



Generating Signed Permutations by Twisting Two-Sided Ribbons

Yuan Qiu and Aaron Williams^(✉) 

Williams College, Williamstown, MA 01267, USA

{yq1,aaron.williams}@williams.edu



<https://csci.williams.edu/people/faculty/aaron-williams/>

Abstract. We provide a simple approach to generating all $2^n \cdot n!$ signed permutations of $[n] = \{1, 2, \dots, n\}$. 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

1 Generating Permutations and Signed Permutations



The generation of permutations is a classic problem that dates back to the dawn of computer science (and several hundred years earlier). The goal is to create all $n!$ permutations of $[n] = \{1, 2, \dots, n\}$ as efficiently as possible. A wide variety of approaches have been considered, some of which can be conceptualized using a specific physical model of the permutation. Let’s consider three such examples.

Zaks’ algorithm [38] can be conceptualized using a stack of n pancakes of varying sizes. Successive permutations are created by flipping some pancakes at the top of the stack, which is equivalent to a *prefix-reversal* in the permutation. For example, if  represents 1234, then flipping the top three pancakes gives  or $\overleftarrow{123}4 = 3214$. Table 1 shows the full order for $n = 4$. Zaks designed his ‘new’ order to have an efficient array-based implementation. Unknown to Zaks, Klügel had discovered this *pancake order* by 1796 [14]; see [1] for further details.

Corbett’s algorithm [3] can be conceptualized using n marbles on a ramp. Successive permutations are created by moving a marble to the top of the ramp,

which is equivalent to a *prefix-rotation* in the permutation. For example, if  represents 1234, then moving the fourth marble gives  or $\overleftarrow{1234} = 4123$.

The algorithms by Zaks and Corbett are well-known, and have their own specific applications. For example, in interconnection networks [4], the algorithms give Hamilton cycles in the pancake network and rotator network, respectively.

Plain changes can be conceptualized using n parallel ropes. Successive permutations are obtained by crossing one rope over a neighboring rope as in weaving. This is equivalent to a *swap* (or *adjacent-transposition*) in the permutation. For example, if  represents 1234, then swapping the middle pair gives  or $1\overline{2}34 = 1324$. Plain changes dates to bell-ringers in the 1600 s [5]. Figure 3 shows the order for $n = 4$ and its zig-zag pattern. It is also known as the *Steinhaus-Johnson-Trotter algorithm* [16, 31, 34] due to rediscoveries circa 1960.

Many other notable approaches to permutation generation exist, with surveys by Sedgewick [30], Savage [27], and Mütze [22], and frameworks by Knuth [17] and Ganapathi and Chowdhury [8]. While some methods have specific advantages [15] or require less additional memory when implemented [19], there is little doubt that plain changes is the solution to permutation generation.

A *signed permutation* of $[n]$ is a permutation of $[n]$ in which every symbol is given a \pm sign. We let S_n and S_n^\pm be the sets of all permutations and signed permutations of $[n]$, respectively. Note that $|S_n| = n!$ and $|S_n^\pm| = 2^n \cdot n!$. For example, $231 \in S_3$ has eight different signings, including $+2-3-1 \in S_3^\pm$. For convenience, we also use bold or overlines for negatives, with **231** and $2\overline{3}1$ denoting $+2-3-1$. Signed permutations arise in many contexts including genomics [7].

The efficient generation of signed permutations has been considered. Suzuki, Sawada, and Kaneko [33] treat signed permutations as stacks of n *burnt pancakes* and provide a signed version of Zaks' algorithm. Korsh, LaFollette, and Lipschutz [18] provide a Gray code that swaps two symbols (and preserves their signs) or changes the rightmost symbol's sign. Both approaches offer improvements over standard *lexicographic orders* (i.e., alphabetic orders) but neither is considered to be the solution for signed permutations. We define a *signed plain change order* to be any extension of plain changes to signed permutations.

Physical Model of Signed Permutations: Two-Sided Ribbons. A *two-sided ribbon* is glossy on one side and matte on the other¹, and we model a signed permutation using n two-sided ribbons in parallel. We modify the ribbons via twists. More specifically, a k -*twist* turns over k neighboring ribbons and reverses their order, as visualized in Fig. 1 for $k = 1, 2$. A twist performs a *complementing substring reversal*, or simply a *reversal* [11], on the signed permutation.

Our goal is to create a *twist Gray code* for signed permutations. This means that each successive entry of S_n^\pm is created by applying a single twist. Equivalently, a sequence of $2^n n! - 1$ twists generates each entry of S_n^\pm in turn. It should be obvious that 1-twists are insufficient for this task on their own, as they do not modify the underlying permutation. Similarly, 2-twists are insufficient on their

¹ Manufacturers refer to this type of ribbon as *single face* as only one side is polished.



(a) The 1-twist changes $1\,2\,3\,4$ into $1\,\bar{2}\,3\,4$. (b) The 2-twist changes $1\,\bar{2}\,3\,4$ into $1\,3\,\bar{2}\,4$.

Fig. 1. Two-sided ribbons with distinct positive (i.e., glossy) and negative (i.e., matte) sides running in parallel. A k -twist reverses the order of k neighboring ribbons and turns each of them over, as shown for (a) $k = 1$ and (b) $k = 2$.

own, as they do not modify the number of positive symbols modulo two. However, we will show that 2-twists and 1-twists are sufficient when used together. Our solution is a signed plain change order that we name *twisted plain changes*.

Application: Train-Based Traveling Salesman Problems. Exhaustive generation is central to many applications, including testing and exact algorithms. Gray code algorithms can also improve the latter. For example, a traveling salesman problem on n cities can be solved by generating all $n!$ permutations of $[n]$, with each member of S_n providing a possible route through the cities (e.g., $p_1p_2\cdots p_n \in S_n$ represents the route $p_1 \rightarrow p_2 \rightarrow \cdots \rightarrow p_n$). Plain changes is advantageous because successive routes differ in at most three segments (e.g., swapping $p_i p_{i+1}$ to $p_{i+1} p_i$ replaces segment $p_i \rightarrow p_{i+1}$ with $p_i \rightarrow p_{i+2}$) [15]. Thus, the distance of each successive route can be *updated* in constant time.

Now consider a TSP-variant involving trains, where each of the n stations can be entered/exited in one of two orientations (e.g., the train may travel along the station’s eastbound or westbound track). Note that the time taken to travel from one station to another depends on these orientations. As a result, there are $2^n \cdot n!$ possible routes and they correspond to the members of S_n^\pm . Our twist Gray code algorithm generates successive routes that differ in at most three segments.

1.1 Outline

Section 2 provides background on combinatorial generation. Section 3 defines our twist Gray code using a simple (but inefficient) greedy algorithm. Section 4 discusses ruler sequences and their applications. Section 5 uses a signed ruler sequence to generate our Gray code in worst-case $\mathcal{O}(1)$ -time per signed permutation. A Python implementation of our final algorithm appears in the appendix. The proofs of Lemma 1–3 are left as exercises to the reader due to page limits.

2 Combinatorial Generation

As Ruskey explains in *Combinatorial Generation* [26], humans have been writing exhaustive lists of various kinds for thousands of years, and more recently, programming computers to do so. Here we review basic concepts and terminology, then we discuss two foundational results and modern reinterpretations of them.

2.1 Gray Codes and Loopless Algorithms

If successive objects in an order differ in a constant amount (by some metric), then it is a *Gray code*. If an algorithm generates each object in amortized or worst-case $\mathcal{O}(1)$ -time, then it is *constant amortized time* (CAT) or *loopless* [6]. To understand these terms, note that a well-written generation algorithm shares one object with an application. It modifies the object and announces that the ‘next’ object can be *visited*, without using linear-time to create a new object. Loopless algorithms make constant-time modifications using a Gray code. For example, Zaks’ order can be generated in CAT as its prefix-reversals have constant average length (see Ord-Smith’s earlier ECONOPERM program [24]), but a loopless algorithm is not possible as a length n prefix-reversal takes $\Theta(n)$ -time².

2.2 Binary Reflected Gray Code and Plain Changes

Plain change’s stature in combinatorial generation is rivaled only by the *binary reflected Gray code*³. The BRGC orders n -bit binary strings by *bit-flips*, meaning successive strings differ in one bit. It is typically defined recursively as

$$\text{brgc}(n) = 0 \cdot \text{brgc}(n-1), 1 \cdot \text{reflect}(\text{brgc}(n-1)) \text{ with } \text{brgc}(1) = 0, 1 \quad (1)$$

where *reflect* denotes *list reflection* (i.e., last string goes first). For example,

$$\text{brgc}(2) = 0 \cdot \text{brgc}(1), 1 \cdot \text{reflect}(\text{brgc}(1)) = 0 \cdot (0, 1), 1 \cdot (1, 0) = 0\overline{0}, \overline{0}1, 1\underline{1}, 1\underline{0}$$

where overlines and underlines have been added for flips from 0 to 1 and 1 to 0, respectively. The order for $n = 4$ is visualized in Fig. 2 using two-sided ribbons, where each bit-flip is a 1-twist of the corresponding ribbon.

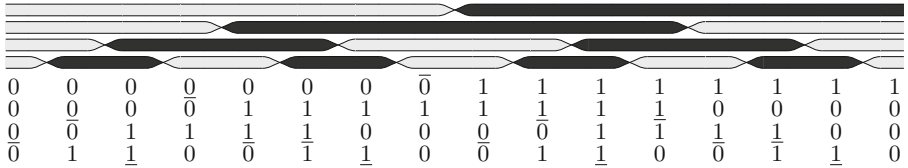


Fig. 2. Binary reflected Gray code using indistinct two-sided ribbons for $n = 4$.

Plain changes recursively zigs and zags n through permutations of $[n-1]$. In (2), *zig* and *zag* give length $n-1$ lists that repeatedly swap n to the left or right.

$$\begin{aligned} \text{plain}(n) = & \text{zig}(p_1 \cdot n), \text{zag}(n \cdot p_2), \dots, \text{zig}(p_{(n-1)!-1} \cdot n), \text{zag}(n \cdot p_{(n-1)!}) \\ & \text{with } \text{plain}(n-1) = p_1, p_2, \dots, p_{(n-1)!} \end{aligned} \quad (2)$$

² If the permutation is stored in a BLL instead of an array, then loopless is possible [35].

³ The eponymous *Gray code* by Gray [10] also demonstrates Stigler’s law [32]: [9, 13].

Formula (2) assumes $(n-1)!$ is even, so we use base case $\text{plain}(2) = \overleftarrow{12}, 21$. Here the arrow denotes a larger value swapping left past its smaller neighbor. Thus,

$$\text{plain}(3) = \text{zig}(12 \cdot 3), \text{zag}(3 \cdot 21) = \overleftarrow{123}, \overleftarrow{132}, \overleftarrow{312}, \overrightarrow{321}, \overrightarrow{231}, 213,$$

Figure 3 visualizes $\text{plain}(4)$ with distinct one-sided ribbons⁴. Note how 4 zigzags.



Fig. 3. Plain changes $\text{plain}(n)$ using distinct one-sided ribbons for $n = 4$.

2.3 The Greedy Gray Code Algorithm

Historically, Gray codes have been created using recursion. In contrast, the *greedy Gray code algorithm* [36] attempts to create a Gray code one object at time. A list $\text{greedy}(\mathbf{s}, \langle o_1, o_2, \dots, o_k \rangle)$ is initialized with a *start object* \mathbf{s} , then it is repeatedly extended as follows: If \mathbf{t} is the last object in the list, then add $o_i(\mathbf{t})$ to the end of the list, where i is the minimum index such that $o_i(\mathbf{t})$ is valid and not in the list. This continues until none of the *operations* o_1, o_2, \dots, o_k produce a new object.

The binary reflected Gray code is a *greedy Gray code*: start at $\mathbf{s} = 0^n$ and flip the rightmost possible bit [36]. That is, $\text{brgc}(n) = \text{greedy}(0^n, \langle f_1, f_2, \dots, f_n \rangle)$ where f_i flips b_i in $b_n b_{n-1} \dots b_1 \in B_n$. For example, the order for $n = 4$ begins

$$\text{brgc}(4) = 000\overline{0}, 00\overline{0}1, 001\overline{1}, 0010, \dots \quad (3)$$

To continue (3) we consider applying the bit-flips to the current last object $\mathbf{t} = 0010$. We can't flip its right bit since $f_1(\mathbf{t}) = f_1(0010) = 001\overline{0} = 0011$ is already in the list. Similarly, $f_2(\mathbf{t}) = f_2(0010) = 00\overline{1}0 = 0000$ is also in the list. But $f_3(\mathbf{t}) = f_3(0010) = 0\overline{0}10 = 0110$ is not in the list, so it is the next string.

Plain changes is also greedy: start at $\mathbf{s} = 12 \dots n$ and swap the largest possible value left or right [36]. That is, $\text{plain}(n) = \text{greedy}(12 \dots n, \langle \overleftarrow{s_n}, \overrightarrow{s_n}, \dots, \overleftarrow{s_2}, \overrightarrow{s_2} \rangle)$ ⁵ where $\overleftarrow{s_v}$ and $\overrightarrow{s_v}$ swap value v to the left and right, respectively, when applied to any member of S_n . For example, the order for $n = 4$ begins

$$\text{plain}(4) = 12\overleftarrow{34}, 12\overleftarrow{43}, \overleftarrow{14}23, 41\overleftarrow{23}, \overrightarrow{41}32, \overrightarrow{14}32, 13\overrightarrow{42}, 1324, \dots \quad (4)$$

We can't apply $\overleftarrow{s_4}$ to $\mathbf{t} = 1324$ since $\overleftarrow{s_4}(1324) = 13\overleftarrow{24} = 1342$ is already in the list. Nor can we apply the next highest-priority operation $\overrightarrow{s_4}$ as $\overrightarrow{s_4}(1324)$ is invalid. But $\overleftarrow{s_3}(\mathbf{t}) = \overleftarrow{13}24 = 3124$ is not in the list, so it is the next permutation. Plain change's greedy formula also holds when the swaps are replaced by *jumps* (i.e., values can only be swapped over smaller values) and with $\overleftarrow{s_1}$ and $\overrightarrow{s_1}$ omitted [12].

⁴ Physically, a ribbon moves above or below its neighbor, but that is not relevant here.

⁵ $\overleftarrow{s_1}$ and $\overrightarrow{s_1}$ are omitted as they equal other swaps. In fact, the swaps are all *jumps* [12].

Reflecting BRGC and Plain Changes. Interestingly, if either of the previous two greedy algorithms is started from their final object, then the entire order is reflected. For example, if $\mathbf{s} = 1000$ is chosen in Fig. 2, or $\mathbf{s} = 2134$ is chosen in Fig. 3, then the greedy algorithms generate the objects from right-to-left. This is in part explained by their palindromic change sequences (see Sect. 4).

Lemma 1. $\text{greedy}(10^{n-1}, \langle f_1, f_2, \dots, f_n \rangle) = \text{reflect}(\text{brgc}(n))$.

Lemma 2. $\text{greedy}(2134 \cdots n, \langle \overleftarrow{s_n}, \overrightarrow{s_n}, \overleftarrow{s_{n-1}}, \overrightarrow{s_{n-1}}, \dots, \overleftarrow{s_2}, \overrightarrow{s_2} \rangle) = \text{reflect}(\text{plain}(n))$.

3 A Signed Plain Change Order: Twisted Plain Changes

Now we present a greedy solution to generating a signed plain change order.

Definition 1. Twisted plain changes $\text{twisted}(n)$ is the signed permutation order visited by Algorithm 1. It starts with $\mathbf{s} = +1+2 \cdots +n \in S_n^\pm$ and prioritizes 2-twists of the largest possible value then 1-twists of the largest possible value. That is, $\text{greedy}(\mathbf{s}, \langle \overleftarrow{\mathbf{t}}_n, \overrightarrow{\mathbf{t}}_n, \overleftarrow{\mathbf{t}}_{n-1}, \overrightarrow{\mathbf{t}}_{n-1}, \dots, \overleftarrow{\mathbf{t}}_2, \overrightarrow{\mathbf{t}}_2, \mathbf{t}_n, \dots, \mathbf{t}_2, \mathbf{t}_1 \rangle)$, where $\overleftarrow{\mathbf{t}}_v$ and $\overrightarrow{\mathbf{t}}_v$ 2-twist value v left or right, and \mathbf{t}_v 1-twists value v (i.e., v 's sign is flipped).

Algorithm 1. Greedy algorithm for generating twisted plain changes $\text{twisted}(n)$.

```

1: procedure Twisted( $n$ )           ▷ Signed permutations are visited in  $\text{twisted}(n)$  order
2:    $T \leftarrow \overleftarrow{\mathbf{t}}_n, \overrightarrow{\mathbf{t}}_n, \overleftarrow{\mathbf{t}}_{n-1}, \overrightarrow{\mathbf{t}}_{n-1}, \dots, \overleftarrow{\mathbf{t}}_2, \overrightarrow{\mathbf{t}}_2, \mathbf{t}_n, \dots, \mathbf{t}_2, \mathbf{t}_1$  ▷ List 2-twists then 1-twists
3:    $\pi \leftarrow +1 +2 \cdots +n$            ▷ Starting signed permutation  $\mathbf{s} = \pi \in S_n^\pm$ 
4:    $\text{visit}(\pi)$                          ▷ Visit  $\pi$  for the first and only time
5:    $S = \{\pi\}$                            ▷ Add  $\pi$  to the visited set
6:    $i \leftarrow 1$                          ▷ 1-based index into  $T$ ;  $T[1] = \overleftarrow{\mathbf{t}}_n$  will 2-twist  $n$  left
7:   while  $i \leq 3n - 2$  do             ▷ Index  $i$  iterates through the  $3n - 2$  twists in  $T$ 
8:      $\pi' \leftarrow T[i](\pi)$            ▷ Apply the  $i^{\text{th}}$  highest priority twist to create  $\pi'$ 
9:     if  $\pi' \notin S$  then                 ▷ Check if  $\pi'$  is a new signed permutation
10:       $\pi \leftarrow \pi'$                  ▷ Update the current signed permutation  $\pi$ 
11:       $\text{visit}(\pi)$                      ▷ Visit  $\pi$  for the first and only time
12:       $S = S \cup \{\pi\}$                  ▷ Add  $\pi$  to the visited set
13:       $i \leftarrow 1$                    ▷ Reset the 1-based index into  $T$ 
14:   else
15:      $i \leftarrow i + 1$                  ▷ If  $\pi' \in S$ , then consider the next twist

```

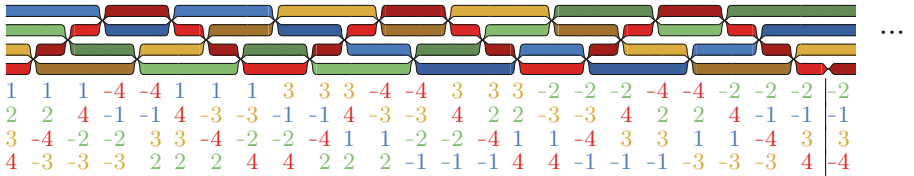


Fig. 4. Twisted plain changes $\text{twisted}(n)$ for $n = 4$ up to its 25th entry.

For example, our twist Gray code for $n = 4$ begins as follows

$$\text{twisted}(4) = 12\overleftarrow{3}4, \overleftarrow{1}2\overleftarrow{4}3, \overleftarrow{1}4\overleftarrow{2}3, \overleftarrow{4}1\overleftarrow{2}3, \overleftarrow{4}1\overrightarrow{3}2, \overrightarrow{1}4\overrightarrow{3}2, \overrightarrow{1}3\overrightarrow{4}2, \overrightarrow{1}3\overrightarrow{2}4, \dots \quad (5)$$

with bold for negatives and arrows for twists. To continue the order note that $\overleftarrow{t}_4(\mathbf{t}) = \overleftarrow{t}_4(1324) = \overrightarrow{1}\overrightarrow{3}\overrightarrow{2}4 = 1342$ is already in the list and $\overrightarrow{t}_4(\mathbf{t})$ is invalid. But $\overleftarrow{t}_3(\mathbf{t}) = \overleftarrow{t}_3(1324) = \overleftarrow{1}\overrightarrow{3}\overrightarrow{2}4 = 3124$ is new, so it is the next signed permutation.

Figure 4 shows the start of $\text{twisted}(n)$ for $n = 4$. Note that the first 24 entries are obtained by 2-twists. The result is a familiar zig-zag pattern, but with every ribbon turning over during each pass. The 25th entry is obtained by a 1-twist.

3.1 2-Twisted Permutohedron and Signed Permutohedra

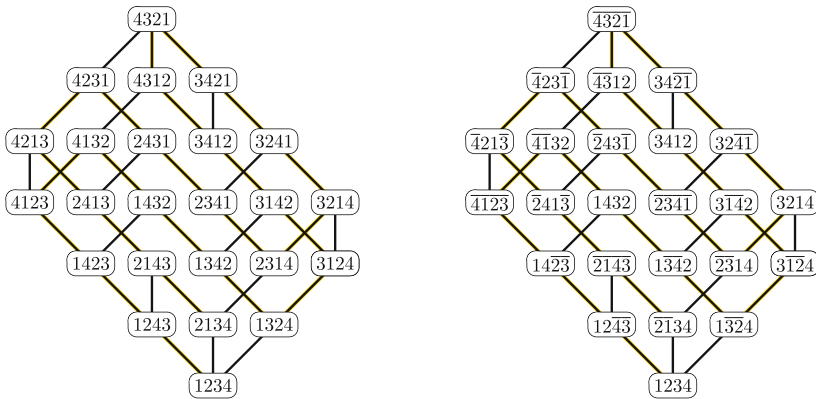
At this point it is helpful to compare the start of plain changes and twisted plain changes. The *permutohedron of order n* is a graph whose vertices are permutations S_n and whose edges join two permutations that differ by a swap. Plain changes traces a Hamilton path in this graph, as illustrated in Fig. 5a.

Now consider signing each vertex $p_1p_2 \cdots p_n$ in the permutohedron as follows:

$$p_j = i \text{ is positive if and only if } i \equiv j \pmod{2}. \quad (6)$$

In particular, the permutation $12 \cdots n$ is signed as $+1+2 \cdots +n$ due to the fact that odd values are in odd positions, and even values are in even positions. One way of interpreting (6) is that swapping a symbol changes its sign. Thus, after this signing, the edges in the resulting graph model 2-twists instead of swaps. For this reason, we refer to the graph as a *2-twisted permutohedron of order n* .

Since twisted plain changes prioritizes 2-twists before 1-twists, the reader should be able to conclude that $\text{twisted}(n)$ starts by creating a Hamilton path in the 2-twisted permutohedron. This is illustrated in Fig. 5b.



(a) The Hamilton path from $1234 \in S_n$ to $2134 \in S_4$ follows plain(4). (b) The Hamilton path from $1234 \in S_n^\pm$ to $\overrightarrow{2}1\overrightarrow{3}4 \in S_n^\pm$ follows $\text{twisted}(4)$.

Fig. 5. The (a) permutohedron and (b) 2-twisted permutohedron for $n = 4$.

In general, there are 2^n *signed permutohedron of order n* . Each signed permutohedron contains $n!$ vertices, including a single signing of the vertex $12 \cdots n$, and edges for every possible 2-flip. In particular, the 2-twisted permutohedron is the signed permutohedron with vertex $+1+2 \cdots +n$ (i.e., $12 \cdots n$ is fully positive).

3.2 Global Structure

Our greedy approach can be verified to work for small n . To prove that it works for all n we need to deduce the global structure of the order that is created. We'll see that the order navigates through successive signed permutohedron.

Theorem 1. *Algorithm 1 visits a twist Gray code of signed permutations. That is, $\text{twisted}(n) = \text{greedy}(s, \langle \overleftarrow{t_n}, \overrightarrow{t_n}, \overleftarrow{t_{n-1}}, \overrightarrow{t_{n-1}}, \dots, \overleftarrow{t_2}, \overrightarrow{t_2}, t_n, \dots, t_2, t_1 \rangle)$ orders S_n^\pm .*

Proof. Since 2-twists are prioritized before 1-twists, the algorithm proceeds in the same manner as plain changes, except for the signs of the visited objects. As a result, it generates sequences of $n!$ signed permutations using 2-twists until a single 1-twist is required. One caveat is that the first signed permutation in a sequence alternates between having the underlying permutation of $1234 \cdots n$ or $2134 \cdots n$. This is due to the fact that plain changes starts at $1234 \cdots n$ and ends at $2134 \cdots n$ and swaps 12 to 21 one time. As a result, 12 will be inverted while traversing every second sequence of length $n!$, and these traversals will be done in reflected plain changes order by Lemma 2. More specifically, the order generated by the algorithm appears in Fig. 6, with an example in Fig. 7. \square

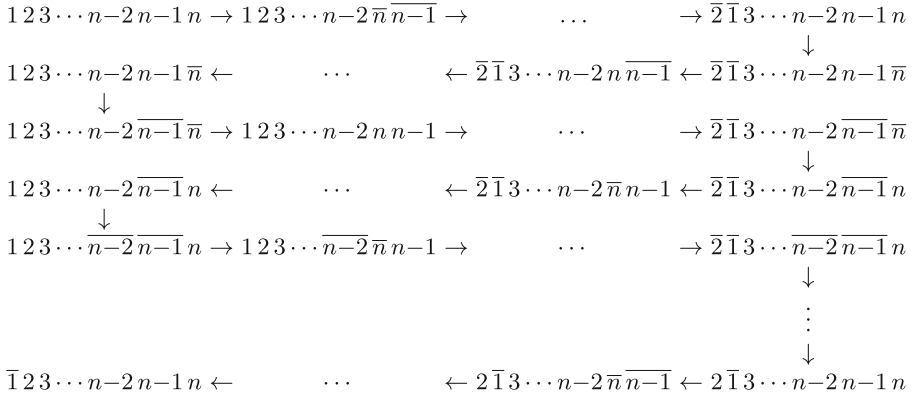


Fig. 6. The global structure of twisted plain changes. Each row greedily applies 2-twists to the largest possible symbol, thus following plain changes. At the end of a row, no 2-twist can be applied, and the down arrows greedily 1-twist the largest possible symbol. The rows alternate left-to-right and right-to-left (i.e., in boustrophedon order) by Lemma 2. The leftmost column contains $12 \cdots n$ signed according to successive strings in the binary reflected Gray code. The overall order is cyclic as a 1-twist on value 1 transforms the last entry into the first.

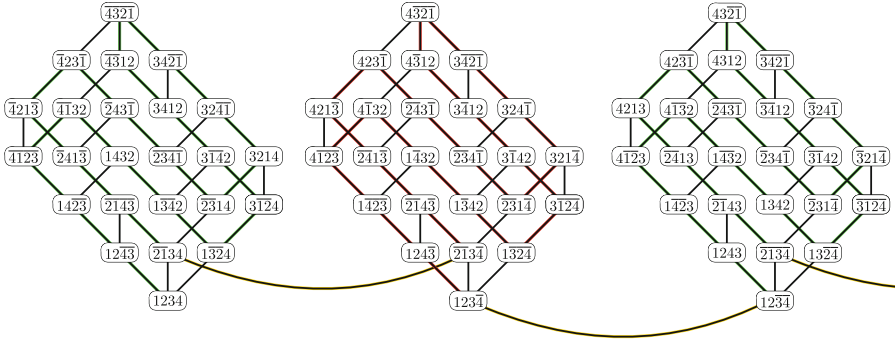


Fig. 7. Our `twisted(4)` order begins by traversing the above signed permutohedron, starting from the 2-twisted permutohedron on the left. Straight lines are the edges of a signed permutohedron (i.e., every possible 2-twist). Curved edges are 1-twists between the vertices shown, and they connect two signed permutohedron. Highlighted edges are used by the greedy algorithm: green subpaths start at a signed $1234 \cdots n$ vertex and proceed in plain changes; red subpaths start at a signed $2134 \cdots n$ vertex and proceed in reflected plain changes.

Theorem 1's proof can be used toward a CAT implementation of `twisted(n)`. We'll instead develop a loopless implementation of `twisted(n)` in Sects. 4–5.

4 Ruler Sequences

Here we consider integer sequences called *ruler sequences*. The sequences are named after the tick marks on rulers and tape measures whose heights follow the *decimal ruler sequence* $\text{ruler}(10, 10, \dots, 10)$. They are central to Algorithm 2 in Sect. 5, and relate `twisted(n)` to other Gray codes and lexicographic orders.

4.1 Ruler Sequences, Mixed-Radix Words, and Lexicographic Orders

The *ruler sequence* with bases b_n, b_{n-1}, \dots, b_1 can be inductively defined as follows, where commas join sequences, and exponentiation denotes repetition.

$$\text{ruler}(b_1) = 1^{b_1-1} = 1, 1, \dots, 1 \quad (\text{i.e., } b_1 - 1 \text{ copies}) \quad (7)$$

$$\text{ruler}(b_n, b_{n-1}, \dots, b_1) = (s, n)^{b_n-1}, s \text{ where } s = \text{ruler}(b_{n-1}, b_{n-2}, \dots, b_1) \quad (8)$$

Hence, $\text{ruler}(5, 3) = 1, 1, 2, 1, 1, 2, 1, 1, 2, 1, 1, 2, 1, 1 = (s, 2)^4, s$ since $s = \text{ruler}(3) = 1, 1$. The length of the ruler sequence is $|\text{ruler}(b_n, b_{n-1}, \dots, b_1)| = (\prod_{i=1}^n b_i) - 1$.

Bases can also be used to define the set of *mixed-radix words* $W_{b_n, b_{n-1}, \dots, b_1}$, where $w_n \cdots w_2 w_1$ is in the set if its digits satisfy $0 \leq w_i < b_i$ for $1 \leq i \leq n$. The number of these words is $|W_{b_n, b_{n-1}, \dots, b_1}| = \prod_{i=1}^n b_i = |\text{ruler}(b_n, b_{n-1}, \dots, b_1)| + 1$.

When mixed-radix words are written in lexicographic order, the ruler sequence is its *change sequence*. Each ruler entry is the number of digits that

“roll over” to create the next word. In particular, the *binary ruler sequence* $\text{ruler}(2, 2, \dots, 2)$ (OEIS A001511 [23]) gives the suffix lengths of the form $011 \dots 1$ that change to $100 \dots 0$ when counting in binary. This is shown below for $n = 3$.

$\text{lex}(B_3) = 00\bar{0}, 00\bar{1}, 01\bar{0}, 0\bar{1}\bar{1}, 10\bar{0}, 10\bar{1}, 11\bar{0}, 111$ since $\text{ruler}(2, 2, 2) = 1, 2, 1, 3, 1, 2, 1$

The *upstairs ruler sequence* $\text{ruler}(1, 2, \dots, n)$ (OEIS A235748) arises when listing *upstairs words* $W_{1,2,\dots,n}$, and the *downstairs ruler sequence* $\text{ruler}(n, n-1, \dots, 1)$ (OEIS A001511) arise when listing *downstairs words* $W_{n,n-1,\dots,1}$. The start of these *factorial patterns* are below for $n = 4$ with full signed versions in Table 1.

$\text{lex}(W_{1,2,3,4}) = 111\bar{1}, 111\bar{2}, 11\bar{1}\bar{3}, 112\bar{1}, 112\bar{2}, 1\bar{1}\bar{2}\bar{3}, 1211, \dots$ as $\text{ruler}(1, 2, 3, 4) = 1, 1, 2, 1, 1, 3, \dots$

$\text{lex}(W_{4,3,2,1}) = 11\bar{1}\bar{1}, 1\bar{1}\bar{2}\bar{1}, 12\bar{1}\bar{1}, 122\bar{1}, 13\bar{1}\bar{1}, \bar{1}32\bar{1}, 2111, \dots$ as $\text{ruler}(4, 3, 2, 1) = 2, 3, 2, 3, 2, 4, \dots$

Note that the *unary bases* $b_n = 1$ (which never changes) and $b_1 = 1$ (which always rolls over to itself) are often omitted from these patterns.

Ruler sequences provide change sequences for various Gray codes, including some from Sect. 1. The downstairs sequence gives the flip lengths in Zaks’ order as seen in Table 1. Corbett’s order uses the upstairs sequence but subtly [36]. Other change sequences are more fully understood as signed ruler sequences.

4.2 Signed Ruler Sequences and (Reflected) Gray Codes

We define the *signed ruler sequence* $\text{ruler} \pm$ as ruler with some entries negated. The overlines complement the sign of each entry, and the R reverses a sequence.

$$\text{ruler} \pm (b_1) = 1^{b_1-1} = 1, 1, \dots, 1 \quad (\text{i.e., } b_1 - 1 \text{ copies}) \quad (9)$$

$$\text{ruler} \pm (b_n, b_{n-1}, \dots, b_1) = \begin{cases} (s, n, \bar{s}^R, n)^{b_n/2}, s & \text{if } b_n \text{ is odd} \\ (s, n, \bar{s}^R, n)^{(b_n-1)/2}, s, n, \bar{s} & \text{if } b_n \text{ is even} \end{cases} \quad (10)$$

where $s = \text{ruler} \pm (b_{n-1}, b_{n-2}, \dots, b_1)$. Note that the subsequence s is repeated b_n times in (10) just as in (8), but every second subsequence is complemented⁶. For example, $\text{ruler} \pm (3) = 1, 1$ so $\text{ruler} \pm (4, 3) = 1, 1, 2, -1, -1, 2, 1, 1, 2, -1, -1$. The specific sequences (and associated orders) discussed below are shown in Table 1.

Signed ruler sequences govern *reflected mixed-radix Gray codes*, which generalize (1) to non-binary bases $\mathbf{b} = b_n, b_{n-1}, \dots, b_1$ by reflecting every 2nd sublist,

$$\text{mix}(\mathbf{b}) = \begin{cases} 0, 1, \dots, b_1-1 & \text{if } n = 1 \\ 0 \cdot \text{mix}(\mathbf{b}'), 1 \cdot \text{reflect}(\mathbf{b}'), \dots, (b_n-1) \cdot \text{mix}(\mathbf{b}') & \text{odd } n > 1 \\ 0 \cdot \text{mix}(\mathbf{b}'), 1 \cdot \text{reflect}(\mathbf{b}'), \dots, (b_n-1) \cdot \text{reflect}(\text{mix}(\mathbf{b}')) & \text{even } n > 1 \end{cases}$$

where $\mathbf{b}' = b_{n-1}, b_{n-2}, \dots, b_1$. The entries of $\text{ruler} \pm (\mathbf{b})$ specify how to change $w_n w_{n-1} \dots w_1 \in W_{\mathbf{b}}$ into the next word: increment w_j for $+j$; decrement w_j for $-j$. The orders are also greedy: increment or decrement the rightmost digit.

⁶ Unsigned ruler sequences are palindromes, so R ’s can be added to (8) to mirror (10).

Table 1. Ruler sequences provide the change sequences of reflected Gray codes of mixed-radix words, and (greedy) Gray codes of various other objects. The left columns show that the unsigned downstairs ruler sequence $\text{ruler}(n, n-1, \dots, 1)$ is the change sequence for the up-words $W_{4,3,2,1}$, and the prefix-reversal lengths (i.e., flip lengths) in Zaks' Gray code. The change sequences of the binary reflected Gray code and plain changes are usually given as unsigned ruler sequences. However, signed versions provide more information. The middle-left columns show that the signed binary ruler sequence $\text{ruler} \pm (2, 2, \dots, 2)$ is the change sequence for $\text{brgc}(n)$, with the sign providing the direction of the flip: $+j$ for $b_j = \bar{0} = 1$ and $-j$ for $b_j = \underline{1} = 0$. Similarly, the middle-right columns show that the signed upstairs ruler sequence $\text{ruler} \pm (1, 2, \dots, n)$ is the change sequence for $\text{plain}(n)$, with the sign providing the direction of the swap: $+j$ for swapping x left and $-j$ for swapping x right where $x = n-j+1$. Our twisted plain change Gray code $\text{twisted}(n)$ uses a signed factorial ruler sequence $\text{ruler} \pm (n, n-1, \dots, 2, 1, 2, 2, \dots, 2)$ (with the unary 1 omitted). Sequence entries from the factorial and binary portions give 2-twists and 1-twists, respectively. In particular, the last row in the right columns is the 1-twist in Fig. 3.

down	ruler	Zaks	ruler \pm	BRGC	up	ruler \pm	plain	up \pm	ruler \pm	twisted
words	4321	$p_4p_3p_2p_1$	2222	$b_4b_3b_2b_1$	words	1234	changes	words	22221234	plain
00 $\bar{0}$ 0	2	$\overrightarrow{1234}$	+1	000 $\bar{0}$	00 $\bar{0}$	+1	$\overleftarrow{1234}$	000000 $\bar{0}$	+1	$\overleftarrow{1234}$
0 $\bar{0}$ 10	3	$\overrightarrow{2134}$	+2	00 $\bar{0}$ 1	00 $\bar{1}$	+1	$\overrightarrow{1243}$	000000 $\bar{1}$	+1	$\overrightarrow{1243}$
01 $\bar{1}$ 0	2	$\overrightarrow{3124}$	-1	001 $\underline{1}$	00 $\bar{2}$	+1	$\overleftarrow{1423}$	000000 $\bar{2}$	+1	$\overleftarrow{1423}$
0 $\bar{1}$ 00	3	$\overrightarrow{1324}$	+3	0 $\bar{0}$ 10	0 $\bar{0}$ 3	+2	$\overleftarrow{4123}$	000000 $\bar{3}$	+2	$\overleftarrow{4123}$
02 $\bar{0}$ 0	2	$\overrightarrow{2314}$	+1	011 $\bar{0}$	01 $\underline{3}$	-1	$\overleftarrow{4132}$	000001 $\underline{3}$	-1	$\overleftarrow{4132}$
$\bar{0}$ 210	4	$\overrightarrow{3214}$	-2	011 $\underline{1}$	01 $\underline{2}$	-1	$\overrightarrow{1432}$	000001 $\underline{2}$	-1	$\overrightarrow{1432}$
12 $\bar{1}$ 0	2	$\overrightarrow{4123}$	-1	010 $\underline{1}$	01 $\underline{1}$	-1	$\overrightarrow{1342}$	000001 $\underline{1}$	-1	$\overrightarrow{1342}$
1 $\bar{2}$ 00	3	$\overrightarrow{1423}$	+4	$\bar{0}$ 100	0 $\bar{1}$ 0	+2	$\overrightarrow{1324}$	00000 $\bar{1}$ 0	+2	$\overrightarrow{1324}$
11 $\bar{0}$ 0	2	$\overrightarrow{2413}$	+1	110 $\bar{0}$	02 $\bar{0}$	+1	$\overleftarrow{3124}$	000002 $\bar{0}$	+1	$\overleftarrow{3124}$
1 $\bar{1}$ 10	3	$\overrightarrow{4213}$	+2	11 $\bar{0}$ 1	02 $\bar{1}$	+1	$\overleftarrow{3142}$	000002 $\bar{1}$	+1	$\overleftarrow{3142}$
10 $\bar{1}$ 0	2	$\overrightarrow{1243}$	-1	111 $\underline{0}$	02 $\bar{2}$	+1	$\overleftarrow{3412}$	000002 $\bar{2}$	+1	$\overleftarrow{3412}$
$\bar{1}$ 000	4	$\overrightarrow{2143}$	-3	1 $\underline{1}$ 10	$\bar{0}$ 23	+3	$\overleftarrow{4312}$	0000 $\bar{0}$ 23	+3	$\overleftarrow{4312}$
20 $\bar{0}$ 0	2	$\overrightarrow{3412}$	+1	101 $\bar{0}$	12 $\underline{3}$	-1	$\overleftarrow{4321}$	000012 $\underline{3}$	-1	$\overleftarrow{4321}$
2 $\bar{0}$ 10	3	$\overrightarrow{4312}$	-2	101 $\underline{1}$	12 $\underline{2}$	-1	$\overleftarrow{3421}$	000012 $\underline{2}$	-1	$\overleftarrow{3421}$
21 $\bar{1}$ 0	2	$\overrightarrow{1342}$	-1	100 $\underline{1}$	12 $\underline{1}$	-1	$\overleftarrow{3241}$	000012 $\underline{1}$	-1	$\overleftarrow{3241}$
2 $\bar{1}$ 00	3	$\overrightarrow{3142}$			12 $\underline{0}$	-2	$\overrightarrow{3214}$	000012 $\underline{0}$	-2	$\overrightarrow{3214}$
22 $\bar{0}$ 0	2	$\overrightarrow{4132}$			11 $\bar{0}$	+1	$\overrightarrow{2314}$	000011 $\bar{0}$	+1	$\overrightarrow{2314}$
$\bar{2}$ 210	4	$\overrightarrow{1432}$			11 $\bar{1}$	+1	$\overrightarrow{2341}$	000011 $\bar{1}$	+1	$\overrightarrow{2341}$
32 $\bar{1}$ 0	2	$\overrightarrow{2341}$			11 $\bar{2}$	+1	$\overrightarrow{2431}$	000011 $\bar{2}$	+1	$\overrightarrow{2431}$
3 $\bar{2}$ 00	3	$\overrightarrow{3241}$			11 $\underline{3}$	-2	$\overleftarrow{4231}$	000011 $\underline{3}$	-2	$\overleftarrow{4231}$
31 $\bar{0}$ 0	2	$\overrightarrow{4231}$			10 $\underline{3}$	-1	$\overleftarrow{4213}$	000010 $\underline{3}$	-1	$\overleftarrow{4213}$
3 $\bar{1}$ 10	3	$\overrightarrow{2431}$			10 $\underline{2}$	-1	$\overleftarrow{2413}$	000010 $\underline{2}$	-1	$\overleftarrow{2413}$
30 $\bar{1}$ 0	2	$\overrightarrow{3421}$			10 $\underline{1}$	-1	$\overleftarrow{2143}$	000010 $\underline{1}$	-1	$\overleftarrow{2143}$
3000		4321			100		2134	0000 $\bar{1}$ 00	+5	$\overrightarrow{2134}$
								0001100	...	2134

The binary reflected Gray code $\text{brgc}(n)$ is the special case where the *signed binary sequence ruler* $\pm(2, 2, \dots, 2)$ (OEIS A164677) gives bit increments and decrements. More interestingly, $\text{plain}(n)$ follows the *signed upstairs sequence ruler* $\pm(1, 2, \dots, n)$: $+j$ swaps value $n-j+1$ left; $-j$ swaps value $n-j+1$ right⁷.

A *signed basis* \mathbf{b} contains $1, 2, \dots, n$ plus n copies of 2. Note that $|\text{ruler}\pm(\mathbf{b})| = 2^n n! - 1 = |S_n^\pm| - 1$. The *twisted basis* concatenates the signed binary and signed upstairs bases to give the *twisted ruler sequence ruler* $\pm(2, 2, \dots, 2, 1, 2, \dots, n)$.

Lemma 3. *A change sequence for twisted(n) is ruler $\pm(2, 2, \dots, 2, 1, 2, \dots, n)$: $+j$ and $-j$ respectively 2-twist value $n-j+1$ to the left and right for $1 \leq |j| \leq n$; $+j$ and $-j$ respectively 1-twist (flip) value $n-j+1$ down and up for $n < |j| \leq 2n$.*

Now we can looplessly generate twisted plain changes $\text{twisted}(n)$ by looplessly generating the twisted ruler sequence $\text{ruler}\pm(2, 2, \dots, 2, 1, 2, \dots, n)$ and its changes.

5 Loopless Generation of Gray Codes via Ruler Sequences

The greedy algorithm for $\text{twisted}(n)$ in Sect. 2.3 is simple but inefficient. It requires exponential space, as all previously created objects must be remembered. Fortunately, greedy Gray codes can often be generated without remembering previous objects [21, 28, 29]. The loopless *history-free* implementation that we provide here uses a signed ruler sequence to generate the changes. Loopless algorithms for non-greedy Gray codes also exist using ruler sequence changes [8, 15].

Algorithm 2 has procedures for generating Gray codes whose changes follow a ruler sequence with any bases \mathbf{b} . The start object is \mathbf{s} and the change functions are in \mathbf{fns} . The ruler sequence is generated one entry at a time, and the current object is updated and visited accordingly. More specifically, if j is the next entry, then $\mathbf{fns}[j]$ is applied to \mathbf{s} to create the next object. The pseudocode is adapted from Knuth's loopless reflected mixed-radix Gray code Algorithm M [17].

Algorithm 2 can looplessly generate various Gray codes in this paper. As a simple example, Zaks' pancake order uses RulerGrayCode with $\mathbf{b} = 2, 3, \dots, n$, $\mathbf{s} = 12 \dots n$, and $\mathbf{fns} = \overleftarrow{r_1}, \overleftarrow{r_2}, \dots, \overleftarrow{r_n}$ where $\overleftarrow{r_i}$ reverses the prefix of length i . The $\text{brgc}(n)$ can be looplessly generated using RulerGrayCode or $\text{RulerGrayCode}\pm$. When generating $\text{plain}(n)$ with $\text{RulerGrayCode}\pm$ we maintain the inverse of the current permutation in order to swap a specific value left or right in $\mathcal{O}(1)$ -time. Maintaining the inverse is also required to looplessly generate our new order.

⁷ Surprisingly, this sequence is not yet in the *Online Encyclopedia of Integer Sequences*, nor is the *signed downstairs sequence ruler* $\pm(n, n-1, \dots, 2) = \text{ruler}\pm(n, n-1, \dots, 1) - 1 = 1, 2, -1, 2, 1, 3, -1, -2, 1, -2, -1, 3, 1, 2, -1, 2, 1, 3, -1, -2, 1, -2, -1, 4, \dots$

Algorithm 2. Generating Gray codes using ruler sequences with bases \mathbf{b} . The \mathbf{fns} modify object \mathbf{s} and are indexed by the sequence. For example, if $\mathbf{b} = 3, 2$ then $\text{RulerGrayCode}\pm(\mathbf{b})$ visits $\text{ruler}\pm(2, 3) = 1, 1, 2, -1, -1$ alongside a Gray code that starts \mathbf{s} and applies \mathbf{fns} with indices $1, 1, 2, -1, -1$. The signed version also generates the reflected mixed-radix Gray code $\text{mix}(\mathbf{b})$ in \mathbf{a} , with the \mathbf{d} values providing ± 1 directions of change. So in the previous example the mixed-radix words $\bar{0}0, \bar{1}0, 2\bar{0}, \underline{2}1, \underline{1}1, 10$ are generated in \mathbf{a} . Focus pointers are stored in \mathbf{f} . The overall algorithm is loopless if each function runs in worst-case $\mathcal{O}(1)$ -time. Note that the indexing is reversed with respect to Sect. 4 with $\mathbf{b} = b_1, b_2, \dots, b_n$. Unary bases should be omitted: $b_i \geq 2$ is required for $0 \leq i < n$.

1: procedure $\text{RulerGrayCode}(\mathbf{b}, \mathbf{s}, \mathbf{fns})$	1: procedure $\text{RulerGrayCode}\pm(\mathbf{b}, \mathbf{s}, \mathbf{fns})$
2: $a_1 a_2 \dots a_n \leftarrow 0\ 0 \dots 0$	2: $a_1 a_2 \dots a_n \leftarrow 0\ 0 \dots 0$
3: $f_1 f_2 \dots f_{n+1} \leftarrow 1\ 2 \dots n+1$	3: $f_1 f_2 \dots f_{n+1} \leftarrow 1\ 2 \dots n+1$
4:	4: $d_1 d_2 \dots d_n \leftarrow 1\ 1 \dots 1$
5: $\text{visit}(\mathbf{s})$	5: $\text{visit}(\mathbf{s})$
6: while $f_1 \leq n$ do	6: while $f_1 \leq n$ do
7: $j \leftarrow f_1$	7: $j \leftarrow f_1$
8: $f_1 \leftarrow 1$	8: $f_1 \leftarrow 1$
9: $a_j \leftarrow a_j + 1$	9: $a_j \leftarrow a_j + d_j$
10: $\mathbf{s} \leftarrow \mathbf{fns}[j](\mathbf{s})$	10: $\mathbf{s} \leftarrow \mathbf{fns}[d_j \cdot j](\mathbf{s})$
11: $\text{visit}(j, \mathbf{s})$	11: $\text{visit}(d_j \cdot j, \mathbf{s})$
12: if $a_j = b_j - 1$ then	12: if $a_j \in \{0, b_j - 1\}$ then
13: $a_j \leftarrow 0$	13: $d_j \leftarrow -d_j$
14: $f_j \leftarrow f_{j+1}$	14: $f_j \leftarrow f_{j+1}$
15: $f_{j+1} \leftarrow j + 1$	15: $f_{j+1} \leftarrow j + 1$

Theorem 2. *Twisted plain changes $\text{twisted}(n)$ and its change sequence are generated looplessly by $\text{RulerGrayCode}\pm(\mathbf{b}, \mathbf{s}, \mathbf{fns})$ with twisted bases \mathbf{b} , the positive identity permutation $\mathbf{s} \in S_n^\pm$, and the change functions \mathbf{fns} given in Lemma 3.*

6 Final Remarks

Alternate Gray codes for signed permutations can be generated using other signed ruler sequences, and some of these generalize to colored permutations [25]. For additional new results involving greedy Gray codes see Merino and Mütze [20].

Open question: Does S_n^\pm have a doubly-adjacent Gray code [2] using twists? We thank the reviewers for their helpful comments, proofreading, and debugging.

A Python Implementation

A loopless implementation of our signed plain change order `twisted(n)` in Python 3. Entries in the twisted ruler sequence $\text{ruler}_{\pm}(n, n-1, \dots, 2, 1, 2, 2, \dots, 2)$ select the 2-twist or 1-twist (i.e., flip) to apply⁸. 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[abs(v)] # Use inverse to get the position of value v.
    u = 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[abs(v)], q[abs(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.

```

⁸ Negative indices give right-to-left access in Python. So the ruler entry `-1` selects the last function `fns[-1] = twist(p,q,n,1)` (i.e., 2-twist n right). Notes: `v=v` is for binding; slice notation `[-1::-1]` reverses a list; indices are reversed from Sect. 4.

References

1. Cameron, B., Sawada, J., Therese, W., Williams, A.: Hamiltonicity of k -sided pancake networks with fixed-spin: efficient generation, ranking, and optimality. *Algorithmica* **85**(3), 717–744 (2023)
2. Compton, R.C., Gill Williamson, S.: Doubly adjacent Gray codes for the symmetric group. *Linear Multilinear Algebra* **35**(3–4), 237–293 (1993)
3. Corbett, P.F.: Rotator graphs: an efficient topology for point-to-point multiprocessor networks. *IEEE Trans. Parallel Distrib. Syst.* **3**(5), 622–626 (1992)
4. Duato, J., Yalamanchili, S., Ni, L.: *Interconnection Networks*. Morgan Kaufmann, Burlington (2003)
5. Duckworth, R., Stedman, F.: *Tintinnalogia: Or, The Art of Ringing*. London (1668)
6. Ehrlich, G.: Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. ACM* **20**(3), 500–513 (1973)
7. Fertin, G., Labarre, A., Rusu, I., Vialette, S., Tannier, E.: *Combinatorics of Genome Rearrangements*. MIT Press, Cambridge (2009)
8. Ganapathi, P., Chowdhury, R.: A unified framework to discover permutation generation algorithms. *Comput. J.* **66**(3), 603–614 (2023)
9. Gardner, M.: Curious properties of the Gray code and how it can be used to solve puzzles. *Sci. Am.* **227**(2), 106 (1972)
10. Gray, F.: Pulse code communication. United States Patent Number 2632058 (1953)
11. Hannenhalli, S., Pevzner, P.A.: Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. In: *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC 1995)*, pp. 178–189. ACM (1995)
12. Hartung, E., Hoang, H., Mütze, T., Williams, A.: Combinatorial generation via permutation languages. I. fundamentals. *Trans. Am. Math. Soc.* **375**(04), 2255–2291 (2022)
13. Heath, F.: Origins of the binary code. *Sci. Am.* **227**(2), 76–83 (1972)
14. Hindenburg, C.F.: *Sammlung combinatorisch-analytischer Abhandlungen*, vol. 1. ben Gerhard Fleischer dem Jungern (1796)
15. Holroyd, A.E., Ruskey, F., Williams, A.: Shorthand universal cycles for permutations. *Algorithmica* **64**, 215–245 (2012)
16. Johnson, S.M.: Generation of permutations by adjacent transposition. *Math. Comput.* **17**(83), 282–285 (1963)
17. Knuth, D.E.: *Art of Computer Programming, Volume 4, Fascicle 4, The: Generating All Trees-History of Combinatorial Generation*. Addison-Wesley, Boston (2013)
18. Korsh, J., LaFollette, P., Lipschutz, S.: A loopless implementation of a Gray code for signed permutations. *Publications de l’Institut Mathématique* **89**(103), 37–47 (2011)
19. Liptak, Z., Masillo, F., Navarro, G., Williams, A.: Constant time and space updates for the sigma-tau problem. In: Nardini, F.M., Pisanti, N., Venturini, R. (eds.) *SPIRE 2023. LNCS*, vol. 14240, pp. 323–330. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-43980-3_26
20. Merino, A., Mutze, T.: Traversing combinatorial 0/1-polytopes via optimization. In: *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 1282–1291 (2023)
21. Merino, A., Mutze, T., Williams, A.: All your bases are belong to us: listing all bases of a matroid by greedy exchanges. In: *11th International Conference on Fun*

- with Algorithms (FUN 2022), vol. 226, p. 22. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2022)
22. Mütze, T.: Combinatorial Gray codes-an updated survey. arXiv preprint [arXiv:2202.01280](https://arxiv.org/abs/2202.01280) (2022)
 23. OEIS Foundation Inc.: The On-Line Encyclopedia of Integer Sequences (2023). <http://oeis.org>
 24. Ord-Smith, R.: Generation of permutation sequences: part 1. *Comput. J.* **13**(2), 152–155 (1970)
 25. Qiu, Y.F.: Greedy and speedy: new iterative gray code algorithms. Bachelor’s thesis, Williams College (2024)
 26. Ruskey, F.: Combinatorial generation. Preliminary working draft. University of Victoria, Victoria, BC, Canada 11, 20 (2003)
 27. Savage, C.: A survey of combinatorial Gray codes. *SIAM Rev.* **39**(4), 605–629 (1997)
 28. Sawada, J., Williams, A.: Greedy flipping of pancakes and burnt pancakes. *Discret. Appl. Math.* **210**, 61–74 (2016)
 29. Sawada, J., Williams, A.: Successor rules for flipping pancakes and burnt pancakes. *Theoret. Comput. Sci.* **609**, 60–75 (2016)
 30. Sedgewick, R.: Permutation generation methods. *ACM Comput. Surv. (CSUR)* **9**(2), 137–164 (1977)
 31. Steinhaus, H.: One hundred problems in elementary mathematics. Courier Corporation (1979)
 32. Stigler, S.M.: Stigler’s law of eponymy. *Trans. New York Acad. Sci.* **39**(1 Series II), 147–157 (1980)
 33. Suzuki, Y., Sawada, N., Kaneko, K.: Hamiltonian cycles and paths in burnt pancake graphs. In: Proceedings of the ISCA 18th International Conference on Parallel and Distributed Computing Systems, pp. 85–90 (2005)
 34. Trotter, H.F.: Algorithm 115: perm. *Commun. ACM* **5**(8), 434–435 (1962)
 35. Williams, A.: $O(1)$ -time unsorting by prefix-reversals in a boustrophedon linked list. In: Boldi, P., Gargano, L. (eds.) FUN 2010. LNCS, vol. 6099, pp. 368–379. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13122-6_35
 36. Williams, A.: The greedy Gray code algorithm. In: Dehne, F., Solis-Oba, R., Sack, J.R. (eds.) WADS 2013. LNCS, vol. 8037, pp. 525–536. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40104-6_46
 37. Williams, A.: Signed-plain-changes (2024). <https://gitlab.com/combinatronics/signed-plain-changes>
 38. Zaks, S.: A new algorithm for generation of permutations. *BIT Numer. Math.* **24**(2), 196–204 (1984)

Maximize the Rightmost Digit: Gray Codes for Restricted Growth Strings

Yuan (Friedrich) Qiu¹, Joe Sawada², and Aaron Williams¹[0000–0001–6816–4368]

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

Abstract. The term *restricted growth string* typically refers to strings of non-negative integers $a_1 a_2 \cdots a_n$ (with $a_1 = 0$) in which the next symbol is at most one more than the maximum of the previous symbols: $0 \leq a_i \leq \max(a_1 \cdots a_{i-1}) + 1$ for $2 \leq i \leq n$. These strings are counted by the Bell numbers \mathcal{B}_n (OEIS A000110) and encode set partitions. Kerr showed that the following greedy algorithm generates a Gray code starting from 0^n : greedily maximize the rightmost symbol that creates a new string (e.g., 000, 001, 011, 012, 010 for $n = 3$). This turns out to be a special case of a more general result for **e**-restricted growth functions Mansour, Nassar, and Vajnovszki's, although those authors did not describe their order greedily. Here we generalize these results to a much broader class of restricted growth strings: $0 \leq a_1 \leq s-1$ and $0 \leq a_i \leq f(a_1 a_2 \cdots a_{i-1}) + c_i$ where f is any function with $f \geq 0$ and $c_i \geq 1$ are constants for each digit. In each case, the orders change a single digit by -1 or -2 (cyclically). Special cases include the binary reflected Gray code ($s = 2$, $f = 0$, $c = 1$) and the aforementioned orders for $\mathbf{c} = \mathbf{e}$. We also introduce a new type of restricted growth string counted by the k -Catalan numbers and provide a loopless algorithm for generating them.

Keywords: restricted growth strings · Bell numbers · set partitions · Catalan numbers · Gray codes · greedy Gray codes.

1 Introduction

In this paper, we are interested in efficiently ordering and generating various types of restricted growth strings. We begin by describing two of the most common types of restricted growth strings, along with their applications.

1.1 Bell and Catalan Strings

The term *restricted growth string* is often defined as a string of integers (called *digits*) of the form $a_1 a_2 \cdots a_n$ that satisfies the following conditions,

$$a_1 = 0 \text{ and } 0 \leq a_i \leq \max(a_1 a_2 \cdots a_{i-1}) + 1 \text{ for } 2 \leq i \leq n. \quad (1)$$

In other words, the first digit is 0, and each subsequent digit is at least 0 and at most one more than the maximum of the previous digits. The number of restricted growth strings of length $n \geq 0$ is the n^{th} Bell number \mathcal{B}_n .

$$1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, \dots \quad \text{OEIS A000110 [8]}. \quad (2)$$

Since they are enumerated by the Bell numbers, we refer to this type of restricted growth string as *Bell strings*. Bell strings provide a convenient representation for the set partitions of $[n] = \{1, 2, \dots, n\}$, which are also Bell objects. The standard bijection puts i into the a_i^{th} part, as shown below for $n \leq 3$ [22].

0	00	01	000	001	010	011	012
{1}	{1,2}	{1},{2}	{1,2,3}	{1,2},{3}	{1,3},{2}	{1},{2,3}	{1},{2},{3}

Note that a small change in a set partition can lead to a large change in its Bell string. For example, the set partition $\{1, 2\}, \{3\}, \{4\}, \dots, \{n\}$ corresponds to the Bell string $00123 \cdots (n-2)$. If we move the 2 into its own subset to create the set partition of singletons, then the corresponding Bell string becomes $0123 \cdots (n-1)$ (i.e., all digits change except the leading 0). On the other hand, changing a single digit in a Bell string always corresponds to moving a single value in its set partition. For this reason, when designing efficient orders of set partitions it can be preferable to instead work with Bell strings.

Perhaps the most well-known ordering of set partitions was created by Knuth and presented by Kaye [9]. Later work by Ruskey and Savage [19] adapted the approach to Bell strings. A student project by Kerr [10] provided an alternate ordering of Bell strings (see [17]) that uses a greedy approach [25]. This paper generalizes Kerr's result from Bell strings to other restricted growth strings.

Another interesting type of restricted growth string is obtained by modifying (1)

$$a_1 = 0 \text{ and } 0 \leq a_i \leq a_{i-1} + 1 \text{ for } 2 \leq i \leq n. \quad (3)$$

The number of *Catalan strings* of length $n \geq 0$ is the n^{th} Catalan number \mathcal{C}_n

$$1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \dots \quad \text{OEIS A000108.} \quad (4)$$

We let $\mathbf{B}(n)$ and $\mathbf{C}(n)$ be the sets of Bell and Catalan strings of length n , respectively. Figure 3 has lists of $\mathbf{B}(n)$ and $\mathbf{C}(n)$ for $n \leq 4$. In particular, the reader can confirm the following: $\mathbf{B}(4) \setminus \mathbf{C}(4) = \{0102\}$.

Catalan strings provide an alternate representation for the large number of other Catalan objects counted by \mathcal{C}_n [23]. We'll also provide a simple generalization to k -Catalan strings $\mathbf{C}_k(n)$ in Section 2. There are several other types of strings counted by Catalan and k -Catalan numbers (e.g., see [26]).

1.2 Generalized Restricted Growth Strings

Although the term restricted growth string often refers specifically to Bell strings, it is also used much more broadly in the literature. Here we consider a generalization that allows for flexibility in the first digit, the function applied to the previous digits, and the constant added to each digit. Formally, an (s, f, \mathbf{c}) -restricted growth string is a string of integers of the form $a_1 a_2 \cdots a_n$ satisfying

$$0 \leq a_1 \leq s - 1 \text{ and } 0 \leq a_i \leq f(a_1 a_2 \cdots a_{i-1}) + c_i \text{ for } 2 \leq i \leq n \quad (5)$$

with $s \geq 1$, $f \geq 0$, and $c \geq 1$. In other words, the *starting digit* a_1 has s possible values $0 \leq a_1 \leq s$, while each subsequent digit a_i is a non-negative integer

limited by the sum of a *function* f that maps the previous digits $a_1 a_2 \cdots a_{i-1}$ to a non-negative integer, and a positive integer *constant* c_i that depends only on the index i . (For notational convenience, we write $c = w$ and $f = w$ when $c_i = w$ and $f(a_1 a_2 \cdots a_{i-1}) = w$ for all $2 \leq i \leq n$, respectively.)

This generalization captures a wide variety of previously studied strings, including those discussed in Section 1.1. Binary strings are obtained with $s = 2$, $f = 0$, and $c = 1$, while mixed-radix strings with bases b_1, b_2, \dots, b_n (see Section 3.3) are obtained with $s = b_1$, $f = 0$, and $c_i = b_i - 1$ for $2 \leq i \leq n$. Additional special cases of (s, f, \mathbf{c}) -restricted growth strings and related previous results include the following.

- *Restricted growth tails* [19] are modeled by using different values of s .
- *Restricted growth strings of order d* are obtained by replacing $+1$ in (1) with $+(d-1)$. More broadly, the *e-restricted growth functions* put forward by Mansour, Nassar and Vajnovszki [14] are modeled by our constants \mathbf{c} . Their paper also uses the greedy max-right order, although it does not identify it as such. In particular, their $\text{succ}_{1,m}$ and $\text{succ}_{2,m}$ lists mirror our g_0 and g_1 expansions (see Section 4).
- *Prefix statistics* (e.g., ascents) were considered by Mansour and Vajnovszki [15] and Sabri and Vajnovszki [20] and are modeled by our functions f . By carefully choosing the ‘statistic’ and ignoring the first digit of 0, the *st*-restricted strings concept in [15] can be used to model (s, f, \mathbf{c}) -RGS strings.
- *K-increment strings* replace $+1$ with $+i$ in (3) and are modeled by \mathbf{c} . Arndt efficiently generates these strings and other restricted growth strings [1].

1.3 Goals and Results

We are interested in creating Gray codes for restricted growth strings. That is, we want to list these sets so that consecutive strings differ in a small constant amount. Furthermore, we want to generate these lists efficiently. In this context, *constant amortized time (CAT)* and *loopless* algorithms generate successive strings in amortized and worst-case $O(1)$ -time, respectively.

An initial roadblock is that Bell strings do not have a ± 1 Gray code [18]. In other words, it is not possible to order the strings in an arbitrary $\mathbf{B}(n)$ so that consecutive strings differ in only one digit, and such that the differing values are within ± 1 of each other. However, Ehrlich [4] constructed a Gray code for $\mathbf{B}(n)$ in which a single digit changes by ± 1 when considered cyclically¹ and provided a loopless implementation. On the other hand, Ruskey [18] created a CAT algorithm that allows ± 1 and ± 2 non-cyclically². Li and Sawada provided a Gray code for $\mathbf{B}(n)$ as part of their *reflectable languages* framework [13], and their special values $x = 0$ and $y = 1$ arise naturally in our results.

¹ A flexible notion of cyclicity is used: $a_i = 0 \leftrightarrow a_i = m$ and $a_i = 0 \leftrightarrow a_i = m + 1$ are allowed when $m = \max(a_1 a_2 \cdots a_{i-1})$.

² This order was also mentioned in a paper by Ruskey and Savage [19], however, the two descriptions are not equivalent.

Our goal is to present an approach to generating restricted growth string Gray codes with the following benefits:

- (a) The approach is very easy to describe.
- (b) The approach generalizes previous results.
- (c) The approach works for all (s, f, \mathbf{c}) -restricted growth strings.
- (d) The approach leads to loopless generation algorithms.

We reach our goals using an approach that can be summarized in one sentence: start a list with 0^n then repeatedly extend it to a new string by greedily changing the rightmost digit to the maximum possible value. We refer to this approach as the *max-right algorithm*, and we note that “allowable” depends on which type of string is being generated. As we’ll see, successive strings in the resulting *max-right orders* differ in a single digit by -1 or -2 (cyclically). In particular, the change from 0 to the maximum possible value is -1 taken cyclically, and 1 to the maximum value is -2 taken cyclically. Moreover, we provide loopless implementations and applications for $\mathbf{B}(n)$ and $\mathbf{C}_k(n)$. We also obtain the ‘original’ Gray code for n -bit binary strings using $s = 2$, $f = 0$, and $c = 1$.

1.4 Outline

Section 2 introduces k -Catalan strings and proves that they are an example of (s, f, \mathbf{c}) -restricted growth strings. Section 3 discusses Gray codes and combinatorial generation. Section 4 provides our Gray codes for (s, f, \mathbf{c}) -restricted growth strings. Section 5 provides new loopless algorithms for mixed-radix, k -Catalan, and Bell strings. The appendix includes additional figures and Python implementations of our algorithms.

2 k -Catalan Strings

In this section we provide a natural generalizations of Catalan strings, and prove that they are an example of (s, f, \mathbf{c}) -restricted growth strings. Our generalization of Catalan strings comes from replacing $+1$ with $+(k-1)$ in (3) as follows.

$$a_1 = 0 \text{ and } 0 \leq a_i \leq a_{i-1} + (k-1) \text{ for } 2 \leq i \leq n. \quad (6)$$

We refer to the resulting strings as *k-Catalan strings* and let $\mathbf{C}_k(n)$ be the set of length n . For example, when $n = 3$ and $k = 3$ we have

$$\mathbf{C}_3(3) = \{000, 001, 002, 010, 011, 012, 013, 020, 021, 022, 023, 024\} \quad (7)$$

and these $|\mathbf{C}_3(3)| = 12$ strings are in bijection with the ternary trees with 3 internal nodes. The Catalan strings $\mathbf{C}(n)$ defined in (3) are obtained from (6) with $k = 2$, and Catalan numbers are also known as 2-Catalan numbers \mathcal{C}_n . More broadly, $\mathcal{C}_{k,n}$ denotes the n th k -Catalan number, and the OEIS sequences A000108, A001764, A002293, A002294, A002295 are for $2 \leq k \leq 6$.

Theorem 1. $|\mathbf{C}_k(n)| = \mathcal{C}_{k,n}$ for all $n \geq 0$ and $k \geq 2$.

Proof. We prove that the members of $\mathbf{C}_k(n)$ are in bijective correspondence with the k -ary trees with n internal nodes, which are known to be counted by $\mathcal{C}_{k,n}$. Note that there is a single k -ary tree with one internal node and that $\mathbf{C}_k(n) = \{0\}$, so the result is true for $n = 1$ and all $k \geq 2$. Suppose the result is true for $n = t$ and all $k \geq 2$. Now we consider how the strings and trees can be extended to $n = t + 1$. Every string in $\mathbf{C}_k(t)$ that ends with the digit d is the prefix of $d + (k - 1)$ strings in $\mathbf{C}_k(t + 1)$. Next consider a k -ary tree with t internal nodes and label its nodes according to a preorder traversal. Consider the location of the node x that is last in preorder (i.e., it has label t). To extend this tree without changing the preorder traversal we can either add the new node as a child of x or as the right-child of any node on the path from the root to x that doesn't have a right-child. In this way, if x was in the d th rightmost leaf in the tree, then the added node can be in any of $d + (k - 1)$ positions.

Theorem 2. The set of k -Catalan strings $\mathbf{C}_k(n)$ are an example of (s, f, \mathbf{c}) -restricted growth strings.

Proof. We claim this is true from $s = 1$, $f(a_1 a_2 \cdots a_{i-1}) = a_{i-1}$, and $c_i = k - 1$ for all $2 \leq i \leq n$. This follows from (5) as these choices force $0 \leq a_1 = s - 1 = 0$ (i.e., $a_1 = 0$) and the following bound for $2 \leq i \leq n$ that matches (6),

$$0 \leq a_i \leq f(a_1 a_2 \cdots a_{i-1}) + c_i = a_{i-1} + (k - 1). \quad (8)$$

(Similarly, $\mathbf{C}_k(n)$ are st-restricted strings [15] using the statistic $a_{i-1} + (k - 1)$.)

3 Gray Codes and Combinatorial Generation

The term *Gray code* refers to an exhaustive list of some combinatorial object (parameterized by size) in which successive objects differ in some (small) way. They are named after the famous order of n -bit binary strings with Hamming distance one (i.e., a single bit's value is complemented or *flipped*) in Frank Gray's 1954 patent titled *Pulse Code Communication* [5]. The order is referred to as the *binary reflected Gray code (brgc)* and it appears below for $n = 3$, with overlines denoting the bit that changes to create the next string.

$$\mathbf{brgc}(3) = 00\overline{0}, 0\overline{0}1, 01\overline{1}, \overline{0}10, 11\overline{0}, 1\overline{1}1, 10\overline{1}, 100 \quad (9)$$

$$\mathbf{plain}(3) = 1\overleftarrow{2}3, \overleftarrow{1}32, 3\overleftarrow{1}2, \overrightarrow{3}21, \overrightarrow{2}31, 213 \quad (10)$$

Plain changes predates the 'first' Gray code by hundred years and is illustrated for $n = 3$ in (10). In this order, consecutive permutations of $[n] = \{1, 2, \dots, n\}$ differ by a *swap* (i.e., adjacent entries are transposed) with the arrows in (10) showing a larger value "jumping over" a smaller value. The order was performed by bell-ringers in the 1600s [24], and is known as the *Steinhaus-Johnson-Trotter algorithm* due to multiple rediscoveries in the mid-20th century.

Traditionally, Gray codes have been discovered and described recursively. For example, note that **brgc**(3) is obtained from two copies of **brgc**(2) = 00, 01, 11, 10 by prefixing 0 to the strings in the first copy, and 1 to the strings in the second copy which is first *reflected* to 10, 11, 01, 00. Similarly, **plain**(3) is obtained from **plain**(2) = 12, 21 by sweeping 3 from right-to-left through 12 then left-to-right through 21. The first approach uses *global recursion* since **brgc**(n) is created from full copies of **brgc**($n-1$), while the second uses *local recursion* since **plain**(n) expands individual objects in **plain**($n-1$).

Countless Gray codes have been constructed over the years. Academic surveys have been written by Savage [21] and more recently Mütze [17], with Ruskey [18] and Knuth [11] devoting extensive textbook coverage to the subject. In fact, one of the issues facing this research area is the sheer breadth of results and the recursive ‘tricks’ that have been used to obtain them. For an interactive introduction to the area, we recommend the *combinatorial object server* combos.org.

3.1 Greedy Gray Code Algorithm

This decade has seen the introduction of the *greedy Gray code algorithm* [25]. The algorithm eschews recursive schemes to focus on a simple idea: build an order one object at a time by prioritizing the possible changes. For example, the BRGC can be constructed starting from 0^n (where exponentiation denotes concatenation) by greedily flipping the rightmost possible bit. Similarly, plain change starts at $12 \cdots n$ and then greedily swaps the largest possible value³. To clarify these descriptions, consider the partial orders below.

$$\mathbf{brgc}(3) = 00\bar{0}, 0\bar{0}1, 01\bar{1}, 010, \dots? \quad (11)$$

$$\mathbf{plain}(3) = 1\bar{2}3, \bar{1}32, 312, \dots? \quad (12)$$

Which binary string should follow 010? Flipping the rightmost bit would give $01\bar{0} = 011$, but that string is already in (11). Similarly, flipping the middle bit would also result in a repeated string $0\bar{1}0 = 000$. However, flipping the leftmost bit gives a new string $\bar{0}10 = 110$, so it is the next string in the order. Likewise, in (12) we cannot swap 3 to the right as it would recreate $3\bar{1}2 = 132$, nor can we swap it to the left as it is already in the leftmost position. Thus, our highest priority change is to swap the next largest value 2 to the left to create $3\bar{1}2 = 321$.

These two greedy descriptions are not efficient in the sense of *combinatorial generation*, which is focused on efficiently generating exhaustive lists of combinatorial objects. More specifically, both algorithms require an exponential amount of space to determine if a specific change creates a new string or not. However, it is often possible to find an alternate description of a greedily defined order, such as the recursive descriptions of **brgc**(n) and **plain**(n) discussed earlier.

The greedy Gray code algorithm has also proven to be extremely adept at uncovering new results. In particular, the greedy description of plain change

³ The latter description has the potential to be ambiguous—should a value be swapped to the left or right?—but in practice there is always a unique choice.

order was the impetus for the *permutation language* series [6] [7] [16] [3] [2]. Similarly, our new results can be seen as a somewhat unexpected generalization of the binary reflected Gray code.

3.2 Four Greedy Definitions of the Binary Reflected Gray Code

Here we provide four different greedy algorithms for generating the binary reflected Gray code starting from 0^n . The first approach was previously discussed, and it should be clear that the other three approaches produce identical results.

1. Greedily complement the rightmost bit.
2. Greedily increment or decrement the rightmost bit.
3. Greedily increment the rightmost bit cyclically modulo 2.
4. Greedily set the rightmost bit to the maximum possible value.

Figure 1 illustrates the four interpretations for **brgc**(4). In the figure, we use the symbols $\bar{}$ for complement, ± 1 for increment / decrement, \oplus for cyclic increment, and max for maximum possible value. While the four algorithms give the same order for binary strings, we'll see that the last three produce different orders for other sets of strings; we henceforth ignore the first algorithm as complements cannot be applied to non-binary digits. Eventually, we'll see that max approach has a particular advantage for restricted growth strings, as observed in [14] [13].

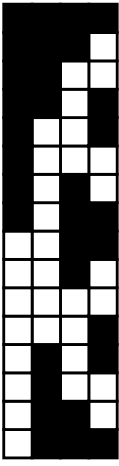
brgc (4)	$b_4b_3b_2b_1$	$\bar{}$	\pm	\oplus	max
	0000	$\bar{1}$	$+1$	\oplus_1	max ₁
	0001	$\bar{2}$	$+2$	\oplus_2	max ₂
	0011	$\bar{1}$	$+1$	\oplus_1	max ₁
	0010	$\bar{3}$	-3	\oplus_3	max ₃
	0110	$\bar{1}$	$+1$	\oplus_1	max ₁
	0111	$\bar{2}$	-2	\oplus_2	max ₂
	0101	$\bar{1}$	-1	\oplus_1	max ₁
	0100	$\bar{4}$	-4	\oplus_4	max ₄
	1100	$\bar{1}$	$+1$	\oplus_1	max ₁
	1101	$\bar{2}$	$+2$	\oplus_2	max ₂
	1111	$\bar{1}$	-1	\oplus_1	max ₁
	1110	$\bar{3}$	-3	\oplus_3	max ₃
	1010	$\bar{1}$	$+1$	\oplus_1	max ₁
	1011	$\bar{2}$	-2	\oplus_2	max ₂
	1001	$\bar{1}$	-1	\oplus_1	max ₁
	1000				

Fig. 1: Four greedy interpretations of **brgc**(4).

3.3 Three Greedy Gray Codes for Mixed-Radix Strings

Let b_1, b_2, \dots, b_n be a list of positive integers called *bases*. A *mixed-radix string* with these bases is any $a_1a_2 \dots a_n$ with $1 \leq a_i \leq b_i - 1$ for all i . In other words,

each b_i provides the number of values that the i^{th} digit can hold. Figure 2 illustrates how three of the greedy approaches mentioned in Section 3.2 generate Gray codes for the strings with bases 1, 2, 3, 4. In each case, the reader's attention should be drawn to the different patterns created in the rightmost digit.

- When using increments and decrements (Figure 2a) the rightmost digit pings-pongs back-and-forth: 0, 1, 2, 3, 3, 2, 1, 0, 0, 1, 2, 3, 3, 2, 1, 0, ... reflectively.
- When using cyclic increments (Figure 2b) the rightmost digit's starting value climbs on each block 0, 1, 2, 3, 3, 0, 1, 2, 2, 3, 0, 1, 1, 2, 3, 0, 0, 1, 2, 3, 3, 0, 1, 2.
- When using maximization (Figure 2c) the rightmost digit alternately starts with 0 and ends with 1 or vice versa 0, 3, 2, 1, 1, 3, 2, 0, 0, 3, 2, 1, 1, 3, 2, 0, ...

The third pattern is quite useful in the context of restricted growth strings. This is because lower values are less likely to exceed their digit's upper bound, so having 0 and 1 as the first and last values (or vice versa) allows the greedy algorithm to uncover safer forms of recursion.

sgc(4, \pm)	$a_3a_2a_1$	\pm
	0000	+4
	0001	+4
	0002	+4
	0003	+3
	0013	-4
	0012	-4
	0011	-4
	0010	+3
	0020	+4
	0021	+4
	0022	+4
	0023	+2
	0123	-4
	0122	-4
	0121	-4
	0120	-3
	0110	+4
	0111	+4
	0112	+4
	0113	-3
	0103	-4
	0102	-4
	0101	-4
	0100	

sgc(4, \oplus)	$a_3a_2a_1$	\oplus
	0000	\oplus_4
	0001	\oplus_4
	0002	\oplus_4
	0003	\oplus_3
	0013	\oplus_4
	0010	\oplus_4
	0011	\oplus_4
	0012	\oplus_3
	0022	\oplus_4
	0023	\oplus_4
	0020	\oplus_4
	0021	\oplus_2
	0121	\oplus_4
	0122	\oplus_4
	0123	\oplus_4
	0120	\oplus_3
	0100	\oplus_4
	0101	\oplus_4
	0102	\oplus_4
	0103	\oplus_3
	0113	\oplus_4
	0110	\oplus_4
	0111	\oplus_4
	0112	

sgc(4, max)	$a_3a_2a_1$	max
	0000	max ₄
	0003	max ₄
	0002	max ₄
	0001	max ₃
	0011	max ₄
	0013	max ₄
	0012	max ₄
	0010	max ₃
	0020	max ₄
	0023	max ₄
	0022	max ₄
	0021	max ₂
	0121	max ₄
	0123	max ₄
	0122	max ₄
	0120	max ₃
	0110	max ₄
	0113	max ₄
	0112	max ₄
	0111	max ₃
	0101	max ₄
	0103	max ₄
	0102	max ₄
	0100	

(a) Increment/decrement.

(b) Cyclic increments.

(c) Maximize digit.

Fig. 2: Three greedy Gray codes for mixed-radix strings with bases 1, 2, 3, 4. Each one greedily applies an operation (or operations) to the rightmost possible digit.

4 Main Result

Theorem 3. *The greedy max-right algorithm generates all (s, f, \mathbf{c}) -restricted growth strings of length n , and successive strings differ by -1 or -2 (cyclically) in one digit.*

Proof. Recall from (5) that $a_1 a_2 \cdots a_n$ is an (s, f, \mathbf{c}) -restricted growth string if

$$0 \leq a_1 \leq s-1 \text{ and } 0 \leq a_i \leq f(a_1 a_2 \cdots a_{i-1}) + c_i \text{ for } 2 \leq i \leq n$$

with $s \geq 1$, $f \geq 0$, and $c \geq 1$. We prove the theorem by induction on $n \geq 1$.

For the base case of $n = 1$, notice that the conditions reduce to $0 \leq a_1 \leq s-1$. Therefore, the greedy max-right algorithm produces the list $0, s-1, s-2, \dots, 1$.

Assume that the result holds for all valid choices and $n = k$. Now consider a specific choice of s , f , and \mathbf{c} with $n = k+1$. Let f' and \mathbf{c}' be the restrictions of f and \mathbf{c} to $n = k$, respectively. By induction, the greedy max-right algorithm creates a Gray code for the (s, f', \mathbf{c}') strings of length k . Let this Gray code be x_1, x_2, \dots, x_p where p is the number of such strings. Now consider the greedy max-right algorithm for the (s, f, \mathbf{c}) strings of length $k+1$. We claim that the algorithm will generate the strings in the following order,

$$\begin{aligned} &g_0(x_1), g_1(x_2), g_0(x_1), g_1(x_2), \dots, g_0(x_p) \text{ if } p \text{ is odd} \\ &g_0(x_1), g_1(x_2), g_0(x_1), g_1(x_2), \dots, g_1(x_p) \text{ if } p \text{ is even} \end{aligned} \quad (13)$$

where the g_0 and g_1 functions expand each x_i string of length k into a sublist of strings of length $k+1$ in a manner described below. Towards these definitions, let $x_i = a_1 a_2 \cdots a_k$. Therefore, $m = f(x_i) + c_{k+1}$ is the maximum value such that $x_i \cdot m$ is a (s, f, \mathbf{c}) string. We also know that $m \geq 1$ due to the conditions that $f \geq 0$ and $\mathbf{c} \geq 1$. The two expansions of x_i are now defined as follows,

$$\begin{aligned} g_0(x_i) &= x_i \cdot 0, x_i \cdot m, x_i \cdot (m-1), \dots, x_i \cdot 2, x_i \cdot 1 \\ g_1(x_i) &= x_i \cdot 1, x_i \cdot m, x_i \cdot (m-1), \dots, x_i \cdot 2, x_i \cdot 0. \end{aligned} \quad (14)$$

In both cases, the expansion sets the last digit to the maximum value m and then repeatedly decrements it. The difference between the two expansions is that the last digit starts at 0 and ends at 1 in the g_0 expansion, and vice versa in the g_1 expansion. To complete the proof we need to argue the following points:

- The greedy max-right algorithm does indeed generate the list in (13).
- The list in (13) includes all (s, f, \mathbf{c}) strings of length $k+1$.
- Successive strings in (13) differ in a single digit by -1 or -2 (cyclically).

To prove the first point, note that the greedy max-right algorithm prefers to change the rightmost digit to the maximum possible value that results in a new string. Therefore, if $x_i \cdot 0$ is the first string to be created with prefix x_i , then the algorithm will proceed by generating the list $g_0(x_i)$. Similarly, if $x_i \cdot 1$ is the first string to be created with prefix x_i , then the algorithm will proceed by generating the list $g_1(x_i)$. In both cases, all of the strings with prefix x_i

are generated in succession. Therefore, when the expansion of x_i is completed, the algorithm will then set the rightmost possible digit in x_i to the maximum possible value. By induction, this means that the prefix x_i will be replaced by x_{i+1} by the next change. Finally, note that the sublist $g_0(x_i)$ ends with $x_i \cdot 1$, so the aforementioned change will result in $x_{i+1} \cdot 1$, which is the first string of $g_1(x_{i+1})$. Similarly, the sublist $g_2(x_i)$ ends with $x_i \cdot 0$, so the aforementioned change will result in $x_{i+1} \cdot 0$, which is the first string of $g_0(x_{i+1})$. Hence, the expanded sublists alternate as per (13).

The second point follows from the fact that a digit's valid values are between 0 and m inclusively. The third point follows from (14) and induction. \square

5 Loopless Algorithms

In this section we provide loopless algorithms for generating multi-radix strings, k -Catalan strings, and Bell strings according to our max-right Gray codes.

5.1 Loopless Mixed-Radix Algorithms

The **MixedRadix** function in Algorithm 1 provides the well-known loopless algorithm for generating a mixed-radix Gray code using increment/decrement (i.e., ± 1) changes (see Knuth's description in [12]). The modified function **MixedRadixMax** in Algorithm 1 instead implements our mixed-radix Gray code using max changes. In this implementation, the s values keep track of the starting value of the corresponding digit: 0 or 1 (as per Section 4).

5.2 Loopless k -Catalan Strings

Our loopless implementation of our max-right k -Catalan Gray code is based on **MixedRadixMax**. One major difference is that the bases for each digit are not provided as inputs. Instead, they are computed as we generate the Gray code: the base of any position is the previous position's value plus $k - 1$.

Theorem 4. *CatalanStrings in Algorithm 2 looplessly generates the max-right Gray code for k -Catalan strings.*

5.3 Loopless Bell Strings

In our loopless implementation of the max-right Gray code for Bell strings, the concept of bases is not directly used, since computing the base of any digit (which is the maximum of previous digits plus 1) is a worst-case $\Theta(n)$ operation. Instead, we store the first positions of digits equal to successively larger values ≥ 2 (i.e., 2, 3, 4, ...) in a stack **S**. If the stack is non-empty, then its size allows us to determine a digit's maximum value. If the stack is empty, then the maximum is typically 2, since the earlier digits are comprised of 0s and 1s by (14). One

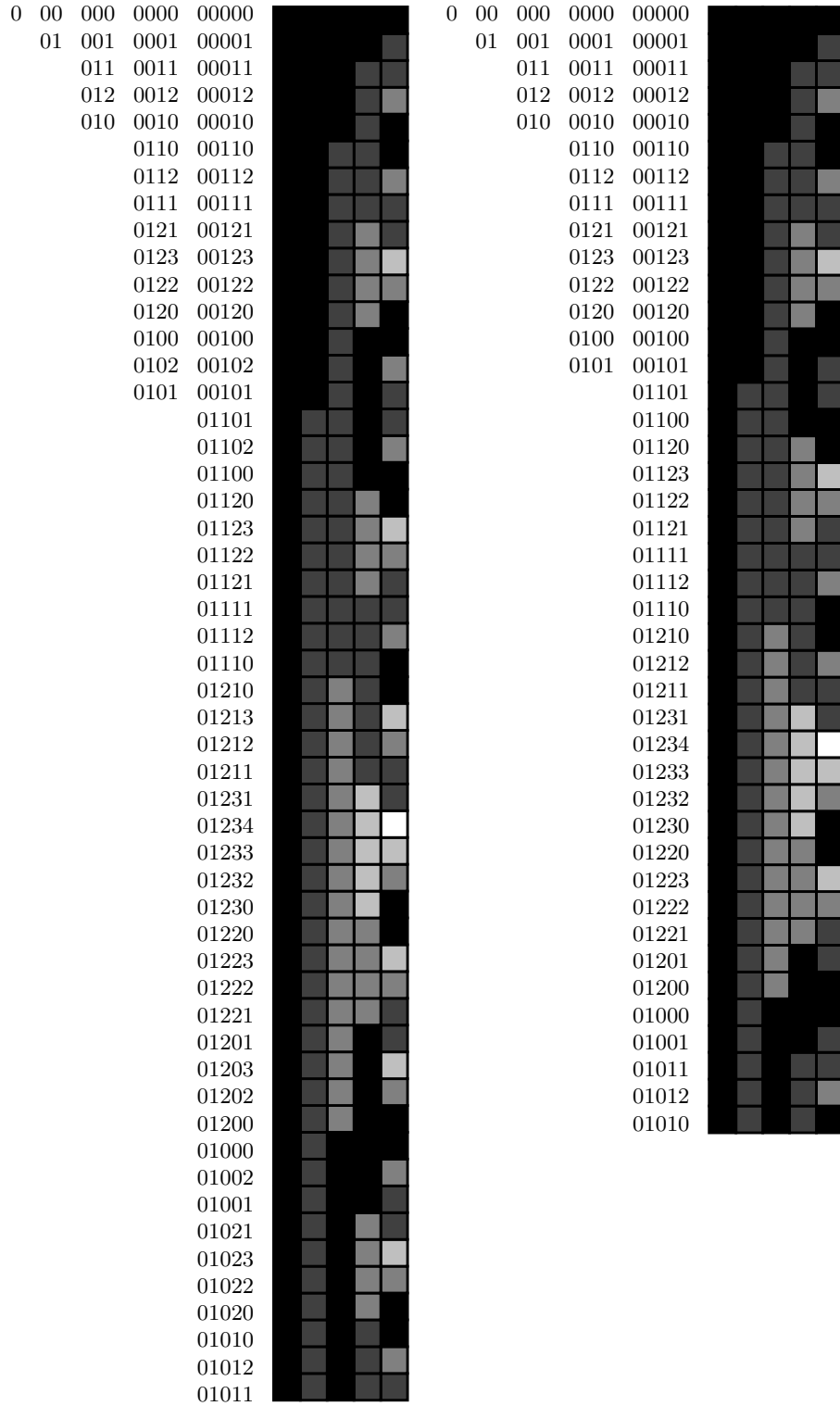


Fig. 3: Gray codes obtained from our max-right algorithm: start from 0^n then greedily maximize the rightmost possible digit.

Algorithm 1 Loopless algorithms for generating mixed-radix Gray codes. The functions modify our target \mathbf{a} and yield it every time it is modified. Focus pointers are stored in \mathbf{f} . In **MixedRadix**, a_i is modified by $+1$ or -1 depending on the direction given by \mathbf{d} . In **MixedRadixMax**, as discussed in Section 3.3, any position has 0 and 1 as the first and last value (or vice versa) in a loop. a_i is raised to maximum ($\mathbf{b}_j - 1$) when $a_i = s_i$ (except in some special cases), and is decreased otherwise (normally it decreases by 1, but decreases by 2 if it is 2 and the start value is 1, in this case it needs to become 0). The overall algorithms are loopless as each iteration runs in worst-case $\mathcal{O}(1)$ -time.

MixedRadix(n)	MixedRadixMax(n)
1: $a_1 a_2 \cdots a_n \leftarrow 0 0 \cdots 0$	1: $a_1 a_2 \cdots a_n \leftarrow 0 0 \cdots 0$
2: $f_1 f_2 \cdots f_{n+1} \leftarrow 1 2 \cdots n+1$	2: $f_1 f_2 \cdots f_{n+1} \leftarrow 1 2 \cdots n+1$
3: $d_1 d_2 \cdots d_n \leftarrow 1 1 \cdots 1$	3: $s_1 s_2 \cdots s_n \leftarrow 0 0 \cdots 0$
4: while $f_1 \leq n$	4: while $f_1 \leq n$
5: $j \leftarrow f_1$	5: $j \leftarrow f_1$
6: $f_1 \leftarrow 1$	6: $f_1 \leftarrow 1$
7: $a_j \leftarrow a_j + d_j$	7: if $a_j = s_j$
8: yield j, \mathbf{a}	8: if $b_j = 2$ and $s_j = 1$
9: if $a_j \in b_j - 1, 0$	9: $a_j \leftarrow 0$
10: $d_j \leftarrow -d_j$	10: else
11: $f_j \leftarrow f_{j+1}$	11: $a_j \leftarrow b_j - 1$
12: $f_{j+1} \leftarrow j + 1$	12: else if $a_j = 2$ and $s_j = 1$
	13: $a_j \leftarrow 0$
	14: else
	15: $a_j \leftarrow 1$
	16: yield j, \mathbf{a}
	17: if $a_j = 1 - s_j$
	18: $s_j \leftarrow a_j$
	19: $f_j \leftarrow f_{j+1}$
	20: $f_{j+1} \leftarrow j + 1$

exception is that these digits are all 0s precisely when the digit is being changed for the first time. To track this special case, we store whether or not a digit has ever been changed in a Boolean list \mathbf{v} ; Collectively, this additional information allows us to determine the maximum value for a digit in worst-case $\mathcal{O}(1)$ -time.

Theorem 5. *BellStrings in Algorithm 2 looplessly generates the max-right Gray code for Bell strings.*

6 Final Remarks

We provided Gray codes for restricted growth strings parameterized by (s, f, \mathbf{c}) . The orders change one digit by -1 or -2 (cyclically) and are generated starting from 0^n by a simple greedy rule: set the rightmost possible digit to the maximum possible value that gives a new string. Our greedy max-right algorithms are not

Algorithm 2 Loopless algorithms for generating k -Catalan Gray codes and Bell Gray codes. The functions modify our target \mathbf{a} and yield it every time it is modified. Focus pointers are stored in \mathbf{f} . **CatalanStrings** largely replicates **MixedRadixMax**, except that the “bases” are calculated on the fly. In **BellStrings**, if the current digit is not visited, the maximum is set to 0 because all earlier digits are 0. If it is visited and the stack storing positions of large numbers is empty, the maximum is set to 1. If the stack is not empty, the maximum is set to the corresponding digit at the position determined by the top of stack. After calculating the maximum, it will be pushed onto the stack. When a digit is decreased, if it corresponds to the top of stack, the stack is popped.

CatalanStrings(n, k)	BellStrings(n)
1: $a_1 a_2 \cdots a_n \leftarrow 0\ 0 \cdots 0$	1: $a_1 a_2 \cdots a_n \leftarrow 0\ 0 \cdots 0$
2: $f_1 f_2 \cdots f_{n+1} \leftarrow 1\ 2 \cdots n+1$	2: $f_1 f_2 \cdots f_{n+1} \leftarrow 1\ 2 \cdots n+1$
3: $s_1 s_2 \cdots s_n \leftarrow 0\ 0 \cdots 0$	3: $s_1 s_2 \cdots s_n \leftarrow 0\ 0 \cdots 0$
4: yield j, \mathbf{a}	4: $S \leftarrow \text{empty}$
5: while $f_1 < n$	5: $v_1 \cdots v_n \leftarrow \text{True} \cdots \text{True}$
6: $j \leftarrow f_1$	6: yield j, \mathbf{a}
7: $f_1 \leftarrow 1$	7: while $f_1 < n$
8: if $a_j = s_j$	8: $j \leftarrow f_1$
9: if $a_j, a_{j+1} = 1, 0$ and $k = 2$	9: $f_1 \leftarrow 1$
10: $a_j \leftarrow 0$	10: if $a_j = s_j$
11: else	11: if $v_j = \text{True}$
12: $a_j \leftarrow a_{j+1} + k - 1$	12: $m \leftarrow 0$
13: else if $a_j = 2$ and $s_j = 1$	13: $v_j \leftarrow \text{False}$
14: $a_j \leftarrow 0$	14: else if S is empty
15: else	15: $m \leftarrow 1$
16: $a_j \leftarrow -1$	16: else
17: yield j, \mathbf{a}	17: $pos \leftarrow \text{top}(S)$
18: if $a_j = 1 - s_j$	18: $m \leftarrow a_{pos}$
19: $s_j \leftarrow a_j$	19: else if $a_j = 2$ and $s_j = 1$
20: $f_j \leftarrow f_{j+1}$	20: $a_j \leftarrow 0$
21: $f_{j+1} \leftarrow j + 1$	21: if $\text{top}(S) = j$
	22: $\text{pop}(S)$
	23: else
	24: $a_j \leftarrow -1$
	25: if $\text{top}(S) = j$
	26: $\text{pop}(S)$
	27: yield j, \mathbf{a}
	28: if $a_j = 1 - s_j$
	29: $s_j \leftarrow a_j$
	30: $f_j \leftarrow f_{j+1}$
	31: $f_{j+1} \leftarrow j + 1$

efficient, but the orders can be efficiently generated by other means. We showed this with loopless algorithms for mixed-radix strings, k -Catalan strings, and

Bell strings. In particular, the k -Catalan strings appear to be new within the literature.

References

1. Arndt, J.: *Matters Computational: ideas, algorithms, source code*. Springer Science & Business Media (2010)
2. Cardinal, J., Hoang, H.P., Merino, A., Mička, O., Mütze, T.: Combinatorial generation via permutation languages. v. acyclic orientations. *SIAM Journal on Discrete Mathematics* **37**(3), 1509–1547 (2023)
3. Cardinal, J., Merino, A., Mütze, T.: Efficient generation of elimination trees and graph associahedra. In: *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. pp. 2128–2140. SIAM (2022)
4. Ehrlich, G.: Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *Journal of the ACM (JACM)* **20**(3), 500–513 (1973)
5. Gray, F.: Pulse code communication. United States Patent Number 2632058 (1953)
6. Hartung, E., Hoang, H., Mütze, T., Williams, A.: Combinatorial generation via permutation languages. i. fundamentals. *Transactions of the American Mathematical Society* **375**(4), 2255–2291 (2022)
7. Hoang, H.P., Mütze, T.: Combinatorial generation via permutation languages. II. lattice congruences. *Israel Journal of Mathematics* **244**(1), 359–417 (2021)
8. Inc., O.F.: Bell or exponential numbers, Entry A000110 in the *On-line Encyclopedia of Integer Sequences*. <https://oeis.org/A000110> (2023), accessed on August 30 2023
9. Kaye, R.: A Gray code for set partitions. *Information Processing Letters* **5**, 171–173 (1976)
10. Kerr, K.: Successor rule for a restricted growth string gray code. unpublished manuscript (2015)
11. Knuth, D.E.: *The art of computer programming, volume 4A: combinatorial algorithms, part 1*. Pearson Education India (2011)
12. Knuth, D.E.: *The art of computer programming: generating all tuples and permutations*. Addison-Wesley. (2005)
13. Li, Y., Sawada, J.: Gray codes for reflectable languages. *Information processing letters* **109**(5), 296–300 (2009)
14. Mansour, T., Nassar, G., Vajnovszki, V.: Loop-free Gray code algorithm for the e-restricted growth functions. *Information Processing Letters* **111**(11), 541–544 (2011)
15. Mansour, T., Vajnovszki, V.: Efficient generation of restricted growth words. *Information Processing Letters* **113**(17), 613–616 (2013)
16. Merino, A., Mütze, T.: Combinatorial generation via permutation languages. III. rectangulations. *Discrete & Computational Geometry* pp. 1–72 (2022)
17. Mütze, T.: Combinatorial Gray codes-an updated survey. *The Electronic Journal of Combinatorics* **30**(3), DS26 (2022)
18. Ruskey, F.: *Combinatorial generation*. Preliminary working draft. University of Victoria, Victoria, BC, Canada **11**, 20 (2003)
19. Ruskey, F., Savage, C.D.: Gray codes for set partitions and restricted growth tails. *Australas. J Comb.* **10**, 85–96 (1994)
20. Sabri, A., Vajnovszki, V.: Two reflected Gray code-based orders on some restricted growth sequences. *The Computer Journal* **58**(5), 1099–1111 (2015)
21. Savage, C.: A survey of combinatorial Gray codes. *SIAM review* **39**(4), 605–629 (1997)

22. Stanley, R.P.: Enumerative combinatorics volume 1 second edition. Cambridge studies in advanced mathematics (2011)
23. Stanley, R.P.: Catalan numbers. Cambridge University Press (2015)
24. Stedman, F.: Campanalogia: or the Art of Ringing Improved, With plain and easie rules to guide the Practitioner in the Ringing all kinds of Changes, To Which is added, great variety of New Peals. London (1677)
25. Williams, A.: The greedy Gray code algorithm. In: Algorithms and Data Structures: 13th International Symposium, WADS 2013, London, ON, Canada, August 12-14, 2013. Proceedings 13. pp. 525–536. Springer (2013)
26. Zaks, S.: Lexicographic generation of ordered trees. Theoretical Computer Science **10**(1), 63–82 (1980)

A Ternary String Gray Codes

Figure 4 illustrates the three greedy approaches to generating ternary strings. (The first approach discussed in Section 3.2 does not generalize to ternary strings since there is no natural notion of complementation in this context.)

tgc(3, \pm)	$a_3a_2a_1$	\pm
	000	+3
	001	+3
	002	+2
	012	-3
	011	-3
	010	+2
	020	+3
	021	+3
	022	+1
	122	-3
	121	-3
	120	-2
	110	+3
	111	+3
	112	-2
	102	-3
	101	-3
	100	+1
	200	+3
	201	+3
	202	+2
	212	-3
	211	-3
	210	+2
	220	+3
	221	+3
	222	

tgc(3, \oplus)	$a_3a_2a_1$	\oplus
	000	\oplus_3
	001	\oplus_3
	002	\oplus_2
	012	\oplus_3
	010	\oplus_3
	011	\oplus_2
	021	\oplus_3
	022	\oplus_3
	020	\oplus_1
	120	\oplus_3
	121	\oplus_3
	122	\oplus_2
	102	\oplus_3
	100	\oplus_3
	101	\oplus_2
	111	\oplus_3
	112	\oplus_3
	110	\oplus_1
	210	\oplus_3
	211	\oplus_3
	212	\oplus_2
	222	\oplus_3
	220	\oplus_3
	221	\oplus_2
	201	\oplus_3
	202	\oplus_3
	200	

tgc(3, max)	$a_3a_2a_1$	max
	000	max ₃
	002	max ₃
	001	max ₂
	021	max ₃
	022	max ₃
	020	max ₂
	010	max ₃
	012	max ₃
	011	max ₁
	211	max ₃
	212	max ₃
	210	max ₂
	220	max ₃
	222	max ₃
	221	max ₂
	201	max ₃
	202	max ₃
	200	max ₁
	100	max ₃
	102	max ₃
	101	max ₂
	121	max ₃
	122	max ₃
	120	max ₂
	110	max ₃
	112	max ₃
	111	

(a) Increment/decrement.

(b) Cyclic increment.

(c) Maximize digit.

Fig. 4: Three greedy Gray codes for ternary strings. Each one greedily applies an operation (or operations) to the rightmost possible digit.

B Python Code

Python implementations of the loopless algorithms in Section 5 are provided.

```

def mixedRadixGrayCodeMax(bases):
    n = len(bases)
    word = [0] * n
    start = [0] * n
    yield word, None
    focus = list(range(n+1))
    while focus[0] < n:
        index = focus[0]
        focus[0] = 0
        if word[index] == start[index]:
            if bases[index] == 2 and start[index] == 1:
                word[index] = 0 # special case of start == max
            else:
                word[index] = bases[index]-1 # set to max
        elif word[index] == 2 and start[index] == 1:
            word[index] -= 2
        else:
            word[index] -= 1
        yield word, index
        if word[index] == 1-start[index]:
            start[index] = word[index]
            focus[index] = focus[index+1]
            focus[index+1] = index+1

bases = [2,3,4]
total = 0
for word, change in mixedRadixGrayCodeMax(bases):
    total += 1
    print(*word, sep="", end=" ")
    print(change)
print("\ntotal: %d / %d" % (total, prod(bases)))

```

```

def looplessCatalanStrings(n):
    word = [0] * n
    yield word
    focus = list(range(n+1))
    start = [0] * n
    while focus[0] < n-1:
        index = focus[0]
        focus[0] = 0
        if word[index] == start[index]:
            # set to max
            # but handle special case where it is both the start and the max already
            if word[index] == 1 and word[index+1] == 0:
                word[index] = 0
            else:
                word[index] = word[index+1]+1
        elif word[index] == 2 and start[index] == 1:
            # skip over 1
            word[index] -= 2
        else:
            word[index] -= 1
        yield word
        if word[index] + start[index] == 1: # last value (i.e., 0+1 or 1+0)
            focus[index] = focus[index+1]
            focus[index+1] = index+1
            start[index] = word[index]

n = 5
total = 0
for word in looplessCatalanStrings(n):
    total += 1
    print(*word, sep=" ")
print("\ntotal: %d" % total)

```

```

def looplessBellStrings(n):
    word = [0] * n
    yield word
    focus = list(range(n+1))
    start = [0] * n
    maxima = [] # indices that create maxima values ...,3,2
    first = [True] * n # first if the digit hasn't been changed yet

    while focus[0] < n-1:
        # maximaValues = [word[index] for index in maxima]
        index = focus[0]
        focus[0] = 0
        if word[index] == start[index]:
            # set to max
            if first[index]: # only time when digits to the right are all 0
                m = 0
                first[index] = False
                assert len(maxima) == 0
            elif len(maxima) == 0: # no 2s and not first so max is 1
                m = 1
            else:
                m = word[maxima[0]]

            word[index] = m+1
            if m+1 != 1:
                maxima = [index] + maxima

        elif word[index] == 2 and start[index] == 1:
            # skip over 1
            word[index] -= 2
            if maxima[0] == index:
                maxima = maxima[1:]
            else:
                word[index] -= 1
                if maxima[0] == index:
                    maxima = maxima[1:]
        yield word

        if word[index] + start[index] == 1: # last value (i.e., 0+1 or 1+0)
            focus[index] = focus[index+1]
            focus[index+1] = index+1
            start[index] = word[index]

n = 7
total = 0
for word in looplessBellStrings(n):
    total += 1
    print(*word, sep=" ")
print("\ntotal: %d" % total)

```