is a prenecklace, then the *parent* of α is $a_1 \cdots a_{t-1}$; we say that α is a *child* of $a_1 \cdots a_{t-1}$. A prenecklace with no children is called a *leaf*.

Lemma 1. *Every leaf in the computation tree has a unique parent.*

Proof. Every prenecklace in $\mathbf{P}(n-1)$ has at most one child and every prenecklace in $\mathbf{P}(n)$ is a leaf. Otherwise, suppose that a prenecklace $a_1 \cdots a_t$ in the computation tree has at least two children $a_1 \cdots a_t u$ and $a_1 \cdots a_t w$ for some characters $u < w$ where $t < n-1$. This means that $a_1 \cdots a_t w$ must be a Lyndon word and thus has a child $a_1 \cdots a_t wu$ (Corollary 1). Therefore every node can have at most one child that is a leaf. $\square$

Let $\mathbf{P}'(j)$ denote the subset of prenecklaces in $\mathbf{P}(j)$ that are not leaves; they have at least one child. Also let $\mathbf{P}_1(j)$ denote the subset of prenecklaces in $\mathbf{P}(j)$ with exactly one child and let $\mathbf{P}_2(j)$ denote the subset of prenecklaces in $\mathbf{P}(j)$ with two or more children. If we let the cardinality of these sets be denoted by $P'(j)$, $P_1(j)$, and $P_2(j)$, respectively, then we have $P'(t) = P_1(t) + P_2(t)$. Using this notation along with the previous lemma we obtain the following bound by mapping each leaf to its unique parent:

$$CompTree(n_0,n_1,\ldots,n_{k-1}) \leqslant 2\sum_{j=1}^{n-1} P'(j).$$

Effectively, we are now examining a subtree of the original computation tree. The next step of this analysis will show that the number of nodes at each successive level of this subtree grows exponentially. This will imply that total number of nodes in this subtree is proportional to the number of nodes at its bottom level $n-1$. Finally, we will show that this number is also proportional to the total number of necklaces generated by the algorithm.

Lemma 2. *The number of prenecklaces in* $\mathbf{P}(n_0,n_1,\ldots,n_{k-1})$ *that are not necklaces and do not end with* 0, *is less than* $L(n_0,n_1,\ldots,n_{k-1})$.

Proof. In the proof of Lemma 4.1 in [5] a mapping is defined that takes a prenecklace that is not a necklace and does not end with a 0 and maps it to a Lyndon word. Since the mapping is both 1-1 and content preserving it is follows that number of prenecklaces in $\mathbf{P}(n_0,n_1,\ldots,n_{k-1})$ that are not necklaces and do not end with 0, is less than $L(n_0,n_1,\ldots,n_{k-1})$. □

Lemma 3. *If* $i<j$ *and* $n_i \leqslant n_j-2$ *then*

$$L(n_0,n_1,\ldots,n_{k-1}) < N(n_0,\ldots,n_{i-1},n_i+1,n_{i+1},\ldots,n_{j-1},n_j-1,n_{j+1},\ldots,n_{k-1}).$$

Proof. Since $n_i n_j < (n_i+1)(n_j-1)$ we can use the enumeration formulas (1), (2), and (3) to obtain:

$$\begin{aligned} L(n_0,n_1,\ldots,n_{k-1}) &\leqslant \frac{1}{n} S(n_0,n_1,\ldots,n_{k-1}) \\ &\leqslant \frac{1}{n} S(n_0,\ldots,n_{i-1},n_i+1,n_{i+1},\ldots,n_{j-1},n_j-1,n_{j+1},\ldots,n_{k-1}) \\ &\leqslant N(n_0,\ldots,n_{i-1},n_i+1,n_{i+1},\ldots,n_{j-1},n_j-1,n_{j+1},\ldots,n_{k-1}). \end{aligned}$$ □

The next lemma shows that the number of prenecklaces with exactly one child is proportional to the number of prenecklaces with at least two children.

Lemma 4. *For* $t \leqslant n-2$:

$$P_1(t) \leqslant 6P_2(t).$$

Proof. We partition $\mathbf{P}_1(t)$ into 3 categories: necklaces, non-necklaces ending with non-zero, and non-necklaces ending with zero. We show that size of each subset is bounded by $2P_2(t)$.

In the first case we let $N_1(t)$ denote the number of necklaces in $\mathbf{P}_1(t)$. It follows from the content preserving map in the proof of Lemma 4.4 in [5], that the number of such necklaces is less than 2 times the number that are Lyndon words. Since each Lyndon word in $\mathbf{P}_1(t)$ has exactly one child, the Lyndon words must be in the set $\mathbf{L}(n_0,\dots,n_{i-1},m_i,n_{i+1},\dots,n_{k-1})$ where $m_i = n_i - (n-t)$ for some $0<i<k-1$. This gives the following bound:

$$N_1(t) \leqslant 2\sum_{i=1}^{k-2} L(n_0,\dots,n_{i-1},m_i,n_{i+1},\dots,n_{k-1}).$$

Since it is assumed that $n_{k-1} \geqslant n_i$ for all $0 \leqslant i < k-1$, we can use Lemma 3 to get

$$N_1(t) \leqslant 2\sum_{i=1}^{k-2} N(n_0,\dots,n_{i-1},m_i+1,n_{i+1},\dots,n_{k-2},n_{k-1}-1).$$

Each necklace α counted by this sum is unique and has two children, namely αi and $\alpha(k-1)$. Thus $N_1(t) \leqslant 2P_2(t)$.

In the second case we consider non-necklaces in $\mathbf{P}_1(t)$ that end with a non-zero character. The number of these strings is bounded by the number of Lyndon words with length t by Lemma 2. Those Lyndon words that have two or more characters left to be placed, form a subset of $\mathbf{P}_2(t)$ (Corollary 1). Otherwise, the number of Lyndon words with only one character left to be placed (not equal to 0 or $k-1$ by the algorithm), is also bounded by $P_2(t)$ as shown in the first case.

In the final case we consider the non-necklaces in $\mathbf{P}_1(t)$ that end with 0; they have the form $a_1a_2\cdots a_{t-1}0$. Again we must focus on how many different characters remain in the unused content. If there are two values left then we can injectively map the prenecklaces to the Lyndon words $0a_1\cdots a_{t-1}$ which have two children by Corollary 1 and hence form a subset of $\mathbf{P}_2(t)$. Otherwise, if there is only one value left (say i which must be non-zero from the algorithm), we can injectively map each prenecklace to the Lyndon words $a_1\cdots a_{t-1}i$ which also have two children—namely $a_1\cdots a_{t-1}i0$ and $a_1\cdots a_{t-1}ii$.

We have now shown that the number of elements in each of the 3 partitions of the set $\mathbf{P}_1(t)$ is bounded by $2P_2(t)$ and thus $P_1(t) \leqslant 6P_2(t)$. □

Lemma 5. *For* $1 \leqslant t \leqslant n-2$:

$$8P'(t) \leqslant 7P'(t+1).$$

Proof. Observe that $P_1(t) + 2P_2(t) \leqslant P(t+1)$ since each node in $P_1(t)$ has one child and each node in $P_2(t)$ has at least two children. Thus, using the previous lemma:

$$\begin{aligned} 8P'(t) &= 8P_1(t) + 8P_2(t) \\ &\leqslant 7P_1(t) + 14P_2(t) \\ &\leqslant 7P'(t+1). \qquad \square \end{aligned}$$

Corollary 2. *For* $1 \leqslant t \leqslant n$:

$$\sum_{j=1}^{t-2} P'(j) \leqslant 7P'(t-1).$$

Proof. We use the previous lemma and induction. The inequality trivially holds for $t = 1, 2, 3$. Inductively, we assume that it is true for $t < n$, and consider $t+1$:

$$\begin{aligned}\sum_{j=1}^{t+1-2} P'(j) &= P'(t-1) + \sum_{j=1}^{t-2} P'(j)\\ &\leqslant P'(t-1) + 7P'(t-1)\\ &\leqslant 8P'(t-1)\\ &\leqslant 7P'(t). \qquad \square\end{aligned}$$

We can now simplify the expression for the bound on the computation tree:

$$\begin{aligned}CompTree(n_0, n_1, \ldots, n_{k-1}) &\leqslant 2\sum_{j=1}^{n-1} P'(j)\\ &\leqslant 2P'(n-1) + 2\sum_{j=1}^{n-2} P'(j)\\ &\leqslant 2P'(n-1) + 14P'(n-1)\\ &= 16P'(n-1).\end{aligned}$$

We require one final lemma to complete this analysis.

Lemma 6. $P'(n-1) \leqslant 2N(n_0, n_1, \ldots, n_{k-1})$.

Proof. Every prenecklace $\alpha \in P(n_0, n_1, \ldots, n_{k-1})$ has a parent in $\mathbf{P}'(n-1)$ unless it ends with a 0 or $k-1$. Thus, since every node in $\mathbf{P}'(n-1)$ has exactly one child, $P'(n-1)$ is bounded above by the number of prenecklaces in $\mathbf{P}(n_0, n_1, \ldots, n_{k-1})$ that do not end with 0 or $k-1$. Using Lemma 2 the number of such prenecklaces that are not necklaces is less than or equal to $L(n_0, n_1, \ldots, n_{k-1})$. Thus, $P'(n-1) \leqslant 2N(n_0, n_1, \ldots, n_{k-1})$. $\square$

This final lemma proves our goal:

$$CompTree(n_0, n_1, \ldots, n_{k-1}) \leqslant cN(n_0, n_1, \ldots, n_{k-1}).$$

Theorem 2. *If* $n_i \leqslant n_{k-1}$ *for all* $0 \leqslant i < k-1$, *then the algorithm* FastFixedContent(t, p, s) *for generating the necklaces* $\mathbf{N}(n_0, n_1, \ldots, n_{k-1})$ *runs in constant amortized time.*

5. Concluding remarks

In this paper, we have described a CAT algorithm to generate necklaces with fixed content. By repeatedly applying this algorithm, we can also generate necklaces where the *sum* of the alphabet symbols is fixed.

The algorithm can also be modified to generate Lyndon words with fixed content (and hence Lyndon words where the sum of the alphabet symbols is fixed) by simply replacing the comparison $(n \bmod p) = 0$ with the comparison $n = p$. Since $L_k(n_0, n_1, \ldots, n_{k-1}) \geqslant \frac{1}{2} N_k(n_0, n_1, \ldots, n_{k-1})$, a result that follows from the content preserving map in the proof of Lemma 4.4 in [5], the modified algorithm will also run in constant amortized time.

The algorithm has been implemented in C and can be downloaded from the "Combinatorial Object Server" at http://www.theory.cs.uvic.ca/∼cos. The C program can also be obtained from the author upon request.

Acknowledgements

The author would like to thank Frank Ruskey for many useful discussions about necklaces.

References

[1] E. Barcucci, A. Del Lungo, E. Pergola, R. Pinzani, ECO: a methodology for the enumeration of combinatorial objects, J. Difference Equations Appl. 5 (1999) 435–490.

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

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

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

[5] F. Ruskey, J. Sawada, An efficient algorithm for generating necklaces of fixed density, SIAM J. Comput. 29 (1999) 671–684.

[6] J. Sawada, A fast algorithm for generating non-isomorphic chord diagrams, SIAM J. Discrete Math. 15 (4) (2002) 546–561.

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