Theoretical Computer Science 475 (2013) 103–112

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Generating bracelets with fixed content

S. Karim [a,*], J. Sawada [b], Z. Alamgir [a], S.M. Husnine [c]

[a] *Department of Computer Science, National University FAST-NU, Faisal Town Lahore, Pakistan*
[b] *School of Computer Science, University of Guelph, Guelph, Ont., Canada N1G 2W1*
[c] *Department of Mathematics, National University FAST-NU, Faisal Town Lahore, Pakistan*

A R T I C L E   I N F O

Article history:
Received 20 December 2011
Received in revised form 9 August 2012
Accepted 16 November 2012
Communicated by G. Italiano

Keywords:
Necklace
Bracelet
Fixed-content
Unicyclic graphs
CAT algorithm

A B S T R A C T

We present an algorithm to generate bracelets with fixed content. An analysis shows that the algorithm runs in constant amortized time. The algorithm can be applied to efficiently list all non-isomorphic unicyclic graphs with n vertices.

1. Introduction

The exhaustive generation of combinatorial objects has become a popular area of algorithmic research and is a major theme in Knuth's latest volume of *The Art of Computer Programming* [5]. Such algorithms are often analyzed with an amortized analysis where the ultimate goal is to develop an algorithm that runs in Constant Amortized Time: each successive object is generated in constant time on average. An algorithm that attains this goal is said to be CAT.

One of the most fundamental of combinatorial objects is the necklace. A *necklace* is defined to be the lexicographically minimal string in an equivalence class of k-ary strings under rotation. An aperiodic necklace is a *Lyndon word* and any prefix of a necklace is said to be a *prenecklace*. CAT algorithms to generate necklaces, Lyndon words, and prenecklaces of length n are well known [3,2,1]. If the number of occurrences of each character i is given by n_i where $n_0 + n_1 + \cdots + n_{k-1} = n$, then such strings are said to have *fixed content*. CAT algorithms are also known for necklaces and Lyndon words with fixed content [7].

If we consider equivalence under reversal in addition to rotation, we obtain a bracelet. Specifically, a *bracelet* is defined to be the lexicographically minimal string in an equivalence class of k-ary strings under rotation and reversal. While a CAT algorithm is known for generating bracelets [6], no such algorithm is known for bracelets with fixed content. A major contribution of this paper is to fill this void. While doing so, we also provide a much simpler analysis that proves the bracelet algorithm in [6] is CAT.

The study of bracelets with fixed content is motivated by an application related to undirected graphs with exactly one cycle. Exhaustively listing all non-isomorphic graphs on n vertices is well known to be a very difficult problem. However, if we restrict such graphs to have exactly 1 cycle, then the crux of an efficient algorithm is to be able to efficiently list bracelets

* Corresponding author. Tel.: +92 42111128128.
E-mail addresses: sairakarim@gmail.com, saira.karim@nu.edu.pk (S. Karim), jsawada@uoguelph.ca (J. Sawada), zareen.alamgir@nu.edu.pk (Z. Alamgir), s.m.husnine@nu.edu.pk (S.M. Husnine).

0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2012.11.024

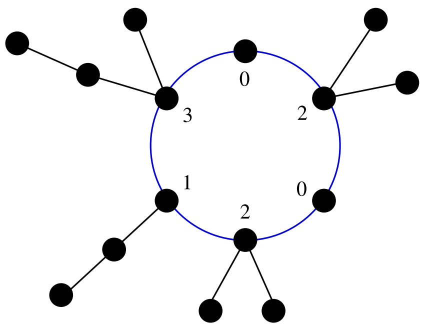


Fig. 1. A unicyclic graph with $n = 15$ vertices. The specification for the sizes of the 6 subtrees is $\langle 1, 1, 3, 3, 3, 4\rangle$. The content assigned to the trees is [0, 0, 1, 2, 2, 3], and the bracelet corresponding to this graph is 020213.

with fixed content. A unicyclic graph G on n vertices can be represented by a sequence of m rooted trees $T_1, T_2, \dots, T_m$ where the root of each tree is a vertex of the unique cycle and the total number of vertices in the m trees is n. Fig. 1 shows a unicyclic graph on $n = 15$ vertices from a sequence of $m = 6$ rooted trees. Equivalence classes are formed by considering rotations and the reversal of the sequences, i.e., bracelets with fixed content. If we partition the unicyclic graphs with n vertices by the size of the cycle m, then the following approach outlines how to exhaustively generate them:

> A numerical partition $\langle p_1, p_2, \dots, p_m\rangle$ of the integer n into m parts corresponds to a specification for the sizes of the m rooted trees on the cycle. For each specification we consider all combinations of rooted trees whose sizes match the specification. Then for each combination of trees, we map each tree to a unique alphabet symbol in $\{0, 1, 2, \dots, m-1\}$: if two rooted trees are the same, they will map to the same alphabet symbol. The resulting multi-set of m symbols yields the content. To handle equivalence under rotation and reversal, the remaining problem is to generate all bracelets with the given content. Fig. 1 illustrates some of these steps.

A more detailed description of this algorithm is given in [4].

The remainder of the paper is outlined as follows. In Section 2, we describe a recursive algorithm to generate necklaces and then describe some simple modifications to obtain a naïve algorithm to generate bracelets with fixed content. We then apply 5 optimizations to obtain a more efficient algorithm. In Section 3, we analyze the optimized algorithm proving that it is CAT. In Section 4, we give a short summary. In the Appendix we provide a complete C implementation of our algorithm.

2. Algorithms to generate bracelets with fixed content

In this section we present two algorithms to generate bracelets with fixed content. The first algorithm applies straightforward modifications to a recursive necklace algorithm, but is unoptimized. The second algorithm is also based on the recursive necklace algorithm, but applies the optimizations from CAT algorithms to generate (i) necklaces with fixed content [7] and (ii) bracelets [6]. When merging the optimizations, one special case must be handled in order to preserve the optimizations used in each approach. Additionally, in order to make the merged algorithm slightly more optimized and easier to analyze, we maintain an additional representation for the k-ary string being generated: its run-length encoding (detailed in Section 2.2.1).

2.1. A simple algorithm

In [1], the *Fundamental Theorem of Necklaces* specifies the exact conditions for a character to be appended to a prenecklace and still remain a prenecklace. All that is required is the length of the current prenecklace α and the length of its longest prefix that is a Lyndon word, given by $lyn(\alpha)$.

Theorem 1 (*Fundamental Theorem of Necklaces*). *Let $\alpha = a_1a_2\cdots a_{t-1}$ be a k-ary prenecklace with $p = lyn(\alpha)$. The string αb is a k-ary prenecklace if and only if $a_{t-p} \le b \le k-1$. Furthermore,*

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

Using this theorem, it is straightforward to produce a recursive algorithm to exhaustively list all prenecklaces of length n in lexicographic order. A pseudocode is provided in Fig. 2(a), where the parameter p represents the longest Lyndon prefix of the current prenecklace. The function Print(p) is used to output each prenecklace and it can easily be modified to output necklaces or Lyndon words. A prenecklace is a necklace if $n \bmod p = 0$; it is a Lyndon word if $n = p$. Each object can be generated in constant amortized time [1]. The initial call is Necklace(1,1) with a_0 initialized to 0.

Using this algorithm we now consider our two restrictions. First, we only want to generate bracelets. Second, we want the strings to satisfy a pre-specified content: $n_0 + n_1 + \cdots + n_{k-1} = n$ where each n_i denotes the number of occurrences of

```
procedure Necklace($t, p$: int)
    $j, p'$: int

    if $t > n$ then Print($p$)
    else
        for $j := a_{t-p}$ to $k-1$ do

            $a_t := j$
            $p' := p$
            if $j \neq a_{t-p}$ then $p' := t$

            Necklace($t+1, p'$)

end.
```

(a)

```
procedure SimpleBFC($t, p, r$: int)
    $c, j, p'$: int

    if $t > n$ then  if $a_{r+1} \cdots a_n \le a_n \cdots a_{r+1}$ then Print($p$)
    else
        for $j := a_{t-p}$ to $k-1$ do
            $n_j := n_j - 1$
            $a_t := j$
            $p' := p$
            if $j \neq a_{t-p}$ then $p' := t$
            $c :=$ CheckRev($t$)
            if $c = 0$ and $n_j \ge 0$ then SimpleBFC($t+1, p', t$)
            if $c = 1$ and $n_j \ge 0$ then SimpleBFC($t+1, p', r$)
            $n_j := n_j + 1$
end.
```

(b)

Fig. 2. (a) A simple recursive algorithm Necklace(t, p) to list all necklaces, Lyndon words or prenecklaces depending on the restrictions given by the function Print(p). (b) A simple algorithm SimpleBFC(t, p, r) to list all bracelets with fixed content.

the character i. To apply the first restriction, it is possible to apply a $O(n)$ time test to determine whether or not the necklace is a bracelet. This can be done by computing the necklace of the reversed string and comparing it to the original necklace. However, this will not lead to a CAT algorithm. Instead, we apply the following result which follows directly from Theorem 3.1 of [6]:

Lemma 1. *If $\alpha = a_1a_2\cdots a_n$ is a necklace where r denotes the length of its longest prefix such that $a_1\cdots a_r = a_r\cdots a_1$, then α is a bracelet iff $a_{r+1}\cdots a_n \le a_n\cdots a_{r+1}$ and there is no index t such that $a_1\cdots a_t > a_t\cdots a_1$.*

To apply this lemma, at each recursive call in the necklace algorithm we must compare $a_1\cdots a_t$ with its reversal. If it is greater than its reversal, we terminate the branch since no extension of the prenecklace will result in a bracelet; if they are equal, then we update the value for a new parameter r. When the prenecklace has length n we compare $a_{r+1}\cdots a_n$ with its reversal to test if it is a bracelet.

To naïvely restrict the content of each bracelet, we only extend the prenecklaces with characters that do not violate the restriction. This is easily handled by updating the number of occurrences n_i for each character i as it gets appended to a prenecklace. Applying these modifications, a pseudocode for a simple algorithm SimpleBFC(t, p, r) to generate bracelets with fixed content is given in Fig. 2(b). The initial call is SimpleBFC(1, 1, 0) with a_0 initialized to 0. The function CheckRev(t) compares the prefix $a_1\cdots a_t$ with its reversal. Its return value is given by:

$$\text{CheckRev}(t) = \begin{cases} -1 & \text{if } a_1\cdots a_t > a_t\cdots a_1 \\ 0 & \text{if } a_1\cdots a_t = a_t\cdots a_1 \\ 1 & \text{if } a_1\cdots a_t < a_t\cdots a_1. \end{cases}$$

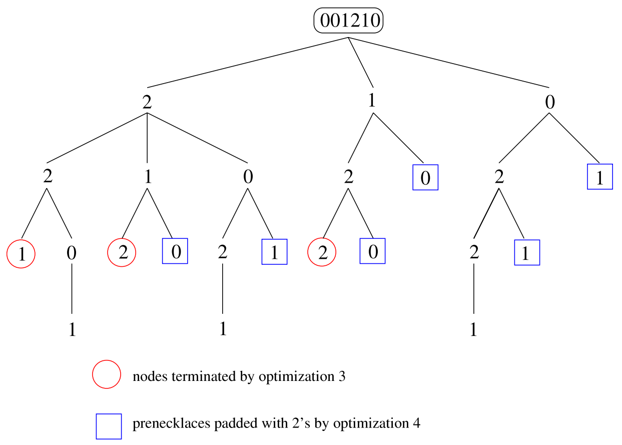
Observe that each call to CheckRev(t) requires $O(t)$ time in the worst case; however when $a_1 \neq a_t$ only one comparison is required. In the next subsection, the run-length encoding of the string is maintained to make this test more efficient.

2.2. An efficient algorithm

In this section we address three optimizations for fixed content necklaces from [7] and one optimization for bracelets from [6]. Since there is a dependence between two of the optimizations, maintaining the run-length encoding for the string being generated is critical to maintaining the efficiency. In total, the 5 optimizations can be summarized as follows:

1. Maintain the run-length encoding, which optimizes the function CheckRev.
2. Use a linked list to maintain the characters remaining to be added
3. Truncate the current branch of computation when only 0's remain to be added, since the string will not result in a bracelet of length n.
4. Initialize the last n_{k-1} characters of α to k−1, which allows a branch of computation to be trimmed when only $k-1$'s remain to be added.
5. Incrementally compare $a_{r+1}\cdots a_n$ with its reversal $a_n\cdots a_{r+1}$ by making one character comparison per recursive call and maintaining a parameter storing the current result.

For completeness, these 5 optimizations are discussed in more detail in the following subsections. To illustrate the optimizations, a fragment of a computation tree is given in Fig. 3. A pseudocode that applies all the optimizations is provided in Fig. 4. The initial call is BraceletFC(2, 1, 1, 2, 1, FALSE) where a_1 is initialized to 0 since all bracelets must start with 0. To apply the fourth optimization, the last n_{k-1} characters of $\alpha = a_1\cdots a_n$ are initialized to k−1.



001210
nodes terminated by optimization 3
prenecklaces padded with 2's by optimization 4

Fig. 3. A fragment of the computation tree for BraceletFC starting with the prenecklace 001210 and remaining content $n_0 = 1, n_1 = 1, n_2 = 2$. This fragment generates 9 prenecklaces: the 3 ending with 1 and the 6 that ended early by the fourth optimization. Only the prenecklaces that end early (padded with 2's) correspond to bracelets in this case. The ones that end with 1 are necklaces but do not pass the bracelet test.

```
procedure BraceletFC($t, p, r, z, b$: int; $RS$: boolean)
    $c, j, z', p'$: int

    // Update RS so it is TRUE iff $a_{t-1}a_{t-2}\cdots a_{n-t+2+r} < a_{n-t+2+r}\cdots a_{t-2}a_{t-1}$
    if $t-1 > \lfloor (n-r)/2 \rfloor + r$ then
        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

    // Termination condition - only characters $k-1$ remain to be appended
    if $n_{k-1} = n-t+1$ then
        if $n_{k-1} > run_{t-p}$ then $p := n$
        if $n_{k-1} > 0$ and $r+1 \neq t$ and $s_{b+1} = k-1$ and $v_{b+1} > n_{k-1}$ then $RS :=$ TRUE
        if $n_{k-1} > 0$ and $r+1 \neq t$ and $(s_{b+1} \neq k-1$ or $v_{b+1} < n_{k-1})$ then $RS :=$ FALSE
        if $RS =$ FALSE then Print($p$)

    // Recursively extend the prenecklace - unless only 0's remain to be appended
    else if $n_0 \neq n-t+1$ then
        $j := head$
        while $j \geq a_{t-p}$ do
            $run_z := t-z$
            UpdateRunLength($j$);
            $n_j := n_j - 1$
            if $n_j = 0$ then ListRemove($j$)

            $a_t := j$
            $z' := z$
            if $j \neq k-1$ then $z' := t+1$
            $p' := p$
            if $j \neq a_{t-p}$ then $p' := t$
            $c :=$ CheckRev($nb$)
            if $c = 0$ then BraceletFC( $t+1, p', t, z', nb$, FALSE )
            if $c = 1$ then BraceletFC( $t+1, p', r, z', b, RS$ )

            if $n_j = 0$ then ListAdd($j$)
            $n_j := n_j + 1$
            RestoreRunLength();

            $j :=$ ListNext($j$)

        $a_t := k-1$
end.
```

Fig. 4. An optimized algorithm BraceletFC(t, p, r, z, b, RS) to list all bracelets with fixed content.

2.2.1. *Maintaining the run-length encoding*

The *run-length encoding* of a k-ary string is a compact representation where the string is represented by a sequence of pairs (s_i, v_i) where s_i is a character element in the string and v_i is the number of occurrences of s_i in a run. Moreover, consecutive pairs (s_i, v_i) and (s_{i+1}, v_{i+1}) must represent different characters: $s_i \neq s_{i+1}$. For example, the run-length encoding of $\alpha = 0000022211112$ is $(0, 5), (2, 3), (1, 4), (2, 1)$. For simplicity, we call each pair (s_i, v_i) a *block* and use *nb* to denote the number of blocks in the run-length encoding of a string. From our example, $nb = 4$.

As a character a_t is appended to a string $a_1 \cdots a_{t-1}$, its run-length encoding is updated as follows: if $a_t = a_{t-1}$ then increment v_{nb}; otherwise add a new block $(a_t, 1)$ and increment *nb*. To restore the encoding after a recursive call, we consider the value v_{nb}: if it is greater than 1 then its value is decremented by 1; otherwise the last block is removed and the value of *nb* is decremented by 1. In the pseudocode in Fig. 4, the run-length encoding and the variable *nb* are stored globally and these constant time operations are performed by the procedures UpdateRunLength(j) and RestoreRunLength( ) respectively.

Using this encoding, it becomes more efficient to implement CheckRev(t), which compares $a_1 \cdots a_t$ to its reversal $a_t \cdots a_1$. Instead of comparing single characters at each step, we can compare entire blocks. Specifically, the following function CheckRev(m) can be used to compare the run-length encoding $(s_1, v_1), (s_2, v_2), \ldots, (s_m, v_m)$ with its reversal. Instead of t (the length of the string), this function now receives the number of blocks m as the parameter.

```
function CheckRev($m$: int) returns int
    $j$: int

    $j := 1$
    while $(s_j, v_j) = (s_{m-j+1}, v_{m-j+1})$ and $j \leq \frac{m}{2}$ do $j := j + 1$

    if $j > \frac{m}{2}$ then return 0
    if $s_j < s_{m-j+1}$ then return 1
    if $s_j > s_{m-j+1}$ then return -1
    if $(v_j < v_{m-j+1}$ and $s_{j+1} < s_{m-j+1})$ or $(v_j > v_{m-j+1}$ and $s_j < s_{m-j})$ then return 1
    return -1
end.
```

2.2.2. *Fixed-content optimizations*

Now we consider optimizations specific to the content restriction. Looking back at our simple algorithm in Fig. 2(b) observe that the **for** loop could iterate multiple times without producing a recursive call. This will happen when many of the n_i are already reduced to 0. An obvious optimization is to maintain a linked list containing only the characters that can be successfully appended to the current prenecklace. By maintaining the list in descending order, a loop can be constructed that produces a recursive call for each iteration. The subroutines ListAdd(j) and ListRemove(j) can easily be implemented to respectively add and remove the element j from the list. The global variable *head* provides the first element in the list, and ListNext(j) returns the value after j in the list. Each function can easily be implemented in constant time using an array representation for a doubly linked list (see C code in Appendix).

The third optimization is to terminate any branch of computation when only the character 0 remains to be appended, since for any $k > 0$ it will not lead to a necklace. This is easily done with a constant time comparison of n_0 to $n - t + 1$.

The fourth optimization is to end a branch of computation early when only the character $k-1$ remains to be appended. This trims the computation by n_{k-1} recursive calls, where n_{k-1} refers to the remaining number of $k-1$'s to be added. By initializing the string α to consist entirely of this character and restoring its value appropriately as we backtrack, the string α will be as desired. With respect to the run-length encoding, it amounts to adding the block $(k - 1, n_{k-1})$. A side effect of truncating such branches early is that the value for p will not be updated to handle these last n_{k-1} characters. This is important since p is used to test if the prenecklace is a necklace or Lyndon word by the Print(p) procedure. The key to updating p in constant time is to determine the number of consecutive $k - 1$'s that begin from position a_{t-p}, if any. As explained in [7], this number can be determined in constant time per recursive call by maintaining an extra parameter z, and an array *run*. In particular, if the prenecklace $a_1a_2 \cdots a_t$ has $a_t = k - 1$, the parameter z indicates the leftmost position of the run of $k-1$'s in the suffix; otherwise if $a_t \neq k-1$ then z is set to $t+1$. The value run_j stores the number of consecutive $k - 1$'s starting at position j. Using this value, if n_{k-1} is greater than run_{t-p}, then p gets updated to n; otherwise it remains unchanged.

2.2.3. *Bracelet optimizations*

We now focus on the optimization specific to bracelets. Observe that the final test before printing compares $a_{r+1} \cdots a_n$ to its reversal. If this test is done as the last character is appended, it may take linear time. However, if we already know whether or not $a_{n-1}a_{n-2} \cdots a_{r+2} < a_{r+2} \cdots a_{n-2}a_{n-1}$ then it is a trivial matter to perform this test in constant time by additionally comparing a_{r+1} with a_n. So the key is to repeat this strategy with a boolean parameter *RS* that maintains whether or not the *R*eversal is *S*maller as we incrementally add characters after the midpoint of $a_{r+1} \cdots a_n$. The initial value for this parameter is FALSE and the incremental updating of this parameter is given in the first block of the pseudocode in Fig. 4. Since the most recent character added is $t-1$, the resulting value for RS indicates whether or not (the reversal) $a_{t-1}a_{t-2}\cdots a_{n-t+2+r}$ is smaller than $a_{n-t+2+r}\cdots a_{t-2}a_{t-1}$. Observe that $t-1=n$ when the final character of the string has been added. In this case, after the fragment has been executed, the value RS will be TRUE if and only if the reversal $a_n\cdots a_{r+1}$ is smaller than $a_{r+1}\cdots a_n$.

2.2.4. Merging the optimizations

There is one complication to merging the fixed content and bracelet optimizations. Since the fourth optimization may truncate the computation early, the final incremental comparisons to accurately update RS will not be performed. Ideally, this would be done in constant time otherwise it renders the fixed-content optimization to be ineffective. Fortunately, this is attainable using the run-length encoding together with maintaining the block index b for the number of blocks used to represent $a_1\cdots a_r$. Thus, b is updated with r is updated. Observe that a new block always starts at position $r+1$ by the definition of r: $a_r=0$ and a_{r+1} must be greater than 0. Using this information, we can update the variable RS in constant time when the computation is truncated by $n_{k-1}>0$ steps by comparing the $b+1$-st block (s_{b+1}, v_{b+1}) with the last block $(k-1, n_{k-1})$. If $t=r+1$, then we are comparing the same block to itself, so no update is required. Otherwise if $t\neq r+1$, then RS gets updated to TRUE if $s_{b+1}=k-1$ and $v_{b+1}>n_{k-1}$ (the reversal is smaller); RS gets updated to FALSE if $s_{b+1}\neq k-1$ or $v_{b+1}<n_{k-1}$.

3. Analysis

In this section, we prove that the algorithm BraceletFC to generate bracelets with fixed content runs in constant amortized time. The algorithm can be loosely thought of as taking the fixed-content necklace algorithm from [7] and adding the reversal tests for bracelets from [6]. However, applying the same analysis that was done for bracelets is not applicable since complex bounding arguments were applied that did not respect the content of the strings, i.e., the merging of two CAT algorithms does not guarantee that the result is a CAT algorithm. The approach used in our new analysis is to map the block comparisons performed by the function CheckRev to prenecklaces in the computation tree. This idea also yields a much simpler analysis of the original bracelet algorithm in [6] when the run-length encoding of the string is maintained.

The recursive computation tree for our algorithm is a subtree of the computation tree for the fixed-content algorithm of [7]. The latter algorithm to generate necklaces with fixed content is CAT when each $n_i\le n_{k-1}$ for $0\le i<k-1$. Thus, since there are at most 2 necklaces in each bracelet equivalence class, the size of the computation tree of BraceletFC will be proportional to the number of bracelets generated. If each recursive call was a result of a constant amount of work, this would be sufficient to prove that our algorithm is CAT. Unfortunately, the function CheckRev may require more than a constant amount of computation. However, by showing that the total work done by all calls to CheckRev is also proportional to the size of the computation tree we will prove that the algorithm BraceletFC is CAT.

The function CheckRev(m), as outlined in Section 2.2.1, is called once for each prenecklace in the computation tree. The parameter m denotes the number of blocks in the run-length encoding of the prenecklace. The work done by a single call is dominated by the **while** loop which iterates until two unequal blocks are compared, or until $m/2$ comparisons have been made. Since there is at most one unequal comparison made per prenecklace, we focus only on the equal block comparisons. To further simplify the analysis, we consider only every second comparison starting from the 4th block comparison. This number of comparisons will be proportional to the total number of comparisons when 4 or more comparisons are required; otherwise the work done by the function is constant. Our strategy is to map each such block comparison to a unique prenecklace in the computation tree.

Let $\beta=B_1B_2\cdots B_m$ be the run-length encoding of prenecklace tested by a call to CheckRev(m), where $B_i=(s_i, v_i)$. Since the first character in any prenecklace generated by the algorithm is 0, $s_1=0$. Moreover, since β is a prenecklace, B_1 must be a block with a maximal run of 0s: there is no block $B_i=(0, v_i)$ such that $v_i>v_1$. Consider the following mapping, where j is even with $4\le j<m/2$ and $B_1B_2\cdots B_j=B_mB_{m-1}\cdots B_{m-j+1}$:

$$f(\beta,j)=\begin{cases}B_mB_1B_jB_2B_3\cdots B_{j-1}B_{j+1}\cdots B_{m-2} & \text{if } s_j=0\\ B_mB_1B_{j-1}B_2B_3\cdots B_{j-2}B_j\cdots B_{m-2} & \text{if } s_j\neq 0.\end{cases}$$

In the following two lemmas we will show that $f(\beta,j)$ maps uniquely (1-1) to a prenecklace in the computation tree for BraceletFC.

Lemma 2. *If $\beta=B_1B_2\cdots B_m$ is the run-length encoding of a prenecklace from the computation tree of BraceletFC such that $B_1B_2\cdots B_j=B_mB_{m-1}\cdots B_{m-j+1}$ and $4\le j\le m/2$ is even, then $f(\beta,j)$ is also a prenecklace of the same computation tree of BraceletFC.*

Proof. Observe that the sequence of blocks in $f(\beta,j)$ does not correspond to a valid run-length encoding since $s_m=s_1=0$: the listing of blocks is not minimal. Also, depending on the case, either B_{j-1} and B_{j+1} or B_{j-2} and B_j may also be blocks of the same character; however that character will not be 0 by the nature of the mapping. Thus, the string given by $f(\beta, j)$ will have a maximum substring of 0s uniquely at the start of the string and hence it is a prenecklace. To see that $f(\beta, j)$ is a prenecklace in a the computation tree for BraceletFC, we must consider 3 items:

1. The content $f(\beta, j)$ is precisely the content of β with the content from B_{m-1} removed. Therefore, $f(\beta, j)$ respects the restriction on content (i.e., the number of occurrences of each symbol i is less than or equal to n_i).
2. The two optimizations that trim or truncate computation detailed in Section 2.2 are not applied to any proper prefix of $f(\beta, j)$. The third optimization is applied when only 0s remain to be added. Since consecutive blocks do not contain the same content and because $s_m = 0$, it must be that $s_{m-1} \neq 0$. Thus, because this non-zero content of B_{m-1} remains to be added, this optimization will not be applied. The fourth optimization is applied when only $k-1$'s remain to be added. Again, since consecutive blocks do not contain the same content, either s_{m-2} or s_{m-1} will not be $k-1$. Thus, this optimization will not be applied to any proper prefix of $f(\beta, j)$.
3. A prefix of $f(\beta, j)$ will not be rejected by a bracelet reversal test since the maximum number of 0s appears uniquely at the beginning of the prenecklace. □

Lemma 3. *If $\beta = B_1B_2 \cdots B_m$ is the run-length encoding of a prenecklace such that $B_1B_2 \cdots B_j = B_mB_{m-1} \cdots B_{m-j+1}$ and $4 \leq j \leq m/2$ is even, then the mapping f is 1-1.*

Proof. The proof is by contradiction. Suppose that the mapping f is not 1-1. Then there exist prenecklaces $\beta = B_1B_2 \cdots B_m$ and $\gamma = B'_1B'_2 \cdots B'_{m'}$ such that $f(\beta, j) = f(\gamma, j')$ for some j and j' satisfying the conditions of the lemma. If $\beta = \gamma$ then $j \neq j'$. WLOG assume that $j \leq j'$. Observe that $B_{m-j+1} \cdots B_{m-1} = B'_{m'-j+1} \cdots B'_{m'-1}$ because the last half of each prenecklace in the mapping remains unchanged except for moving the last block of 0s to the beginning and dropping the second to last block. Thus, since $B_1 \cdots B_j = B_m \cdots B_{m-j+1}$ and $B'_1 \cdots B'_{j'} = B'_{m'} \cdots B'_{m'-j'+1}$ we have $B_3 \cdots B_j = B'_3 \cdots B'_j$. If $j = j'$ then this implies that $\beta = \gamma$, a contradiction. Thus $j' \geq j + 2$ and the first j+2 blocks from each mapping are illustrated as follows:

$$f(\beta, j) = B_mB_1B_xB_2 \;\; B_3 \cdots B_{j-2} \;\; B_yB_{j+1} \cdots$$
$$f(\gamma, j') = B'_{m'}B'_1B'_zB'_2 \;\; B_3 \cdots B_{j-2} \; B_{j-1}B_j \cdots ,$$

where x and y are either $j-1$ or j, and $z > j$. Since adjacent blocks in the original run-length encodings of β and γ must represent different characters, it is not difficult to see that the strings represented by the first 4 blocks specified in these mappings must be the same. The next $j-4$ blocks are also the same in each mapping. However, the following 2 blocks in each mapping will correspond to different strings since $B_j \neq B_{j+1}$, a contradiction. Thus f is 1-1. □

Together, Lemmas 2 and 3 imply that every second comparison (after a small constant amount) required by the routine CheckRev can be mapped uniquely to a prenecklace in the computation tree. As discussed earlier, the number of such prenecklaces is proportional to the number of bracelets generated. This gives the following lemma.

Lemma 4. *From an initial call to BraceletFC, the total amount of computation for all calls to CheckRev is proportional to the number of bracelets generated given that $n_0, n_1, \ldots, n_{k-1}$ where $n_i \leq n_{k-1}$ for all $0 \leq i < k-1$.*

This result immediately gives us our main theorem.

Theorem 2. *Given content $n_0, n_1, \ldots, n_{k-1}$ where $n_i \leq n_{k-1}$ for all $0 \leq i < k-1$, the algorithm BraceletFC runs in constant amortized time.*

Lemmas 2 and 3 also provide a simple proof that the algorithm to generate k-ary bracelets given in [6] is CAT, provided the algorithm also maintains the run-length encoding of the prenecklaces.

4. Summary

We develop an algorithm to list all bracelets with fixed content. Using a fairly simple technique of mapping comparisons to nodes in the recursive computation tree, we are able to prove that the algorithm runs in constant amortized time. The analysis also yields a simpler proof that the bracelet algorithm in [6] is CAT, as long as the run-length encoding is maintained. As an application, the algorithm is critical to the efficient generation of all non-isomorphic uni-cyclic graphs [4].

A complete C implementation of our algorithm is given in the Appendix.

Acknowledgments

First author's research supported by FAST-NU. Second author's research supported by NSERC.

Appendix. Complete C program to generate bracelets with fixed content

```
#include <stdio.h>
#define TRUE  1
#define FALSE 0

typedef struct cell {
        int next,prev;
} cell;

typedef struct element {
        int s, v;
} element;

cell avail[50];
element B[50];            // run length encoding data structure
int nb = 0;                // number of blocks
int num[50], a[50],run[50],n,k,total,head, NECK=1, LYN=0;

/*-------------------------------------------------------------*/
void ListRemove(int i) {
    int p,n;

    if (i == head) head = avail[i].next;
    p = avail[i].prev;
    n = avail[i].next;
    avail[p].next = n;
    avail[n].prev = p;
}

void ListAdd(int i) {
    int p,n;

    p = avail[i].prev;
    n = avail[i].next;
    avail[n].prev = i;
    avail[p].next = i;
    if (avail[i].prev == k+1) head = i;
}

int ListNext(int i) {

    return avail[i].next;
}

/*-------------------------------------------------------------*/
void Print(int p) {
    int j;

    if (NECK && n %p != 0) return;
    if (LYN  && n != p) return;

    for(j=1; j<=n; j++) printf("%d ",a[j]-1);
    printf("\n");
    total++;
}
/*-------------------------------------------------------------*/
void UpdateRunLength(int v) {

    if (B[nb].s == v) B[nb].v = B[nb].v + 1;
    else {
        nb++;
        B[nb].v = 1;
        B[nb].s = v;
    }
}

void RestoreRunLength() {

    if (B[nb].v == 1) nb--;
    else B[nb].v = B[nb].v - 1;
}
/*-------------------------------------------------------------------*/
// return -1 if reverse smaller, 0 if equal, and 1 if reverse is larger
/*-------------------------------------------------------------------*/
int CheckRev() {
    int j;

    j = 1;
    while (B[j].v == B[nb-j+1].v && B[j].s == B[nb-j+1].s && j<= nb/2) j++;

    if (j > nb/2) return 0;
    if (B[j].s < B[nb-j+1].s) return 1;
    if (B[j].s > B[nb-j+1].s) return -1;

    if (B[j].v < B[nb-j+1].v && B[j+1].s < B[nb-j+1].s) return 1;
    if (B[j].v > B[nb-j+1].v && B[j].s < B[nb-j].s) return 1;
    return -1;
}
/*-----------------------------------------------------------*/
void Gen(int t, int p, int r, int z, int b, int RS) {
    int j,z2,p2,c;

    // Incremental comparison of a[r+1...n] with its reversal
    if (t-1 > (n-r)/2 + r) {
        if (a[t-1] > a[n-t+2+r]) RS = FALSE;
        else if (a[t-1] < a[n-t+2+r]) RS = TRUE;
    }
    // Termination condition - only characters k remain to be appended
    if (num[k] == n-t+1) {
        if (num[k] > run[t-p]) p = n;
        if (num[k] > 0 && t != r+1 && B[b+1].s == k && B[b+1].v > num[k]) RS =  TRUE;
        if (num[k] > 0 && t != r+1 && (B[b+1].s != k || B[b+1].v < num[k])) RS = FALSE;
        if (RS == FALSE) Print(p);
    }
    // Recursively extend the prenecklace - unless only 0s remain to be appended
    else if (num[1] != n-t+1) {
        j = head;
        while( j >= a[t-p]) {

            run[z] = t-z;
            UpdateRunLength(j);
            num[j]--;
            if (num[j] == 0) ListRemove(j);

            a[t] = j;
            z2 = z;
            if (j != k)  z2 = t+1;
            p2 = p;
            if (j != a[t-p]) p2 = t;
            c = CheckRev();
            if (c == 0) Gen(t+1,p2,t,z2,nb,FALSE);
            if (c == 1) Gen(t+1,p2,r,z2,b,RS);

            if (num[j] == 0) ListAdd(j);
            num[j]++;
            RestoreRunLength();

            j = ListNext(j);
        }
        a[t] = k;
}    }

/*-------------------------------------------------------------*/
int main() {
    int j;

    printf("enter n k: "); scanf("%d %d", &n, &k);
    for (j=1; j<=k; j++) {
        printf("  enter # of %d's: ", j);
        scanf("%d", &num[j]);
    }

    for (j=k+1; j>=0; j--) {
        avail[j].next = j-1;
        avail[j].prev = j+1;
    }
    head = k;

    for (j=1; j<=n; j++) {
        a[j] = k;
        run[j] = 0;
    }

    total = 0;
    a[1] = 1;
    num[1]--;
    if (num[1] == 0) ListRemove(1);

    B[0].s = 0;
    UpdateRunLength(1);

    Gen(2,1,1,2,1,FALSE);

    printf("Total = %d\n", total);
}
```

References

[1] 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.
[2] H. Fredricksen, I.J. Kessler, An algorithm for generating necklaces of beads in two colors, Discrete Math. 61 (2–3) (1986) 181–188.
[3] H. Fredricksen, J. Maiorana, Necklaces of beads in *k* colors and *k*-ary de Bruijn sequences, Discrete Math. 23 (3) (1978) 207–210.
[4] S. Karim, Z. Alamgir, S.M. Husnine, Generating non-isomorphic uni-cyclic graphs, manuscript, 2011.
[5] D.E. Knuth, The Art of Computer Programming, Volume 4A: Combinatorial Algorithms Part 1, Addison-Wesley, 2011.
[6] J. Sawada, Generating bracelets in constant amortized time, SIAM J. Comput. 31 (1) (2001) 259–268.
[7] J. Sawada, A fast algorithm to generate necklaces with fixed content, Theoret. Comput. Sci. 301 (1–3) (2003) 477–489.