Shift Gray Codes

Aaron Michael Williams
Combinatorial Generation
Combinatorial Generation

- Gray codes
Background

Combinatorial Generation

- Gray codes
- Universal cycles

Historical Foundations

- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
- Johnson-Trotter-Steinhaus (1962)

Contemporary Resources

- The Art of Computer Programming
- new fascicles by Knuth
- Combinatorial Generation
- by Ruskey

Thesis Goals

- Gray codes and universal cycles using shifts
- Efficient algorithms with simple implementations
- General results applicable to multiple combinatorial objects
Background

Combinatorial Generation

- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations

Binary reflected Gray code (1947)

de Bruijn cycle (1946)

Johnson-Trotter-Steinhaus (1962)

Contemporary Resources

The Art of Computer Programming

new fascicles by Knuth

Combinatorial Generation

by Ruskey

Generalizations of de Bruijn cycles and Gray codes

Thesis Goals

Gray codes and universal cycles using shifts

Efficient algorithms with simple implementations

General results applicable to multiple combinatorial objects
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
- Johnson-Trotter-Steinhaus (1962)

Contemporary Resources
- The Art of Computer Programming
- new fascicles by Knuth
- Combinatorial Generation
- by Ruskey

Thesis Goals
- Gray codes and universal cycles using shifts
- Efficient algorithms with simple implementations
- General results applicable to multiple combinatorial objects
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)

The binary reflected Gray code for \( n = 3 \)

000

000

000

000
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)

The binary reflected Gray code for $n = 3$
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)

The binary reflected Gray code for $n = 3$

The Art of Computer Programming
- New fascicles by Knuth

Combinatorial Generation by Ruskey

Generalizations of de Bruijn cycles and Gray codes

Thesis Goals
- Gray codes and universal cycles using shifts
- Efficient algorithms with simple implementations
- General results applicable to multiple combinatorial objects
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)

The binary reflected Gray code for $n = 3$

\[
\begin{array}{c}
0 \leftrightarrow 0 \\
1 \leftrightarrow 1 \\
0 \leftrightarrow 0 \\
\end{array}
\]
Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)

The binary reflected Gray code for $n = 3$
The binary reflected Gray code for \( n = 3 \)
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)

The binary reflected Gray code for $n = 3$
Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)

The binary reflected Gray code for $n = 3$
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)

The de Bruijn cycle for \( n = 3 \) using the FKM algorithm
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
- Johnson-Trotter-Steinhaus (1962)
### Background

#### Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

#### Historical Foundations
- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
- Johnson-Trotter-Steinhaus (1962)

![Johnson-Trotter-Steinhaus for $n = 4$](image-url)
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
- Johnson-Trotter-Steinhaus (1962)

Johnson-Trotter-Steinhaus for $n = 4$
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
- Johnson-Trotter-Steinhaus (1962)

Johnson-Trotter-Steinhaus for $n = 4$
## Background

### Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

### Historical Foundations
- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
- Johnson-Trotter-Steinhaus (1962)

$\begin{array}{c}
4 \\
\leftarrow \\
1 \\
2 \\
3 \\
\end{array}$

Johnson-Trotter-Steinhaus for $n = 4$
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
- Johnson-Trotter-Steinhaus (1962)

Contemporary Resources
- The Art of Computer Programming
  new fascicles by Knuth
- Combinatorial Generation
  by Ruskey

Thesis Goals
- Gray codes and universal cycles using shifts
- Efficient algorithms with simple implementations
- General results applicable to multiple combinatorial objects
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
- Johnson-Trotter-Steinhaus (1962)

Contemporary Resources
- *The Art of Computer Programming* new fascicles by Knuth
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
- Johnson-Trotter-Steinhaus (1962)

Contemporary Resources
- *The Art of Computer Programming* new fascicles by Knuth
- *Combinatorial Generation* by Ruskey
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
- Johnson-Trotter-Steinhaus (1962)

Contemporary Resources
- The Art of Computer Programming new fascicles by Knuth
- Combinatorial Generation by Ruskey
- Generalizations of de Bruijn cycles and Gray codes
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
- Johnson-Trotter-Steinhaus (1962)

Contemporary Resources
- *The Art of Computer Programming* new fascicles by Knuth
- *Combinatorial Generation* by Ruskey
- *Generalizations of de Bruijn cycles and Gray codes*

Thesis Goals
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
- Johnson-Trotter-Steinhaus (1962)

Contemporary Resources
- *The Art of Computer Programming* new fascicles by Knuth
- *Combinatorial Generation* by Ruskey
- *Generalizations of de Bruijn cycles and Gray codes*

Thesis Goals
- Gray codes and universal cycles using shifts
Background

Combinatorial Generation
- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations
- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
- Johnson-Trotter-Steinhaus (1962)

Contemporary Resources
- *The Art of Computer Programming* new fascicles by Knuth
- *Combinatorial Generation* by Ruskey
- *Generalizations of de Bruijn cycles and Gray codes*

Thesis Goals
- Gray codes and universal cycles using shifts
- Efficient algorithms with simple implementations
Background

Combinatorial Generation

- Gray codes
- Universal cycles
- Efficient algorithms

Historical Foundations

- Binary reflected Gray code (1947)
- de Bruijn cycle (1946)
- Johnson-Trotter-Steinhaus (1962)

Contemporary Resources

- *The Art of Computer Programming* new fascicles by Knuth
- *Combinatorial Generation* by Ruskey
- *Generalizations of de Bruijn cycles and Gray codes*

Thesis Goals

- Gray codes and universal cycles using shifts
- Efficient algorithms with simple implementations
- General results applicable to multiple combinatorial objects
Cool-lex Order
Reordering the bits in a binary string
Shift the first bit to the right until it passes over 01 or the last bit.
Reordering the bits in a binary string
Shift the first bit to the right until it passes over $01$ or the last bit.

$\begin{array}{c}
\text{1} \\
\text{1} \\
\text{1}
\end{array}$ $\begin{array}{c}
\text{1} \\
\text{1} \\
\text{1} \\
\text{0} \\
\text{0} \\
\text{0} \\
\text{0}
\end{array}$
Reordering the bits in a binary string
Shift the first bit to the right until it passes over 01 or the last bit.
Reordering the bits in a binary string
Shift the first bit to the right until it passes over 01 or the last bit.

10000111
Reordering the bits in a binary string
Shift the first bit to the right until it passes over $01$ or the last bit.

000111 → 000111
Reordering the bits in a binary string
Shift the first bit to the right until it passes over 01 or the last bit.

001011
Reordering the bits in a binary string
Shift the first bit to the right until it passes over 01 or the last bit.

010011
Reordering the bits in a binary string
Shift the first bit to the right until it passes over $001$ or the last bit.
Reordering the bits in a binary string
Shift the first bit to the right until it passes over 01 or the last bit.

001101
Reordering the bits in a binary string
Shift the first bit to the right until it passes over $01$ or the last bit.

$0 \rightarrow 0101011$
Reordering the bits in a binary string
Shift the first bit to the right until it passes over 01 or the last bit.

\[
\begin{array}{cccccc}
1 & 0 & 1 & 0 & 0 & 1 \\
\end{array}
\]
Reordering the bits in a binary string
Shift the first bit to the right until it passes over $01$ or the last bit.

$011001$
Reordering the bits in a binary string
Shift the first bit to the right until it passes over 01 or the last bit.
Reordering the bits in a binary string
Shift the first bit to the right until it passes over 00 or the last bit.
Reordering the bits in a binary string
Shift the first bit to the right until it passes over $01$ or the last bit.

$0011110$
Reordering the bits in a binary string
Shift the first bit to the right until it passes over 01 or the last bit.

0 1 0 1 1 1 0
Reordering the bits in a binary string
Shift the first bit to the right until it passes over $01$ or the last bit.

\[
\begin{array}{c}
10101010
\end{array}
\]
Reordering the bits in a binary string
Shift the first bit to the right until it passes over \(01\) or the last bit.

\[
\begin{array}{c}
0\\1\\1\\1\\0\\0\\1\\1\\0\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\0\\1\\1\\0\\0\\0\\0\end{array}
\]
Reordering the bits in a binary string
Shift the first bit to the right until it passes over 01 or the last bit.

1101100
Reordering the bits in a binary string
Shift the first bit to the right until it passes over 01 or the last bit.

1011100
Reordering the bits in a binary string
Shift the first bit to the right until it passes over $01$ or the last bit.

$0111100$
Reordering the bits in a binary string

Shift the first bit to the right until it passes over \(01\) or the last bit.

\[
\begin{array}{cccc}
1 & 1 & 1 & 0 \quad 0 \quad 0 \quad 0 \\
\end{array}
\]
Reordering the bits in a binary string
Shift the first bit to the right until it passes over 01 or the last bit.

This rule generates all combinations for any fixed number of 0s and any fixed number of 1s.
Reordering the bits in a binary string
Shift the first bit to the right until it passes over 01 or the last bit.

11110000

This rule generates all combinations for any fixed number of 0s and any fixed number of 1s.
The inverse rule is more efficient, since repeated applications can avoid scanning for the leftmost 01.
The inverse rule for binary string $s = s_1s_2 \cdots s_n$, where $k$ is the length of the longest $1^*0^*$ prefix in $s$: 

$$\text{cool}(s) = \begin{cases} 
\text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\
\text{shift}(s, k+2, 1) & \text{otherwise}.
\end{cases}$$
The inverse rule for binary string $s = s_1s_2 \cdots s_n$, where $k$ is the length of the longest $1^*0^*$ prefix in $s$:

$$\text{cool}(s) = \begin{cases} 
\text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\
\text{shift}(s, k + 2, 1) & \text{otherwise.}
\end{cases}$$
The inverse rule for binary string $s = s_1 s_2 \cdots s_n$, where $k$ is the length of the longest $1^* 0^*$ prefix in $s$:

$$
\overline{\text{cool}}(s) = \begin{cases} 
\overline{\text{shift}}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\
\overline{\text{shift}}(s, k + 2, 1) & \text{otherwise}.
\end{cases}
$$

Equivalently,

$$
\overline{\text{cool}}(s) = \begin{cases} 
\overline{\text{shift}}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\
\overline{\text{shift}}(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\
\overline{\text{shift}}(s, k + 2, 1) & \text{otherwise}
\end{cases}
$$
A Shift-Gray Code for Combinations

The inverse rule for binary string $s = s_1s_2 \cdots s_n$, where $k$ is the length of the longest $1^*0^*$ prefix in $s$:

$$\text{cool}(s) = \begin{cases} \text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\ \text{shift}(s, k + 2, 1) & \text{otherwise.} \end{cases}$$

Equivalently,

$$\text{cool}(s) = \begin{cases} \text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\ \text{shift}(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\ \text{shift}(s, k + 2, 1) & \text{otherwise.} \end{cases}$$

since $\cdots \overline{011} \cdots$ is equal to $\cdots \overline{0111} \cdots$. 
More generally, for a multiset permutation $s = s_1 s_2 \cdots s_n$, the rule generates the permutations of any multiset.
More generally, for a multiset permutation $s = s_1 s_2 \cdots s_n$, where $k = \lvert \lceil \downarrow (s) \rceil \rvert$ is the length of the non-increasing prefix:

$$\cool(s) = \begin{cases} 
\shift(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\
\shift(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\
\shift(s, k + 2, 1) & \text{otherwise.}
\end{cases}$$
A Shift-Gray Code for Multiset Permutations

More generally, for a multiset permutation $s = s_1 s_2 \cdots s_n$, where $k = |\\ll(s)|$ is the length of the *non-increasing prefix*:

$$\leftarrow \text{cool}(s) = \begin{cases} \leftarrow \text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\ \leftarrow \text{shift}(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\ \leftarrow \text{shift}(s, k + 2, 1) & \text{otherwise.} \end{cases}$$
A Shift-Gray Code for Multiset Permutations

More generally, for a multiset permutation \( s = s_1s_2 \cdots s_n \), where \( k = |\严格(s)| \) is the length of the non-increasing prefix:

\[
\text{cool}(s) = \begin{cases} 
\text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\
\text{shift}(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\
\text{shift}(s, k + 2, 1) & \text{otherwise.}
\end{cases}
\]
A Shift-Gray Code for Multiset Permutations

More generally, for a multiset permutation $s = s_1s_2 \cdots s_n$, where $k = |\prec(s)|$ is the length of the non-increasing prefix:

$$\text{cool}(s) = \begin{cases} \text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\ \text{shift}(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\ \text{shift}(s, k + 2, 1) & \text{otherwise.} \end{cases}$$
A Shift-Gray Code for Multiset Permutations

More generally, for a multiset permutation $s = s_1 s_2 \cdots s_n$, where $k = |\downarrow(s)|$ is the length of the non-increasing prefix:

$$\text{cool}(s) = \begin{cases} 
\text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\
\text{shift}(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\
\text{shift}(s, k + 2, 1) & \text{otherwise.}
\end{cases}$$

$\downarrow$
More generally, for a multiset permutation $s = s_1 s_2 \cdots s_n$, where $k = |\ll(s)|$ is the length of the non-increasing prefix:

$$\text{cool}(s) = \begin{cases} \text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\ \text{shift}(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\ \text{shift}(s, k + 2, 1) & \text{otherwise.} \end{cases}$$
A Shift-Gray Code for Multiset Permutations

More generally, for a multiset permutation \( s = s_1 s_2 \cdots s_n \), where \( k = |\preceq(s)| \) is the length of the non-increasing prefix:

\[
\text{cool}(s) = \begin{cases} 
\text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\
\text{shift}(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\
\text{shift}(s, k + 2, 1) & \text{otherwise.}
\end{cases}
\]

The rule generates the permutations of any multiset.
A Shift-Gray Code for Multiset Permutations

More generally, for a multiset permutation \( s = s_1 s_2 \cdots s_n \), where \( k = |\prec \!(s)| \) is the length of the \textit{non-increasing prefix}:

\[
\text{cool}(s) = \begin{cases} 
\text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\
\text{shift}(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\
\text{shift}(s, k + 2, 1) & \text{otherwise.}
\end{cases}
\]

The rule generates the permutations of any multiset.
More generally, for a multiset permutation $s = s_1s_2 \cdots s_n$, where $k = |\nabla(s)|$ is the length of the non-increasing prefix:

$$
\text{cool}(s) = \begin{cases} 
\text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\
\text{shift}(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\
\text{shift}(s, k + 2, 1) & \text{otherwise.}
\end{cases}
$$
More generally, for a multiset permutation $s = s_1 s_2 \cdots s_n$, where $k = |\prec(s)|$ is the length of the non-increasing prefix:

$$\text{cool}(s) = \begin{cases} \text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\ \text{shift}(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\ \text{shift}(s, k + 2, 1) & \text{otherwise.} \end{cases}$$
More generally, for a multiset permutation $s = s_1 s_2 \cdots s_n$, where $k = |\prec(s)|$ is the length of the non-increasing prefix:

$$\text{cool}(s) = \begin{cases} 
\text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\
\text{shift}(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\
\text{shift}(s, k + 2, 1) & \text{otherwise.}
\end{cases}$$
More generally, for a multiset permutation $s = s_1 s_2 \cdots s_n$, where $k = |\nabla(s)|$ is the length of the non-increasing prefix:

$$\text{cool}(s) = \begin{cases} 
\text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\
\text{shift}(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\
\text{shift}(s, k + 2, 1) & \text{otherwise.}
\end{cases}$$

The rule generates the permutations of any multiset.
More generally, for a multiset permutation $s = s_1s_2 \cdots s_n$, where $k = |\downarrow(s)|$ is the length of the non-increasing prefix:

$$\text{cool}(s) = \begin{cases} \text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\ \text{shift}(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\ \text{shift}(s, k + 2, 1) & \text{otherwise.} \end{cases}$$

The rule generates the permutations of any multiset.
More generally, for a multiset permutation $s = s_1s_2 \cdots s_n$, where $k = |\sqcap(s)|$ is the length of the non-increasing prefix:

$$\text{cool}(s) = \begin{cases} 
\text{shift}(s, n, 1) & \text{if } k = n \text{ or } k = n - 1 \\
\text{shift}(s, k + 1, 1) & \text{if } s_k < s_{k+2} \\
\text{shift}(s, k + 2, 1) & \text{otherwise.}
\end{cases}$$

The rule generates the permutations of any multiset.
Co-lex Order vs Cool-lex Order

<table>
<thead>
<tr>
<th>co-lex order</th>
<th>cool-lex order</th>
</tr>
</thead>
<tbody>
<tr>
<td>3221</td>
<td>1322</td>
</tr>
<tr>
<td>2321</td>
<td>3122</td>
</tr>
<tr>
<td>2231</td>
<td>2312</td>
</tr>
<tr>
<td>3212</td>
<td>1232</td>
</tr>
<tr>
<td>2312</td>
<td>2132</td>
</tr>
<tr>
<td>3122</td>
<td>3212</td>
</tr>
<tr>
<td>1322</td>
<td>2321</td>
</tr>
<tr>
<td>2132</td>
<td>2231</td>
</tr>
<tr>
<td>1232</td>
<td>1223</td>
</tr>
<tr>
<td>2213</td>
<td>2123</td>
</tr>
<tr>
<td>2123</td>
<td>2213</td>
</tr>
<tr>
<td>1223</td>
<td>3221</td>
</tr>
</tbody>
</table>

The permutations of \{1, 2, 2, 3\} in two different orders.
### Co-lex Order vs Cool-lex Order

<table>
<thead>
<tr>
<th>co-lex order</th>
<th>cool-lex order</th>
</tr>
</thead>
<tbody>
<tr>
<td>3221</td>
<td>1322</td>
</tr>
<tr>
<td>2321</td>
<td>3122</td>
</tr>
<tr>
<td>2231</td>
<td>2312</td>
</tr>
<tr>
<td>3212</td>
<td>1232</td>
</tr>
<tr>
<td>2312</td>
<td>2132</td>
</tr>
<tr>
<td>3122</td>
<td>3212</td>
</tr>
<tr>
<td>1322</td>
<td>2321</td>
</tr>
<tr>
<td>2132</td>
<td>2231</td>
</tr>
<tr>
<td>1232</td>
<td>1223</td>
</tr>
<tr>
<td>2213</td>
<td>2123</td>
</tr>
<tr>
<td>2123</td>
<td>2213</td>
</tr>
<tr>
<td>1223</td>
<td>3221</td>
</tr>
</tbody>
</table>

The permutations of \{1, 2, 2, 3\} in two different orders.

Co-lex order is (usually) defined recursively by increasing rightmost symbol.
### Co-lex Order vs Cool-lex Order

<table>
<thead>
<tr>
<th>co-lex order</th>
<th>cool-lex order</th>
</tr>
</thead>
<tbody>
<tr>
<td>3221</td>
<td>1322</td>
</tr>
<tr>
<td>2321</td>
<td>3122</td>
</tr>
<tr>
<td>2231</td>
<td>2312</td>
</tr>
<tr>
<td>3212</td>
<td>1232</td>
</tr>
<tr>
<td>2312</td>
<td>2132</td>
</tr>
<tr>
<td>3122</td>
<td>3212</td>
</tr>
<tr>
<td>1322</td>
<td>2321</td>
</tr>
<tr>
<td>2132</td>
<td>2231</td>
</tr>
<tr>
<td>1232</td>
<td>1223</td>
</tr>
<tr>
<td>2213</td>
<td>2123</td>
</tr>
<tr>
<td>2123</td>
<td>2213</td>
</tr>
<tr>
<td>1223</td>
<td>3221</td>
</tr>
</tbody>
</table>

The permutations of \(\{1, 2, 2, 3\}\) in two different orders.

Co-lex order is (usually) defined recursively by increasing rightmost symbol.
Co-lex Order vs Cool-lex Order

<table>
<thead>
<tr>
<th>co-lex order</th>
<th>cool-lex order</th>
</tr>
</thead>
<tbody>
<tr>
<td>322 1</td>
<td>1322</td>
</tr>
<tr>
<td>232 1</td>
<td>3122</td>
</tr>
<tr>
<td>223 1</td>
<td>2312</td>
</tr>
<tr>
<td>321 2</td>
<td>1232</td>
</tr>
<tr>
<td>231 2</td>
<td>2132</td>
</tr>
<tr>
<td>312 2</td>
<td>3212</td>
</tr>
<tr>
<td>132 2</td>
<td>2321</td>
</tr>
<tr>
<td>213 2</td>
<td>2231</td>
</tr>
<tr>
<td>123 2</td>
<td>1223</td>
</tr>
<tr>
<td>221 3</td>
<td>2123</td>
</tr>
<tr>
<td>212 3</td>
<td>2213</td>
</tr>
<tr>
<td>122 3</td>
<td>3221</td>
</tr>
</tbody>
</table>

The permutations of \{1, 2, 2, 3\} in two different orders.

Co-lex order is (usually) defined recursively by increasing rightmost symbol.
The permutations of \{1, 2, 2, 3\} in two different orders.

The *scut* of a string is its shortest suffix that is not a suffix of its non-increasing arrangement of symbols (3221).
The permutations of \( \{1, 2, 2, 3\} \) in two different orders.

The *scut* of a string is its shortest suffix that is not a suffix of its non-increasing arrangement of symbols (\(3221\)).
The permutations of \{1, 2, 2, 3\} in two different orders.

The *scut* of a string is its shortest suffix that is not a suffix of its non-increasing arrangement of symbols (3221).
The permutations of \{1, 2, 2, 3\} in two different orders.

The \textit{scut} of a string is its shortest suffix that is not a suffix of its non-increasing arrangement of symbols (3221).
The permutations of \{1, 2, 2, 3\} in two different orders.

The scut of a string is its shortest suffix that is not a suffix of its non-increasing arrangement of symbols (3221).
The permutations of \{1, 2, 2, 3\} in two different orders.

The \textit{scut} of a string is its shortest suffix that is not a suffix of its non-increasing arrangement of symbols (3 2 2 1).
The permutations of \{1, 2, 2, 3\} in two different orders.

The *scut* of a string is its shortest suffix that is not a suffix of its non-increasing arrangement of symbols (3221).
Figure: Co-lex order for (5, 5)-combinations.
Figure: Cool-lex order for (5, 5)-combinations.
Figure: Co-lex order for permutations of \{1, 2, 3, 4, 5, 6, 7\}.
Figure: Cool-lex order for permutations of \{1, 2, 3, 4, 5, 6, 7\}. 

Visualizations
Algorithms for Combinations

\[ b \leftarrow \text{array}(1^t 0^s) \]
\[ x \leftarrow t \]
\[ y \leftarrow t \]
\[ \text{visit}(b) \]

\textbf{while} \: x < s + t \: \textbf{do}

\[ b[x] \leftarrow 0 \]
\[ b[y] \leftarrow 1 \]
\[ x \leftarrow x + 1 \]
\[ y \leftarrow y + 1 \]
\[ \textbf{if} \: b[x] = 0 \]
\[ b[x] \leftarrow 1 \]
\[ b[1] \leftarrow 0 \]
\[ \textbf{if} \: y > 2 \]
\[ x \leftarrow 2 \]
\[ \textbf{end} \]
\[ y \leftarrow 1 \]
\[ \textbf{end} \]
\[ \text{visit}(b) \]
\[ \textbf{end} \]

\[ R_2 \leftarrow (1 \ll s + t) \]
\[ R_3 \leftarrow (1 \ll t) - 1 \]

\textbf{while} \: R_3 \land R_2 = 0 \: \textbf{do}

\[ \text{visit}(R_3) \]
\[ R_0 \leftarrow R_3 \land (R_3 + 1) \]
\[ R_1 \leftarrow R_0 \oplus (R_0 - 1) \]
\[ R_0 \leftarrow R_1 + 1 \]
\[ R_1 \leftarrow R_1 \land R_3 \]
\[ R_0 \leftarrow (R_0 \land R_3) \ominus 1 \]
\[ R_3 \leftarrow R_3 + R_1 - R_0 \]
\[ \textbf{end} \]

The computer word algorithm appears in \textit{The Art of Computer Programming} and is distributed with \textit{mmix}. The array algorithm appears in the special issue of \textit{Discrete Mathematics}.
Algorithms for Combinations

\[ b \leftarrow \text{array}(1^t0^s) \]
\[ x \leftarrow t \]
\[ y \leftarrow t \]
\[ \text{visit}(b) \]
\[ \textbf{while} \ x < s + t \ \textbf{do} \]
\[ b[x] \leftarrow 0 \]
\[ b[y] \leftarrow 1 \]
\[ x \leftarrow x + 1 \]
\[ y \leftarrow y + 1 \]
\[ \textbf{if} \ b[x] = 0 \]
\[ b[x] \leftarrow 1 \]
\[ b[1] \leftarrow 0 \]
\[ \textbf{if} \ y > 2 \]
\[ x \leftarrow 2 \]
\[ \textbf{end} \]
\[ y \leftarrow 1 \]
\[ \textbf{end} \]
\[ \text{visit}(b) \]
\[ \textbf{end} \]

\[ R_2 \leftarrow (1 \ll s + t) \]
\[ R_3 \leftarrow (1 \ll t) - 1 \]
\[ \textbf{while} \ R_3 \land R_2 = 0 \ \textbf{do} \]
\[ \text{visit}(R_3) \]
\[ R_0 \leftarrow R_3 \land (R_3 + 1) \]
\[ R_1 \leftarrow R_0 \oplus (R_0 - 1) \]
\[ R_0 \leftarrow R_1 + 1 \]
\[ R_1 \leftarrow R_1 \land R_3 \]
\[ R_0 \leftarrow (R_0 \land R_3) \ominus 1 \]
\[ R_3 \leftarrow R_3 + R_1 - R_0 \]
\[ \textbf{end} \]

- The computer word algorithm appears in *The Art of Computer Programming* and is distributed with MMIX.
\begin{verbatim}
\texttt{b} ← \texttt{array(1^{t0^s})}
\texttt{x} ← t
\texttt{y} ← t
\texttt{visit(b)}
\texttt{while} \texttt{x} < s + t \texttt{ do}
   \texttt{b}[x] ← 0
   \texttt{b}[y] ← 1
   \texttt{x} ← x + 1
   \texttt{y} ← y + 1
   \texttt{if} \texttt{b[x]} = 0
      \texttt{b[x]} ← 1
      \texttt{b}[1] ← 0
      \texttt{if} \texttt{y} > 2
         \texttt{x} ← 2
   \texttt{end}
\texttt{y} ← 1
\texttt{end}
\texttt{visit(b)}
\texttt{end}
\texttt{R_2} ← (1 \ll s + t)
\texttt{R_3} ← (1 \ll t) − 1
\texttt{while} \texttt{R_3} \land \texttt{R_2} = 0 \texttt{ do}
   \texttt{visit(R_3)}
   \texttt{R_0} ← \texttt{R_3} \land (\texttt{R_3} + 1)
   \texttt{R_1} ← \texttt{R_0} \oplus (\texttt{R_0} − 1)
   \texttt{R_0} ← \texttt{R_1} + 1
   \texttt{R_1} ← \texttt{R_1} \land \texttt{R_3}
   \texttt{R_0} ← (\texttt{R_0} \land \texttt{R_3}) \lor 1
   \texttt{R_3} ← \texttt{R_3} + \texttt{R_1} − \texttt{R_0}
\texttt{end}
\end{verbatim}

- The computer word algorithm appears in *The Art of Computer Programming* and is distributed with MMIX.
- The array algorithm appears in the special issue of *Discrete Mathematics*.
Algorithm for Multiset Permutations

\[ [h, i, j] \leftarrow \text{init}(M) \]

\text{visit}(h)

\textbf{while} \ j.\text{next} \neq \phi \ \text{or} \ j.\text{val} < h.\text{val} \ \textbf{do}

\hspace{1em} \textbf{if} \ j.\text{next} \neq \phi \ \text{and} \ i.\text{val} \geq j.\text{next}.\text{val}

\hspace{2em} s \leftarrow j

\hspace{1em} \textbf{else}

\hspace{2em} s \leftarrow i

\hspace{1em} \textbf{end}

\hspace{1em} t \leftarrow s.\text{next}

\hspace{1em} s.\text{next} \leftarrow t.\text{next}

\hspace{1em} t.\text{next} \leftarrow h

\hspace{1em} \textbf{if} \ t.\text{val} < h.\text{val}

\hspace{2em} i \leftarrow t

\hspace{1em} \textbf{end}

\hspace{1em} j \leftarrow i.\text{next}

\hspace{1em} h \leftarrow t

\hspace{1em} \text{visit}(h)

\textbf{end}
Algorithm for Multiset Permutations

[h, i, j] ← init(M)
visit(h)

while j.next ≠ φ or j.val < h.val do
    if j.next ≠ φ and i.val ≥ j.next.val
        s ← j
    else
        s ← i
    end
    t ← s.next
    s.next ← t.next
    t.next ← h
if t.val < h.val
    i ← t
end
j ← i.next
h ← t
visit(h)
end

Simple.
Algorithm for Multiset Permutations

\[ [h, i, j] \leftarrow \text{init}(M) \]
\[ \text{visit}(h) \]
\[ \textbf{while} \ j.\text{next} \neq \phi \text{ or } j.\text{val} < h.\text{val} \text{ do} \]
\[ \quad \textbf{if} \ j.\text{next} \neq \phi \text{ and } i.\text{val} \geq j.\text{next}.\text{val} \]
\[ \quad \quad s \leftarrow j \]
\[ \quad \textbf{else} \]
\[ \quad \quad s \leftarrow i \]
\[ \quad \textbf{end} \]
\[ t \leftarrow s.\text{next} \]
\[ s.\text{next} \leftarrow t.\text{next} \]
\[ t.\text{next} \leftarrow h \]
\[ \textbf{if} \ t.\text{val} < h.\text{val} \]
\[ \quad i \leftarrow t \]
\[ \textbf{end} \]
\[ j \leftarrow i.\text{next} \]
\[ h \leftarrow t \]
\[ \text{visit}(h) \]
\[ \textbf{end} \]

- Simple.
- The first loopless algorithm using a constant number of additional variables for generating multiset permutations.
Algorithm for Multiset Permutations

[h, i, j] ← init(M)
visit(h)

while j.next ≠ φ or j.val < h.val do
  if j.next ≠ φ and i.val ≥ j.next.val
    s ← j
  else
    s ← i
  end
  t ← s.next
  s.next ← t.next
  t.next ← h
  if t.val < h.val
    i ← t
  end
  j ← i.next
  h ← t
  visit(h)
end

• Simple.

• The first loopless algorithm using a constant number of additional variables for generating multiset permutations.

• This linked list algorithm was presented at SODA 2009 (Symposium on Discrete Algorithms).
Application: Stacker-Crane Problem

Deliver objects from origins to destinations. Solutions modeled by multiset permutations. Minimize time. NP-complete decision problem.

Number times changed / operation adjacent-transpose prefix-shift

3 4 2 3

Prefix-shift Gray codes are ideal for exhaustive searches. Adjacent-transposition Gray codes do not exist for multiset permutations.
Application: Stacker-Crane Problem

Deliver objects from origins to destinations.
Solutions modeled by multiset permutations.
Minimize time.
NP-complete decision problem.

Number times changed / operation adjacent-transpose prefix-shift

3 4 2 3

Prefix-shift Gray codes are ideal for exhaustive searches.
Adjacent-transposition Gray codes do not exist for multiset permutations.
Application: Stacker-Crane Problem

Deliver objects from origins to destinations.

Solutions modeled by multiset permutations.

Minimize time.

NP-complete decision problem.

Number times changed / operation

Adjacent-transpose prefix-shift

3

4

2

3

Prefix-shift Gray codes are ideal for exhaustive searches.

Adjacent-transposition Gray codes do not exist for multiset permutations.
Application: Stacker-Crane Problem

- Deliver objects from origins to destinations.

Solutions modeled by multiset permutations.

Minimize time.

NP-complete decision problem.

Number times changed / operation adjacent-transpose prefix-shift

Prefix-shift Gray codes are ideal for exhaustive searches.

Adjacent-transposition Gray codes do not exist for multiset permutations.
Application: Stacker-Crane Problem

- Deliver objects from origins to destinations.
- Solutions modeled by multiset permutations.
Application: Stacker-Crane Problem

- Deliver objects from origins to destinations.
- Solutions modeled by multiset permutations.
- Minimize time.
Application: Stacker-Crane Problem

- Deliver objects from origins to destinations.
- Solutions modeled by multiset permutations.
- Minimize time.
- NP-complete decision problem.
Application: Stacker-Crane Problem

- Deliver objects from origins to destinations.
- Solutions modeled by multiset permutations.
- Minimize time.
- NP-complete decision problem.

Number times changed / operation
Application: Stacker-Crane Problem

- Deliver objects from origins to destinations.
- Solutions modeled by multiset permutations.
- Minimize time.
- NP-complete decision problem.

Number times changed / operation

<table>
<thead>
<tr>
<th>Operation</th>
<th>3</th>
<th>4</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjacent-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>transpose</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>prefix-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Application: Stacker-Crane Problem

- Deliver objects from origins to destinations.
- Solutions modeled by multiset permutations.
- Minimize time.
- NP-complete decision problem.

Number times changed / operation

<table>
<thead>
<tr>
<th>adjacent-</th>
<th>transpose</th>
<th>prefix-</th>
<th>shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

- Prefix-shift Gray codes are ideal for exhaustive searches.
Application: Stacker-Crane Problem

- Deliver objects from origins to destinations.
- Solutions modeled by multiset permutations.
- Minimize time.
- NP-complete decision problem.

Number times changed / operation

<table>
<thead>
<tr>
<th>adjacent-</th>
<th>transpose</th>
<th>prefix-</th>
<th>shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

- Prefix-shift Gray codes are ideal for exhaustive searches.
- Adjacent-transposition Gray codes do not exist for multiset permutations.
Bubble Languages
A *fixed-content language* is a set of strings in which every string has the same content.
A fixed-content language is a set of strings in which every string has the same content.

\[
L = \{11233, 11323, 12133\}
\]
A *fixed-content language* is a set of strings in which every string has the same content. (*Fixed-density language* if content($L$) has only 0s and 1s.)

$$L = \{11233, 11323, 12133\}$$
A *fixed-content language* is a set of strings in which every string has the same content.

\[
L = \{11233, 11323, 12133\}
\]
A fixed-content language is a set of strings in which every string has the same content.

$$ L = \{11233, 11323, 12133\} $$

The frozen prefix of $$ s \in L $$ is the longest prefix of $$ s $$ that cannot be rearranged to create a different string in the language.
A *fixed-content language* is a set of strings in which every string has the same content. 

\[ L = \{11233, 11323, 12133\} \]

The *frozen prefix* of \(s \in L\) is the longest prefix of \(s\) that cannot be rearranged to create a different string in the language.

\[ \text{f}_{L}(11233) = 11 \quad \text{f}_{L}(11323) = 113 \]
A *fixed-content language* is a set of strings in which every string has the same content.

\[ L = \{11233, 11323, 12133\} \]

The *frozen prefix* of \( s \in L \) is the longest prefix of \( s \) that cannot be rearranged to create a different string in the language.

\[ \text{frozen prefix}_L(11233) = 11 \quad \text{frozen prefix}_L(11323) = 113 \]

A *greedy left-shift* moves a symbol to the left until the result is outside of the language.
A *fixed-content language* is a set of strings in which every string has the same content.

$$L = \{11233, 11323, 12133\}$$

The *frozen prefix* of $s \in L$ is the longest prefix of $s$ that cannot be rearranged to create a different string in the language.

$$\downarrow_L(11233) = 11 \quad \downarrow_L(11323) = 113$$

A *greedy left-shift* moves a symbol to the left until the result is outside of the language.

$$\text{greedy}_L(11323, 4) = 11323 = 12133$$
A *fixed-content language* is a set of strings in which every string has the same content.

\[ L = \{11233, 11323, 12133\} \]

The *frozen prefix* of \( s \in L \) is the longest prefix of \( s \) that cannot be rearranged to create a different string in the language.

\[ \text{frozen prefix}_L(11233) = 11 \quad \text{frozen prefix}_L(11323) = 113 \]

A *greedy left-shift* moves a symbol to the left until the result is outside of the language.

\[ \text{greedy}_L(11323, 4) = 11323 = 12133 \]

A *bubble left-shift* moves a symbol to the left past one differing symbol.
A **fixed-content language** is a set of strings in which every string has the same content.

\[
L = \{11233, 11323, 12133\}
\]

The *frozen prefix* of \(s \in L\) is the longest prefix of \(s\) that cannot be rearranged to create a different string in the language.

\[
\downarrow_L(11233) = 11 \quad \downarrow_L(11323) = 113
\]

A **greedy left-shift** moves a symbol to the left until the result is outside of the language.

\[
\text{greedy}_L(11323, 4) = 11323 = 12133
\]

A **bubble left-shift** moves a symbol to the left past one differing symbol.

\[
\text{bubble}(11233, 5) = 11233 = 11323
\]
A fixed-content language $L$ is a *bubble language* if it satisfies:

(A) For every $s \in L$ with $s \neq \varepsilon$,

$$s \leftarrow \text{bubble}(s, |\varepsilon(s)| + 1) \in L.$$

(B) For every $s \in L$,

$$s \leftarrow \text{bubble}(s, i) \in L \text{ for all } i \text{ within } |s| < i \leq |\varepsilon(s)|.$$

**Lemmas:**

Prefixes can be "bubble sorted" into non-increasing order by (A).

(A) implies (B) when $\varepsilon(s)$ contains two distinct symbols.
A fixed-content language $L$ is a \textit{bubble language} if it satisfies:

\textbf{(A)} For every $s \in L$ with $s \neq \bot(s)$

\[
\text{bubble}(s, |\bot(s)| + 1) \in L.
\]
A fixed-content language $L$ is a *bubble language* if it satisfies:

**(A)** For every $s \in L$ with $s \neq \neg(s)$

$$\text{bubble}(s, |\neg(s)| + 1) \in L.$$

**(B)** For every $s \in L$

$$\text{bubble}(s, i) \in L \text{ for all } i \text{ within } |\bullet(s)| < i \leq |\neg(s)|.$$
A fixed-content language $L$ is a *bubble language* if it satisfies:

**(A)** For every $s \in L$ with $s \neq \bot(s)$

$$ \text{bubble}(s, |\bot(s)| + 1) \in L. $$

**(B)** For every $s \in L$

$$ \text{bubble}(s, i) \in L \text{ for all } i \text{ within } |\text{bullet}(s)| < i \leq |\bot(s)|. $$

**Lemmas:**

- Prefixes can be “bubble sorted” into non-increasing order by **(A).**
A fixed-content language $L$ is a *bubble language* if it satisfies:

**(A)** For every $s \in L$ with $s \neq \bot(s)$

$$\text{bubble}(s, |\bot(s)| + 1) \in L.$$ 

**(B)** For every $s \in L$

$$\text{bubble}(s, i) \in L \text{ for all } i \text{ within } |\cdot(s)| < i \leq |\bot(s)|.$$ 

**Lemmas:**

- Prefixes can be “bubble sorted” into non-increasing order by **(A)**.
- **(A)** implies **(B)** when $\bot(s)$ contains two distinct symbols.
Examples of Bubble Languages

Motzkin paths from (0,0) to (4,0) with two “east” edges.
Examples of Bubble Languages

Motzkin paths from (0,0) to (4,0) with two “east” edges. Restricted Motzkin paths can be represented by bubble languages.

\{2011, 2101, 2110, 1201, 1210, 1120\}
Motzkin paths from $(0,0)$ to $(4,0)$ with two “east” edges. Restricted Motzkin paths can be represented by bubble languages.

\[
\{2011, 2101, 2110, 1201, 1210, 1120\}
\]

Balanced parentheses (Catalan paths) can also be represented by bubble languages.
Examples of Bubble Languages

Walther Franz Anton von Dyck
Examples of Bubble Languages

$k$-ary Dyck words can be represented by bubble languages.

\[
\{1110000000, 1101000000, 1100110000, 1100011000, \\
1100010100, 1100001100, 1010100000, 1010010000, 1010001000, \\
1001100000, 1001010000, 1001001000\}
\]

2-ary Dyck words of length 9

Walther Franz Anton von Dyck .
Examples of Bubble Languages

Walther Franz Anton von Dyck

$k$-ary Dyck words can be represented by bubble languages.

\{
1110000000, 1110100000, 1110010000,
1100010000, 1100001000, 1011100000,
1010110000, 1010011000, 1010001100,
1001110000, 1001011000, 1001001100
\}

2-ary Dyck words of length 9

Linear-extensions of $B$-posets can also be represented by bubble languages.
Examples of Bubble Languages

Connected unit interval graphs with five vertices.
Examples of Bubble Languages

Connected unit interval graphs with five vertices. Connected unit intervals can be represented by bubble languages.

The bubble language uses the interval representation.
Examples of Bubble Languages

Ordered trees with branching sequence \( \{0, 0, 0, 1, 1, 3\} \).
Examples of Bubble Languages

Ordered trees with branching sequence \{0, 0, 0, 1, 1, 3\}.

Ordered trees with a fixed branching sequence can be represented by bubble languages.

\{31100, 31010, 31001, 30110, 30101, 30011, 13100, 13010, 13001, 11300\}
Examples of Bubble Languages

. Necklaces containing \{1, 1, 2, 2, 3, 3\}. 
Examples of Bubble Languages

- Necklaces containing \{1, 1, 2, 2, 3, 3\}.

Multiset necklaces can be represented by bubble languages.

\{3322111, 3321112, 3312112, 3232111, 3321211, 3312211, 3311222, 3231211, 3231112, 3221311, 3213112, 313122, 3223111, 3213211, 3212311, 3123112\}. 
Examples of Bubble Languages

Necklaces containing \{1, 1, 2, 2, 3, 3\}.

Multiset necklaces can be represented by bubble languages.

\{33322111, 33321112, 33312112, 32322111, 33321211, 33312211, 33111222, 32312111, 32311122, 32213111, 32131211, 31312222, 32231111, 32132111, 32123111, 31231212\}.

Lyndon words over a multiset can also be represented by bubble languages.
If $L$ is a bubble language, then the following operation generates a circular left-shift Gray code where $s \in L$ and $k = |\mathcal{L}(s)|$. 

\[
\text{cool}_L(s) = \begin{cases} 
\text{greedy}(s, n) & \text{if } k = n \text{ or } k = n - 1 \\
\text{greedy}(s, k + 1) & \text{if } s_k < s_{k+2} \\
\text{greedy}(s, k + 2) & \text{otherwise}
\end{cases}
\]
If $L$ is a bubble language, then the following operation generates a circular left-shift Gray code where $s \in L$ and $k = |\mathbb{L}(s)|$.

$$
cool_L(s) = \begin{cases}
  \text{greedy}(s, n) & \text{if } k = n \text{ or } k = n - 1 \\
  \text{greedy}(s, k + 1) & \text{if } s_k < s_{k+2} \text{ or } \text{greedy}(s, k + 2) = s \\
  \text{greedy}(s, k + 2) & \text{otherwise}
\end{cases}
$$
If $L$ is a bubble language, then the following operation generates a circular left-shift Gray code where $s \in L$ and $k = |\mathbb{L}(s)|$.

$$\text{cool}_L(s) = \begin{cases} \text{greedy}(s, n) & \text{if } k = n \text{ or } k = n - 1 \\ \text{greedy}(s, k + 1) & \text{if } s_k < s_{k+2} \text{ or } \text{greedy}(s, k + 2) = s \\ \text{greedy}(s, k + 2) & \text{otherwise} \end{cases}$$

For example, ordered trees with branching sequence $\{0, 0, 0, 1, 1, 3\}$
Main Theorem

If \( L \) is a bubble language, then the following operation generates a circular left-shift Gray code where \( s \in L \) and \( k = |L(s)| \).

\[
\text{cool}_L(s) = \begin{cases} 
\text{greedy}(s, n) & \text{if } k = n \text{ or } k = n - 1 \\
\text{greedy}(s, k + 1) & \text{if } s_k < s_{k+2} \text{ or } \text{greedy}(s, k + 2) = s \\
\text{greedy}(s, k + 2) & \text{otherwise}
\end{cases}
\]

For example, ordered trees with branching sequence \( \{0, 0, 0, 1, 1, 3\} \):

\[
\text{cool}_{\{0, 0, 0, 1, 1, 3\}},
\]

\( k = 1, 2 \)
Main Theorem

If $L$ is a bubble language, then the following operation generates a circular left-shift Gray code where $s \in L$ and $k = |\mathcal{L}(s)|$.

$$\text{cool}_L(s) = \begin{cases} 
\text{greedy}(s, n) & \text{if } k = n \text{ or } k = n - 1 \\
\text{greedy}(s, k + 1) & \text{if } s_k < s_{k+2} \text{ or } \text{greedy}(s, k + 2) = s \\
\text{greedy}(s, k + 2) & \text{otherwise}
\end{cases}$$

For example, ordered trees with branching sequence \{0, 0, 0, 1, 1, 3\}

$$30110, 13010$$
If $L$ is a bubble language, then the following operation generates a circular left-shift Gray code where $s \in L$ and $k = |\mathbb{L}(s)|$.

$$\text{cool}_L(s) = \begin{cases} \text{greedy}(s, n) & \text{if } k = n \text{ or } k = n - 1 \\ \text{greedy}(s, k + 1) & \text{if } s_k < s_{k+2} \text{ or } \text{greedy}(s, k + 2) = s \\ \text{greedy}(s, k + 2) & \text{otherwise} \end{cases}$$

For example, ordered trees with branching sequence $\{0, 0, 0, 1, 1, 3\}$

$$30110, 13010, 31010,$$

$k = 1, 2$
Main Theorem

If $L$ is a bubble language, then the following operation generates a circular left-shift Gray code where $s \in L$ and $k = |\mathcal{L}(s)|$.

$$
\text{cool}_L(s) = \begin{cases} 
\text{greedy}(s, n) & \text{if } k = n \text{ or } k = n - 1 \\
\text{greedy}(s, k + 1) & \text{if } s_k < s_{k+2} \text{ or } \text{greedy}(s, k + 2) = s \\
\text{greedy}(s, k + 2) & \text{otherwise}
\end{cases}
$$

For example, ordered trees with branching sequence $\{0, 0, 0, 1, 1, 3\}$

$$
\begin{align*}
\text{301110, 130110, 310110, 301101,} \\
k_{12}
\end{align*}
$$
If $L$ is a bubble language, then the following operation generates a circular left-shift Gray code where $s \in L$ and $k = |\mathbb{L}(s)|$.

$$
\text{cool}_L(s) = \begin{cases} 
\text{greedy}(s, n) & \text{if } k = n \text{ or } k = n - 1 \\
\text{greedy}(s, k + 1) & \text{if } s_k < s_{k+2} \text{ or } \text{greedy}(s, k + 2) = s \\
\text{greedy}(s, k + 2) & \text{otherwise}
\end{cases}
$$

For example, ordered trees with branching sequence $\{0, 0, 0, 1, 1, 3\}$

$$
30110, 13010, 31010, 30101, 30011, \quad k = 1, 2
$$
If $L$ is a bubble language, then the following operation generates a circular left-shift Gray code where $s \in L$ and $k = |\mathbb{L}(s)|$.

$$\text{cool}_L(s) = \begin{cases} \text{greedy}(s, n) & \text{if } k = n \text{ or } k = n - 1 \\ \text{greedy}(s, k + 1) & \text{if } s_k < s_{k+2} \text{ or } \text{greedy}(s, k + 2) = s \\ \text{greedy}(s, k + 2) & \text{otherwise} \end{cases}$$

For example, ordered trees with branching sequence \{0, 0, 0, 1, 1, 3\}

\[
\begin{align*}
\text{30110}, & \quad \text{13010}, \quad \text{31010}, \quad \text{30101}, \quad \text{30011}, \\
\text{13001}, & \quad k = 1, 2
\end{align*}
\]
If \( L \) is a bubble language, then the following operation generates a circular left-shift Gray code where \( s \in L \) and \( k = |\mathbb{L}(s)| \).

\[
\text{cool}_L(s) = \begin{cases} 
\text{greedy}(s, n) & \text{if } k = n \text{ or } k = n - 1 \\
\text{greedy}(s, k + 1) & \text{if } s_k < s_{k+2} \text{ or } \text{greedy}(s, k + 2) = s \\
\text{greedy}(s, k + 2) & \text{otherwise}
\end{cases}
\]

For example, ordered trees with branching sequence \( \{0, 0, 0, 1, 1, 3\} \)

\[
\begin{align*}
\text{\underline{301110}}, & \quad \text{\underline{130110}}, \quad \text{\underline{310110}}, \quad \text{\underline{301011}}, \quad \text{\underline{300111}}, \\
\text{\underline{130011}}, & \quad \text{\underline{310011}},
\end{align*}
\]
Main Theorem

If $L$ is a bubble language, then the following operation generates a circular left-shift Gray code where $s \in L$ and $k = |\mathbb{L}(s)|$.

$$\text{cool}_L(s) = \begin{cases} 
\text{greedy}(s, n) & \text{if } k = n \text{ or } k = n - 1 \\
\text{greedy}(s, k + 1) & \text{if } s_k < s_{k+2} \text{ or } \text{greedy}(s, k + 2) = s \\
\text{greedy}(s, k + 2) & \text{otherwise}
\end{cases}$$

For example, ordered trees with branching sequence $\{0, 0, 0, 1, 1, 3\}$

$$\text{cool}_L(s) = \begin{cases} 
301110, 130110, 310110, 301011, 300111, \\
130011, 310011, 131000, k = 1, 2
\end{cases}$$
If \( L \) is a bubble language, then the following operation generates a circular left-shift Gray code where \( s \in L \) and \( k = |\mathbb{L}(s)| \).

\[
\text{cool}_L(s) = \begin{cases} 
\text{greedy}(s, n) & \text{if } k = n \text{ or } k = n - 1 \\
\text{greedy}(s, k + 1) & \text{if } s_k < s_{k+2} \text{ or } \text{greedy}(s, k + 2) = s \\
\text{greedy}(s, k + 2) & \text{otherwise}
\end{cases}
\]

For example, ordered trees with branching sequence \( \{0, 0, 0, 1, 1, 3\} \)

\[
\begin{align*}
\text{301110, 130110, 310110, 301101, 300111,} \\
\text{130011, 310011, 13100, 113000,} \\
\text{11300,}
\end{align*}
\]
If \( L \) is a bubble language, then the following operation generates a circular left-shift Gray code where \( s \in L \) and \( k = |\mathbb{L}(s)| \).

\[
\text{cool}_L(s) = \begin{cases} 
\text{greedy}(s, n) & \text{if } k = n \text{ or } k = n - 1 \\
\text{greedy}(s, k + 1) & \text{if } s_k < s_{k+2} \text{ or } \text{greedy}(s, k + 2) = s \\
\text{greedy}(s, k + 2) & \text{otherwise}
\end{cases}
\]

For example, ordered trees with branching sequence \( \{0, 0, 0, 1, 1, 3\} \)

\[
\begin{align*}
301110, & \quad 130110, \quad 310110, \quad 301101, \quad 300111, \\
130011, & \quad 310011, \quad 131000, \quad 113000, \quad 311000.
\end{align*}
\]
Main Theorem

If $L$ is a bubble language, then the following operation generates a circular left-shift Gray code where $s \in L$ and $k = |\mathbb{L}(s)|$.

$$
\text{cool}_L(s) = \begin{cases} 
\text{greedy}(s, n) & \text{if } k = n \text{ or } k = n - 1 \\
\text{greedy}(s, k + 1) & \text{if } s_k < s_{k+2} \text{ or } \text{greedy}(s, k + 2) = s \\
\text{greedy}(s, k + 2) & \text{otherwise}
\end{cases}
$$

For example, ordered trees with branching sequence $\{0, 0, 0, 1, 1, 3\}$

$$
\begin{align*}
\text{co} & \text{ol}_L(s) = \text{greedy}(s, n) \\
\text{co} & \text{ol}_L(s) = \text{greedy}(s, k + 1) \\
\text{co} & \text{ol}_L(s) = \text{greedy}(s, k + 2)
\end{align*}
$$

Furthermore, the strings are generated in cool-lex order.
Shorthand Universal Cycles
A circular string that contains each string in language $L$ exactly once as a substring is a *universal cycle* for $L$. 

Consider alternate representations for the strings in $L$. 

**Universal Cycles**
A circular string that contains each string in language $L$ exactly once as a substring is a *universal cycle* for $L$.

Construct a universal cycle for the permutations of $\{1, 2, 2, 3\}$

It must have length $\frac{4!}{2!} = 12$. 
A circular string that contains each string in language $L$ exactly once as a substring is a *universal cycle* for $L$.

Construct a universal cycle for the permutations of \{1, 2, 2, 3\}

It must contain 1223 as a substring in clockwise order.
A circular string that contains each string in language $L$ exactly once as a substring is a \textit{universal cycle} for $L$.

Construct a universal cycle for the permutations of $\{1, 2, 2, 3\}$.

The symbol following $223$ must be $1$. 
Universal Cycles

A circular string that contains each string in language $L$ exactly once as a substring is a *universal cycle* for $L$.

Construct a universal cycle for the permutations of $\{1, 2, 2, 3\}$

The symbol following $231$ must be $2$. 
A circular string that contains each string in language $L$ exactly once as a substring is a *universal cycle* for $L$.

Construct a universal cycle for the permutations of $\{1, 2, 2, 3\}$

The symbol following $312$ must be $2$. 

Universal Cycles

A circular string that contains each string in language $L$ exactly once as a substring is a *universal cycle* for $L$.

Construct a universal cycle for the permutations of $\{1, 2, 2, 3\}$

The symbol following $122$ must be $3$. 
A circular string that contains each string in language $L$ exactly once as a substring is a *universal cycle* for $L$.

Construct a universal cycle for the permutations of $\{1, 2, 2, 3\}$

The first string $1223$ is repeated.
A circular string that contains each string in language $L$ exactly once as a substring is a *universal cycle* for $L$.

Construct a universal cycle for the permutations of $\{1, 2, 2, 3\}$

Universal cycles for the permutations of $\{1, 2, 2, 3\}$ do not exist.
A circular string that contains each string in language $L$ exactly once as a substring is a *universal cycle* for $L$.

Construct a universal cycle for the permutations of \{1, 2, 2, 3\}.

Universal cycles for the permutations of \{1, 2, 2, 3\} do not exist.

Consider alternate representations for the strings in $L$. 

\begin{center}
\begin{tikzpicture}
  \foreach \x in {1,2,3,4}
  \node[draw, circle] (\x) at (90-\x*360/4:1.5cm) {\x};
  \foreach \x in {1,2,3,4}
  \foreach \y in {1,2,3,4}
  \draw[thick] (\x) to (\y);
\end{tikzpicture}
\end{center}
The last symbol of every string in a fixed-content language is redundant.
The last symbol of every string in a fixed-content language is redundant. The *shorthand representation* removes this redundant symbol.
The last symbol of every string in a fixed-content language is redundant. The *shorthand representation* removes this redundant symbol.

A shorthand universal cycle for the permutations of \{1, 2, 2, 3\}

For example, \(132\) is shorthand for \(1322\).
The last symbol of every string in a fixed-content language is redundant. The *shorthand representation* removes this redundant symbol.

A shorthand universal cycle for the permutations of \{1, 2, 2, 3\}

All twelve permutations appear in shorthand.
The last symbol of every string in a fixed-content language is redundant. The *shorthand representation* removes this redundant symbol.

A shorthand universal cycle for the permutations of \{1, 2, 2, 3\}.

Shorthand universal cycles also provide shift Gray codes.
The last symbol of every string in a fixed-content language is redundant. The *shorthand representation* removes this redundant symbol.

A shorthand universal cycle for the permutations of \{1, 2, 2, 3\}

Shorthand universal cycles also provide shift Gray codes.

\[ 3211, 2213, 2132, 1322, 3212, 2123, 1232, 2312, 3122, 1223, 2231, 2321. \]
The last symbol of every string in a fixed-content language is redundant. The *shorthand representation* removes this redundant symbol.

A shorthand universal cycle for the permutations of \{1, 2, 2, 3\}

[Diagram of a cycle with numbers 1, 2, 3, and arrows showing the cycle]

Shorthand universal cycles also provide shift Gray codes.

\[\text{3211, 2213, 2132, 1322, 3212, 2123, 1232, 2312, 3122, 1223, 2231, 2321}\]

The string following \(\mathbf{s}\) must be \(\text{shift}(\mathbf{s}, 1, n)\) or \(\text{shift}(\mathbf{s}, 1, n - 1)\).
The following construction works for the permutations of any multiset.
<table>
<thead>
<tr>
<th>combinations</th>
<th>necklaces</th>
<th>prime</th>
<th>shorthand</th>
<th>combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Start with reverse cool-lex order for the multiset permutations.
# Shorthand Universal Cycles for Multiset Permutations

<table>
<thead>
<tr>
<th>combinations</th>
<th>necklaces</th>
<th>prime</th>
<th>shorthand</th>
<th>combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000011</td>
<td>00000011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001111</td>
<td>00001111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101111</td>
<td>00101111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001111</td>
<td>01001111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000111</td>
<td>01000111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010111</td>
<td>01010111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010011</td>
<td>01010011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100111</td>
<td>01100111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10100111</td>
<td>10100111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101011</td>
<td>10101011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101001</td>
<td>10101001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000110</td>
<td>11000110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11001110</td>
<td>11001110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010110</td>
<td>11010110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11011010</td>
<td>11011010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11011000</td>
<td>11011000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101000</td>
<td>11101000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111000</td>
<td>11111000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. Remove non-necklaces to get the cool-lex order for necklaces.
3. Take the prefix that is prime (non-repeating) of each necklace.
<table>
<thead>
<tr>
<th>combinations</th>
<th>necklaces</th>
<th>prime</th>
<th>shorthand</th>
<th>combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110000</td>
<td>11110000</td>
<td>11110000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11100001</td>
<td></td>
<td>11100001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1000111</td>
<td></td>
<td>1110000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0001111</td>
<td></td>
<td>1110000</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0010111</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0100111</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1001011</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0011011</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0110101</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0101001</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1010011</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0111001</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1100110</td>
<td>1100110</td>
<td>1100110</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1001110</td>
<td></td>
<td>1100110</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0011110</td>
<td></td>
<td>1100110</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0101110</td>
<td></td>
<td>1100110</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1011010</td>
<td>1011010</td>
<td>1011010</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0110110</td>
<td>1101100</td>
<td>1101100</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1101000</td>
<td>1101000</td>
<td>1101000</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1011000</td>
<td>1101000</td>
<td>1101000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0111000</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

4. Concatenate the results to obtain the shorthand universal cycle.
## Shorthand Universal Cycles for Multiset Permutations

<table>
<thead>
<tr>
<th>combinations</th>
<th>necklaces</th>
<th>prime</th>
<th>shorthand</th>
<th>combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110000</td>
<td>1110000</td>
<td>1110000</td>
<td>1</td>
<td>1110000</td>
</tr>
<tr>
<td>1100001</td>
<td>1110000</td>
<td>1110000</td>
<td>1</td>
<td>1110000</td>
</tr>
<tr>
<td>1000011</td>
<td></td>
<td>1110000</td>
<td>1</td>
<td>1110000</td>
</tr>
<tr>
<td>0001111</td>
<td></td>
<td>1110000</td>
<td>0</td>
<td>1110000</td>
</tr>
<tr>
<td>0010111</td>
<td></td>
<td></td>
<td>0</td>
<td>1110000</td>
</tr>
<tr>
<td>0100111</td>
<td></td>
<td></td>
<td>0</td>
<td>1110000</td>
</tr>
<tr>
<td>1001011</td>
<td></td>
<td></td>
<td>0</td>
<td>1110000</td>
</tr>
<tr>
<td>0011011</td>
<td></td>
<td></td>
<td>1</td>
<td>1110000</td>
</tr>
<tr>
<td>0101011</td>
<td></td>
<td></td>
<td>0</td>
<td>1110000</td>
</tr>
<tr>
<td>1010011</td>
<td></td>
<td></td>
<td>0</td>
<td>1110000</td>
</tr>
<tr>
<td>0110001</td>
<td></td>
<td></td>
<td>1</td>
<td>1110000</td>
</tr>
<tr>
<td>1100010</td>
<td>1100010</td>
<td>1100010</td>
<td>0</td>
<td>1100010</td>
</tr>
<tr>
<td>1000100</td>
<td></td>
<td>1100010</td>
<td>1</td>
<td>1100010</td>
</tr>
<tr>
<td>0010110</td>
<td></td>
<td>1100010</td>
<td>0</td>
<td>1100010</td>
</tr>
<tr>
<td>0101010</td>
<td>1101010</td>
<td></td>
<td>1</td>
<td>1101010</td>
</tr>
<tr>
<td>1010010</td>
<td></td>
<td>1101010</td>
<td>0</td>
<td>1101010</td>
</tr>
<tr>
<td>0110010</td>
<td></td>
<td>1101010</td>
<td>0</td>
<td>1101010</td>
</tr>
<tr>
<td>1101001</td>
<td>1101001</td>
<td>1101001</td>
<td>1</td>
<td>1101001</td>
</tr>
<tr>
<td>1011000</td>
<td></td>
<td>1101001</td>
<td>0</td>
<td>1101001</td>
</tr>
<tr>
<td>0111000</td>
<td></td>
<td>1101001</td>
<td>0</td>
<td>1101001</td>
</tr>
</tbody>
</table>

(It has the correct length by considering the rotations of any string.)
<table>
<thead>
<tr>
<th>combinations</th>
<th>necklaces</th>
<th>prime</th>
<th>shorthand</th>
<th>combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>⬤⬤⬤⬤⬤⬤⬤</td>
<td>⬤⬤⬤⬤⬤⬤</td>
<td>⬤⬤⬤⬤⬤⬤</td>
<td>⬤</td>
<td>⬤⬤⬤⬤⬤➋</td>
</tr>
<tr>
<td>⬤⬤⬤⬤⬤⬤➋</td>
<td>⬤⬤⬤⬤⬤⬤</td>
<td>⬤⬤⬤⬤⬤⬤</td>
<td>⬤</td>
<td>⬤⬤⬤⬤⬤➋</td>
</tr>
<tr>
<td>⬤⬤⬤⬤➋⬤➋</td>
<td>⬤⬤⬤⬤➋⬤➋</td>
<td>⬤⬤⬤⬤➋⬤➋</td>
<td>⬤</td>
<td>⬤⬤⬤➋➋⬤➋</td>
</tr>
<tr>
<td>⬤⬤➋➋➋➋➋➋</td>
<td>⬤⬤➋➋➋➋➋➋</td>
<td>⬤⬤➋➋➋➋➋➋</td>
<td>⬤</td>
<td>⬤➋➋➋➋➋➋</td>
</tr>
<tr>
<td>⬤➋➋➋➋➋➋➋</td>
<td>⬤➋➋➋➋➋➋➋</td>
<td>⬤➋➋➋➋➋➋➋</td>
<td>⬤</td>
<td>⬤➋➋➋➋➋➋</td>
</tr>
<tr>
<td>⬤➋➋➋➋➋➋➋</td>
<td>⬤➋➋➋➋➋➋➋</td>
<td>⬤➋➋➋➋➋➋➋</td>
<td>⬤</td>
<td>⬤➋➋➋➋➋➋</td>
</tr>
<tr>
<td>⬤➋➋➋➋➋➋➋</td>
<td>⬤➋➋➋➋➋➋➋</td>
<td>⬤➋➋➋➋➋➋➋</td>
<td>⬤</td>
<td>⬤➋➋➋➋➋➋</td>
</tr>
<tr>
<td>⬤➋➋➋➋➋➋➋</td>
<td>⬤➋➋➋➋➋ⷥⷥ</td>
<td>⬤➋ⷥⷥⷥⷥⷥ</td>
<td>⬤</td>
<td>⬤ⷥⷥⷥⷥⷥ</td>
</tr>
<tr>
<td>⬤ⷥⷥⷥⷥⷥⷥ</td>
<td>⬤ⷥⷥⷥⷥⷥⷥ</td>
<td>⬤ⷥⷥⷥⷥⷥⷥ</td>
<td>⬤</td>
<td>⬤ⷥⷥⷥⷥⷥ</td>
</tr>
<tr>
<td>⬤ⷥⷥⷥⷥⷥⷥ</td>
<td>⬤ⷥⷥⷥⷥⷥⷥ</td>
<td>⬤ⷥⷥⷥⷥⷥⷥ</td>
<td>⬤</td>
<td>⬤ⷥⷥⷥⷥⷥ</td>
</tr>
</tbody>
</table>

5. Substrings are shorthand for the original permutations.
<table>
<thead>
<tr>
<th>combinations</th>
<th>necklaces</th>
<th>prime</th>
<th>shorthand</th>
<th>combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110000</td>
<td>1110000</td>
<td>1110000</td>
<td>1110000</td>
<td>1110000</td>
</tr>
<tr>
<td>1100011</td>
<td></td>
<td></td>
<td></td>
<td>1100011</td>
</tr>
<tr>
<td>1000111</td>
<td></td>
<td></td>
<td></td>
<td>1000111</td>
</tr>
<tr>
<td>0001111</td>
<td></td>
<td></td>
<td>0001111</td>
<td>0001111</td>
</tr>
<tr>
<td>0010111</td>
<td></td>
<td></td>
<td>0011100</td>
<td>0011101</td>
</tr>
<tr>
<td>0100111</td>
<td></td>
<td></td>
<td>0110001</td>
<td>0110011</td>
</tr>
<tr>
<td>1001011</td>
<td></td>
<td></td>
<td>1100100</td>
<td>1100100</td>
</tr>
<tr>
<td>0011011</td>
<td></td>
<td></td>
<td>1001000</td>
<td>1001001</td>
</tr>
<tr>
<td>0101011</td>
<td></td>
<td></td>
<td>0010101</td>
<td>0010111</td>
</tr>
<tr>
<td>1010011</td>
<td></td>
<td></td>
<td>0101010</td>
<td>0101011</td>
</tr>
<tr>
<td>0110011</td>
<td></td>
<td></td>
<td>1010100</td>
<td>1010101</td>
</tr>
<tr>
<td>1100110</td>
<td>1100110</td>
<td>1100110</td>
<td>0101111</td>
<td>0101101</td>
</tr>
<tr>
<td>1001110</td>
<td></td>
<td></td>
<td>1011000</td>
<td>1011001</td>
</tr>
<tr>
<td>0011110</td>
<td></td>
<td></td>
<td>0110100</td>
<td>0110101</td>
</tr>
<tr>
<td>0101110</td>
<td></td>
<td></td>
<td>1101000</td>
<td>1101001</td>
</tr>
<tr>
<td>1011010</td>
<td>1011010</td>
<td>1011010</td>
<td>0100111</td>
<td>0100011</td>
</tr>
<tr>
<td>0111010</td>
<td></td>
<td></td>
<td>1100110</td>
<td>1100110</td>
</tr>
<tr>
<td>1101000</td>
<td>1101000</td>
<td>1101000</td>
<td>1100110</td>
<td>1100110</td>
</tr>
<tr>
<td>1011000</td>
<td></td>
<td></td>
<td>0111010</td>
<td>0111010</td>
</tr>
<tr>
<td>0111000</td>
<td></td>
<td></td>
<td>0111000</td>
<td>0111000</td>
</tr>
</tbody>
</table>

6. Suffix the unique missing symbol to obtain the permutations.
Therefore, cool-lex is a fixed-content analogue of lex order in FKM algorithm.
## Shorthand Universal Cycles for Multiset Permutations

<table>
<thead>
<tr>
<th>combinations</th>
<th>necklaces</th>
<th>prime</th>
<th>·</th>
<th>shorthand</th>
<th>combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110000</td>
<td>1110000</td>
<td>1110000</td>
<td>1</td>
<td>1110000</td>
<td>1110000</td>
</tr>
<tr>
<td>1100011</td>
<td>1100011</td>
<td>1100011</td>
<td>1</td>
<td>1100011</td>
<td>1100011</td>
</tr>
<tr>
<td>1000111</td>
<td>1000111</td>
<td>1000111</td>
<td>1</td>
<td>1000111</td>
<td>1000111</td>
</tr>
<tr>
<td>0001111</td>
<td>0001111</td>
<td>0001111</td>
<td>0</td>
<td>0001111</td>
<td>0001111</td>
</tr>
<tr>
<td>0010111</td>
<td>0010111</td>
<td>0010111</td>
<td>0</td>
<td>0010111</td>
<td>0010111</td>
</tr>
<tr>
<td>0100111</td>
<td>0100111</td>
<td>0100111</td>
<td>0</td>
<td>0100111</td>
<td>0100111</td>
</tr>
<tr>
<td>1010101</td>
<td>1010101</td>
<td>1010101</td>
<td>1</td>
<td>1010101</td>
<td>1010101</td>
</tr>
<tr>
<td>1010001</td>
<td>1010001</td>
<td>1010001</td>
<td>1</td>
<td>1010001</td>
<td>1010001</td>
</tr>
<tr>
<td>0110011</td>
<td>0110011</td>
<td>0110011</td>
<td>1</td>
<td>0110011</td>
<td>0110011</td>
</tr>
<tr>
<td>1100010</td>
<td>1100010</td>
<td>1100010</td>
<td>0</td>
<td>010111</td>
<td>010111</td>
</tr>
<tr>
<td>1001100</td>
<td>1001100</td>
<td>1001100</td>
<td>1</td>
<td>1001100</td>
<td>1001100</td>
</tr>
<tr>
<td>0011100</td>
<td>0011100</td>
<td>0011100</td>
<td>0</td>
<td>001110</td>
<td>001110</td>
</tr>
<tr>
<td>0101100</td>
<td>0101100</td>
<td>0101100</td>
<td>0</td>
<td>0101100</td>
<td>0101100</td>
</tr>
<tr>
<td>1101010</td>
<td>1101010</td>
<td>1101010</td>
<td>1</td>
<td>1101010</td>
<td>1101010</td>
</tr>
<tr>
<td>0111010</td>
<td>0111010</td>
<td>0111010</td>
<td>0</td>
<td>0111010</td>
<td>0111010</td>
</tr>
<tr>
<td>1101000</td>
<td>1101000</td>
<td>1101000</td>
<td>1</td>
<td>100111</td>
<td>100111</td>
</tr>
<tr>
<td>1011000</td>
<td>1011000</td>
<td>1011000</td>
<td>0</td>
<td>001110</td>
<td>001110</td>
</tr>
<tr>
<td>0111000</td>
<td>0111000</td>
<td>0111000</td>
<td>0</td>
<td>011100</td>
<td>011100</td>
</tr>
</tbody>
</table>

Note: Successive permutations differ by \( \text{shift}(s, 1, n) \) or \( \text{shift}(s, 1, n - 1) \).
<table>
<thead>
<tr>
<th>combinations</th>
<th>necklaces</th>
<th>prime</th>
<th>shorthand</th>
<th>combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110000</td>
<td>11110000</td>
<td>1110000</td>
<td>1110000</td>
<td>1110000</td>
</tr>
<tr>
<td>1100001</td>
<td>1100001</td>
<td>1100001</td>
<td>1100001</td>
<td>1100001</td>
</tr>
<tr>
<td>1000111</td>
<td>1000111</td>
<td>1000111</td>
<td>1000111</td>
<td>1000111</td>
</tr>
<tr>
<td>0001111</td>
<td>0001111</td>
<td>0001111</td>
<td>0001111</td>
<td>0001111</td>
</tr>
<tr>
<td>0010111</td>
<td>0010111</td>
<td>0010111</td>
<td>0010111</td>
<td>0010111</td>
</tr>
<tr>
<td>0100111</td>
<td>0100111</td>
<td>0100111</td>
<td>0100111</td>
<td>0100111</td>
</tr>
<tr>
<td>1001011</td>
<td>1001011</td>
<td>1001011</td>
<td>1001011</td>
<td>1001011</td>
</tr>
<tr>
<td>0011011</td>
<td>0011011</td>
<td>0011011</td>
<td>0011011</td>
<td>0011011</td>
</tr>
<tr>
<td>0101011</td>
<td>0101011</td>
<td>0101011</td>
<td>0101011</td>
<td>0101011</td>
</tr>
<tr>
<td>1010001</td>
<td>1010001</td>
<td>1010001</td>
<td>1010001</td>
<td>1010001</td>
</tr>
<tr>
<td>0110011</td>
<td>0110011</td>
<td>0110011</td>
<td>0110011</td>
<td>0110011</td>
</tr>
<tr>
<td>1100010</td>
<td>1100010</td>
<td>1100010</td>
<td>1100010</td>
<td>1100010</td>
</tr>
<tr>
<td>1000110</td>
<td>1000110</td>
<td>1000110</td>
<td>1000110</td>
<td>1000110</td>
</tr>
<tr>
<td>0011100</td>
<td>0011100</td>
<td>0011100</td>
<td>0011100</td>
<td>0011100</td>
</tr>
<tr>
<td>0101010</td>
<td>0101010</td>
<td>0101010</td>
<td>0101010</td>
<td>0101010</td>
</tr>
<tr>
<td>1010010</td>
<td>1010010</td>
<td>1010010</td>
<td>1010010</td>
<td>1010010</td>
</tr>
<tr>
<td>0110010</td>
<td>0110010</td>
<td>0110010</td>
<td>0110010</td>
<td>0110010</td>
</tr>
<tr>
<td>1101000</td>
<td>1101000</td>
<td>1101000</td>
<td>1101000</td>
<td>1101000</td>
</tr>
<tr>
<td>1011000</td>
<td>1011000</td>
<td>1011000</td>
<td>1011000</td>
<td>1011000</td>
</tr>
<tr>
<td>0111000</td>
<td>0111000</td>
<td>0111000</td>
<td>0111000</td>
<td>0111000</td>
</tr>
</tbody>
</table>

Note: For \((t, t)\)-combinations the result is a universal cycle for the middle levels.
Figure: Universal cycle for middle levels (5, 4)- and (4, 5)-combinations.
Figure: Shorthand universal cycle for (5, 5)-combinations.
Summary

- New order for fixed-content languages called *cool-lex*.
Summary

- New order for fixed-content languages called *cool-lex*.
- New class of fixed-content languages called *bubble languages*. 

Thank You!
New order for fixed-content languages called *cool-lex*.

New class of fixed-content languages called *bubble languages*.

Shift Gray codes for bubble languages in *cool-lex* order.

Introduced shorthand universal cycles.

Shorthand universal cycles for multiset permutations from *cool-lex* order.
New order for fixed-content languages called *cool-lex*.

New class of fixed-content languages called *bubble languages*.

Shift Gray codes for bubble languages in cool-lex order.

Efficient algorithms for specific bubble languages in cool-lex order.
Summary

- New order for fixed-content languages called *cool-lex*.
- New class of fixed-content languages called *bubble languages*.
- Shift Gray codes for bubble languages in cool-lex order.
- Efficient algorithms for specific bubble languages in cool-lex order.
- Introduced shorthand universal cycles.
Summary

- New order for fixed-content languages called *cool-lex*.
- New class of fixed-content languages called *bubble languages*.
- Shift Gray codes for bubble languages in cool-lex order.
- Efficient algorithms for specific bubble languages in cool-lex order.
- Introduced shorthand universal cycles.
- Shorthand universal cycles for multiset permutations from cool-lex order.
Summary

- New order for fixed-content languages called *cool-lex*.
- New class of fixed-content languages called *bubble languages*.
- Shift Gray codes for bubble languages in cool-lex order.
- Efficient algorithms for specific bubble languages in cool-lex order.
- Introduced shorthand universal cycles.
- Shorthand universal cycles for multiset permutations from cool-lex order.

Thank You!