Practical Algorithms to Rank Necklaces, Lyndon Words, and de Bruijn Sequences

Joe Sawada[1] and Aaron Williams[2]

[1] School of Computer Science, University of Guelph, Canada, `jsawada@uoguelph.ca`
[2] Science, Mathematics & Computing, Bard College at Simon's Rock, USA `awilliams@simons-rock.edu`

Abstract. We present practical algorithms for ranking k-ary necklaces and Lyndon words of length n. The algorithms are based on simple counting techniques. By repeatedly applying the ranking algorithms, both necklaces and Lyndon words can be efficiently unranked. Then, explicit details are given to rank and unrank the length n substrings of the lexicographically smallest de Bruijn sequence of order n.

1 Introduction

Given a list of distinct combinatorial objects $\alpha_1, \alpha_2, \ldots, \alpha_m$ the *rank* of α_i is i. In other words, the rank of an object is its position in the list. The process of determining the rank of an object in a specific listing is called *ranking*. The process of determining α_i for a given i is called *unranking*. For both problems, it is assumed that the actual objects and their listings are *not* stored in memory. Thus, the most naïve approach to ranking is to generate the listing and determine the position of the object in question. However, if the number of objects in a listing is exponential (as is the case for most combinatorial objects), then this approach yields an exponential time ranking algorithm in the worst case.

Example 1 The lexicographic listing of the $m = 14$ necklaces with length $n = 6$ over $\Sigma = \{a, b\}$:

1. aaaaaa	6. aabaab	11. ababbb
2. aaaaab	7. aababb	12. abbabb
3. aaaabb	8. aabbab	13. abbbbb
4. aaabab	9. aabbbb	14. bbbbbb.
5. aaabbb	10. ababab	

A *ranking* algorithm to determine the rank of the necklace ababbb in lexicographic order will return 11. An *unranking* algorithm to determine the necklace at rank 11 in lexicographic order will return ababbb.

The challenge of ranking combinatorial objects efficiently was first investigated by Lehmer [13] in 1964 when he studied permutations, combinations, compositions and partitions. Later in 1977, Wilf [23] presented a unified setting using directed graphs to list, rank and unrank a class of combinatorial objects and applied it to sets, partitions, permutations, and tableaux. The setting was later extended to subsets, and some restricted classes of partitions and permutations [24]. Around the same time Williamson [26] also provided a general model based on chain partitions for ranking and unranking combinatorial sets including set partitions and some restricted classes of permutations. Since then, efficient ranking and unranking algorithms have been developed separately for most basic combinatorial objects [8, 9, 14, 16–18, 22, 25]. For a more general survey of results, see [19].

For necklaces and Lyndon words, however, the existence of polynomial time ranking and unranking algorithms remained an open problem [15, 19] for many years. Then, at the *25th Annual Symposium on Combinatorial Pattern Matching* (CPM) in June 2014, Kociumaka, Radoszewski, and Rytter [11] presented an $O(n^3)$-time algorithm to rank Lyndon words. Three weeks later at the *41st International Colloquium on Automata, Languages, and Programming* (ICALP) in July 2014, Kopparty, Kumar, and Saks [12] independently presented a polynomial-time algorithm (with no tight bound) to rank necklaces. It is important to note that the former paper [11] applies a WORD-RAM model of computation in their analysis, even though it is not made explicitly clear in the proceedings. The main results of each paper are remarkably similar. They each rely on the construction of an automaton specific to the input string, and they both rank in lexicographic order. Neither result leads to a practical implementation.

1.1 Applications

Ranking and unranking algorithms for necklaces and Lyndon words have the following applications:

1. The problem of efficiently ranking the length n substrings of the lexicographically smallest de Bruijn sequence of order n was listed as an open problem in [19]. The crux of this problem is to rank Lyndon words. A solution to this problem is outlined in [11].
2. There is a well known correspondence between k-ary Lyndon words and irreducible polynomials over $\mathbb{GF}(k)$ [7]. Given a primitive polynomial, a unique irreducible polynomial can be generated for each Lyndon word of length n as outlined in [3] for $k = 2$. As a result, an efficient unranking algorithm for Lyndon words can be used to reverse-index irreducible polynomials as outlined in [12]. An efficient indexing algorithm for irreducible polynomials remains an open problem.
3. The ranking algorithms can be applied to compute the number of necklaces or Lyndon words with a given prefix. Details are presented later in this paper.
4. The unranking algorithms can be applied directly to randomly generate Lyndon words and necklaces. An average case algorithm to randomly generate Lyndon words runs in $O(n)$-time, as noted in [1].

1.2 New results

We provide a practical algorithm for ranking k-ary necklaces and Lyndon words of length n in lexicographic order using straightforward counting techniques. By repeatedly applying the ranking algorithms, both necklaces and Lyndon words can be efficiently unranked. An implementation in C is provided in the Appendix. It is important to note our analysis applies the standard unit-cost RAM model of computation, although there are mathematical operations (multiplication and addition) on integers of exponential size with respect to n. Thus, the results for Lyndon words are asymptotically equivalent to the ones obtained in [11] where a WORD-RAM model is assumed, although not explicitly stated.

Additionally, we provide explicit implementation-level details for ranking and unranking the lexicographically smallest de Bruijn sequence, expanding on the outline given in [11]. We also provide details for computing the number of necklaces and Lyndon words with a given prefix.

The enumeration framework outlined in this paper can also been extended to restricted classes of necklaces, including fixed-density necklaces as described in a follow up work [10].

1.3 Outline for remainder of paper

The rest of the paper is outlined as follows. In Section 2 we provide background definitions and notation along with some preliminary algorithmic results that will be used later in the paper. Then in Section 3 and 4, practical ranking and unranking algorithms for necklaces and Lyndon words are presented. In Section 5, these algorithms are applied to give explicit details for ranking and unranking the lexicographically smallest de Bruijn sequence for a given n. In Section 6 we describe how to compute the number of necklaces and Lyndon words with a given prefix. Section 7 presents two related open problems. Finally, a complete C implementation of all our algorithms is given in the Appendix.

2 Preliminaries

The strings considered in this paper are assumed to be over an alphabet Σ of size $k \geq 2$. A string $a_1a_2\cdots a_n$ is *lexicographically smaller* than $b_1b_2\cdots b_m$ if either

1. $a_1a_2\cdots a_{j-1} = b_1b_2\cdots b_{j-1}$ and $a_j < b_j$ for some $j \leq n$ and $j \leq m$, or
2. $a_1a_2\cdots a_n = b_1b_2\cdots b_n$ and $n < m$.

A *necklace* is the lexicographically smallest string in an equivalence class of strings under rotation. A string $\alpha = a_1a_2\cdots a_n$ is *periodic* (non-primitive) if there exists a string β such that $\alpha = \beta^j$ (where exponentiation denotes repetition) for some integer $j > 1$; otherwise α is *aperiodic* (primitive). A *Lyndon word* is an aperiodic necklace.

Example 2 The necklaces aaaaaa, aabaab, ababab, abbabb and bbbbbb from Example 1 are periodic and the remaining nine are Lyndon words.

Lemma 1. *Let $\alpha = a_1a_2\cdots a_n$ be a necklace and let $1 \leq i \leq j \leq n$. Then $a_ia_{i+1}\cdots a_{j-1}x$ is lexicographically larger than α if $x > a_j$.*

Proof. Consider $1 \leq i \leq j \leq n$. Since α is a necklace, any rotation satisfies $a_i\cdots a_na_1a_2\cdots a_{i-1} \geq \alpha$. Thus, $a_ia_{i+1}\cdots a_j \geq a_1a_2\cdots a_{j-i+1}$. Therefore if $x > a_j$, then $a_ia_{i+1}\cdots a_{j-1}x$ will be lexicographically larger than α. □

Given a string α, let its *necklace representative* (i.e., its lexicographically minimal rotation) be denoted $neck(\alpha)$. The number of strings in its equivalence class under rotation is equal to the length of the longest prefix of $neck(\alpha)$ that is a Lyndon word. Let this number be denoted by $lyn(\alpha)$. For example $lyn(\texttt{aaaaaa}) = 1$, $lyn(\texttt{ababab}) = 2$, and $lyn(\texttt{aaaaab}) = 6$.

Using standard techniques based on factoring a string into its Lyndon components [2,4], the length of the longest prefix of a string α that is a Lyndon word can be determined in $O(n)$ time. Such a function can be directly used to compute $lyn(\alpha)$. The basic idea is to maintain a value p corresponding to the length of the longest Lyndon prefix while performing a single scan of the string α. Using the same ideas, we can determine whether or not a string is a necklace in $O(n)$ time. Pseudocode for these two functions LYN(α) and ISNECKLACE(α) is given in Algorithm 1.

Algorithm 1 includes one more auxiliary function, LARGESTNECKLACE(α), that returns the largest necklace of length n that is smaller than or equal to $\alpha = a_1a_2\cdots a_n$. The function assumes an alphabet $\Sigma = \{1,2,\ldots,k\}$. It uses the fact[3] that if $lyn(\alpha) = p$ then for any string $\gamma < a_{p+1}a_{p+2}\cdots a_n$, $lyn(a_1a_2\cdots a_p\gamma)$ is also p. Thus, if α is not a necklace, $\beta = a_1a_2\cdots a_{p-1}(a_p-1)k^{n-p}$ is the largest possible string of length n smaller than α that might be a necklace. If it is a necklace, we are done. Otherwise, we repeatedly update β using this same procedure until a necklace is found. At the start of each iteration, β is not a necklace. Because appending a k to any Lyndon word of length $n > 1$ will yield a longer Lyndon word, it must be that $p = lyn(\beta)$ is strictly decreasing with each iteration. Observe that $lyn(\beta) \geq 1$ for any string β. Thus if $p = 1$, then observe that updated value of β must be xk^{n-1} for some $1 \leq x < k$ which is a Lyndon word. Therefore, the **while** loop in this function iterates at most n times which implies the function runs in $O(n^2)$ time.

[3] This fact can be observed by tracing the function LYN(α).

Algorithm 1 Auxiliary functions used by the upcoming ranking algorithm, assuming $n \geq 1$.

```
 1: function LYN($a_1a_2\cdots a_n$)
 2:     $p \leftarrow 1$
 3:     for $i$ from 2 to $n$ do
 4:         if $a_i > a_{i-p}$ then $p \leftarrow i$
 5:         if $a_i < a_{i-p}$ then return $p$
 6:     return $p$

 7: function ISNECKLACE($a_1a_2\cdots a_n$)
 8:     $p \leftarrow 1$
 9:     for $i$ from 2 to $n$ do
10:         if $a_i > a_{i-p}$ then $p \leftarrow i$
11:         if $a_i < a_{i-p}$ then return False
12:     return ($n$ mod $p = 0$)

13: function LARGESTNECKLACE($\alpha = a_1a_2\cdots a_n$)    ▷ Return the largest necklace $\beta$ of length $n$ smaller than or equal to $\alpha$
14:     $\beta = b_1b_2\cdots b_n = \alpha$
15:     while not ISNECKLACE($\beta$) do
16:         $p \leftarrow$ LYN($\beta$)
17:         $b_p \leftarrow b_p - 1$
18:         for $i$ from $p+1$ to $n$ do $b_i \leftarrow k$
19:     return $\beta$
```

2.1 A special set $\mathbf{B}_\alpha(t, j)$

In order to rank necklaces and Lyndon words efficiently, we need to efficiently determine the cardinality of a special set of strings that has both prefix and suffix restrictions. Let $\alpha = a_1a_2\cdots a_n$ be a necklace. Let $\mathbf{B}_\alpha(t, j)$ denote the set of strings of length $t \geq j$ with prefix $a_1a_2\cdots a_j$ such that every non-empty suffix is greater than α. Let the cardinality of $\mathbf{B}_\alpha(t, j)$ be denoted by $B_\alpha(t, j)$. Note that when $j = 0$, there is no prefix restriction on the strings.

Example 3 Let $\alpha = \texttt{aaabcc}$ and $\Sigma = \{\texttt{a,b,c}\}$. $\mathbf{B}_\alpha(2, 0) = \{\texttt{ab,ac,bb,bc,cb,cc}\}$ consists of strings of length 2 where every non-empty suffix is greater than α. The strings in $\{\texttt{aa,ba,ca}\}$ are not in this set because the suffix a is lexicographically smaller than α. Now consider $\mathbf{B}_\alpha(5, 2)$ partitioned on the 3rd (underlined) character:

$\texttt{aa}\underline{\texttt{a}}\,\texttt{cb}$	$\texttt{aa}\underline{\texttt{b}}\,\texttt{ab}$	$\texttt{aa}\underline{\texttt{c}}\,\texttt{ab}$
$\texttt{aa}\underline{\texttt{a}}\,\texttt{cc}$	$\texttt{aa}\underline{\texttt{b}}\,\texttt{ac}$	$\texttt{aa}\underline{\texttt{c}}\,\texttt{ac}$
	$\texttt{aa}\underline{\texttt{b}}\,\texttt{bb}$	$\texttt{aa}\underline{\texttt{c}}\,\texttt{bb}$
	$\texttt{aa}\underline{\texttt{b}}\,\texttt{bc}$	$\texttt{aa}\underline{\texttt{c}}\,\texttt{bc}$
	$\texttt{aa}\underline{\texttt{b}}\,\texttt{cb}$	$\texttt{aa}\underline{\texttt{c}}\,\texttt{cb}$
	$\texttt{aa}\underline{\texttt{b}}\,\texttt{cc}$	$\texttt{aa}\underline{\texttt{c}}\,\texttt{cc}$.

Observe how this partition decomposes the set revealing recursive structure:

$$\mathbf{B}_\alpha(5, 2) = \mathbf{B}_\alpha(5, 3) \,\cup\, \texttt{aab} \cdot \mathbf{B}_\alpha(2, 0) \,\cup\, \texttt{aac} \cdot \mathbf{B}_\alpha(2, 0).$$

An enumeration formula for $B_\alpha(t, j)$, where $0 \leq j \leq t \leq n$, can be derived based on the recursive structure illustrated in the previous example. For the formulation we assume an alphabet $\Sigma = \{1, 2, \ldots, k\}$. Considering the empty string, $B_\alpha(0, 0) = 1$. When $t > 0$, $B_\alpha(t, t) = 0$ because the suffix $a_1a_2\cdots a_t$ is lexicographically smaller than or equal to α. For $0 \leq j < t$, the strings in $\mathbf{B}_\alpha(t,j)$ can be partitioned based on the symbol in position $j{+}1$.

▷ If the $j{+}1$st symbol is smaller than a_{j+1} then the suffix starting from the first index would be smaller than α, a contradiction.

▷ If the $j{+}1$st symbol is a_{j+1} then the number of such strings is $B_\alpha(t, j+1)$.

▷ If the $j{+}1$st symbol is larger than a_{j+1}, then any suffix starting at index $1, 2, \ldots, j+1$ is larger than α by Lemma 1. Thus, for each of the $k - a_{j+1}$ choices for this $j{+}1$st symbol, the remaining $t-j-1$ positions can be filled recursively in $B_\alpha(t-j-1, 0)$ ways.

Therefore, for $0 \leq j \leq t \leq n$,

$$B_\alpha(t,j) = \begin{cases} 1 & \text{if } \ j = t \text{ and } t = 0, \\ 0 & \text{if } \ j = t \text{ and } t > 0, \\ B_\alpha(t, j+1) + (k - a_{j+1}) \cdot B_\alpha(t-j-1, 0) & \text{if } \ 0 \leq \ j \ < \ t. \end{cases}$$

An $O(n^2)$-time dynamic programming algorithm to compute these values is provided in Section 3.2.

3 Ranking necklaces and Lyndon words

Let $\mathbf{N}(\alpha)$ and $\mathbf{L}(\alpha)$ denote the set of necklaces and Lyndon words of length n, respectively, that are lexicographically smaller than or equal to α. Let $\mathit{RankNecklace}(\alpha)$ and $\mathit{RankLyndon}(\alpha)$ denote the cardinality of the sets $\mathbf{N}(\alpha)$ and $\mathbf{L}(\alpha)$, respectively. Given a string $\alpha = a_1a_2\cdots a_n$, let $\mathbf{T}(\alpha)$ denote the set of strings $w = w_1w_2\cdots w_n$ such that $\mathit{neck}(w) \leq \alpha$. In other words, $\mathbf{T}(\alpha)$ contains all length n strings whose necklace representatives are no larger than α in lexicographic order. The key step in the Lyndon word and necklace ranking algorithms from [11] and [12] is to determine the cardinality of $\mathbf{T}(\alpha)$, denoted by $T(\alpha)$. In this section we describe a simple and practical $O(n^2)$ implementation to perform this calculation assuming the unit-cost RAM model of computation.

Example 4 The set $\mathbf{T}(\texttt{aabbab})$ over $\Sigma = \{\texttt{a,b}\}$ of 40 strings grouped by rotational equivalence:

aaaaaa	aaaaab	aaaabb	aaabab	aaabbb	aabaab	aababb	aabbab
	baaaaa	baaaab	baaaba	baaabb	baabaa	baabab	baabba
	abaaaa	bbaaaa	abaaab	bbaaab	abaaba	bbaaba	abaabb
	aabaaa	abbaaa	babaaa	bbbaaa		abbaab	babaab
	aaabaa	aabbaa	ababaa	abbbaa		babbaa	bbabaa
	aaaaba	aaabba	aababa	aabbba		ababba	abbaba.

The first string in each column is a necklace. Observe that $\mathit{RankNecklace}(\texttt{aabbab}) = 8$ and $\mathit{RankLyndon}(\texttt{aabbab}) = 6$.

As outlined in [11, 12], standard enumerative techniques and Möbius inversion can be used to obtain the following formula for $\alpha = a_1a_2\cdots a_n$:

$$\mathit{RankLyndon}(\alpha) = \frac{1}{n}\sum_{d|n} \mu\left(\frac{n}{d}\right) T(a_1a_2\cdots a_d),$$

where $\mu(j)$ is the Möbius function. Applying Burnside's Lemma, we obtain

$$\mathit{RankNecklace}(\alpha) = \frac{1}{n}\sum_{d|n} \phi\left(\frac{n}{d}\right) T(a_1a_2\cdots a_d),$$

where $\phi(j)$ is Euler's totient function. This formula is described in [12] without explicitly being stated.

In the following subsection we demonstrate that $T(\alpha)$ can be computed in $O(n^2)$ time. Since for any real number $r > 1$ we have $\sum_{d|n} d^r = O(n^r)$ (a simple proof is given in [11]), we obtain the following theorem.

Theorem 1. *The rank of a necklace (Lyndon word) $\alpha = a_1a_2\cdots a_n$ in the lexicographic order of necklaces (Lyndon words) of length n can be determined in $O(n^2)$ time.*

3.1 Computing $T(\alpha)$

The approaches from [11, 12] to enumerate $T(\alpha)$ involve creating a finite automaton dependent on α. The best analysis yields an $O(n^3)$-time algorithm. In this subsection we present a practical $O(n^2)$-time algorithm to compute $T(\alpha)$ using simple counting methods. Observe that $T(\alpha) = T(\alpha')$ where α' is the lexicographically largest necklace (of the same length) that is smaller than or equal to α. Thus, in our upcoming formulae for $T(\alpha)$, we assume that α is a necklace.

To compute $T(\alpha)$ we partition the strings $\omega = w_1w_2\cdots w_n \in \mathbf{T}(\alpha)$ based on the smallest index t such that

$$w_tw_{t+1}\cdots w_nw_1w_2\cdots w_{t-1} \leq \alpha.$$

Each of these blocks is then further partitioned based on the largest integer $0 \leq j \leq n$ such that $a_1a_2\cdots a_j$ is a prefix of $w_tw_{t+1}\cdots w_nw_1w_2\cdots w_{t-1}$. This means the symbol x following this prefix must be smaller than a_{j+1}. Formally, let the strings in each such block be denoted by $\mathbf{A}_\alpha(t,j)$. Let the cardinality of $\mathbf{A}_\alpha(t,j)$ be denoted by $A_\alpha(t,j)$. Then

$$T(\alpha) = \sum_{t=1}^{n}\sum_{j=0}^{n} A_\alpha(t,j).$$

Example 5 Let $\alpha = \texttt{aabbab}$ and let $\Sigma = \{\texttt{a},\texttt{b}\}$. The 40 strings in $\mathbf{T}(\alpha)$ listed in Example 4 can be partitioned into subsets $\mathbf{A}_\alpha(t,j)$ for $0 \leq j \leq 6$ and $1 \leq t \leq 6$. When $j = 0, 1$ or 4, the set $\mathbf{A}_\alpha(t,j)$ is empty. The remaining subsets are illustrated in the table below. The substring corresponding to $a_1a_2\cdots a_j$ is underlined.

$\mathbf{A}_\alpha(t,j)$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$
	$\underline{\texttt{aa}}\,\texttt{aaaa}$	$\texttt{b}\,\underline{\texttt{aa}}\,\texttt{aaa}$	$\texttt{ab}\,\underline{\texttt{aa}}\,\texttt{aa}$	$\texttt{abb}\,\underline{\texttt{aa}}\,\texttt{a}$	$\texttt{abab}\,\underline{\texttt{aa}}$	$\underline{\texttt{a}}\,\texttt{abbb}\,\underline{\texttt{a}}$
	$\underline{\texttt{aa}}\,\texttt{aaab}$	$\texttt{b}\,\underline{\texttt{aa}}\,\texttt{aab}$	$\texttt{bb}\,\underline{\texttt{aa}}\,\texttt{aa}$	$\texttt{bab}\,\underline{\texttt{aa}}\,\texttt{a}$	$\texttt{abbb}\,\underline{\texttt{aa}}$	
	$\underline{\texttt{aa}}\,\texttt{aaba}$	$\texttt{b}\,\underline{\texttt{aa}}\,\texttt{aba}$	$\texttt{ab}\,\underline{\texttt{aa}}\,\texttt{ab}$	$\texttt{bbb}\,\underline{\texttt{aa}}\,\texttt{a}$		
$j=2$	$\underline{\texttt{aa}}\,\texttt{aabb}$	$\texttt{b}\,\underline{\texttt{aa}}\,\texttt{abb}$	$\texttt{bb}\,\underline{\texttt{aa}}\,\texttt{ab}$			
	$\underline{\texttt{aa}}\,\texttt{abaa}$					
	$\underline{\texttt{aa}}\,\texttt{abab}$					
	$\underline{\texttt{aa}}\,\texttt{abba}$					
	$\underline{\texttt{aa}}\,\texttt{abbb}$					
	$\underline{\texttt{aab}}\,\texttt{aaa}$	$\texttt{b}\,\underline{\texttt{aab}}\,\texttt{aa}$	$\texttt{ab}\,\underline{\texttt{aab}}\,\texttt{a}$	$\texttt{abb}\,\underline{\texttt{aab}}$	$\underline{\texttt{b}}\,\texttt{abb}\,\underline{\texttt{aa}}$	$\underline{\texttt{ab}}\,\texttt{abb}\,\underline{\texttt{a}}$
$j=3$	$\underline{\texttt{aab}}\,\texttt{aab}$	$\texttt{b}\,\underline{\texttt{aab}}\,\texttt{ab}$	$\texttt{bb}\,\underline{\texttt{aab}}\,\texttt{a}$			
	$\underline{\texttt{aab}}\,\texttt{aba}$					
	$\underline{\texttt{aab}}\,\texttt{abb}$					
$j=5$	$\underline{\texttt{aabba}}\,\texttt{a}$					
$j=6$	$\underline{\texttt{aabbab}}$	$\underline{\texttt{b}}\,\underline{\texttt{aabba}}$	$\underline{\texttt{ab}}\,\underline{\texttt{aabb}}$	$\underline{\texttt{bab}}\,\underline{\texttt{aab}}$	$\underline{\texttt{bbab}}\,\underline{\texttt{aa}}$	$\underline{\texttt{abbab}}\,\underline{\texttt{a}}$

The problem of ranking necklaces and Lyndon words has been reduced to computing $A_\alpha(t,j)$. In the special case where $j = n$ the strings in $\mathbf{A}_\alpha(1,n) \cup \mathbf{A}_\alpha(2,n) \cup \cdots \cup \mathbf{A}_\alpha(n,n)$ are precisely the rotations of α. The total number of such strings is $lyn(\alpha)$. Otherwise, for $j < n$, each set $\mathbf{A}_\alpha(t,j)$ falls into one of the following two cases depending on whether or not the symbol x following the substring $a_1a_2\cdots a_j$ in question is involved in the wraparound.

Case 1 ($t + j \le n$). Each $\omega \in \mathbf{A}_\alpha(t,j)$ is of the form $\sigma\, a_1a_2\cdots a_j\, x\, \tau$ where:
- ▷ $\sigma \in \Sigma^{t-1}$ such that every non-empty suffix is larger than α, [4]
- ▷ $x \in \Sigma$ is smaller than a_{j+1}, and
- ▷ $\tau \in \Sigma^{n-t-j}$ has no restrictions.

Recall from Section 2.1 that the number of possibilities for σ is given by $B_\alpha(t-1,0)$. Since x is smaller than a_{j+1}, there are $a_{j+1} - 1$ values it can have. For τ, there are trivially k^{n-t-j} possibilities. Thus, in this case,

$$A_\alpha(t,j) = B_\alpha(t-1,0) \cdot (a_{j+1} - 1) \cdot k^{n-t-j}.$$

Case 2 ($t + j > n$). Each $\omega \in \mathbf{A}_\alpha(t,j)$ is of the form $a_{n-t+2}\cdots a_{j-1}a_j\, x\, \sigma\, a_1a_2\cdots a_{n-t+1}$ where:
- ▷ $x \in \Sigma$ is smaller than a_{j+1},
- ▷ $\sigma \in \Sigma^{n-j-1}$, and
- ▷ $a_{n-t+2}\cdots a_{j-1}a_j\, x\, \sigma$ has every non-empty suffix larger than α.[5]

Let $\delta = a_{n-t+2}\cdots a_{j-1}a_j$. Since δ is a substring of the necklace α, any suffix of δ must be larger than or equal to the prefix of α with the same length (by the definition of a necklace). Therefore we determine the *longest* suffix of δ that is equal to the prefix of α with the same length. Suppose this suffix has length s which implies $a_{j-s+1}\cdots a_{j-1}a_j = a_1a_2\cdots a_s$. This means any suffix of ω starting from an index less than or equal to $|\delta| - s$ is larger than α. Consider three sub-cases depending on x.

- If $x < a_{s+1}$ then $a_{j-s+1}\cdots a_{j-1}a_jx$ is smaller than α, a contradiction.
- If $x = a_{s+1}$ then $a_1a_2\cdots a_{s+1}\sigma \in \mathbf{B}_\alpha(n-j+s, s+1)$.
- If $x > a_{s+1}$ then any suffix of ω starting from an index less than or equal to $|\delta| + 1$ will be larger than α by Lemma 1. Thus, $\sigma \in \mathbf{B}_\alpha(n-j-1,0)$. Since x must be smaller than a_{j+1}, there are $(a_{j+1} - a_{s+1} - 1)$ possible values for x, provided $a_{j+1} > a_{s+1}$.

Thus, in this case, if $a_{j+1} > a_{s+1}$,

$$A_\alpha(t,j) = B_\alpha(n-j+s, s+1) + (a_{j+1} - a_{s+1} - 1) \cdot B_\alpha(n-j-1,0).$$

Otherwise $A_\alpha(t,j) = 0$.

Example 6 Let $\alpha = a_1a_2\cdots a_9 = \texttt{aaabbaccb}$ and $\Sigma = \{\texttt{a},\texttt{b},\texttt{c}\}$. Consider $\mathbf{A}_\alpha(8,6)$. For this set we have the restriction that $x < a_7 = \texttt{c}$. The following illustrates this set partitioned by its 5th character x, with the substring $a_1a_2\cdots a_6$ underlined:

$$
\begin{array}{ll}
\underline{\texttt{abba}}\ \texttt{a}\ \texttt{ac}\ \underline{\texttt{aa}} & \underline{\texttt{abba}}\ \texttt{b}\ \texttt{ab}\ \underline{\texttt{aa}} \\
\underline{\texttt{abba}}\ \texttt{a}\ \texttt{bb}\ \underline{\texttt{aa}} & \underline{\texttt{abba}}\ \texttt{b}\ \texttt{ac}\ \underline{\texttt{aa}} \\
\underline{\texttt{abba}}\ \texttt{a}\ \texttt{bc}\ \underline{\texttt{aa}} & \underline{\texttt{abba}}\ \texttt{b}\ \texttt{bb}\ \underline{\texttt{aa}} \\
\underline{\texttt{abba}}\ \texttt{a}\ \texttt{cb}\ \underline{\texttt{aa}} & \underline{\texttt{abba}}\ \texttt{b}\ \texttt{bc}\ \underline{\texttt{aa}} \\
\underline{\texttt{abba}}\ \texttt{a}\ \texttt{cc}\ \underline{\texttt{aa}} & \underline{\texttt{abba}}\ \texttt{b}\ \texttt{cb}\ \underline{\texttt{aa}} \\
 & \underline{\texttt{abba}}\ \texttt{b}\ \texttt{cc}\ \underline{\texttt{aa}}.
\end{array}
$$

Note $s = 1$. Observe how this partition follows the formula outlined for this case:

[4] This follows from the definition of t noting that $t - 1 < n$.

[5] This follows from the definition of t noting that $|a_{n-t+2}\cdots a_{j-1}a_j\, x\, \sigma| < n$.

$$A_\alpha(8,6) = B_\alpha(4,2) \ + \ (1) \cdot B_\alpha(2,0).$$

3.2 An $O(n^2)$ algorithm to compute $T(\alpha)$

Given a necklace $\alpha = a_1a_2\cdots a_n$, let $suf_\alpha(i,j)$ denote the length of the longest suffix of $a_ia_{i+1}\cdots a_j$ that is equal to a prefix of α. Then applying the formulae presented in the previous subsection, the pseudocode in Algorithm 2 computes $T(\alpha)$ in $O(n^2)$ time. The algorithm precomputes the values $B_\alpha(t,j)$ using a standard dynamic programming approach. It also includes pre-computation of the values $suf_\alpha(i,j)$ for $2 \leq i \leq j \leq n$. Note that each value $suf_\alpha(i,j)$ can be computed with a single scan of the string $a_ia_{i+1}\cdots a_j$ applying Lemma 1.

Algorithm 2 Computing $T(\alpha)$ for a given necklace $\alpha = a_1a_2\cdots a_n$.

```
 1: function T($\alpha$)

 2:     ▷ Precompute $B_\alpha(t,j)$ using dynamic programming
 3:     $B_\alpha(0,0) \leftarrow 1$
 4:     for $t$ from 1 to $n$ do
 5:         $B_\alpha(t,t) \leftarrow 0$
 6:         for $j$ from $t-1$ down to 0 do   $B_\alpha(t,j) \leftarrow B_\alpha(t,j+1) + (k - a_{j+1}) \cdot B_\alpha(t-j-1,0)$

 7:     ▷ Precompute $suf_\alpha(i,j)$ for $2 \leq i \leq j \leq n$
 8:     for $i$ from 2 to $n$ do
 9:         $s \leftarrow i$
10:         for $j$ from $i$ to $n$ do
11:             if $a_j > a_{j-s+1}$ then  $s \leftarrow j+1$
12:             $suf_\alpha(i,j) \leftarrow j-s+1$

13:     ▷ Compute $T(\alpha)$
14:     $total \leftarrow$ LYN($\alpha$)
15:     for $t$ from 1 to $n$ do
16:         for $j$ from 0 to $n{-}1$ do
17:             if $j+t \leq n$ then  $total \leftarrow total + B_\alpha(t-1,0) \cdot (a_{j+1}-1) \cdot k^{n-t-j}$
18:             else
19:                 if $j < n{-}t{+}2$ then  $s \leftarrow 0$
20:                 else  $s \leftarrow suf_\alpha(n{-}t{+}2, j)$
21:                 if $a_{j+1} > a_{s+1}$ then  $total \leftarrow total + B_\alpha(n{-}j{+}s, s{+}1) + (a_{j+1} - a_{s+1} - 1) \cdot B_\alpha(n-j-1,0)$
22:     return $total$
```

Lemma 2. *Given a necklace $\alpha = a_1a_2\cdots a_n$, $T(\alpha)$ can be computed in $O(n^2)$ time.*

Recall that if α is not a necklace then $T(\alpha) = T(\alpha')$ where α' is the lexicographically largest necklace that is smaller than α. By applying the function LARGESTNECKLACE(α) outlined in Section 2, the necklace α' is obtained in $O(n^2)$ time.

Corollary 1. *Given a string $\alpha = a_1a_2\cdots a_n$, $T(\alpha)$ can be computed in $O(n^2)$ time.*

This result proves Theorem 1, which states that necklaces and Lyndon words can be ranked in $O(n^2)$ time.

4 Unranking necklaces and Lyndon words

The *unranking* problem for necklaces is to find the necklace $\alpha = a_1a_2\cdots a_n$ at a given rank r in the lexicographic ordering of necklaces of length n. Again, for algorithmic purposes, let $\Sigma = \{1, 2, \ldots, k\}$. Starting with the lexicographically largest necklace $\alpha = k^n$, the correct value for each index j (from left to right) can be determined by performing a binary search on the possible values $1, 2, \ldots, k$, initializing $min = 1$ and $max = k$. Observe that if $a_1a_2\cdots a_jxk^{n-j-1}$ is not a necklace, then $a_1a_2\cdots a_j\gamma$ is not a necklace for any $\gamma \in \Sigma^{n-j}$ such that $\gamma \leq xk^{n-j-1}$. At the start of each iteration of the binary search the following loop invariants are maintained:

1. $a_1a_2\cdots a_{j-1}$ is a prefix of the necklace at rank r,
2. the current string α with $a_j = max$ is a necklace with rank that is greater than or equal to r, and
3. replacing a_j with $min{-}1$ results in a string that is either not a necklace, or it is a necklace with rank less than r.

When testing the middle value $t = (min + max)/2$ and setting $a_j = t$, if the new string is a necklace and its rank is still greater than or equal to r, then max is updated to t. Otherwise, min is updated to $t + 1$.

Pseudocode for this procedure is given in Algorithm 3. A similar approach works for Lyndon words starting from the lexicographically largest Lyndon word $\alpha = (k{-}1)k^{n-1}$, using LYN(α) $= n$ instead of ISNECKLACE(α) and $RankLyndon(\alpha)$ instead of $RankNecklace(\alpha)$.

Algorithm 3 An $O(n^3 \log k)$ unranking algorithm for necklaces applying a binary search at each index j.

```
function UNRANK($n, r$)
    $\alpha = a_1a_2\cdots a_n \leftarrow k^n$
    for $j$ from $1$ to $n$ do
        $min \leftarrow 1$
        $max \leftarrow k$
        while $min < max$ do    ▷ binary search on the initial values $1, 2, \ldots, k$
            $prev \leftarrow a_j$
            $t \leftarrow (min + max)/2$
            $a_j \leftarrow t$
            if ISNECKLACE($\alpha$) and $RankNecklace(\alpha) \geq r$ then  $max \leftarrow t$
            else
                $a_j \leftarrow prev$
                $min \leftarrow t + 1$
    return $\alpha$
```

Since there are $O(\log k)$ calls made to $RankNecklace(\alpha)$ for each index j, UNRANK(n, r) runs in $O(n^3 \log k)$ time.

Theorem 2. *The necklace (Lyndon word) at rank r in the lexicographic order of necklaces (Lyndon words) of length n over an alphabet of size k can be determined in $O(n^3 \log k)$ time.*

5 Application: Ranking and unranking de Bruijn sequences

Consider an alphabet Σ of size k. A circular sequence of length k^n is called a *de Bruijn sequence* if it contains each string of length n as a substring exactly once. Amazingly, the lexicographically smallest de Bruijn sequence for a given n, denoted $\mathcal{DB}(n)$, can be constructed by concatenating together $L_1, L_2, L_3, \ldots, L_m$ which are the Lyndon words of length that divide n listed in lexicographic order. Thus,

$$\mathcal{DB}(n) = L_1 \cdot L_2 \cdot L_3 \cdots L_m.$$

Equivalently, if $N_1, N_2, N_3, \cdots, N_m$ are the necklaces of length n listed in lexicographic order, where $ap(N_i)$ denotes the longest Lyndon prefix of N_i, then $ap(N_i) = L_i$ and

$$\mathcal{DB}(n) = ap(N_1) \cdot ap(N_2) \cdot ap(N_3) \cdots ap(N_m).$$

This construction was presented by Fredricksen, Kessler, and Maiorana [5, 6] and it can be generated in $O(1)$ time per symbol [20].

The *rank* of $\omega = w_1 w_2 \cdots w_n$ in $\mathcal{DB}(n) = d_1 d_2 d_3 \cdots d_{k^n}$ is the index r such that the substring of length n starting at position r is ω. We denote this rank by $RankDB(w_1 w_2 \cdots w_n)$. The *unranking* problem for de Bruijn sequences is to find the substring of length n starting at index r in the sequence $\mathcal{DB}(n)$.

Example 7 From Example 1, there are 14 necklaces of length 6 over $\Sigma = \{a, b\}$. The concatenation of their aperiodic prefixes yields $\mathcal{DB}(6) =$

```
a aaaaab aaaabb aaabab aaabbb aab aababb aabbab aabbbb ab ababbb abb abbbbb b.
```

It has length $2^6 = 64$ and when considered circularly it contains each length 6 string as a substring exactly once. The rank of the underlined string is $RankDB(\texttt{aabaaa}) = 5$. Since a de Bruijn sequence is considered to be circular, note that $RankDB(\texttt{baaaaa}) = 64$.

The problem of finding efficient algorithms to rank and unrank the strings in $\mathcal{DB}(n)$ was stated as an open problem in [19]. A polynomial-time solution that applies the ranking of Lyndon words, was given in [11]. Their algorithms have running times of $O(n^3)$ for ranking and $O(n^4 \log k)$ unranking respectively using the WORD-RAM model. In the following two subsections, we expand on the outline of their algorithm, providing clear implementation ready descriptions of these algorithms using the alphabet $\Sigma = \{1, 2, \ldots, k\}$. A complete C implementation is given in the Appendix.

5.1 Ranking de Bruijn sequences

In this subsection we provide algorithmic details to compute the rank of $\omega = w_1 w_2 \cdots w_n$ in $\mathcal{DB}(n)$. The description expands on Theorem 23 from [11] which is based on the proof of Theorem 3.4 from [5]. Recall the algorithm to construct $\mathcal{DB}(n)$ depends on the lexicographic ordering of necklaces $N_1, N_2, N_3, \ldots, N_m$ or Lyndon words $L_1, L_2, L_3, \ldots, L_m$. The ranking algorithm is partitioned into three cases:

1. ω is found at the end of $\mathcal{DB}(n)$ ($\omega = N_m$) or in the wrap-around,
2. ω is a necklace N_i where $i \in \{1, 2, \ldots, m-1\}$,
3. ω is not a necklace and not found in the wrap-around.

Case 1: Observe that $\mathcal{DB}(n)$ begins with 1^n and ends with k^n. Thus, if $\omega = k^t 1^{n-t}$ for $0 < t \leq n$, then its rank is $k^n - t + 1$.

Case 2: If ω is a necklace N_i, where $1 \le i < m$, then either $N_i = L_i$, or N_i is periodic. In the latter case, N_i is a prefix of $L_i \cdot L_{i+1}$ [5]. Thus, the rank of ω is given by:

$$\begin{aligned} \mathit{RankDB}(\omega) &= 1 + \sum_{j=1}^{i-1} |L_i| \\ &= 1 - |L_i| + \sum_{j=1}^{i} |L_i| \\ &= 1 - \mathit{lyn}(\omega) + \sum_{d|n} d \cdot \mathit{RankLyndon}(w_1 w_2 \cdots w_d) \\ &= 1 - \mathit{lyn}(\omega) + T(n, w). \end{aligned}$$

Case 3: If ω is not a necklace or found in the wraparound of $\mathcal{DB}(n)$, then let $N_i = \mathit{neck}(\omega)$ where s is the smallest offset such that $N_i = w_{s+1}w_{s+2}\cdots w_n w_1 w_2 \cdots w_s$. There are three sub-cases [5]:

(a) If $w_1 w_2 \cdots w_s \ne k^s$, then ω is a substring of $L_i \cdot L_{i+1}$ and $\mathit{RankDB}(\omega) = \mathit{RankDB}(N_i) + \mathit{lyn}(N_i) - s$.
(b) If $w_1 w_2 \cdots w_s = k^s$ and N_i is periodic ($\mathit{lyn}(N_i) < n$), then ω is a substring of $L_{i-1} \cdot L_i \cdot L_{i+1}$ and $\mathit{RankDB}(\omega) = \mathit{RankDB}(N_i) - s$.
(c) If $w_1 w_2 \cdots w_s = k^s$ and N_i is aperiodic ($\mathit{lyn}(N_i) = n$), then let N_j be the largest necklace that is less than $w_{s+1}w_{s+2}\cdots w_n 1^s$. Then ω is a substring of $L_{j-1} \cdot L_j \cdot L_{j+1}$ and $\mathit{RankDB}(\omega) = \mathit{RankDB}(N_j) + \mathit{lyn}(N_j) - s$.

The pseudocode in Algorithm 4 summarizes these three cases computing the rank of ω in $\mathcal{DB}(n)$. It includes a simple $O(n^2)$ time method for computing the necklace of a string at the start of Case 3. In fact, this can be done in $O(n)$ time [2]. As mentioned earlier, for any real number $r > 1$ we have $\sum_{d|n} d^r = O(n^r)$. Thus, the time required in Case 2 is $O(n^2)$.

Theorem 3. *The rank of $w_1 w_2 \cdots w_n$ in $\mathcal{DB}(n)$ can be determined in $O(n^2)$ time.*

Algorithm 4 Computing $\mathit{RankDB}(\omega)$ for a given $\omega = w_1 w_2 \cdots w_n$.

```
1:  function RANKDB($\omega = w_1 w_2 \cdots w_n$)

2:      ▷ Case 1
3:      if $\omega = k^t 1^{n-t}$ then  return $k^n - t + 1$

4:      ▷ Case 2
5:      if ISNECKLACE($\omega$) then  return $1 -$ LYN($\omega$) $+$ T($\omega$)

6:      ▷ Case 3
7:      $s \leftarrow 0$
8:      $\alpha \leftarrow \omega$
9:      while not ISNECKLACE($\alpha$) do
10:         $s \leftarrow s + 1$
11:         $\alpha \leftarrow w_{s+1}w_{s+2}\cdots w_n w_1 w_2 \cdots w_s$

12:     if $w_1 w_2 \cdots w_s \ne k^s$ then  return RANKDB($\alpha$) + LYN($\alpha$) $-s$
13:     if LYN($\alpha$) $< n$ then  return RANKDB($\alpha$) $-s$
14:     $\beta \leftarrow$ LARGESTNECKLACE($w_{s+1}w_{s+2}\cdots w_n 1^s$)
15:     return RANKDB($\beta$) + LYN($\beta$) $-s$
```

5.2 Unranking de Bruijn sequences

To unrank a de Bruijn sequence $\mathcal{DB}(n)$ for a given rank r, we consider the same three cases described in previous subsection on ranking. If $d = k^n - r + 1 \leq n$, then the string at rank r is $k^d 1^{n-d}$ and it is found at the end of $\mathcal{DB}(n)$ or in the wraparound (Case 1). Otherwise, let $\alpha = a_1 a_2 \cdots a_n$ be the lexicographically smallest necklace with $r' = \mathit{RankDB}(\alpha) \geq r$. By the construction of $\mathcal{DB}(n)$, $d = r' - r < n$. As mentioned in the previous subsection, the string starting at index r' in $\mathcal{DB}(n)$ is α, even if α is periodic. Thus, if $d = 0$, then the string starting at rank r is the necklace α (Case 2). Observe that the strings of rank $r = 1$ and $r = 2$ in $\mathcal{DB}(n)$ are the necklaces 1^n and $1^{n-1}2$, respectively. Thus, if $d > 0$, the necklace α is at least the third one in lexicographic order. Let $\beta = b_1 b_2 \cdots b_n$ be the necklace before α in lexicographic order, and let $\delta = c_1 c_2 \cdots c_n$ be the necklace before β in lexicographic order. If $p = \mathit{lyn}(\beta)$ and $p \geq d$, then the string at rank r is $b_{n-d+1} b_{n-d+2} \cdots b_n a_1 a_2 \cdots a_{n-d}$; otherwise, the string at rank r is $c_{n-d+p+1} c_{n-d+p+2} \cdots c_n b_1 b_2 \cdots b_p a_1 a_2 \cdots a_{n-d}$.

Algorithm 5 An $O(n^3 \log k)$ unranking algorithm for $\mathcal{DB}(n)$.

```
1: function UNRANKDB($n, r$)

2:     ▷ Case 1: The string is found at the end of $\mathcal{DB}(n)$ or in the wraparound
3:     $d \leftarrow k^n - r + 1$
4:     if $d \leq n$ then  return $k^d 1^{n-d}$

5:     ▷ Apply a binary search to find the smallest necklace $\alpha$ with rank greater than or equal to $r$
6:     $\alpha = a_1 a_2 \cdots a_n \leftarrow k^n$
7:     for $j$ from 1 to $n$ do
8:         $min \leftarrow 1$
9:         $max \leftarrow k$
10:        while $min < max$ do
11:            $prev \leftarrow a_j$
12:            $t \leftarrow (min + max)/2$
13:            $a_j \leftarrow t$
14:            if ISNECKLACE($\alpha$) and $\mathit{RankDB}(\alpha) \geq r$ then  $max \leftarrow t$
15:            else
16:                $a_j \leftarrow prev$
17:                $min \leftarrow t + 1$

18:    ▷ Case 2: $\alpha$ is a necklace
19:    $d \leftarrow \mathit{RankDB}(\alpha) - r$
20:    if $d = 0$ then  return $\alpha$

21:    ▷ Case 3
22:    $\beta = b_1 b_2 \cdots b_n \leftarrow$ LARGESTNECKLACE($a_1 a_2 \cdots a_{n-1}(a_n - 1)$)
23:    $\delta = c_1 c_2 \cdots c_n \leftarrow$ LARGESTNECKLACE($b_1 b_2 \cdots b_{n-1}(b_n - 1)$)
24:    $p \leftarrow$ LYN($\beta$)
25:    if $p \geq d$ then  return $b_{n-d+1} b_{n-d+2} \cdots b_n a_1 a_2 \cdots a_{n-d}$
26:    else  return $c_{n-d+p+1} c_{n-d+p+2} \cdots c_n b_1 b_2 \cdots b_p a_1 a_2 \cdots a_{n-d}$
```

Pseudocode for unranking $\mathcal{DB}(n)$ is given in Algorithm 5. By applying a binary search strategy, similar to the one that was employed to unrank necklaces, the necklace α can be found in $O(n^3 \log k)$ time. The necklace β can be found in $O(n^2)$ time using the function LARGESTNECKLACE(α') where α' is obtained from α by decrementing the value of a_n. The necklace δ can similarly be found in $O(n^2)$ time.

Theorem 4. *The string $w_1 w_2 \cdots w_n$ with rank r in $\mathcal{DB}(n)$ over an alphabet of size k can be determined in $O(n^3 \log k)$ time.*

6 Computing the number of necklaces and Lyndon words with a given prefix

Let $\alpha = a_1a_2 \cdots a_j$ for $1 \leq j \leq n$ over $\Sigma = \{1, 2, \ldots, k\}$. Let $N_n(\alpha)$ denote the number of necklaces of length n with prefix α and let $L_n(\alpha)$ denote the number of Lyndon words of length n with prefix α. In this section we describe a polynomial time algorithm to compute $N_n(\alpha)$ and $L_n(\alpha)$.

We begin by considering two special cases. The first special case is when $j = n$. If α is a necklace then $N_n(\alpha) = 1$; otherwise $N_n(\alpha) = 0$. Similarly, if α is a Lyndon word then $L_n(\alpha) = 1$; otherwise $L_n(\alpha) = 0$. This case can easily be resolved in $O(n)$ time using the functions IsNecklace(α) and Lyn(α). For the second special case, suppose $\alpha = 1^j$ for $1 \leq j < n$. In this case there will be no necklace or Lyndon word with a prefix smaller than α. The largest possible length n string with prefix α is αk^{n-j} which is clearly a Lyndon word. Thus $N_n(\alpha) = \mathit{RankNecklace}(\alpha k^{n-j})$ and $L_n(\alpha) = \mathit{RankLyndon}(\alpha k^{n-j})$.

Now assume that $j < n$ and $\alpha \neq 1^j$. The largest string with α as a prefix is αk^{n-j}. The smallest string with α as a prefix is $\alpha 1^{n-j}$, and clearly $\alpha 1^{n-j}$ is not a necklace. Thus:

$$N_n(\alpha) = \mathit{RankNecklace}(\textsc{LargestNecklace}(\alpha k^{n-j})) - \mathit{RankNecklace}(\textsc{LargestNecklace}(\alpha \mathit{1}^{n-j})).$$

Similarly, for Lyndon words we have

$$L_n(\alpha) = \mathit{RankLyndon}(\textsc{LargestLyndon}(\alpha k^{n-j})) - \mathit{RankLyndon}(\textsc{LargestLyndon}(\alpha \mathit{1}^{n-j})).$$

The function LargestLyndon(α) returns the lexicographically largest Lyndon word of length $|\alpha| = n$ that is less than or equal to α. This function can be obtained by modifying LargestNecklace(α) shown in Algorithm 1, replacing line 15 with: **while** Lyn(β) $\neq n$ **do**.

Theorem 5. *$N_n(\alpha)$ and $L_n(\alpha)$ can be computed in $O(n^2)$ time.*

7 Conclusions and future work

We have provided a re-interpretation of the ranking algorithm for Lyndon words given in [11]. Our approach applies straightforward counting techniques that allow for a more practical implementation. Using this same framework, it is possible to efficiently rank and unrank other restricted classes of necklaces and Lyndon words. In particular, ranking algorithms for fixed-density necklaces have recently been developed [10], where the *density* of a binary string is the number of 1s in the string.

A *bracelet* is the lexicographically smallest string in an equivalence class of strings under both rotation and reversal. For example, while `aababb` and `aabbab` are both necklaces, only `aababb` is a bracelet; a reversed rotation of `aabbab` yields `aababb`. They can be listed in lexicographic order in constant amortized time [21].

Open problem #1: Is there a polynomial-time algorithm to rank/unrank bracelets, and if so, can it be done in $O(n^3)$-time or better?

An *unlabeled* necklace is the lexicographically smallest string in an equivalence class of strings under both rotation and permutation of the alphabet symbols. For example, while `aaaabb` and `aabbbb` are both necklaces, only `aaaabb` is an unlabeled necklace; permuting `a` and `b` in `aabbbb` yields `bbaaaa` which is a rotation of `aaaabb`. They can be listed in lexicographic order in constant amortized time [3].

Open problem #2: Is there a polynomial-time algorithm to rank/unrank unlabeled necklaces, and if so, can it be done in $O(n^3)$-time or better? What if the alphabet is binary?

8 Acknowledgements

We would like to thank the anonymous reviewers who have helped improve the accuracy and presentation of this paper. Joe Sawada is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) grant RGPIN 400673-2012.

References

1. F. Bassino, J. Clément, and C. Nicaud. The standard factorization of Lyndon words: an average point of view. *Discrete Mathematics*, 290(1):1–25, 2005.
2. K. S. Booth. Lexicographically least circular substrings. *Information Processing Letters*, 10(4/5):240–242, 1980.
3. K. Cattell, F. Ruskey, J. Sawada, M. Serra, and C. Miers. Fast algorithms to generate necklaces, unlabeled necklaces, and irreducible polynomials over GF(2). *Journal of Algorithms*, 37(2):267–282, 2000.
4. J. P. Duval. Factorizing words over an ordered alphabet. *Journal of Algorithms*, 4(4):363–381, 1983.
5. H. Fredricksen and I. Kessler. Lexicographic compositions and de Bruijn sequences. *Journal of Combinatorial Theory, Series A*, 22(1):17 – 30, 1977.
6. H. Fredricksen and J. Maiorana. Necklaces of beads in k colors and k-ary de Bruijn sequences. *Discrete Mathematics*, 23:207–210, 1978.
7. S. W. Golomb. Irreducible polynomials, synchronizing codes, primitive necklaces and cyclotomic algebra. In *Conference on Combinatorial Mathematics and Its Applications*, pages 358–370. 1969.
8. U. I. Gupta, D. T. Lee, and C. K. Wong. Ranking and unranking of 2-3 trees. *SIAM Journal on Computing*, 11:582–590, 1982.
9. U. I. Gupta, D. T. Lee, and C. K. Wong. Ranking and unranking of B-trees. *Journal of Algorithms*, 4:51–60, 1983.
10. P. Hartman and J. Sawada. Ranking fixed-density necklaces and Lyndon words. *manuscript*, 2016.
11. T. Kociumaka, J. Radoszewski, and W. Rytter. Computing k-th Lyndon word and decoding lexicographically minimal de Bruijn sequence. In A. Kulikov, S. Kuznetsov, and P. Pevzner, editors, *Combinatorial Pattern Matching*, volume 8486 of *Lecture Notes in Computer Science*, pages 202–211. Springer International Publishing, 2014.
12. S. Kopparty, M. Kumar, and M. Saks. Efficient indexing of necklaces and irreducible polynomials over finite fields. In J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsoupias, editors, *Automata, Languages, and Programming*, volume 8572 of *Lecture Notes in Computer Science*, pages 726–737. Springer Berlin Heidelberg, 2014.
13. D. H. Lehmer. The machine tools of combinatorics. In E. Beckenbach, editor, *Applied Combinatorial Mathematics*, pages 5–31. John Wiley and Sons, 1964.
14. L. Li. Ranking and unranking of AVL trees. *SIAM Journal on Computing*, 15:1025–1035, 1986.
15. C. Martínez and X. Molinero. An efficient generic algorithm for the generation of unlabelled cycles. In *Mathematics and Computer Science III*, Trends in Mathematics, pages 187–197. Birkhäuser Verlag Basel, 2004.
16. W. Myrvold and F. Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79:281–284, 2001.
17. J. Pallo. Enumerating, ranking, and unranking binary trees. *The Computer Journal*, 29:171–175, 1986.
18. E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice Hall College Div, 1977.
19. F. Ruskey. *Combinatorial Generation*. Working version (1i) edition, 1996.
20. F. Ruskey, C. Savage, and T. M. Y. Wang. Generating necklaces. *Journal of Algorithms*, 13:414–430, 1992.
21. J. Sawada. Generating bracelets in constant amortized time. *SIAM Journal of Computing*, 31(1):259–268, 2001.
22. A. E. Trojanowski. Ranking and listing algorithms for k-ary trees. *SIAM Journal on Computing*, 7:492–509, 1978.
23. H. S. Wilf. A unified setting for sequencing, ranking, and selection algorithms for combinatorial objects. *Advances in Mathematics*, 24:281–291, 1977.
24. H. S. Wilf. A unified setting for selection algorithms (II). *Annals of Discrete Mathematics*, 2:135–148, 1978.
25. S. G. Williamson. Ranking algorithms for lists of partitions. *SIAM Journal on Computing*, 5:602–617, 1976.
26. S. G. Williamson. On the ordering, ranking, and random generation of basic combinatorial sets. In *Combinatoire et Représentation du Groupe Symétrique*, volume 579 of *Lecture Notes in Mathematics*, pages 187–193. Springer-Verlag, 1977.

Appendix - C code

```c
//=================================================================================
// Ranking and unranking algorithms for k-ary necklaces, Lyndon words and de Bruijn
// sequences.
//=================================================================================
#include <stdio.h>
#define MAX 63 // n=62 is largest feasible for long long integers when k=2

int mu[MAX] = { 0,1,-1,-1,0,-1,1,-1,0,0,1,-1,0,-1,1,1,0,-1,0,-1,0,
             1,1,-1,0,0,1,0,0,-1,-1,-1,0,1,1,1,0,-1,1,1,0,-1,
            -1,-1,0,0,1,-1,0,0,0,1,0,-1,0,1,0,1,1,-1,0,-1,1};

int phi[MAX] = { 0,1,1,2,2,4,2,6,4,6,4,10,4,12,6,8,8,16,6,18,8,12,
            10,22,8,20,12,18,12,28,8,30,16,20,16,24,12,36,18,
            24,16,40,12,42,20,24,22,46,16,42,20,32,24,52,18,
            40,24,36,28,58,16,60,30};

int k, NECKLACE=0, LYNDON=0, DB=0;

long long int power[MAX];

//=================================================================================
// Find the longest prefix of w[1..n] that is a Lyndon word
//=================================================================================
int Lyn(int n, int w[]) {
   int i,p=1;

   for (i=2; i<=n; i++) {
      if (w[i] < w[i-p]) return p;
      if (w[i] > w[i-p]) p = i;
   }
   return p;
}
//=================================================================================
// Return whether or not w[1..n] is a necklace
//=================================================================================
int IsNecklace(int n, int w[]) {
   int i,p=1;

   for (i=2; i<=n; i++) {
      if (w[i] < w[i-p]) return 0;
      if (w[i] > w[i-p]) p=i;
   }
   if (n%p == 0) return 1;
   return 0;
}
//=================================================================================
// Compute the largest necklace neck[1..n] <= w[1..n]
//=================================================================================
void LargestNecklace(int n, int w[], int neck[]) {
   int i,p;

   for (i=1; i<=n; i++) neck[i] = w[i];
   while (!IsNecklace(n,neck)) {
      p = Lyn(n,neck);
      neck[p]--;
      for (i=p+1; i<=n; i++) neck[i] = k;
   }
}
//=================================================================================
// Compute the largest Lyndon word neck[1..n] <= w[1..n]
//=================================================================================
void LargestLyndon(int n, int w[], int neck[]) {
   int i,p;

   for (i=1; i<=n; i++) neck[i] = w[i];
   while (Lyn(n,neck) != n) {
      p = Lyn(n,neck);
      neck[p]--;
      for (i=p+1; i<=n; i++) neck[i] = k;
   }
}
//================================================================================
// Return the number of strings whose necklace is <= w[1..n]
//================================================================================
long long int T(int n, int w[]) {
   int i,j,t,s,neck[MAX],suf[MAX][MAX];
   long long int tot, B[MAX][MAX];

   // Sets neck[1..n] to the largest necklace less than or equal to w[1..n]
   LargestNecklace(n, w, neck);

   // Compute B[t][j] = number of strings of length t with prefix neck[1..j] but
   // no suffix less than neck[1..n]
   B[0][0] = 1;
   for (t=1; t<=n; t++) {
      B[t][t] = 0;
      for (j=t-1; j>=0; j--) B[t][j] = B[t][j+1] + ((k - neck[j+1]) * B[t-j-1][0]);
   }

   // Compute suf[i][j] = longest suffix of neck[i..j] that is a prefix of neck[1..n]
   for (i=2; i<=n; i++) {
      s = i;
      for (j=i; j<=n; j++) {
         if (neck[j] > neck[j-s+1]) s = j+1;
         suf[i][j] = j-s+1;
      }
   }

   // Compute T(...)
   tot = Lyn(n,neck);
   for (t=1; t<=n; t++) {
      for (j=0; j<n; j++) {
         if (j+t <= n) tot += B[t-1][0] * (neck[j+1]-1) * power[n-t-j];
         else {
            if (j < n-t+2) s = 0;
            else s = suf[n-t+2][j];
            if (neck[j+1] > neck[s+1]) tot += B[n-j+s][s+1] + (neck[j+1]-neck[s+1]-1) * B[n-j-1][0];
         }
      }
   }
   return(tot);
}
//================================================================================
// Return the rank of w[1..n] - a valid Lyndon word or necklace - in lex order
//================================================================================
long long int Rank(int n, int w[]) {
   int i;
   long long int r=0;

   for (i=1; i<=n; i++) {
      if (n%i == 0) {
         if (NECKLACE) r += phi[n/i] * T(i,w);
         if (LYNDON) r += mu[n/i] * T(i,w);
      }
   }
   return(r/n);
}
//================================================================================
// Return the necklace or Lyndon w[1..n] for a given valid rank in lex order
//================================================================================
void UnRank(int n, long long int r, int w[]) {
   int j,min,max,prev,t;

   // Start with necklace or Lyndon word with largest rank
   for (j=1; j<=n; j++) w[j] = k;
   if (LYNDON) w[1] = k-1;

   // Determine character w[j] from left to right using a binary search
   for (j=1; j<=n; j++) {
      min = 1; max = k;
      while (min < max) {
         prev = w[j];
         t = (min+max)/2;
         w[j] = t;
         if (NECKLACE && IsNecklace(n,w) && Rank(n,w) >= r) max = t;
         else if (LYNDON && Lyn(n,w) == n && Rank(n,w) >= r) max = t;
         else {
            w[j] = prev;
            min = t+1;
         }
      }
   }
}
//================================================================================
// Return the rank of w[1..n] in the lexicographically smallest de Bruijn sequence
//================================================================================
long long int RankDB(int n, int w[]) {
   int i,t,j,s=0,neck[MAX],prev[MAX];

   // w[1..n] = k^t 1^{n-t} for t >= 1, which includes all wraparounds
   t=0; while (w[t+1] == k && t+1 <=n) t++;
   j=t; while (w[j+1] == 1 && j+1 <=n) j++;
   if (t >= 1 && j == n) return(power[n]-t+1);

   // w[1..n] is a necklace
   if (IsNecklace(n,w)) return(1 - Lyn(n,w) + T(n,w));

   // Find the necklace neck[1..n] of w[1..n] and its offset
   for (i=1; i<=n; i++) neck[i] = w[i];
   while (!IsNecklace(n,neck)) {
      s++;
      for (i=1; i<=n; i++) {
         if (i+s <= n) neck[i] = w[i+s];
         else neck[i] = w[i+s-n];
      }
   }

   if (s != t) return (RankDB(n,neck) + Lyn(n,neck) - s);
   if (Lyn(n,neck) < n) return (RankDB(n,neck) - s);

   for (i=n-s+1; i<=n; i++) neck[i] = 1;
   LargestNecklace(n,neck,prev);
   return (RankDB(n,prev) + Lyn(n,prev) - s);
}
//================================================================================
// Return the string starting at index r in the lex smallest de Bruijn sequence
//================================================================================
void UnRankDB(int n, long long int r, int w[]) {
   int i,j,min,max,last,t,p,neck[MAX],prev[MAX],prev2[MAX];
   long long int d;

   // Special case for strings in the wraparound
   d = power[n] - r+1;
   if (d < n) {
      for (j=1; j<= d; j++) w[j] = k;
      for (j=d+1; j<=n; j++) w[j] = 1;
      return;
   }
   //----------------------------------------------------------------------------
   // Find the smallest necklace neck[1..n] whose rank is greater than or equal to r
   for (j=1; j<=n; j++) neck[j] = k;
```

```
    // Determine character neck[j] from left to right using a binary search
    for (j=1; j<=n; j++) {
       min = 1; max = k;
       while (min < max) {
          last = neck[j];
          t = (min+max)/2;
          neck[j] = t;
          if (IsNecklace(n,neck) && RankDB(n,neck) >= r) max = t;
          else {
             neck[j] = last;
             min = t+1;
          }
       }
    }
    //-----------------------------------------------------------------------------------
    d = RankDB(n,neck) - r;
    if (d == 0) {
       for (i=1; i<=n; i++) w[i] = neck[i];
    }
    else {
       // Get the previous 2 necklaces prev and prev2 in lex order
       neck[n]--;
       LargestNecklace(n,neck,prev);
       neck[n]++;

       prev[n]--;
       LargestNecklace(n,prev,prev2);
       prev[n]++;

       p = Lyn(n,prev);
       j = 1;
       if (p >= d) {
          for (i=1; i<=d; i++) w[j++] = prev[n-d+i];
       }
       else {
          for (i=1; i<=d-p; i++) w[j++] = prev2[n-(d-p)+i];
          for (i=1; i<=p; i++) w[j++] = prev[i];
       }
       for (i=1; i<=n-d; i++) w[j++] = neck[i];
    }
}
//====================================================================================
// Compute the number of necklaces (or Lyndon words) with a given prefix by computing
// the rank of the largest necklace with the given prefix (r2) and subtracting the
// rank of the largest necklace with a smaller prefix (r1).
//====================================================================================
long long int CountGivenPrefix(int pre[], int n, int j) {
    int i,p,w[MAX];
    long long int r1,r2;

    if (j == n) {
       if (LYNDON && Lyn(n,pre) == n) return 1;
       if (!LYNDON && IsNecklace(n,pre)) return 1;
       return 0;
    }

    for (i=j+1; i<=n; i++) pre[i] = 1;
    if (LYNDON) LargestLyndon(n,pre,w);
    else LargestNecklace(n,pre,w);
    r1 = Rank(n,w);

    for (i=j+1; i<=n; i++) pre[i] = k;
    if (LYNDON) LargestLyndon(n,pre,w);
    else LargestNecklace(n,pre,w);
    r2 = Rank(n,w);

    // Check if prefix is of form 111...11
    i = 1;
    while (pre[i] == 1 && i <=j) i++;
    if (i == j && pre[1] == 1) return r2;
    else return r2 - r1;

}
//==================================================================================
int main() {
    int i,j,n,option,w[MAX],pre[MAX];
    long long int r;

    printf("1 = Rank Necklaces \n");
    printf("2 = Rank Lyndon Words \n");
    printf("3 = Rank de Bruijn Sequence \n\n");
    printf("4 = UnRank Necklaces \n");
    printf("5 = UnRank Lyndon Words \n");
    printf("6 = UnRank de Bruijn Sequence \n\n");
    printf("7 = Count Necklaces with a given prefix \n");
    printf("8 = Count Lyndon Words with a given prefix \n\n");

    printf("Select option #: "); scanf("%d", &option);
    printf("Enter n k: "); scanf("%d %d", &n, &k);

    // PRECOMPUTE power[i] = k^i
    power[0] = 1;
    for (i=1; i<=n; i++) power[i] = power[i-1] * k;

    if (option == 1 || option == 4 || option == 7) NECKLACE = 1;
    if (option == 2 || option == 5 || option == 8) LYNDON = 1;
    if (option == 3 || option == 6) DB = LYNDON = 1;

    if (option == 1 || option == 2 || option == 3) {
       printf("Enter string w[1..n] over alphabet {1,2,...,k}\n");
       for (i=1; i<=n; i++) {
          printf(" w[%d] = ", i);
          scanf("%d", &w[i]);
       }
       if (option == 3) printf("\nRank = %lld\n", RankDB(n,w));
       else printf("\nRank = %lld\n", Rank(n,w));
    }
    if (option == 4 || option == 5 || option == 6) {
       printf("Enter rank: "); scanf("%lld", &r);
       if (option == 6) UnRankDB(n,r,w);
       else UnRank(n,r,w);

       for (i=1; i<=n; i++) printf("%d", w[i]);
       printf("\n");
    }
    if (option == 7 || option == 8) {
       printf("Enter prefix length 1 < j < n: "); scanf("%d", &j);
       printf("Enter prefix over alphabet {1,2,...,k}\n");
       for (i=1; i<=j; i++) {
          printf(" p[%d] = ", i);
          scanf("%d", &pre[i]);
       }
       printf("Count = %lld\n", CountGivenPrefix(pre,n,j));
    }
}
```