# [Knu11, Chapter 7.2.1.1]
# Binary

Anakin

# Outline

Generating Tuples

The Gray Code

Towers of Hanoi and A Chinese Ring Puzzle

# What Are We Doing?

What is Combinatorics?

# What Are We Doing?

What is Combinatorics?

- Existence
- Construction
- Enumeration
- Generation
- Optimization

# What Are We Doing?

What is Combinatorics?

- ~~Existence~~
- ~~Construction~~
- ~~Enumeration~~
- Generation (Our focus for today!)
- ~~Optimization~~

$\Sigma$

# Section 1

## Generating Tuples

# A Classic Problem

- Suppose we wanted to generate through all binary numbers from $\texttt{00000000} = 0$ through to $\underbrace{\texttt{11111111}}_{\textbf{8}\ \text{1s}} = 2^8 - 1$
  - Or more generally, 0 through $2^n$ - 1

# A Classic Problem

- Suppose we wanted to generate through all binary numbers from $\texttt{00000000} = 0$ through to $\underbrace{\texttt{11111111}}_{\textbf{8}\ \text{1s}} = 2^8 - 1$
  - Or more generally, 0 through $2^n$ - 1
- Equivalent to generate tuples $(a_n, \ldots, a_1)$ with $a_i \in \{\, \texttt{0}, \texttt{1} \,\}$
  - We write the tuple in this direction since we write numbers with bigger “places” to the left of smaller places

# A Classic Problem

- Suppose we wanted to generate through all binary numbers from $\texttt{00000000} = 0$ through to $\underbrace{\texttt{11111111}}_{\mathbf{8}\ \text{1s}} = 2^8 - 1$
  - Or more generally, 0 through $2^n$ - 1
- Equivalent to generate tuples $(a_n, \ldots, a_1)$ with $a_i \in \{\texttt{0}, \texttt{1}\}$
  - We write the tuple in this direction since we write numbers with bigger "places" to the left of smaller places
- We could even talk about other bases, like wanted to visit all base 10 numbers from 0 through $10^n - 1$

# The Obvious Algorithm (for $n = 8$)

# The Obvious Algorithm (for $n = 8$)

```
GenBinary():
  For a_1 ∈ { 0, 1 } :
    For a_2 ∈ { 0, 1 } :
      For a_3 ∈ { 0, 1 } :
        For a_4 ∈ { 0, 1 } :
          For a_5 ∈ { 0, 1 } :
            For a_6 ∈ { 0, 1 } :
              For a_7 ∈ { 0, 1 } :
                For a_8 ∈ { 0, 1 } :
                  PRINT(a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8)
```

# We Can Do Better

- What if we wanted to change our base from binary to base 10, or arbitrary base?
  - Mixed base, also known as **mixed radix** [Knu97], numbers:

$$\begin{bmatrix} a_n, & a_{n-1}, & \ldots, & a_1 \\ m_n, & m_{n-1}, & \ldots, & m_1 \end{bmatrix}$$

# We Can Do Better

- What if we wanted to change our base from binary to base 10, or arbitrary base?
  - Mixed base, also known as **mixed radix** [Knu97], numbers:

$$\begin{bmatrix} a_n, & a_{n-1}, & \dots, & a_1 \\ m_n, & m_{n-1}, & \dots, & m_1 \end{bmatrix}$$

  - Examples for base 2 and time (0 index your days and months):

$$\texttt{101001}_2 = \begin{bmatrix} 1, & 0, & 1, & 0, & 0, & 1 \\ 2, & 2, & 2, & 2, & 2, & 2 \end{bmatrix},$$

$$\text{2002-06-29 03:25:789} = \begin{bmatrix} 2002, & 06, & 29, & 03, & 25, & 789 \\ \infty, & 12, & 30, & 24, & 60, & 1000 \end{bmatrix}$$

# We Can Do Better

- What if we wanted to change our base from binary to base 10, or arbitrary base?
  - Mixed base, also known as **mixed radix** [Knu97], numbers:

$$\begin{bmatrix} a_n, & a_{n-1}, & \dots, & a_1 \\ m_n, & m_{n-1}, & \dots, & m_1 \end{bmatrix}$$

  - Examples for base 2 and time (0 index your days and months):

$$\texttt{101001}_2 = \begin{bmatrix} 1, & 0, & 1, & 0, & 0, & 1 \\ 2, & 2, & 2, & 2, & 2, & 2 \end{bmatrix},$$

$$\text{2002-06-29 03:25:789} = \begin{bmatrix} 2002, & 06, & 29, & 03, & 25, & 789 \\ \infty, & 12, & 30, & 24, & 60, & 1000 \end{bmatrix}$$

# Mixed-Radix Generation

$\underline{\textsc{Algorithm-M}(m[1..n])\text{:}}$

- $a[i] \leftarrow 0$ **for** $1 \le i \le n$
- $a[n+1] \leftarrow 0,\ m[n+1] \leftarrow 2$ ⟪***Exercise:*** *why do we need this?*⟫
- **while** TRUE:
  - $\textsc{print}(a[n] \cdots a[1])$
  - $j \leftarrow 1$
  - **while** $a[j] = m[j] - 1$
    - $a[j] \leftarrow 0$
    - $j \leftarrow j + 1$
  - **if** $j = n + 1$:
    - **return**
  - $a[j] \leftarrow a[j] + 1$

By setting $m[i] = 2$ for all $i$, we can print every binary number from 0 to $2^n - 1$

Questions?

Section 2

# The Gray Code

# Becoming Lazier

- Consider the fact that I am extremely lazy

# Becoming Lazier

- Consider the fact that I am extremely lazy
- If we are counting in binary and we count `01111` $\to$ `10000`
  - We have to change a whole **5** digits The horror! The horror!
  - The inner **while** loop (Line 6) of ALGORITHM-M runs 5 times

# Becoming Lazier

- Consider the fact that I am extremely lazy
- If we are counting in binary and we count `01111` $\rightarrow$ `10000`
  - We have to change a whole **5** digits The horror! The horror!
  - The inner **while** loop (Line 6) of ALGORITHM-M runs 5 times
- $n$ digits means $2^n$ numbers are generated, so $O\left(2^n\right)$ is our limit

# Becoming Lazier

- Consider the fact that I am extremely lazy
- If we are counting in binary and we count `01111` $\to$ `10000`
  - We have to change a whole **5** digits The horror! The horror!
  - The inner **while** loop (Line 6) of ALGORITHM-M runs 5 times
- $n$ digits means $2^n$ numbers are generated, so $O\left(2^n\right)$ is our limit
- For each digit, this inner **while** loop runs $O(n)$ times, resulting in total runtime $O\left(n2^n\right)$

# Becoming Lazier

- Consider the fact that I am extremely lazy
- If we are counting in binary and we count `01111` $\rightarrow$ `10000`
  - We have to change a whole **5** digits The horror! The horror!
  - The inner **while** loop (Line 6) of ALGORITHM-M runs 5 times
- $n$ digits means $2^n$ numbers are generated, so $O\left(2^n\right)$ is our limit
- For each digit, this inner **while** loop runs $O(n)$ times, resulting in total runtime $O\left(n2^n\right)$
- Is there a way to avoid this and shave off a factor of $O(n)$?

# Gray Binary Code

- First appeared in a 1941 patent #2307868 by George R. Stibitz
- Frank Gray mentioned the code in a 1947 patent #2632058
  - As tradition, he gets the credit rather than the people before him

# Gray Binary Code

- First appeared in a 1941 patent #2307868 by George R. Stibitz
- Frank Gray mentioned the code in a 1947 patent #2632058
  - As tradition, he gets the credit rather than the people before him
- He described a systematic way to generate binary where each successive number changes by exactly 1 digit for each step

# Gray Binary Code

- First appeared in a 1941 patent #2307868 by George R. Stibitz
- Frank Gray mentioned the code in a 1947 patent #2632058
  - As tradition, he gets the credit rather than the people before him
- He described a systematic way to generate binary where each successive number changes by exactly 1 digit for each step
- Louis Gros published an anonymous note “Théorie du Baguenodier” in Lyonnais, 1872, describing the Gray binary code in relation to solving an ancient Chinese puzzle [Gro72].
  - So in reality, he is the inventor

# Generating the Gray Code

| | |
|---|---|
| $Zero$ | `0` |
| $One$ | `1` |

# Generating the Gray Code

| | |
|---|---|
| *Zero* | 0 |
| *One* | 1 |
| | 1 |
| | 0 |

# Generating the Gray Code

| | |
|---|---|
| *Zero* | 00 |
| *One* | 01 |
| | 11 |
| | 10 |

# Generating the Gray Code

| | |
|---|---|
| *Zero* | `00` |
| *One* | `01` |
| *Three* | `11` |
| *Two* | `10` |

# Generating the Gray Code

| | |
|---|---|
| *Zero* | 00 |
| *One* | 01 |
| *Three* | 11 |
| *Two* | 10 |
| | 10 |
| | 11 |
| | 01 |
| | 00 |

# Generating the Gray Code

| | |
|---|---|
| *Zero* | 000 |
| *One* | 001 |
| *Three* | 011 |
| *Two* | 010 |
| | 110 |
| | 111 |
| | 101 |
| | 100 |

# Generating the Gray Code

| | |
|---|---|
| *Zero* | 000 |
| *One* | 001 |
| *Three* | 011 |
| *Two* | 010 |
| *Six* | 110 |
| *Seven* | 111 |
| *Five* | 101 |
| *Four* | 100 |

# Generating the Gray Code

| | |
|---|---|
| *Zero* | 000 |
| *One* | 001 |
| *Three* | 011 |
| *Two* | 010 |
| *Six* | 110 |
| *Seven* | 111 |
| *Five* | 101 |
| *Four* | 100 |

# Recursive Definition

We can define the Gray Code recursively as follows

$$\begin{aligned} \Gamma_0 &= \varepsilon \\ \Gamma_{n+1} &= \mathtt{0} \cdot \Gamma_n,\ \mathtt{1} \cdot \Gamma_n^R \end{aligned} \tag{1}$$

where $\Gamma_n^R$ is $\Gamma_n$ and $0 \cdot \Gamma_n$ stands for appending 0 to every element in $\Gamma_n$

# Recursive Definition

We can define the Gray Code recursively as follows

$$\begin{aligned} \Gamma_0 &= \varepsilon \\ \Gamma_{n+1} &= \mathtt{0} \cdot \Gamma_n,\ \mathtt{1} \cdot \Gamma_n^R \end{aligned} \tag{1}$$

where $\Gamma_n^R$ is $\Gamma_n$ and $0 \cdot \Gamma_n$ stands for appending 0 to every element in $\Gamma_n$

You can use this to prove that $gray(k) = k \oplus \left\lfloor \frac{k}{2} \right\rfloor$

**Exercise:** Prove that $\Gamma_n$ generates all binary strings 0 to $2^n - 1$

# Recursive Gray Code Generation

Here is some recursive code that makes use of the formula in Eq. 1

RecursiveGray(n):

**if** $n = 0$:<br>
    **return** [“ ”]<br>
$\Gamma_n \leftarrow [\,]$<br>
$\Gamma_{n-1} \leftarrow \textsc{RecursiveGray}(n-1)$<br>
**for** $num \in \Gamma_{n-1}$:<br>
    $\Gamma_n.\textsc{append}(\mathtt{0} \cdot num)$<br>
**for** $num \in \Gamma_{n-1}^R$:<br>
    $\Gamma_n.\textsc{append}(\mathtt{1} \cdot num)$<br>
**return** $\Gamma_n$

# Recursive Gray Code Generation

Here is some recursive code that makes use of the formula in Eq. 1

$\underline{\textsc{RecursiveGray}(\mathrm{n})}$:

**if** $n = 0$:<br>
    **return** [“ ”]<br>
$\Gamma_n \leftarrow [\,]$<br>
$\Gamma_{n-1} \leftarrow \textsc{RecursiveGray}(n-1)$<br>
**for** $num \in \Gamma_{n-1}$:<br>
    $\Gamma_n.\textsc{append}(\mathtt{0} \cdot num)$<br>
**for** $num \in \Gamma_{n-1}^R$:<br>
    $\Gamma_n.\textsc{append}(\mathtt{1} \cdot num)$<br>
**return** $\Gamma_n$

Can we do this iteratively?

# Non-recursive Gray Code Generation

ALGORITHM-G$(n)$:

$a[i] \leftarrow 0$ **for** $1 \leq i \leq n$

$a[0] \leftarrow 0$ ⟪***Exercise:*** *why do we need this?*⟫

**while** TRUE:

  PRINT$(a[n] \cdots a[1])$

  $a[0] \leftarrow 1 - a[0]$

  $j \leftarrow$ **minimum** $j \geq 1$ such that $a[j-1] = 1$

  **if** $j = n + 1$:

    **return**

  $a[j] \leftarrow 1 - a[j]$

# Non-recursive Gray Code Generation

<u>Algorithm-G($n$)</u>:

1: $a[i] \leftarrow 0$ **for** $1 \leq i \leq n$
2: $a[0] \leftarrow 0$ ⟪***Exercise:*** *why do we need this?*⟫
3: **while** True:
4:     Print$(a[n] \cdots a[1])$
5:     $a[0] \leftarrow 1 - a[0]$
6:     $j \leftarrow$ **minimum** $j \geq 1$ such that $a[j-1] = 1$
7:     **if** $j = n + 1$:
8:         **return**
9:     $a[j] \leftarrow 1 - a[j]$

We haven't really gotten rid of the inner **while** loop (the **minimum** on Line 6 is kind of a **while** loop). However, we only edit the array $a$ once per iteration of the outer loop.

# Loopless Non-recursive Gray Code Generation

**Algorithm-L**($n$):

$a[i] \leftarrow 0$ **for** $1 \le i \le n$
$f[i] \leftarrow i$ **for** $1 \le i \le n+1$
**while** True:
  print$(a[n] \cdots a[1])$
  $j \leftarrow f[1]$
  $f[1] \leftarrow 1$
  **if** $j = n + 1$:
    **return**
  $a[j] \leftarrow 1 - a[j]$
  $f[j] \leftarrow f[j+1]$
  $f[j+1] \leftarrow j+1$

# What About Other Bases?

- Can we do this change-one-digit-at-a-time thing with other bases?

# What About Other Bases?

- Can we do this change-one-digit-at-a-time thing with other bases?
  - Yes! We can even do it looplessly

# What About Other Bases?

- Can we do this change-one-digit-at-a-time thing with other bases?
  - Yes! We can even do it looplessly
- There are two somewhat natural ways of doing this: *reflected*

$$000, 001, \ldots, 009, 019, 018, \ldots, 011, 010, 020, 021, 022, \ldots, 091, 090, 190, 191, \ldots$$

# What About Other Bases?

- Can we do this change-one-digit-at-a-time thing with other bases?
  - Yes! We can even do it looplessly
- There are two somewhat natural ways of doing this: *reflected*

$$000, 001, \ldots, 009, 019, 018, \ldots, 011, 010, 020, 021, 022, \ldots, 091, 090, 190, 191, \ldots$$

- and *modular*

$$000, 001, \ldots, 009, 019, 010, \ldots, 017, 018, 028, 029, 020, \ldots, 099, 090, 190, 191, \ldots$$

# What About Other Bases?

- Can we do this change-one-digit-at-a-time thing with other bases?
  - Yes! We can even do it looplessly
- There are two somewhat natural ways of doing this: *reflected*

$$000, 001, \ldots, 009, 019, 018, \ldots, 011, 010, 020, 021, 022, \ldots, 091, 090, 190, 191, \ldots$$

- and *modular*

$$000, 001, \ldots, 009, 019, 010, \ldots, 017, 018, 028, 029, 020, \ldots, 099, 090, 190, 191, \ldots$$

- The following algorithm will generate the *reflected* sequence.
  - **Exercise:** Modify it to produce the modular sequence

# Loopless Reflected Mixed-Radix Gray Generation

<u>ALGORITHM-H($m[1..n]$):</u>

$a[i] \leftarrow 0$ **for** $1 \leq i \leq n$
$f[i] \leftarrow i$ **for** $1 \leq i \leq n+1$
$d[i] \leftarrow 1$ **for** $1 \leq i \leq n$ ⟨⟨*directions*⟩⟩
**while** TRUE:
  PRINT($a[n] \cdots a[1]$)
  $j \leftarrow f[1]$
  $f[1] \leftarrow 1$
  **if** $j = n+1$:
    **return**
  $a[j] \leftarrow a[j] + d[j]$
  **if** $a[j] = 0$ **or** $a[j] = m[j] - 1$:
    $d[j] \leftarrow -d[j]$ ⟨⟨*change directions*⟩⟩
    $f[j] \leftarrow f[j+1]$
    $f[j+1] \leftarrow j+1$

Questions?

Section 3

# Towers of Hanoi and A Chinese Ring Puzzle

# Monks Moving Disks Until the World Ends

- In 1883, French mathematician Édouard Lucas introduced his puzzle “The Towers of Hanoi”

# Monks Moving Disks Until the World Ends

- In 1883, French mathematician Édouard Lucas introduced his puzzle “The Towers of Hanoi”
- Since then, this puzzle has had many mythical origin stories written about it

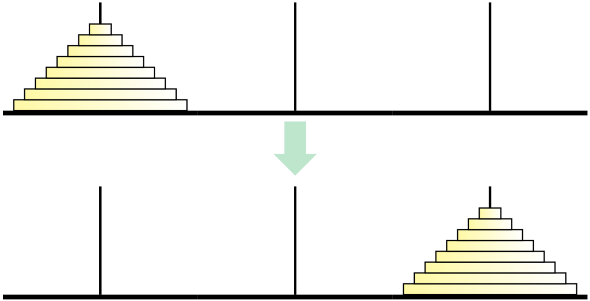

Figure: From [Eri19]

# Recursively Solving the Puzzle

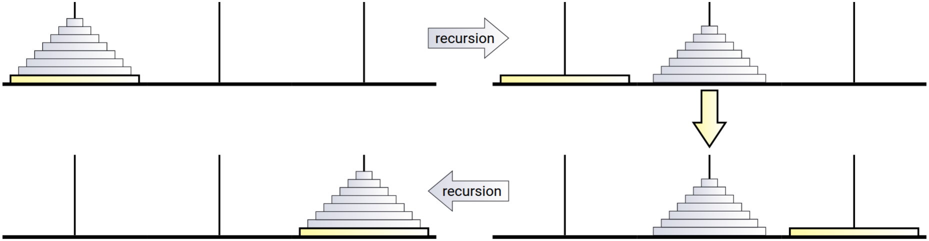


Figure: From [Eri19]

# Recursively Solving the Puzzle

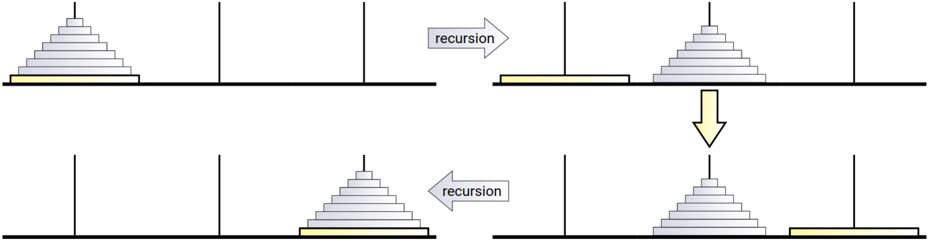


Figure: From [Eri19]

RECURSIVEHANOI(n):

1: Move the top $n-1$ disks using RECURSIVEHANOI$(n-1)$
2: Move the $n$th disk
3: Move the top $n-1$ disks using RECURSIVEHANOI$(n-1)$

# Iteratively Solving the Puzzle

- The recursive solution takes $O\left(2^n\right)$ time
- We can also solve the problem iteratively as follows

```
IterativeHanoi(n):
    until solved:
        Move the small disk to the right
        Make the only legal move not involving the small disk
```

Figure: From [Sed03]

# Solving the Puzzle using Binary

- The following iterative algorithm uses binary to iteratively solve the puzzle
- Suppose the smallest disk is disk 1 and the largest disk is $n$

```
BinaryHanoi(n):
  i ← 0
  while i < 2^n − 1:
      i ← i + 1
      d ← position of least significant 1 in binary(i)
      if d = 1:
          Move the small disk to the right
      else:
          Move disk d to the only legal position
```

Figure: From [3Bl16]

# A Chinese Ring Puzzle

- There is a similar puzzle, whose exact origin is unknown
  - In Chinese, it is known as “Jiu Lian Huan”
  - In French, it is known as “Baguenaudier”

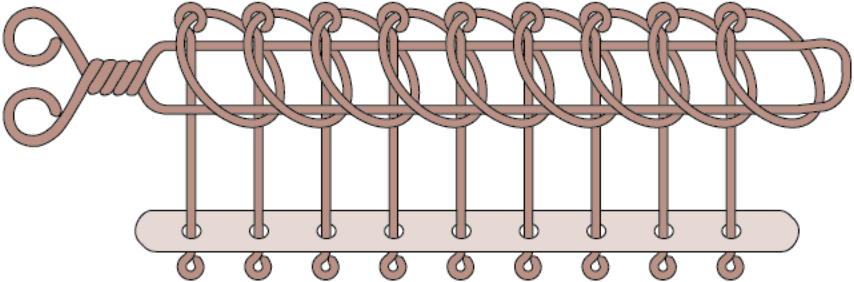

Figure: From [ZR21]

## Allowed Moves

There are two legal moves we can make at each time

- We can remove and replace the rightmost ring at any time
- Any other ring can be removed or replaced as long as the following two conditions are met:
  - The ring to its right is on the bar
  - Every ring to the right of that is off the bar

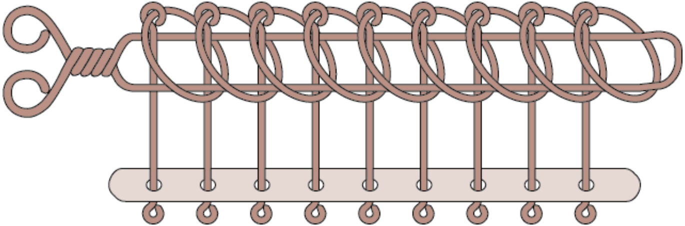

Figure: From [ZR21]

# Solving the Puzzle

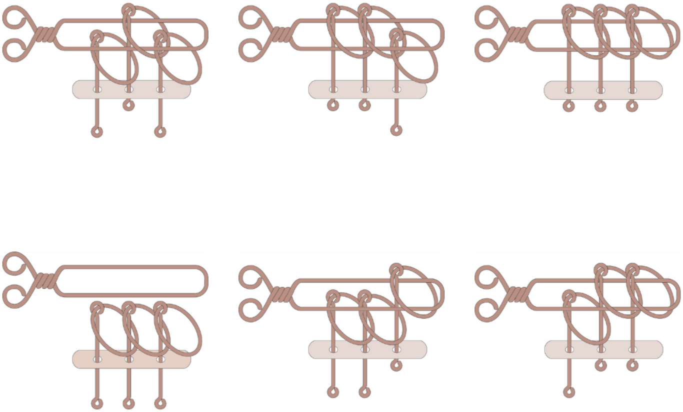

Figure: From [ZR21]

# Algorithmically Solving the Puzzle

- In his “Théorie du Baguenodier” Louis Gros connected the Gray binary code to solving this ring puzzle
- Let us abstract the puzzle into a series of binary digits

```
1111
```

- The binary digit will be `1` if the ring is on and `0` otherwise

# Algorithmically Solving the Puzzle

```
1111
```

# Algorithmically Solving the Puzzle

1111

1101

# Algorithmically Solving the Puzzle

1111

1101

1100

## Algorithmically Solving the Puzzle

1111

1101

1100

0100

0101

0111

0110

0010

0011

0001

0000

# Algorithmically Solving the Puzzle

1111

1101

1100

0100

0101

0111

0110

0010

0011

0001

0000

# Algorithmically Solving the Puzzle

1111

1101

1100

0100

0101

0111

0110

0010

0011

0001

0000

This is Gray binary code starting from 1111 and counting down to 0000

Questions?

*It has been said that combinatorics is both the easiest and hardest field of mathematics. Easy since a lot of it requires no prerequisite knowledge. Hence a High School Student can do work in it. Hard because a lot of it requires no prerequisite knowledge. Hence you can't easily apply continuous techniques.*

— WILLIAM GASARCH (2019)

# Bibliography


3Blue1Brown.
Binary, hanoi and sierpinski, part 1, Nov. 2016.

Jeff Erickson.
*Algorithms.*
1st edition, 06 2019.

Louis Gros.
*Théorie du Baguenodier.*
Imprimerie D'Aimé Aimé Vingtrinier, 1872.

Donald E. Knuth.
*The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms.*
Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

Donald E. Knuth.
*The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1.*
Addison-Wesley Professional, 1st edition, 2011.

R. Sedgewick.
*Algorithms In Java, Parts 1-4, 3/E.*
Pearson Education, 2003.

Wei Zhang and Peter Rasmussen.
Chinese nine linked rings puzzle, 2021.