Ranking and Unranking Fixed-Density Necklaces and Lyndon Words

Patrick Hartman[1] and Joe Sawada[2]

[1] School of Computer Science, University of Guelph, Canada,
[2] School of Computer Science, University of Guelph, Canada, `jsawada@uoguelph.ca`

Abstract. We present the first polynomial-time ranking and unranking algorithms for fixed-density necklaces and Lyndon words. Using the unit-cost RAM model, the ranking algorithm runs in $O(n^3)$ time and the unranking algorithm runs in $O(n^4)$ time. By applying the ranking algorithms, the number of fixed-density necklaces and Lyndon words with a given prefix can also be computed in $O(n^3)$ time.

1 Introduction

Given a list of distinct combinatorial objects $\alpha_1, \alpha_2, \ldots, \alpha_m$ the *rank* of α_i is i. 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 discovery of polynomial-time ranking and unranking algorithms is a fundamental pursuit within combinatorics. Indeed, for most elementary combinatorial objects, there exist exhaustive listings for which efficient ranking and unranking algorithms are known [6]. Only recently, however, were such algorithms discovered for listings of necklaces [5] and Lyndon words [4] of length n. The results were discovered independently and they each apply a similar strategy that requires the construction of a finite automaton to enumerate a related object. This enumeration step was simplified in [9] using a partition-based approach, giving rise to a more practical implementation.

The main result of this paper is the first polynomial-time ranking and unranking algorithms for binary necklaces and Lyndon words of length n and a fixed-density (the number of 1s) d. The algorithms we present will apply to these objects when they are listed in lexicographic order. The algorithms follow the same strategy used in both [4, 5], but applying the partition-based approach from [9].

Example 1 The lexicographic listing of the 14 binary necklaces with length $n = 9$ and density $d = 4$:

1. `000001111`	6. `000101011`	11. `001001011`
2. `000010111`	7. `000101101`	12. `001001101`
3. `000011011`	8. `000110011`	13. `001010011`
4. `000011101`	9. `000110101`	14. `001010101`.
5. `000100111`	10. `000111001`	

A *ranking* algorithm to determine the rank of `001001011` will return 11. An *unranking* algorithm to determine the necklace at rank 11 will return `001001011`.

Previously, several papers focussed on the efficient generation of fixed-density necklaces and Lyndon words but without any related ranking or unranking results [7, 8, 10, 11]. Enumeration results with respect to fixed-density necklaces and Lyndon words were studied in [2].

In the next subsection we provide the formal definitions of our primary objects, fixed-density necklaces and Lyndon words, along with some related notation. In Section 2 we present a practical $O(n^3)$-time algorithm to rank fixed-density necklaces and Lyndon words. Through repeated application of the ranking algorithm, an $O(n^4)$-time algorithm to unrank fixed-density necklaces and Lyndon words is presented in Section 3. In Section 4 an $O(n^3)$-time algorithm is presented to count the number of fixed-density necklaces or Lyndon words with a given prefix. A complete C implementation of the algorithms is available in the appendix. For all algorithms, the unit-cost RAM model is assumed. For a word-RAM implementation, the reader is encouraged to consider the techniques used in [4] to potentially obtain a more efficient algorithm than the one presented in this paper.

1.1 Background and Notation

All strings in this paper are considered to be binary. A string $a_1a_2\cdots a_n$ is *lexicographically less* than a string $b_1b_2\cdots b_m$ 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 $j \leq m$ where $j \geq 1$, or
2. $a_1a_2\cdots a_n = b_1b_2\cdots b_n$ and $n < m$.

A *necklace* is defined as the lexicographically smallest string in an equivalence class of strings under rotation. The *necklace representative* of a string α, denoted $neck(\alpha)$, is its lexicographically smallest rotation. Since every rotation of a necklace α is greater than or equal to α, we obtain the following remark that will be used later.

Remark 1. If $\alpha = a_1a_2\cdots a_n$ is a necklace where $a_j = 0$ for some $1 \leq j \leq n$, then $a_ia_{i+1}\cdots a_{j-1}1 > \alpha$ for $1 \leq i \leq j$.

The following property of necklaces will also be applied later, in Section 2.1.

Lemma 1. *Let $\alpha = a_1a_2\cdots a_n$ be a necklace where $a_j = 1$ for some $1 \leq j \leq n$ and let $\beta = b_1b_2\cdots b_ia_1a_2\cdots a_{j-1}0$ where $b_1b_2\cdots b_i \leq a_1a_2\cdots a_i$ for some $i < n$. If $i+j \leq n$ then $\beta < \alpha$ and if $i+j > n$ then the length n prefix of β is less than or equal to α.*

Proof. Suppose $i+j \leq n$. If $b_1b_2\cdots b_i < a_1a_2\cdots a_i$ then the result is trivially true. If $b_1b_2\cdots b_i = a_1a_2\cdots a_i$ then because α is a necklace $a_1a_2\cdots a_j \leq a_{i+1}a_{i+2}\cdots a_{j+i}$. Thus, $a_1a_2\cdots a_{j-1}0 < a_{i+1}a_{i+2}\cdots a_{j+i}$ and hence $\beta < \alpha$. If $i+j > n$ then from the previous arguments either $\beta < \alpha$ or α is a prefix of β. Together this implies that the length n prefix of β is less than or equal to α.

A string $\alpha = a_1a_2\cdots a_n$ is *periodic* if there exists a string β such that $\alpha = \beta^j$ (where exponentiation denotes repetition) for some $j > 1$; otherwise we say α is *aperiodic* (or primitive). A *Lyndon word* is an aperiodic necklace. The *density* of a string α, denoted $den(\alpha)$, is the number of 1s in α. Let $\mathbf{N}(n,d)$ denote the set of necklaces of length n and density d, and let $\mathbf{L}(n,d)$ denote the set of Lyndon words of length n and density d. Let the cardinality of these sets be denoted by $N(n,d)$ and $L(n,d)$ respectively.

From [2], the number of fixed-density necklaces and Lyndon words are given by the following formulae:

$$N(n,d) = \frac{1}{n}\sum_{i \mid gcd(n,d)} \phi(i)\binom{n/i}{d/i},$$

$$L(n,d) = \frac{1}{n}\sum_{i \mid gcd(n,d)} \mu(i)\binom{n/i}{d/i},$$

where $\phi(i)$ denotes Euler's totient function and $\mu(i)$ is the Möbius function. The first formula is a direct result of Burnside's Lemma, and the latter result applies Möbius inversion. For these formulae, the underlying objects in question are all $\binom{n}{d}$ binary strings with length n and density d. Similar formulae can also be derived for subsets of these strings that are closed under rotation, such as the ones we define in the following section.

2 Ranking Fixed-Density Necklaces and Lyndon Words

In this section, polynomial-time ranking algorithms are developed for $\mathbf{N}(n,d)$ and $\mathbf{L}(n,d)$ when listed in lexicographic order.

Given a binary string $\alpha = a_1a_2\cdots a_n$, let $\mathbf{T}_\alpha(n,d)$ denote the set of all fixed-density binary strings of length n and density d whose necklace representatives are less than or equal to α. Note that α itself is not required to have density d. This set is closed under rotation. Let the cardinality of $\mathbf{T}_\alpha(n,d)$ be denoted by $T_\alpha(n,d)$.

Example 2 Let $\alpha =$ `011110` and consider $\mathbf{T}_\alpha(6,3)$ grouped by rotational equivalence:

```
000111 001011 001101 010101
100011 100101 100110 101010
110001 110010 010011
111000 011001 101001
011100 101100 110100
001110 010110 011010
```

Note that $T_\alpha(6,3) = 20$. The first string in each column (highlighted) is a necklace. Note that `010101` is periodic and hence not a Lyndon word.

Remark 2. Let α be an arbitrary binary string of length n. Let β be the largest binary necklace of length n that is less than or equal to α. Let γ be the largest necklace in $\mathbf{N}(n,d)$ that is less than or equal to β. Then $\mathbf{T}_\alpha(n,d) = \mathbf{T}_\beta(n,d) = \mathbf{T}_\gamma(n,d)$.

Let $\mathbf{N}_\alpha(n,d)$ and $\mathbf{L}_\alpha(n,d)$ denote the set of necklaces and Lyndon words of length n and density d, respectively, that are lexicographically less than or equal to α. Let $RankN_\alpha(n,d)$ and $RankL_\alpha(n,d)$ denote the cardinality of these sets. By applying Burnside's Lemma and Möbius inversion, these values can be computed using the following formulae when $\alpha \in \mathbf{N}(n,d)$:

$$RankN_\alpha(n,d) = \frac{1}{n} \sum_{i \,|\, gcd(n,d)} \phi(i)\, T_{a_1a_2\cdots a_{n/i}}(\tfrac{n}{i}, \tfrac{d}{i}), \tag{1}$$

$$RankL_\alpha(n,d) = \frac{1}{n} \sum_{i \,|\, gcd(n,d)} \mu(i)\, T_{a_1a_2\cdots a_{n/i}}(\tfrac{n}{i}, \tfrac{d}{i}). \tag{2}$$

The rank of a necklace α in $\mathbf{N}(n,d)$, with respect to lexicographic order, is given by $RankN_\alpha(n,d)$. Similarly, the rank of a Lyndon word α in $\mathbf{L}(n,d)$, with respect to lexicographic order, is given by $RankL_\alpha(n,d)$.

Example 3 Consider a necklace $\alpha = \texttt{010101}$. Observe from Example 2 that $RankN_\alpha(6,3) = 4$ and $RankL_\alpha(6,3) = 3$. Given that $\mathbf{T}_{\texttt{01}}(2,1) = \{\texttt{01}, \texttt{10}\}$, the calculations from their formulae in (1) and (2) are as follows:

$$RankN_\alpha(6,3) = \frac{1}{6}(\phi(1) \cdot T_\alpha(6,3) + \phi(3) \cdot T_{\texttt{01}}(2,1)) = \frac{1}{6}(1 \cdot 20 + 2 \cdot 2) = 4,$$

$$RankL_\alpha(6,3) = \frac{1}{6}(\mu(1) \cdot T_\alpha(6,3) + \mu(3) \cdot T_{\texttt{01}}(2,1)) = \frac{1}{6}(1 \cdot 20 + (-1) \cdot 2) = 3.$$

The following lemma proves that the formulae in equations (1) and (2) also hold if α is an arbitrary necklace.

Lemma 2. *Let α be a binary necklace of length n. Then $RankN_\alpha(n,d)$ is given by equation (1) and $RankL_\alpha(n,d)$ is given by equation (2).*

Proof. Let $\alpha = a_1a_2\cdots a_n$ be a binary necklace and let $\beta = b_1b_2\cdots b_n$ be the largest necklace in $\mathbf{N}(n,d)$ that is less than or equal to α. From Remark 2, $RankN_\alpha(n,d) = RankN_\beta(n,d)$ and $RankL_\alpha(n,d) = RankL_\beta(n,d)$. Now, suppose there exists an i that divides n such that $T_{\beta_i}(\frac{n}{i}, \frac{d}{i}) \neq T_{\alpha_i}(\frac{n}{i}, \frac{d}{i})$, where $\alpha_i = a_1a_2\cdots a_{n/i}$ and $\beta_i = b_1b_2\cdots b_{n/i}$. Then there must exist some necklace γ of length n/i and density d/i that is greater than β_i but less than or equal to α_i. But then γ^i is a necklace of length n and density d that is greater than β and less than or equal to α (since γ^i is the lexicographically least necklace of length n with prefix γ), a contradiction. Thus for each i that divides n we have $T_{\beta_i}(\frac{n}{i}, \frac{d}{i}) = T_{\alpha_i}(\frac{n}{i}, \frac{d}{i})$. Therefore the formula for equation (1) will produce the same results for both α and β respectively, as desired. The same holds for equation (2).

The formulae in equations (1) and (2) may not hold if α is an arbitrary string. To see this consider $\alpha = 010000$. The formula from (1) yields

$$\frac{1}{6}(\phi(1) \cdot T_\alpha(6,3) + \phi(3) \cdot T_{\texttt{01}}(2,1)) = \frac{1}{6}(1 \cdot 18 + 2 \cdot 2) = 22/6,$$

instead of 3. However, from Remark 2, we can compute $RankN_\alpha(n,d)$ for an arbitrary binary string α by computing $RankN_\beta(n,d)$, where β is the largest necklace of length n that is less than or equal to α. A similar method holds for $RankL_\alpha(n,d)$.

Let $\alpha = a_1a_2\cdots a_n$. Let the function LargestNecklace(α, n) return the largest binary necklace β of length n that is less than or equal to α (such a necklace always exists since 0^n is a necklace). Based on the previous discussion, Algorithm 1 will compute the values $RankN_\alpha(n,d)$ and $RankL_\alpha(n,d)$, respectively, for an arbitrary binary string α of length n.

Theorem 1. *The rank of a fixed-density necklace (or Lyndon word) $\alpha = a_1a_2\cdots a_n$ in the lexicographic ordering of $\mathbf{N}(n,d)$ (or $\mathbf{L}(n,d)$) can be determined in $O(n^3)$ time.*

The proof of this theorem relies on the following results:

- The function LargestNecklace(α, n) can be implemented in $O(n^2)$ time [9].
- In Section 2.1 an $O(n^3)$ algorithm is developed to compute $T_\alpha(n,d)$.
- For any real number $r > 1$ we have $\sum_{d|n} d^r = O(n^r)$, see e.g. [4].
- The functions $\phi(n)$ and $\mu(n)$ can be computed in $O(n)$ time, see e.g. [3].

Together, these three results imply that the functions RankN(α, n, d) and RankL(α, n, d) run in $O(n^3)$ time.

Algorithm 1 Computing the number of necklaces (Lyndon words) of length n and density d that are less than or equal to $\alpha = a_1a_2\cdots a_n$.

> **function** $\textsc{RankN}(\alpha, n, d)$ **returns integer**
> $r \leftarrow 0$
> $b_1b_2\cdots b_n \leftarrow \textsc{LargestNecklace}(\alpha, n)$
>   **for** $i \in$ divisors of $gcd(n,d)$ **do** $r \leftarrow r + \phi(i) \cdot T_{b_1b_2\cdots b_{n/i}}(\frac{n}{i}, \frac{d}{i})$
>   **return** r/n
>
> **function** $\textsc{RankL}(\alpha, n, d)$ **returns integer**
> $r \leftarrow 0$
> $b_1b_2\cdots b_n \leftarrow \textsc{LargestNecklace}(\alpha, n)$
>   **for** $i \in$ divisors of $gcd(n,d)$ **do** $r \leftarrow r + \mu(i) \cdot T_{b_1b_2\cdots b_{n/i}}(\frac{n}{i}, \frac{d}{i})$
>   **return** r/n

2.1 Computing $T_\alpha(n, d)$

In this subsection we present an efficient algorithm to compute $T_\alpha(n,d)$ for an arbitrary binary string α of length n. Recall that $\textsc{LargestNecklace}(\alpha, n)$ computes the lexicographically largest necklace β that is less than or equal to α. From Remark 2, $T_\alpha(n,d) = T_\beta(n,d)$. Thus, for the discussion in the remainder of this section, we make the assumption that α is a necklace.

The partition-based approach we present is similar to the one used in [9], but in our case we have an added density constraint. Before presenting the algorithm, we first study a special set.

A Special Set $\mathbf{B}_\alpha(t, j, d)$ Let $\alpha = a_1a_2\cdots a_n$ be a necklace. Let $\mathbf{B}_\alpha(t,j,d)$ denote the set of binary strings of length t with prefix $a_1a_2\cdots a_j$, density d, and no suffix that is less than or equal to α. Let the cardinality of this set be denoted $B_\alpha(t,j,d)$. When $j = 0$, observe that there is no prefix restriction on the string. In the cases that $d < den(a_1a_2\cdots a_j)$ or $d > t{-}j{+}den(a_1a_2\cdots a_j)$, the density constraint is not attainable and thus $B_\alpha(t,j,d) = 0$. This includes the case when d is negative.

Example 4 Let $\alpha = \texttt{00010011}$. Consider the set $\mathbf{B}_\alpha(6,1,3)$ partitioned at the $(j{+}1)^{st} = 2^{nd}$ element (underlined):

$\texttt{0}\underline{\texttt{0}}\texttt{0111}$	$\texttt{0}\underline{\texttt{1}}\texttt{0011}$
$\texttt{0}\underline{\texttt{0}}\texttt{1011}$	$\texttt{0}\underline{\texttt{1}}\texttt{0101}$
$\texttt{0}\underline{\texttt{0}}\texttt{1101}$	$\texttt{0}\underline{\texttt{1}}\texttt{1001}$

Notice that no string ending with a $\texttt{0}$ is in $\mathbf{B}_\alpha(6,1,3)$ since its length 1 suffix is lexicographically less than α. Observe that the contents of the first column correspond to the set $\mathbf{B}_\alpha(6,2,3)$, and the contents of the second column following the underlined element correspond to the set $\mathbf{B}_\alpha(4,0,2)$. Thus,

$$\mathbf{B}_\alpha(6,1,3) = \mathbf{B}_\alpha(6,2,3) \cup \texttt{01} \cdot \mathbf{B}_\alpha(4,0,2),$$

where the notation $\alpha \cdot \mathbf{S}$ denotes the set obtained by prepending the string α to each string in the set $\mathbf{S}$. From this it follows that:

$$B_\alpha(6,1,3) = B_\alpha(6,2,3) + B_\alpha(4,0,2).$$

Based on the recursive structure observed in the above example, we present an enumeration formula for $B_\alpha(t,j,d)$ where $0 \le j \le t \le n$ and $0 \le d \le t$. For $t{=}j{=}d{=}0$, we define $B_\epsilon(0,0,0) = 1$ by accounting for the empty string. For $t > 0$, $B_\alpha(t,t,d) = 0$ since the suffix $a_1a_2\cdots a_t$ is a prefix of α and hence is less than or equal to α. In the remaining case where $t > j$, consider a partition of $\mathbf{B}_\alpha(t,j,d)$ based on the symbol in position $j{+}1$. For any string in this set the $(j{+}1)$st symbol must be greater than or equal to a_{j+1} because otherwise the suffix starting from the first index would be less than α. Now observe that:

– If the $j{+}1$st symbol is a_{j+1}, then the number of such strings in $\mathbf{B}_\alpha(t,j,d)$ is $B_\alpha(t,j{+}1,d)$.
– If the $j{+}1$st symbol is greater than a_{j+1} (which will be possible only if $a_{j+1} = 0$), then any suffix starting at index $1,2,\ldots,j+1$ is larger than α from Remark 1. Thus, the length $t-j-1$ suffix of such a string in $\mathbf{B}_\alpha(t,j,d)$ can be an arbitrary element of $\mathbf{B}_\alpha(t{-}j{-}1, 0, d{-}den(a_1\cdots a_j){-}1)$.

Thus, for $0 \le j \le t \le n$ and $d \le n$:

$$B_\alpha(t,j,d) = \begin{cases} 1 & \text{if } t{=}j{=}d{=}0, \\ 0 & \text{if } (t{=}j \text{ and } t>0) \text{ or } (t{=}j{=}0 \text{ and } d \neq 0), \\ B_\alpha(t,j{+}1,d) & \text{if } 0 \le j < t \text{ and } a_{j+1} = \mathtt{1}, \\ B_\alpha(t,j{+}1,d) + B_\alpha(t{-}j{-}1,0,d{-}den(a_1\cdots a_j){-}1) & \text{if } 0 \le j < t \text{ and } a_{j+1} = \mathtt{0}. \end{cases}$$

By applying dynamic programming, these values can easily be computed in $O(n^3)$ time as presented in Algorithm 2. In this algorithm, the last two cases of the recurrence are combined. As an additional simplification, when $d{-}den(a_1a_2\cdots a_j){-}1 < 0$ note that repeated application of the recurrence yields $B_\alpha(t,j,d) = 0$.

Partitioning $\mathbf{T}_{\boldsymbol{\alpha}}(\boldsymbol{n},\boldsymbol{d})$ Recall our goal is to compute $T_\alpha(n,d)$ where α is a necklace. Our first step is to partition the strings $\omega = w_1w_2\cdots w_n \in \mathbf{T}_\alpha(n,d)$ into subsets based on the smallest index t such that

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

Such an index t exists since $neck(\omega) \le \alpha$ by the definition of $\mathbf{T}_\alpha(n,d)$. We further partition each of these subsets based on the largest integer $0 \le j \le n$ such that $a_1a_2\cdots a_j$ is a prefix of $w_tw_{t+1}\cdots w_nw_1w_2\cdots w_{t-1}$. We denote the set of strings in each subpartition by $\mathbf{A}_\alpha(t,j,d)$, and the cardinality of $\mathbf{A}_\alpha(t,j,d)$ by $A_\alpha(t,j,d)$. Thus,

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

Example 5 Let $\alpha = \mathtt{0010101}$. The $T_\alpha(7,3) = 35$ strings can be partitioned into subsets $\mathbf{A}_\alpha(t,j,3)$ for $0 \le j \le 7$ and $1 \le t \le 7$. Each respective substring $a_1a_2\cdots a_j$ is underlined.

$\mathbf{A}_\alpha(t,j,3)$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$
$j=2$	$\underline{00}\,00111$ $\underline{00}\,01011$ $\underline{00}\,01101$ $\underline{00}\,01110$	$1\,\underline{00}\,0011$ $1\,\underline{00}\,0101$ $1\,\underline{00}\,0110$	$11\,\underline{00}\,001$ $11\,\underline{00}\,010$ $01\,\underline{00}\,011$	$111\,\underline{00}\,00$ $011\,\underline{00}\,01$ $101\,\underline{00}\,01$	$0111\,\underline{00}\,0$ $1011\,\underline{00}\,0$ $1101\,\underline{00}\,0$	$00111\,\underline{00}$ $01011\,\underline{00}$ $01101\,\underline{00}$	$\underline{0}\,01011\,\underline{0}$ $\underline{0}\,01101\,\underline{0}$
$j=4$	$\underline{0010}\,011$	$1\,\underline{0010}\,01$	$11\,\underline{0010}\,0$	$011\,\underline{0010}$	$\underline{0}\,011\,\underline{001}$	$\underline{10}\,011\,\underline{00}$	$\underline{010}\,011\,\underline{0}$
$j=7$	$\underline{0010101}$	$\underline{1}\,\underline{001010}$	$\underline{01}\,\underline{00101}$	$\underline{101}\,\underline{0010}$	$\underline{0101}\,\underline{001}$	$\underline{10101}\,\underline{00}$	$\underline{010101}\,\underline{0}$

For $j \in \{0,1,3,5,6\}$ note that $\mathbf{A}_\alpha(t,j,3) = \emptyset$.

We have reduced the problem of computing $T_\alpha(n,d)$ to that of computing $A_\alpha(t,j,d)$. When $j=n$, the set $\mathbf{A}_\alpha(1,n,d) \cup \mathbf{A}_\alpha(2,n,d) \cup \cdots \cup \mathbf{A}_\alpha(n,n,d)$ contains the unique rotations of α when α has density d; otherwise it is empty. The size of this set can easily be computed in $O(n)$-time using standard methods [1]. For $j<n$, each set $\mathbf{A}_\alpha(t,j,d)$ falls into one of the following two cases depending on whether the symbol x following the substring $a_1a_2\cdots a_j$ in question is involved in the wraparound. By the definition of $\mathbf{T}(\alpha,n,d)$, for each string $\omega \in \mathbf{A}_\alpha(t,j,d)$ we have $neck(\omega) < \alpha$. Thus x must be less than a_{j+1} and hence $x=0$ and $a_{j+1}=1$.

Case 1 ($t+j \leq n$). Each $\omega \in \mathbf{A}_\alpha(t,j,d)$ is of the form $\sigma\, a_1a_2\cdots a_j\, 0\, \tau$ where:

- $\sigma \in \Sigma^{t-1}$ such that every non-empty suffix is larger than α (by Lemma 1 and definition of t)[3],
- $\tau \in \Sigma^{n-t-j}$.

To satisfy the density constraint d, the density of the strings σ and τ when added together must be $d - den(a_1\cdots a_j)$. Recall from Section 2.1 that the number of possibilities for σ with density i is given by $B_\alpha(t-1,0,i)$. Thus, in this case, we have:

$$A_\alpha(t,j,d) = \sum_{i=0}^{d-den(a_1\cdots a_j)} B_\alpha(t-1,0,i) \cdot \binom{n-t-j}{d-den(a_1\cdots a_j)-i}.$$

Recall that if $i<0$ or if $i>t-1$, then by definition $B_\alpha(t-1,0,i)=0$.

Case 2 ($t+j>n$). Each $\omega \in \mathbf{A}_\alpha(t,j,d)$ is of the form $a_{n-t+2}\cdots a_{j-1}a_j\, 0\, \sigma\, a_1a_2\cdots a_{n-t+1}$ where:

- $\sigma \in \Sigma^{n-j-1}$, and
- $a_{n-t+2}\cdots a_{j-1}a_j\, 0\, \sigma$ has every non-empty suffix larger than α (by Lemma 1 and definition of t)[4].

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. If this suffix has length s, then $a_{j-s+1}\cdots a_{j-1}a_j = a_1a_2\cdots a_s$. This means any suffix of ω starting from an index less than or equal to $|\delta|-s$ is larger than α. Now, if $a_{s+1}=1$, then $a_{j-s+1}\cdots a_{j-1}a_j0 < \alpha$, which contradicts the definition of t in $\mathbf{A}_\alpha(t,j,d)$. Otherwise, if $a_{s+1}=0$, then $a_1a_2\cdots a_s0\sigma \in \mathbf{B}_\alpha(n-j+s, s+1, d-den(a_1\cdots a_j)+den(a_1\cdots a_s))$. Thus, since we already determined that $a_{j+1}=1$, we have:

$$A_\alpha(t,j,d) = \begin{cases} B_\alpha(n-j+s, s+1, d-den(a_1\cdots a_j)+den(a_1\cdots a_s)) & \text{if } a_{j+1} > a_{s+1}, \\ 0 & \text{otherwise.} \end{cases}$$

An $O(n^3)$-time Algorithm to Compute $T_\alpha(n,d)$ Given an arbitrary binary string $\alpha = a_1a_2\cdots a_n$, the function $\mathrm{T}(\alpha,n,d)$ in Algorithm 2 computes $T_\alpha(n,d)$ in $O(n^3)$ time. The first step is to re-assign α to LARGESTNECKLACE(α,n). Given that α is a now a necklace, we can apply the formulae derived in the previous subsection. 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. The algorithm includes precomputation of the values $suf_\alpha(i,j)$ for $2 \leq i \leq j \leq n$ as well as the values $B_\alpha(t,j,d)$ using a standard dynamic programming approach. Note the values $den(a_1a_2\cdots a_j)$ for each $1 \leq j \leq n$ can be precomputed in $O(n)$ time.

Lemma 3. *$T_\alpha(n,d)$ can be computed in $O(n^3)$ time for any binary string α of length n.*

This completes the proof of Theorem 1.

[3] Note that no suffix will equal α since $t-1<n$.

[4] Note that no suffix will equal α since $|a_{n-t+2}\cdots a_{j-1}a_j\, 0\, \sigma| < n$.

Algorithm 2 Computing $T_\alpha(n,d)$ for a given binary string $\alpha = a_1 a_2 \cdots a_n$.

function T(α, n, d) **returns integer**

  $\alpha \leftarrow$ LARGESTNECKLACE(α, n)

  ▷ Precompute $B_\alpha(t,j,d)$ using dynamic programming
  $B_\alpha(0,0,0) \leftarrow 1$
  for t **from** 1 **to** n **do**
    for i **from** 0 **to** n **do**
      $B_\alpha(t,t,i) \leftarrow 0$
      for j **from** $t-1$ **down to** 0 **do**
        if $i - den(a_1 \cdots a_j) - 1 < 0$ **then** $B_\alpha(t,j,i) \leftarrow 0$
        else $B_\alpha(t,j,i) \leftarrow B_\alpha(t,j{+}1,i) + (1 - a_{j+1}) \cdot B_\alpha(t{-}j{-}1, 0, i{-}den(a_1 \cdots a_j){-}1)$

  ▷ Precompute $suf_\alpha(i,j)$ for $2 \le i \le j \le n$
  for i **from** 2 **to** n **do**
    $z \leftarrow i$
    for j **from** i **to** n **do**
      if $a_j > a_{j-z+1}$ **then** $z \leftarrow j+1$
      $suf_\alpha(i,j) \leftarrow j - z + 1$

  ▷ Compute $T_\alpha(n,d)$
  if $den(\alpha) = d$ **then** $total \leftarrow$ the number of unique rotations of α ▷ The case when $j = n$
  else $total \leftarrow 0$
  for t **from** 1 **to** n **do**
    for j **from** 0 **to** $n{-}1$ **do**
      if $j + t \le n$ **then**
        if $a_{j+1} > 0$ **then**
          for i **from** 0 **to** $d - den(a_1 \cdots a_j)$ **do** $total \leftarrow total + B_\alpha(t-1,0,i) \cdot \binom{n-t-j}{d-den(a_1\cdots a_j)-i}$
      else
        if $n{-}t{+}2 > j$ **then** $s \leftarrow 0$
        else $s \leftarrow suf_\alpha(n{-}t{+}2, j)$
        $d' \leftarrow d{-}den(a_1 \cdots a_j){+}den(a_1 \cdots a_s)$
        if $a_{j+1} > a_{s+1}$ **and** $d' \ge 0$ **then** $total \leftarrow total + B_\alpha(n{-}j{+}s, s{+}1, d')$
  return $total$

3 Unranking Necklaces and Lyndon Words

The *unranking* problem for fixed-density necklaces is to find the necklace α in the lexicographic ordering of $\mathbf{N}(n,d)$ with rank r, where $1 \le r \le N(n,d)$. Let UNRANK(n,d,r) denote this necklace. Starting from 1^n successive calls to RANKN(α,n,d) can be used to determine each bit of α as illustrated in Algorithm 3. A similar approach works for Lyndon words using RANKL(α,n,d) instead of RANKN(α,n,d).

Algorithm 3 An $O(n^4)$-time unranking algorithm for fixed-density necklaces.

function UNRANK(n,d,r) **returns** fixed-density necklace
  $\alpha = a_1 a_2 \cdots a_n \leftarrow 1^n$
  for i **from** 1 **to** n **do**
    $a_i \leftarrow 0$
    if $r >$ RANKN(α,n,d) **then** $a_i \leftarrow 1$
  return α

Theorem 2. *The fixed-density necklace (Lyndon word) at rank r in the lexicographic order of fixed-density necklaces (Lyndon words) of length n respectively can be determined in $O(n^4)$ time.*

4 Fixed-Density Necklaces (Lyndon Words) with a Given Prefix

Consider a binary string $\alpha = a_1a_2 \cdots a_j$ for $1 \leq j \leq n$ and assume $0 \leq d \leq n$. Let $preN_\alpha(n, d)$ denote the number of necklaces in $\mathbf{N}(n, d)$ with prefix α and let $preL_\alpha(n, d)$ denote the number of Lyndon words in $\mathbf{L}(n, d)$ with prefix α. In this section we describe a polynomial-time algorithm to compute $preN_\alpha(n, d)$ and $preL_\alpha(n, d)$.

First, consider the special case when $j = n$. If $\alpha \in \mathbf{N}(n, d)$ then $preN_\alpha(n, d) = 1$; otherwise $preN_\alpha(n, d) = 0$. Similarly, if $\alpha \in \mathbf{L}(n, d)$ then $preL_\alpha(n, d) = 1$; otherwise $preL_\alpha(n, d) = 0$. Testing whether or not a string is a necklace or a Lyndon word can be determined in $O(n)$ time using standard techniques [1]. Therefore this case can be resolved in $O(n)$ time.

Now assume that $j < n$. If $d = 0$, then $preL_\alpha(n, d) = 0$. Also $preN_\alpha(n, d) = 0$, unless $\alpha = 0^j$ in which case $preN_\alpha(n, d) = 1$. Otherwise assume that $1 \leq d \leq n$. The largest binary string of length n with α as a prefix is $\delta = \alpha 1^{n-j}$. The smallest binary string of length n with α as a prefix is $\gamma = \alpha 0^{n-j}$. Since $d \geq 1$, γ is not a necklace. Thus,

$$preN_\alpha(n, d) = RankN_\delta(n, d) - RankN_\gamma(n, d).$$

Similarly, for Lyndon words we have:

$$preL_\alpha(n, d) = RankL_\delta(n, d) - RankL_\gamma(n, d).$$

Theorem 3. *$preN_\alpha(n, d)$ and $preL_\alpha(n, d)$ can be computed in $O(n^3)$ time for $0 \leq d \leq n$.*

5 Summary and Future Work

We have presented the first polynomial-time algorithms to rank and unrank binary fixed-density necklaces and Lyndon words of length n listed in lexicographic order. By applying a similar enumeration framework, it appears that binary unlabeled necklaces, which are strings with equivalence classes under both rotation and complementing of the alphabet symbols, can also efficiently be ranked/unranked. It remains an open problem to efficiently rank/unrank *bracelets*, which are strings with equivalence under both rotation and string reversal.

A C implementation of our algorithms for n up to 66 is provided in the appendix. When $n = 66$, the largest integer computed will be stored in the variable r in the RANK function when $d = 33$. It will have value equal to n times $N(66, 33)$. This number is less than the largest available integer of 2^{63} using the `long long int` data type to store the integers. For simplicity, the values of $\phi(n)$ and $\mu(n)$ are pre-computed for n up to 66.

6 Acknowledgements

We would like to commend 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 298335-2012

References

1. K. S. Booth. Lexicographically least circular substrings. *Information Processing Letters*, 10(4/5):240–242, 1980.
2. E. N. Gilbert and J. Riordan. Symmetry types of periodic sequences. *Illinois J. Math.*, 5(4):657–665, 12 1961.
3. R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994.
4. T. Kociumaka, J. Radoszewski, and W. Rytter. Efficient ranking of Lyndon words and decoding lexicographically minimal de Bruijn sequence. *SIAM J. Discrete Math*, 30(4):2027–2046, 2016.
5. S. Kopparty, M. Kumar, and M. Saks. Efficient indexing of necklaces and irreducible polynomials over finite fields. *Theory of Computing*, 12(7):1–27, 2016.
6. F. Ruskey. Combinatorial generation. *Preliminary working draft. University of Victoria, Victoria, BC, Canada*, 11:20, 2003.
7. J. Sawada and F. Ruskey. An efficient algorithm for generating necklaces with fixed density. *SIAM J. Comput*, 29:671–684, 1999.
8. J. Sawada and A. Williams. A Gray code for fixed-density necklaces and Lyndon words in constant amortized time. *Theoretical Computer Science*, 502:46 – 54, 2013. Generation of Combinatorial Structures.
9. J. Sawada and A. Williams. Practical algorithms to rank necklaces, Lyndon words, and de Bruijn sequences. *Journal of Discrete Algorithms*, 43:95–110, 2017.
10. T. Ueda. Gray codes for necklaces. *Discrete Math.*, 219:235–248, 2000.
11. T. M. Wang and C. D. Savage. A gray code for necklaces of fixed density. *SIAM J. Discrete Math*, 9:654–673, 1997.

Appendix - C code

```
//==============================================================================
// Ranking and unranking algorithms for binary fixed-density necklaces and
// Lyndon words
//
// Program by Patrick Hartman and Joe Sawada 2016, updated Oct 2017
//==============================================================================
#include <stdio.h>
#include <string.h>
#define MAX 67 /* max length of string (-1) */

// Precomputed values up to n=66
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,-1,1,0,
   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,
   40,24,36,28,58,16,60,30,36,32,48,20};

long long int choose[MAX][MAX], B[MAX][MAX][MAX];
int D[MAX],suf[MAX][MAX], NECKLACE=0, LYNDON=0;

//================================================
void PrintString(int a[], int n) {

   for (int i=1; i<=n; i++) printf("%d", a[i]);
   printf("\n");
}
//=================================
// Compute (n choose r) up to n=max
//=================================
void ComputeChoose(int max) {
   int n,r;

   for (n=0; n<=max; n++) {
      for (r=0; r<=max; r++) {
         if (r > n) choose[n][r] = 0;
         else if (n == r || r == 0) choose[n][r] = 1;
         else choose[n][r] = choose[n-1][r-1] + choose[n-1][r];
      }
   }
}
//======================================
// Compute D[i] = den(a[1..i]) for i=1..n
//======================================
void ComputeD(int a[], int n) {

   D[0] = 0;
   for (int i=1; i<=n; i++) {
      if (a[i] > 0) D[i] = D[i-1] + 1;
      else D[i] = D[i-1];
   }
}
//=============================================
// Compute B[t][j][d] for 0<=j<=t<=n and 0<=d<=t
//=============================================
void ComputeB(int a[], int n) {

   B[0][0][0] = 1;
   for (int t=1; t<=n; t++) {
      for (int d=0; d<=n; d++) {
         B[t][t][d] = 0;
         for (int j=t-1; j>=0; j--) {
            if (d - D[j] - 1 < 0) B[t][j][d] = 0;
            else B[t][j][d] = B[t][j+1][d] + (1 - a[j+1]) * B[t-j-1][0][d-D[j]-1];
```

```
            }
         }
      }
   }
//==================================================
// Compute suffix function suf[i][j] for 2<=i<=j<=n
//==================================================
void ComputeSuf(int a[], int n) {
   int p;

   for (int i=2; i<=n; i++) {
      p = i - 1;
      for (int j=i; j<=n; j++) {
         if (a[j] > a[j-p]) p = j;
         suf[i][j] = j - p;
      }
   }
}
//==========================================================
// Find the longest prefix of a[1..n] that is a Lyndon word
//==========================================================
int Lyn(int a[], int n) {
   int i,p=1;

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

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

   for (i=1; i<=n; i++) neck[i] = a[i];
   while (!IsNecklace(neck,n)) {
      p = Lyn(neck,n);
      neck[p]=0;
      for (i=p+1; i<=n; i++) neck[i]=1;
   }
}
//============================================================
// Precompute D, B, and suf. Compute cardinality of T_a(n,d)
//============================================================
long long int T(int a[], int n, int d) {
   long long int total=0;
   int i,j,s,t,neck[MAX];

   LargestNecklace(a,n,neck);
   ComputeD(neck,n);
   ComputeB(neck,n);
   ComputeSuf(neck,n);

   if (D[n] != d) total = 0;
```

```
    else total = Lyn(neck, n);

    for (t=1; t<=n; t++) {
        for (j=0; j<=n-1; j++) {
            if (t+j <= n) {
                if (neck[j+1] > 0) {
                    for (i=0; i<=d-D[j]; i++) total += B[t-1][0][i] * choose[n-t-j][d-D[j]-i];
                }
            }
            else{
                if (n-t+2 > j) s = 0;
                else s = suf[n-t+2][j];
                if (neck[j+1] > neck[s+1] && d-D[j]+D[s] >=0) total += B[n-j+s][s+1][d-D[j]+D[s]];
            }
        }
    }
    return total;
}
//==========================================================================
// Compute the number of fixed-density necklaces or Lyndon words <= a[1..n]
//==========================================================================
long long int Rank(int a[], int n, int d) {
    long long int r=0;
    int neck[MAX];

    LargestNecklace(a,n,neck);

    for (int i=1; i<=n; i++) {
        if (n % i == 0 && d % i == 0) {
            if (NECKLACE) r += phi[i] * T(neck, n/i, d/i);
            else if (LYNDON) r += mu[i] * T(neck, n/i, d/i);
        }
    }
    return r/n;
}
//=================================================================================
// Return the fixed-density necklace or Lyndon word a[1..n] at rank r in lex order
//=================================================================================
int Unrank(long long int r, int n, int d, int a[]) {

    // Start with string with largest rank
    for (int j=1; j<=n; j++) a[j] = 1;

    // Determine a[j] from left to right
    for (int j=1; j<=n; j++) {
        a[j] = 0;
        if (r > Rank(a,n,d)) a[j] = 1;
    }
    if (r == Rank(a,n,d)) return 1;
    return 0; // Invalid rank r
}
//===========================================================================================
// Count number of necklaces or Lyndon words of length n and density d with prefix a[1..pn]
//===========================================================================================
long long int NumPrefix(int n, int d, int a[], int pn) {
    long long int max, min;
    int pd=0, i;

    // Special cases when pn = n and d = 0
    if (pn == n || d == 0) {
        for (i=1; i<=pn; i++) if (a[i] == 1) pd++;
        if (pd == d && ((NECKLACE && IsNecklace(a,pn)) || (LYNDON && Lyn(a,pn) == n))) return 1;
        else return 0;
    }

    for (i=pn+1; i<=n; i++) a[i] = 0;
    min = Rank(a,n,d);
    for (i=pn+1; i<=n; i++) a[i] = 1;
    max = Rank(a,n,d);
```

```
    return max-min;
}
//==============================================================================
int main() {
    char input[MAX];
    long long int r,i;
    int n,j,d,option,a[MAX];

    printf("----------------------------\n");
    printf(" Fixed-density necklaces\n");
    printf("----------------------------\n");
    printf(" 1. Rank \n");
    printf(" 2. Unrank \n");
    printf(" 3. Count with given prefix\n");
    printf(" 4. Exhaustive generation\n");
    printf("----------------------------\n");
    printf(" Fixed-density Lyndon words\n");
    printf("----------------------------\n");
    printf(" 5. Rank \n");
    printf(" 6. Unrank \n");
    printf(" 7. Count with given prefix \n");
    printf(" 8. Exhaustive generation\n\n");

    printf("Enter option: "); scanf("%d", &option);

    if (option < 1 || option > 8) return 0;
    if (option == 1 || option == 2 || option == 3 || option == 4) NECKLACE = 1;
    else LYNDON = 1;

    printf("Enter length N (< %d): ", MAX); scanf("%d", &n);
    printf("Enter density D: "); scanf("%d", &d);
    if (d > n) { printf("Invalid density\n"); return 0; }

    ComputeChoose(n); // Pre-compute binomial co-efficients

    // RANK
    if (option == 1 || option == 5) {
       printf("Enter necklace/Lyndon word (eg. 001101): "); scanf("%s", input);
       if (strlen(input) != n) { printf("Invalid length\n"); return 0; }
       j = strlen(input);
       for (i=1; i<=j; i++) a[i] = input[i-1] - '0';
       printf("Rank = %lld\n", Rank(a, n, d));
    }
    // UNRANK
    else if (option == 2 || option == 6) {
       printf("Enter rank: "); scanf("%lld", &r);
       if (!Unrank(r, n, d, a)) printf("Invalid rank\n");
       else PrintString(a, n);
    }
    // COUNT w given prefix
    else if (option == 3 || option == 7) {
       printf("Enter prefix (eg. 0010): "); scanf("%s", input);
       j = strlen(input);
       for (i=1; i<=j; i++) a[i] = input[i-1] - '0';
       printf("%lld\n", NumPrefix(n, d, a, strlen(input)));
    }
    // GENERATE ALL USING RANK/UNRANK
    else {
       i = 1; printf("\n");
       while (Unrank(i,n,d,a)) {
          r = Rank(a,n,d);
          if (i != r) { printf("Broken at rank %lld\n", i); return 0; }
          PrintString(a, n);
          i++;
       }
       printf("\nTotal = %lld\n", i-1);
    }
}
```