A simple shift rule for k-ary de Bruijn sequences

Joe Sawada* Aaron Williams† Dennis Wong‡

September 6, 2016

Abstract

A k-ary de Bruijn sequence of order n is a cyclic sequence of length k^n in which each k-ary string of length n appears exactly once as a substring. A shift rule for a de Bruijn sequence of order n is a function that maps each length n substring to the next length n substring in the sequence. We present the first known shift rule for k-ary de Bruijn sequences that runs in $O(1)$-amortized time per symbol using $O(n)$ space. Our rule generalizes the authors' recent shift rule for the binary case (*A surprisingly simple de Bruijn sequence construction*, Discrete Mathematics 339, pages 127-131).

1 A new de Bruijn sequence construction

A *k-ary de Bruijn sequence* is a cyclic sequence of length k^n in which each k-ary string of length n appears exactly once as a substring. As an example, the cyclic sequence 111222333232221231211321331 3 is a 3-ary de Bruijn sequence for $n = 3$; the 27 unique length 3 substrings when considered cyclicly are:

111, 112, 122, 222, 223, 233, 333, 332, 323,
232, 322, 221, 212, 123, 231, 312, 121, 211,
113, 132, 321, 213, 133, 331, 313, 131, 311.

As illustrated in this example, a k-ary de Bruijn sequence of order n induces a very specific type of cyclic order of k-ary strings of length n: the length $n-1$ suffix of a given string is the same as the length $n-1$ prefix of the next string in the ordering.

The number of unique k-ary de Bruijn sequences for a given n and k is equal to $k!^{k^{n-1}}/k^n$ [3]; however, only a few efficient constructions are known. In particular, there are

- ▷ a Lyndon word concatenation algorithm by Fredricksen and Maiorana [11] that generates the lexicographically smallest de Bruijn sequence (also known as the Ford sequence),
- ▷ a block concatenation algorithm by Ralston [16],
- ▷ a lexicographic composition concatenation algorithm by Fredricksen and Kessler [10], and
- ▷ two different pure cycle concatenation algorithms by Fredricksen [8], and Etzion and Lempel [5].

*School of Computer Science, University of Guelph, Canada. Research supported by NSERC. email: jsawada@uoguelph.ca
†Division of Science, Mathematics, and Computing, Bard College at Simon's Rock, USA. email: haron@uvic.ca
‡School of Computer Science and Information Systems, Northwest Missouri State University, USA. email: cwong@uoguelph.ca

Each algorithm requires only $O(n)$ space and generates their de Bruijn sequences in $O(n)$ time per symbol, except the pure cycle concatenation algorithm by Etzion and Lempel which requires $O(n^2)$ space. The Lyndon word concatenation algorithm by Fredricksen and Maiorana achieves $O(1)$-amortized time per symbol. There also exist interesting greedy constructions including the "prefer-higher" approach by Martin [15] (and also Ford [6]), and a preference function approach by Alhakim [1]; however, they require $\Omega(k^n)$ space. The linear feedback shift register approach for binary de Bruijn sequences (see Golomb [12]) can also be generalized to larger alphabet sizes. However, this approach uses primitive polynomials over various finite fields, and there is no known efficient algorithm to find these polynomials in general (see Lidl and Niederreiter [14] and Rees [17]).

A *shift rule* for a de Bruijn sequence of order n is a function that maps each length n substring to the next length n substring in the sequence. A *necklace* is the lexicographically smallest string in an equivalence class of strings under rotation. In [20], the authors proved that the following shift rule f can be applied to produce a de Bruijn sequence for binary strings, where $\overline{b}$ denotes the complement of the bit b:

$$f(b_1 b_2 \cdots b_n) = \begin{cases} b_2 b_3 \cdots b_n \overline{b}_1 & \text{if } b_2 b_3 \cdots b_n 1 \text{ is a necklace;} \\ b_2 b_3 \cdots b_n b_1 & \text{otherwise.} \end{cases}$$

In this paper, we generalize this result to construct a novel k-ary de Bruijn sequence. We claim that the following shift rule f_k can be applied to produce a k-ary de Bruijn sequence:

$$f_k(a_1 a_2 \cdots a_n) = \begin{cases} 1^n & \text{if } a_1 a_2 \cdots a_n = k1^{n-1}; \\ a_2 a_3 \cdots a_n b & \text{if } a_1 = k \text{ and } a_1 a_2 \cdots a_n \neq k1^{n-1}; \\ a_2 a_3 \cdots a_n (a_1 + 1) & \text{if } a_1 \neq k \text{ and } a_2 a_3 \cdots a_n (a_1+1) \text{ is a necklace;} \\ a_2 a_3 \cdots a_n a_1 & \text{otherwise,} \end{cases}$$

where b is the largest positive integer such that $a_2 a_3 \cdots a_n b$ is not a necklace. As an example, successive applications of f_k for $n = 3$ and $k = 3$ starting with the string 111 produce the example listing shown earlier in this section. This leads to the following theorem, where $\mathbf{T}(n,k)$ denotes the set of k-ary strings with length n.

Theorem 1. *The shift rule f_k induces a cyclic ordering on $\mathbf{T}(n,k)$.*

Observe that concatenating the first bit of each string in the exhaustive listing produces a k-ary de Bruijn sequence. We denote this k-ary de Bruijn sequence by $\mathbf{dB}_k(n)$. Furthermore, by analyzing the shift rule f_k in more detail, we are able to generate the de Bruijn sequence in $O(1)$-amortized time per symbol.

De Bruijn sequences have been studied under many different names including memory wheels and universal cycles. Specific results have also been rediscovered numerous times. For example, Flye Sainte-Marie [19] counted the number of de Bruijn sequences before de Bruijn and van Aardenne-Ehrenfest [3], and Ford [6] constructed the lexicographically least de Bruijn sequence after Martin [15]. Furthermore, perhaps the best historical overview of the area is somewhat outdated (see Fredricksen [9]).

These factors make it difficult to make historical claims with complete certainty. However, to the best of the authors' knowledge, Theorem 1 represents the first shift rule for a k-ary de Bruijn sequence that can be implemented in $O(1)$-amortized time per symbol. In fact, the authors are only aware of two other de Bruijn sequence constructions that work for all orders n and is explicitly stated as a shift rule. These results are due to Fredricksen [7] (also see [9]) and Huang [13], and only apply for the binary alphabet. The authors are unaware of similar shift rules that generalize their results for larger alphabet sizes.

The rest of the paper is outlined as follows. In Section 2, we prove our main result Theorem 1 which leads to a new k-ary de Bruijn sequence construction. In Section 3, we present an algorithm that produces this new de Bruijn sequence in $O(1)$-amortized time per symbol.

The main results of this paper are also found in Wong's PhD thesis [21].

2 Proof of Theorem 1

The proof for Theorem 1 is done in two steps. First we show that the function f_k is a bijection. Then, we show that every string can be obtained from 1^n by repeatedly applying the function f_k.

Consider a k-ary string $\alpha = a_1a_2\cdots a_n$. A left rotation of α is $a_2a_3\cdots a_na_1$ and is denoted by LR(α). Let LR$^r(\alpha)$ denote the string that results from applying a left rotation r times to α. Thus LR$^r(\alpha) = a_{r+1}a_{r+2}\cdots a_na_1a_2\cdots a_r$ when $0 \le r < n$. The set of strings rotationally equivalent to α is denoted by $\mathbf{Rots}(\alpha)$, and the set of all k-ary necklaces of length n is denoted by $\mathbf{N}(n,k)$. We also say a string β is *reachable* from α if β can be obtained from α by repeatedly applying the function f_k.

We prove that f_k is bijective by showing that the following function f_k^{-1} is the inverse of f_k:

$$f_k^{-1}(\alpha) = \begin{cases} k1^{n-1} & \text{if } \alpha = 1^n; \\ ka_1a_2\cdots a_{n-1} & \text{if } a_1a_2\cdots a_{n-1}(a_n+1) \in \mathbf{N}(n,k) \text{ and } \alpha \text{ is not a necklace;} \\ (a_n-1)a_1a_2\cdots a_{n-1} & \text{if } \alpha \text{ is a necklace and } \alpha \neq 1^n; \\ a_na_1a_2\cdots a_{n-1} & \text{otherwise.} \end{cases}$$

Lemma 1. *The function f_k^{-1} is the inverse function of f_k.*

Proof. Let $\alpha = a_1a_2\cdots a_n \in \mathbf{T}(n,k)$. We prove that f_k^{-1} is the inverse function of f_k by showing that $f_k(f_k^{-1}(\alpha)) = \alpha$. When $\alpha = 1^n$, clearly $f_k(f_k^{-1}(1^n)) = 1^n$. We then consider the remaining three cases.

Case 1: $a_1a_2\cdots a_{n-1}(a_n+1)$ is a necklace in $\mathbf{N}(n,k)$ and α is not a necklace: By definition, $f_k^{-1}(\alpha) = ka_1a_2\cdots a_{n-1}$. Now observe that $f_k(f_k^{-1}(\alpha)) = a_1a_2\cdots a_{n-1}b$ where b is the largest positive integer such that $a_1a_2\cdots a_{n-1}b$ is not a necklace. Since $a_1a_2\cdots a_{n-1}(a_n+1)$ is a necklace in $\mathbf{N}(n,k)$ but $\alpha = a_1a_2\cdots a_n$ is not a necklace, $b = a_n$ and $f_k(f_k^{-1}(\alpha)) = \alpha$.

Case 2: α is a necklace and $\alpha \neq 1^n$: Since α is a necklace and $\alpha \neq 1^n$, $a_n \neq 1$. Observe that $f_k^{-1}(\alpha) = (a_n-1)a_1a_2\cdots a_{n-1}$. Thus, $f_k(f_k^{-1}(\alpha)) = a_1a_2\cdots a_{n-1}(a_n-1+1)) = \alpha$.

Case 3: Otherwise: Since α is not a necklace and $a_1a_2\cdots a_{n-1}(a_n+1)$ is not a necklace in $\mathbf{N}(n,k)$, $f_k^{-1}(\alpha) = a_na_1a_2\cdots a_{n-1}$ and $f_k(f_k^{-1}(\alpha)) = a_1a_2\cdots a_n = \alpha$.

Therefore, $f_k(f_k^{-1}(\alpha)) = \alpha$ and f_k^{-1} is the inverse function of f_k. $\square$

Corollary 1. *The function f_k is a bijection.*

Lemma 2. *Let $\alpha \in \mathbf{N}(n,k)$ and $\beta \in \mathbf{Rots}(\alpha)$. Then β is reachable from α.*

Proof. Let $\alpha = a_1a_2\cdots a_n$ and $q(\alpha) = kn - \sum_{i=1}^{n} a_i$. If $\alpha = 1^n$, then the only necklace is 1^n and the only string in $\mathbf{Rots}(\alpha)$ is 1^n. For the remaining of the proof, $q(\alpha) < kn - n$ and we apply strong induction on $q(\alpha)$ of α. In the base case, the only necklace is k^n when $q(\alpha) = 0$ and the only string in $\mathbf{Rots}(\alpha)$ is k^n. When $q(\alpha) = 1$, the only necklace is $(k-1)k^{n-1}$. By applying the function f_k $n+1$ times, we get all strings in $\mathbf{Rots}((k-1)k^{n-1})$ and k^n. Inductively, assume that for $\alpha \in \mathbf{N}(n,k)$ with $q(\alpha) \leq j$ where $0 \leq j < kn - n - 1$, each string $\beta \in \mathbf{Rots}(\alpha)$ is reachable from α. Now consider a necklace α with $q(\alpha) = j+1$. We show by induction that $\mathrm{LR}^{r+1}(\alpha)$ is reachable from $\mathrm{LR}^{r}(\alpha)$ where $r = \{0, 1, \ldots, n-2\}$.

> In the base case, $\mathrm{LR}^0(\alpha) = \alpha$ when $r = 0$ and it is reachable from α. Inductively, assume $\mathrm{LR}^t(\alpha)$ is reachable from α, where $0 \leq t < n-1$. Consider $\mathrm{LR}^{t+1}(\alpha) = b_1 b_2 \cdots b_n$ which obviously is not a necklace. If $b_1 b_2 \cdots b_{n-1}(b_n + 1)$ is not a necklace in $\mathbf{N}(n,k)$, then $f_k^{-1}(b_1 b_2 \cdots b_n) = b_n b_1 b_2 \cdots b_{n-1} = \mathrm{LR}^t(\alpha)$ since $\mathrm{LR}^{t+1}(\alpha)$ is not a necklace. Otherwise if $b_1 b_2 \cdots b_{n-1}(b_n+1)$ is a necklace in $\mathbf{N}(n,k)$, then $f_k^{-1}(b_1 b_2 \cdots b_n) = k b_1 b_2 \cdots b_{n-1}$. Observe that $q(k b_1 b_2 \cdots b_{n-1}) = j+1+b_n-k \leq j$ since $k \geq b_n+1$ because $b_1 b_2 \cdots b_{n-1}(b_n+1)$ is a necklace in $\mathbf{N}(n,k)$. In addition, $b_1 b_2 \cdots b_{n-1} k$ is the necklace representative of $k b_1 b_2 \cdots b_{n-1}$ since $b_1 b_2 \cdots b_{n-1}(b_n+1)$ is a necklace in $\mathbf{N}(n,k)$ and $k \geq b_n+1$. Therefore, $k b_1 b_2 \cdots b_{n-1}$ is reachable from its necklace representative $b_1 b_2 \cdots b_{n-1} k$ by the (external) inductive hypothesis. Now observe that $q((k-1) b_1 b_2 \cdots b_{n-1}) = j+2+b_n-k$, and $f_k^{-1}(b_1 b_2 \cdots b_{n-1} k) = (k-1) b_1 b_2 \cdots b_{n-1}$ since $b_1 b_2 \cdots b_{n-1} k \neq 1^n$ is a necklace. If $q((k-1) b_1 b_2 \cdots b_{n-1}) = j+1$, then $b_n = k-1$ and thus $(k-1) b_1 b_2 \cdots b_{n-1} = b_n b_1 b_2 \cdots b_{n-1} = \mathrm{LR}^t(\alpha)$. Otherwise, let $q((k-1) b_1 b_2 \cdots b_{n-1}) = j+2+b_n-k = h$ for some h such that $0 \leq h < j+1$, observe that $b_1 b_2 \cdots b_{n-1}(k-1)$ is a necklace since $b_1 b_2 \cdots b_{n-1}(b_n+1)$ is a necklace in $\mathbf{N}(n,k)$ and $k-1 \geq b_n+1$ because $h < j+1$. Thus, $f_k^{-1}(b_1 b_2 \cdots b_{n-1}(k-1)) = (k-2) b_1 b_2 \cdots b_{n-1}$. By repeatedly applying the (external) inductive hypothesis, $(k-1) b_1 b_2 \cdots b_{n-1}$ and $(k-2) b_1 b_2 \cdots b_{n-1}$ are reachable from $b_1 b_2 \cdots b_{n-1}(b_n+1)$, where $f_k^{-1}(b_1 b_2 \cdots b_{n-1}(b_n+1)) = b_n b_1 b_2 \cdots b_{n-1} = \mathrm{LR}^t(\alpha)$ since $b_1 b_2 \cdots b_{n-1}(b_n+1)$ is a necklace in $\mathbf{N}(n,k)$.

Since $\mathrm{LR}^{r+1}(\alpha)$ is reachable from $\mathrm{LR}^r(\alpha)$, each $\beta \in \mathbf{Rots}(\alpha)$ is reachable from α by transitivity. □

Lemma 3. *Each string $\beta \in \mathbf{T}(n,k)$ is reachable from 1^n.*

Proof. Let $\beta = b_1 b_2 \cdots b_n$ and $w(\beta) = (\sum_{i=1}^{n} b_i) - n$. Apply induction on $w(\beta)$ of β. In the base case, the only string with $w(\beta) = 0$ is 1^n which is reachable from 1^n. Inductively, assume any string β with $w(\beta) = t$ is reachable from 1^n, where $0 \leq t < kn - n$. Now consider a string β with $w(\beta) = t+1$, and assume $\beta \in \mathbf{Rots}(\alpha)$ where $\alpha = a_1 a_2 \cdots a_n$ is a necklace. Note that $a_n > 1$ since $\alpha \neq 1^n$ is a necklace. By Lemma 2, β is reachable from α. Observe that the string $\alpha' = f_k^{-1}(\alpha) = (a_n - 1) a_1 a_2 \cdots a_{n-1}$ since $\alpha \neq 1^n$ is a necklace. Thus, $w(\alpha') = t$ and by the inductive hypothesis, α' is reachable from 1^n. Thus, β is reachable from 1^n by transitivity. □

Together, Corollary 1 and Lemma 3 prove Theorem 1.

3 Generating the de Bruijn sequence efficiently

In this section we present algorithms to generate our k-ary de Bruijn sequence $\mathbf{dB}_k(n)$. First we show that f_k can be computed in $O(n)$ time. This immediate leads to a $O(n)$ time per symbol construction for the sequence. Then by studying the properties of the sequence, a slightly more sophisticated approach will generate the sequence in $O(1)$-amortized time per symbol.

A string is a *prenecklace* if it is a prefix of some necklace. A string α is said to be *periodic* if there exists some shorter string β such that $\alpha = \beta^t$ for some $t > 1$, where the exponent t denotes the number of repeated concatenations. A string that is not periodic is *aperiodic*. A *Lyndon word* is an aperiodic necklace.

It is well known that testing whether or not a string $\alpha = a_1a_2\cdots a_n$ is a prenecklace and finding the length of the longest prefix of α that is a Lyndon word can be done in $O(n)$ time. The algorithm can be easily derived from a standard necklace membership tester [4]. The function IsPRENECKLACE shown in Algorithm 1 determines whether or not the input string α is a prenecklace. If it is a prenecklace, it returns the value p corresponding to the length of the longest prefix of α that is a Lyndon word; otherwise it returns 0. Note that if $n \bmod p = 0$, then α is a necklace; otherwise if $p = n$, then α is a Lyndon word.

Algorithm 1 If $\alpha = a_1a_2\cdots a_n$ is a prenecklace, then this function returns the length of the longest prefix of α that is a Lyndon word; otherwise it returns 0.

```
function IsPrenecklace(a_1 a_2 ··· a_n)
    p ← 1
    for j from 2 to n do
        if a_j < a_{j−p} then return 0
        if a_j > a_{j−p} then p ← j
    return p
```

Lemma 4. *The function f_k can be computed in $O(n)$ time.*

Proof. Let $\alpha = a_1a_2\cdots a_n \in \mathbf{T}(n,k)$. If $a_1 \neq k$, then $f_k(\alpha)$ can be computed easily in $O(n)$ time by Algorithm 1. Otherwise if $a_1 = k$, then there are three subcases. If $a_2a_3\cdots a_n = 1^{n-1}$, then clearly $f_k(\alpha) = 1^n$ which can be computed in $O(n)$ time. Then if $a_2a_3\cdots a_n$ is not a prenecklace, $f_k(\alpha) = a_2a_3\cdots a_nk$ as appending any symbol after $a_2a_3\cdots a_n$ will not create a necklace. Otherwise if $a_2a_3\cdots a_n$ is a prenecklace, then let p be the length of the longest prefix of $a_2a_3\cdots a_n$ that is a Lyndon word. If $n \bmod p = 0$, then $a_2a_3\cdots a_na_{n-p}$ is a necklace while $a_2a_3\cdots a_n(a_{n-p}-1)$ is not a necklace, thus $f_k(\alpha) = a_2a_3\cdots a_n(a_{n-p}-1)$. Otherwise if $n \bmod p \neq 0$, then $a_2a_3\cdots a_n(a_{n-p}+1)$ is a necklace in $\mathbf{N}(n,k)$ while $a_2a_3\cdots a_na_{n-p}$ is not a necklace. Thus $f_k(\alpha) = a_2a_3\cdots a_na_{n-p}$. Since the value of p and the membership tester for prenecklaces can be computed in $O(n)$ time by Algorithm 1, f_k can be computed in $O(n)$ time. $\square$

Since f_k can be computed in $O(n)$ time, our k-ary de Bruijn sequence $\mathbf{dB}_k(n)$ can be generated in $O(n)$ time per symbol. To generate the sequence in $O(1)$-amortized time per symbol, we focus on the strings α such that $f_k(\alpha) \neq \mathrm{LR}(\alpha)$. By the definition of f_k, the strings that have such property are of the form $\alpha = a_1a_2\cdots a_n$ such that either $a_2a_3\cdots a_na_1$ is a necklace with $a_1 = k$, or $a_2a_3\cdots a_n(a_1+1)$ is a necklace in $\mathbf{N}(n,k)$ with $a_1 \neq k$.

In Table 1 we list the 3-ary strings of length 4 obtained by starting from 1111 and successively applying the function f_k for a total of $3^4 = 81$ times. Each row ends with a string β such that $f_k(\beta) \neq \mathrm{LR}(\beta)$. Hence when f_k is applied to this final string, it changes the final symbol after rotation. This means that the first string $\alpha = a_1a_2\cdots a_n$ in each row is a necklace, or $a_1a_2\cdots a_{n-1}(a_n+1)$ is a necklace in $\mathbf{N}(n,k)$. Observe there are 37 rows in this table, which is bounded by $2|\mathbf{N}(4,3)| = 2(24) = 48$. We will prove this observation for all n and k later in this section. In the third column of this table, the value $g_k(\alpha)$

i	$\alpha, f_k(\alpha), f_k(f_k(\alpha)), \ldots$	$g_k(\alpha)$	*symbols*
1	1111	1	1
2	1112	1	1
3	1122	1	1
4	1222	1	1
5	2222	1	2
6	2223	1	2
7	2233	1	2
8	2333	1	2
9	3333	1	3
10	3332, 3323, 3233	3	333
11	2332, 3322, 3223	3	233
12	2232	1	2
13	2323, 3232	2	23
14	2322, 3222	2	23
15	2221, 2212, 2122	3	222
16	1223, 2231, 2312, 3122	4	1223
17	1221, 2211, 2112	3	122
18	1123	1	1
19	1232, 2321, 3212, 2123	4	1232
20	1233, 2331, 3312, 3123	4	1233
21	1231, 2311, 3112	3	123
22	1121	1	1
23	1212, 2121	2	12
24	1213, 2131	2	12
25	1313, 3131	2	13
26	1312, 3121	2	13
27	1211, 2111	2	12
28	1113	1	1
29	1132	1	1
30	1322, 3221, 2213, 2132	4	1322
31	1323, 3231, 2313, 3132	4	1323
32	1321, 3211, 2113	3	132
33	1133	1	1
34	1332, 3321, 3213, 2133	4	1332
35	1333, 3331, 3313, 3133	4	1333
36	1331, 3311, 3113	3	133
37	1131, 1311, 3111	3	113

Table 1: The cyclic order of $\mathbf{T}(4,3)$ starting from 1111 induced by the function f_k. The rows break down the order based on when f_k applies an operation which is not a left rotation of the previous string in the listing. The value $g_k(\alpha)$ corresponds to the number of strings in each row, and the column *symbols* is the concatenation of the first symbol of the strings in each row. The underlined strings are of the form $\alpha = a_1a_2\cdots a_n$ such that α is not a necklace but $a_1a_2\cdots a_{n-1}(a_n+1)$ is a necklace in $\mathbf{N}(n,k)$. Concatenating the strings in the column *symbols* gives $\mathbf{dB}_3(4)$.

corresponds to the number of strings in each row. Let $f_k^j(\alpha)$ denote successively applying the function f_k on $\alpha = a_1a_2\cdots a_n$ for j times. More formally, $g_k(\alpha)$ is a function that computes the smallest value j such that $f_k^j(\alpha) \neq \mathrm{LR}^j(\alpha)$.

Still focusing on Table 1, note that the concatenation of the first symbol of each string in each row is highlighted in the final column. By concatenating all the strings together in this final column we obtain $\mathbf{dB}_3(4)$. Also observe that the strings in each row of Table 1 are obtained by repeatedly applying a left rotation starting from the initial string α. Therefore, if we show that g_k can be computed in $O(n)$ time, we can output the string in the final column in constant time per symbol.

Pseudocode of the implementation of FASTSHIFT is given in Algorithm 2. A complete C implementation of FASTSHIFT is given in the Appendix.

Algorithm 2 Optimized shift-based algorithm to generate $\mathbf{dB}_k(n)$ in $O(1)$-amortized time per symbol.

procedure FASTSHIFT
    $a_1a_2\cdots a_n \leftarrow 1^n$
    do
        $j \leftarrow g_k(a_1a_2\cdots a_n)$
        Print$(a_1a_2\cdots a_j)$
        $a_1a_2\cdots a_n \leftarrow f_k(a_ja_{j+1}\cdots a_na_1a_2\cdots a_{j-1})$
    while $a_1a_2\cdots a_n \neq 1^n$

3.1 Analysis

The function g_k can be computed by modifying Booth's algorithm [2]. Given a string $\alpha = a_1a_2\cdots a_n$, Booth's algorithm computes the smallest value t such that $a_ta_{t+1}\cdots a_na_1a_2\cdots a_{t-1}$ is the necklace representative of α with $t > 1$. The algorithm scans the string $\alpha\cdot\alpha = a_1a_2\cdots a_na_1a_2\cdots a_n = b_1b_2\cdots b_{2n}$ and maintains the variables j and t such that $b_tb_{t+1}\cdots b_j$ is a prenecklace. The algorithm also maintains a variable p which is the length of the longest prefix of $b_tb_{t+1}\cdots b_j$ that is a Lyndon word. If $p\lfloor\frac{j-t}{p}\rfloor = n$, then $b_tb_{t+1}\cdots b_j$ is a necklace and t is the smallest value such that $a_ta_{t+1}\cdots a_na_1a_2\cdots a_{t-1}$ is the necklace representative of α with $t > 1$. Booth's algorithm runs in $O(n)$ time.

To compute g_k, we modify Booth's algorithm to maintain the variables t and j such that $b_tb_{t+1}\cdots b_j$ or $b_tb_{t+1}\cdots b_{j-1}(b_j+1)$ is a prenecklace in $\mathbf{T}(n,k)$ with $t > 1$. Thus if $p\lfloor\frac{j-t}{p}\rfloor = n$, then t is the smallest value such that $a_ta_{t+1}\cdots a_na_1a_2\cdots a_{t-1}$ or $a_ta_{t+1}\cdots a_na_1a_2\cdots a_{t-2}(a_{t-1}+1)$ is a necklace in $\mathbf{N}(n,k)$ with $t > 1$. Thus, $g_k(\alpha) = t-1$. As an example, when $n = 8$, $k = 3$ and $\alpha = 12121213$, then $b_1b_2\cdots b_{2n} = 1212121312121213$. Now observe that when $t = 5$, $b_5b_6\cdots b_{12}(b_{13}+1) = 12131213$ is a prenecklace with $b_{13}+1 \leq 3$, and $p\lfloor\frac{j-t}{p}\rfloor = n$ with $p = 4$. Thus, $g(12121213) = t-1 = 4$. Clearly this modified Booth's algorithm also runs in $O(n)$ time.

Lemma 5. *The function g_k can be computed in $O(n)$ time.*

Pseudocode of the implementation of g_k is given in Algorithm 3. A C implementation of g_k can also be found in the C implementation of FASTSHIFT in the Appendix.

To analyze the runtime of FASTSHIFT, we need to consider how often the algorithm applies the function f_k. Recall that $\mathbf{N}(n,k)$ denotes the set of k-ary necklaces of length n; we use $N(n,k)$ to denote the size of this set. It is well known [12, 18] that

$$N(n,k) = \frac{1}{n}\sum_{d|n}\phi(d)\,k^{n/d} = \Theta(\frac{k^n}{n}),$$

Algorithm 3 Pseudocode of the function g_k.

function $g_k(a_1a_2\cdots a_n)$
  $b_1b_2\cdots b_{2n} \leftarrow a_1a_2\cdots a_na_1a_2\cdots a_n$
  $t \leftarrow 2; j \leftarrow 2; p \leftarrow 1$
  do
    $t \leftarrow t + p\lfloor \frac{j-t}{p} \rfloor$
    $j \leftarrow t + 1$
    $p \leftarrow 1$
    while $j \leq 2n$ **and** $b_{j-p} \leq b_j$ **do**
      if $b_{j-p} \leq b_j$ **then** $p \leftarrow j - t + 1$
      $j \leftarrow j + 1$
      if $j - t + 1 = n$ **and** $a_j < k$ **and** $(a_j + 1 > a_{j-p}$ **or** $(a_j + 1 = a_{j-p}$ **and** $n \bmod p = 0))$ **then**
        return $t - 1$
  while $p\lfloor \frac{j-t}{p} \rfloor < n$
  return $t - 1$

where ϕ is Euler's totient function.

Lemma 6. *The number of times* FASTSHIFT *applies the function f_k is bounded by $2N(n, k)$.*

Proof. The number of times FASTSHIFT applies the function f_k is equal to the number of strings of the form $\alpha = a_1a_2\cdots a_n$ such that α or $a_1a_2\cdots a_{n-1}(a_n + 1)$ is a necklace in $\mathbf{N}(n, k)$. We partition the set of strings of this form into two subsets. The first subset contains strings that are necklaces which clearly has the cardinality $N(n, k)$. The second subset contains strings of the form $\beta = b_1b_2\cdots b_n$ such that β is not a necklace while $b_1b_2\cdots b_{n-1}(b_n + 1)$ is a necklace in $\mathbf{N}(n, k)$. The cardinality of the second subset is clearly also bounded by $N(n, k)$. Hence, the number of times FASTSHIFT applies the function f_k is bounded by $2N(n, k)$. $\square$

Theorem 2. *The algorithm* FASTSHIFT *generates $\mathbf{dB}_k(n)$ in $O(1)$-amortized time per symbol.*

Proof. By Lemma 4 and Lemma 5, the functions f_k and g_k can be computed in $O(n)$ time. Thus, it is easy to see that each iteration of the **do/while** loop in Algorithm 2 requires $O(n)$ time. By Lemma 6, the number of times the function f_k is applied is bounded by $2N(n, k)$, thus there are $O(N(n, k))$ iterations of the **do/while** loop. Thus, the overall running time will be proportional to $O(nN(n, k)) = O(k^n)$. $\square$

4 Acknowledgement

The first author's research is supported by NSERC grant 400673.

References

[1] A. Alhakim. Spans of preference functions for de Bruijn sequences. *Discrete Appl. Math.*, 160(7-8):992–998, 2012.

[2] K. S. Booth. Lexicographically least circular substrings. *Inf. Proc. Letters*, 10(4/5):240–242, 1980.

[3] N. G. de Bruijn and T. Aardenne-Ehrenfest. Circuits and trees in oriented linear graphs. *Simon Stevin (Bull. Belgian Math. Soc.)*, 28:203–217, 1951.

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

[5] T. Etzion and A. Lempel. Algorithms for the generation of full-length shift-register sequences. *IEEE Trans. Inform. Theory*, 30(3):480–484, 1984.

[6] L. R. Ford. A cyclic arrangement of m-tuples. *Report No. P-1071, RAND Corp.*, 1957.

[7] H. Fredricksen. Generation of the Ford sequence of length 2^n, n large. *J. Comb. Theory (A)*, 12:153–154, 1972.

[8] H. Fredricksen. A class of nonlinear de Bruijn cycles. *J. Comb. Theory (A)*, 19(2):192–199, 1975.

[9] H. Fredricksen. A survey of full length nonlinear shift register cycle algorithms. *SIAM Rev.*, 24:195–221, 1982.

[10] H. Fredricksen and I. Kessler. Lexicographic compositions and de Bruijn sequences. *J. Comb. Theory (A)*, 22(1):17–30, 1977.

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

[12] S. W. Golomb. *Shift register sequences*. Aegean Park Press, 1982.

[13] Y. Huang. A new algorithm for the generation of binary de Bruijn sequences. *J. Algorithms*, 11(1):44–51, 1990.

[14] R. Lidl and H. Niederreiter. *Finite Fields*, volume 20 of *Encyclopedia Math. Appl.*. Cambridge University Press, 1997.

[15] M. H. Martin. A problem in arrangements. *Bull. Amer. Math. Soc.*, (40):859–864, 1934.

[16] A. Ralston. A new memoryless algorithm for de Bruijn sequences. *J. Algorithms*, 2(1):50–62, 1981.

[17] D. Rees. Notes on a paper by I. J. Good. *J. London Math. Soc.*, 21:169–172, 1946.

[18] J. Riordan. *An Introduction to Combinatorial Analysis*. Princeton University Press, 1980.

[19] C. F. Sainte-Marie. Solution to question nr. 48. *Intermédiaire des Mathématiciens*, 1:107–110, 1894.

[20] J. Sawada, A. Williams, and D. Wong. A surprisingly simple de Bruijn sequence construction. *Discrete Math.*, 339:127–131, 2016.

[21] D. Wong. *Novel universal cycle constructions for a variety of combinatorial objects*. PhD thesis, University of Guelph, Canada, 2015.

Appendix: C code to generate $\mathrm{dB}_k(n)$ in $O(1)$-amortized time per symbol

```
#include<stdio.h>
int n,k,a[50];
//-------------------------------------------------------------------------
int g_k(){
   int i,j=2,t=2,p=1;
   for (i=1; i<=n; i++) a[n+i] = a[i];
   do {
      t = t + p*((j-t)/p);
      j = t + 1;
      p = 1;
      while (j <= 2*n && a[j-p] <= a[j]) {
         if (a[j-p] < a[j]) p = j-t+1;
         j++;
         if (j-t+1 == n && a[j] < k && (a[j]+1 > a[j-p] || (a[j]+1 == a[j-p] && n%p == 0)))
              return t - 1;
      }
   } while (p*((j-t)/p) < n);
   return t - 1;
}
//-------------------------------------------------------------------------
void f_k() {
   int i,j,p=1;
   for (i=0; i<n; i++) a[i] = a[i+1];
   if (a[0] == k) {
      for (i=2; i<=n-1 && p; i++) {
         if (a[i-p] > a[i]) {
            a[n] = k;
            p = 0;
         }
         if (a[i-p] < a[i]) p = i;
      }
      if(a[n-p] == 1 && p == 1) a[n] = 1;
      else if (p && n%p) a[n] = a[n-p];
      else a[n] = a[n-p] - 1;
   }
   else a[n] = a[0] + 1;
}
//-------------------------------------------------------------------------
int Ones() {
   int i,j=0;
   for (i=1; i<=n; i++) if (a[i] == 1) j++;
   return j;
}
//-------------------------------------------------------------------------
// Generate a k-ary DB sequence in O(1) time per symbol
//-------------------------------------------------------------------------
void DB() {
   int i,j,b[50];
   for (i=1; i<=n; i++) a[i] = 1;
   do {
      j = g_k();
      for (i=1; i<=j; i++) printf("%d", a[i]);
      for (i=1; i<=n; i++) b[i] = a[i];
      for (i=1; i<=n-j+1; i++) a[i] = b[i+j-1];
      for (i=1; i<j; i++) a[n-j+1+i] = b[i];
      f_k();
   } while (Ones() < n);
}
//-------------------------------------------------------------------------
int main() {
   printf("Enter n: "); scanf("%d", &n);
   printf("Enter k: "); scanf("%d", &k);
   DB(); printf("\n");
}
```