Efficient construction of the Prefer-same de Bruijn sequence

Evan Sala
School of Computer Science, University of Guelph, Canada
esala@uoguelph.ca

Joe Sawada
School of Computer Science, University of Guelph, Canada
jsawada@uoguelph.ca

Abbas Alhakim
Department of Mathematics, American University of Beirut, Lebanon
aa145@aub.edu.lb

Abstract

The greedy Prefer-same de Bruijn sequence construction was first presented by Eldert et al. [*AIEE Transactions* 77 (1958)] in 1958. As a greedy algorithm, it has one major downside: it requires an exponential amount of space to store the length 2^n de Bruijn sequence. Though de Bruijn sequences have been heavily studied over the last 60 years, finding an efficient construction for the Prefer-same de Bruijn sequence has remained a tantalizing open problem. In this paper, we unveil the underlying structure of the Prefer-same de Bruijn sequence and solve the open problem by presenting an efficient algorithm to construct it using $O(n)$ time per bit and only $O(n)$ space.

2012 ACM Subject Classification Mathematics of computing → Discrete mathematics

Keywords and phrases de Bruijn sequence, prefer-same, greedy algorithm, pure run-length register, Euler cycle

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Greedy algorithms often provide some of the nicest algorithms to exhaustively generate combinatorial objects, especially in terms of the simplicity of their descriptions. An excellent discussion of such algorithms is given by Williams [24] with examples given for a wide range of combinatorial objects including permutations, set partitions, binary trees, and de Bruijn sequences. A downside to greedy constructions is that they generally require exponential space to keep track of which objects have already been visited. Fortunately, most greedy constructions can also be constructed efficiently by either an iterative successor-rule approach, or by applying a recursive technique. Such efficient constructions often provide extra underlying insight into both the combinatorial objects and the actual listing of the object being generated.

A *de Bruijn* sequence of order n is a sequence of bits that when considered cyclicly contains every length n binary string as a substring exactly once; each such sequence has length 2^n. They have been studied as far back as 1894 with the work by Flye Sainte-Marie [9], receiving more significant attention starting in 1946 with the work of de Bruijn [4]. Since then, many different de Bruijn sequence constructions have been presented in the literature (see surveys in [11] and [16]). Generally, they fall into one of the following categories: (i) greedy approaches (ii) iterative successor-rule based approaches which includes linear (and non-linear) feedback shift registers (iii) string concatenation approaches (iv) recursive approaches. Underlying all of these algorithms is the fact that every de Bruijn sequence is in 1-1 correspondence with an Euler cycle in a related de Bruijn graph.

Perhaps the most well-known de Bruijn sequence is the one that is the lexicographically largest. It has the following greedy Prefer-1 construction [21]: (i) Start with the seed string 0^{n-1} (ii) **Repeat**

© Evan Sala, Joe Sawada and Abbas Alhakim;
licensed under Creative Commons License CC-BY

until no new bit is added: Append a 1 if it does not create a duplicate length n substring; otherwise append a 0 if it does not create a duplicate length n substring (iii) Remove the seed. For example, applying this construction for $n = 4$ we obtain the string: ~~000~~ 1111011001010000. Like all greedy de Bruijn sequence constructions, this algorithm has a major downside: it requires an exponential amount of space to remember which substrings have already been visited. Fortunately, the resulting sequence can also be constructed efficiently by applying an $O(n)$ time per bit successor-rule which requires $O(n)$ space [10]. By applying a necklace concatenation approach, it can even be generated in amortized $O(1)$ time per bit and $O(n)$ space [13].

Another interesting greedy de Bruijn sequence construction is the Prefer-same construction. It was first presented by Eldert et al. [6] in 1958 and was revisited with a proof of correctness by Fredricksen [11] in 1982. Recently, the description of the algorithm was simplified [3] as follows:

Prefer-same algorithm [3]

1. Seed with length $n-1$ string $\cdots 01010$
2. Append 1
3. **Repeat** until no new bit is added: Append the **same** bit as the last if it does not create a duplicate length n substring; otherwise append the opposite bit as the last if it does not create a duplicate length n substring
4. Remove the seed

For $n = 4$, the sequence generated by this Prefer-same construction is ~~010~~ 1111000011010010. Unlike the Prefer-1 sequence, and despite the vast research on de Bruijn sequences, the resulting Prefer-same sequence has no known efficient construction; it has remained an elusive open problem for over 60 years. The closest attempt came in 1977 when Fredricksen and Kessler devised a construction based on lexicographic compositions [12]; it matched the Prefer-same sequence for a very long prefix. To simplify our discussion, let:

- $\mathcal{S}_n$ = the de Bruijn sequence of order n generated by Prefer-same algorithm, and
- $\mathcal{L}_n$ = the de Bruijn sequence of order n generated using lexicographic compositions in [12].

The main result of this paper is to solve the above open problem by providing a successor-rule based construction of $\mathcal{S}_n$. It generates the sequence in $O(n)$ time per bit using only $O(n)$ space. As we solve this problem, we also provide some extra insight into the sequence $\mathcal{L}_n$. Our results rely on two key findings related to run-length encodings:

1. $\mathcal{S}_n$ is the lexicographically largest de Bruijn sequence of order n with respect to its run-length encoding, as noted in our preliminary work [3].
2. Successor-rules for both $\mathcal{S}_n$ and $\mathcal{L}_n$ are based on the underling feedback function $f(w_1w_2\cdots w_n) = w_1 \oplus w_2 \oplus w_n$, where $\oplus$ denotes addition modulo 2. We demonstrate this feedback function has nice run-length properties when used to partition the set of all binary strings of length n.

Before introducing our main result, we first provide an insight into greedy constructions for de Bruijn sequences that we feel has not been properly emphasized in the literature. In particular, we demonstrate how all such constructions, which are generalized by the notion of preference or look-up tables [2, 25], are in fact just special cases of a standard Euler cycle algorithm on the de Bruijn graph. This discussion is found in Section 2. In Section 3 we present some background on run-length encoding. In Section 4, we present an overview of important feedback functions that lead to our study of the function $f(w_1w_2\cdots w_n) = w_1 \oplus w_2 \oplus w_n$, which is studied in Section 5. In Section 6 we present our main result: an efficient successor-rule to generate the Prefer-same de Bruijn sequence. We conclude by presenting a summary of our results and directions for future

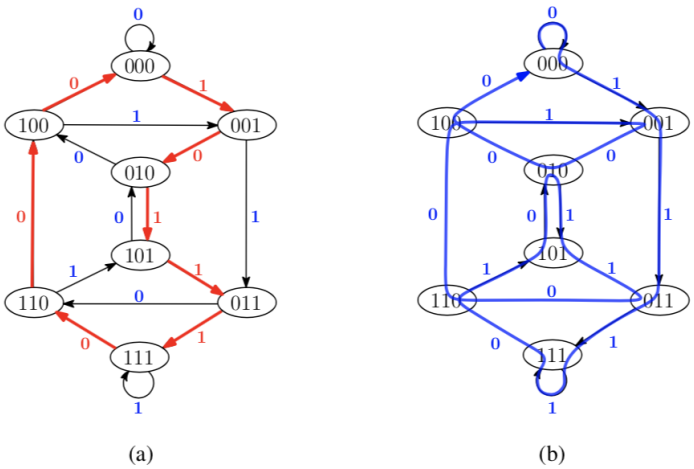


Figure 1 (a) A Hamilton cycle in $G(3)$ starting from 000 corresponding to the de Bruijn sequence 10111000 of order 3. (b) An Euler cycle in $G(3)$ starting from 000 corresponding to the de Bruijn sequence 0111101011001000 of order 4.

research in Section 7. The appendix contains proofs of our technical results. A C implementation of the algorithms discussed in this paper can be found in the appendix and are available for download at `http://debruijnsequence.org`.

2 Euler cycle algorithms and the de Bruijn graph

The *de Bruijn graph* of order n is the directed graph $G(n) = (V, E)$ where V is the set of all binary strings of length n and there is a directed edge $u = u_1u_2\cdots u_n$ to $v = v_1v_2\cdots v_n$ if $u_2\cdots u_n = v_1\cdots v_{n-1}$. Each edge e is labeled by v_n. Outputting the edge labels in a Hamilton cycle of $G(n)$ produces a de Bruijn sequence. Figure 1(a) illustrates a Hamilton cycle in the de Bruijn graph $G(3)$. Starting from 000, its corresponding de Bruijn sequence is 10111000.

Each de Bruijn graph is connected and the in-degree and the out-degree of each vertex is two; the graph $G(n)$ is Eulerian. $G(n)$ is the line graph of $G(n-1)$ which means an Euler cycle in $G(n-1)$ corresponds to a Hamilton cycle in $G(n)$. Thus, the sequence of edge labels visited in an Euler cycle is a de Bruijn sequence. Figure 1(b) illustrates an Euler cycle in $G(3)$. The corresponding de Bruijn sequence of order four when starting from the vertex 000 is 0111101011001000.

Finding an Euler cycle in an Eulerian graph is linear-time solvable with respect to the size of the graph. However, since the graph must be stored, applying such an algorithm to find a de Bruijn sequence requires $O(2^n)$ space. One of the most well-known Euler cycle algorithms for directed graphs is the following due to Fleury [8] with details in [11]. The basic idea is to not burn bridges. In other words, do not visit (and use up) an edge if it leaves the remaining graph disconnected.

Fleury's Euler cycle algorithm (do not burn bridges)

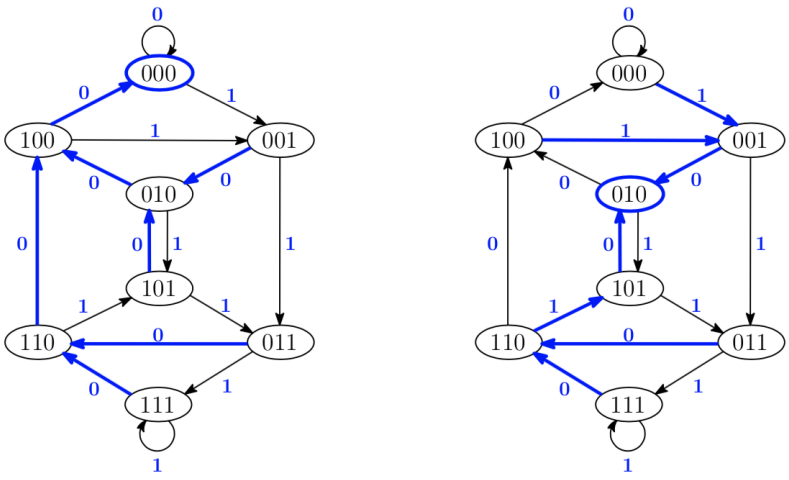
1. Pick a root vertex and compute a spanning in-tree T
2. Make each edge of T (the bridges) the last edge on the adjacency list of the corresponding vertex
3. Starting from the root, traverse edges in a depth-first manner by visiting the first unused edge in the current vertex's adjacency list

Prefer-1

Vertex	Edge pref
000	1,0
010	1,0
100	1,0
110	1,0
001	1,0
011	1,0
101	1,0
111	1,0

Prefer-same

Vertex	Edge pref
000	0,1
010	0,1
100	0,1
110	0,1
001	1,0
011	1,0
101	1,0
111	1,0



(a) (b)

Figure 2 (a) A preference table corresponding to the Prefer-1 greedy construction along with its corresponding spanning in-tree rooted at 000. (b) A preference table corresponding to the Prefer-same greedy construction along with its corresponding spanning in-tree rooted at 010.

Finding a spanning in-tree T can be done by reversing the direction of the edges in the Eulerian graph and computing a spanning out-tree with a standard depth first search on the resulting graph. The corresponding edges in the original graph will be a spanning in-tree. Using this approach, all de Bruijn sequences can be generated by considering all possible spanning in-trees.

Although not well documented, this algorithm is the basis for all greedy de Bruijn sequence constructions along with their generalizations using preference tables [2] or look-up tables [25]. Specifically, a preference table specifies the precise order that the edges are visited for each vertex when performing Step 3 in Fleury's Euler cycle algorithm. Thus given a preference table and a root vertex, Step 3 in the algorithm can be applied to construct a de Bruijn sequence if combining the last edge from each non-root vertex forms a spanning in-tree to the root. For example, the preference tables and corresponding spanning in-trees for the Prefer-1 (rooted at 000) and Prefer-same (rooted at 010) constructions are given in Figure 2 for $G(3)$. Notice how these strings relate to the seeds in their respective greedy constructions. For the Prefer-same, a root of 101 could also have been chosen, and doing so will yield the complement of the Prefer-same sequence when applying this Euler cycle algorithm.

A second well-known Euler cycle algorithm for directed graphs, attributed to Hierholzer [19], is as follows:

Hierholzer's Euler cycle algorithm (cycle joining)

1. Start at an arbitrary vertex v visiting edges in a depth-first manner until returning to v, creating a cycle.
2. **Repeat until all edges are visited:** Start from any vertex u on the current cycle and visit remaining edges in a DFS manner until returning to u, creating a new cycle. Join the two cycles together.

This cycle-joining approach is the basis for all successor-rule constructions of de Bruijn sequences. A general framework for joining smaller cycles together based on an underlying feedback shift register is given for the binary case in [16], and then more generally for larger alphabets in [17]. It is the basis for the efficient algorithm presented in this paper, where the initial cycles are induced by a specific feedback function.

3 Run-length encoding

The sequences $\mathcal{S}_n$ and $\mathcal{L}_n$ are related in the sense that they have properties based on a run-length encoding of binary strings. The *run-length encoding* of a string $\omega = w_1 w_2 \cdots w_n$ is a compressed representation that stores consecutively the lengths of the maximal runs of each symbol. The *run-length* of ω is the length of its run-length encoding. For example the string 11000110 has run-length encoding 2321 and run-length 4. Since we are dealing with binary strings, we only require information regarding the starting symbol to obtain a given binary string from its run-length encoding. As a further example:

$\mathcal{S}_5 = 11111000001110110011010001001010$ has run-length encoding 5531222113121111.

The following fact is proved in [3].

▶ **Fact 1.** *The sequence $\mathcal{S}_n$ is the de Bruijn sequence of order n starting with 1 that has the lexicographically largest run-length encoding.*

Let $alt(n)$ denote the alternating sequence of 0s and 1s of length n that ends with 0: For example, $alt(6) = 101010$. The following fact about $\mathcal{S}_n$ is also immediate from [3]:

▶ **Fact 2.** *$\mathcal{S}_n$ has prefix 1^n and has suffix $alt(n-1)$.*

The sequence $\mathcal{L}_n$ also has run-length properties: it is constructed by concatenating lexicographic compositions which are represented using a run-length encoding.

4 Feedback functions and de Bruijn successors

Let $\mathbf{B}(n)$ denote the set of all binary strings of length n. A function $f : \mathbf{B}(n) \to \{0, 1\}$ is a *feedback function*. Let $\omega = w_1 w_2 \cdots w_n$ be a string in $\mathbf{B}(n)$. A *feedback shift register* is a function $F : \mathbf{B}(n) \to \mathbf{B}(n)$ that takes the form $F(\omega) = w_2 w_3 \cdots w_n f(w_1 w_2 \cdots w_n)$ for a given feedback function f. A feedback function $g : \mathbf{B}(n) \to \{0, 1\}$ is a *de Bruijn successor* if there exists a de Bruijn sequence of order n such that each string $\omega \in \mathbf{B}(n)$ is followed by $g(\omega)$ in the given de Bruijn sequence. Given a de Bruijn successor g and a seed string $\omega = w_1 w_2 \cdots w_n$, the following function $\mathrm{DB}(g, \omega)$ will return a de Bruijn sequence of order n:

```
1: function DB(g, ω)
2:     for i ← 1 to 2^n do
3:         x_i ← g(ω)
4:         ω ← w_2 w_3 ··· w_n x_i
5:     return x_1 x_2 ··· x_{2^n}
```

The goal of this paper is to present an efficient de Bruijn successor for $\mathcal{S}_n$.

In this remainder of this section we discuss two of the simple feedback shift registers discussed by Golomb [18]: the pure cycling register and the complementing cycling register. Then in the following section, we discuss their relationship to a feedback function, first considered in [22], that induces interesting run-length properties. For the upcoming feedback functions we will be discussing representative strings for the cycles they induce. In particular, we define the *RL-rep* to be the string with the lexicographically largest run-length encoding, and if there are two such strings, the RL-rep is the one beginning with 1.

4.1 The pure cycling register (PCR)

The pure cycling register, denoted PCR, is the feedback shift register with the feedback function $f(\omega) = w_1$. It is well-known that the PCR partitions $\mathbf{B}(n)$ into cycles of strings that are equivalent under rotation. Note that the cycle $1100, 1001, 0011, 0110$ has two strings 1100, and 0011 with run-length encoding of 22. By definition, 1100 is the RL-rep.

Example 1 The PCR partitions $\mathbf{B}(5)$ into the following eight cycles $\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_8$ where the top string in bold is the RL-rep for the given cycle.

$\mathbf{P}_1$	$\mathbf{P}_2$	$\mathbf{P}_3$	$\mathbf{P}_4$	$\mathbf{P}_5$	$\mathbf{P}_6$	$\mathbf{P}_7$	$\mathbf{P}_8$
11010	**00101**	**11110**	**00001**	**11100**	**00011**	**11111**	**00000**
10101	01010	11101	00010	11001	00110		
01011	10100	11011	00100	10011	01100		
10110	01001	10111	01000	00111	11000		
01101	10010	01111	10000	01110	10001		

The PCR is the underlying feedback function used to construct the Prefer-1 greedy algorithm corresponding to the lexicographically largest de Bruijn sequence. It has also been applied in some of the simplest and most efficient de Bruijn sequence constructions [5, 16, 23].

4.2 The complementing cycling register (CCR)

The complementing cycling register, denoted CCR, is the FSR with the feedback function $f(\omega) = w_1 \oplus 1$. A string and its complement will belong to the same cycle induced by the CCR.

Example 2 The CCR partitions $\mathbf{B}(5)$ into the following four cycles $\mathbf{C}_1, \mathbf{C}_2, \mathbf{C}_3, \mathbf{C}_4$ where the top string in bold is the RL-rep for the given cycle.

$\mathbf{C}_1$	$\mathbf{C}_2$	$\mathbf{C}_3$	$\mathbf{C}_4$
10101	**11101**	**11001**	**11111**
01010	11010	10010	11110
	10100	00100	11100
	01000	01001	11000
	10001	10011	10000
	00010	00110	00000
	00101	01101	00001
	01011	11011	00011
	10111	10110	00111
	01110	01100	01111

The CCR has been applied to efficiently construct de Bruijn sequences in variety of ways [7, 20, 16]. An especially efficient construction applies a concatenation scheme to construct a de Bruijn sequence with discrepancy (maximum difference between the number of 0s and 1s in any prefix) bounded above by $2n$ [14, 15].

5 The pure run-length register (PRR)

The feedback function of particular focus in this paper is $f(\omega) = w_1 \oplus w_2 \oplus w_n$. We will demonstrate that FSR based on this feedback function partitions $\mathbf{B}(n)$ into cycles of strings with the same run-length. Because of

this property, we call this FSR the *pure run-length register* and denote it by PRR. This follows the naming of the pure cycling register (PCR) and the pure summing register (PSR), which is based on the feedback function $f(\omega) = w_1 \oplus w_2 \oplus \cdots \oplus w_n$ [18].

For the remainder of this paper, let $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_t$ denote the cycles induced by the PRR on $\mathbf{B}(n)$. It is easily observed that each cycle $\mathbf{R}_i$ has an interesting property: either all the strings start and end with the same bit, or all the strings start and end with different bits. In the former cycle, if we remove the last bit of each string we obtain a cycle of the PCR of order $n-1$. In the latter cycle, if we remove the last bit of each string we obtain a cycle of the CCR of order $n-1$. These observations were first made in [22] and are illustrated in Example 3 below. Furthermore, all the strings in a given $\mathbf{R}_i$ have the same run-length.

▶ **Lemma 3.** *All the strings in a given cycle $\mathbf{R}_i$ have the same run-length.*

Proof. Consider a string $\omega = w_1 w_2 \cdots w_n$ and the feedback function $f(\omega) = w_1 \oplus w_2 \oplus w_n$. It suffices to show that $w_2 \cdots w_n f(\omega)$ has the same run-length as ω. This is easily observed since if $w_1 = w_2$ then $w_n = f(\omega)$ and if $w_1 \neq w_2$ then $w_n \neq f(\omega)$. ◀

Recall that the RL-rep of a given $\mathbf{R}_i$ is defined to be the string with the lexicographically largest run-length encoding, and if there are two such strings, the RL-rep is the string beginning with 1. For each PRR, its RL-rep has properties relating to the two cases described earlier: if the first and last bits of each string are the same then the RL-rep is the string $s_1 s_2 \cdots s_n$ such that $s_1 s_2 \cdots s_{n-1}$ is an RL-rep with respect to the PCR; otherwise the RL-rep is the string $s_1 s_2 \cdots s_n$ such that $s_1 s_2 \cdots s_{n-1}$ is an RL-rep with respect to the CCR.

Example 3 The PRR partitions $\mathbf{B}(6)$ into the following 12 cycles $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_{12}$ where the top string in bold is the RL-rep for the given cycle. The cycles are ordered in non-increasing order with respect to the run-lengths of their RL-reps.

$\mathbf{R}_1$	$\mathbf{R}_2$	$\mathbf{R}_3$	$\mathbf{R}_4$	$\mathbf{R}_5$	$\mathbf{R}_6$	$\mathbf{R}_7$	$\mathbf{R}_8$	$\mathbf{R}_9$	$\mathbf{R}_{10}$
101010	**110101**	**001010**	**111010**	**110010**	**111101**	**000010**	**111001**	**000110**	**111110**
010101	101011	010100	110100	100100	111011	000100	110011	001100	111100
	010110	101001	101000	001001	110111	001000	100111	011000	111000
	101101	010010	010001	010011	101111	010000	001110	110001	110000
	011010	100101	100010	100110	011110	100001	011100	100011	100000
			000101	001101					000001
			001011	011011					000011
			010111	110110					000111
			101110	101100					001111
			011101	011001					011111

$\mathbf{R}_{11}$	$\mathbf{R}_{12}$
111111	**000000**

By omitting the last bit of each string, the columns are precisely the cycles of the PCR and CCR for $n = 5$. The cycles $\mathbf{R}_1, \mathbf{R}_4, \mathbf{R}_5, \mathbf{R}_{10}$ relating to the CCR start and end with the different bits. The remaining cycles relate to the PCR. They start and end with the same bits.

Let $\mathbf{RL}(n)$ denote the set of all RL-reps with respect to the PRR of order n. For the rest of the paper, anytime we use the term RL-rep, it is assumed to be with respect to the PRR.

5.1 De Bruijn successors based on joining cycles of the PRR

Define the *conjugate* of $\omega = w_1 w_2 \cdots w_n$ to be $\hat{\omega} = \overline{w_1} w_2 \cdots w_n$, where $\overline{w_i}$ denotes the complement of the bit w_i. By applying the framework from [16], the cycles of the PRR can be systematically joined together to produce a variety of de Bruijn successors. After the cycles induced by the PRR are ordered $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_t$, the framework only requires defining a representative α_i for each $\mathbf{R}_i$ for $i > 1$ such that its conjugate $\hat{\alpha}_j$ belongs to some $\mathbf{R}_j$ where $j < i$. Together, the ordering of the cycles and the spanning sequence effectively describe a rooted tree where the nodes are the cycles $\mathbf{R}_i$ with $\mathbf{R}_1$ designated as the root. There is an edge between two nodes $\mathbf{R}_i$ and $\mathbf{R}_j$ where $j < i$, if $\hat{\alpha}_i$ is in $\mathbf{R}_j$. Each edge effectively represents the joining of two cycles (recall Hierholzer's Euler cycle algorithm).

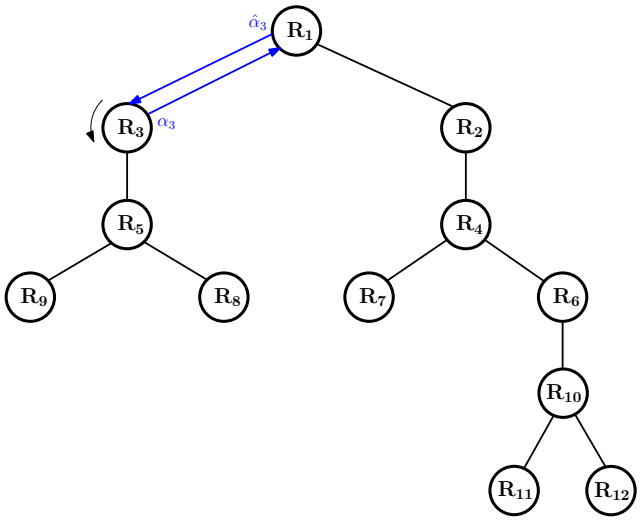


Figure 3 Illustrating the joining of the cycles $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_{12}$ listed in Example 3 based on the de Bruijn successor g_r. Note that the conjugate $\hat{\alpha}_3$ of the RL-rep α_3 is in found in $\mathbf{R}_1$.

As an application of this framework, consider the cycles $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_t$ to be ordered in non-increasing order based on the run-length of the strings in each cycle. Such an ordering is given in Example 3. Using this ordering, each $\alpha_i = a_1 a_2 \cdots a_n$ can be any string in $\mathbf{R}_i$ such that $a_1 = a_2$. Such a string clearly exists since $i > 1$. Its conjugate $\hat{\alpha}_i$ clearly has run-length that is one more than the run-length of α_i and thus belongs to some $\mathbf{R}_j$ where $j < i$. Thus, by applying Theorem 3.5 from [16], the following describes a generic de Bruijn successor based on the PRR.

▶ **Theorem 4.** *Let $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_t$ be listed in non-increasing order with respect to the run-length of the strings in each cycle. Let $\alpha_i = a_1 a_2 \cdots a_n$ denote a representative in $\mathbf{R}_i$ such that $a_1 = a_2$, for each $1 < i \leq t$. Let $\omega = w_1 w_2 \cdots w_n$ and let $f(\omega) = w_1 \oplus w_2 \oplus w_n$. Then the function:*

$$g(\omega) = \begin{cases} \overline{f(\omega)} & \text{if } \omega \text{ or } \hat{\omega} \text{ is in } \{\alpha_2, \alpha_3, \ldots, \alpha_t\}; \\ f(\omega) & \text{otherwise.} \end{cases}$$

is a de Bruijn successor.

When this theorem is applied generically, it is not efficient since storing the set $\{\alpha_2, \alpha_3, \ldots, \alpha_t\}$ requires exponential space. However, if a membership tester for the set can be defined efficiently, then there is no need for the set to be stored. As an example, if each representative α_i of $\mathbf{R}_i$ corresponds to its RL-rep, then since $i > 1$, the representative will begin with 00 or 11. Thus, the run-length of the conjugate $\hat{\alpha}_i$ will be one more than the run-length of α_i. This leads directly to the following corollary.

▶ **Corollary 5.** *Let $\omega = w_1 w_2 \cdots w_n$ and let $f(\omega) = w_1 \oplus w_2 \oplus w_n$. The function*

$$g_r(\omega) = \begin{cases} \overline{f(\omega)} & \text{if } \omega \text{ or } \hat{\omega} \text{ is in } \mathbf{RL}(n); \\ f(\omega) & \text{otherwise,} \end{cases}$$

is a de Bruijn successor.

An illustration of how the de Bruijn successor g_r joins the 12 cycles for $n = 6$ is given in Figure 3, based on the cycle listing from Example 3.

6 Efficiently constructing the prefer-same de Bruijn sequence $\mathcal{S}_n$

In this section we present a de Bruijn successor that can be used to construct $\mathcal{S}_n$ in $O(n)$ time per bit using $O(n)$ space. We begin by presenting a de Bruijn successor for $\mathcal{L}_n$ that is used as a building block in our successor-rule

for $\mathcal{S}_n$. We conclude the section with a brief discussion of implementation details.

6.1 A de Bruijn successor for $\mathcal{L}_n$

As noted by Fredricksen and Kessler [12], the de Bruijn sequence $\mathcal{L}_n$, which is based on concatenating lexicographic compositions, is very close to the prefer-same sequence $\mathcal{S}_n$. At the end of their paper they conjecture it is different only at the tail end. We demonstrate that their algorithm also has a successor-rule interpretation based on the PRR. We do not formally prove this, but have have verified that the upcoming de Bruijn successor can be applied to construct $\mathcal{L}_n$ for $n < 30$.

Recall the partition $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_t$ of $\mathbf{B}(n)$ induced by the PRR. Let $\text{PRR}^j(\omega)$ denote j applications of the PRR starting with ω for $j \geq 0$. We define a new representative for each cycle $\mathbf{R}_i$ from its RL-rep σ_i as follows. Let r_i denote the number of consecutive 1s at the end of the run-length encoding of σ_i. For example, if $\sigma_i = 111101110101$, then its run-length encoding is $4131\underline{1111}$ and $r_i = 4$. We define the *LC-rep* of $\mathbf{R}_i$ to be the string α_i such that $\text{PRR}^{r_i+1}(\alpha_i) = \sigma_i$. Thus, from our example $\alpha_i = \underline{110101}1111011$. Let $\mathbf{LC}(n)$ denote the set of all LC-reps of order n. From the definition of this representative, the run-length encoding of each α_i will begin with 2 except for the case when α_i has run-length n. Thus, when $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_t$ is in non-increasing order with respect to the run-length of the strings in each cycle, the run-length of the conjugate $\hat{\alpha}_i$ will be one greater than α_i, for $i > 1$. Thus, we can immediately apply Theorem 4 to obtain the following de Bruijn successor.

▶ **Corollary 6.** *Let $\omega = w_1 w_2 \cdots w_n$ and let $f(\omega) = w_1 \oplus w_2 \oplus w_n$. The function*

$$g_{lc}(\omega) = \begin{cases} \overline{f(\omega)} & \text{if } \omega \text{ or } \hat{\omega} \text{ is in } \mathbf{LC}(n); \\ f(\omega) & \text{otherwise,} \end{cases}$$

is a de Bruijn successor.

6.2 A de Bruijn successor for $\mathcal{S}_n$

The successor rule g_{lc} comes very close to being a de Bruijn successor for $\mathcal{S}_n$. In fact, there are only a small number of representatives that need to be changed in the set $\mathbf{LC}(n)$. The key to understanding these cases comes down to refining the partition of $\mathbf{B}(n)$ with respect to the PRR, and then investigating properties of the LC-reps with respect to this ordering.

Recall we had ordered the partition $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_t$ of $\mathbf{B}(n)$ induced by the PRR in non-increasing order with respect to the run-length of the strings in each cycle. We further refine this ordering for the cycles containing strings with the same run-length; we order them in reverse lexicographic order with respect to their RL-reps. With respect to properties of this refined ordering, the key to modifying the LC-reps comes down to finding the cycles $\mathbf{R}_i$ where the conjugates of the LC-rep and the RC-rep belong to the same cycle $\mathbf{R}_j$. Understanding the underlying reasons for this requires a much more in-depth study that will be presented in the full version of this paper. It turns out that these special cycles are the ones with RL-rep beginning and ending with 0 (which means their run-length is odd) with run-length encoding of the form:

$(21^{2x})^y 1^z$, where $x \geq 0$, $y \geq 2$, and $z \geq 2$.

We call such RL-reps *special*. When n is even, this implies that y is odd and z is even. When n is odd, it implies that y is even and z is odd.

Example 4 The run-length encoding for the special RL-reps for $n = 10, 11, 12, 13$.

$n = 10$: 2221111
$n = 11$: 2222111, 221111111, 211211111
$n = 12$: 2222211, 222111111
$n = 13$: 222211111, 22111111111, 21121111111

For these very few cycles, we replace the LC-rep with its RL-rep. Thus, we define the *same-rep* of each $\mathbf{R}_i$ to be the LC-rep, except when the RL-rep is special, in which case it is defined to be the RL-rep. Let $\mathbf{Same}(n)$

denote the set of all same-reps of order n.

Again, we can apply Theorem 5.2 to obtain a de Bruijn successor.

▶ **Corollary 7.** *Let $\omega = w_1 w_2 \cdots w_n$ and let $f(\omega) = w_1 \oplus w_2 \oplus w_n$. The function*

$$g_s(\omega) = \begin{cases} \overline{f(\omega)} & \text{if } \omega \text{ or } \hat{\omega} \text{ is in } \mathbf{Same}(n); \\ f(\omega) & \text{otherwise,} \end{cases}$$

is a de Bruijn successor.

Recall that $alt(n)$ denotes the alternating sequence of 0s and 1s of length n that ends with 0. Let $\mathcal{X}_n = x_1 x_2 \cdots x_{2^n}$ be the de Bruijn sequence returned by $\mathrm{DB}(g_s, 0alt(n-1))$. It remains to show that $\mathcal{X}_n = \mathcal{S}_n$. Before giving a proof we state some preliminary facts about the sequence $\mathcal{X}_n$ and same-reps. The first two facts are trivial to observe.

▶ **Fact 8.** *If α_i is a same-rep for $\mathbf{R}_i$ where $i > 1$, then the run-length of α_i is one less than the run-length of $\hat{\alpha}_i$.*

▶ **Fact 9.** *$\mathcal{X}_n$ has prefix 1^n and has suffix $alt(n-1)$.*

Proofs for the next two facts are omitted. They rely on a deeper study of the RL-reps of the conjugates for each string in a given cycle $\mathbf{R}_i$, and the specific ordering of the cycles $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_t$ noted above.

▶ **Fact 10.** *If $\alpha_i = a_1 a_2 \cdots a_n$ is a same-rep for $\mathbf{R}_i$ where $i > 1$, $\hat{\alpha}_i$ comes before α_i in $alt(n-1)\mathcal{X}_n$.*

▶ **Fact 11.** *If the run-length of $\omega = w_1 w_2 \cdots w_n$ is one more than the run-length of $\hat{\omega}$ and neither ω nor $\hat{\omega}$ are same-reps, then $\hat{\omega}$ comes before ω in $alt(n-1)\mathcal{X}_n$.*

We apply these facts to prove our main result.

▶ **Theorem 12.** *The de Bruijn sequences $\mathcal{S}_n$ and $\mathcal{X}_n$ are the same.*

Proof. Let $\mathcal{S}_n = s_1 s_2 \cdots s_{2^n}$, let $\mathcal{X}_n = x_1 x_2 \cdots x_{2^n}$. From Fact 2 and Fact 9, $s_1 \cdots s_n = x_1 \cdots x_n = 1^n$ and both $\mathcal{S}_n$ and $\mathcal{X}_n$ end with $alt(n-1)$. Suppose there exists some $n < t \leq 2^n - n$ such that $s_t \neq x_t$. Let $\alpha = x_{t-n} \cdots x_{t-1}$ denote the length n substring of $\mathcal{X}_n$ ending at position $t-1$. Then $x_t \neq x_{t-1}$, because otherwise the run-length encoding of $\mathcal{X}_n$ is lexicographically larger than that of $\mathcal{S}_n$, contradicting Fact 1. We claim that $\hat{\alpha}$ comes before α in $alt(n-1)\mathcal{X}_n$, by considering two cases:

- If $x_t = \mathrm{PRR}(\alpha)$, then by the definition of g_s, neither α nor $\hat{\alpha}$ are in $\mathbf{Same}(n)$. Thus the claim holds by Fact 11.
- If $x_t \neq \mathrm{PRR}(\alpha)$, then either α or $\hat{\alpha}$ are in $\mathbf{Same}(n)$. Since α and $a_2 \cdots a_n s_t$ are on the same cycle induced by the PRR which means that $a_2 \cdots a_n x_t$ has run-length one greater than that of α. From Fact 8, this implies $\hat{\alpha}$ is not a same-rep, which means α is a same-rep. The claim thus holds by Fact 10.

If $\hat{\alpha}$ appears before α in $alt(n-1)\mathcal{X}_n$ then it must be a substring of $alt(n-1)x_1 \cdots x_{t-2}$. Thus, either $x_{t-n+1} \cdots x_{t-1} x_t$ or $x_{t-n+1} \cdots x_{t-1} s_t$ must be in $alt(n-1)x_1 \cdots x_{t-1}$ which contradicts the fact that both $\mathcal{X}_n$ and $\mathcal{S}_n$ are de Bruijn sequences. Thus, there is no $n < t \leq 2^n$ such that $s_t \neq x_t$ and hence $\mathcal{S}_n = \mathcal{X}_n$. ◀

Implementations have also verified this correctness of this theorem for $n \leq 30$.

6.3 Implementation details

The key to an efficient implementation of the successors g_r, g_{lc} and g_s is an efficient membership tester for the set $\mathbf{RL}(n)$. Using relatively standard techniques, this tester can be implemented in $O(n)$-time. With this tester, only linear time and space is required to develop a membership tester for $\mathbf{LC}(n)$ and subsequently a membership tester for $\mathbf{Same}(n)$ in a straightforward manner. More complete implementation details will be given in the full version of this paper.

▶ **Theorem 13.** *The de Bruijn successors g_r, g_{lc} and g_s can be implemented in $O(n)$-time using $O(n)$ space.*

7 Summary

The goal of this research was to unveil the underlying structure of the Prefer-same de Bruijn sequence $\mathcal{S}_n$. In doing so, we presented a simple successor-rule that can generate the sequence using $O(n)$ time per bit and $O(n)$ space. This solves a 60 year old open problem. The details of such an efficient implementation will be provided in the full version of this extended abstract. The efficient algorithm was based on understanding how the cycles of the PRR, which is based on the feedback function $f(w_1 w_2 \cdots w_n) = w_1 \oplus w_2 \oplus w_n$, can be joined together by studying their run-length properties.

Although not presented in this paper, we have also decoded the greedy Prefer-opposite de Bruijn sequence [1]. It is also based on the same underlying feedback function, but requires an alternate ordering of the induced cycles.

References

1 A. Alhakim. A simple combinatorial algorithm for de Bruijn sequences. *The American Mathematical Monthly*, 117(8):728–732, 2010.

2 A. Alhakim. Spans of preference functions for de Bruijn sequences. *Discrete Applied Mathematics*, 160(7-8):992 – 998, 2012.

3 A. Alhakim, E. Sala, and J. Sawada. Revisiting the prefer-same and prefer-opposite de Bruijn sequence constructions. *Theoretical Computer Science*, 2020 (to appear).

4 N. G. de Bruijn. A combinatorial problem. *Indagationes Mathematicae*, 8:461–467, 1946.

5 P. B. Dragon, O. I. Hernandez, J. Sawada, A. Williams, and D. Wong. Constructing de Bruijn sequences with co-lexicographic order: the k-ary Grandmama sequence. *European J. Combin.*, 72:1–11, 2018.

6 C. Eldert, H. Gray, H. Gurk, and M. Rubinoff. Shifting counters. *AIEE Trans.*, 77:70–74, 1958.

7 T. Etzion. Self-dual sequences. *Journal of Combinatorial Theory, Series A*, 44(2):288 – 298, 1987.

8 M. Fleury. Deux problemes de geometrie de situation. *Journal de mathematiques elementaires*, 42:257–261, 1883.

9 C. Flye Sainte-Marie. Solution to question nr. 48. *L'intermédiaire des Mathématiciens*, 1:107–110, 1894.

10 H. Fredricksen. Generation of the Ford sequence of length 2^n, n large. *Journal of Combinatorial Theory, Series A*, 12(1):153 – 154, 1972.

11 H. Fredricksen. A survey of full length nonlinear shift register cycle algorithms. *Siam Review*, 24(2):195–221, 1982.

12 H. Fredricksen and I. Kessler. Lexicographic compositions and de Bruijn sequences. *J. Combin. Theory Ser. A*, 22(1):17 – 30, 1977.

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

14 D. Gabric and J. Sawada. Constructing de Bruijn sequences by concatenating smaller universal cycles. *Theoretical Computer Science*, 743:12 – 22, 2018.

15 D. Gabric and J. Sawada. Investigating the discrepancy property of de Bruijn sequences. *Submitted manuscript*, 2020.

16 D. Gabric, J. Sawada, A. Williams, and D. Wong. A framework for constructing de Bruijn sequences via simple successor rules. *Discrete Mathematics*, 341(11):2977 – 2987, 2018.

17 D. Gabric, J. Sawada, A. Williams, and D. Wong. A successor rule framework for constructing k -ary de Bruijn sequences and universal cycles. *IEEE Transactions on Information Theory*, 66(1):679–687, 2020.

18 S. W. Golomb. *Shift Register Sequences*. Aegean Park Press, Laguna Hills, CA, USA, 1981.

19 C. Hierholzer. Deux problemes de geometrie de situation. *Journal de mathematiques elementaires*, 42:257–261, 1873.

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

21 M. H. Martin. A problem in arrangements. *Bull. Amer. Math. Soc.*, 40(12):859–864, 1934.

22 E. Sala. Exploring the greedy constructions of de Bruijn sequences. Master's thesis, University of Guelph, 2018.

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

24 A. Williams. The greedy Gray code algorithm. In F. Dehne, R. Solis-Oba, and J.-R. Sack, editors, *Algorithms and Data Structures*, pages 525–536, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

25 S. Xie. Notes on de Bruijn sequences. *Discrete Applied Mathematics*, 16(2):157 – 177, 1987.

A Proving technical results

A.1 Proof of Fact 10

We actually prove a more general result that could apply to any de Bruijn successor resulting from the framework from [16]. In particular, consider any de Bruijn successor g derived from Theorem 12 with representatives $\alpha_2, \alpha_3, \ldots, \alpha_t$ and a de Bruijn sequence $\mathcal{D}_n = \mathrm{DB}(g, \omega)$, where $\omega = w_1 w_2 \cdots w_n$ and $w_2 w_3 \cdots w_n g(\omega)$ is in $\mathbf{R}_1$. Then based on the tree like construction that joins the cycles (see Figure 3), clearly the conjugates $\hat{\alpha}_i$ of each representative α_i will appear before α_i itself in the sequence $w_2 \cdots w_n \mathcal{D}_n$.

A.2 Proof of Fact 11

Let $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_t$ be the cycles induced by the PRR ordered in non-increasing order with respect to the run-lengths of the strings in each cycle, then refined so the cycles with the same run-lengths are ordered in decreasing order with respect to the run-length encoding of the RL-rep. If two reps have the same run-length encoding (RLE), then the cycle with RL-rep starting with 1 comes first. Let $\alpha_i, \gamma_i, \sigma_i$ denote the same-rep, LC-rep, and the RL-rep, respectively, for $\mathbf{R}_i$.

Recall that $\mathcal{X}_n = \mathrm{DB}(g, \omega)$, where $\omega = 0alt(n-1) = w_1 w_2 \cdots w_n$. Note that $w_2 w_3 \cdots w_n g(\omega)$ is the length n alternating string of 0s and 1s ending with 1; it is a string in $\mathbf{R}_1$. Thus, the first string of length n in the linearized de Bruijn sequence $alt(n-1)\mathcal{X}_n$ is in $\mathbf{R}_1$. Then based on the related tree of cycles (see Figure 3) rooted at $\mathbf{R}_1$, we make the following observation that can easily be generalized to any de Bruijn sequence constructed using the cycle-joining framework from [16]. This strengthens the result proved in Fact 10.

▶ **Observation 14.** *Consider $\mathbf{R}_i$ for $i > 1$ containing m strings. Then with respect to $alt(n-1)\mathcal{X}_n$ we have:*

1. *$\hat{\alpha}_i$ appears before all strings in $\mathbf{R}_i$,*
2. *α_i appears last amongst all strings in $\mathbf{R}_i$,*
3. *the strings of $\mathbf{R}_i$ appear in the following order: $PRR(\alpha_i), PRR^2(\alpha_i), \ldots, PRR^m(\alpha_i) = \alpha_i$,*
4. *the strings in all descendant cycles of $\mathbf{R}_i$ appear after $\hat{\alpha}_i$ and before α_i.*

Let $\mathbf{Special}(n)$ denote the set of strings of length n that begin and end with 0 with RLE of the form $(21^{2x})^y 1^z$, where $x \geq 0$, $y \geq 2$, and $z \geq 2$. The following example illustrates a key observation as to why $\mathcal{L}_N \neq \mathcal{S}_N$, and is elaborated on in the next lemma.

Example 5 Consider $\mathbf{R}_i$ and $\mathbf{R}_j$ with RL-reps $\sigma_i = 00101101010$ and $\sigma_j = 11010010101$, respectively. Both have RLEs 211211111. By definition only the first is in $\mathbf{Special}(11)$. The corresponding LC-reps are $\gamma_i = 11010100101$ and $\gamma_j = 00101011010$. All four strings belong to the same cycle $\mathbf{R}_k$. Importantly, this only happens for these special RLEs. Below is the order the strings in $\mathbf{R}_k$ appear in $alt(11-1).\mathcal{X}_{11}$, based on Observation 14. In particular take notice of the positions of the four conjugates.

01010101011
10101010110
01010101101
10101011010 ← $\hat{\gamma}_j$
01010110101
10101101010 ← $\hat{\sigma}_i$
01011010101
10110101010
01101010101
11010101010 ← σ_k, the RL-rep, with run length 2111111111
10101010100
01010101001
10101010010
01010100101 ← $\hat{\gamma}_i$
10101001010
01010010101 ← $\hat{\sigma}_j$
10100101010
01001010101
10010101010
00101010101 ← $\alpha_k = \gamma_k$, the same-rep and LC-rep for this cycle

Since $\hat{\sigma}_i$ appears before $\hat{\gamma}_i$ in the above ordering, σ_i (RL-rep) is the same-rep for $\mathbf{R}_i$. Since $\hat{\gamma}_j$ appears before $\hat{\sigma}_j$, γ_i (LC-rep) is the same-rep for $\mathbf{R}_j$.

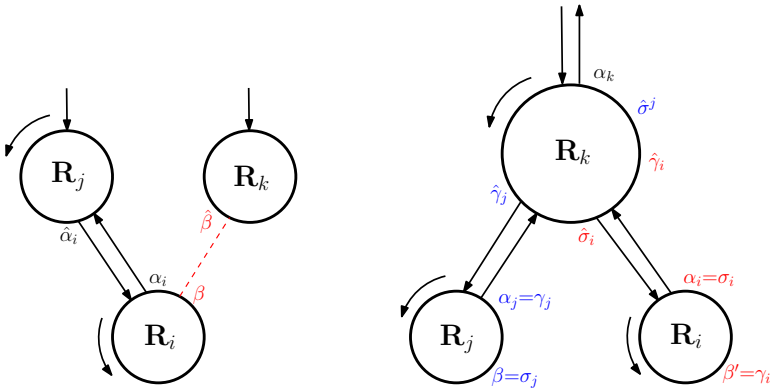


Figure 4 (a) Illustrating Lemma 17 where either (i) α_i is special and $\beta \neq \gamma_i$ or (ii) α_i is not special and either $\overline{\alpha_i}$ is not special or $\beta \neq \sigma_j$. (b) Illustrating Lemma 16 where α_i is special and $\beta' = \gamma_i$, and when $\alpha_j = \overline{\alpha_i}$ is special and $\beta = \sigma_j$.

▶ Remark 15. If β is in $\mathbf{Special}(n)$, then β is the RL-rep and same-rep for some $\mathbf{R}_i$. Furthermore, $\overline{\beta_1}$ is also an RL-rep for some $\mathbf{R}_j$.

▶ **Lemma 16.** *Let $\mathbf{R}_i$ be a cycle such that $\alpha_i = \sigma_i \in \mathbf{Special}(n)$. Let $\sigma_j = \overline{\sigma_i}$ be the RL-rep for $\mathbf{R}_j$. Then $\hat{\gamma}_i$ and $\hat{\sigma}_i$ along with $\hat{\gamma}_j$ and $\hat{\sigma}_j$ belong to the same $\mathbf{R}_k$. Moreover in $alt(n{-}1)\mathcal{X}_n$, (a) $\hat{\sigma}_i$ appears before $\hat{\gamma}_i$ and (b) $\hat{\gamma}_j$ appears before $\hat{\sigma}_j$.*

Proof. Since $\sigma_i = s_1 s_2 \cdots s_n \in \mathbf{Special}(n)$, it has RLE of the form $(21^{2x})^y 1^z$, where $x \geq 0$, $y \geq 2$, and $z \geq 2$ and $s_1 = s_n = 0$ (the run-length is even). Let $\hat{\sigma}_i$ be in $\mathbf{R}_k$. It will have RLE $1^{2x+2}(21^{2x})^{y-1}1^z$ where the first bit is 1. Since, $\mathbf{R}_k$ relates to a CCR cycle (it has odd run-length), and considering the RLE $\hat{\sigma}_i$, the $\mathbf{R}_k$ will have $2n-2$ distinct strings. Thus σ_k begins with 1 and has the RLE $(21^{2x})^{y-1}1^{z+2x+2}$, and furthermore $\mathrm{PRR}^{2x+2}(\hat{\sigma}_i) = \sigma_k$. This means that $\mathrm{PRR}^{(z+2x+2)+2x+1}(\gamma_k) = \sigma_k$ by the definition of the LC-rep. Note clearly that $\alpha_k = \gamma_k$ since the cycle is a CCR based cycle - the strings begin and end with different symbols since the RLE of σ_k is even. By the definition of LC-rep and noting the parity conditions on y and z, γ_i will have RLE $(21^{2x+t-1})(21^{2x})^{y-1}1$ beginning with 1. Thus $\hat{\gamma}_i$ begins with 0 and has RLE $1^{2x+t+1}(21^{2x})^{y-1}1$. Observe that $\mathrm{PRR}^{2x+t+1}(\hat{\gamma}_i)$ will have RLE $(21^{2x})^{y-1}1^{z+2x+2}$ but beginning with 0; it is the complement of σ_k. Since $\mathbf{R}_k$ is related to the CCR, both of these strings are on the same cycle and $n-1$ strings apart. By considering the relative positions of these strings in $\mathbf{R}_i$ based on the details above, we can apply Observation 14 (3) to see that $\hat{\sigma}_i$ appears before $\hat{\gamma}_i$ in $alt(n{-}1)\mathcal{X}_n$. A similar analysis hold for σ_j. The result is illustrated in Figure 4 (b). ◀

▶ **Lemma 17.** *Let $\beta \in \mathbf{R}_i - \{\alpha_i\}$, where $i > 1$ such that either (i) α_i is special and $\beta \neq \gamma_i$ or (ii) α_i is not special and either $\overline{\alpha_i}$ is not special or $\beta \neq \sigma_j$. If $\hat{\beta}$ has run-length that is one more than the run-length of β, then $\hat{\alpha}_i$ belongs to a cycle $\mathbf{R}_j$ and $\hat{\beta}$ belongs to a cycle $\mathbf{R}_k$ such that $\sigma_j > \sigma_k$ with both strings beginning with the same symbol.*

Proof. By considering the RLE for β, and under the conditions provided, this result is observed by carefully considering the RLE of both $\hat{\beta}$ and the RLE of $\hat{\alpha}_i$. From these RLEs, it is not difficult to see that $\sigma_j > \sigma_k$ with both strings beginning with the same symbol. We leave it to the reader to verify. The result is illustrated in Figure 4 (a). ◀

The following remark follows easily by considering the related tree of cycles and Observation 14

▶ Remark 18. If both $\mathbf{R}_i$ and $\mathbf{R}_j$ have strings with the same run-length r, then either every string in $\mathbf{R}_i$ comes before every string in $\mathbf{R}_j$ in $alt(n{-}1)\mathcal{X}_n$ or vice-versa.

Applying this remark, and performing a simple induction on the tree of nodes based on levels, focussing on each α_i, we obtain the following lemma.

▶ **Lemma 19.** *If $\mathbf{R}_i$ and $\mathbf{R}_j$ contain strings with the same run-length where $i < j$ and their respective RL-reps begin with the same bit value, then every string in $\mathbf{R}_i$ appears in $alt(n-1)\mathcal{X}_n$ before any string in $\mathbf{R}_j$.*

The fact we are to prove can be restated as: If the run-length of $\beta \in \mathbf{R}_i$ is one less than the run-length of $\hat{\beta}$ and neither β nor $\hat{\beta}$ are same-reps, then β comes before $\hat{\beta}$ in $alt(n-1)\mathcal{X}_n$. The cases for β can be visualized in Figure 4.

PROOF OF FACT 11. Suppose $\beta \in \mathbf{R}_i$ and $\hat{\beta} \in \mathbf{R}_k$. Since they are not same reps $\beta \neq \alpha_i$. Let $\mathbf{R}_j$ contain $\hat{\alpha}_j$. Note that the strings in $\mathbf{R}_j$ and $\mathbf{R}_k$ have run-length. If (a) α_i is special and $\beta_i = \gamma_i$ or (b) $\overline{\alpha_i}$ is special and $\beta_i = \sigma_i$, then $j = k$ and by Lemma 16 β comes before $\hat{\beta}$ in $alt(n-1)\mathcal{X}_n$. For the remaining cases, we can apply Lemma 17, together with Lemma 19 to obtain β comes before $\hat{\beta}$ n $alt(n-1)\mathcal{X}_n$. □

Congratulations if you made it here, we hope you enjoyed this read!

B Implementation of the de Bruijn successors in C

```
#include<stdio.h>
#include<math.h>
#define N_MAX 50
int n;

// ============================================================================
// Compute the RLE of a[1..m] in run[1..r], returning r = ruh length
// ============================================================================
int RLE(int a[], int run[], int m) {
     int i,j,r,old;

     old = a[m+1];
     a[m+1] = 1 - a[m];
     r = j = 0;
     for (i=1; i<=m; i++) {
          if (a[i] == a[i+1]) j++;
          else {  run[++r] = j+1;  j = 0;  }
     }
     a[m+1] = old;
     return r;
}
// ============================================================================
// Check if a[1..n] is a "special" RL representative.  It must be that a[1] = a[n]
// and the RLE of a[1..n] is of the form (21^j)^s1^t where j is even, s >=2, t>=2
// ============================================================================
int Special(int a[]) {
     int i,j,r,s,t,run[N_MAX];

     if (a[1] != 0  || a[n] != 0) return 0;
     r = RLE(a,run,n);

     // Compute j of prefix 21^j
     if (run[1] != 2) return 0;
     j = 0;
     while (run[j+2] == 1 && j+2 <= r) j++;

     // Compute s of prefix (21^j)^s
     s = 1;
     while (s <= r/(1+j) -1 && run[s*(j+1)+1] == 2) {
          for (i=1; i<=j; i++) if (run[s*(j+1)+1+i] != 1) return 0;
          s++;
     }

     // Test remainder of string is (21^j)^s is 1^t
     for (i=s*(j+1)+1; i<=r; i++) if (run[i] != 1) return 0;
     t = r - s*(1+j);

     if (s >= 2 && t >= 2 && j%2 == 0) return 1;
     return 0;
}
// ============================================================================
// Apply PRR^{t+1} to a[1..n] to get b[1..n], where t is the length of the
// prefix before the first 00 or 11 in a[2..n] up to n-2
// ============================================================================
int Shift(int a[], int b[]) {
     int i,t = 0;
     while (a[t+2] != a[t+3] && t < n-2) t++;
     for (i=1; i<=n; i++) b[i] = a[i];
     for (i=1; i<=n; i++) b[i+n] = (b[i] + b[i+1] + b[n+i-1]) % 2;
     for (i=1; i<=n; i++) b[i] = b[i+t+1];
     return t;
}
// ============================================================================
// Test if b[1..len] is the lex largest rep (under rotation), if so, return the
// period p; otherwise return 0. Eg. (411411, p=3)(44211, p=5) (411412, p=0).
// ============================================================================
int IsLargest(int b[], int len) {
     int i, p=1;
     for (i=2; i<=len; i++) {
          if (b[i-p] < b[i]) return 0;
          if (b[i-p] > b[i]) p = i;
     }
     if (len % p != 0) return 0;
     return p;
}
```

```
// ================================================================================
// Membership testers not including the cycle containing 0101010...
// ================================================================================
int RLrep(int a[]) {
    int p,r,rle[N_MAX];

    r = RLE(a,rle,n-1);
    p = IsLargest(rle,r);

    // PCR-related cycle
    if (a[1] == a[n]) {
        if (r == n-1 && a[1] == 1) return 0;  // Ignore root a[1..n] = 1010101..
        if (r == 1) return 1;  // Special case: a[1..n] = 000..0 or 111..1
        if (p > 0 && a[1] != a[n-1] && (p == r || a[1] == 1 || p%2 == 0)) return 1;
    }
    // CCR-related cycle
    if (a[1] != a[n]) {
        if (p > 0 && a[1] == 1 && (a[n-1] == 1)) return 1;
    }
    return 0;
}
// ================================================================================
int LCrep(int a[]) {
    int b[N_MAX];

    if (a[1] != a[2]) return 0;
    Shift(a,b);
    return RLrep(b);
}
// ================================================================================
int SameRep(int a[]) {
    int b[N_MAX];

    Shift(a,b);
    if (Special(a) || (LCrep(a) && !Special(b))) return 1;
    return 0;
}
// ================================================================================
// Repeatedly apply the Prefer-Same or LC or RL successor rule starting with 1^n
// ================================================================================
void DB(int type) {
    int i,j,v,a[N_MAX],REP;

    for (i=1; i<=n; i++) a[i] = 1;  // Initial string

    for (j=1; j<=pow(2,n); j++) {
        printf("%d", a[1]);

        v = (a[1] + a[2] + a[n]) % 2;
        REP = 0;
        // Membership testing of a[1..n]
        if (type == 1 && SameRep(a)) REP = 1;
        if (type == 2 && LCrep(a)) REP = 1;
        if (type == 3 && RLrep(a)) REP = 1;


        // Membership testing of conjugate of a[1..n]
        a[1] = 1 - a[1];
        if (type == 1 && SameRep(a)) REP = 1;
        if (type == 2 && LCrep(a)) REP = 1;
        if (type == 3 && RLrep(a)) REP = 1;

        // Shift String and add next bit
        for (i=1; i<n; i++) a[i] = a[i+1];
        if (REP) a[n] = 1 - v;
        else a[n] = v;
    }
}
//------------------------------------------------------
int main() {
    int type;

    printf("Enter (1) Prefer-same (2) LC (3) Run-length: ");  scanf("%d", &type);
    printf("Enter n: ");   scanf("%d", &n);

    DB(type);
}
```