On the generation of necklaces and bracelets in R

Elvira Di Nardo\textsuperscript{a} and Giuseppe Guarino\textsuperscript{b}

\textsuperscript{a}Dipartimento di Matematica "G. Peano", Università degli Studi di Torino, Torino, Italy; \textsuperscript{b}Medical School, Università Cattolica del Sacro Cuore, Roma, Italy

ABSTRACT
This note introduces a code snippet in R aiming to generate necklaces as well as bracelets. Among various uses, necklaces are useful tools to manage traces of products of random matrices. Functionality for necklaces and bracelets is provided with some examples of applications such as Lyndon words and de Bruijn sequences. The routines are collected in the Necklaces package available from the Comprehensive R Archive Network.

ARTICLE HISTORY
Received 22 August 2022
Accepted 24 March 2023

KEYWORDS
Bracelet; de Bruijn sequence; Lyndon word; Multi-index composition; Necklace

1. Introduction
Necklaces and bracelets are fundamental combinatorial objects arising in the field of text algorithms (Kociumaka, Radoszewski, and Rytter 2014), in the construction of single-track Gray codes (Schwartz and Etzion 1999), in the analysis of circular DNA and splicing systems (De Felice, Zaccagnino, and Zizza 2017) or in managing the distribution of individual wire faults in parallel on-chip links (Vitkovskiy, Christodoulides, and Soteriou 2012). Applications in probability and statistics are related to random splitting (Alon et al. 2021), necklace processes (Mallows and Shepp 2008) or fixed-density $m$-ary necklaces (Ruskey and Sawada 1999). Dealing with matrix algebra, they are used for the well-known invariance property of the trace, that is the trace of a matrix product is invariant under cyclic permutations. For example if $\Sigma$ is a $p \times p$ non-negative definite matrix and $H_1, H_2 \in \mathbb{C}^{p \times p}$ are complex matrices, then $\text{Tr}[(\Sigma H_1) \cdot (\Sigma H_2)] = \text{Tr}[(\Sigma H_1) \cdot (\Sigma H_2)] = \text{Tr}[(\Sigma H_2) \cdot (\Sigma H_1) \cdot (\Sigma H_1)]$. The set of indexes $\{112, 121, 211\}$ is an example of binary necklace of length 3 on the alphabet $\{1, 2\}$.

More in details, a $m$-ary necklace of length $n$ is an equivalence class of $n$-character strings over an alphabet $X$ of size $m$, taking all rotations equivalent (Flajolet and Sedgewick 2009). That is, if $\sigma$ denotes the rotation pictured as $x_1 \rightarrow x_2, x_2 \rightarrow x_3, \ldots, x_n \rightarrow x_1$, the $m$-ary $n$-tuples $y$ and $z$ are in the same necklace if and only if $\sigma^j(y) = z$ for some integer $j$. Sometimes a $m$-ary necklace of length $n$ is represented with $n$ circularly connected beads colored with up to $m$ different colors. Observe that the $n$-strings of a necklace are $n$-tuples of the set $X^n$. Without losing generality, the elements of $X$ can be labeled with the first $m$ positive integers, that is $X = [m] = \{1, 2, \ldots, m\}$, or with $X = \{0, 1, 2, \ldots, m-1\}$ including the zero. A $m$-ary bracelet of length $n$ is an equivalence class where not only all rotations are equivalent but also all reflections (reverse order).

It is quite natural to choose the lexicographically smallest string $a$ as the representative of a necklace (or bracelet). Let us denote the class with $\langle a \rangle$.
Example 1.1. There are 6 binary necklaces of length 4:
\[
\begin{align*}
\langle 0 & 0 0 1 \rangle = \{1000, 0100, 0010, 0001\}, \langle 0 & 0 1 1 \rangle = \{1100, 1001, 0011, 0110\}, \\
\langle 0 & 1 1 1 \rangle = \{1110, 1101, 1011, 0111\}, \\
\langle 0 & 1 0 1 \rangle = \{1010, 0101\}, \langle 0 & 1 1 0 \rangle = \{1110\}, \langle 0 & 0 0 0 \rangle = \{0000\},
\end{align*}
\]
all of which are also binary bracelets. For \( n = 6 \), we have 14 binary necklaces but only 13 binary bracelets as the necklaces obtained rotating 101100 and 001101 are reversal of each other, resulting into a single bracket class, see Example 3.4 in Sec. 3.

Necklaces turn to be useful also in free probability (Nica and Speicher 2006) when computing moments of suitable products of random matrices. For example, using necklaces a closed form expression is given in Di Nardo (2014) to compute
\[
\mathbb{E}(\text{Tr}[W(n)H_1]^{i_1} \cdots \text{Tr}[W(n)H_m]^{i_m})
\]  
(1.1)
where \( H_1, \ldots, H_m \in \mathbb{C}^{p \times p} \) are complex matrices, \( i_1, \ldots, i_m \) are non-negative integers and \( W(n) \) is a non-central Wishart square random matrix of order \( p \) and parameter \( n \).

Example 1.2. For example\(^{1} \) for \( m = 2 \) and \( i_1 = 1, i_2 = 1 \) we have
\[
\mathbb{E}(\text{Tr}[W(n)H_1]\text{Tr}[W(n)H_2]) = \text{Tr}((\Omega \Sigma H_1)\text{Tr}(\Omega \Sigma H_2)) - \text{Tr}(\Omega \Sigma H_1 \Sigma H_2)
\]
\[
\quad - \text{Tr}(\Omega \Sigma H_2 \Sigma H_1) + n^2\text{Tr}(\Sigma H_1)\text{Tr}(\Sigma H_2) + n\text{Tr}(\Sigma H_1 \Sigma H_2) - n\text{Tr}(\Omega \Sigma H_1) \times
\]
\[
\quad \times \text{Tr}(\Sigma H_2) - n\text{Tr}(\Sigma H_1)\text{Tr}(\Omega \Sigma H_2),
\]
where \( \Sigma \) is the covariance matrix and \( \Omega \) is the non-centrality matrix of the non-central Wishart random matrix \( W(n) \).

In computing (1.1), the notion of type of a necklace has been considered, according to the occurrence of each character of the alphabet in the \( n \)-tuple. Suppose \( i = (i_1, i_2, \ldots, i_m) \) and \( X = \lceil m \rceil \).

Definition 1.3. A necklace is said to be of type \( i \) if the character 1 appears \( i_1 \) times in the \( n \)-tuples, 2 appears \( i_2 \) times and so on.

Therefore a necklace of type \( i \) is such that \( i_j \) denotes the number of times the non-negative integer \( j \) occurs in a string of the necklace. The choice \( X = \lceil m \rceil \) is not a constrain as 1 can be replaced by whatever integer, 2 by the subsequent one and so on. A similar definition can be given for bracelets.

In \( \mathbb{R} \), up to now the only routines managing necklaces and bracelets are in the numbers package (Borchers 2021), where the Polya’s enumeration theorem (Harary 1994) is used to compute the number of representatives of \( m \)-ary necklaces (resp. bracelets) of length \( n \), see Example 3.4. Instead the Necklaces package can be used not only for producing the lists of all the \( n \)-tuples in a \( m \)-ary necklace (resp. bracelet) of length \( n \), but also the lists of all the representatives of \( m \)-ary necklaces (resp. bracelets) of type \( i \) as well as of \( m \)-ary necklaces (resp. bracelets) of length \( n \).

This note is organized as follows. Next section describes the functionality of those routines in the kStatistics package called by the routines of the Necklaces package. Section 3 adds more details about functionality of the routines of the Necklaces package as well as the implemented methods. Their performance is illustrated with many examples. Section 4 is devoted to examples of applications: the generation of Lyndon words and de Bruijn sequences. Some concluding remarks end the paper.

\(^{1}\)A Maple procedure to compute these moments is available in Di Nardo and Guarino (2013).
2. Functions of the kStatistics package

In order to use the functions of the Necklaces package, first install the kStatistics package (Di Nardo and Guarino 2021). Indeed the Necklaces package calls the nPerm function of the kStatistics package. This function returns all the permutations of an input vector of non-negative integers.

Example 2.1. Run \( \text{nPerm}(c(1, 0, 1, 1)) \) to get \{1110, 1101, 1011, 0111\}.

Example 2.2. Run \( \text{nPerm}(c(0, 1, 2)) \) to get \{012, 021, 102, 120, 201, 210\}.

The Necklaces package calls also the mkT function, which returns all the compositions of a multi-index \( i = (i_1, i_2, ..., i_m) \) in \( n \) multi-indexes (Di Nardo and Guarino 2022), that is all the lists \( (v_1, ..., v_n) \) of \( m \)-length multi-indexes such that \( v_1 + \cdots + v_n = i \).

Example 2.3. To get all the compositions of the multi-index \( (1, 0, 1) \) in two multi-indexes of length 3, run

\[
\text{mkT}(c(1,0,1),2, \text{TRUE})
\]

\[
\begin{bmatrix}
(0 0 1)(1 0 0) \\
(1 0 0)(0 0 1) \\
(1 0 1)(0 0 0) \\
(0 0 0)(1 0 1)
\end{bmatrix}
\]

In the Necklaces package, the mkT function is employed to recover the set \( C(m, n) \) of all counting vectors \( i \) of length \( m \) summing up to \( n \). Recall that the counting vector of an integer \( n \) is a sequence \( i = (i_1, i_2, ..., i_m) \) of non-negative integers, named parts of \( i \), such that \( i_1 + i_2 + \cdots + i_m = n \), with \( m \leq n \). The integer \( m = |i| \) is the number of parts and is usually called the length of the counting vector \( i \).

Example 2.4. To get all the counting vectors of \( n = 4 \) in \( m = 2 \) parts run

\[
\text{mkT}(4,2, \text{TRUE})
\]

\[
\begin{bmatrix}
(2)(2) \\
(1)(3) \\
(3)(1) \\
(4)(0) \\
(0)(4)
\end{bmatrix}
\]

The Necklaces package uses the set \( C(m, n) \). Indeed, fix an alphabet \( X \) of size \( m \) and suppose to denote with

i. \( N[i] \) (resp. \( B[i] \)) the set of all representatives of \( m \)-ary necklaces (resp. bracelets) of a fixed type \( i \);

ii. \( N_m(n) \) (resp. \( B_m(n) \)) the set of all representatives of \( m \)-ary necklaces \( N[i] \) (resp. bracelets \( B[i] \)) of type \( i \) with \( |i| = n \).

Then we have

\[
N_m(n) = \bigcup_{i \in C(m,n)} N[i] \quad \text{and} \quad B_m(n) = \bigcup_{i \in C(m,n)} B[i] \tag{2.1}
\]
where the union is intended to be disjointed. The Necklaces package provides the sets $N_m(n)$ and $B_m(n)$.

**Example 2.5.** In Example 1.1, for $X = \{0,1\}$, we have

\[
N[3,1] = \{0001\}, N[1,3] = \{0111\}, N[2,2] = \{0011, 0101\}
\]
\[
N[0,4] = \{1111\}, N[4,0] = \{0000\}.
\]

Using (2.1) and Example 2.4, it is straightforward to get

\[
N_2(4) = \{0000, 0001, 0011, 0101, 0111, 1111\}.
\]

### 3. Package Necklaces in use

Given a vector in input, the `cNecklaces` (resp. `cBracelets`) function generates the necklace (resp. bracelet) whose this vector belongs to. When equal to TRUE, the input flag `bOut` permits to enumerate the vectors in output, and to print the elements of the class into a more compact form. In addition, the cardinality of the class is provided by the integer in parenthesis on the right of the last vector in the output list.

**Example 3.1.** To print the elements of the necklace whose 001101 belongs to, run

```r
> cNecklaces(c(0,0,1,1,0,1),TRUE)
[0 0 1 1 0 1] (1)
[0 1 0 0 1 1] (2)
[0 1 1 0 1 0] (3)
[1 0 0 1 1 0] (4)
[1 0 1 0 0 1] (5)
[1 1 0 1 0 0] (6)
```

To print the elements of the bracelet whose 1021 belongs to, run

```r
> cBracelets(c(1,0,2,1),TRUE)
[0 1 1 2] (1)
[0 2 1 1] (2)
[1 0 2 1] (3)
[1 1 0 2] (4)
[1 1 2 0] (5)
[1 2 0 1] (6)
[2 0 1 1] (7)
[2 1 1 0] (8)
```

The set $N[i]$ (resp. $B[i]$) of all the representatives of necklaces (resp. bracelets) of a fixed type $i$ (see Definition 1.3) is generated running the function `fNecklaces` (resp. `fBracelets`).

**Example 3.2.** To print the representatives of necklaces of type $i = (2,1,1)$ run

```r
> fNecklaces(c(2,1,1),TRUE)
[1 1 2 3] (1)
[1 1 3 2] (2)
[1 2 1 3] (3)
```

To print the representatives of bracelets of type $i = (2,1,1)$ run

```r
> fBracelets(c(2,1,1),TRUE)
[1 2 3 1] (1)
[1 3 2 1] (2)
[2 1 3 1] (3)
```
Note that the elements of the class \(1 1 2 3\) and \(1 1 3 2\) (obtained from the first calling) are merged in just one bracelet (obtained from the second calling) with representative 1123. Indeed in the necklace \(1 1 2 3\) = \{1123, 1231, 2311, 3112\}, the element 1231 \(\in \langle 1 1 2 3 \rangle\) is obtained from 1321 \(\in \langle 1 1 3 2 \rangle\) = \{1132, 1321, 3211, 2113\} by reversing the order and likewise for 2311, 3112, 1123 \(\in \langle 1 1 2 3 \rangle\) from 1132, 2113, 3211 \(\in \langle 1 1 3 2 \rangle\) respectively. The input vector \(i = (2, 1, 1)\) refers to the alphabet \(X = [3]\) and identifies 4-length strings with 1 occurring twice, 2 and 3 occurring once. To change the alphabet in \(X = \{0, 1, 2\}\)—for example—add a last parameter equal to 0, as shown in the following example.

Finally, the \texttt{Necklaces} (resp. \texttt{Bracelets}) function returns the list of representatives of all necklaces (resp. bracelets) of length \(n\) over the alphabet \(X = [m]\). The function returns a vector: the second component contains the list and the first component indicates its length, as the following example shows.

**Example 3.3.** To get the number of representatives of binary necklaces of length 4, run

```r
defNecklaces(c(2,1,1),TRUE)
[1] 6
```

To get the elements of \(N_2(4)\) run

```r
fNecklaces(c(2,2),TRUE)
[1] 1
```

```r
for (w in v[[2]]) cat("[",unlist(w),"]\n")
[1] 1 1 1 1
[1] 1 1 1 2
[1] 1 1 2 2
[1] 1 2 1 2
[1] 1 2 2 2
[2] 2 2 2 2
```

In agreement with (2.1), the following strategy is implemented in the \texttt{Necklaces} (resp. \texttt{Bracelets}) function. First the \texttt{mkT} function generates all the counting vectors of the integer 4 in 2 parts, that are \(i \in \{2, (2, 1, 1), (1, 3), (3, 1), (0, 4), (4, 0)\}\) see Example 2.4. Then the \texttt{fNecklaces} (resp. \texttt{fBracelets}) function is used aiming to find all the representatives of type \(i\). Thus for example one has

```r
fNecklaces(c(2,2),TRUE)
[1] 1 1 2 2
```

```r
fNecklaces(c(1,3),TRUE)
[1] 1 2 2 2
```
and so on.

Note that the cardinality of the sets $N_m(n)$ and $B_m(n)$ can also be calculated by running the necklace and bracelet functions of the numbers package (Borchers 2021).

**Example 3.4.** To compute the number of binary necklaces and bracelets of length 6 given in Example 1.1, run

```r
> Necklaces(6,2)[[1]]
[1] 14
> Bracelets(6,2)[[1]]
[1] 13
```

```r
> library(numbers)
> necklace(2, 6)
[1] 14
> bracelet(2, 6)
[1] 13
```

### 4. Applications

In most cases, a necklace has full size, which is equal to the length $n$ of a string. But, there are necklaces of size less than $n$. In Example 1.1, the necklace \{1000,0100,0010,0001\} has full size, but \{1010, 0101\} has size 2. In such a case the necklace is said periodic as its representative is a periodic string. Recall that a string $z$ is periodic if the string consists in a repetition of a single substring $\beta$. Otherwise, the string is aperiodic. For example 0000 is periodic of period 4, as 0000 consists in a repetition four times of the substring 0. We might write $0000 = 0^4$. Similarly 0101 is periodic of period 2 as $0101 = (01)^2$. In contrast, the string $0011 = (0011)^1$ is aperiodic. A necklace is said aperiodic if its representative is an aperiodic string. Next subsec. 4.1 gives an example of construction of aperiodic Necklaces using the Necklaces package. In subsec. 4.2, as a further possible application of the necklaces package, de Bruijn sequences of order $n$ are generated. In such a case we use the aperiodic prefix of the representative of a necklace. Recall that the aperiodic prefix of a string $x$ is its substring $\beta$ such that $x = \beta^r$ for some positive integer $r$. Hence for 0000 the aperiodic prefix is 0, for 0011 the aperiodic prefix is the same string 0011.

#### 4.1. Lyndon words

A Lyndon word is an aperiodic necklace, that is a full size necklace with an aperiodic string as representative (Flajolet and Sedgewick 2009). These words are involved in the expression of (1.1). Additional applications involve compression analysis (Nakashima et al. 2013), string matching (Crochemore and Perrin 1991) and bioinformatics (Clare et al. 2019).

To generate all Lyndon words of length $n$ over the alphabet $X = \{m\}$, run the `cNecklaces` function and discard all the necklaces whose size is less than $n$. The R code is:

```r
LyndonW <- function(n=1,m=2,bOut = FALSE, fn = 1){
  if (n<1 || m<1) stop("n and m must be positive integers");
  i<-0; oL<-list(); iL<-Necklaces(n,m,fn);
  for (u in iL[[2]]) {
    if (length(cNecklaces(u))===n)
      {i<-i+1; oL[[i]]<-u};
  }
  oL<-lSort(oL);
  if (!bOut) return(oL);
  i<-1; for (w in oL) {cat("[",unlist(w),"] (",i,"
  n"));i<-i+1;
  }
}
```
The LyndonW function calls the lSort function, taking in input a list of vectors and returning the same list ordered in lexicographical way. By default, the alphabet is $[m]$. Different alphabets $\{fn, fn+1, ..., fn+m-1\}$ can be considered, changing the value of the input variable fn. In particular if bOut = TRUE the output is given in a compact form.

Example 4.1. To generate all Lyndon words of length 3 over the alphabet $[3] = \{1,2,3\}$, run

```
> LyndonW(3,3,TRUE)
[1] 1 2 1
[2] 1 3 2
[3] 2 3 3
```

4.2. de Bruijn sequences

de Bruijn sequences are cyclic $m$-ary strings in which every possible string of length $n$ of the alphabet occurs exactly once as a substring (Fredricksen and Maiorana 1978). The following example clarifies the definition.

Example 4.2. For $n = 4$ and $X = \{0,1\}$ ($m = 2$), the minimum (in lexicographic order) de Bruijn sequence is given in Figure 1.

Table 1 shows all the $2^4$ binary strings (rows) of $\{0,1\}^4$ occurring (exactly once) as substring of the de Bruijn sequence of Figure 1. In particular the columns of Table 1 shows where each string

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{figure1.png}
\caption{A cyclic binary de Bruijn sequence of 4-length strings.}
\end{figure}

\begin{table}
\centering
\caption{The occurrence of all $2^4$ strings of the set $\{0,1\}^4$ as substring of the de Bruijn sequence of Figure 1.}
\begin{tabular}{cccccccccccc}
\hline
& 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{tabular}
\end{table}
is located in the sequence: in bold the representatives of the binary necklaces of length 4. The de Bruijn sequence is given in the first row and the last 3 digits have been added to reproduce the property of circularity.

Applications of de Bruijn sequences can be found in various fields. For example, computer programs make use of de Bruijn sequences when they play chess. A chess board is a square of $8 \times 8$ boxes that can be represented with integer numbers from 1 to 64 stored in a binary de Bruijn sequence of length 6. In chess programming (Leiserson, Prokop, and Randall 1998) the pieces of a given type are represented as a 64-bit sequence and each bit flags the presence or absence of the piece type on a particular square of the chessboard (Frey and Atkin 1988). In robotics these sequences are used to move the eye-camera of a robot. For example, if the space around the camera is labeled circularly as in Figure 1, then selecting a binary 4-length string of Table 1 a direction is identified in which moving the camera. Lastly, a further application is within modern DNA sequencing techniques. Usually the genome is broken down into small pieces (short reads) because it cannot be sequenced all at once. Available methods can only handle short DNA fragments from which the original sequence is reassembled, as if reconstructing a jigsaw puzzle. Various reassembly methods exist, among which de Bruijn sequences are widely used because they represent overlapping strings useful to manage overlapped short reads (Compeau, Pevzner, and Tesler 2011).

There are various methods and tools to generate de Bruijn sequences (see, e.g., Ralston 1982). Necklaces are one of these tools relied on a concatenation of their aperiodic prefix as the following example shows.

**Example 4.3.** Consider again Example 4.2 with $n = 4, m = 2$ and $X = \{0, 1\}$. To construct the de Bruijn sequence in Figure 1, first generate the set $N_2(4)$ of all the representatives of binary necklaces of length 4, that is

```r
> v = Necklaces(4, 2, 0)
> for (w in v[[2]]) cat("[", unlist(w), "]\n")
[0 0 0 0 ]
[0 0 0 1 ]
[0 0 1 1 ]
[0 1 0 1 ]
[0 1 1 1 ]
[1 1 1 1 ]
```

Then concatenate the aperiodic prefix of each representative (see Table 2) in the resulting sequence 0000100110101111.

The `sBruijn` function generates the minimum string in lexicographic order using necklace concatenation (Sawada, Williams, and Wong 2016) as described in Example 4.3. The R code is

```r
sBruijn <- function(n=1, m=2, fn=0, bSep=FALSE){
  if (n<1) stop("n and m must be positive integers");
  sB<-"";sSep<-ifelse(bSep,"","");
  for (u in lSort(Necklaces(n,m,fn)[[2]])){
    apPrefix<-length(cNecklaces(u));
    sB<-paste0(sB, sSep, paste(u[1:apPrefix], collapse=""))
  }
  noquote(ifelse(bSep, substr(sB, 2, nchar(sB)), sB))}
```

In the code, first the function `Necklaces` is ran aiming to generate all the representatives of $m$-ary necklaces of length $n$. Then, after an increasing lexicographic ordering, the output is set by extracting their aperiodic prefix and concatenating them in a final sequence. The alphabet is
Table 2. In the first column the representatives of all binary necklaces of length 4 are given.

| Necklace | Aperiodic prefix |
|----------|------------------|
| 0 0 0 0  | 0                |
| 0 0 0 1  | 0001             |
| 0 0 1 1  | 0011             |
| 0 1 0 1  | 01               |
| 0 1 1 1  | 0111             |
| 1 1 1 1  | 1                |

In the second column their aperiodic prefix is given.

\{0, 1, ..., m – 1\} by default. Different alphabets such as \{fn, fn + 1, ..., fn + m – 1\} can be considered, changing the value of the input variable fn.

**Example 4.4.** To get the sequence in Figure 1 run

```r
> sBruijn(4,2,0)
[1] 0000100110101111
```

Setting the input variable `bSep = TRUE`, a separator is inserted among the output blocks to show the concatenation of the aperiodic prefixes.

```r
> sBruijn(4,2,0,TRUE)
[1] 0.0001.0011.01.0111.1
```

**Example 4.5.** From the de Bruijn sequence, it is possible to generate the original overlapping strings as follows. Refer again to Table 1. Set \(n = 4\) to manage 4-lenght strings and generate the binary de Bruijn sequence of Example 4.4. Then, extract the first \(n – 1 = 3\) digits of the sequence, using the `substr` function and concatenate these digits at the end of the sequence, after conversion to characters, with the `paste0` function. In such a way, the first row of Table 1 is reproduced. Lastly, repeatedly use the `substr` function to print all the `2^4` substrings of length 4. The R code is the following:

```r
> n = 4; s <- sBruijn(n,2); s <- paste0(s,substr(s,1,n-1));
> for (i in 1:(nchar(s)-(n-1))) print(noquote(substr(s,i,i+(n-1))));
```

**Example 4.6.** Suppose \(n = 2, m = 3\) and \(X = [3]\). Then the set \(N_3(2)\) of all the representatives of ternary necklaces of length 2 is

```r
> v = Necklaces(2,3)
> for (w in v[[2]]) cat("[",unlist(w),"]\n")
[1] 1 1
[1] 1 2
[1] 1 3
[1] 2 2
[1] 2 3
[1] 3 3
```

The concatenation consists in joining the aperiodic prefixes 1,12,13,2,23,3.

```r
> sBruijn(2,3,1,TRUE)
[1] 1.12.13.2.23.3
```
The final sequence is

> sBruijn(2,3,1)
[1] 112132233

Note that all possible strings of length 2 are 11, 12, 21, 13, 32, 22, 23, 33 and they appear exactly once as substring of 112132233. To detect 31, the sequence 112132233 should be closed circularly. Use the R code of Example 4.5 to recover all the $3^2$ strings of $\{1,2,3\}^2$ from the de Bruijn sequence.

5. Concluding remarks

The Necklaces package was developed to list the elements of necklaces and bracelets under various hypothesis. Further work would include the development of an algorithm in R to compute numerically (1.1) and to list these sequences under more general hypothesis. The results in this note were obtained using R 3.4.1. The kStatistics package and the Necklaces package are available from the Comprehensive R Archive Network (CRAN) at https://CRAN.R-project.org/.

References

Alon, N., D. Elboim, J. Pach, and G. Tardos. 2021. Random necklaces require fewer cuts. arXiv preprint arXiv: 2112.14488.

Borchers, H. W. 2021. Numbers: Number-Theoretic Functions. R package version 0.8-2.

Clare, A., J. W. Daykin, T. Mills, and C. Zarges. 2019. Evolutionary search techniques for the lyndon factorization of biosequences. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, 1543–50. doi:10.1145/3319619.3326872.

Compeau, P. E., P. A. Pevzner, and G. Tesler. 2011. Why are de Bruijn graphs useful for genome assembly? Nature Biotechnology 29 (11):987–91. doi:10.1038/nbt.2023.

Crochemore, M., and D. Perrin. 1991. Two-way string-matching. Journal of the ACM 38 (3):650–74. doi:10.1145/116825.116845.

De Felice, C., R. Zaccagnino, and R. Zizza. 2017. Unavoidable sets and circular splicing languages. Theoretical Computer Science 658:148–58. doi:10.1016/j.tcs.2016.09.008.

Di Nardo, E. 2014. On a symbolic representation of non-central Wishart random matrices with applications. Journal of Multivariate Analysis 125:121–35. doi:10.1016/j.jmva.2013.12.001.

Di Nardo, E., and G. Guarino. 2022. kStatistics: Unbiased estimates of joint cumulant products from the multivariate fà di bruno’s formula. The R Journal 14 (2):209–29. doi:10.32614/RJ-2022-033.

Flajolet, P., and R. Sedgewick. 2009. Analytic combinatorics. New York, NY: Cambridge University Press.

Fredricksen, H., and J. Maiorana. 1978. Necklaces of beads in k colors and k-ary de bruijn sequences. Discrete Mathematics 23 (3):207–10. doi:10.1016/0012-365X(78)90002-X.

Frey, P. W., and L. R. Atkin. 1988. Creating a chess player. In Computer games I, 226–324. New York, NY.

Harary, F. 1994. Pólya’s enumeration theorem. In Graph theory, 180–4. Reading, Massachusetts: Addison-Wesley.

Kociumaka, T., J. Radoszewski, and W. Rytter. 2014. Computing k-th lyndon word and decoding lexicographically minimal de bruijn sequence. In Symposium on Combinatorial Pattern Matching, 202–11. Springer.

Leiserson, C. E., H. Prokop, and K. H. Randall. 1998. Using de bruijn sequences to index a 1 in a computer word. Available on the Internet from 3 (5), http://supertech.csail.mit.edu/papers.html.

Mallows, C., and L. Shepp. 2008. The necklace process. Journal of Applied Probability 45 (1):271–8. doi:10.1239/jap/1208358967.

Di Nardo, E., and G. Guarino. 2013. A new algorithm for computing moments of complex non-central wishart distributions. Maple algorithm.

Di Nardo, E., and G. Guarino. 2021. kStatistics: Unbiased Estimators for Cumulant Products and Faà Di Bruno’s Formula. R package version 2.1.

Nica, A., and R. Speicher. 2006. Lectures on the combinatorics of free probability, volume 13. New York, NY: Cambridge University Press.
Ralston, A. 1982. De bruijn sequences—a model example of the interaction of discrete mathematics and computer science. *Mathematics Magazine* 55 (3):131–43. doi:10.2307/2690079.

Ruskey, F., and J. Sawada. 1999. An efficient algorithm for generating necklaces with fixed density. *SIAM Journal on Computing* 29 (2):671–84. doi:10.1137/S0097539798344112.

Sawada, J., A. Williams, and D. Wong. 2016. Generalizing the classic greedy and necklace constructions of de Bruijn sequences and universal cycles. *The Electronic Journal of Combinatorics* 23 (1):1–24. doi:10.37236/5517.

Schwartz, M., and T. Etzion. 1999. The structure of single-track gray codes. *IEEE Transactions on Information Theory* 45 (7):2383–96. doi:10.1109/18.796379.

Tomohiro, I., Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. 2013. Efficient lyndon factorization of grammar compressed text. In *Combinatorial pattern matching*, ed. J. Fischer and P. Sanders, 153–64. Berlin, Heidelberg: Springer.

Vitkovskiy, A., P. Christodoulides, and V. Soteriou. 2012. A combinatorial application of necklaces: Modeling individual link failures in parallel network-on-chip interconnect links.