Chapter 7

DeBruijn Cycles and Relatives

Dominoes is a simple game, enjoyed by children and adults alike. There are 28 domino pieces, each of which consists of an unordered pair of numbers taken from the set $\{0, 1, 2, 3, 4, 5, 6\}$ and printed or stamped as dots on a 1-by-2 piece of wood or plastic. Each of two players takes turns alternately placing dominoes, subject to the restriction that the new domino must abut a domino with the same number (there are various other rules that we ignore). A sample game which used all pieces is shown in Figure 7.1. As is customary, dominoes with identical numbers are placed perpendicular to the other dominoes. Now imagine a solitary player who simply wants to create a configuration like that shown in the figure. We consider a couple of questions. First, does it matter which pieces have already been placed? Can you blindly place the pieces and still know that a successful completion is assured? Secondly, note that we have created a kind of Gray code, in the sense that successive dominoes satisfy a natural closeness condition, but trying to model the problem as one of finding a Hamilton path in a graph with 28 vertices is not as straightforward as you might imagine. Give it a try!

Here's a more serious topic. The following circular bitstring has a rather curious property.

$$00011101 \tag{7.1}$$

Consider the set of all of its contiguous substrings of length three. There are eight such substrings and they are all distinct: 000, 001, 011, 111, 110, 101, 010, 100. In other words, they are all 8 bitstrings of length three. In general it is natural to wonder whether there is a circular k-ary string of length k^n with the property that all of its k^n length n contiguous substrings are distinct. Such circular strings do indeed exist and have come to be known as *De Bruijn Cycles*. De Bruijn cycles have found use in coding and communications, as pseudo-random number generators, in the theory of numbers, in digital fault testing, in the

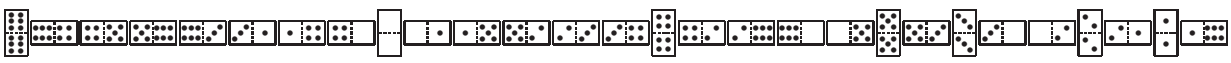

Figure 7.1: Example (maximal) domino game.

design of Sanskrit memory wheels[1], and by illusionists in mind-reading effects.

What ties these two topics together is that they are more naturally modelled as questions about Eulerian cycles in graphs rather than about Hamilton paths in graphs. One of our main aims in this chapter is to show that De Bruijn cycles always exist, and how to generate them quickly. Along the way we encounter other important combinatorial objects, such as necklaces, Lyndon words, and primitive polynomials over finite fields.

7.1 Eulerian Cycles

Certain combinatorial Gray code questions are more naturally posed as Eulerian cycle questions rather than as Hamiltonian cycle questions. Recall that an Eulerian cycle in a (multi)graph is a cycle that includes every edge exactly once. There is a simple characterization of Eulerian graphs, namely as given in Lemma 2.6: a connected (multi)graph is Eulerian if and only if every vertex has even degree.

Now back to the domino problem. Consider the complete graph K_7 with vertices labelled $0, 1, 2, 3, 4, 5, 6$ and with the self-loops $\{i, i\}$ added to each vertex i. There are 28 edges in this graph and each edge corresponds to a domino. An Eulerian cycle corresponds to a maximal domino game like that shown in Figure 7.1. Eulerian cycles clearly exist since each vertex has even degree 8.

7.1.1 Generating an Eulerian cycle

Let G be an directed Eulerian multigraph with n vertices. In this section we develop an algorithm that will generate an Eulerian cycle in in G. Along the way we will discover a nice formula for the number of Eulerian cycles and a CAT algorithm for generating all Eulerian cycles of G.

If G is a directed multigraph then $\overline{G}$ is the directed multigraph obtained by reversing the directions of the edges of G (i.e., for each edge (u, v) of G, there is a corresponding edge (v, u) in $\overline{G}$.

There is a simple algorithm for finding an Eulerian cycle in G given an in-tree rooted at some vertex r. We simply start at r and successively pick edges subject to the restriction that an edge of T is *not* used unless there is no other choice. A simple implementation of this rule is contained in Algorithm 7.1, and an example of its use is to be found in Figure 7.2.

At step (E1) we find a spanning out-tree T of $\overline{G}$ that is rooted at r. In general there may be several spanning out-trees rooted at r and it doesn't matter which one is used; depth-first search is a convenient and efficient way to find such a tree. The tree T is a spanning in-tree of G. We now modify (at line (E2)) the adjacency lists of G so that the unique edge $(u, v) \in T$ on the list for u is at the end of the list. We now say that the adjacency lists are *extreme* with respect to T. The remaining lines simply extract edges from the adjacency lists, destroying the lists in the process.

The running time of this algorithm is $O(m)$ where m is the number of edges in G. Since G is Eulerian, $m \geq n$. The depth-first search at line (E2) runs in time $O(n + m)$. The

[1]The nonsense Indian word yamátárájabhánasalagám is a mnemonic way of remembering the sequence (7.1), where an accented vowel represents a 1 and an unaccented vowel represents a 0. This word was used by medieval Indian poets and musicians as an aid in remembering all possible rythms. (There are 10 bits because the last two have been wrapped around.)

(E1) Use depth-first-search to find a spanning out-tree T of $\overline{G}$ rooted at r.
(E2) Compute adjacency lists `adj` of G that are extreme with respect to T.
(E3) $u := r$;
(E4) **while** $adj[u] \neq$ **null** **do**
(E5) $v := adj[u].vert$; $adj[u] := adj[u].next$;
(E6) Output((u, v));
(E7) $u := v$;

Algorithm 7.1: Algorithm to find an Eulerian cycle in a directed multigraph.

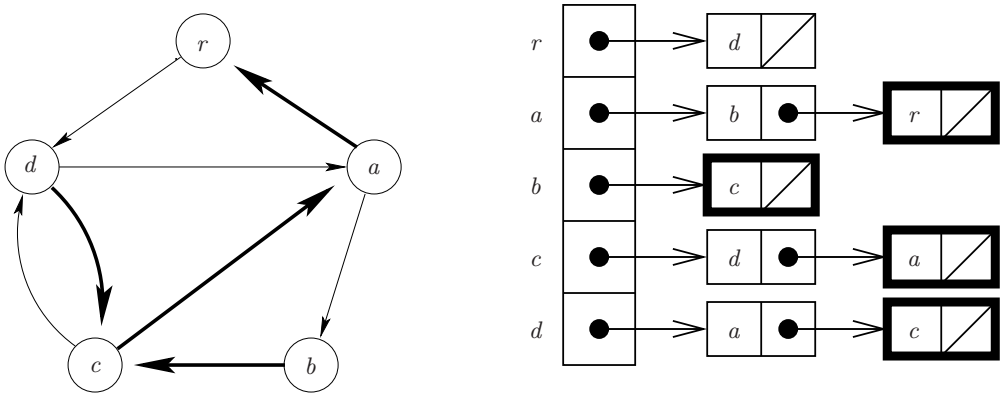

Figure 7.2: Example of finding an Euler cycle in a directed multigraph. (a) A directed multigraph G, with spanning in-tree found by depth-first search of $\overline{G}$ shown in bold, and (b) extreme adjacency lists with respect to T, with edges of T thicker. The cycle produced by this example is $r, d, a, b, c, d, c, a, r$.

computation of $\overline{G}$ takes time $O(m)$ the remaining computation (line (E3-E6)) takes time $O(m)$ since the body of the while loop at line (E3) is executed exactly m times.

THEOREM 7.1 *Given a directed Eulerian multigraph G, Algorithm 7.1 outputs a list of edges along an Eulerian cycle of G.*

PROOF: Since G is Eulerian, the path P produced by the algorithm must end at r. Imagine that there is some edge (v, w) that is not in P. Since the algorithm terminated it must be the case that $v \neq r$. Clearly, any edge on the adjacency list for v that follows (v, w) must also not be in P. Thus, because the edge lists are extreme with respect to T, we may assume that (v, w) is in T. Since $G - P$ is balanced, there is an edge (u, v) also not in P, which again we can take to be in T. Continuing in this manner we obtain a path of edges in $(G - P) \cap T$ that terminates at r. But then, since $G - P$ is balanced, it must contain an edge (r, q), contrary to the terminating condition of the algorithm. □

How many Eulerian does a connected, balanced multigraph G have? In answering this question we regard an Eulerian cycle as being a *circular* list of edges; the edge that starts the list is immaterial. The answer is provided by our algorithm. Clearly, different in-trees T produce different cycles, as do different adjacency lists that are extreme with respect to T. A graph G has $\tau(G)$ different spanning in-trees rooted at a given vertex r and there are $(d^+(v) - 1)!$ ways of arranging the adjacency list of v so that it is extreme with respect to T. Thus it is plausible that the number of Eulerian cycles in G is

$$\tau(G) \prod_{v \in V} (d^-(v) - 1)!. \tag{7.2}$$

To prove (7.2) we must show that we can recover the adjacency lists and tree T from an Eulerian cycle C. Fix an edge (r, s) to be the first on the cycle. Define the adjacency list for vertex v to simply be the edges of the form (v, w) in the order that they are encountered on C. With these adjacency lists lines (E3-E8) will produce the cycle C. To finish the proof we need to show that the the collection of edges

$$S = \{(v, v') \in E \mid v \neq r \text{ and } (v, v') \text{ is the last occurence of } v \text{ on } C\}$$

is an in-tree rooted at r. The set S contains $n - 1$ edges; we must show that it forms no cycles. Assume to the contrary that such a cycle X exists and let (y, z) be the first of its edges that occur on C, and let (x, y) be the previous edge on X. Unless $y = r$, there is some edge (y, z') that follows (x, y) on C, in contradiction to the way (y, z) was chosen. But we cannot have $y = r$ either since then $(y, z) = (r, z)$ would be in S.

How fast can we generate all Eulerian cycles in a graph? We need to generate permutations of edges on adjacency lists. There are many CAT algorithms for generating permutations (e.g., Algorithms ???, ???, and ???). We also need to generate spanning trees of a graph. This topic is taken up in Chapter ???. There are CAT algorithms for generating spanning trees of undirected graphs, but what about spanning in-trees of directed graphs? **!!! Is this a simple reduction or a research problem ???**

7.1.2 An Eulerian Cycle in the Directed n-cube

Given the central role played by hypercubes in the previous chapters, it is fitting that the next problem is one that can be modeled on the n-cube. We must admit, however, that it is

a rather distant relative of De Bruijn cycles. The solution of the following problem of Bate and Miller [32] is useful in circuit testing. Generate a cyclic sequence of $n2^n$ bitstrings of length n with the following three properties: (a) Each distinct bitstring must appear exactly n times; (b) each bitstring must differ by exactly one bit from the previous bitstring; (c) each possible *pair* of successive bitstrings must appear exactly once. An example of such a sequence for $n = 3$ is shown below.

000, 001, 011, 001, 101, 111, 101, 011, 000, 100, 110, 110,

101, 100, 000, 010, 110, 111, 011, 111, 110, 010, 011, 010,

The three occurences of 011 are underlined. Note that the three bitstrings which follow them are all distinct. Let $\vec{Q}_n$ be the *directed n-cube*; every edge of the n-cube Q_n is replaced by two directed edges, one in each direction. The following theorem is obviously true since any digraph for which the in-degree of each vertex is equal to its out-degree is Eulerian. Our interest in this theorem lies in its proof, which shows an explict construction of the Eulerian cycle without building the graph.

THEOREM 7.2 *There is an Eulerian cycle in the directed n-cube $\vec{Q}_n$ for all $n > 0$.*

PROOF: We argue by induction on n. The cycle will be expressed as a list of $n2^n + 1$ vertices, starting and ending at 0^n. The list for $n = 1$ is $0, 1, 0$.

Let $\mathcal{L}$ denote the list of bitstrings of the Eulerian cycle for $n - 1$. Produce from this a list $\mathcal{L}'$ obtained by doing the following steps in order.

1. Append a 0 to each bitstring of $\mathcal{L}$.

2. Replace the *first* occurence of a *non-zero* bitstring $X0$ by the 3 bitstrings $X0, X1, X0$.

3. Concatenate a second copy of $\mathcal{L}$ with a 1 appended to each bitstring.

4. Concatenate to the list a final bitstring of 0's.

It is easy to visualize what this algorithm is doing if $\vec{Q}_n$ is recursively thought of as two copies E_0 and E_1 of $\vec{Q}_{n-1}$, where one copy contains those vertices that end with a 0 and the other with those that end with a 1. The only other edges are the 2-cycles of the form $X0, X1$. It is these 2-cycles that get added to the Eulerian cycle by step 2; call the result E_0'. They are added the first time X is encountered in E_0, except if X is 0. The 2-cycle $\mathbf{0}0, \mathbf{0}1$ is used to join E_0' and E_1. □

For $n = 2$ the cycle produced is

$$00, 10, 11, 10, 00, 01, 11, 01, 00.$$

The table below shows the Eulerian cycle for $n = 3$.

$n=2$	$E_0'0$	$E_1 1$
00	000	001
10	100,101,100	101
11	110,111,110	111
10	100	101
00	000	001
01	010,011,010	011
11	110	111
01	010	011
00	000	001
		000

How can we develop and efficient algorithm based on this proof? The main difficulty comes from deciding which is the first occurence of a non-zero string. A bitstring $\mathbf{b}$ is uniquely identified in the list by the bit that changes, call it c, in obtaining the successor of $\mathbf{b}$, call it $\mathbf{b}'$. If, given $\mathbf{b}$ and c, we can specify c', the bit that changes in obtaining the successor of $\mathbf{b}'$, then we will be able to produce the entire list. Initially $\mathbf{b} = \mathbf{0} = 0^n$ and $c = 1$; these can also be used for the terminating conditions. Below we give rules for obtaining c' from $\mathbf{b}$ and c. These rules will be justified in the paragraphs to follow. In these rules x denotes a single bit (a "don't care"), $\mathbf{X}$ is a generic bitstring, and $\mathbf{0}$ is a string of 0's.

A. If $c = n$ and $\mathbf{b} = \mathbf{X}0$ then $c' = 1$ if $\mathbf{X} = \mathbf{0}$ and $c' = n$ if $\mathbf{X} \neq \mathbf{0}$.

B. If $c = n$ and $\mathbf{b} = \mathbf{X}x1$ then $c' = 1$ if $\mathbf{X} = \mathbf{0}$ and $c' = n-1$ if $\mathbf{X} \neq \mathbf{0}$.

C. If $c = n-1$ and $\mathbf{b} = \mathbf{0}1x$ then $c' = n$.

D. If $c = n-k$ $(k \geq 1)$ and $\mathbf{b}$ has more than k trailing 0's, then $c' = n$.

E. If none of the rules A-D above apply, then recursively apply them to the string consisting of the first $n-1$ bits.

The last transition is from $0^{n-1}1$ to 0^n. Thus when the sequence E_0' or E_1 is completed, the next transition is in position n. So if $\mathbf{b} = \mathbf{0}1x$ and $c = n$, then c' should be n. This explains rule C.

Whenever bit n changes from 0 to 1 ($\mathbf{b} = \mathbf{X}0$ and $c = n$), then it changes back to 0 ($c' = n$) unless $\mathbf{X} = \mathbf{0}$. If $\mathbf{X} = \mathbf{0}$, then a new sequence on $n-1$ bits is starting and c' should be 1. This explains rule A.

In expanding E_0 to E_0' and a bitstring is generated for the first time, c' should be n. This happens whenever bit $n-1$ is about to change from a 0 to a 1 ($\mathbf{b} = \mathbf{X}00$ and $c = n-1$), or whenever bit $n-1$ is a 0 and a pattern on the first $n-2$ bits is about to appear for the first time. Recursively, this gives rise to the patterns $\mathbf{b} = \mathbf{X}000$ and $c = n-2$, and more generally to $\mathbf{b} = \mathbf{X}0^{k+1}$ and $c = n-k$. This explains rule D.

We now consider the case in which bit n changes from 1 to 0. This happens at the penultimate bitstring (i.e., when $\mathbf{b} = 0^{n-1}1$ and $c = n$). It also occurs when finishing up a two cycle; that is, when bit n has changed from 0 to 1 and is now immediately changing from 1 to 0. The sequence on $n-1$ bits must now be resumed. This implies that rule D was used, followed by rule A, and that the following bitstrings have just been generated.

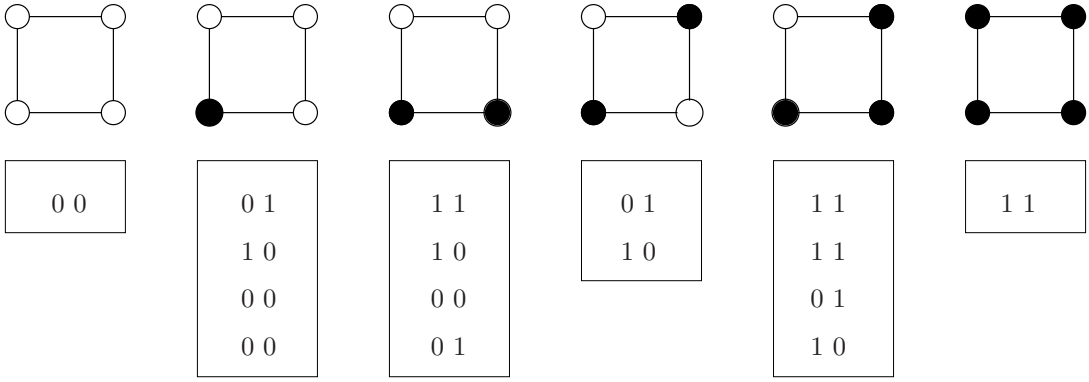

Figure 7.3: The six two-color necklaces with 4 beads. The necklace strings are shown in bold. Each equivalence class of strings under rotation is boxed.

b	c	
$\mathbf{X}0\ 0\ldots0\ 0$	$n-k$	
$\mathbf{X}1\ 0\ldots0\ 0$	n	
$\mathbf{X}1\ 0\ldots0\ 1$	n	current
$\mathbf{X}1\underbrace{0\ldots0}_{k-1}0$	?	$\mathbf{b}', c'$

The interrupted sequence on $n-1$ bits must now be resumed. The two lines where $c = n$ may be ignored, as may bit n. The first bitstring of the four should determin c'. This explains rule B.

Rules A-D cover all cases where $c = n$ or $c' = n$. If bit n is not involved, then we are generating the Eulerian cycle in $\vec{Q}_{n-1}$. This explains rule E.

A straighforward implemenation of these rules overcomes the exponential space obstacle, reducing it to $O(n)$. Time is also clearly $O(n)$ per bitstring produced. Unfortunately, the rules don't lend themselves to a CAT algorithm since about $n^2 2^{n-1}$ applications of the rules (including the recursive applications) are required to generate the Eulerian cycle.

However, a circuit may be designed, based on a refinement of these rules, which produces each bitstring in constant time from its predecessor, by operating in parallel, a similar circuit used for each position.

7.2 Necklaces

Mathematically, a necklace is usually defined as an equivalence class of strings under rotation. This definition is not exactly in accord with our intuition about what constitutes a real necklace, since we expect to be able to pick up a necklace and turn it over. However, we stick with the mathematical tradition and thus regard 001101 as being a different necklace than 001011, even though one may be obtained from the other by scanning backwards (and then rotating). Our principle goal in this section is to develop an efficient algorithm for generating necklaces.

Figure 7.3 shows the six two-color necklaces with 4 beads.

Recall that $\Sigma_k = \{0, 1, \ldots, k-1\}$, that Σ_k^n is the set of all k-ary strings of length n, that

Σ_k^* is the set of all k-ary strings, and that $\Sigma_k^+ = \Sigma_k^* \setminus \{\varepsilon\}$. Define an equivalence relation $\sim$ on Σ_k^* by $\alpha \sim \beta$ if and only if there exist $u, v \in \Sigma_k^+$ such that $\alpha = uv$ and $\beta = vu$. Instead of defining a *necklace* as an equivalence class we choose to define it as the lexicographically least representative of some equivalences class of the relation $\sim$. The set of all necklaces is denoted $\mathbf{N}$ and $\mathbf{N}_k(n)$ denotes the set of all necklaces of length n over a k-ary alphabet.

$$\mathbf{N}_k(n) \stackrel{\text{def}}{=} \{\alpha \in \Sigma_k^n \mid \alpha \leq \beta \text{ for all } \beta \sim \alpha\} \tag{7.3}$$

For example $\mathbf{N}_2(4) = \{0000, 0001, 0011, 0101, 0111, 1111\}$. The cardinality of $\mathbf{N}_k(n)$ is denoted $N_k(n)$.

Recall that a string α is *periodic* if $\alpha = \beta^k$ where β is non-empty and $k > 0$. If β is aperiodic, then it is called the *periodic reduction* of α. An aperiodic necklace is called a *Lyndon word*. The set of all Lyndon words is denoted $\mathbf{L}$ and $\mathbf{L}_k(n)$ denotes the set of all k-ary Lyndon words of length n.

$$\mathbf{L}_k(n) \stackrel{\text{def}}{=} \{\alpha \in \mathbf{N}_k(n) \mid \alpha \text{ is aperiodic}\}$$

For example, $\mathbf{L}_2(4) = \{0001, 0011, 0111\}$. The cardinality of $\mathbf{L}_k(n)$ is denoted $L_k(n)$.

A word α is called a *pre-necklace* if it is the prefix of some necklace. The set of all pre-necklaces is denoted $\mathbf{P}$ and the set of all k-ary pre-necklaces of length n is denoted $\mathbf{P}_k(n)$.

$$\mathbf{P}_k(n) \stackrel{\text{def}}{=} \{\alpha \in \Sigma_k^n \mid \alpha\beta \in \mathbf{N}(n+m, k) \text{ for some } m \geq 0 \text{ and } \beta \in \Sigma_k^m\}$$

For example $\mathbf{P}_2(4) = \mathbf{N}_2(4) \cup \{0010, 0110\}$. The cardinality of $\mathbf{P}_k(n)$ is denoted $P_k(n)$. We also define $W_k(n)$ to be the number of pre-necklaces of length at most n. These numbers will prove useful in analyzing the algorithms developed below. We define

$$W_k(n) \stackrel{\text{def}}{=} 1 + \sum_{i=1}^{n} P_k(i). \tag{7.4}$$

Let $\alpha = a_0 a_1 \cdots a_{n-1}$ be a string that can be written $\alpha = xy = yx$ where neither of x or y is empty. In other words, α is equal to one of its non-trivial circular shifts. Suppose $x = a_0 a_1 \cdots a_{m-1}$. Then $xy = yx$ implies the equation $a_i = a_{i+m}$, where index addition is taken modulo n. Iterating the equation we obtain

$$a_i \;=\; a_{i+jm} \quad \text{for all } 0 \leq i, j < n.$$

By Lemma 2.2 on page 16, $jm \pmod n$ for $0 \leq j < n$ gives us the multiples of m modulo n, so that

$$\{jm \pmod n : j = 0, 1, \ldots\} \;=\; \{0, d, 2d, \ldots, n-d\},$$

where $d = \gcd(m, n)$. We have now proven the following lemma.

LEMMA 7.1 *If $\alpha = xy = yx$ then*

$$\alpha \;=\; (a_0 a_1 \cdots a_{d-1})^{n/d}, \tag{7.5}$$

where $n = |\alpha|$ and $d = \gcd(|x|, n)$.

Note the following corollary.

COROLLARY 7.1 *If $\alpha = xy = yx$ with $x \neq \varepsilon$, $y \neq \varepsilon$, then α is periodic.*

We now count the objects under consideration. In the expressions below ϕ is the Euler totient function and μ is the Möbius function .

THEOREM 7.3 *The following formulae are valid for all $n \geq 1$, $k \geq 1$:*

$$L_k(n) \;=\; \frac{1}{n}\sum_{d\backslash n} \mu\left(\frac{n}{d}\right) k^d, \tag{7.6}$$

$$N_k(n) \;=\; \frac{1}{n}\sum_{j=1}^{n} k^{\gcd(j,n)} \;=\; \frac{1}{n}\sum_{d\backslash n} \phi(d) k^{n/d}, \tag{7.7}$$

$$P_k(n) \;=\; \sum_{i=1}^{n} L_k(i). \tag{7.8}$$

PROOF: Let $A_k(n)$ be the number of k-ary aperiodic strings of length n. Since every string can be expressed as an integral power of some aperiodic string, it follows that

$$k^n \;=\; \sum_{d\backslash n} A_k(d). \tag{7.9}$$

Now apply Möbius inversion (2.11) to obtain

$$A_k(n) \;=\; \sum_{d\backslash n} \mu(n/d) k^d.$$

Since every circular shift of an aperiodic string is distinct, $A_k(n) = n \cdot L_k(n)$, thereby proving (7.6). In the special case where p is a prime (7.6) or (7.7) imply "Fermat's Little Theorem": $k^{p-1} \equiv 1 \pmod{p}$.

To prove (7.7) we use Burnside's lemma . Necklaces are obtained by having the cyclic group $\mathbb{C}_n$ act on the set of k-ary strings. Let σ denote a left rotation by one position. Then the group elements are σ^j for $j = 0, 1, \ldots, n-1$. We need to determine the number of strings α for which $\sigma^j(\alpha) = \alpha$. By Lemma 7.1 this can occur only if $\alpha = \beta^t$ where $\beta \in \mathbf{L}$ and $|\beta| = \gcd(j, n)$. The number of such strings α is $k^{\gcd(j,n)}$. Thus the number of equivalence classes is $k^{\gcd(j,n)}$ summed over all $j = 1, 2, \ldots, n$ (noting that $\sigma^0 = \sigma^n$) divided by $|\mathbb{C}_n| = n$. The second equality follows from equation (2.10).

Equation (7.8) will be proven later. □

It is also worth noting that, since every necklace has the form β^t where $\beta \in \mathbf{L}$,

$$N_k(n) \;=\; \sum_{d\backslash n} L_k(d). \tag{7.10}$$

Equation (7.10) can also be used to prove (7.7); see exercise 5.

Below is a table of these numbers for the most important case, $k = 2$. Note that $2^n/n$ is sandwiched between $L_2(n)$ and $N_2(n)$, a property that holds for all values of n, and more generally, for all $k \geq 2$.

$k = 2$, $n = 6$

000000 N	000110	001101 L	011011 N
000001 L	000111 L	001110	011101
000010	001001 N	001111 L	011110
000011 L	001010	010101 N	011111 L
000100	001011 L	010110	111111 N
000101 L	001100	010111 L	

$k = 3$, $n = 4$

0000 N	0022 L	0122 L	1111 N
0001 L	0101 N	0202 N	1112 L
0002 L	0102 L	0210	1121
0010	0110	0211 L	1122 L
0011 L	0111 L	0212 L	1212 N
0012 L	0112 L	0220	1221
0020	0120	0221 L	1222 L
0021 L	0121 L	0222 L	2222 N

Figure 7.4: Output of the FKM algorithm (read down columns).

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$L_2(n)$	2	1	2	3	6	9	18	30	56	99	186	335	630	1161	2182
$\lceil 2^n/n \rceil$	2	2	3	4	7	11	19	32	57	103	187	342	631	1171	2185
$N_2(n)$	2	3	4	6	8	14	20	36	60	108	188	352	632	1182	2192
$P_2(n)$	2	3	5	8	14	23	41	71	127	226	412	747	1377	2538	4720
$W_2(n)$	3	6	11	19	33	56	97	168	295	521	933	1680	3057	5595	10315

A simple and elegant algorithm was proposed in Fredricksen and Maiorana [155] and Fredricksen and Kessler [154] to generate the sets $\mathbf{P}_k(n)$ and $\mathbf{N}_k(n)$. We will refer to this algorithm as the *FKM algorithm*.

For a given n and k, the FKM algorithm generates a list, $\mathbf{P}_k(n)$, in lexicographic order, where as usual we use the same notation for both the list and the set. The list $\mathbf{P}_k(n)$ begins with the string 0^n and ends with $(k-1)^n$. For a given α in $\mathbf{P}_k(n)$, the successor of α, $succ(\alpha)$, is obtained from $\alpha = a_1 a_2 \cdots a_n$ as follows.

DEFINITION 7.1 *For $\alpha < (k-1)^n$, $\mathrm{succ}(\alpha) = (a_1 a_2 \cdots a_{i-1}(a_i+1))^t a_1 \cdots a_j$, where i is the largest integer $1 \le i \le n$ such that $a_i < k-1$ and t, j are such that $ti + j = n$ and $j < i$.*

We also define the predecessor function *pred*, where if $succ(\alpha) = \beta$, then $pred(\beta) = \alpha$.

It is shown in [155] that the successor function *succ* sequences through the elements of $\mathbf{P}_k(n)$ in lexicographic order and that $succ(\alpha)$ is a necklace if and only if the i of Definition 7.1 is a divisor of n. We will prove these assertions later in this section.

Figure 7.4 shows the output of the FKM algorithm for $n = 6$, $k = 2$, and $n = 4$, $k = 3$. Lyndon words are followed by an "L"; periodic necklaces are followed by a "N" and the periodic reduction of each necklace is underlined. An iterative implementation is given as Algorithm 7.2.

For $\alpha \in \Sigma_k^*$, let $lyn(\alpha)$ be the length of the longest prefix of α that is a Lyndon word.

```
for j := 0 to n do a_j := 0;
Printlt( 1 );
i := n;
repeat
    a_i := a_i + 1;
    for j := 1 to n - i do a[j + i] := a[j];
    Printlt( i );
    i := n;
    while a_i = k - 1 do i := i - 1;
until i = 0;
```

Algorithm 7.2: The original iterative FKM Algorithm (note: a[0] = 0.)

This function is well-defined since $\Sigma_k \subseteq \mathbf{L}$. More formally,

$$lyn(a_1 a_2 \cdots a_n) \stackrel{\text{def}}{=} \max\{1 \le p \le n \mid a_1 a_2 \cdots a_p \in \mathbf{L}_k(n)\}. \tag{7.11}$$

The next theorem provides useful characterizations of necklaces and Lyndon words.

THEOREM 7.4 *The following conditions characterize the sets* $\mathbf{N}$ *and* $\mathbf{L}$*.*

$\alpha = xy \in \mathbf{N}$ *if and only if* $xy \le yx$*, for all* x, y*.*

$\alpha = xy \in \mathbf{L}$ *if and only if* $xy < yx$*, for all non-empty* x, y*.* (7.12)

PROOF: [$\mathbf{N}_k(n)$] This is just a restatement of the definition (7.3).

[$\mathbf{L}_k(n)$] Suppose $\alpha = xy = yx$ with $xy \ge yx$ and $x \ne \varepsilon$ and $y \ne \varepsilon$. If $xy > yx$ then $\alpha \notin \mathbf{N}_k(n)$, so $\alpha \notin \mathbf{L}_k(n)$. If $xy = yx$ then by Corollary 7.1, α is periodic, so $\alpha \notin \mathbf{L}_k(n)$. Conversely, if $\alpha \in \mathbf{N}_k(n)$ is periodic, say $\alpha = \beta^t$ with $t > 1$, then $\alpha = xy = yx$ where $x = \beta$ and $y = \beta^{t-1}$. □

LEMMA 7.2 *If* $\alpha \in \mathbf{N}$*, then* $\alpha^t \in \mathbf{N}$ *for* $t \ge 1$*.*

PROOF: For $t > 1$, if $\alpha^t = xy$, then yx has the form $\gamma\alpha^{t-1}\delta$ where $\alpha = \delta\gamma$. Since α is a necklace $\delta\gamma \le \gamma\delta$. Thus

$$\alpha^t = (\delta\gamma)^t \le (\gamma\delta)^t = \gamma\alpha^{t-1}\delta,$$

and so α^t must also be a necklace. □

LEMMA 7.3 *If* $\alpha \in \mathbf{L}$ *and* $\alpha = \beta\gamma$ *for some* β *and* γ *with* γ *non-empty, then for any* $t \ge 1$

(a) $\alpha^t\beta \in \mathbf{P}$*, and*

(b) $\alpha^t\beta \in \mathbf{N}$ *if and only if* $|\beta| = 0$*.*

PROOF: (a) Since α is a necklace, both α^t and α^{t+1} are necklaces by the Lemma 7.2. Thus $\alpha^t\beta$ is a pre-necklace and is a necklace if $\beta = \varepsilon$. If $\beta \ne \varepsilon$ then $\alpha = \beta\gamma < \gamma\beta$, since $\alpha \in \mathbf{L}$. Therefore,

$$\alpha^t\beta = (\beta\gamma)^t\beta = \beta(\gamma\beta)^t > \beta(\beta\gamma)^t = \beta\alpha^t,$$

so that $\alpha^t\beta$ is not a necklace. □

LEMMA 7.4 *Let $\alpha = a_1a_2\cdots a_n$ be a string and $p = lyn(\alpha)$. Then $\alpha \in \mathbf{P}$ if and only if $a_{j-p} = a_j$ for $j = p+1, \ldots, n$.*

PROOF: If $a_{j-p} = a_j$ for $j = p+1, \ldots, n$, then $\alpha = \beta^t\delta$ for some $t \geq 1$ where $\beta = a_1a_2\cdots a_p \in \mathbf{L}$ and δ is a prefix of β. Thus $\alpha \in \mathbf{P}$ by Lemma 7.3.

Conversely, assume that $\alpha \in \mathbf{P}$, and let j be the smallest index $j > p$ such that $a_{j-p} \neq a_j$. We will derive separate contradictions, depending whether $a_{j-p} < a_j$ or $a_{j-p} > a_j$. By the definition of j, the string α has the form $\beta^t\delta a_j \cdots a_n$ for some $t \geq 1$ where $\beta = a_1a_2\cdots a_p$ and δ is a proper prefix of β. For some γ, $\beta = \delta\gamma$ where the first symbol of γ is a_{j-p}.

Since $\alpha \in \mathbf{P}$ there is a string ω such that $\alpha\omega \in \mathbf{N}$. If $a_{j-p} < a_j$, then

$$\beta^{t-1}\delta a_j \cdots a_n\omega\delta\gamma \;<\; \beta^{t-1}\delta\gamma\delta a_j \cdots a_n\omega,$$

which implies that $\alpha\omega$ is not a necklace, a contradiction.

If $a_{j-p} < a_j$ then consider the string $\rho = \beta^t\delta a_j$; we will use (7.12) to show that $\rho \in \mathbf{L}$, in contradiction to the definition of $p = lyn(\alpha)$. Write $\rho = xy$ with $x \neq \varepsilon$ and $y \neq \varepsilon$. We consider two cases: (a) β^t is a prefix of x and (b) x is a prefix of β^t.

[Case (a)] Here $x = \beta^t u$ and $y = va_j$ where $uv = \delta$. Since uv is a proper prefix of β there is a $w \neq \varepsilon$ such that $\beta = uvw$. Note that the first symbol of w is a_{j-p}. By (7.12), $uvw < vwu < va_j$. Thus

$$xy = uvw\beta^{t-1}\delta a_j < va_j\beta^t u = yx.$$

[Case (b)] Here $x = \beta^p u$ and $y = v\beta^q\delta a_j$ where $uv = \beta$ and $t = 1+p+q$. If $u = \varepsilon$ or $v = \varepsilon$, then we may assume without loss of generality that $p \geq 1$ and $u = \varepsilon$. Thus

$$xy = \beta^p\beta^q\delta a_j < \beta^q\delta a_j\beta^p = yx,$$

a contradiction. Thus we may assume that neither of u or v is empty and that $\beta = uv < vu$ since $\beta \in \mathbf{L}$. Now if $q = 0$, then since $uv < vu < va_j$,

$$xy = uv\beta^p\delta a_j < v\delta a_j\beta^p u = yx,$$

again a contradiction. The only remaining case is $q > 0$. But then

$$xy = \beta^p uv\beta^q\delta a_j = u(vu)^p(vu)^q v\delta a_j < v(uv)^q\delta a_j\beta^p u = v\beta^q\delta a_j\beta^p u = yx,$$

a contradiction. □

The following theorem leads to a recursive version of the FKM algorithm. It's proof is inherent in the proof of the previous lemma. This theorem is very useful. We are tempted to call it the "Fundamental Theorem of Necklaces"!

THEOREM 7.5 *Let $\alpha = a_1a_2\cdots a_{n-1} \in \mathbf{P}_k(n-1)$ and $p = lyn(\alpha)$. The string $\alpha b \in \mathbf{P}_k(n)$ if and only if $a_{n-p} \leq b \leq k-1$. Furthermore,*

$$lyn(\alpha b) \;=\; \begin{cases} p & \text{if } b = a_{n-p} \\ n & \text{if } a_{n-p} < b \leq k-1. \end{cases}$$

```
procedure gen( t,p : ℕ );
local j : ℕ;
begin
    if t > n then PrintIt( p )
    else
        a[t] := a[t − p]; gen( t + 1, p );
        for j := a[t − p] + 1 to k − 1 do
            a[t] := j; gen( t + 1, t );
end {of gen};
```

Algorithm 7.3: Recursive FKM Algorithm (note: a[0] = 0.)

Algorithm 7.3 is the recursive FKM algorithm. It follows directly from the Theorem 7.5. The initial call is `gen(1,1)`. We assume, as for the iterative FKM algorithm, that $a_0 = 0$. Various types of objects may be produced, depending on `PrintIt(p)`, as shown in the table below.

Sequence type	`PrintIt(p)`
Pre-necklaces ($\mathbf{P}_k(n)$)	Println($a[1..n]$)
Lyndon words ($\mathbf{L}_k(n)$)	**if** $p = n$ **then** Println($a[1..n]$)
Necklaces ($\mathbf{N}_k(n)$)	**if** $n \bmod p = 0$ **then** Println($a[1..n]$)
De Bruijn sequence	**if** $n \bmod p = 0$ **then** Print($a[1..p]$)

The call "Println($a[1..n]$)" prints the array $a[1], a[2], \ldots, a[n]$ on a separate line. Each time `PrintIt` is called a new prenecklace is produced. Since the parameter p to `PrintIt` is the value of $lyn(a[1..n])$, a Lyndon word is produced exactly when $p = n$. By part (b) of Lemma 7.3 a necklace is produced whenever p divides n. De Bruijn cycles will be discussed in the following section.

We can now also understand why Algorithm 7.2, the iterative FKM algorithm, is correct. Consider again the successor function *succ*. It is clear that the rightmost position i that can change is the largest index i for which $a_i < k - 1$. We then increment a_i. What was not so clear was how the remainder of the sequence was to be completed in the lexicographically smallest way. Theorem 7.5 provided the answer.

How fast is the FKM algorithm? We analyze the recursive version; the same conclusions hold for the iterative version. Call the number of nodes in the computation tree $W_k(n)$. From the structure of the algorithm, $W_k(n)$ is equal to the number of prenecklaces of length at most n, as expressed in (7.4).

From the expressions (7.6) and (7.7) we obtain the following bounds.

$$\frac{1}{n}(k^n - (n-1)k^{n/2}) \leq L_k(n) \leq \frac{1}{n}k^n \leq N_k(n) \leq \frac{1}{n}(k^n + (n-1)k^{n/2}) \tag{7.13}$$

We have

$$P_k(n) = \sum_{i=1}^{n} L_k(i) \leq \sum_{i=1}^{n} \frac{1}{i}k^i, \tag{7.14}$$

The equality in (7.14) follows because every prenecklace is obtained as a prefix of β^*, where β is some Lyndon word. Hence

$$W_k(n) - 1 = \sum_{j=1}^{n} P_k(i) \leq \sum_{j=1}^{n} \sum_{i=1}^{j} \frac{1}{i}k^i$$

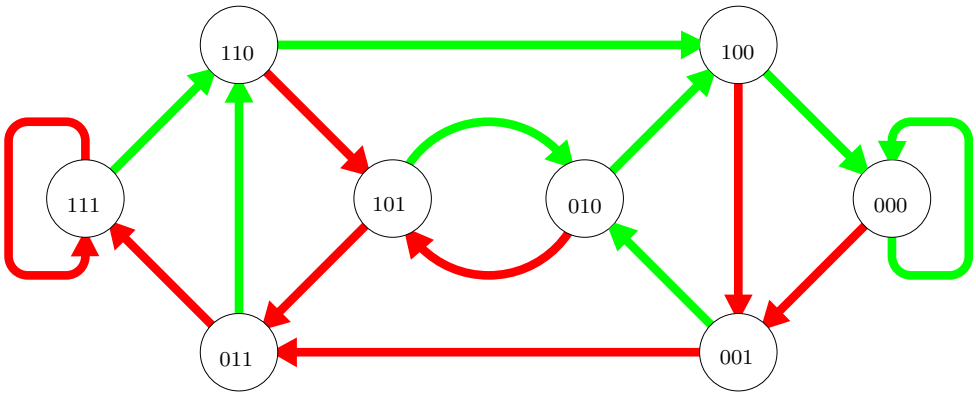

Figure 7.5: The De Bruijn graph for $k = 2$ and $n = 4$.

Thus

$$\frac{W_k(n) - 1}{N_k(n)} \leq \frac{n}{k^n} \sum_{j=1}^{n} \sum_{i=1}^{j} \frac{1}{i} k^i$$

In [388] it is shown that this last expression converges to $(k/(k-1))^2$ as $n \to \infty$. Thus the asymptotic running time per necklace (or per Lyndon word) is constant; the necklace Algorithms 7.2 and 7.3 are both CAT.

7.3 De Bruijn Sequences

An example of a De Bruijn sequence was given at the beginning of this chapter. All De Bruijn sequences arise as Eulerian cycles in a certain graph which we introduce below. The FKM algorithm can be used to generate the lexicographically least Eulerian cycle.

The *De Bruijn graph* , $G_k(n)$ has vertex set consisting of all k-ary strings of length $n-1$; i.e., it is Σ_k^{n-1}. There is a directed edge, labelled b, from $d_1 d_2 \cdots d_{n-1}$ to $d_2 \cdots d_{n-1} b$ for each $b \in \Sigma_k$. Thus the out-degree of each vertex is k and the number of edges is k^n. Figure 7.5 shows $G_2(4)$. An Eulerian cycle in $G_k(n)$ is called a *De Bruijn cycle*. More precisely, the sequence of k^n edge labels is a De Bruijn cycle. A De Bruijn cycle is characterized by the property that each element of Σ_k^n occurs exactly once as a substring on the cycle.

There is a very simple modification of the FKM algorithm that will produce a De Bruijn sequence, as indicated by the appropriate line in the table in the previous section. The idea is to successively concatenate the reduction of each necklace as it is produced by the FKM algorithm. Thus we are concatenating all Lyndon words of length divisible by n in lexicographic order. At first glace it appears most amazing that this algorithm should work. Let's first look at the sequences it produces for the examples given in Figure 7.4. Here's the resulting De Bruijn cycle for $n = 6$ and $k = 2$.

0 000001 000011 000101 000111 001 001011 001101 001111 01 010111 011 011111 1

Here's the resulting cycle for $n = 4$ and $k = 3$.

0 0001 0002 0011 0012 0021 0022 01 0102 0111 0112
0121 0122 02 0211 0212 0221 0222 1 1112 1122 12 1222 2

These are, in fact, the lexicographically smallest De Bruijn cycles.

We now prove that the algorithm is correct. The algorithm is so elegant that one would hope the same of the proof. Unfortunately, it is a rather uninspiring case analysis. To understand the difficulties involved the reader should, before reading the proof, try to figure out the location of a few specific strings in the output of the algorithm. For example, in the list for $k = 3$ and $n = 9$, where does 220121012 appear? For $k = 3$ and $n = 8$, where does 22012201 appear?

THEOREM 7.6 *The list of successive periodic reductions of necklaces as produced by the FKM algorithm forms a De Bruijn cycle.*

PROOF: Let D denote the list of k-ary symbols produced by concatenating the successive periodic reductions of necklaces from the FKM algorithm. Since (7.9) may be written as

$$k^n = \sum_{d \backslash n} d \cdot L_k(d),$$

the list D contains the correct number of digits. We now argue that each k-ary string α of length n occurs as a substring of D, which will finish the proof.

Note that the first two outputs of the algorithms are 0 and $0^{n-1}1$, and the last two outputs are $(k-2)(k-1)^{n-1}$ and $(k-1)$. Thus D has a prefix 0^n and a suffix $(k-1)^n$, from which it follows that all strings of the form $(k-1)^p 0^{n-p}$ (where $0 \leq p \leq n$) occur as substrings in D. All other strings have the form

$$\alpha = (k-1)^p (xy)^t$$

where $0 \leq p < n$, $t \geq 1$, $yx \in \mathbf{L}$, xy contains some non-zero symbol, and the first symbol of x is not $k-1$. We will classify the possibilities for α according to whether $p = 0$, whether $t = 1$, and whether $y = \varepsilon$.

Case 1 [$p = 0$, $t = 1$, $y = \varepsilon$]: Here $\alpha = x$ with $x \in \mathbf{L}$. Trivially x appears as substring in D since x is output by the algorithm.

Case 2 [$p = 0$, $t > 1$, $y = \varepsilon$]: Here $\alpha = x^t$ with $x \in \mathbf{L}$. The string x is output by the algorithm, and the next string output by the algorithm is $x^{t-1}S(x)$, where $S(x)$ is the *necklace* the lexicographically follows x. Thus the string x^t occurs in D since $xx^{t-1}S(x)$ is a substring of D.

Case 3 [$p = 0$, $t = 1$, $y \neq \varepsilon$]: Here $\alpha = xy$ and $yx \in \mathbf{L}$. The string yx is output by the algorithm; what output follows it? Note that $succ(yx) = yz$ for some string z since x is not all $(k-1)$'s (and $y(k-1)^{n-|y|} \in \mathbf{L}$). What is the periodic reduction of yz? We claim that it is a string with prefix y, which will finish this case. Observe from Theorem 7.5 that *if $\beta \in \mathbf{L}$, then β^r is the lexicographically smallest necklace of length $r|\beta|$ with prefix β*. Thus it cannot be that $yz = \beta^r$ and β is a proper substring of y, since yx is a lexicographically smaller necklace.

Case 4 [$p = 0$, $t > 1$, $y \neq \varepsilon$]: Here $\alpha = (xy)^t$ and $yx \in \mathbf{L}$. The string yx is output by the algorithm, and the next string output by the algorithm is $S((yx)^t) = (yx)^{t-1}yS(x)$, which is aperiodic. Thus D contains the substring $yx(yx)^{t-1}yS(x)$; a string which contains α as substring.

Case 5 [$p > 0$, $t = 1$, $y = \varepsilon$]: Here $\alpha = (k-1)^p x$ with $x \in \mathbf{L}$. The string D contains $\beta = pred(x)(k-1)^p$ since $\beta \in \mathbf{L}$. The next string output by the algorithm has prefix x, and so D contains the substring $pred(x)(k-1)^p x$.

Case 6 [$p > 0$, $t > 1$, $y = \varepsilon$]: Here $\alpha = (k-1)^p x^t$ with $x \in \mathbf{L}$. The string D contains $\beta = pred(x)(k-1)^{n-d}$ where $d = |x|$, since $\beta \in \mathbf{L}$. Let $n = md + r$ where $0 \leq r < m$. If $r > 0$ (i.e., d does not divide n), then $succ(\beta) = x^m S(y) \in \mathbf{L}$ where y is the string consisting of the first r symbols of x. Since $m \geq t$, the string α occurs on D. If $r = 0$, then following β the algorithm outputs x, followed by $x^{m-1}S(x)$. Thus D contains the string $pred(x)(k-1)^{n-d}xx^{m-1}S(x)$ which in turn contains α.

Case 7 [$p > 0$, $t = 1$, $y \neq \varepsilon$]: Here $\alpha = (k-1)^p xy$ with $p \geq 1$ and $yx \in \mathbf{L}$. Note that $\gamma = neck(\alpha)$ is either (a) $\gamma = xy(k-1)^p$ or (b) $\gamma = y(k-1)^p x$. In case (a) $\gamma \in \mathbf{L}$. We may now proceed exactly as in Case 5, with xy playing the role of x. That is to say, D will contain the string $pred(xy)(k-1)^p xy$. In case (b), if γ is aperiodic, then γ is output by the algorithm and the following string output is of the form $y(k-1)^p z$, so α appears on D. In case (b) if γ is periodic, then $y(k-1)^p$ must be the periodic reduction, where, say, $\gamma = [y(k-1)^p]^q$. The algorithm outputs $pred(y)(k-1)^{n-|y|}$, followed by $y(k-1)^p$, followed by $[y(k-1)^{q-1}S(y(k-1)^p)$, and thus D contains α.

Case 8 [$p > 0$, $t > 1$, $y \neq \varepsilon$]: Here $\alpha = (k-1)^p (xy)^t$ with $p \geq 1$, $t > 1$, and $yx \in \mathbf{L}$. Note that $\gamma = neck(\alpha)$ must be $\gamma = y(xy)^{t-1}(k-1)^p x \in \mathbf{L}$. The next string output by the algorithm is $y(xy)^{t-1}(k-1)^p S(x)$, so again α occurs as a substring of D. □

From our previous analysis of the FKM algorithm we know that the total amount of work, aside from outputting the symbols, in producing this cycle is $\Theta(k^n/n)$. Thus the time required to produce the De Bruijn cycle is dominated by the time to output the digits; i.e., it is $\Theta(k^n)$ which is best possible.

7.4 Computing the Necklace of a String

Given a string, it is often useful to compute its necklace. Such applications arise in several diverse areas such as graph drawing, where it is used to help determine the symmetries of a graph to be drawn in the plane.

Recall that for any string $\alpha = a_1 a_2 \ldots a_N \in \Sigma_k^N$ its necklace, $neck(\alpha)$ is the lexicographically smallest of its circular shifts. The question naturally arises as how to efficiently compute the necklace given the string. In this section we present an $O(N)$ algorithm for the task.

First we consider the problem of factoring a word as specified in the following theorem.

THEOREM 7.7 (CHEN, FOX, LYNDON) *Any word $\alpha \in \Sigma_k^+$ admits a unique factorization*

$$\alpha = \alpha_1 \alpha_2 \cdots \alpha_m,$$

such that $\alpha_i \in \mathbf{L}$ for $i = 1, 2, \ldots, m$ and

$$\alpha_1 \geq \alpha_2 \geq \cdots \geq \alpha_m.$$

Here are two examples of Lyndon factorizations.

011 011 00111 0 0 and 0102 0102 01 0 0

It is easy to see that such a factorization exists, since each letter is a Lyndon word and any two Lyndon words x and y for which $x < y$ can be concatenated to get another Lyndon word xy. Uniqueness is also not hard to show. See Exercises 12 and 13.

There is also a version of this theorem that deals with factorizations into necklaces.

THEOREM 7.8 *Any word $\alpha \in \Sigma_k^+$ admits a unique factorization*

$$\alpha = \alpha_1 \alpha_2 \cdots \alpha_m,$$

such that $\alpha_i \in \mathbf{N}$ for $i = 1, 2, \ldots, m$ and

$$\alpha_1 > \alpha_2 > \cdots > \alpha_m.$$

Here are two examples of necklace factorizations, using the same words as above.

011011 00111 00 and 01020102 01 00

Duval [105] has developed an elegant, efficient algorithm for factoring a word. Our version of the algorithm is essentially based on Theorem 7.5. The output of the algorithm consists of indices $0 = k_0, k_1, k_2, \ldots, k_m = N$ such that

$$\alpha_i = a_{k_{i-1}+1}, \ldots, a_{k_i}$$

Informally, the idea of the algorithm is to keep extending n and updating p until a value $a_n > a_{n-p}$ is encountered. Then $a_1 \cdots a_p$ is the longest prefix of α that is in $\mathbf{L}$ and $\beta^t \gamma = a_1 \cdots a_{n-1}$ where $pt + |\gamma| = n - 1$ with $|\gamma| < p$. The words

$$\underbrace{\beta, \beta, \ldots, \beta}_{t}, \gamma, \text{ causing output } k_i = i|\beta| \text{ for } i = 1, 2, \ldots, t$$

are the first t factors in the Lyndon factorization of α. Note that γ is the prefix of a Lyndon word that is lexicographically less than β. Now apply the same algorithm to γ and the remainder of α; i.e., to $\gamma a_n a_{n+1} \cdots a_N$. The details may be found in Algorithm 7.4.

```
D       k := 0; a[N + 1] := −1;
        while k < N do
            n := k + 2; p := 1;
            while a[n − p] ≤ a[n] do
                if a[n − p] < a[n] then p := n − k;
                n := n + 1;
            repeat PrintIt( k ); k := k + p;
            until k ≥ n − p;
        PrintIt( N );
```

Algorithm 7.4: Duval algorithm for factoring a string.

By moving the `PrintIt(k)` statement at line `(D8)` outside and just before the repeat loop (i.e., between lines `(D7)` and `(D8)`), the algorithm will produce the necklace factorization.

What is the running time of Duval's algorithm? Note that the comparison $a_{n-p} = a_n$ at line `(D4)` is the most often executed statement. Let us assume that we are doing a necklace factorization and that there are m factors. Let k_i, n_i, p_i be the values of k, n, and p at the end of the ith iteration of the outer while loop. The total number of comparisons done at line `(D4)` on the i-th iteration is $n_i - k_{i-1} - 1$. But by `(D9)`, $n_i - k_i \leq p_i$ so that

$$\begin{aligned}
\sum_{i=1}^{m} (n_i - k_{i-1} - 1) &= k_m + \sum_{i=1}^{m} (n_i - k_i - 1) \\
&\leq N - m + \sum_{i=1}^{m} p_i \\
&\leq 2N - m
\end{aligned}$$

Thus the total number of comparisons done at line (D4) is at most $2N$ and the running time of the algorithm is therefore $O(N)$.

How to find a necklace

To find the necklace of a word α, use $\alpha\alpha$ as the input to the necklace factorization algorithm (or modify the algorithm to do arithmetic mod n). Suppose that β^t is the necklace of α, where $\beta \in \mathbf{L}$. Then β^t will occur as t factors in the Lyndon factorization of $\alpha\alpha$. We need simply wait until a necklace factor of length $|\alpha|$ appears. When $n - k > N$ (tested after line (D9)), the string $a[k+1..k+N]$ is the necklace of α.

7.5 Universal Cycles

In this section we give a brief introduction to "Universal Cycles". These are a very interesting generalization of DeBruijn cycles introduced by Chung, Diaconis and Graham [77]. Many types of combinatorial objects are represented by strings of fixed length; suppose that there are N total objects and each representation has length n. The problem is to find a (circular) string D of length N such that every representative occurs exactly once as a contiguous substring of D. For De Bruijn cycles we had $N = 2^n$ and the representations were all bitstrings of length n. But what about combinations, permutations, set and numerical partitions, etc?

k-permutations of an n-set

Suppose that we try to extend the De Bruijn cycle idea from subsets (bitstings of length n) to k-permutations of an $[n]$. That is, we would like a circular string of length $(n)_k$ over the alphabet $[n]$ such that each k-permutation occurs exactly once as a substring. It is natural to define a digraph $G(n, k+1)$ whose vertices are $([n])_k$, the k-permutations of $[n]$, and where a generic vertex $a_1a_2\cdots a_k$ has $n-k+1$ outgoing edges whose endpoints are $a_2\cdots a_kb$ for $b \in [n] \setminus \{a_2, \cdots, a_k\}$. For example if $k = 1$ then $G(n, 2)$ is the complete directed graph; i.e., there is an edge between every pair of distinct vertices. Note that $G(n, k)$ is vertex-transitive.

One Eulerian cycle in $G(4, 2)$ is shown below.

1 2 3 4 1 4 2 4 3 2 1 3

LEMMA 7.5 *For $1 \le k < n$, the graph $G(n, k)$ is Eulerian.*

PROOF: It is easy to see that the in-degree of each vertex is also $n - k + 1$. We need only show that $G(n, k)$ is strongly connected. Given $\mathbf{a} = a_1a_2\cdots a_k \in ([n])_k$, note that there is a cycle of length k in $G(n, k)$ that contains every circular permutation of $\mathbf{a}$, namely the one obtained by repeatedly using edges of the form $x\alpha \to \alpha x$. We now show that there is a path from $\mathbf{a}$ to $\mathbf{a}$ with any two of its adjacent elements transposed. To do this, we need only show the existence of a path from $\mathbf{a}$ to $a_2a_1a_3\cdots a_k$. Let $x \notin \{a_1, a_2, \ldots, a_k\}$. Then there is a path

$$a_1a_2a_3\cdots a_k \to a_2a_3\cdots a_kx \to a_3\cdots a_kxa_1 \stackrel{*}{\to}$$
$$xa_1a_3\cdots a_k \to a_1a_3\cdots a_ka_2 \stackrel{*}{\to} a_2a_1a_3\cdots a_k$$

as claimed, where $\stackrel{*}{\rightarrow}$ denotes a path of some length. This implies that there is a path from $\mathbf{a}$ to $\mathbf{b} = b_1 b_2 \cdots b_k$ where $b_1 < b_2 < \cdots < b_k$ and $\{b_1, b_2, \ldots, b_k\} = \{a_1, a_2, \ldots, a_k\}$. We now show that there is a path from $\mathbf{b}$ to $12\cdots k$. Since $G(n,k)$ is vertex-transitive, that will finish the proof. Let i be the smallest value for which $b_i \neq i$ and $b_{i+1} > i+1$; if there is no such value, then we are done. The following path is in $G(n,k)$.

$$b_1 b_2 \cdots b_k \stackrel{*}{\rightarrow} b_{i+1} \cdots b_k 1 \cdots i \rightarrow b_{i+2} \cdots b_k 1 \cdots i(i+1) \stackrel{*}{\rightarrow} 12 \cdots i(i+1) b_{i+2} \cdots b_k.$$

Continuing in this manner we eventually reach $12\cdots k$. □

It would be interesting to develop a fast algorithm to output an Eulerian cycle whose existence is guaranteed by the lemma.

If $n = k$ then $G(n,k)$ consists of $n!/2$ 2-cycles of the form $\pi_1\pi_2\cdots\pi_n \rightleftharpoons \pi_2\cdots\pi_n\pi_1$, and is thus not Eulerian. Another way to view this is as follows. Consider $n = 3$. The substring 123 must occur, but what symbol follows the 3? It must be 1. And then 2 and then 3. But 123123 does not satisfy our criteria since, for example, 321 does not occur as a substring. The problem occurs because we have insisted on $[n]$ as our alphabet. Consider the string

1 4 5 2 4 3

This contains the substring 431 which we consider to represent the permutation 321 (which we missed before). Below are the six substrings of length 3 and the corresponding permutations of [3].

substring	145	452	524	243	431	314
permutation	123	231	312	132	321	213

More formally, we say that permutations of natural numbers, π and π', both of length n, are *order isomorphic* if $\pi_i < \pi_j$ if and only if $\pi'_i < \pi'_j$ for all $1 \leq i, j \leq n$. Our problem is to find a string of numbers whose substrings are order isomorphic to the $n!$ permutations of $[n]$.

Here's a string that works for $n = 4$.

1 2 3 4 1 2 5 3 4 1 5 3 2 1 4 5 3 2 4 1 3 2 5 4

How could we construct such a string? Do they always exist? How many extra symbols must be used? Taking our clue from De Bruijn sequences we define a graph and look for Eulerian cycles in that graph. **!!! need to put in the rest of the stuff from the CDG paper !!!**.

Let $N(n)$ be the number of extra symbols that are necessary to construct a U-cycle for permutations. Chung, Graham and Diaconis conjecture that $N(n) = n + 1$; that $n + 1 \leq N(n) \leq n + 6$ is known.

Set Partitions

a b c b c c c c d d c d e e f

Murasaki Universal Cycles?

Combinations

Here's a U-cycle for $n = 8$ and $k = 3$, where the underlying set is Σ_8.

024561457123612467036713450346012501356725602347235701 47

LEMMA 7.6 *If there is a U-cycle for* $\mathbf{A}(n, k)$ *then* k *divides* $\binom{n-1}{k-1}$.

Chung, Graham and Diaconis conjecture that this necessary condition is also sufficient as long as n is large enough, but there are also conjectures to the contrary!

7.6 Polynomials over finite fields

Polynomials whose coefficients are the elements of a finite field have many applications in mathematics and engineering. Of particular interest are what are known as irreducible polynomials and primitive polynomials. Somewhat surprisingly, we can use our algorithm for generating Lyndon words as the basis of an algorithm for generating all such polynomials of a given degree. This section will be brief and most proofs will be omitted as the mathematics necessary is non-trivial and would take more room than we have here. Several excellent introductions to finite fields are mentioned in the bibliographic remarks. Nevertheless, the careful reader should have little trouble turning the discussion here into a functioning program, particulary if written in a language for symbolic calculations, such as Maple. There is another connection between polynomials and the material presented earlier in this chapter, namely that a primitive polynomial ca be used to efficiently generate a de Bruijn sequence.

Let $p(x)$ be a monic polynomial over GF(q). Recall that GF(q) refers to the field of integers mod q and that q must be a prime power. The polynomial $p(x)$ is *irreducible* if it cannot be expressed as the product of two polynomials of lower degree.

If β is a root of a degree n polynomial $p(x)$ over GF(q), then the *conjugates* of β are $\beta, \beta^q, \ldots, \beta^{q^{n-1}}$. Each conjugate is also a root of $p(x)$. Irreducible polynomials are characterized in the following theorem.

THEOREM 7.9 *Let* p *be a degree* n *polynomial over GF(q) and* β *a root of* p. *The polynomial* p *is irreducible if and only if the conjugates of* β *are distinct.*

An element α of GF(q) is *primitive* if $k = q^n - 1$ is the smallest non-zero value for which $\alpha^k = 1$. A degree n polynomial $p(x)$ over GF(q) is *primitive* if it is irreducible and contains a primitive element of GF(q^n) as a root. In other words, to test whether an irreducible polynomial $p(x)$ is primitive.....

The number of primitive polynomials over GF(q) is known to be

$$P_n(q) \;=\; \frac{1}{n}\phi(q^n - 1)$$

Here is a small table for $q = 2$.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$P_n(q)$	1	1	2	2	6	6	18	16	48	60	176	144	630	756	1800	2048

The number of irreducible polynomials of degree n over GF(q) is given by the so-called "Witt formula"

$$I_q(n) \;=\; \frac{1}{n}\sum_{d\backslash n}\mu(n/d)q^d$$

This is the same as the number of Lyndon words, $I_k(n) = L_k(n)$. This remarkable coincidence cries out for an explanation and a purpose of this section is to supply one. The key to the correspondence is Theorem 7.9.

If α is a generator of GF(q^n) $-$ {0}, then $\beta = \alpha^k$ for some k. The conjugates of β are thus

$$\alpha^k, (\alpha^k)^q, \ldots, (\alpha^k)^{q^{n-1}}.$$

Which k's will give rise to β's that are the roots of irreducible polynomials? Since the conjugates are all distinct, they are the k's for which $k, qk, q^2k, \ldots, q^{n-1}k$ are all distinct mod $q^n - 1$. Thinking of k as a length n base q number, the conjugates are all the circular shifts of k. For them to be distinct k has to be a q-ary length n Lyndon word.

Lyndon word	k	irreducible polynomial	order e
000001	1	1000011	63
000011	3	1010111	21
000101	5	1100111	63
000111	7	1001001	9
001011	11	1101101	63
001101	13	1011011	63
001111	15	1110101	21
010111	23	1110011	63
011111	31	1100001	63

Given k the order of the polynomial is $(q^n-1)/gcd(q^n-1, k)$. The primitive polynomials are those of order $q^n - 1$.

The polynomial $p(x) = x^6 + x + 1$ is known to be primitive over GF(2). Let β be a root of $p(x)$. Thus $\beta^6 = 1 + \beta$. We now illustrate the correspondence on the Lyndon words 000001 and 001011; corresponding to $\alpha = 1$ and $\alpha = 11$, respectively. We now compute (note that $(1 + \alpha)^2 = 1 + \alpha^2$). First, with $\alpha = 1$.

$$\begin{aligned}
r(x) &= (x+\beta)(x+\beta^2)(x+\beta^4)(x+\beta^8)\,(x+\beta^{16})(x+\beta^{32}) \\
&= (x^2+(\beta+\beta^2)x+\beta^3)(x+\beta^4)(x+\beta^2(1+\beta))\times \\
&\quad (x+\beta^4(1+\beta)^2)(x+\beta^2(1+\beta)^2(1+\beta)^2(1+\beta)) \\
&= (x^2+(\beta+\beta^2)x+\beta^3)(x+\beta^4)(x+\beta^2+\beta^3))\times \\
&\quad (x+\beta^4(1+\beta^2))(x+\beta^2(1+\beta^2)^2(1+\beta)) \\
&= (x+\beta)(x+\beta^2)(x+\beta^4)(x+\beta^2+\beta^3))(x+1+\beta+\beta^4)(x+\beta^2(1+\beta^4)(1+\beta)) \\
&= (x+\beta)(x+\beta^2)(x+\beta^4)(x+\beta^2+\beta^3))(x+1+\beta+\beta^4)(x+1+\beta^3) \\
&= (x+(\beta+\beta^2)x+x^2)(x+\beta^4)(x+\beta^2+\beta^3))(x+1+\beta+\beta^4)(x+1+\beta^3) \\
&= (x+(\beta+\beta^2)x+x^2)(x+\beta^4)(x+\beta^2+\beta^3))(x+1+\beta+\beta^4)(x+1+\beta^3) \\
&= x^6+x+1
\end{aligned}$$

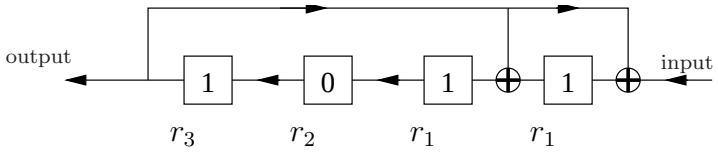

Figure 7.6: The LFSR corresponding to the polynomial $x^4 + x + 1$.

Now with $\alpha = 11$ (note that $\beta^{64} = 1$).

$$\begin{aligned} r(x) &= (x+\beta^{11})(x+\beta^{22})(x+\beta^{44})(x+\beta^{88})(x+\beta^{176})(x+\beta^{352}) \\ &= (x+\beta^{11})(x+\beta^{22})(x+\beta^{44})(x+\beta^{24})(x+\beta^{48})(x+\beta^{32}) \\ &= (x+\beta^5(1+\beta))(x+\beta^4(1+\beta)^3)(x+\beta^2(1+\beta)^7) \times \\ &\quad (x+(1+\beta)^4)(x+(1+\beta)^8)(x+\beta^2(1+\beta)^5) \\ &= x^6+x^5+x^3+x^2+1 \end{aligned}$$

THEOREM 7.10 *Every string of length n, except 0^n occurs in a linear recurring sequence generated by a primitive polynomial over* GF(p^n).

For example, the polynomial $p(x) = x^4 + x + 1$ is primitive over GF(2). It generates a linear recurring sequence defined by the recurrence relation

$$c_n = c_{n-3} + c_{n-4}.$$

Using the initial conditions $c_1, c_2, c_3, c_4 = 1, 0, 0, 0$ we obtain the sequence

1 0 0 0 1 0 0 1 1 0 1 0 1 1 1

Inserting another 0 into the block of $n - 1$ 0's produces a De Bruijn sequence, but not all De Bruijn sequences are obtained in this manner.

7.6.1 Linear Feedback Shift Registers

A *linear feedback shift register*, or LFSR, is a device like that illustrated in Figure 7.6 On each clock cycle the bit in *cell* r_{k-1} is output and fed back into some subset of the cells, depending on the placement of the feedback lines. There is one feedback line corresponding to each non-zero term of its polynomial. If the feedback line is present, then the new value of r_i is $r_{i-1} \oplus r_{k-1}$, otherwise it is just r_{i-1}. The value of r_{-1} is taken to be zero, which means that the exclusive-or gate feeding into r_0 is redundant; it is shown for the sake of consistency.

```
do { if ((x <<= 1) >> k) x = x & ONES ^ PP;
} while (x != INITIAL);
```

Algorithm 7.5: C code for iterating an LFSR.

Algorithm 7.5 generates the De Bruijn sequence in the least significant bit of the unsigned integer x, whose initial value is `INITIAL`. The algorithm is simulating the action of the LFSR.

Unsigned integer `PP` is a bitstring encoding of a primitive polynomial, 0011 in the example above. Constant `ONES` is a bitstring of k 1's in the lower order bits. This code essentially generates each new value of x in constant time. The only problematic operation is the shift right by k positions. On most modern computers this shift should be executed in one or two machine instructions.

7.6.2 Another look at the BRGC

We may identify a bitstring $b_0 b_1 \cdots b_{n-1}$ with a degree $n-1$ polynomial over GF(2) under the bijection

$$b_0 b_1 \cdots b_{n-1} \leftrightarrow \sum_{i=0}^{n-1} b_i x^i.$$

Under this bijection a right shift corresponds to multiplying by x and componentwise exclusive-or to addition mod 2. Thus if $g(x)$ is the polynomial corresponding to the bth word in the BRGC, then

$$g(x) = (1+x)b(x).$$

Since the algebraic inverse of $1+x$ is the polynomial $1 + x + \cdots + x^{n-1}$ (check this!), the procedure for converting from the BRGC to binary corresponds to multiplication by $1 + x + \cdots + x^{n-1}$. That is,

$$b(x) = (1 + x + \cdots + x^{n-1})g(x).$$

7.7 Exercises

1. **[1]** What is the length of a maximal domino game for a general value of n (dots taken from $[n]$)?

2. **[2+]** Show that there is a string $s_0 s_2 \cdots s_{2n!-1}$ of length $2n!$ over the alphabet $[n]$ such that

   $$\{s_i s_{i+1} \cdots s_{i+n!-1} : i = 0, 2, \ldots, 2n!\},$$

   where the index arithmetic is taken mod $2n!$, is the set of all $n!$ permutations of $[n]$.

3. **[*R*−]** How many Eulerian cycles does the directed n-cube $\vec{Q}_n$ have?

4. **[*R*−]** Develop a CAT or, better yet, loopfree algorithm for generating an Eulerian cycle in $\vec{Q}_n$.

5. **[2]** Use (7.10) to prove (7.7).

6. **[2+]** Let $N(n_0, n_1, \ldots, n_t)$ denote the number of necklaces composed of n_i occurrences of the symbol i, for $i = 0, 1, \ldots, t$. Let $n = n_0 + n_1 + \cdots + n_t$. Prove that

   $$N(n_0, n_1, \ldots, n_t) = \frac{1}{n} \sum_{d \backslash \gcd(n_0, \ldots, n_t)} \phi(d) \frac{(n/d)!}{(n_0/d)! \cdots (n_t/d)!}$$

In particular, if $t = 1$, then

$$N(r, n-r) \;=\; \frac{1}{n} \sum_{d \backslash \gcd(r,n-r)} \phi(d) \binom{n/d}{r/d}$$

gives the number of necklaces with r black beads and $n - r$ white beads.

7. [**1+**] Find a simple one-to-one correspondence between length $2n$ necklaces with n black beads and n white beads and rooted plane trees with n edges.

8. [**2+**] Derive a formula for the number of binary Lyndon words of length n and weight r. The *weight* of a string of digits is the sum of those digits. [**2+**] Let $eL_k(n)$ be the number of length n Lyndon words of even weight, $oL_k(n)$ be the number of odd weight, and $dL_k(n)$ be the difference $dL_k(n) = eL_k(n) - oL_k(n)$. Show that $dL_2(n) = n^{-1} \sum \mu(d) 2^{n/d}$ where the sum is over all odd $d \backslash n$. Show that $dL_2(2n) = -oL_2(n)$.

9. [**1**] For odd n, show that $eN_2(n)$, the number of binary necklaces with an even number of 1's, is equal to $oN_2(n)$, the number of binary necklaces with an odd number of 1's.

10. [**1+**] What bitwise operations preserve necklaces (or prenecklaces, or Lyndon words)? I.e., is the intersection of necklaces a necklace? What about union and exclusive-or?

11. [**1+**] Modify both the iterative and the recursive versions of the necklace generating algorithms so that the produce necklaces in relex order, instead of lex order. Which algorithm was easier to modify?

12. [**2**] Prove that $w \in \mathbf{L}$ if and only if $w < v$ for all $uv = w$ where $v \neq \varepsilon$. I.e., a word is a Lyndon word if and only if it is strictly less than all of its proper suffixes.

13. [**2**] Prove that if x and y are both Lyndon words and $x < y$ then xy is also a Lyndon word. Finish the proof of the Chen, Fox, Lyndon Theorem 7.7 by showing that the Lyndon factorization is unique.

14. [**3**] Define an involution τ on $\{0,1\}^n$ by

$$\tau(x_1 \ldots x_n) = x_1 \ldots x_{n-1} \overline{x_n},$$

where $\overline{x_n}$ denotes the complement of the bit x_n, and let $\sigma(x)$ denote the rotation of string x one position left. Show that the calls: Print(0^n); Gen($0^{n-1}1$); generate all necklaces of length n in two colors, where Gen is the procedure shown below.

```
procedure Gen ( x : necklace );
begin
    Print( x );
    x := τ(x);
    while IsNecklace(x) do begin
        Gen( x );
        x := τστ^{-1}(x);
    end;
end {of Gen};
```

15. **[2]** Let $\rho = \tau\sigma$ where τ and σ are as in the previous exercise. In other words,

$$\rho(b_1 b_2 \cdots b_n) = b_2 \cdots b_n \overline{b_1}$$

Define an equivalence relation $\sim$ between bitstrings $\mathbf{x}$ and $\mathbf{y}$ of length n as follows: $\mathbf{x} \sim \mathbf{y}$ if and only if there is a number k such that $\rho^k(\mathbf{x}) = \mathbf{y}$. Show that the number of such equivalence classes is

$$\frac{1}{2n} \sum_{\substack{d \backslash n \\ d \text{ odd}}} 2^{n/d} \phi(d).$$

There is a one-to-one correspondence between such equivalence classes and what are known as "vortex-free" tournaments.

16. **[*R*−]** Let $\mathbf{V}(n)$ denote the set of lexicographically least representatives of length n of the equivalence classes of $\sim$, as defined in the previous exercise. For example $\mathbf{V}(n) = \{00000, 00010, 00100, 01010\}$. Develop an efficient, and preferably CAT, algorithm to generate $\mathbf{V}(n)$.

17. **[2]** How many n-bead necklaces are there, composed of white and black beads, and with no two adjacent black beads? How many such necklaces are aperiodic? Prove the following, a kind of Fermat's Little Theorem for Fibonacci numbers: If p is a prime, then $F_{p+1} + F_{p-1} \equiv 1 \pmod{p}$. **[*R*−]** Given a forbidden pattern P (a bitstring — 00 in the first part of this problem), how many necklaces of length n are there without P as a substring? How fast can you compute the number?

18. **[*R*−]** Develop an efficient ranking algorithm for prenecklaces, necklaces, and Lyndon words in the order that they are generated by the FKM algorithm.

19. **[*R*]** Develop an efficient ranking algorithm for some De Bruijn sequence. If the De Bruijn sequence arising from the FKM algorithm is used then the results of the previous exercise should be useful.

20. **[1+]** Prove that if $\alpha \in \mathbf{P}_k(n)$ then $\alpha(k-1)^n \in \mathbf{N}_k(n)$. This means that every prenecklace α is also the prefix of a Lyndon word, unless α is a string of $(k-1)$s of length greater than 1.

21. **[2]** Prove: $\alpha \in \mathbf{P}$ if and only if $xy \leq yz$ and $x \leq z$ for all x, y, z such that $\alpha = xyz$ with $|x| = |z|$.

22. **[*R*]** Find a Gray code (successive words differ by one bit) of Lyndon words when $k = 2$ or prove that no such code exists.

23. **[3+]** Let $n > 1$ and $p, q > 1$. Show that $L_{q+4}(n) - L_q(n)$ is even and hence that if $p \equiv q \pmod{4}$, then $L_q(n) \equiv L_p(n) \pmod{2}$. Use this result to determine the parity of $L_q(n)$.

24. **[2]** Prove that $L_{pq}(n) = \sum \gcd(i,j) L_p(i) L_q(j)$ where the sum is taken over all i and j such that $\operatorname{lcm}(i,j) = n$.

25. [**2**] Find a class of words that causes the Duval algorithm to use $2N-o(N)$ comparisons for infinitely many values of N.

26. [***R—***] For a binary alphabet, what is the maximum number of comparisons that can be used by the Duval algorithm on a string of length N? For a k-ary alphabet?

27. [***R—***] Develop a CAT algorithm for generating all necklaces where the number of beads of each color is fixed. The number of such necklaces was determined in Exercise 6. Even the two color case is open.

28. [***R—***] Develop a CAT algorithm for generating the lexicographically least representatives of the equivalence classes of k-ary strings of length n that are equivalent under rotation *or reversal*. In other words, the dihedral group is acting on the strings and not just the cyclic group. Such equivalence classes are sometimes called *bracelets*.

29. [**2**] What about 2-dimensional analogues of De Bruijn cycles? These are sometimes called *De Bruijn torii*. (a) Consider the set of 2 by 2 tiles where each square of the tile is colored black or white. There are 16 such tiles. Can they be placed in a 4 by 4 arrangement so that their borders match, where the border wrap-around in the manner of a torus? (b) Now let each tile be colored with three colors, say white, gray, and black. There are 81 tiles. Can they be placed in a 9 by 9 arangements so that the colors along their borders match?

30. [**2**] For those who know about context-free languages: Use a closure property to prove that **N** and **L** are not context-free languages. Use the "pumping lemma" to prove that **N** and **L** are not context-free languages.

31. [**2**] Let $\mathbf{I}(n,k)$ be the set of permutations $\pi \in \mathbb{S}_n$ such that $\pi^k = \mathbf{1}$, the identity permutation, and $I(n,k) = |\mathbf{I}(n,k)|$. Thus $I(n,2)$ counts the number of involutions in $\mathbb{S}_n$. Show that

$$I(n,k) = \sum_{d \backslash k} \binom{n-1}{d-1} (d-1)! I(n-d,k).$$

[***R—***] Develop a CAT algorithm for generating the elements of $\mathbf{I}(n,k)$.

32. [**1**] Modify algorithm 7.5 so that it outputs a De Bruijn sequence.

7.8 Bibliographic Remarks

The Eulerian cycle in $\vec{Q}_n$ is from Bate and Miller [32].

An early paper containing an algorithm for constructing a DeBruijn cycle is Martin [292]. More recent papers about generating De Bruijn cycles include Fredricksen and Kessler [157], Fredricksen [156], Ralston [355], Huang [202], and Xie [503]. It is rather remarkable that there is a simple closed form formula for the number of distinct De Bruijn cycles.

$$(k!)^{k^{n-1}}/k^n.$$

This formula was proven by Van Aardenne-Ehrenfest and De Bruijn [464], but was anticipated by Flye-St. Marie [149]. It is even possible to count the number of Eulerian

cycles (directed or undirected) in a graph. See Fleishner's book [146], which contains everything you could want to know about Eulerian cycles. Generating all Eulearian cycles was considered by Fleishner [147].

Interesting material about De Bruijn cycles may be found in Stewart [440].

The algorithm ??? for generating necklaces is from Fredricksen and Maiorana [155] and Fredricksen and Kessler [154]. Independently, Duval [106] developed a version of the algorithm that generates Lyndon words. Duval [105] is our source for the material of the section on finding the necklace of a string.

The formula (7.7) has been attributed by Comtet [80] to the 1892 paper of Jablonski [208]. It is credited to "Colonel Moreau of the French Army" by Metroplis and Rota [305]. Cummings and Mays [88] determine the parity of the Witt formula.

Papers about generating primitive and irreducible polynomials include Lüneburg [285], and Gulliver, Serra and Bhargava [176]. The relationship between irreducible polynomials and Lyndon words is explored in Golomb [170], Lüneburg [285], [286], and Reutenauer [371]. Tables of primitive polynomials may be found there and in Peterson and Weldon [336] and Serra [415].

An analysis of the FKM algorithm is given in Ruskey, Savage, and Wang [388]. Duval's nearly identical algorithm is analyzed in Berstel and Pocchiola [39]. The recursive version of the FKM algorithm (Algorithm 7.3) is believed to be new.

A CAT algorithm for necklaces where the number of 0's is fixed may be found in Ruskey and Sawada [390]. A CAT algorithm for binary bracelets was developed by Sawada [406].

More material on Lyndon words may be found in Lothaire [279].

Universal cycles were introduced in Chung, Diaconis and Graham [77]. Later developments may be found in Jackson [209] and Hurlbert [204], [205].

There is a delightful chapter called "The Autovoracious Ourotorus" about De Bruijn cycles and generalizations in Stewart [440].