Research Proposal

Greedy and Speedy:
New Iterative Gray Code Algorithms

Yuan (Friedrich) Qiu
Williams College

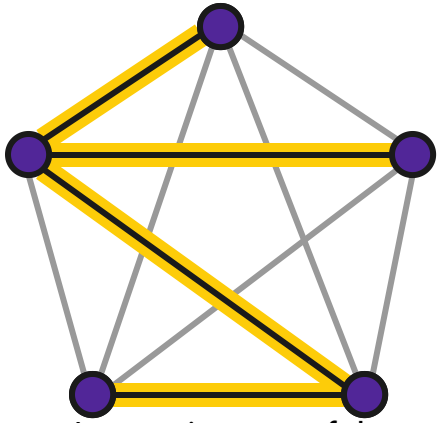
Combinatorial Generation

Combinatorial Objects

In computer science, we often work with different types of *combinatorial objects*.
In many applications, we need to find an **optimal object** of a particular type and/or size according to some metric (e.g., a minimum weight spanning tree of a graph).

Binary string with $n = 5$ bits.
It can model a knapsack solution for $n = 5$.

A permutation of $[n] = \{1,2, ..., n\}$ for $n = 5$.
It can model a TSP solution for $n = 5$.



A spanning tree of the complete graph K_n for $n = 5$.

The number of objects of a given type is often exponential with respect to the size.

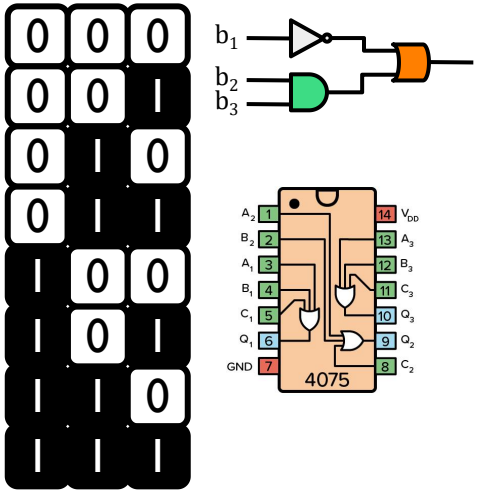
- There are 2^n binary strings with n bits.
- There are $n!$ permutations of $[n]$.
- There are n^{n-2} spanning trees of K_n.

Therefore, we usually don't want to generate every object when finding an optimal one.

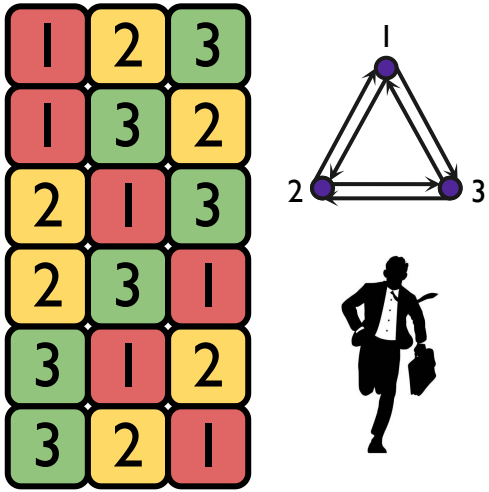
Combinatorial Generation

In some applications, it is necessary or beneficial to consider **every object** of a given type and size.

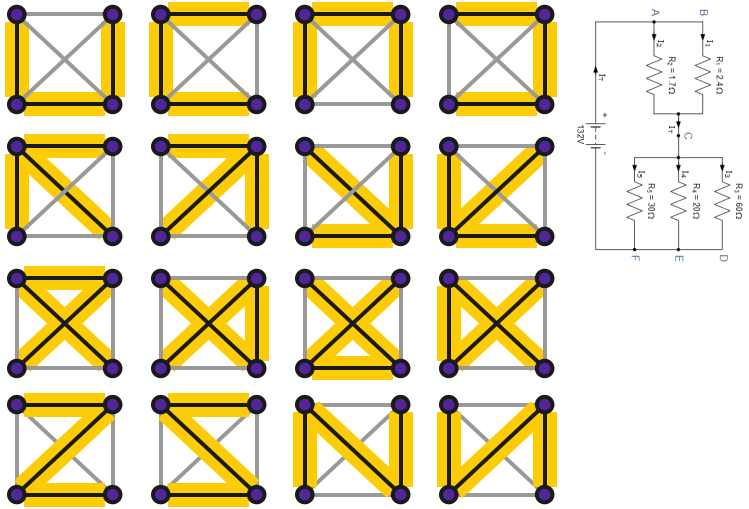
- *Testing*. Binary strings with n bits represent all possible states for a circuit or chip with n inputs.
- *Exact algorithms*. Permutations of $[n]$ represent directed paths in a traveling salesman problem.
- *Calculations*. Some electrical engineering formulae sum over all spanning trees of a circuit.



All binary strings with $n = 3$.



All permutations with $n = 3$.



All spanning trees of K_n with $n = 4$.

Combinatorial generation refers to the efficient creation of every object in computer memory.
Since there are exponentially many objects, they should be created one at a time, not all at once.
Time complexity: How much time to create the next object (e.g., amortized $O(n)$-time per object)?

Programming Language Support

Many contemporary programming languages have *iterators* that streamline the process of providing one object at a time, and *libraries* of iterators for different combinatorial objects.

```
from itertools import permutations

for perm in permutations([1,2,3,4]):
  # do something with permutation
  print(perm)

>> (1,2,3,4)
>> (1,2,4,3)
>> …
>> (4,3,2,1)
```

tuples (non-mutable)

Generating permutations with Python's `itertools` library.

```
from itertools import permutations

for perm in permutations([1,2,3,3]):
  # do something with permutation
  print(perm)

>> (1,2,3,3)
>> (1,2,3,3)
>> …
>> (3,3,2,1)
```

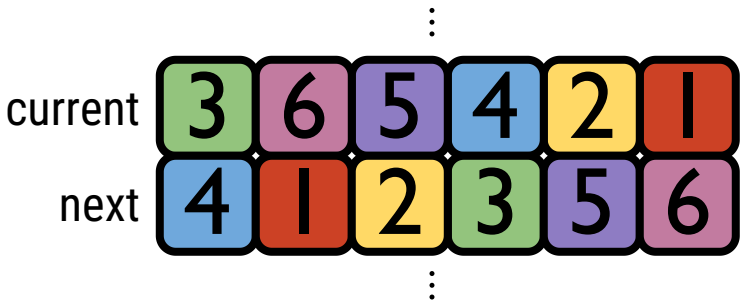
Standard Python libraries do not generate multiset permutations

Unfortunately, this support is limited and flawed in many fundamental ways.

- Few combinatorial objects are available (e.g., permutations but not multiset permutations).
- Every object is returned as a separate entity. Thus $\Omega(n)$-time (i.e. at least $O(n)$-time) is used. This is wasteful since applications typically *use* each object and don't need to *store* all of them.
- Objects are generated in *lexicographic order*. This "normal" order is familiar but not efficient.

Standard Approach

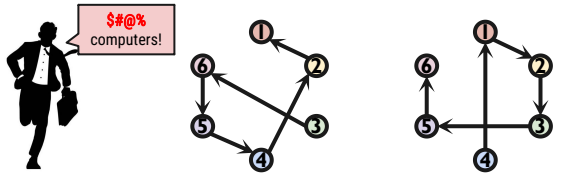
Lexicographic order



Successive objects can differ **everywhere** i.e., θ **(n) changes**

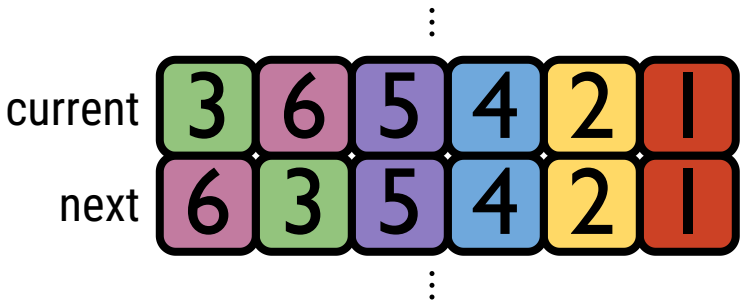
Iterators use **Ω(n)-time** to create the next **new object**

Applications use **Ω(n)-time** just to read the new next object



Better Approach

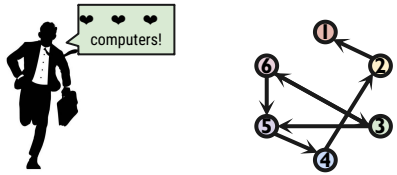
A *Gray code* is a minimal change order



Successive objects differ by a **constant** amount, i.e., θ **(1) change**

Iterators use **θ(1)-time** to create the **next change**

Applications use **θ(1)-time** to process the change

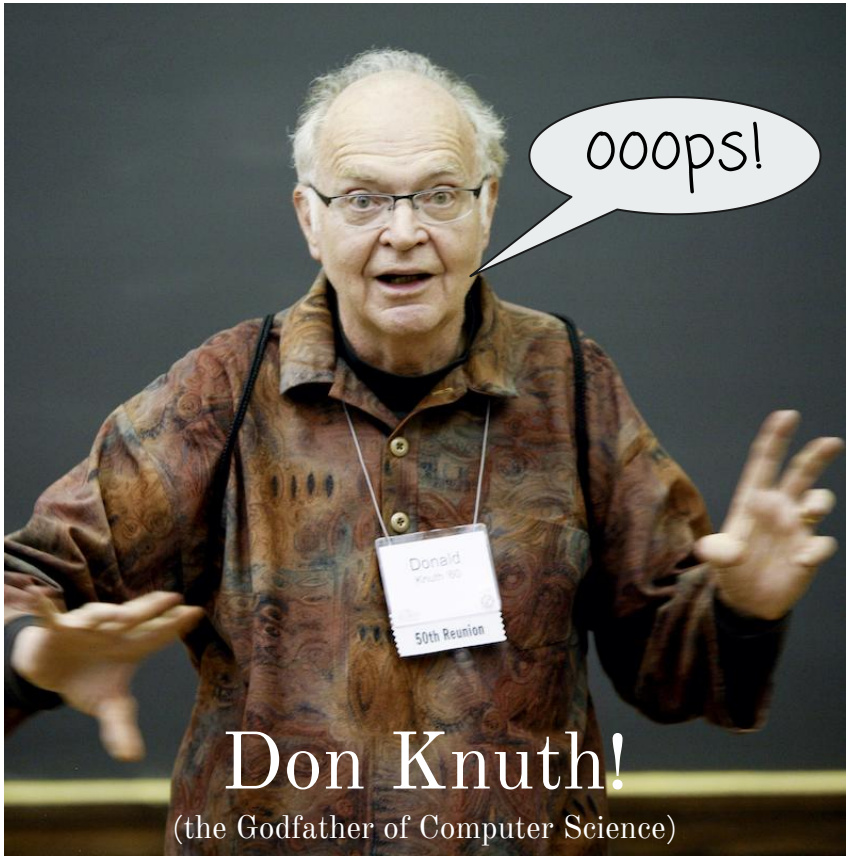


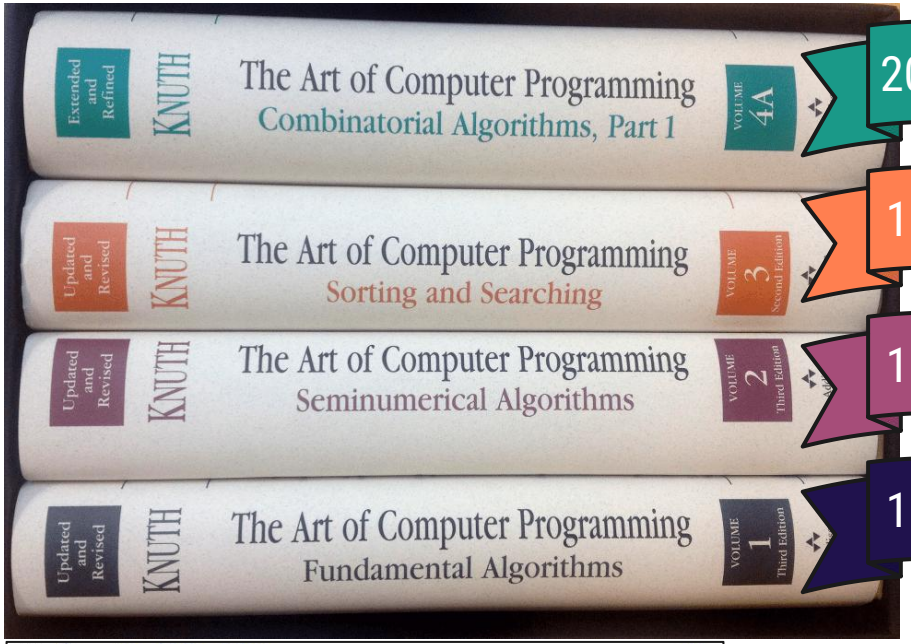
The standard approach is unacceptable for such a common computational task.
The situation is worse for less common objects (e.g., multiset permutations).

Programmers ask the same questions year after year for objects that are not in standard libraries.

- This is repeated for every combinatorial object and every programming language.

Who is to blame?





7.2. Generating All Possibilities 281
7.2.1. Generating Basic Combinatorial Patterns 281
7.2.1.1. Generating all n-tuples 281
7.2.1.2. Generating all permutations 319
7.2.1.3. Generating all combinations 355
7.2.1.4. Generating all partitions 390
7.2.1.5. Generating all set partitions 415
7.2.1.6. Generating all trees 440
7.2.1.7. History and further references 486

200+ pages on combinatorial generation in Volume 4

The most influential Computer Science textbook has a nearly 40 year gap between topics.

- Sorting and searching (1973) is taught in every algorithms course.
- Combinatorial generation (2011) is taught in almost no algorithms courses.

Thesis Goals: Greedy and Speedy

General Goals

Promote a modern approach to combinatorial generation.

- *Greedy*. One sentence descriptions of Gray code orders.
- *Speedy*. Loopless implementations (i.e., worst-case $O(1)$-time per object) that are simple enough to be implemented in any programming language.

Specific Goals

Focus on a combinatorial object that does not yet have a well-known generation algorithm.

- A *signed permutation* is a permutation in which each symbol is given a $\pm$ sign.

Stretch Goals

Solve one of Knuth's open problems involving combinatorial generation.

A simple edge-exchange algorithm for generating the spanning trees of the complete graph K_n.

millions of downloads

Loopless Generation of Multiset Permutations using a Constant Number of Variables by Prefix Shifts

Aaron Williams *

```
[h, i, j] ← init(𝔼)
visit(h)
while j.n ≠ φ or j.v < h.v do
  if j.n ≠ φ and i.v ≥ j.n.v then
    s ← j
  else
    s ← i
  end if
  t ← s.n
  s.n ← t.n
  t.n ← h
  if t.v < h.v then
    i ← t
  end if
  j ← i.n
  h ← t
  visit(h)
end while
```

ACM-SIAM Symposium on Discrete Algorithms

Research paper on multiset permutations.

Package 'multicool'

February 5, 2024

Type Package

Title Permutations of Multisets in Cool-Lex Order

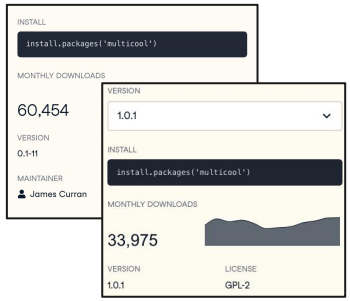
Version 1.0.1

Date 2024-02-05

Author James Curran, Aaron Williams, Jerome Kelleher, Dave Barber

Maintainer James Curran <j.curran@auckland.ac.nz>

Description A set of tools to permute multisets without loops or hash tables and to generate integer partitions. The permutation functions are based on C code from Aaron Williams. Cool-lex order is similar to colexicographical order. The algorithm is described in Williams, A. Loopless Generation of Multiset Permutations by Prefix Shifts. SODA 2009, Symposium on Discrete Algorithms, New York, United States. The permutation code is distributed without restrictions. The code for stable and efficient computation of multinomial coefficients comes from Dave Barber. The code can be download from <http://tamivox.org/dave/multinomial/index.html> and is distributed without conditions. The package also generates the integer partitions of a positive, non-zero integer n. The C++ code for this is based on Python code from Jerome Kelle-



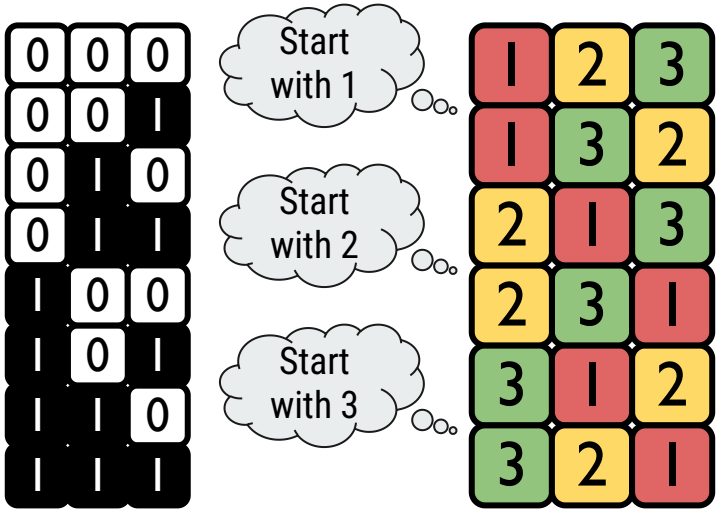
Implementation in R.

Example of a combinatorial generation algorithm that is used in practice.

Ruler Sequences

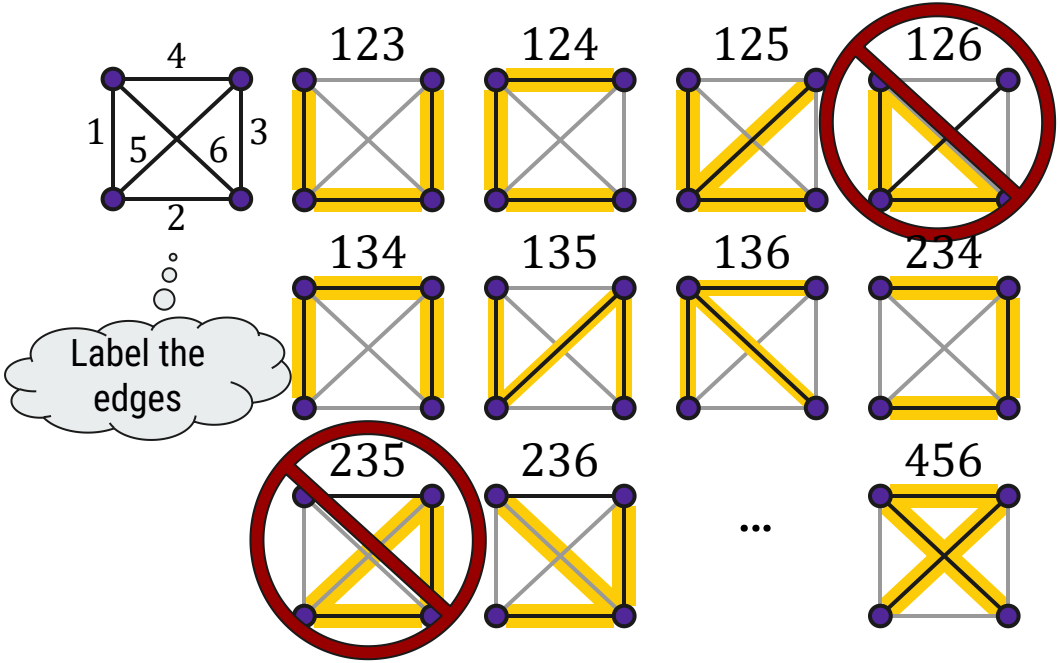
Lexicographic Order

Combinatorial objects can always be encoded as strings of symbols (e.g., binary representation).
So they can be put in *lexicographic order*, which generalizes counting and alphabetical order.

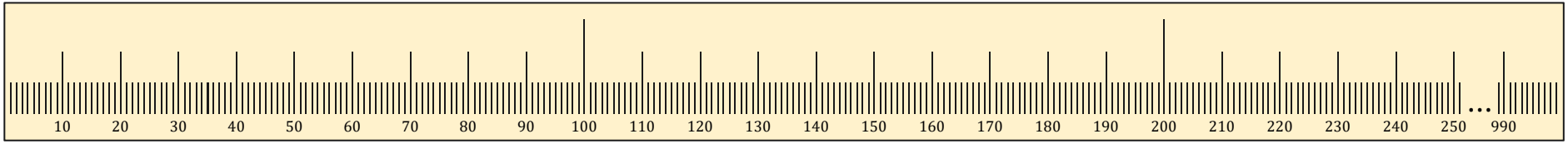


Binary strings $n = 3$.

Permutations $n = 3$.



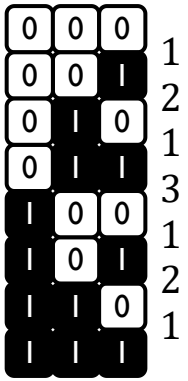
The issue with lexicographic order is that successive objects can be completely different.
A *Gray code* is an order in which successive objects differ in a constant amount by some metric.

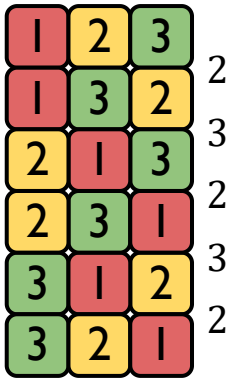


A ruler's tick marks indicate how many digits change when counting in decimal. For example, the tick mark at 200 has height 3 since that number of digits change from 199 to 200. The sequence of heights is the *decimal ruler sequence*.
A *meter stick* has numbers from 0 to 999 (i.e., three base-10 digits).
$\text{ruler}(10,10,10) = 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, \ldots, 1$

A binary ruler's ticks follow the *binary ruler sequence* and counting in binary.
$\text{ruler}(2, 2, 2) = 1, 2, 1, 3, 1, 2, 1$

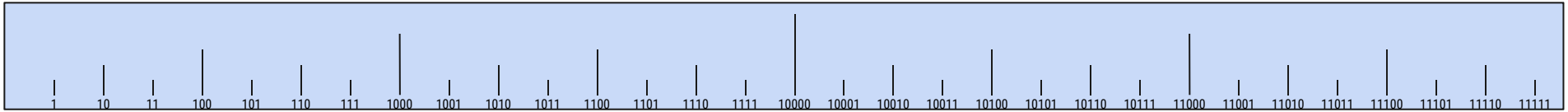
For permutations there is a *factorial ruler sequence*.
$\text{ruler}(3, 2, 1) = 2, 3, 2, 3, 2$
It gives the number of changed symbols in the lexicographic order of permutations.





The ruler sequence with bases $\mathbf{b} = b_n, b_{n-1}, \ldots, b_1$ has length $(b_n \cdot b_{n-1} \cdots b_1)-1$.
$\text{ruler}(b_1) = 1, 1, \ldots, 1$
$\text{ruler}(\mathbf{b}) = \text{ruler}(\mathbf{b'}), n, \text{ruler}(\mathbf{b'}), n, \ldots, n, \text{ruler}(\mathbf{b'})$ where $\mathbf{b'} = b_{n-1}, b_{n-2}, \ldots, b_1$

These sequence are easy to generate efficiently: worst-case $O(1)$-time per entry.
They provide *change sequences* for lexicographic order and for various types of Gray codes.



Gray Codes

Minimal-Change Orders

Binary Reflected Gray Code

for binary strings

Binary Reflected Gray Code

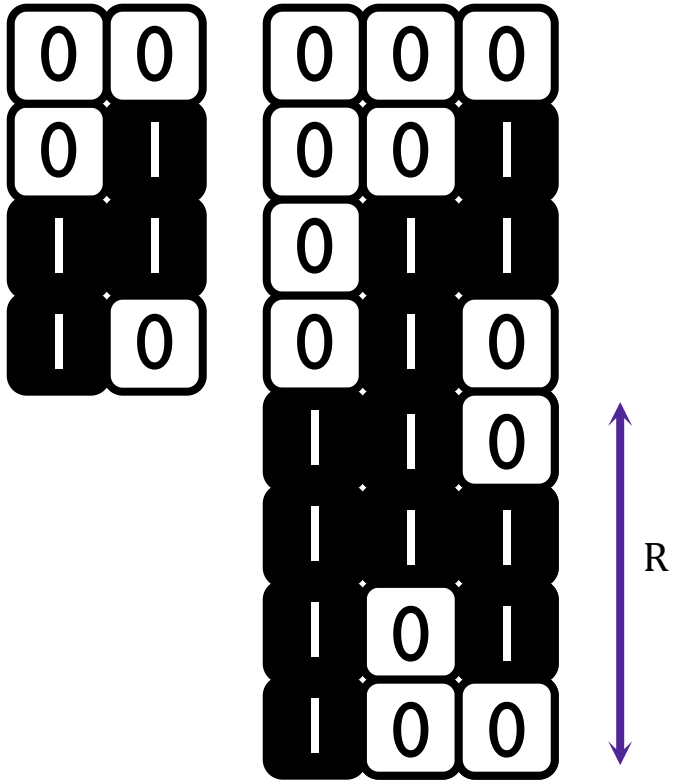
The *binary reflected Gray code* orders the n-bit binary strings so that consecutive strings differ in one bit. It is typically defined recursively as follows

$$B(n) = 0 \cdot B(n\text{-}1), 1 \cdot B(n\text{-}1)^R \text{ with } B(1) = 0, 1$$

where $\cdot$ denotes concatenation and ^{R} is list reflection.

The **Gray code** is attributed to **Frank Gray** who referred to it as *reflected order* in a 1945 patent from Bell Labs. However, it was known (even at Bell Labs) before this.

While the recursive definition of $B(n)$ explains its global structure, it is neither the simplest or most efficient definition for generating it iteratively.



Binary reflected Gray code $B(n)$ for $n = 2$ (left) and then $n = 3$ (right).

The Greedy Gray Code Algorithm

Aaron Williams haron@uvic.ca *

Department of Mathematics and Statistics, McGill University

Abstract. We reinterpret classic Gray codes for binary strings, permutations, combinations, binary trees, and set partitions using a simple greedy algorithm. The algorithm begins with an initial object and an ordered list of operations, and then repeatedly creates a new object by applying the first possible operation to the most recently created object.

GreedyGray(s, $\langle f_1, f_2, \ldots, f_m \rangle$)

- Initialize a list $L = s$.
- Repeatedly extend the list L as follows:
  - Let x be the last object in the list.
  - Let i be the minimum index with $f_i(x) \notin L$. If there is no such index i, then stop.
  - Add $f_i(x)$ to the end of the list.

The greedy Gray code algorithm can be used to generate a list by making two choices.

- Choose a start object s.
- Choose a prioritization of the operations $\langle f_1, f_2, \ldots, f_m \rangle$ used to change the objects.

The algorithm *succeeds* when it generates a list of all of the objects. Otherwise, it *fails*.
It is simple (i.e., just make two choices) but not efficient (i.e., it stores all previous objects).

Simpler: Greedily Flip the Rightmost Bit

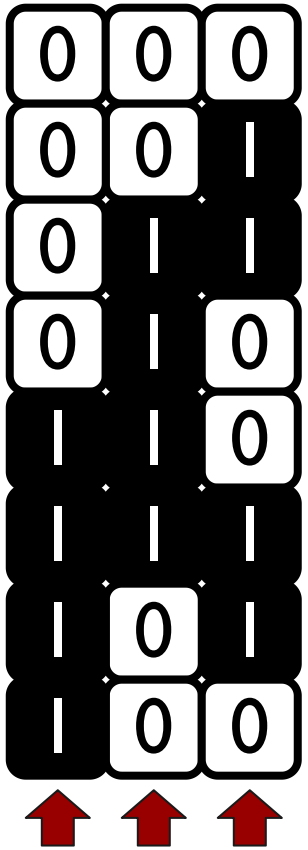
The binary reflected Gray code is also generated greedily.

- Start with $s = 0^n$ (i.e., all zeros).
- Prioritize flipping bits from right to left.

This is not an efficient algorithm in terms of memory because it needs to remember previous objects.

However, it is very simple to describe, so long as the greedy Gray code algorithm has been explained.

It is also possible to generate this order in a more efficient manner.



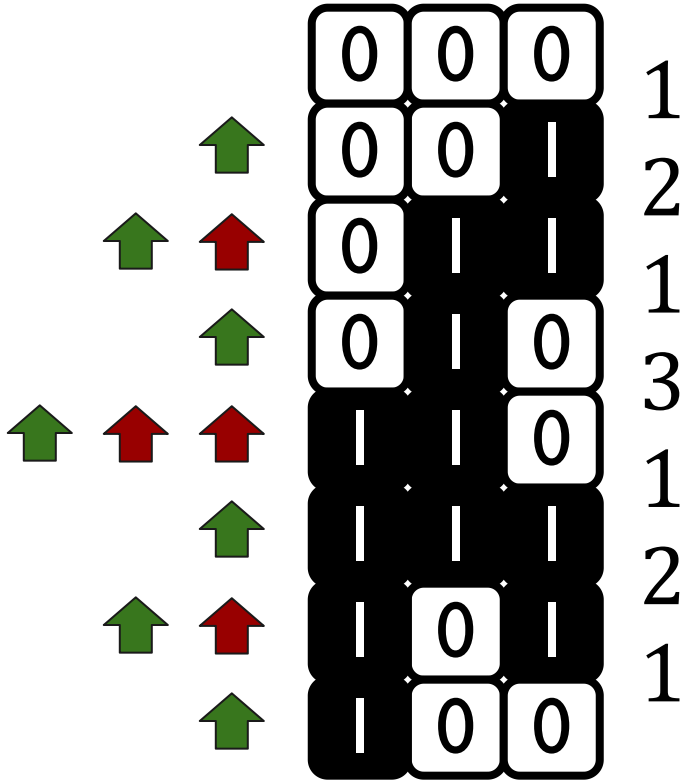
Greedily generating $B(3)$

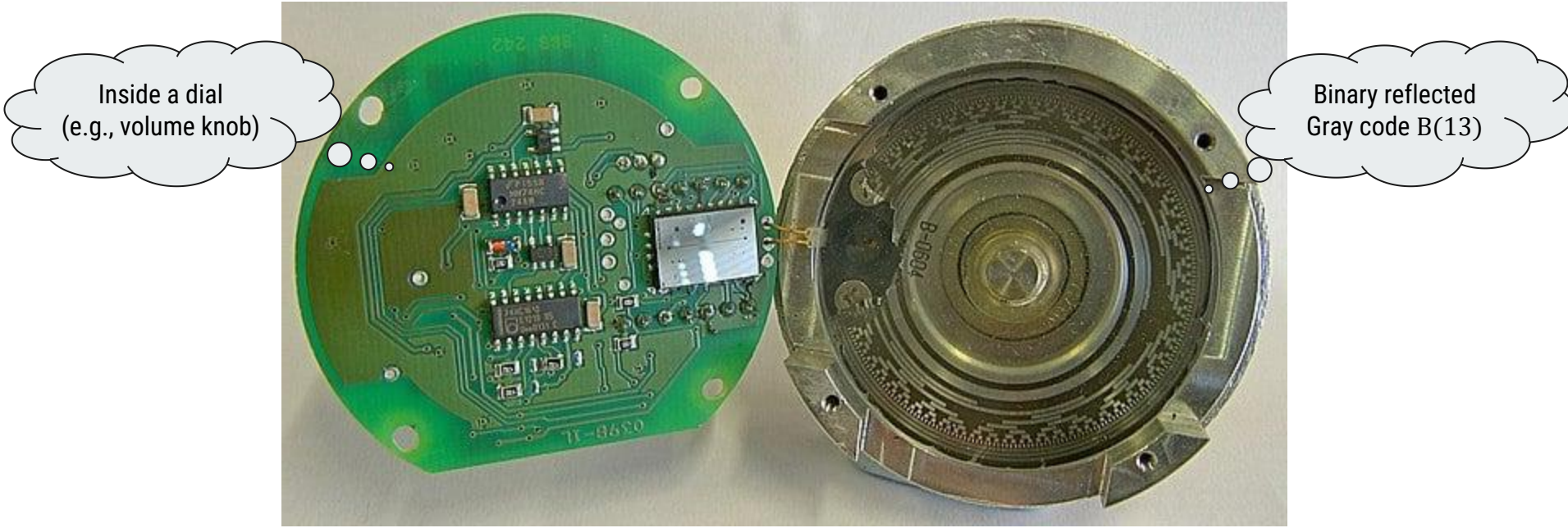
Faster: Binary Ruler Sequence

The binary reflected Gray code's change sequence is the binary ruler sequence. In other words, successive entries in ruler(2, 2, …, 2) specify which bit to change.

In other words, the difference between the lexicographic order of binary strings (i.e., counting in binary) and the binary reflected Gray code is that the ruler sequence specifies the number of bits to change instead of the single bit to change.

This leads to a loopless implementation of the binary reflected Gray code.





Rotary encoders use the binary reflected Gray code.

Plain Changes

for permutations

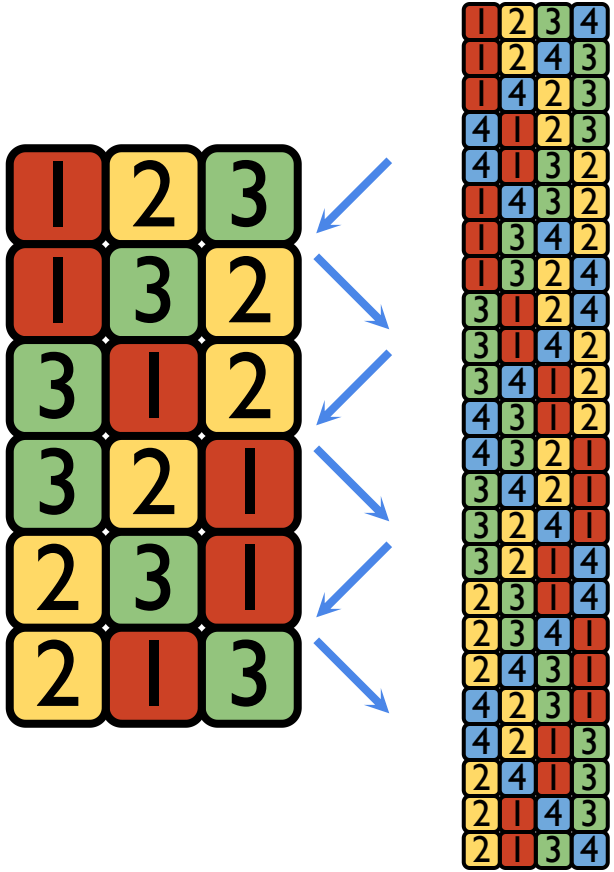
Plain Changes

Plain changes orders the permutations of $[n]$ so that consecutive permutations differ by a *swap* (i.e,. exchange a pair of adjacent entries).

It is typically defined recursively by alternately zigging (right to left) and zagging (left to right) the value n through successive permutations of $[n-1]$.

$$P(n) = \mathrm{zig}(p_1), \mathrm{zag}(p_2), \ldots, \mathrm{zig}(p_{(n-1)!-1}), \mathrm{zag}(p_{(n-1)!}),$$
$$\text{with } P(2) = p_1, p_2 = 12, 21$$

The order was discovered in 1600s by bell-ringers.
It was rediscovered multiple times in the 1960s and is often known as *Steinhaus-Johnson-Trotter order*.



Plain changes $P(n)$
for $n = 3$ (left) and then $n = 4$ (right).

Simpler: Greedily Swap the Largest Value

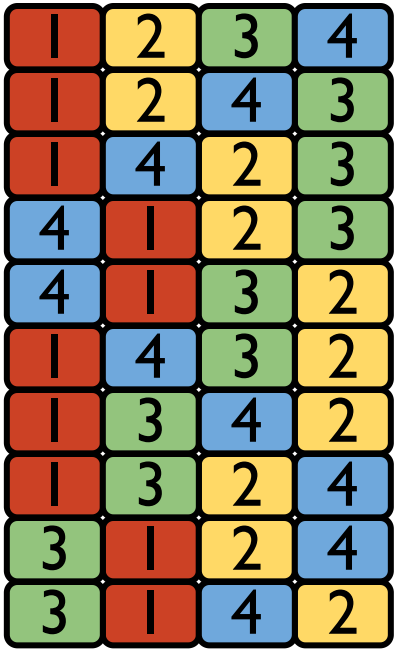
Plain changes is also generated greedily.

- Start with $s = 1\ 2 \cdots n$ (i.e., identity permutation).
- Prioritize swaps involving the largest possible value.

This is not an efficient algorithm in terms of memory because it needs to remember previous objects.

However, it is very simple to describe, so long as the greedy Gray code algorithm has been explained.

It is also possible to generate this order in a more efficient manner.



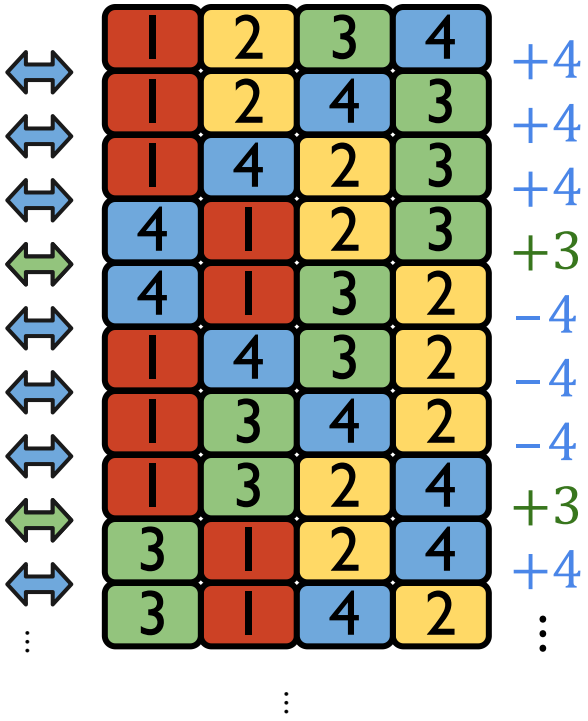
Greedily generating $P(4)$

Faster: Factorial Ruler Sequence

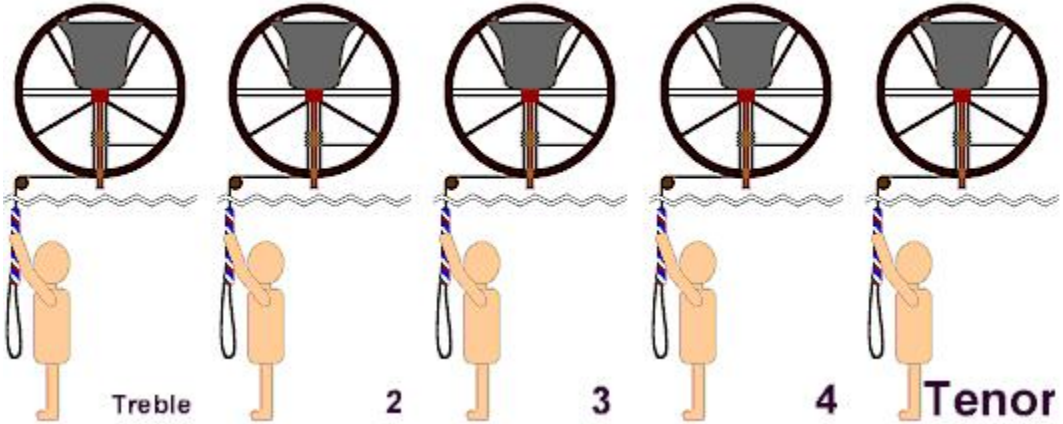
The changes in plain changes follow one of the factorial ruler sequences with signs: $\text{ruler}\pm(n, n\text{-}1, \ldots, 1)$.

- Entries of $+k$ for swapping k to the left.
- Entries of $-k$ for swapping k to the right.

This leads to a loopless algorithm for generating plain changes $P(n)$.



Greedily generating $P(4)$



Bell ringing.

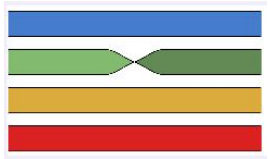
New Result:
Signed Plain Changes

Signed Permutations

A *signed permutation* is a permutation of $[n]$ in which each symbol is given a + or - sign.
There are $2^n \cdot n!$ signed permutations and they are the product of a permutation and binary string.

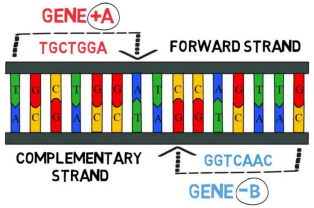
$$+1+2,\ +1\text{-}2,\ \text{-}1+2,\ \text{-}1\text{-}2,\ +2+1,\ +2\text{-}1,\ \text{-}2+1,\ \text{-}2\text{-}1$$

Signed permutations for $n = 2$.

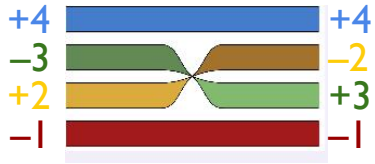


Signed permutations model orders in which the elements also have an orientation.

Trains and subways have orientations (e.g., eastbound or westbound track).



Genes either appear in the forward or reverse direction in DNA.



A 2-twist which changes the signs and order of symbols 2 and 3.

Signed permutations can be generated as composite objects (i.e., start with one of the $n!$ permutations, and create all 2^n of its signings, then repeat for the next permutation, and so on).

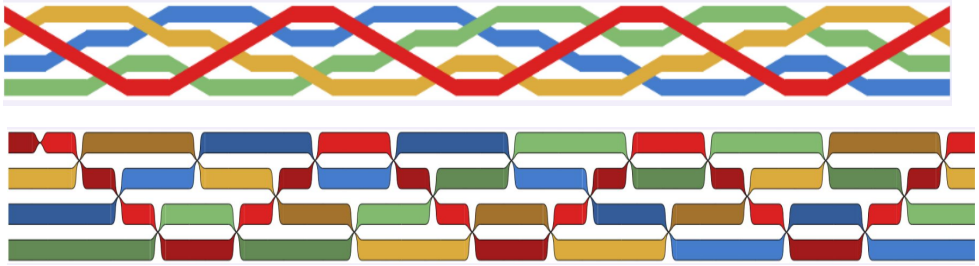
We instead consider natural operations that can change both parts of the object simultaneously.
We model the object using n two-sided ribbons, and we twist ribbons to create a new object.

Greedy and Speedy Generation of Signed Permutations

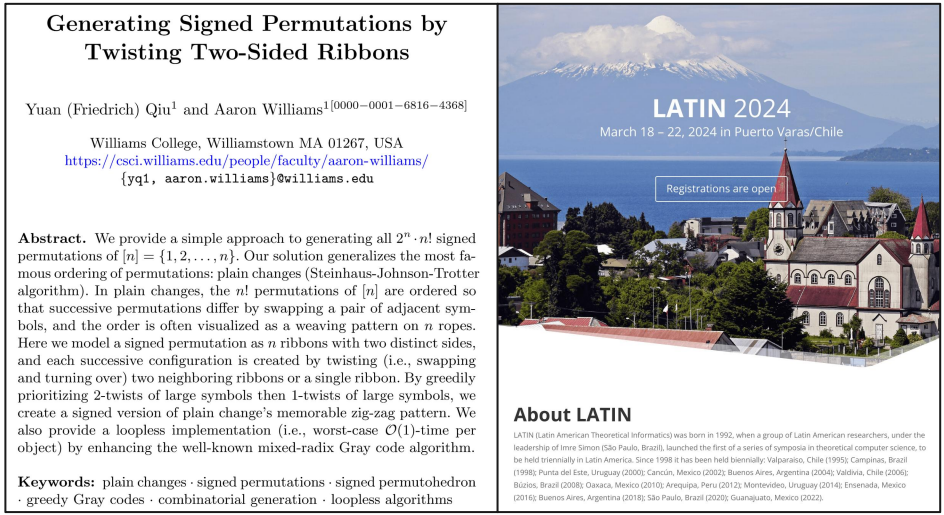
Greedily generate signed permutations by prioritizing 2-twists of the largest possible symbol then 1-twists of the largest possible symbol. We name the resulting Gray code *signed plain changes*.

Loopless generation via the ruler sequence $\text{ruler}\pm(n, n\text{-}1, \ldots, 2, 1, 2, 2, \ldots, 2)$ where

- $+k$ for 2-twisting k to the left.
- $-k$ for 2-twisting k to the right.
- $\pm(n+k)$ for 1-twisting k.



Plain changes (above with ropes) and the start of our new signed plain changes (below with ribbons) for $n = 4$.



Generating Signed Permutations by Twisting Two-Sided Ribbons

Yuan (Friedrich) Qiu[1] and Aaron Williams[1][0000−0001−6816−4368]

Williams College, Williamstown MA 01267, USA
https://csci.williams.edu/people/faculty/aaron-williams/
{yq1, aaron.williams}@williams.edu

Abstract. We provide a simple approach to generating all $2^n \cdot n!$ signed permutations of $[n] = \{1, 2, \ldots, n\}$. Our solution generalizes the most famous ordering of permutations: plain changes (Steinhaus-Johnson-Trotter algorithm). In plain changes, the $n!$ permutations of $[n]$ are ordered so that successive permutations differ by swapping a pair of adjacent symbols, and the order is often visualized as a weaving pattern on n ropes. Here we model a signed permutation as n ribbons with two distinct sides, and each successive configuration is created by twisting (i.e., swapping and turning over) two neighboring ribbons or a single ribbon. By greedily prioritizing 2-twists of large symbols then 1-twists of large symbols, we create a signed version of plain change's memorable zig-zag pattern. We also provide a loopless implementation (i.e., worst-case $\mathcal{O}(1)$-time per object) by enhancing the well-known mixed-radix Gray code algorithm.

Keywords: plain changes · signed permutations · signed permutohedron · greedy Gray codes · combinatorial generation · loopless algorithms

LATIN 2024 conference paper.

We considered a dozen different greedy algorithms and consider this one to be the best.

A Python Implementation

A loopless implementation of our signed plain change order twisted(n) in Python 3. Entries in the twisted ruler sequence ruler±($n, n-1, \ldots, 2, 1, 2, 2, \ldots, 2$) select the 2-twist or 1-twist (i.e., flip) to apply[8]. Programs are available online [37].

```
# Flip sign of value v in signed permutation p with unsigned inverse q
def flip(p, q, v):  # with 1-based indexing, ie p[0] and q[0] are ignored.
    p[q[v]] = -p[q[v]]
    return p, q

# 2-twists value v to the left / right using delta = -1 / delta = 1
def twist(p, q, v, delta):  # with 1-based indexing into both p and q.
    pos = q[v]            # Use inverse to get the position of value v.
    u = abs(p[pos+delta]) # Get value to the left or right of value v.
    p[pos], p[pos+delta] = -p[pos+delta], -p[pos]      # Twist u and v.
    q[v], q[u] = pos+delta, pos              # Update unsigned inverse.
    return p, q  # Return signed permutation and its unsigned inverse.

# Generate each signed permutation in worst-case O(1)-time.
def twisted(n):
  m = 2*n-1  # The mixed-radix bases are n, n-1, ..., 2, 1, 2,..., 2.
  bases = tuple(range(n,1,-1)) + (2,) * n     # but the 1 is omitted.
  word = [0] * m             # The mixed-radix word is initially 0^m.
  dirs = [1] * m             # Direction of change for digits in word.
  focus = list(range(m+1))  # Focus pointers select digits to change.
  flips =   [lambda p,q,v=v: flip(p,q,v)     for v in range(n,0,-1)]
  twistsL = [lambda p,q,v=v: twist(p,q,v,-1) for v in range(n,1,-1)]
  twistsR = [lambda p,q,v=v: twist(p,q,v, 1) for v in range(n,1,-1)]
  fns = [None] + twistsL + flips + flips[-1::-1] + twistsR[-1::-1]
  p = [None] + list(range(1,n+1))  # To use 1-based indexing we set
  q = [None] + list(range(1,n+1))  # and ignore p[0] = q[0] = None.
  yield p[1:]  # Pause the function and return signed permutation p.
  while focus[0] < m: # Continue if the digit to change is in word.
    index = focus[0]  # The index of the digit to change in word.
    focus[0] = 0      # Reset the first focus pointer.
    word[index] += dirs[index]  # Adjust the digit using its direction.
    change = dirs[index] * (index+1)  # Note: change can be negative.
    if word[index] == 0 or word[index] == bases[index]-1:  # If the
      focus[index] = focus[index+1] # mixed-radix word's digit is at
      focus[index+1] = index+1      # its min or max value, then update
      dirs[index] = -dirs[index]    # focus pointers, change direction.
    p, q = fns[change](p, q)  # Apply twist or flip encoded by change.
    yield p[1:]

# Demonstrating the use of our twisted function for n = 4.
for p in twisted(4): print(p)  # Print all 2^n n! signed permutations.
```

Python implementation in the LATIN 2024 paper.