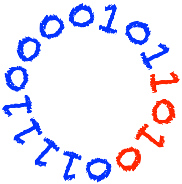
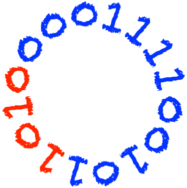
De Bruijn Sequence Constructions



Joe Sawada
University of Guelph
CANADA



UNIVERSITÀ DI VERONA
June 18, 2019

Canada ⇒ Toronto ⇒ Guelph (Fun facts)

- Canada 2nd largest country by area, and has more lakes than the rest of the world
- Toronto is the 4th largest city in North America

Canada ⇒ Toronto ⇒ Guelph (Fun facts)

- Canada 2nd largest country by area, and has more lakes than the rest of the world
- Toronto is the 4th largest city in North America
- **Home of NBA champion Toronto Raptors (basketball)**

Canada ⇒ Toronto ⇒ Guelph (Fun facts)

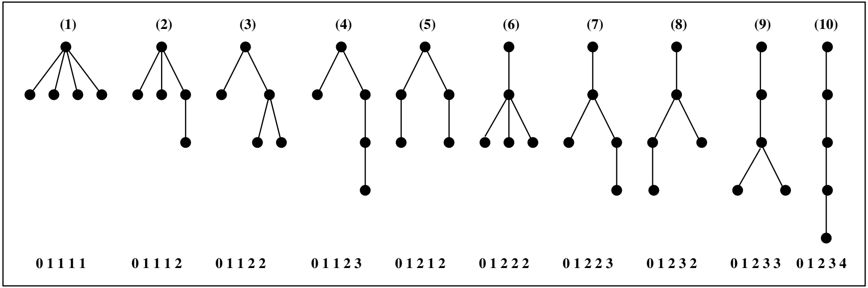
- Canada 2nd largest country by area, and has more lakes than the rest of the world
- Toronto is the 4th largest city in North America
- **Home of NBA champion Toronto Raptors (basketball)**
- **Championship parade held yesterday - expected up to 2 million people!**

Canada ⇒ Toronto ⇒ Guelph (Fun facts)

- Canada 2nd largest country by area, and has more lakes than the rest of the world
- Toronto is the 4th largest city in North America
- **Home of NBA champion Toronto Raptors (basketball)**
- **Championship parade held yesterday - expected up to 2 million people!**
- University of Guelph is 1 hour from Toronto, 30 minutes from Waterloo

Combinatorial Generation

Primary Goal: Given a combinatorial object (permutations, trees, necklaces, graphs), find an **efficient** algorithm to exhaustively list each instance exactly once



Important considerations

- Representation
- Ordering: lexicographic, Gray code

Related issues

- Enumeration
- Random generation
- Ranking, unranking

Such algorithms are often very short but hard to locate and usually are surprisingly subtle

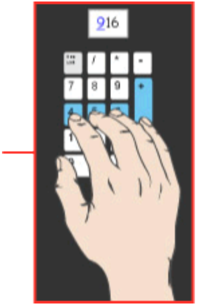
– Steven Skiena, The Stony Brook Algorithm Repository

The Combinatorial Object Server

Together with Torsten Mütze (UK) and Aaron Williams (USA), we recently began revitalizing Frank Ruskey's Combinatorial Object Server from the mid 1990s.

> The revitalized Combinatorial Object Server:
>
> http://combos.org

An ATM with no Enter Key



Consider an ATM that does **not** have an ENTER key. It accepts the last n digits as an attempted password.

To crack a 4 digit password, a brute force attack requires

- $4 \cdot 2^4 = 64$ key presses on a 2 digit keypad
- $4 \cdot 10^4 = 40,000$ key presses on a 10 digit keypad

Can we do better?

De Bruijn Sequences

A **de Bruijn (DB) sequence** is a circular string of length 2^n where every binary string of length n occurs as a substring (exactly once).

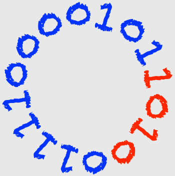
De Bruijn Sequences

A **de Bruijn (DB) sequence** is a circular string of length 2^n where every binary string of length n occurs as a substring (exactly once).

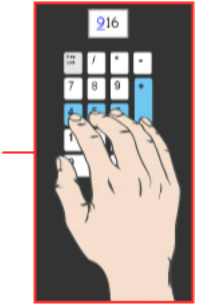
0000101101001111 is a DB sequence for $n = 4$

The 16 unique substrings of length 4 are:

$$0000,\ 0001,\ 0010,\ 0101,\ 1011,\ 0110,\ 1101,\ 1010,$$
$$0100,\ 1001,\ 0011,\ 0111,\ 1111,\ 1110,\ 1100,\ 1000.$$



Back to Cracking the ATM Password



How can we crack the password more efficiently?

Enter a DB sequence then repeat the first $n-1$ symbols to get the wraparound.

- The binary keypad requires $16+3=19$ key presses instead of 64
- The 10-digit keypad requires $10^4+3=10,003$ key presses instead of 40,000

PROBLEM: How to **efficiently** construct a DB sequence?

Outline of Seminar

1. **DB sequence Construction Methods**
   - Greedy approaches
   - Graph theoretic approach - de Bruijn graphs and Euler cycles
   - (Linear) Feedback shift registers
   - Successor-based approaches
   - Concatenation approaches

2. **Future Directions**

Greedy Approaches - Prefer 1

- 6th century BCE: *DB sequence* examples appear in early **Sanskrit prosody**
- 1894: **Rivière** questioned the existence of a *DB sequence* for arbitrary n. It was solved by **Fly Sainte-Marie** in the same year

Prefer-1 greedy algorithm (Martin 1934, Ford 1957)

1. Seed with 0^{n-1} (very important!)
2. **Repeat**: append the **largest** bit that does not create a duplicate length n substring
3. Remove the **seed**

Greedy Approaches - Prefer 1

- 6th century BCE: *DB sequence* examples appear in early **Sanskrit prosody**
- 1894: **Rivière** questioned the existence of a *DB sequence* for arbitrary n. It was solved by **Fly Sainte-Marie** in the same year

Prefer-1 greedy algorithm (Martin 1934, Ford 1957)

1. Seed with 0^{n-1} (very important!)
2. **Repeat**: append the **largest** bit that does not create a duplicate length n substring
3. Remove the **seed**

Example $n = 4$

000

Greedy Approaches - Prefer 1

- 6th century BCE: *DB sequence* examples appear in early **Sanskrit prosody**
- 1894: **Rivière** questioned the existence of a *DB sequence* for arbitrary n. It was solved by **Fly Sainte-Marie** in the same year

Prefer-1 greedy algorithm (Martin 1934, Ford 1957)

1. Seed with 0^{n-1} (very important!)
2. **Repeat**: append the **largest** bit that does not create a duplicate length n substring
3. Remove the **seed**

Example $n = 4$

000**1**

Greedy Approaches - Prefer 1

- 6th century BCE: *DB sequence* examples appear in early **Sanskrit prosody**
- 1894: **Rivière** questioned the existence of a *DB sequence* for arbitrary n. It was solved by **Fly Sainte-Marie** in the same year

Prefer-1 greedy algorithm (Martin 1934, Ford 1957)

1. Seed with 0^{n-1} (very important!)
2. **Repeat**: append the **largest** bit that does not create a duplicate length n substring
3. Remove the **seed**

Example $n = 4$

00011

Greedy Approaches - Prefer 1

- 6th century BCE: *DB sequence* examples appear in early **Sanskrit prosody**
- 1894: **Rivière** questioned the existence of a *DB sequence* for arbitrary n. It was solved by **Fly Sainte-Marie** in the same year

Prefer-1 greedy algorithm (Martin 1934, Ford 1957)

1. Seed with 0^{n-1} (very important!)
2. **Repeat**: append the **largest** bit that does not create a duplicate length n substring
3. Remove the **seed**

Example $n = 4$

000111**1**

Greedy Approaches - Prefer 1

- 6th century BCE: *DB sequence* examples appear in early **Sanskrit prosody**
- 1894: **Rivière** questioned the existence of a *DB sequence* for arbitrary n. It was solved by **Fly Sainte-Marie** in the same year

Prefer-1 greedy algorithm (Martin 1934, Ford 1957)

1. Seed with 0^{n-1} (very important!)
2. **Repeat**: append the **largest** bit that does not create a duplicate length n substring
3. Remove the **seed**

Example $n = 4$

0001111

Greedy Approaches - Prefer 1

- 6th century BCE: *DB sequence* examples appear in early **Sanskrit prosody**
- 1894: **Rivière** questioned the existence of a *DB sequence* for arbitrary n. It was solved by **Fly Sainte-Marie** in the same year

Prefer-1 greedy algorithm (Martin 1934, Ford 1957)

1. Seed with 0^{n-1} (very important!)
2. **Repeat**: append the **largest** bit that does not create a duplicate length n substring
3. Remove the **seed**

Example $n = 4$

$$000\underline{1111}$$

Greedy Approaches - Prefer 1

- 6th century BCE: *DB sequence* examples appear in early **Sanskrit prosody**
- 1894: **Rivière** questioned the existence of a *DB sequence* for arbitrary n. It was solved by **Fly Sainte-Marie** in the same year

Prefer-1 greedy algorithm (Martin 1934, Ford 1957)

1. Seed with 0^{n-1} (very important!)
2. **Repeat**: append the **largest** bit that does not create a duplicate length n substring
3. Remove the **seed**

Example $n = 4$

$$00011110$$

Greedy Approaches - Prefer 1

- 6th century BCE: *DB sequence* examples appear in early **Sanskrit prosody**
- 1894: **Rivière** questioned the existence of a *DB sequence* for arbitrary n. It was solved by **Fly Sainte-Marie** in the same year

Prefer-1 greedy algorithm (Martin 1934, Ford 1957)

1. Seed with 0^{n-1} (very important!)
2. **Repeat**: append the **largest** bit that does not create a duplicate length n substring
3. Remove the **seed**

Example $n = 4$

000111101**1**

Greedy Approaches - Prefer 1

- 6th century BCE: *DB sequence* examples appear in early **Sanskrit prosody**
- 1894: **Rivière** questioned the existence of a *DB sequence* for arbitrary n. It was solved by **Fly Sainte-Marie** in the same year

Prefer-1 greedy algorithm (Martin 1934, Ford 1957)

1. Seed with 0^{n-1} (very important!)
2. **Repeat**: append the **largest** bit that does not create a duplicate length n substring
3. Remove the **seed**

Example $n = 4$

000111101**1**

Greedy Approaches - Prefer 1

- 6th century BCE: *DB sequence* examples appear in early **Sanskrit prosody**
- 1894: **Rivière** questioned the existence of a *DB sequence* for arbitrary n. It was solved by **Fly Sainte-Marie** in the same year

Prefer-1 greedy algorithm (Martin 1934, Ford 1957)

1. Seed with 0^{n-1} (very important!)
2. **Repeat**: append the **largest** bit that does not create a duplicate length n substring
3. Remove the **seed**

Example $n = 4$

$$000111\underline{0111}$$

Greedy Approaches - Prefer 1

- 6th century BCE: *DB sequence* examples appear in early **Sanskrit prosody**
- 1894: **Rivière** questioned the existence of a *DB sequence* for arbitrary n. It was solved by **Fly Sainte-Marie** in the same year

Prefer-1 greedy algorithm (Martin 1934, Ford 1957)

1. Seed with 0^{n-1} (very important!)
2. **Repeat**: append the **largest** bit that does not create a duplicate length n substring
3. Remove the **seed**

Example $n = 4$

00011110110

Greedy Approaches - Prefer 1

- 6th century BCE: *DB sequence* examples appear in early **Sanskrit prosody**
- 1894: **Rivière** questioned the existence of a *DB sequence* for arbitrary n. It was solved by **Fly Sainte-Marie** in the same year

Prefer-1 greedy algorithm (Martin 1934, Ford 1957)

1. Seed with 0^{n-1} (very important!)
2. **Repeat**: append the **largest** bit that does not create a duplicate length n substring
3. Remove the **seed**

Example $n = 4$

~~000~~ 1111011001010000

Greedy Approaches - Prefer 1

- 6th century BCE: *DB sequence* examples appear in early **Sanskrit prosody**
- 1894: **Rivière** questioned the existence of a *DB sequence* for arbitrary n. It was solved by **Fly Sainte-Marie** in the same year

Prefer-1 greedy algorithm (Martin 1934, Ford 1957)

1. Seed with 0^{n-1} (very important!)
2. **Repeat**: append the **largest** bit that does not create a duplicate length n substring
3. Remove the **seed**

Example $n = 4$

$$1111011001010000$$

Proved to be the **lex largest** DB sequence (**Fredricksen 1970**)

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

$$101$$

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

1010

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

10100

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

101000**0**

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

1010000

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

$$101\underline{000\mathbf{0}}$$

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

10100001

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

101000011

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

101000011**1**

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

$$1010001111$$

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

$$101000011\underline{111\mathbf{1}}$$

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

101000011110

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

101000011110**0**

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

101000011110010110 1

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

~~101~~ 0000111100101101

Greedy Approaches - Prefer Same

- 1958: **Eldert et al.** gives an alternative greedy construction (no proof)
- 1982: **Fredricksen** re-states the algorithm and outlines proof
- 2018: **Alhakim, Sala, S.** simplified with new seed

Prefer-same greedy algorithm

1. Seed with length $n-1$ string $\cdots 10101$ (very important!)
2. Append 0
3. **Repeat**: append the **same** bit as the last if it does not create a duplicate length n substring; otherwise try the opposite
4. Remove the **seed**

Example $n = 4$

$$\cancel{101}\ 0000111100101101$$

Note: Run length = 44211211 is the lex largest amongst all DB sequences

Greedy Approaches - Prefer Opposite

Prefer-opposite greedy algorithm (Alhakim, 2010)

1. Seed with length n string 0^n (very important!)
2. **Repeat**: append the **opposite** bit as the last if it does not create a duplicate length n substring; otherwise try the same **until** reaching 01^{n-1}
3. Append 1

Greedy Approaches - Prefer Opposite

Prefer-opposite greedy algorithm (Alhakim, 2010)

1. Seed with length n string 0^n (very important!)
2. **Repeat**: append the **opposite** bit as the last if it does not create a duplicate length n substring; otherwise try the same **until** reaching 01^{n-1}
3. Append 1

Example $n = 4$

0000

Greedy Approaches - Prefer Opposite

Prefer-opposite greedy algorithm (Alhakim, 2010)

1. Seed with length n string 0^n (very important!)
2. **Repeat**: append the **opposite** bit as the last if it does not create a duplicate length n substring; otherwise try the same **until** reaching 01^{n-1}
3. Append 1

Example $n = 4$

0000**1**

Greedy Approaches - Prefer Opposite

Prefer-opposite greedy algorithm (Alhakim, 2010)

1. Seed with length n string 0^n (very important!)
2. **Repeat**: append the **opposite** bit as the last if it does not create a duplicate length n substring; otherwise try the same **until** reaching 01^{n-1}
3. Append 1

Example $n = 4$

000010

Greedy Approaches - Prefer Opposite

Prefer-opposite greedy algorithm (Alhakim, 2010)

1. Seed with length n string 0^n (very important!)
2. **Repeat**: append the **opposite** bit as the last if it does not create a duplicate length n substring; otherwise try the same **until** reaching 01^{n-1}
3. Append 1

Example $n = 4$

0000101

Greedy Approaches - Prefer Opposite

Prefer-opposite greedy algorithm (Alhakim, 2010)

1. Seed with length n string 0^n (very important!)
2. **Repeat**: append the **opposite** bit as the last if it does not create a duplicate length n substring; otherwise try the same **until** reaching 01^{n-1}
3. Append 1

Example $n = 4$

00001010

Greedy Approaches - Prefer Opposite

Prefer-opposite greedy algorithm (Alhakim, 2010)

1. Seed with length n string 0^n (very important!)
2. **Repeat**: append the **opposite** bit as the last if it does not create a duplicate length n substring; otherwise try the same **until** reaching 01^{n-1}
3. Append 1

Example $n = 4$

$$0000\underline{1010\mathbf{1}}$$

Greedy Approaches - Prefer Opposite

Prefer-opposite greedy algorithm (Alhakim, 2010)

1. Seed with length n string 0^n (very important!)
2. **Repeat**: append the **opposite** bit as the last if it does not create a duplicate length n substring; otherwise try the same **until** reaching 01^{n-1}
3. Append 1

Example $n = 4$

0000101000

Greedy Approaches - Prefer Opposite

Prefer-opposite greedy algorithm (Alhakim, 2010)

1. Seed with length n string 0^n (very important!)
2. **Repeat**: append the **opposite** bit as the last if it does not create a duplicate length n substring; otherwise try the same **until** reaching 01^{n-1}
3. Append 1

Example $n = 4$

000010100**1**

Greedy Approaches - Prefer Opposite

Prefer-opposite greedy algorithm (Alhakim, 2010)

1. Seed with length n string 0^n (very important!)
2. **Repeat**: append the **opposite** bit as the last if it does not create a duplicate length n substring; otherwise try the same **until** reaching 01^{n-1}
3. Append 1

Example $n = 4$

0000101001101**111**

Greedy Approaches - Prefer Opposite

Prefer-opposite greedy algorithm (Alhakim, 2010)

1. Seed with length n string 0^n (very important!)
2. **Repeat**: append the **opposite** bit as the last if it does not create a duplicate length n substring; otherwise try the same **until** reaching 01^{n-1}
3. Append 1

Example $n = 4$

0000101001101111**1**

Implementing Greedy Approaches

For each greedy approach we need to either:

- Store the current sequence and search to see if a substring already exists - or -
- Keep track of which substrings have already been visited

Implementing Greedy Approaches

For each greedy approach we need to either:

- Store the current sequence and search to see if a substring already exists - or -
- Keep track of which substrings have already been visited

✗ Either approach requires exponential $O(2^n)$ space

✔ The latter approach can generate each new bit in $O(n)$-time

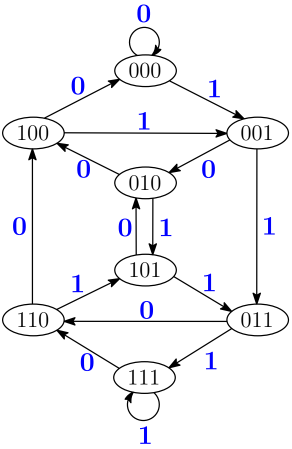
Graph Theoretic Approach

- 1944: **Posthumus** conjectures the number of DB sequences
- 1946: **de Bruijn** proves the conjecture using ta graph model
- 1946: **Good** independently uses graphs to prove existence

A **de Bruijn graph** of order n is a directed graph

- the vertices are length $n-1$ binary strings
- each directed edge labeled y goes from $xb_1b_2\cdots b_{n-2}$ to $b_1b_2\cdots b_{n-2}y$

Each length n binary string corresponds to an edge, and is recovered by considering a vertex together with an outgoing edge label.

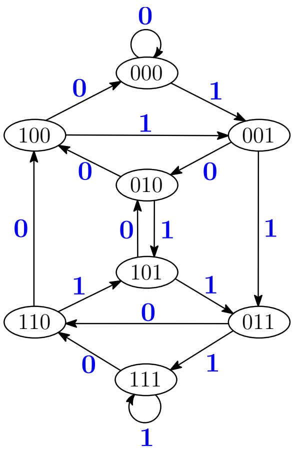


Euler Cycles in the de Bruijn Graph

An **Euler cycle** visits every edge exactly once.

A directed graph $G = (V, E)$ has an Euler cycle if and only if:

- for every vertex the in-degree equals the out-degree
- the graph is strongly connected (there is a path between every pair of vertices)

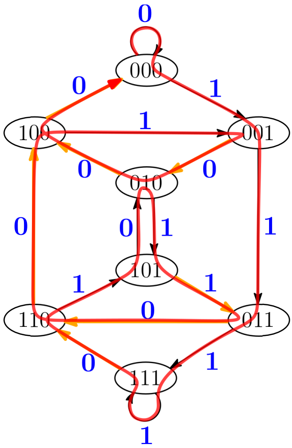


Euler Cycles in the de Bruijn Graph

An **Euler cycle** visits every edge exactly once.

A directed graph $G = (V, E)$ has an Euler cycle if and only if:

- for every vertex the in-degree equals the out-degree
- the graph is strongly connected (there is a path between every pair of vertices)



A **DB sequence** is in 1-1 correspondence with an **Euler cycle** in the de Bruijn graph. It is obtained by outputting the edge labels from tracing the cycle.

$$0111101011001000$$

How to Find Euler Cycles

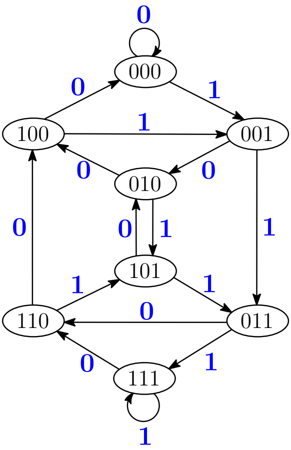
Method 1: Hierholzer algorithm

(1) Start at random vertex v and traverse edges until returning to v, thus creating a cycle

(2) Start from a vertex already on the cycle to find a new disjoint cycle and then merge the 2 cycles together

(3) Repeat (2) until no edges are left

A **cycle joining approach** that we will see again.



How to Find Euler Cycles

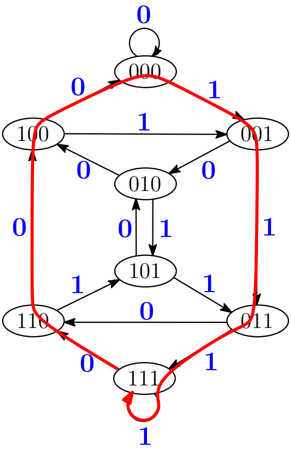
Method 1: Hierholzer algorithm

(1) Start at random vertex v and traverse edges until returning to v, thus creating a cycle

(2) Start from a vertex already on the cycle to find a new disjoint cycle and then merge the 2 cycles together

(3) Repeat (2) until no edges are left

A **cycle joining approach** that we will see again.



How to Find Euler Cycles

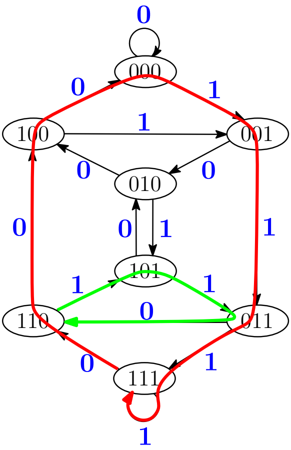
Method 1: Hierholzer algorithm

(1) Start at random vertex v and traverse edges until returning to v, thus creating a cycle

(2) Start from a vertex already on the cycle to find a new disjoint cycle and then merge the 2 cycles together

(3) Repeat (2) until no edges are left

A **cycle joining approach** that we will see again.



How to Find Euler Cycles

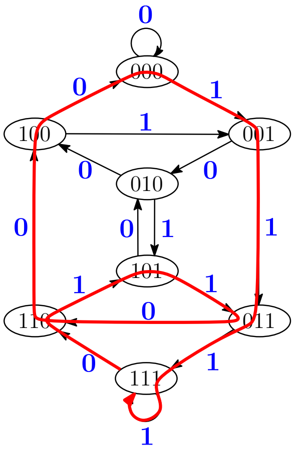
Method 1: Hierholzer algorithm

(1) Start at random vertex v and traverse edges until returning to v, thus creating a cycle

(2) Start from a vertex already on the cycle to find a new disjoint cycle and then merge the 2 cycles together

(3) Repeat (2) until no edges are left

A **cycle joining approach** that we will see again.



How to Find Euler Cycles

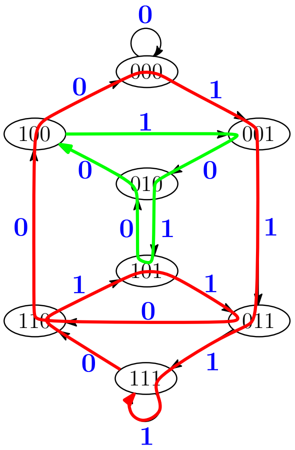
Method 1: Hierholzer algorithm

(1) Start at random vertex v and traverse edges until returning to v, thus creating a cycle

(2) Start from a vertex already on the cycle to find a new disjoint cycle and then merge the 2 cycles together

(3) Repeat (2) until no edges are left

A **cycle joining approach** that we will see again.



How to Find Euler Cycles

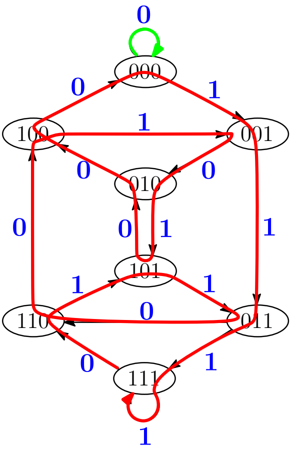
Method 1: Hierholzer algorithm

(1) Start at random vertex v and traverse edges until returning to v, thus creating a cycle

(2) Start from a vertex already on the cycle to find a new disjoint cycle and then merge the 2 cycles together

(3) Repeat (2) until no edges are left

A **cycle joining approach** that we will see again.



How to Find Euler Cycles

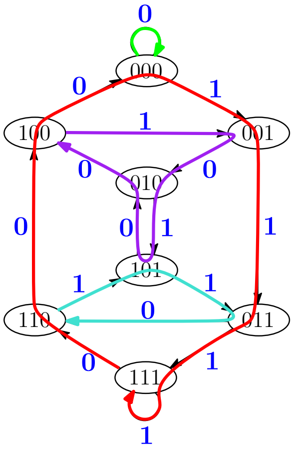
Method 1: Hierholzer algorithm

(1) Start at random vertex v and traverse edges until returning to v, thus creating a cycle

(2) Start from a vertex already on the cycle to find a new disjoint cycle and then merge the 2 cycles together

(3) Repeat (2) until no edges are left

A **cycle joining approach** that we will see again.



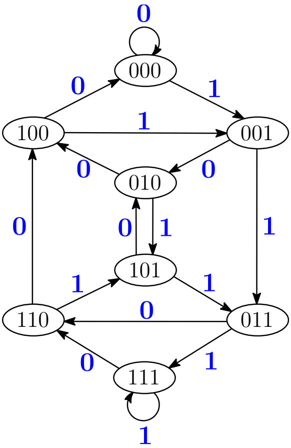
How to Find Euler Cycles

Method 2: Fleury's Algo (don't burn bridges)

(1) Pick a root vertex and compute a spanning in-tree

(2) Make edges of spanning tree (the bridges) the last edge on the adjacency list of each vertex

(3) Traverse edges starting from root



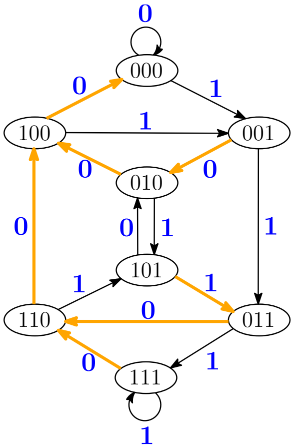
How to Find Euler Cycles

Method 2: Fleury's Algo (don't burn bridges)

(1) Pick a root vertex and compute a spanning in-tree

(2) Make edges of spanning tree (the bridges) the last edge on the adjacency list of each vertex

(3) Traverse edges starting from root



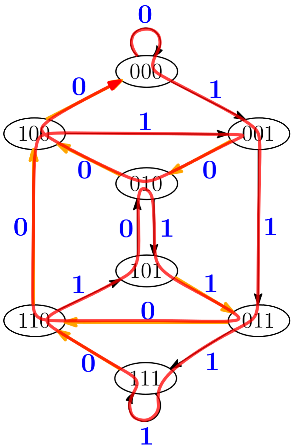
How to Find Euler Cycles

Method 2: Fleury's Algo (don't burn bridges)

(1) Pick a root vertex and compute a spanning in-tree

(2) Make edges of spanning tree (the bridges) the last edge on the adjacency list of each vertex

(3) Traverse edges starting from root



$$\# \text{ of spanning in-trees } = \# \text{ of DB sequences } = \frac{2^{2^{n-1}}}{2^n}$$

Implementing Euler Cycle Algorithms

Each Euler cycle algorithm requires that the graph be stored in memory

✗ Either algorithm requires exponential $O(2^n)$ space

✗ Difficult to analyze the properties of any specific DB sequence

✔ Euler cycle approaches can generate **all** $\frac{2^{2^{n-1}}}{2^n}$ DB sequences

Open question: Is there a direct counting argument for the number of spanning trees and DB sequences? The initial enumeration proof uses matrix analysis techniques.

Application: Pseudorandom Bit Generation

DB sequences have the following nice properties:

- **Balanced**: they contain the same number of 0s as 1s
- **Run property**: there are an equal number of runs of 0s and 1s of same length
- **Span property**: they contain every distinct length n binary string as a substring

Application: Pseudorandom Bit Generation

DB sequences have the following nice properties:

- **Balanced**: they contain the same number of 0s as 1s
- **Run property**: there are an equal number of runs of 0s and 1s of same length
- **Span property**: they contain every distinct length n binary string as a substring

The **discrepancy** of a sequence is the maximum difference in the number of 0s as 1s over all substrings.

$$0111\mathbf{0001010100}11 \quad \text{has discrepancy } |7-3| = 4$$

Some DB sequences have discrepancy $\leq 2n$ or up to $\Theta(2^n \log n/n)$. What is best for a pseudo-random number generator?

Feedback Shift Registers

The construction methods up to this point have **significant limitations**

- 1967: **Golomb** publishes Shift Register Sequences focusing on a feedback shift register construction
- 1982: **Fredricksen** publishes A survey of full length nonlinear shift register cycle algorithms
- 2012-2016: **Dubrova** and **Li et al.** present non-linear feedback shift register constructions

Feedback Shift Registers

The construction methods up to this point have **significant limitations**

- 1967: **Golomb** publishes Shift Register Sequences focusing on a feedback shift register construction
- 1982: **Fredricksen** publishes A survey of full length nonlinear shift register cycle algorithms
- 2012-2016: **Dubrova** and **Li et al.** present non-linear feedback shift register constructions

A **feedback shift register** (FSR) is a shift register whose input bit is a function of its previous state (string of length n)

$$f(b_1 b_2 \cdots b_n) = b_2 b_3 \cdots b_n \ g(b_1 \cdots b_n)$$

where $g(b_1 \cdots b_n)$ is the feedback function.

Each **primitive polynomial** for a given n corresponds to a unique **linear** FSR that outputs a DB sequence less the 0^n string - called m**-sequences**

Feedback Shift Registers

A **feedback function** for a LFSR based on a primitive polynomial of degree $n = 12$

$$g(b_1 b_2 \cdots b_{12}) \;=\; b_1 + b_2 + b_3 + b_9 \pmod 2$$

n	3	4	5	6	7
primitive polynomials	2	2	6	6	18
DB sequences	2	16	2048	67108864	144115188075855872

Feedback Shift Registers

A **feedback function** for a LFSR based on a primitive polynomial of degree $n = 12$

$$g(b_1 b_2 \cdots b_{12}) \;=\; b_1 + b_2 + b_3 + b_9 \pmod 2$$

n	3	4	5	6	7
primitive polynomials	2	2	6	6	18
DB sequences	2	16	2048	67108864	144115188075855872

✗ Need a different primitive polynomial for each n

✔ Efficient implementation - $O(n)$ per bit using $O(n)$ space

✔ Suitable for implementing in hardware

✗ Not ideal for psuedorandom bit generation – the LFSR can be determined after $2n$ bits using the **Berlekamp-Massey** algorithm

Successor-based constructions

- 1972: **Fredricksen** gives an efficient **successor rule** for the prefer-1 (which is the complement of the prefer-0) greedy construction

Quote by Fredericksen (1982)

When the mathematician on the street is presented with the problem of generating a full cycle [DB sequence], one of the three things happens: he gives up, or produces a sequence based on a primitive polynomial, or produces the prefer-one sequence. Only rarely is a new algorithm proposed.

Successor-based constructions

- 1972: **Fredricksen** gives an efficient **successor rule** for the prefer-1 (which is the complement of the prefer-0) greedy construction

> **Quote by Fredericksen (1982)**
>
> *When the mathematician on the street is presented with the problem of generating a full cycle [DB sequence], one of the three things happens: he gives up, or produces a sequence based on a primitive polynomial, or produces the prefer-one sequence. Only rarely is a new algorithm proposed.*

- 1984, 1987: **Etzion and Lempel** present **successor rules** based on simple FSRs
- 1990: **Huang** presents a new **successor rule** construction using the CCR
- 1991: **Jansen**, **Franx** and **Boekee** present a generic FSR-based **successor rule** approach
- 2013, 2017: **Dragon**, **Hernandez**, **S.**, **Williams**, **Wong**, present a simple successors for k-ary alphabet

Successor Rules

A **DB successor** for a given DB sequence is a (feedback) function

$$NEXT(b_1 b_2 \cdots b_n) = \begin{cases} 0 & \text{if } \textbf{conditions} \\ 1 & \text{otherwise} \end{cases}$$

that returns the bit following the substring $b_1 b_2 \cdots b_n$

Successor Rules

A **DB successor** for a given DB sequence is a (feedback) function

$$NEXT(b_1 b_2 \cdots b_n) = \begin{cases} 0 & \text{if } \textbf{conditions} \\ 1 & \text{otherwise} \end{cases}$$

that returns the bit following the substring $b_1 b_2 \cdots b_n$

Feedback functions based on primitive polynomials are "almost" **DB successors**

Successor Rules

A **DB successor** for a given DB sequence is a (feedback) function

$$NEXT(b_1 b_2 \cdots b_n) = \begin{cases} 0 & \text{if } \textbf{conditions} \\ 1 & \text{otherwise} \end{cases}$$

that returns the bit following the substring $b_1 b_2 \cdots b_n$

Feedback functions based on primitive polynomials are “almost” **DB successors**

Most published successors are based on **simple feedback functions**

- $PCR(b_1 b_2 \cdots b_n) = b_1$ induces **necklace** equivalence classes
- $CCR(b_1 b_2 \cdots b_n) = \bar{b}_1$
- $PSR(b_1 b_2 \cdots b_n) = b_1 + b_2 + \cdots + b_n \pmod 2$
- $CSR(b_1 b_2 \cdots b_n) = 1 + b_1 + b_2 + \cdots + b_n \pmod 2$

Necklaces

A **necklace** is the lexicographically least representative in an equivalence class of strings under rotation.

000000	000001	000011	000101	000111	001001	001011
	000010	000110	001010	001110	010010	010110
	000100	001100	010100	011100	100100	101100
	001000	011000	101000	111000		011001
	010000	110000	010001	110001		110010
	100000	100001	100010	100011		100101

001101	001111	010101	010111	011011	011111	111111
011010	011110	101010	101110	110110	111110	
110100	111100		011101	101101	111101	
101001	111001		111010		111011	
010011	110011		110101		110111	
100110	100111		101011		101111	

Testing whether or not a string is a necklace can be done in $O(n)$ time (**Booth, 1980**)

Simple DB Successors

Successor for the prefer-0 greedy construction (Fredricksen, 1972)

Let j be the smallest index of $b_1b_2\cdots b_n$ such that $b_j = 0$ and $j > 1$, or $j = n+1$ if no such index exists. Let $\gamma = b_jb_{j+1}\cdots b_n01^{j-2}$.

$$NEXT(b_1b_2\cdots b_n) = \begin{cases} \overline{b}_1 & \text{if } \gamma \text{ is a necklace} \\ b_1 & \text{otherwise} \end{cases}$$

Simple DB Successors

Successor for the prefer-0 greedy construction (Fredricksen, 1972)

Let j be the smallest index of $b_1 b_2 \cdots b_n$ such that $b_j = 0$ and $j > 1$, or $j = n+1$ if no such index exists. Let $\gamma = b_j b_{j+1} \cdots b_n 0 1^{j-2}$.

$$NEXT(b_1 b_2 \cdots b_n) = \begin{cases} \overline{b}_1 & \text{if } \gamma \text{ is a necklace} \\ b_1 & \text{otherwise} \end{cases}$$

(Jansen, Franx, Boekee, 1991) and later by (**Wong 2013**)

$$NEXT(b_1 b_2 \cdots b_n) = \begin{cases} \overline{b}_1 & \text{if } b_2 b_3 \cdots b_n \mathbf{1} \text{ is a necklace} \\ b_1 & \text{otherwise} \end{cases}$$

Simple DB Successors

Successor for the prefer-0 greedy construction (Fredricksen, 1972)

Let j be the smallest index of $b_1 b_2 \cdots b_n$ such that $b_j = 0$ and $j > 1$, or $j = n+1$ if no such index exists. Let $\gamma = b_j b_{j+1} \cdots b_n 0 1^{j-2}$.

$$NEXT(b_1 b_2 \cdots b_n) = \begin{cases} \overline{b}_1 & \text{if } \gamma \text{ is a necklace} \\ b_1 & \text{otherwise} \end{cases}$$

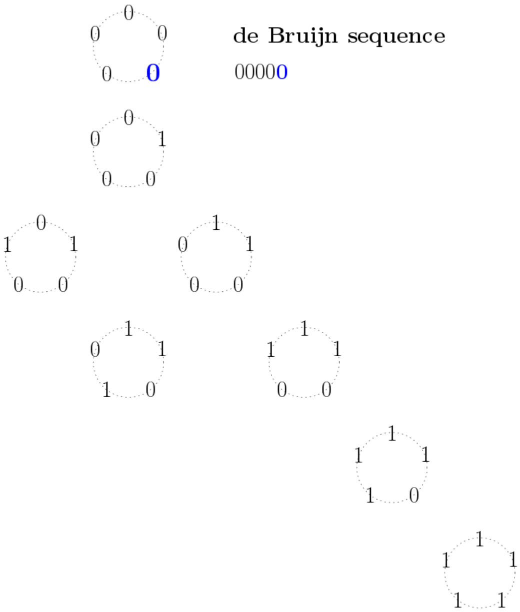
(Jansen, Franx, Boekee, 1991) and later by (**Wong 2013**)

$$NEXT(b_1 b_2 \cdots b_n) = \begin{cases} \overline{b}_1 & \text{if } b_2 b_3 \cdots b_n \mathbf{1} \text{ is a necklace} \\ b_1 & \text{otherwise} \end{cases}$$

(Gabric, S., Williams, Wong, 2018)

$$NEXT(b_1 b_2 \cdots b_n) = \begin{cases} \overline{b}_1 & \text{if } \mathbf{0} b_2 b_3 \cdots b_n \text{ is a necklace} \\ b_1 & \text{otherwise} \end{cases}$$

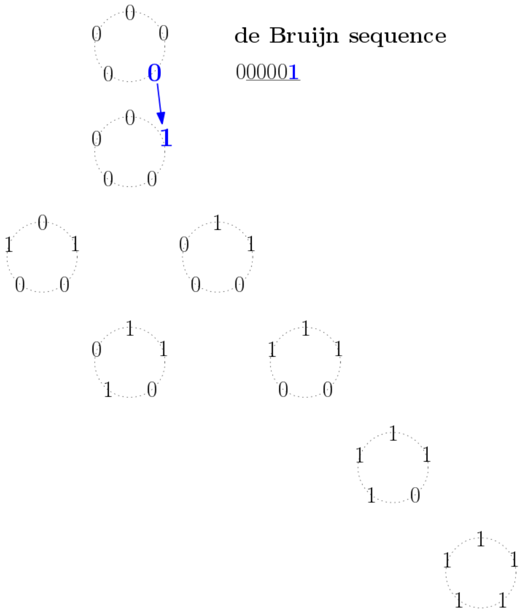
JFB and Wong successor rule (necklace spanning tree $n = 5$)



de Bruijn sequence

00000

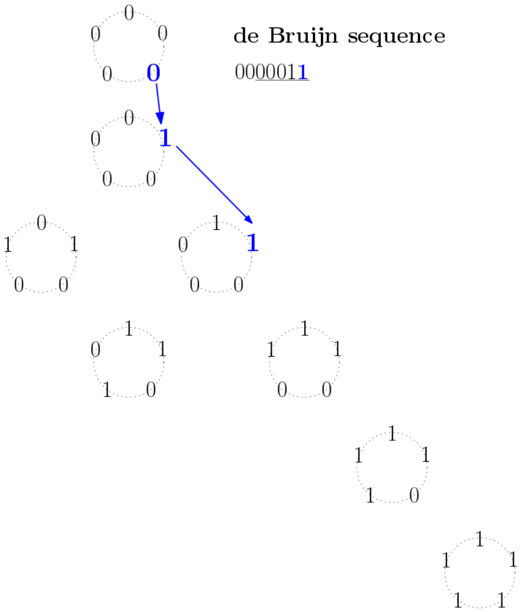
JFB and Wong successor rule (necklace spanning tree $n = 5$)



de Bruijn sequence

$0\underline{00001}$

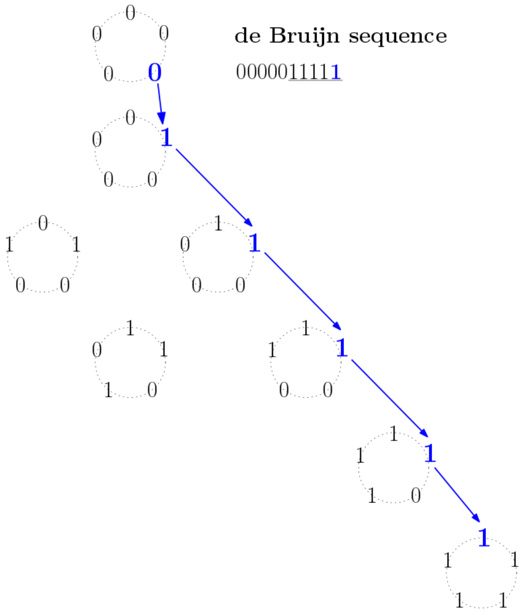
JFB and Wong successor rule (necklace spanning tree $n = 5$)



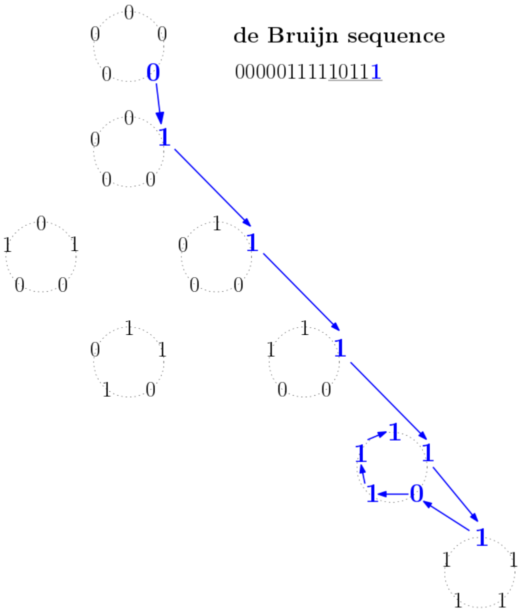
de Bruijn sequence

0000011

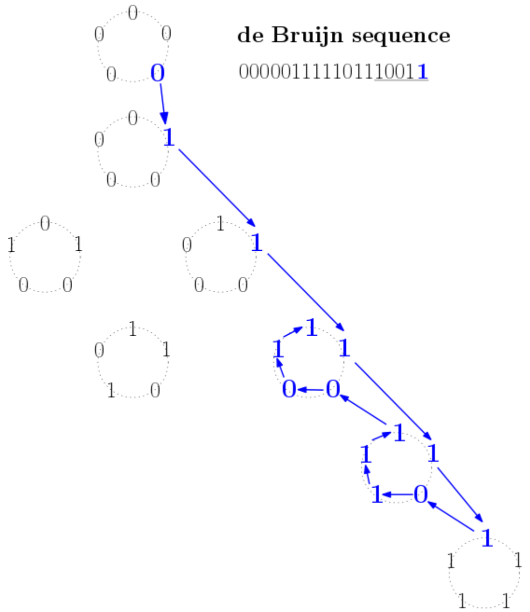
JFB and Wong successor rule (necklace spanning tree $n = 5$)



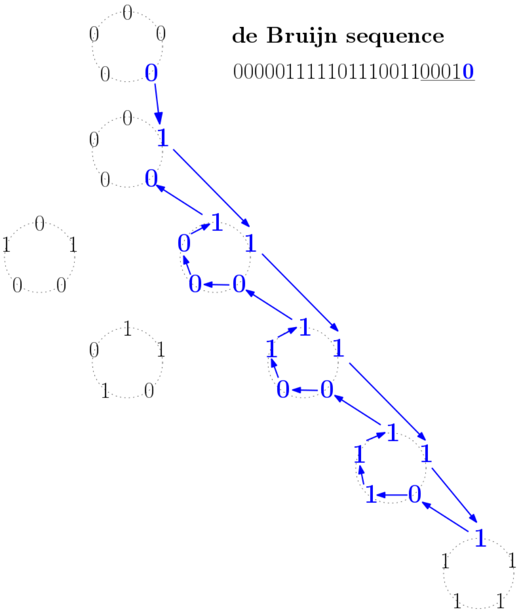
JFB and Wong successor rule (necklace spanning tree $n = 5$)



JFB and Wong successor rule (necklace spanning tree $n = 5$)



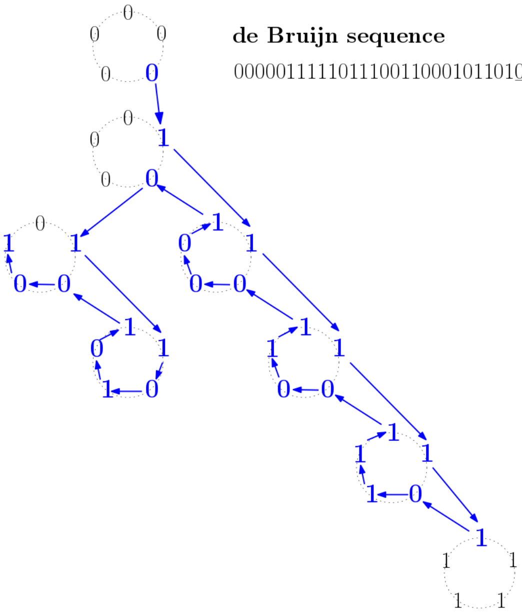
JFB and Wong successor rule (necklace spanning tree $n = 5$)



de Bruijn sequence

0000011111011100110001**0**

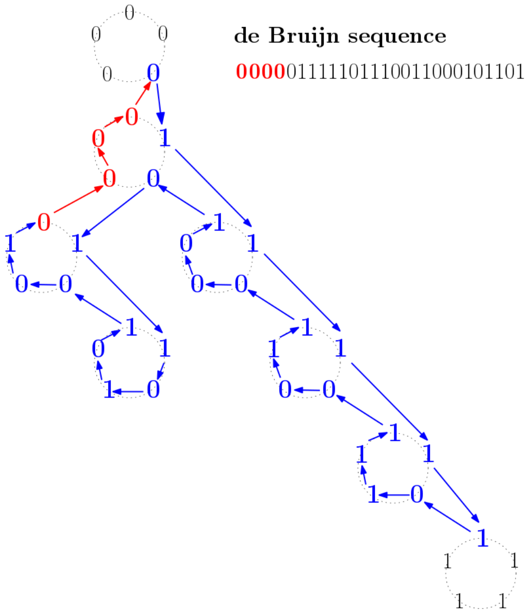
JFB and Wong successor rule (necklace spanning tree $n = 5$)



de Bruijn sequence

0000011111011100110001011010100**1**

JFB and Wong successor rule (necklace spanning tree $n = 5$)



de Bruijn sequence

0000011111011100110001011010100 1

Very Simple DB Successors

One based on the CCR (**Gabric, S., Williams, Wong, 2018**)

$$NEXT(b_1b_2\cdots b_n) = \begin{cases} b_1 & \text{if } b_2b_3\cdots b_n\mathbf{0} \neq 0^n \text{ is a co-necklace} \\ \overline{b}_1 & \text{otherwise} \end{cases}$$

Note: α is a co-necklace if $\alpha\overline{\alpha}$ is a necklace.

Very Simple DB Successors

One based on the CCR (**Gabric, S., Williams, Wong, 2018**)

$$NEXT(b_1 b_2 \cdots b_n) = \begin{cases} b_1 & \text{if } b_2 b_3 \cdots b_n \mathbf{0} \neq 0^n \textbf{ is a co-necklace} \\ \overline{b}_1 & \text{otherwise} \end{cases}$$

Note: α is a co-necklace if $\alpha\overline{\alpha}$ is a necklace.

✔ Each bit can be generated in $O(n)$-time using $O(n)$-space **for any** n

Can we do better?

Concatenation Approaches

1977-1978: **Fredricksen**, **Kessler**, and **Maiorana** (FKM) present the first concatenation construction

FKM Algorithm

1. List the necklaces of length n in lexicographic order
2. Concatenate together their periodic reductions

Concatenation Approaches

1977-1978: **Fredricksen**, **Kessler**, and **Maiorana** (FKM) present the first concatenation construction

FKM Algorithm

1. List the necklaces of length n in lexicographic order
2. Concatenate together their periodic reductions

Example for $n = 6$

$$000000, 000001, 000011, 000101, 000111, 001001, 001011,$$
$$001101, 001111, 010101, 010111, 011011, 011111, 111111$$

Concatenation Approaches

1977-1978: **Fredricksen**, **Kessler**, and **Maiorana** (FKM) present the first concatenation construction

FKM Algorithm

1. List the necklaces of length n in lexicographic order
2. Concatenate together their periodic reductions

Example for $n = 6$

$$0\mathbf{00000}, 000001, 000011, 000101, 000111, 001\mathbf{001}, 001011,$$
$$001101, 001111, 01\mathbf{0101}, 010111, 011\mathbf{011}, 011111, 1\mathbf{11111}$$

Concatenation Approaches

1977-1978: **Fredricksen**, **Kessler**, and **Maiorana** (FKM) present the first concatenation construction

FKM Algorithm

1. List the necklaces of length n in lexicographic order
2. Concatenate together their periodic reductions

Example for $n = 6$

$$0\mathbf{00000}, 000001, 000011, 000101, 000111, 001\mathbf{001}, 001011,$$
$$001101, 001111, 01\mathbf{0101}, 010111, 011\mathbf{011}, 011111, 1\mathbf{11111}$$

DB seq: 0000001000011000101000111001001011001101001111010101110110111111

Concatenation Approaches

Amazingly, the FKM algorithm produces the lexicographically smallest DB sequence and it is equivalent to the prefer-0 greedy construction

✔✔ Each bit can be generated in $O(1)$-**amortized time** using $O(n)$-space for any n (**Ruskey**, **Savage, Wang, 1992**)

Concatenation Approaches

Amazingly, the FKM algorithm produces the lexicographically smallest DB sequence and it is equivalent to the prefer-0 greedy construction

✔✔ Each bit can be generated in $O(1)$-**amortized time** using $O(n)$-space for any n (**Ruskey**, **Savage, Wang, 1992**)

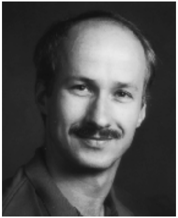
Equivalent construction

1. List the **Lyndon words** of length that divide n in lexicographic order
2. Concatenate them together

While equivalent for lexicographic ordering, the two constructions are not the same when we consider other necklace orderings! **Try it with colex ordering.**

Concatenation Approaches

- 2012: **Ruskey**, **S.**, and **Williams** apply cool-lex ordering on necklaces
- 2013, 2017: **Dragon**, **Hernandez**, **S.**, **Williams**, **Wong** use colex order on necklaces
- 2017: **Gabric** and **S.** find a co-necklace concatenation construction



Summary of Concatenation Approaches

1. Greedy approaches - **exponential space**

2. Graph theoretic, Euler cycles - **exponential space**

3. Linear feedback shift registers - **require primitive polynomial for each** n

4. Successor rules - **very simple and efficient**

5. Concatenation approaches **very efficient, but only several known approaches**

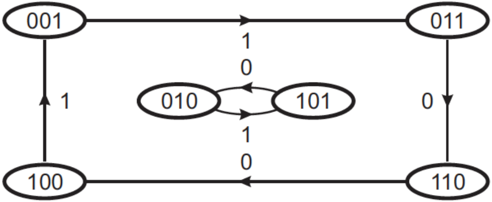
Generalization: Universal Cycles

Universal cycles are **generalizations** of DB sequences to other sets **S** of objects.

A **universal cycle** for a set $\mathbf{S}$ is a circular sequence of length $|S|$ such that each element $s \in \mathbf{S}$ is represented as a length n substring exactly once.

The notion of **de Bruijn graphs** can be generalized as well. Such a set $\mathbf{S}$ will have a universal cycle if and only if its corresponding de Bruijn graph has an Euler cycle.

- The set of permutations of order n in one line notation does not have a universal cycle. However, they do exist for a shorthand representation.
- Combinations $C(n, k)$ do not have a universal cycle. The following de Bruijn graph for $C(4, 2)$ is not connected.



debruijnsequence.org

A new resource with information on these constructions and more:

http://debruijnsequence.org

Contributors: JS, Aaron Williams, Dennis Wong, Daniel Gabric, Torsten Mütze

Future Directions

1. Find efficient successor rules for the prefer-same and prefer-opposite greedy approaches (recently solved!)

2. Find constructions for a 2-dimensional de Bruijn torus (existence questions)

3. Find universal cycles for other combinatorial objects (unlabeled necklaces)

4. Efficiently construct the lexicographically smallest shorthand permutation UC

5. Construct a DB sequence with discrepancy close to what is expected from a random sequence

Future Directions

1. Find efficient successor rules for the prefer-same and prefer-opposite greedy approaches (recently solved!)
2. Find constructions for a 2-dimensional de Bruijn torus (existence questions)
3. Find universal cycles for other combinatorial objects (unlabeled necklaces)
4. Efficiently construct the lexicographically smallest shorthand permutation UC
5. Construct a DB sequence with discrepancy close to what is expected from a random sequence

– Grazie –