



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 $\mathbf{231}$ 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 $1\,\bar{2}\,3\,\bar{4}$ into $1\,\bar{3}\,2\,\bar{4}$.

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.

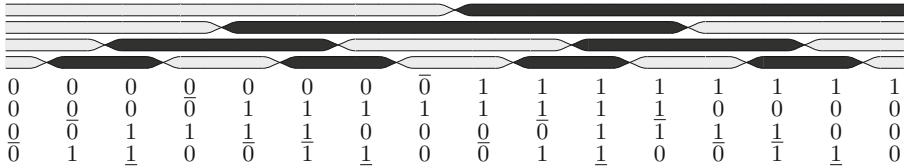


Fig. 2. Binary reflected Gray code using indistinct two-sided ribbons for $n = 4$.

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

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

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 $f_1(\mathbf{t}) = f_1(0010) = 001\bar{0} = 0011$ is already in the list. Similarly, $f_2(\mathbf{t}) = f_2(0010) = 00\bar{1}0 = 0000$ is also in the list. But $f_3(\mathbf{t}) = f_3(0010) = 0\bar{0}10 = 0110$ 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

$$\text{plain}(4) = 12\bar{3}4, 1\bar{2}43, \bar{1}423, 41\bar{2}3, \bar{4}132, 14\bar{3}2, 134\bar{2}, 1324, \dots \quad (4)$$

We can't apply $\overleftarrow{s_4}$ to $\mathbf{t} = 1324$ since $\overleftarrow{s_4}(1324) = 13\bar{2}4 = 1342$ 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 $\overleftarrow{s_3}(\mathbf{t}) = \bar{1}324 = 3124$ 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

```

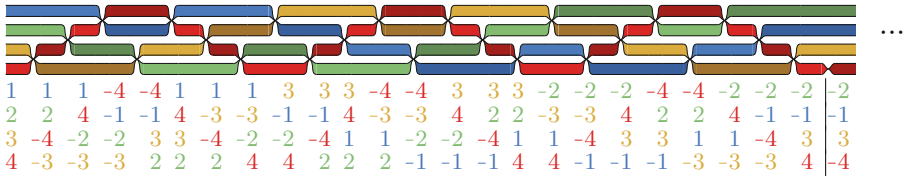


Fig. 4. Twisted plain changes $\text{twisted}(n)$ for $n = 4$ up to its 25th entry.

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

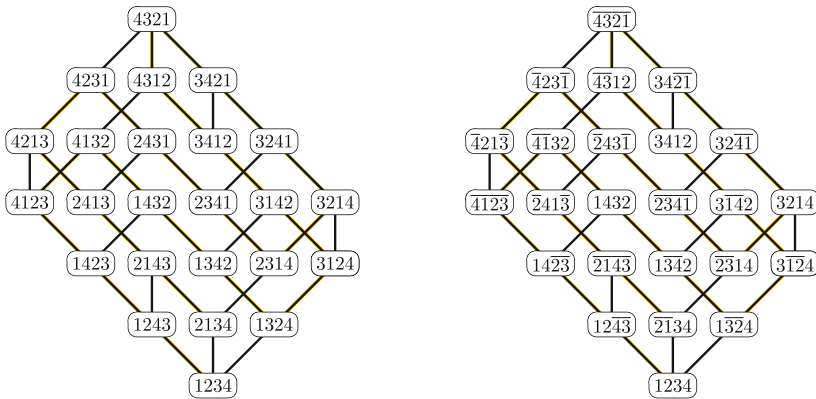
At this point it is helpful to compare the start of plain changes and twisted plain changes. The *permutohedron of order n* is a graph whose vertices are permutations S_n and whose edges join two permutations that differ by a swap. Plain changes traces a Hamilton path in this graph, as illustrated in 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 $\overline{2}1\overline{3}4 \in S_n^\pm$ follows 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, $\text{twisted}(n) = \text{greedy}(\mathbf{s}, \langle \overleftarrow{\mathbf{t}}_n, \overrightarrow{\mathbf{t}}_n, \overleftarrow{\mathbf{t}}_{n-1}, \overrightarrow{\mathbf{t}}_{n-1}, \dots, \overleftarrow{\mathbf{t}}_2, \overrightarrow{\mathbf{t}}_2, \mathbf{t}_n, \dots, \mathbf{t}_2, \mathbf{t}_1 \rangle)$ 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

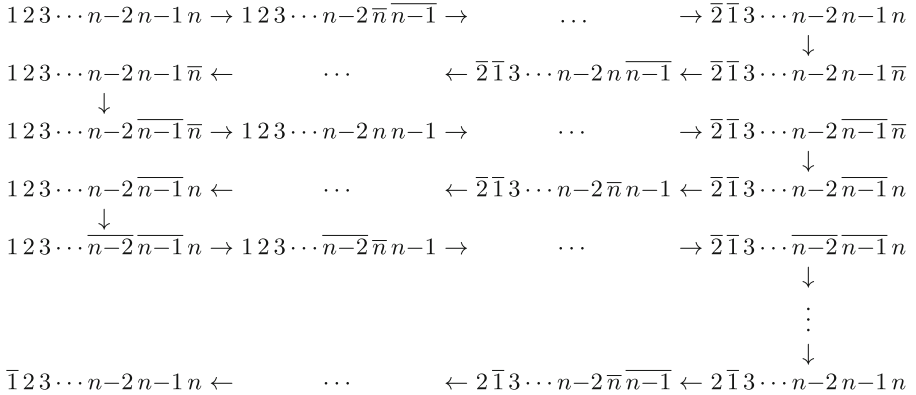


Fig. 6. The global structure of twisted plain changes. Each row greedily applies 2-twists to the largest possible symbol, thus following plain changes. At the end of a row, no 2-twist can be applied, and the down arrows greedily 1-twist the largest possible symbol. The rows alternate left-to-right and 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.

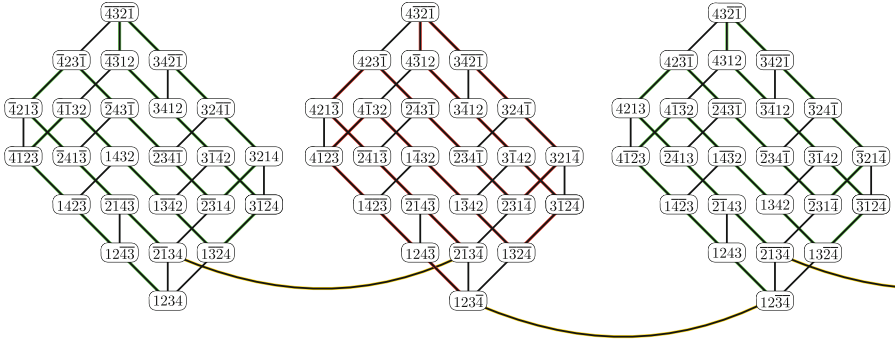


Fig. 7. Our $\text{twisted}(4)$ order begins by traversing the above signed permutohedron, starting from the 2-twisted permutohedron on the left. Straight lines are the edges of a signed permutohedron (i.e., every possible 2-twist). Curved edges are 1-twists between the vertices shown, and they connect 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 $\text{twisted}(n)$. We'll instead develop a loopless implementation of $\text{twisted}(n)$ 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 $\text{twisted}(n)$ 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 $\bar{0}$ | -2 | $\overrightarrow{3214}$ | 000012 $\bar{0}$ | -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 | $\bar{2}$ 134 |
| | | | | | | | | 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 $\text{twisted}(n)$ 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 $\bar{0}0, \bar{1}0, 2\bar{0}, \bar{2}1, \bar{1}1, 10$ 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)