Generating Necklaces and Strings with Forbidden Substrings

Frank Ruskey[1] and Joe Sawada[1]

University of Victoria, Victoria, B.C. V8W 3P6, Canada,
{fruskey,jsawada}@csr.csc.uvic.ca

Abstract. Given a length m string f over a k-ary alphabet and a positive integer n, we develop efficient algorithms to generate
(a) all k-ary strings of length n that have no substring equal to f,
(b) all k-ary circular strings of length n that have no substring equal to f, and
(c) all k-ary necklaces of length n that have no substring equal to f, where f is an aperiodic necklace.
Each of the algorithms runs in amortized time $O(1)$ per string generated, independent of k, m, and n.

1 Introduction

The problem of generating discrete structures with forbidden sub-structures is an area that has been studied for many objects including graphs (e.g., with forbidden minors), permutations (e.g., which avoid the subsequence 312 of relative values), and trees (e.g., of bounded degree). In this paper we are concerned with generating strings that avoid some particular substring. For example, the set of binary strings that avoid the pattern 11 are known as *Fibonacci strings*, since they are counted by the Fibonacci numbers. The set of circular binary strings that avoid 11 are counted by the Lucas numbers. Within combinatorics, the counting of strings which avoiding particular substrings can be handled with the "transfer matrix method" as explained, for example, in Stanley [5]. The ordinary generating function of the number of such strings is always rational, even in the case of circular strings. In spite of the importance of these objects within combinatorics, we know of no papers that explicitly address the problem of efficiently generating all strings or necklaces avoiding a given substring.

The problem of generating strings with forbidden substrings is naturally related to the classic pattern matching problem, which takes as input a pattern P of length m and a text T of length n, and finds all occurrences of the pattern in the text. Several algorithms perform this task in linear time, $O(n+m)$, including the Boyer-Moore algorithm, the Knuth-Morris-Pratt (KMP) algorithm and an automata-based algorithm (which requires non-linear initialization) [2].

The Boyer-Moore algorithm is not suitable for our purposes since it does not operate in real-time. On the other hand, the automata-based algorithm operates in real-time and the KMP algorithm can be adapted to do so [2].

Our algorithms are recursive, generating the string from left-to-right and applying the pattern matching as each character is generated. It is straight forward to generate unrestricted strings in such a recursive manner and adding the pattern matching is easy as well. However, when the pattern is taken circularly, the algorithm and its analysis become considerably more complicated.

The algorithm for generating the necklaces uses the recursive scheme introduced in [4]. This scheme has been used to generate other restricted classes of necklaces, such as unlabelled necklaces [4], fixed-density necklaces [3], and bracelets (necklaces that can be turned over), and is ideally suited for the present problem.

Within the context of generating combinatorial objects, usually the primary goal is to generate each object so that the amount of computation is $O(1)$ per object in an amortized sense. Such algorithms are said to be CAT (for Constant Amortized Time). Clearly, no algorithm can be asymptotically faster. Note that we do not take into account the time to print or process each object; rather we are counting the total amount of data structure change that occurs.

The main result of this paper is the development of CAT algorithms to generate:

- all k-ary strings of length n that have no substring equal to f,
- all k-ary circular strings of length n that have no substring equal to f, and
- all k-ary necklaces of length n that have no substring equal to f, given that f is Lyndon word.

Each algorithm has an embedded automata-based pattern matching algorithm. In principle we could use the same approach to generate unlabelled necklaces, fixed density necklaces, bracelets, or other types of necklaces, all avoiding a forbidden necklace pattern. We expect that such algorithms will also be CAT, but the analysis will be more difficult.

In the following section we provide background and definitions for these objects along with a brief description of the automata-based pattern matching algorithm. In Section 3, we outline the details of each algorithm. We analyze the algorithms, proving that they run in constant amortized time, in Section 4.

2 Background

We denote the set of all k-ary strings of length n with no substring equal to f by $\mathbf{I}_k(n, f)$. The cardinality of this set is $I_k(n, f)$. For the remainder of this paper we will assume that the forbidden string f has length m. Clearly if $m > n$, then $I_k(n, f) = k^n$, and if $m = n$ then $I_k(n, f) = k^n - 1$. If $m < n$, then an exact formula will depend on the forbidden substring f, but can be computed using the transfer matrix method. In Section 4 we derive several bounds on the value of $I_k(n, f)$.

We denote the set of all k-ary circular strings of length n with no substring equal to f by $\mathbf{C}_k(n, f)$. The cardinality of this set is $C_k(n, f)$. In this case, we allow the forbidden string to make multiple passes around the circular string.

Thus, if a string α is in $\mathbf{I}_k(n,f)$ and $m > n$, then it is still possible for the string α to contain the forbidden string f. For example, if $\alpha = 0110$ and $f = 11001100$, then α is *not* in the set $\mathbf{C}_k(4,f)$. We prove that $C_k(n,f)$ is proportional to $I_k(n,f)$ in Section 4.1.

Under rotational equivalence, the set of strings of length n breaks down into equivalence classes of sizes that divide n. We define a *necklace* to be the lexicographically smallest string in such an equivalence class of strings under rotation. An aperiodic necklace is called a *Lyndon word.* A word α is called a *pre-necklace* if it is the prefix of some necklace. Background information, including enumeration formulas, for these objects can be found in [4].

The set of all k-ary necklaces of length n with no substring equal to f is denoted $\mathbf{N}_k(n,f)$ and has cardinality $N_k(n,f)$. The set of all k-ary Lyndon words of length n with no substring equal to f is denoted $\mathbf{L}_k(n,f)$ and has cardinality $L_k(n,f)$. Of course for N_k and L_k we consider the string to be circular when avoiding f. The set of all k-ary pre-necklaces of length n with no substring equal to f is denoted $\mathbf{P}_k(n,f)$ and has cardinality $P_k(n,f)$. A standard application of Burnside's Lemma will yield the following formula for $N_k(n,f)$:

$$N_k(n,f) = \frac{1}{n}\sum_{d|n} \phi(d) C_k(n/d, f). \tag{1}$$

2.1 The automata-based string matching algorithm

One of the best tools for pattern recognition problems is the finite automaton. If $f = f_1 f_2 \cdots f_m$ is the pattern we are trying to find in a string α, then a deterministic finite automaton can be created to process the string α one character at a time, in constant time per character. In other words, we can process the string α in *real-time.* The preprocessing steps required to set up such an automaton can be done in time $O(km)$, where k denotes the size of the alphabet (see [1], pg. 334).

The automaton has $m+1$ states, which we take to be the integers $0, 1, \ldots, m$. The state represents the length of the current match. Suppose we have processed t characters in the string $\alpha = a_1 a_2 \cdots a_n$ and the current state is s. The transition function $\delta(s,j)$ is defined so that if $j = a_{t+1}$ matches f_{s+1}, then $\delta(s,j) = s+1$. Otherwise, $\delta(s,j)$ is the largest state q such that $f_1 \cdots f_q = a_{t-q+2} \cdots a_{t+1}$. If the automaton reaches state m, the only accepting state, then the string f has been found in α. The transition function is efficiently created using an auxiliary function *fail.* The failure function is defined for $1 \le i \le m$ such that *fail*(i) is the length of the longest proper suffix of $f_1 \cdots f_i$ equal to a prefix of f. If there is no such suffix, then *fail*$(i) = 0$. This fail function is the same as the fail function in the classic KMP algorithm.

3 Algorithms

In this section we describe an efficient algorithm to generate necklaces with forbidden necklace substrings. We start by looking at the simpler problem of generating k-ary strings with forbidden substrings and then consider circular strings, focusing on how to handle the wraparound.

If n is the length of the strings being generated and m is the length of the forbidden substring f, then the following algorithms apply for $2 < m \leq n$. We prove that each algorithm runs in constant amortized time in the following section. In the cases where $m = 1$ or 2, trivial algorithms can be developed. For circular strings and necklaces, if $m > n$, then the forbidden substring can be truncated to a length n string, as long as it repeats in a circular manner after the nth character.

3.1 Generating k-ary strings

A naïve algorithm to generate all strings in $\mathbf{I}_k(n, f)$ will generate all k-ary strings of length n, and then upon generation of each string, perform a linear time test to determine whether or not it contains the forbidden substring. A simple and efficient approach for generating strings is to construct a length n string by taking a string of length $n-1$ and appending each of the k characters in the alphabet to the end of the string. This strategy suggests a simple recursive scheme, requiring one parameter for the length of the current string. Since this recursive algorithm runs in constant amortized time, the naïve algorithm will take linear time per string generated.

A more advanced algorithm will embed a real-time automata-based string matching algorithm into the string generation algorithm. Since an automata-based string matching algorithm takes constant time to process each character, we can generate each new character in constant time. We store the string being generated in $\alpha = a_1 a_2 \cdots a_n$ and at each step, we maintain two parameters: t and s. The parameter t represents the next position in the string to be filled, and the parameter s represents the state in the finite automata produced for the string f. Recall that each state s is an integer value that represents the length of the current match. Thus if we begin a recursive call with parameters t and s, then $f_1 \cdots f_s = a_{t-s} \cdots a_{t-1}$. We continue generating the current string as long as $s \neq m$. When $t > n$, we print out the string using the function $\mathsf{PrintIt}()$. Pseudocode for this algorithm is shown in Figure 1. The transition function $\delta(s, j)$ is used to update the state s as described in Section 2.1. The initial call is $\mathsf{GenStr}(1{,}0)$.

Following this approach, each node in the computation tree will correspond to a unique string in $\mathbf{I}_k(j, f)$ where j ranges from 1 to n. Since the amount of computation at each node is constant, the total computation is proportional to

$$CompTree_k(n, f) = \sum_{j=1}^{n} I_k(j, f).$$

We can show that this sum is proportional to $I_k(n, f)$, which proves the following theorem. (Recall that the preprocessing required to set up the automata for f takes time $O(km)$. This amount is negligible compared to the size of the computation tree.)

```
procedure GenStr ( t, s : integer );
local j, q : integer;
begin
      if t > n then PrintIt()
      else begin
            for j ∈ {0, 1, ..., k − 1} do begin
                  a_t := j;
                  q := δ(s, j);
                  if q ≠ m then GenStr(t + 1, q);
end; end; end;
```

Fig. 1. An algorithm for generating k-ary strings with no substring equal to f.

Theorem 1. *The algorithm GenStr(t, s) for generating k-ary strings of length n with no substring equal to f is CAT.*

3.2 Generating k-ary circular strings

We now focus on the more complicated problem of generating circular strings with forbidden substring f. To solve this problem we use the previous algorithm, but now we must also check that the wraparound of the string does not yield the forbidden substring. More precisely, if $\alpha = a_1 a_2 \cdots a_n$ is in $\mathbf{I}_k(n, f)$ then the additional substrings we must implicitly test against the forbidden string f are $a_{n-m+1+i} \cdots a_n a_1 \cdots a_i$ for $i = 1, 2, \ldots, m-1$. To perform these additional tests, we could continue the pattern matching algorithm by appending the first $m-1$ characters to the end of the string. This will result in $m-1$ additional checks for each generated string, yielding an algorithm that runs in time $O(mI_k(n, f))$. This approach can be tweaked to yield a CAT algorithm for circular strings, but leads to difficulties in the analysis when applied in the necklace context.

If we wish to use the algorithm GenStr(t, s), we need another way to test the substrings starting in the last $m-1$ positions of α. We accomplish this feat by maintaining a new boolean data structure *match*(i, t) and dividing the work into two separate steps. In the first step we compare the substring $a_1 \cdots a_i$ against the last i characters in f. If they match, then the boolean value *match*(i, i) is set to TRUE; otherwise it is set to FALSE. In the second step, we check to see if $a_{n-m+1+i} \cdots a_n$ matches the first $m-i$ characters in f. If they match and *match*(i, i) is TRUE, then we reject the string. If there is no match for all $1 \leq i \leq m-1$, then α is in $\mathbf{C}_k(n, f)$.

To execute the first step, we start by initializing *match*$(i, 0)$ to TRUE for all i from 1 to $m-1$. We define *match*(i, t) for $1 \leq i \leq m-1$ and $i \leq t \leq m-1$ to be TRUE if *match*$(i, j-1)$ is TRUE and $a_t = f_{m-j+t}$. Otherwise *match*(i, t) is FALSE. This definition implies that *match*(i, i) will be TRUE if $a_1 \cdots a_i$ matches the last i characters of f. Pseudocode for a routine that sets these values for each t is shown in Figure 2. The procedure SetMatch(t) is called after the t-th character in the string α has been assigned for $t < m$. Thus, if $t < m$, we must

```
procedure SetMatch ( t : integer );
local i : integer;
begin
    for i ∈ {t, t+1, ..., m−1} do begin
        if match(i, t−1) and f_{m−i+t} = a_t then match(i, t) := TRUE;
        else match(i, t) := FALSE;
end; end;
```

Fig. 2. Procedure used to set the values of $match(i, t)$.

```
function CheckSuffix ( s : integer ) returns boolean;
begin
    while s > 0 do begin
        if match(m − s, m − s) then return(FALSE);
        else s := fail(s);
    end;
    return(TRUE);
end;
```

Fig. 3. Function used to test the wraparound of circular strings.

perform additional work proportional to $m - t$ for all strings in $\mathbf{I}_k(t, f)$. We will prove later that this extra work will not affect the asymptotic running time of the algorithm.

To execute the second step, we observe that after the nth character has been generated $(t = n+1)$, the string $a_{n-s+1} \cdots a_n$ is the longest suffix of α to match a prefix of f. Using the array *fail*, as described in Section 2.1, we can find all other suffixes that match a prefix of f in constant time per suffix. Then, for each suffix with length j found equal to a prefix of f, we check $match(m-j, m-j)$. If $match(m-j, m-j)$ is TRUE, then α is not in $\mathbf{C}_k(n, f)$. Note that $I_k(n-j, f)$ is an upper bound on the number of strings where $a_{n-j+1} \cdots a_n$ matches a prefix of f. Thus, for each $1 \leq j \leq m-1$ the extra work done is proportional to $I_k(n-j, f)$. Pseudocode for the tests required by this second step is shown in Figure 3. The function CheckSuffix(s) takes as input the parameter s which represents the length of the longest suffix of α to match a prefix of f. It returns TRUE if α is in $\mathbf{C}_k(n, f)$ and FALSE otherwise.

Following this approach, we can generate all circular strings with forbidden substrings by adding the routines SetMatch(t) and CheckSuffix(s) to GenStr(t, s). Pseudocode for the resulting algorithm is shown in Figure 4. The initial call is GenCirc(1,0).

Observe that the size of the computation tree will be the same as before; however, in this case, the computation at each node is not always constant. The

```
procedure GenCirc ( t, s : integer );
local j, q : integer;
begin
    if t > n then begin
        if CheckSuffix(s) then PrintIt();
    end else begin
        for j ∈ {0, 1, ..., k − 1} do begin
            a_t := j;
            q := δ(s, j);
            if t < m then SetMatch(t);
            if q ≠ m then GenCirc(t + 1, q);
end; end; end;
```

Fig. 4. An algorithm for generating k-ary circular strings with no substring equal to f.

extra work performed at these nodes is bounded by

$$ExtraWork_k(n, f) \;\leq\; \sum_{j=1}^{m-1} (m-j) I_k(j, f) + \sum_{j=1}^{m-1} I_k(n-j, f)$$

The first sum represents the work done by SetMatch and the second the work done by CheckSuffix. In Section 4.1 we show that this extra work is proportional to $I_k(n, f)$. In addition, we also prove that $C_k(n, f)$ is proportional to $I_k(n, f)$. These results prove the following theorem.

Theorem 2. *The algorithm GenCirc(t, s) for generating k-ary circular strings of length n with no substring equal to f is CAT.*

3.3 Generating k-ary necklaces

Using the ideas from the previous two algorithms, we now outline an algorithm to generate necklaces with forbidden necklace substrings. First, we embed the real-time automata based string matching algorithm into the necklace generation algorithm described in [4]. Then, since we must also test the wraparound for necklaces, we add the same tests as outlined in the circular string algorithm. Applying these two simple steps will yield an algorithm for necklace generation with no substring equal to the forbidden necklace f. Pseudocode for such an algorithm is shown in Figure 5. The additional parameter p in GenNeck(t, p, s), represents the length of the longest Lyndon prefix of the string being generated. Lyndon words can be generated by replacing the test "$n \bmod p = 0$" with the test "$n = p$." The initial call is GenNeck(1,1,0).

To analyze the running time of this algorithm, we again must show that the number of necklaces generated, $N_k(n, f)$, is proportional to the amount of

```
procedure GenNeck ( t, p, s : integer );
local j, q : integer;
begin
    if t > n then begin
        if n mod p = 0 and CheckSuffix(s) then PrintIt();
    end else begin
        a_t := a_{t-p};
        q := δ(s, a_t);
        if t < m then SetMatch(t);
        if q ≠ m then GenNeck(t + 1, p, q);
        for j ∈ {a_{t-p} + 1, ..., k - 1} do begin
            a_t := j;
            q := δ(s, j);
            if t < m then SetMatch(t);
            if q ≠ m then GenNeck(t + 1, t, q);
end; end; end;
```

Fig. 5. An algorithm for generating k-ary necklaces with no substring equal to f.

computation done. In this case the size of the computation tree is

$$NeckCompTree_k(n, f) = \sum_{j=1}^{n} P_k(j, f)$$

However, as with the circular string case, not all nodes perform a constant amount of work. The extra work performed by these nodes is bounded by

$$NeckExtraWork_k(n, f) \leq \sum_{j=1}^{m-1} (m - j) P_k(j, f) + \sum_{j=1}^{m-1} P_k(n - j, f)$$

Note that this expression is the same as the extra work in the circular string case, except we have replaced $I_k(n, f)$ with $P_k(n, f)$.

In Section 4.2 we show that $NeckCompTree_k(n, f)$ and $NeckExtraWork_k(n, f)$ are both proportional to $\frac{1}{n} I_k(n, f)$. In addition, we also prove that $N_k(n, f)$ is proportional to $\frac{1}{n} I_k(n, f)$. These results prove the following theorem.

Theorem 3. *The algorithm GenNeck(t, p, s) for generating k-ary necklaces of length n with no substring equal to f is CAT, so long as f is a Lyndon word.*

We remark that the algorithm works correctly even if f is not a Lyndon word and appears to be CAT for most strings f.

4 Analysis of the Algorithms

In this section we will state the results necessary to prove that the work done by each of the forbidden substring algorithms is proportional to the number of strings generated. The constants in the bounds derived in this section can be reduced with a more complicated analysis. The algorithms are very efficient in practice. Space limitations prevent us from giving proofs or analyzing the first algorithm, which has the simplest analysis. We do however, need one lemma from that analysis.

Lemma 1. *If* $|f| > 2$, *then* $\sum_{j=1}^{n} I_k(j, f) \leq 3I_k(n, f)$.

4.1 Circular strings

In the circular string algorithm, the size of the computation tree is the same as the previous algorithm, where it was shown to be proportional to $I_k(n, f)$. In this case, however, there is some extra work required to test the wrap-around of the string. Recall that this extra work is proportional to $ExtraWork_k(n, f)$ which is bounded as follows.

$$\begin{aligned} ExtraWork_k(n, f) &\leq \sum_{j=1}^{m-1} (m - j) I_k(j, f) + \sum_{j=1}^{m-1} I_k(n - j, f) \\ &\leq \sum_{j=1}^{n} \sum_{t=1}^{j} I_k(t, f) + \sum_{j=1}^{n} I_k(j, f). \end{aligned}$$

We now use Lemma **??** to simplify the above bound.

$$ExtraWork_k(n, f) \leq 3 \sum_{j=1}^{n} I_k(j, f) + \sum_{j=1}^{n} I_k(j, f) \;\leq\; 12 I_k(n, f).$$

We have now shown that the total work done by the circular string algorithm is proportional to $I_k(n, f)$. Since the total number of strings generated is $C_k(n, f)$, we must show that $C_k(n, f)$ is proportional to $I_k(n, f)$.

Theorem 4. *If* $|f| > 2$, *then* $3C_k(n, f) \geq I_k(n, f)$.

These lemmas and Theorem 4 prove Theorem 2.

4.2 Necklaces

To prove Theorem 3, we must show that the computation tree along with the extra work done by the necklace generation algorithm is proportional to $N_k(n, f)$. To get a good bound on $NeckCompTree_k(n, f)$ we need three additional lemmas; the bound of the first lemma does not necessarily hold if f is not a Lyndon word.

Lemma 2. *If* f *is Lyndon word where* $|f| > 2$, *then* $P_k(n, f) \leq \sum_{j=1}^{n} L_k(j, f)$.

Lemma 3. *If* $|f| > 2$, *then* $L_k(n, f) \leq \frac{1}{n} C_k(n, f)$.

Lemma 4. *If* $|f| > 2$, *then* $\sum_{j=1}^{n} \frac{1}{j} I_k(j, f) \leq \frac{8}{n} I_k(n, f)$.

Applying the previous three lemmas, we show that the computation tree is proportional to $\frac{1}{n}I_k(n,f)$.

$$\begin{aligned} NeckCompTree_k(n,f) &= \sum_{j=1}^{n} P_k(j,f) \;\le\; \sum_{j=1}^{n}\sum_{t=1}^{j} L_k(t,f) \\ &\le \sum_{j=1}^{n}\sum_{t=1}^{j} \frac{1}{t}C_k(t,f) \;\le\; \sum_{j=1}^{n}\sum_{t=1}^{j} \frac{1}{t}I_k(t,f) \\ &\le \sum_{j=1}^{n} \frac{12}{j}I_k(j,f) \;\le\; \frac{144}{n}I_k(n,f). \end{aligned}$$

This inequality is now used to show that the extra work done by the necklace algorithm is also proportional to $\frac{1}{n}I_k(n,f)$. Recall that the extra work is given by:

$$\begin{aligned} NeckExtraWork_k(n,f) &\le \sum_{j=1}^{m-1}(m-j)P_k(j,f) + \sum_{j=1}^{m-1} P_k(n-j,f) \\ &\le \sum_{j=1}^{n}\sum_{t=1}^{j} P_k(t,f) + \sum_{j=1}^{n} P_k(j,f). \end{aligned}$$

Further simplification of this bound follows from the bound on $NeckCompTree_k(n,f)$ along with Lemma 3.

$$\begin{aligned} NeckExtraWork_k(n,f) &\le 144\sum_{j=1}^{n}\frac{1}{j}I_k(j,f) + \frac{144}{n}I_k(n,f) \\ &\le \frac{12^3+12^2}{n}I_k(n,f). \end{aligned}$$

We have now shown that the total computation performed by the necklace generation algorithm is proportional to $\frac{1}{n}I_k(n,f)$. From equation (1), $N_k(n,f) \ge \frac{1}{n}C_k(n,f)$, and since $C_k(n,f) \ge \frac{1}{3}I_k(n,f)$, Theorem 3 is proved.

References

1. A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addisom-Wesley , 1974.
2. D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.
3. F. Ruskey, J. Sawada, An efficient algorithm for generating necklaces of fixed density, SIAM Journal on Computing, 29 (1999) 671-684.
4. F. Ruskey, J. Sawada, A fast algorithm to generate unlabeled necklaces, 11th Annual ACM-SIGACT Symposium on Discrete Algorithms (SODA), 2000, 256-262.
5. R.P. Stanley, *Enumerative Combinatorics, Volume I*, Wadsworth & Brooks/Cole, 1986.