Skipping Ropes: An Efficient Gray Code Algorithm for Generating Wiggly Permutations

Vincent Pilaud
Departament de Matemàtiques i Informàtica, Universitat de Barcelona, Spain

Aaron Williams
Department of Computer Science, Williams College, Williamstown, MA, USA

Abstract

Wiggly permutations were introduced by Bapat and Pilaud (*Wigglyhedron* Mathematische Zeitschrift 2025). We positively answer one of their conjectures by finding a Hamilton path in the wiggly flip graph that is isomorphic to the wigglyhedron. Our path provides a Gray code in which successive wiggly permutations are obtained by a single jump or hop, meaning that one or two consecutive symbols move past some number of smaller symbols. The Gray code has a simple greedy description that produces a recursive zig-zag pattern reminiscent of *plain changes* for permutations. More broadly, our results extend Algorithm J and the series of papers on zig-zag languages initiated by Hartung, Hoang, Mütze and Williams (*Combinatorial Generation via Permutation Languages* SODA 2020). Finally, we use *wiggly changes* as the basis for an $\mathcal{O}(n)$-time delay generation algorithm.

2012 ACM Subject Classification Mathematics of computing → Permutations and combinations; Mathematics of computing → Combinatorial algorithms; Mathematics of computing → Paths and connectivity problems

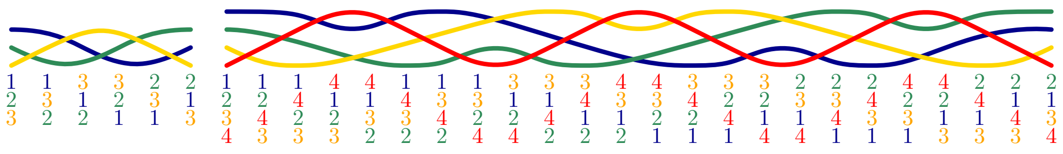
Keywords and phrases permutations, wiggly permutations, pattern avoidance, permutahedron, wigglyhedron, Hamilton path, flip graph, Gray code, combinatorial generation, generation algorithm

Digital Object Identifier 10.4230/LIPIcs.WADS.2025.46

1 Introduction

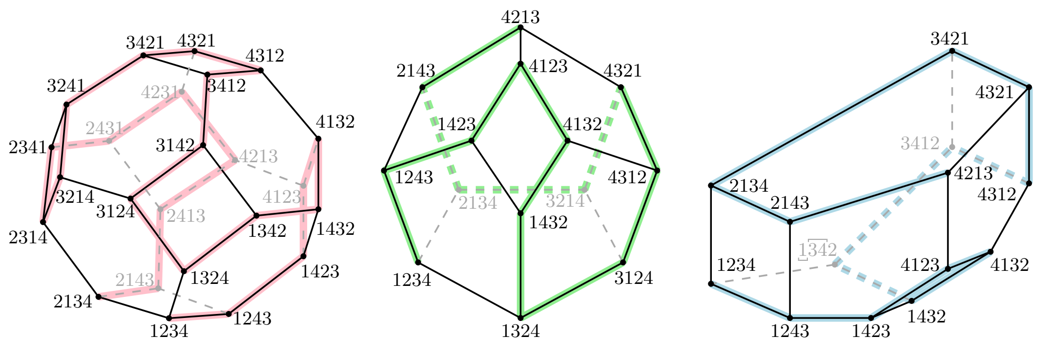
We use a classic order of permutations called *plain changes* as inspiration for ordering an interesting new subset of permutations called *wiggly permutations*. Plain changes can be described as a *swap Gray code of permutations*, meaning that it lists all $n!$ permutations so that successive permutations differ by a swap (i.e., the transposition of two adjacent symbols)[1]. It is arguably the most prominent non-lexicographic order of permutations, and it is often used when generating permutations efficiently. Likewise, we use our *wiggly changes* as the basis for the first efficient generation algorithm of wiggly permutations. This introductory section also discusses how our results can be seen as the next step in the broad generalization of plain changes from the recent *Permutation Languages* series of papers.

[1] For excellent background information on Gray codes see Mütze's dynamic survey [34].



(a) $n = 3$. **(b)** $n = 4$.

Figure 1 Plain changes is a swap Gray code for the permutations of $\{1, 2, \ldots, n\}$. It is visualized here with one rope per number, where each swap moves a larger rope over an adjacent smaller rope.



(a) Permutahedron for $n = 4$. **(b)** Associahedron for $n = 4$. **(c)** Wigglyhedron for $n = 4$.

Figure 2 Polyhedra with Hamilton paths on their 1-skeletons. The vertices are (a) permutations, (b) 231-avoiding permutations, and (c) wiggly permutations. The edges are cover relations in Hasse diagrams (see Figure 6) and are (a) swaps, (b) minimal jumps, and (c) wiggly flips. The paths follow (a) plain changes, (b) Algorithm J [16], and (c) wiggly changes. The paths are all created greedily by Algorithm F via the edge labels in Definition 6. We also generate wiggly changes efficiently.

1.1 Plain Changes for Permutations

In the 17th century bell-ringers faced a combinatorial problem. They wished to list all $n!$ permutations of $[n] = \{1, 2, \ldots, n\}$ so that successive permutations differ by a swap (i.e., the transposition of two adjacent symbols). In this context, a permutation of $[n]$ represents the order in which n large church bells can be rung by n individuals, the ringing of all $n!$ permutations is a *peal*, and the closeness condition on successive permutations represents a physical limitation in altering the cadence of each bell. Over the course of several decades the community devised a pattern for $n = 4$ bells and then generalized it to arbitrary n [21]. The resulting *plain changes* is often illustrated using ropes, where each rope illustrates the movement of an individual bell. Figure 1 provides the order for $n = 3$ and $n = 4$ along with a subtle but important graphical feature: larger ropes go over smaller ropes. For example, we view the transition from 1243 to 1423 not just as a swap, but as 4 moving over 2. To emphasize this perspective, we will often write such a transition as $1\underline{2}\overline{4}3$ to $1\overline{4}\underline{2}3$.

In the 20th century computer scientists faced the same combinatorial problem. This time the goal was to generate all $n!$ permutations as quickly as possible within a computer program. Plain changes was discovered independently multiple times, and the pattern became known in this community as the *Steinhaus-Johnson-Trotter algorithm* [51, 19, 54].

We can visualize this type of problem using a *flip graph*. Its vertices are the objects to create and its edges represent the allowable changes between objects. A solution is a Hamilton path. We refer to the order of successive objects on the path as a *Gray code* for the associated objects using the associated operations or changes. The flip graph for the bell-ringing problem is shown in Figure 2a for $n = 4$ and it can be described in a number of other ways. For example, it is the Cayley graph of the symmetric group under swaps [12]. It is also the Hasse diagram of the permutations under the weak order [55] as seen in Figure 6a. The flip graph is also isomorphic to the permutahedron [38] as seen in Figure 2a.

Plain changes is typically defined using *local recursion* meaning that each permutation of $[n-1]$ is expanded into n permutations of $[n]$. The first n permutations in the order for $[n]$ is obtained by sweeping the value n from right-to-left through the first permutation in the order for $[n-1]$. Then n is swept from left-to-right through the second permutation in the order for $[n-1]$, and so on. This zig-zag pattern is illustrated by the red rope in Figure 1.

Alternatively, plain changes for $[n]$ can be defined using a simple greedy algorithm without any reference to the order for $[n-1]$. The algorithm starts a list from $12 \cdots n$, then repeatedly extends the list to a new permutation as follows: from the most recently added permutation swap the largest value that gives a new permutation. For example, consider the algorithm once it has generated the partial list $1234, 1243, 1423, 4123$. At this point the most recently added permutation is 4123. The algorithm cannot swap the largest value 4 to the left (since it is in the leftmost position) or to the right (since $\overline{4}\underline{1}23 = 1423$ is already in the list). So it considers swapping the next largest value 3, which produces the next permutation in the list $41\underline{2}\overline{3} = 4132$. Note that the algorithm description is potentially ambiguous since it doesn't specify whether a particular value should prefer to swap to the left or right, however, it turns out that such a choice never arises. This greedy interpretation was introduced at WADS 2013 as part of a larger investigation that showed how simple greedy algorithms can recreate the recursive definitions used in classic Gray code constructions [58].

1.2 Jump Gray Codes for Zig-Zag Languages

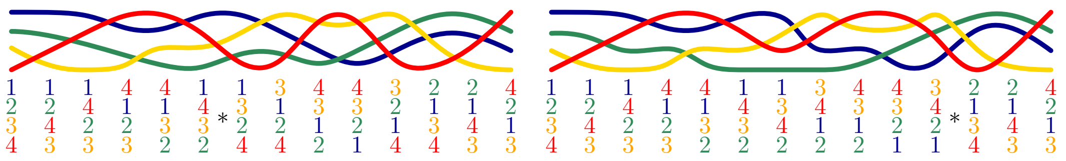
Permutations can be used to represent other combinatorial objects. For example, a permutation $p_1 p_2 \cdots p_n$ *avoids* the pattern 123 if there is no increasing subsequence of length three. That is, $\nexists i < j < k$ with $p_i < p_j < p_k$. These permutations were first studied over a hundred years ago by MacMahon [29] and were shown to be counted by the Catalan numbers. Knuth then showed that the same numbers count the 231-avoiding permutations. Both sets of permutations have natural bijections with other objects counted by the Catalan numbers, including binary trees with n nodes [50]. For basic concepts in pattern avoidance see [2].

Using these types of bijections we can create lists of various combinatorial objects by creating lists of corresponding permutations. In this context it is sometimes the case that a small change in the permutation corresponds to a small change in the combinatorial object that it represents. In other words, a Gray code for the permutations may provide a Gray code for the combinatorial objects. This connection has been explored in great detail by the recent *Permutation Languages* series of papers starting from [16]. In these papers, it is shown that every "zig-zag language" has a *jump Gray code*, meaning that consecutive permutations differ by moving one larger symbol over some number of smaller symbols. Furthermore, these Gray codes can be seen as natural generalizations of plain changes. They are generated by a simple greedy algorithm called Algorithm J: jump the largest possible value the shortest possible distance that creates a new permutation in the language. When applied to 231-avoiding permutations the algorithm generates a Hamilton path in a flip graph that is isomorphic to the associahedron [38] as shown in Figure 2b. And just like in plain changes, this algorithm creates a locally recursive zig-zag pattern, except this time a value may skip over some positions when it sweeps from side to side as shown in Figure 3a.

1.3 Not-So Plain Changes: Wiggly Changes

We consider *wiggly permutations* which were recently introduced by Babat and Pilaud [1]. Informally, a wiggly permutation restricts the values between pairs of the form $(2j-1, 2j)$. More specifically, if the smaller value $2j-1$ appears to the left/right of the larger value $2j$, then the values between them must be larger/smaller. To make this idea more concrete, consider a wiggly permutation π for $n = 8$ and the restrictions placed on it by the pair $(3, 4)$.

- If $\pi = \_3\_4\_$, then the middle piece is restricted to values selected from $\{5, 6, 7, 8\}$.
- If $\pi = \_4\_3\_$, then the middle piece is restricted to values selected from $\{1, 2\}$.



(a) 231-avoiding permutations. The order uses jumps that are not swaps (e.g., the 4 jumps in $*$).

(b) Wiggly permutations. The order uses hops that are not jumps (e.g., the 34 hop together in $*$).

Figure 3 Gray codes for two permutations languages visualized using $n = 4$ ropes. To avoid invalid permutations, the red rope changes speed in (a), and also reverses direction earlier in (b). These two "skipping rope" patterns (and plain changes) are generated by Algorithm F.

Just as permutations are the vertices of the permutahedron, and 231-avoiding permutations are the vertices of the associahedron, the wiggly permutations are the vertices of the wigglyhedron (see Figure 2). When viewed as a graph, the edges of the wigglyhedron represent wiggly flips, which are formally defined in Section 3 along with other wiggly concepts.

▶ **Conjecture 1** ([1]). *The wigglyhedron has a Hamilton path. In other words, there is a Gray code for wiggly permutations using wiggly flips.*

We affirm a stronger version of Conjecture 1 that only allows jumps and hops. A *hop* is like a jump except that a pair of consecutive and adjacent symbols moves over some adjacent smaller values. For example, $65\overline{78}\underline{413}2 = 65413782$ is a hop as the consecutive and adjacent pair 78 moves over the adjacent smaller values 413. It is also *minimal* since the larger symbols move the shortest possible distance to create another valid wiggly permutation.

▶ **Theorem 2.** *There is a minimal jump and minimal hop Gray code of wiggly permutations.*

1.4 Generation Algorithms

We proved Theorem 2 using a simple greedy algorithm that generalizes Algorithm J. Furthermore, the Gray code creates a locally recursive zig-zag pattern. Unlike the jump Gray codes for zig-zag languages, more than one symbol moves between the end of one sweep and the start of the next sweep (see Figure 3b). Nevertheless, we use our new Gray code order in an efficient generation algorithm. It runs with $\mathcal{O}(n)$*-time delay* meaning each wiggly permutation of $[n]$ is generated exactly once, and worst-case $\mathcal{O}(n)$-time is used between successive wiggly permutations; further analysis may reduce this to amortized $\mathcal{O}(1)$-time delay.

Our algorithm is the first to generate wiggly permutations, and it adds to a new trend in the area of combinatorial generation [42, 24]. When developing a generation algorithm, one must first decide on which order to follow. The most common choices are lexicographic orders [43, 60] followed by Gray code orders. While Gray code orders have certain benefits they are typically assumed to be more challenging. This assumption has long been false for permutations [49] and other basic combinatorial objects [11, 21] including combinations [46]. Gray code algorithms for more general objects have also been simplified over the years – compare the pioneering Gray code algorithms for multiset permutations [22, 23] to more modern examples [57]. However, Gray code algorithms have almost always been developed after lexicographic algorithms. But this practice has begun to change with the rise of greedily constructed Gray codes (see Section 2) which often produce orders that can be used in generation algorithms that are simpler, faster, and easier to analyze. Other recent examples of this trend include the generation of various permutation languages representing rectangulations [30] and s-Stirling permutations representing s-increasing trees [3].

1.5 Outline

Section 2 formulates the greedy Gray code algorithm using edge-labeled graphs. Section 3 provides background on wiggly concepts. Section 4 describes *wiggly changes* both greedily and using local recursion. Finally, Section 5 generates our Gray code efficiently. Appendix A has a full Python implementation. Wiggly changes for $n = 4, 5, 6$ appears in Table 2 and can serve as a handy reference.

2 Greedy Gray Code Algorithm

The *greedy Gray code algorithm (GGA)* was originally presented using an initial object $x \in \mathbf{X}$ and a prioritized list of operations $\mathcal{O} = o_1, o_2, \ldots, o_k$ where $o_i : \mathbf{X} \to \mathbf{X}$ for all $1 \leq i \leq k$. We now reformulate the approach with edge-labeled flip graphs[2]. For example, plain changes is created in the permutahedron by labeling the edges with the larger value that is swapped.

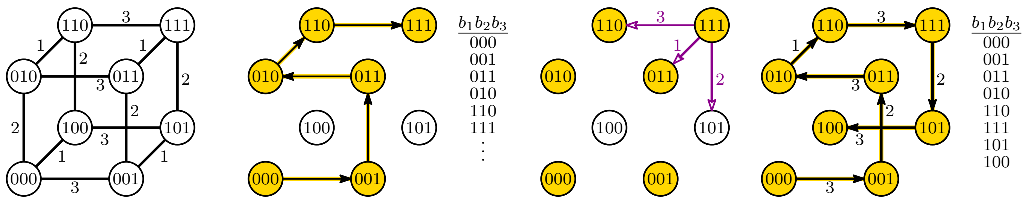
Algorithm F (Greedy max-flips).

This algorithm attempts to greedily traverse a Hamilton path in an edge-labeled flip graph starting from an initial node.

1. [Initialize] Start a path at the initial node.
2. [Greedy] Extend the path to a new node using an edge with a label that is as large as possible. If there is no such edge (i.e., every neighbour is in the path) or ambiguity (i.e., multiple suitable edges with the same label), then halt. Otherwise repeat 2.

Figure 4 shows how Algorithm F creates the *binary reflected Gray code* (BRGC) [14]. The edges of the hypercube are labeled by the index of the bit that differs between adjacent vertices. The BRGC was previously described in [58] as "greedily flip the rightmost bit". Algorithm F is sensitive to the choice of edge labels. For example, a Hamilton path is not created in the permutahedron when labeling its edges by the index of the swap (see Figure 5).

One of the most famous Gray codes is the middle levels theorem by Mütze [33], which is a Hamilton cycle in the middle two levels of the hypercube for odd n. While the proof has been simplified over the past decade [35], it seems very unlikely that the theorem could be proven with any simple greedy construction. Nevertheless, the GGA has been used to reinterpret many previous Gray code results that were originally described recursively. For example, see the Lucas–Roelants van Baronaigien-Ruskey tree rotation Gray code [28] described in [15].



(a) Edge-labeled flip graph. **(b)** Partial path from 000. **(c)** Choose next edge. **(d)** Completed path.

Figure 4 Algorithm F is run on the hypercube in (a) starting at node 000. The partial path after five steps is shown in (b). The next step is shown in (c) where the edges incident to the most recently added node 111 are considered; the largest label that leads to a new node is 2, so that edge is selected. It halts in (d) with a Hamilton path that follows the binary reflected Gray code.

[2] This reformulation is too strong as it can generate any Hamilton path (i.e., label edges on the path with 1 and others with 0). The algorithm is intended to be used with simple locally-defined labels.

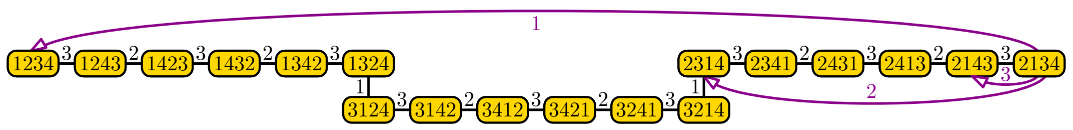


Figure 5 Algorithm F does not work on the flip graph of S_n with swaps (i.e., the permutahedron) with the (smaller) index of the swap as edge labels. When $n = 4$ the above path of length 18 is created from the initial node 1234. The algorithm terminates at 2134 because each of its edges is incident to a previously visited node. Note that none of the vertices starting with 4 were visited.

The GGA has also been used to create new Gray codes. One of the most striking new results is a simple greedy algorithm for generating an exchange Gray code for the bases of any matroid [31]. In particular, the spanning trees of any graph are generated by giving its edges distinct labels and then by greedily performing any edge exchange that minimizes the larger of the two labels. When approaching a new problem like Conjecture 1 we can try labeling the flip graph's edges (i.e., prioritizing the flips) in various ways. But the "correct" prioritization may not be obvious, so it is important to be creative and exercise patience. For example, the second author tried hundreds of greedy approaches for generating spanning trees before the simple solution described above was found by Merino [31] (also see [32]).

The GGA is inefficient as it remembers visited objects. However, it tends to produce simple orders that can be re-expressed to support efficient generation. We aim for *history-free* (i.e., previous objects are not remembered) and *iterative* (i.e., non-recursive) algorithms. Prior examples of this involve pancake flipping for signed and colored permutations [47, 5].

3 Wiggly Concepts

This section formally defines the objects and flip operations in our Gray code. The latter requires a discussion of the weak order. Additional motivation for our results is then provided.

3.1 Wiggly Permutations

Wiggly permutations restrict the values that are between *partners* which are $(2j-1, 2j)$ pairs. When the smaller odd-value is to the left of its larger even-value partner, then the values between them are larger. In contrast, if the larger even-value is to the left of the smaller odd-value, then the values between them are smaller. This is formalized in Definition 3.

▶ **Definition 3.** *A permutation π over $[n]$ is* wiggly *if the following two points hold for all pairs of values the form $(2j-1, 2j)$.*

- *Upward order. If $2j-1$ is before $2j$ in π, then symbols between them in π are larger.*
- *Downward order. If $2j$ is before $2j-1$ in π, then symbols between them in π are smaller.*

In other words, partners in their *upward order* can only have larger values between them, while those in their *downward order* can only have smaller values between them. Note that we allow n to be odd in Definition 3 (cf. [1]), and in this case n doesn't have a partner.

The relatively simple conditions in Definition 3 cannot be expressed using standard concepts from pattern avoidance [2]. However, they forbid pairs of partners from interacting with each other in particular manner. More specifically, a pair of partners in upward order can be beside each other, or nested with the larger partnership inside. For example, the partners $(1, 2)$ and $(3, 4)$ can appear in upward order as $-1-2-3-4-$, or $-3-4-1-2-$, or $-1-3-4-2-$. However, they cannot be interlaced as in $-1-3-2-4-$ or $-3-1-4-2-$. This is formalized for upward and downward orders in the following lemma.

▶ **Lemma 4.** *Let π be a wiggly permutation with distinct partnerships $\{u, u'\} = \{2x-1, 2x\}$ and $\{v, v'\} = \{2y-1, 2y\}$. If $\pi = -u-v-u'-v'-$, then the partners have opposite order.*

Proof. If $u < v$ then both partners cannot be upward by $v{-}u'{-}v'$, nor downward by $u{-}v{-}u'$. If $u > v$ then both partners cannot be downward by $v{-}u'{-}v'$, nor upward by $u{-}v{-}u'$. ◀

Let $W_n \subseteq S_n$ be the set of wiggly permutations over $[n]$. For example, $W_2 = \{12, 21\}$ and the sets for $n = 3, 4$ appear below; Table 2 contains W_5 and W_6 in our wiggly changes order.

$$W_3 = \{123, 132, 213, 312, 321\}$$
$$W_4 = \{1234, 1243, 1342, 1423, 1432, 2134, 2143, 3412, 3421, 4123, 4132, 4213, 4312, 4321\}$$

The number of wiggly permutations is not yet known in general. However, the counts for small n have been computed via recurrences on generating functions [1] and the sequence does not appear in the Online Encyclopedia of Integer Sequences (Oeis) [18].

Table 1 The number of wiggly permutations $|W_n|$ for small n.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	5	14	51	176	807	3232	17449	78384	479897	2366248	16041147	85534176	631596455	3602770400

3.2 Weak Order

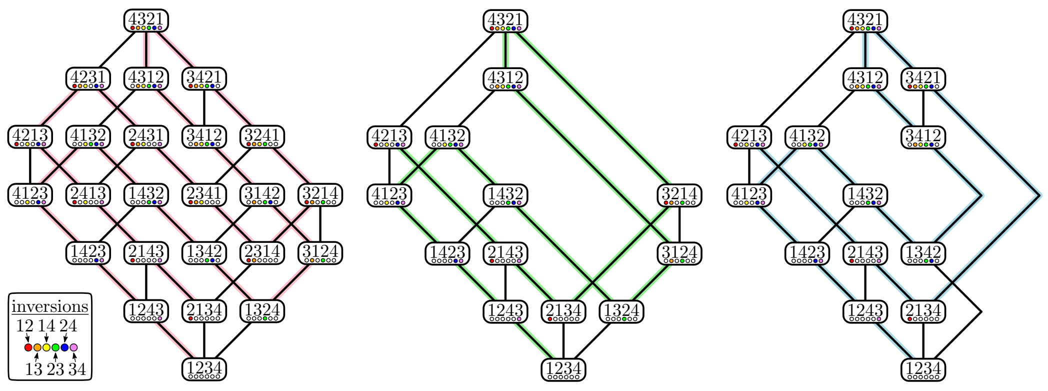
An *inversion* in a permutation $\pi = \pi_1\pi_2\cdots\pi_n$ is a pair of values that is out of order relative to the identity $12\cdots n$. In other words, it is a smaller value that appears to the right of a larger value. The *inversion set* of a permutation is its set of inversions.

$$\mathsf{inv}(\pi_1\pi_2\cdots\pi_n) = \{(\pi_i, \pi_j) \mid \pi_i < \pi_j \text{ and } i > j\} \tag{1}$$

Each permutation of S_n is uniquely determined by its inversion set.

A swap changes the inversion set by adding or removing one inversion. More specifically, if the larger value swaps to the left, then an inversion is added, and if it swaps to the right, then an inversion is removed. For example, $\mathsf{inv}(52\overline{4}\underline{1}36) \setminus \{(1,4)\} = \mathsf{inv}(521436)$. In this way, plain changes is a Gray code that changes the inversion set by one inversion per operation. More broadly, the number of inversions added or removed by a jump is equal to the length of the jump. For example, $\mathsf{inv}(52\overline{4}\underline{13}6) \setminus \{(1,4),(3,4)\} = \mathsf{inv}(521346)$. Furthermore, Algorithm J only makes *minimal jumps* meaning that the jump is as short as possible to create another permutation in the underlying language. Thus, Algorithm J also produces Gray codes that change the inversion set in a minimal way. The same will be true for the wiggly flip operations that we apply to wiggly permutation. More specifically, it will add or remove a minimal subset of inversions to create another wiggly permutation. To formalize and contextualize this idea we will introduce some additional background material. Then we will provide a direct definition of a wiggly flip in Definition 5. A similar presentation is provided in [1].

The *weak order* of permutations is a partial order based on setwise inclusion of inversion sets. In other words, if $\pi, \sigma \in S_n$ then we consider $\pi < \sigma$ if $\mathsf{inv}(\pi) \subsetneq \mathsf{inv}(\sigma)$. Similarly, distinct permutations are incomparable if neither inversion set is a subset of the other. Given a subset $S \subseteq S_n$, the *Hasse diagram* is a graph whose vertices are S and whose edges are the cover relations, where π is *covered* by σ if $\mathsf{inv}(\pi) \subsetneq \mathsf{inv}(\sigma)$ and $\nexists \tau$ where $\mathsf{inv}(\pi) \subsetneq \mathsf{inv}(\tau) \subsetneq \mathsf{inv}(\sigma)$. In other words, $\pi < \sigma$ and there is no other permutation τ with between them $\pi < \tau < \sigma$. When $S = S_n$ (i.e., all permutations) the cover relations are swaps. Similarly, when S is the set of 231-avoiding permutations $\mathsf{Av}_n(231)$, the cover relations are minimal jumps. Likewise, when $S = W_n$ (i.e., wiggly permutations) the cover relations are *wiggly flips*. In other words, our Gray code operations involve adding or removing minimal subsets of inversions. Figure 6 illustrates Hasse diagrams for the three aforementioned subsets of permutations of S_4.



(a) All permutations S_4. **(b)** Pattern avoiding $\mathsf{Av}_4(231)$. **(c)** Wiggly permutations W_4.

Figure 6 Hasse diagrams for sets of permutations under the weak order. The cover relations are (a) swaps, (b) minimal jumps, and (c) wiggly flips. The graphs are isomorphic to those in Figures 2 and 9. The dots indicate which pairs of symbols are inverted in each permutation as per the legend in (a). For example, the hop between $\underline{21}\overline{34}$ and $\overline{34}\underline{21}$ in (c) changes the number of inversions by ± 4.

These concepts have lovely generalizations to s-permutations [6], and this paper inspired a new Hamilton path in the s-permutahedron [3]. Readers may enjoy other classic and new works on related concepts including the weak order and associahedron [53, 40, 41, 25, 36, 37].

3.3 Wiggly Flips

Wiggly flips are defined in terms of cover relations, but this is somewhat of an indirect description. We now develop some intuition on our way to a direct definition. As a first step, note that the graphs in Figure 6 are 3-regular (i.e., each vertex has degree 3). More generally, all three are $(n-1)$-regular graphs as vertices have one neighbor associated with each *ascent* (i.e., $\pi_{j-1} < \pi_j$) and *descent* (i.e., $\pi_{j-1} > \pi_j$). We discuss this below; see [1] for proofs.

Note that $12 \cdots n \in W_n$ and $n \cdots 21 \in W_n$. In other words, the permutations with no inversions and all inversions are wiggly. Therefore, if we add one inversion (i.e., swap an ascent), then we can continue "up" to at least one vertex in the Hasse diagram. Similarly, if we remove one inversion (i.e., swap a descent), then we can continue "down" to at least one vertex. Thus, there is at least one cover relation associated with each ascent and descent.

To illustrate this point, the neighborhood of $\pi = 6\,4\,1\,3\,5\,7\,9\,10\,8\,2 \in W_{10}$ is in Figure 7. Consider its ascent $\mathbf{5}\,\mathbf{7}$. By our discussion, π is covered by at least one wiggly permutation whose inversion set includes $\mathsf{inv}(\pi) \cup \{(5,7)\}$. Now consider the partners of $\mathbf{5}$ and $\mathbf{7}$. Note that the smaller partners $(\mathbf{5}, 6)$ are downward and the larger partners $(\mathbf{7}, 8)$ are upward in π. Thus, the symbols in $6\,4\,1\,3\,\mathbf{5}$ cannot be between $\mathbf{7}$ and 8, and similarly, the symbols in $\mathbf{7}\,9\,10\,8$ cannot be between 6 and $\mathbf{5}$. So adding inversion $(\mathbf{5}, \mathbf{7})$ forces the run $6\,4\,1\,3\,\mathbf{5}$ to move left of the run $\mathbf{7}\,9\,10\,8$. The transposition of these two abutting runs is shown below.

$$\pi = 6\,4\,1\,3\,\mathbf{5}\,\mathbf{7}\,9\,10\,8\,2 \qquad \underline{6\,4\,1\,3\,\mathbf{5}}\,\overline{\mathbf{7}\,9\,10\,8}\,2 = \mathbf{7}\,9\,10\,8\,6\,4\,1\,3\,\mathbf{5}\,2 = \sigma. \tag{2}$$

Note that this change adds a total of $5 \cdot 4 = 20$ inversions (i.e., every value in $7\,9\,10\,8$ is inverted with every value in $6\,4\,1\,3\,5$) without removing any, so $\mathsf{inv}(\pi) \subsetneq \mathsf{inv}(\sigma)$ and $\pi < \sigma$. With some additional reasoning, we can conclude that this is the unique minimal addition in terms of inversions. In other words, there is no other wiggly τ with $\mathsf{inv}(\pi) \subsetneq \mathsf{inv}(\tau) \subsetneq \mathsf{inv}(\sigma)$. To briefly justify this, note that the internal values in $\overline{\mathbf{7}\,9\,10\,8}$ must move left with 8 since existing inversions $(8, 9), (8, 10) \in \mathsf{inv}(\pi)$ cannot be removed. Similarly, the internal values in

4 1 3 5 must move right with 4 lest additional inversions (1, 4) or (3, 4) be added. Finally, neither run needs to move past additional values. In particular, the runs cannot improperly interlace with another partnership otherwise the initial permutation violated Lemma 4.

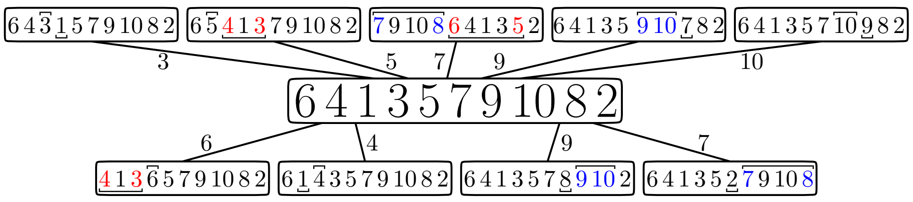


Figure 7 Neighbors of wiggly permutation $\pi = 6\,4\,1\,3\,5\,7\,9\,10\,8\,2 \in W_{10}$ in the wiggly flip graph. The neighbors are drawn above or below to match the Hasse diagram. There are five jump edges (including one swap edge) where there is one larger value, and two hop edges where there are two larger values; the other two edges are guaranteed not be used in our Hamilton path (see Theorem 2). The edges are drawn into π to highlight their associated ascent or descent. However, they are labeled by the minimum larger value (see Definition 6) for use with Algorithm F. The downward and upward partners are colored when there is more than one smaller and larger value, respectively.

More generally, if $\pi_{j-1}\pi_j$ is an ascent and π_j begins an upward order partnership, then the associated edge will move that entire run to the left; otherwise, only π_j moves left. Similarly, if π_{j-1} ends a downward order partnership, then the associated edge moves the entire run to the right; otherwise, only π_{j-1} moves right. This is formalized for the edges associated with ascents in Definition 5; edges associated with descents are defined in reverse.

▶ **Definition 5.** *Let $\pi = \pi_1\pi_2\cdots\pi_n \in W_n$ be a wiggly permutation with $u = \pi_{j-1} < \pi_j = v$ being one of its ascents*[3] *and let $\pi^{-1} = q_1q_2\cdots q_n$ be π's inverse (i.e., $q_x = y \iff \pi_y = x$). Define two runs $\alpha = \pi_i\pi_{i+1}\cdots\pi_{j-1}$ and $\beta = \pi_j\pi_{j+1}\cdots\pi_k$ where*

- *$i = q_{u+1}$ if u is odd and $q_{u+1} < q_u$; otherwise, $i = j - 1$.*
- *$k = q_{v+1}$ if v is odd and $q_{v+1} > q_v$; otherwise, $k = j$.*

The wiggly (left) flip *associated with this ascent moves β to the left over α. That is, $\pi_1\pi_2\cdots\pi_{i-1}\,\underline{\alpha}\,\overline{\beta}\,\pi_{k+1}\pi_{k+2}\cdots\pi_n = \pi_1\pi_2\cdots\pi_{i-1}\beta\alpha\pi_{k+1}\pi_{k+2}\cdots\pi_n$. If $|\beta| = 1$ (i.e., $j = k$), then the flip is a* wiggly jump*. If $|\beta| = 2$ (i.e., $j = k - 1$), then the flip is a* wiggly hop.

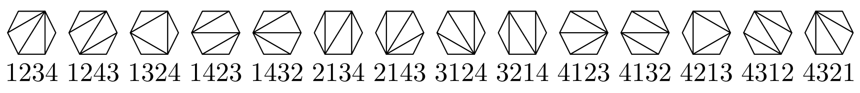
Note that wiggly flips are always minimal in the sense that β moves the shortest possible distance to create a new wiggly permutation. In particular, wiggly jumps are identical to the notion of minimal jumps in the *Permutation Languages* series [16].

3.4 Motivation: Why Wiggly Permutations and Wiggly Flips?

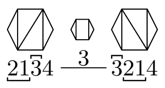
Wiggly permutations are a new combinatorial object. However, Lemma 4 is reminiscent of pairings in certain Catalan objects like well-formed parentheses [50]. This similarity arises in the context of triangulations. Triangulations of a convex polygon are counted by the Catalan numbers [50]. Similarly, wiggly numbers $|W_{2n}|$ count wiggly triangulations [1]. See Figure 8. Many other connections involving wiggly permutations are discussed in [1].

Historically, Gray code research has focused on array operations like swaps and transpositions [34]. This is in large part due to the use of arrays in efficient generation algorithms [42]. For example, the first investigation into efficiently generating pattern avoiding permutation Gray codes allowed a small number of permutation entries to change (i.e., four for $\mathsf{Av}_n(231)$) [9].

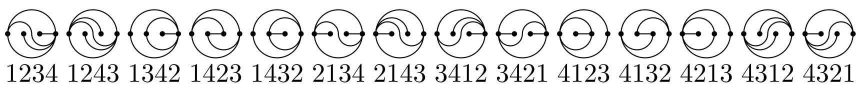
[3] Note that Lemma 31 in [1] instead indexes ascents as $\pi_j < \pi_{j+1}$.



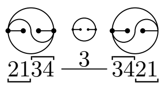
(a) 231-avoiding permutations and their corresponding triangulations.



(b) A jump edge.



(c) Wiggly permutations and their corresponding wiggly triangulations.



(d) A hop edge.

Figure 8 Permutation languages representing other combinatorial objects. The flips in (b) and (d) correspond to removing then adding a (wiggly) diagonal in the corresponding (wiggly) triangulation.

Our paper is the first such investigation into wiggly permutations, so it is natural to ask why we consider wiggly flips (which can change *every* entry). One response is that wiggly flips arise in two natural graphs (see Figure 2c and 6c). Another is that wiggly flips flip diagonals in wiggly triangulations (see Figure 8d). However, the true answer is much deeper.

The *Permutation Languages* series [16] broadened interest in Gray code operations that are computationally motivated to those that are mathematically motivated. One pleasant consequence has been natural Gray codes for related combinatorial objects. Given two combinatorial objects and a bijection between them, a *simultaneous Gray code* is a single order that is a Gray code for both. To make this concrete, consider balanced parenthesis strings of length $2n$ and binary trees with n nodes [50]. Making a constant-size change in the string (e.g., transposing a pair of symbols) can cause a non-constant change (e.g., $\omega(1)$ pointer changes) in the corresponding trees. Thus, Gray codes for the former [44, 4, 56] and the latter [28] were not simultaneous. A Gray code known as cool-lex order [46] was shown to be simultaneous [45, 10, 26], however, it doesn't apply as naturally to other Catalan objects [8]. In contrast, the orders from the *Permutation Languages* series appear almost universal in their application. For example, Figure 5 in [16] shows that Algorithm J's Gray code for $\mathsf{Av}_n(231)$ is a simultaneous Gray code for binary trees, triangulations, and Dyck paths.

Wiggly permutations are also an ideal challenge for extending the *Permutation Languages* series. It was observed that wiggly permutations do not form a *zig-zag language* [1]. This means that wiggly permutations are not closed under inserting a largest value on the left and the right. In other words, if $\pi \in W_{n-1}$, then it is not always true that (i) $n\pi \in W_n$ and (ii) $\pi n \in W_n$. Thus, the results of [16] do not apply to W_n. But (i) is true for even n, and both (i) and (ii) are true for odd n. Indeed, the $n = 5$ column of Table 2 uses the same local recursion as Algorithm J for zig-zag languages. Furthermore, the $n = 6$ columns illustrate a similar local recursion except that the value of 5 acts as a right delimiter or sign-post for 6. We hope that our results will spur a new generalization of the *Permutations Languages* series.

Finally, our Gray code might be very efficient in terms of array-based generation. It only uses wiggly jumps and wiggly hops, so its efficiency depends only on the expected lengths of these operations. Theorem 9 provides a worst-case analysis of our efficient algorithm; further analysis may reveal that it runs in *constant amortized time* (i.e., amortized $\mathcal{O}(1)$-time delay). In this way, wiggly flips may mirror prefix-shifts [59]: the operation is inefficient in general, but Gray codes may use them efficiently by restricting the average length [57], biasing or restricting the lengths [7, 52, 48], or using alternate data structures [27].

4 Wiggly Changes

Now we describe *wiggly changes*. The Gray code was discovered greedily, and we start by discussing our experiments with Algorithm F. Then we define the local recursion that arose from a successful experiment. Finally, we prove that the order only uses jumps and hops.

4.1 Choosing Edge Labels

The wiggly flip graph consists of wiggly permutations under wiggly flips. To apply Algorithm F from Section 2 we must choose an initial node (e.g., $12\cdots n \in W_n$) and provide an edge labeling. Wiggly flips correspond to indices (i.e., one edge per ascent or descent $\pi_{j-1}\pi_j$) and the index j would provide natural edge labels. Indeed, this choice works for $n \leq 4$

$$12\underbracket{3}\overbracket{4}, 1\underbracket{2}\overbracket{4}3, 14\underbracket{2}\overbracket{3}, 1\overbracket{4}\underbracket{3}2, 1\overbracket{34}\underbracket{2}, 34\underbracket{1}\overbracket{2}, \overbracket{34}\underbracket{21}, 21\underbracket{3}\overbracket{4}, \underbracket{21}\overbracket{4}3, 4\underbracket{21}\overbracket{3}, 43\overbracket{2}\underbracket{1}, 4\overbracket{3}\underbracket{1}2, 41\overbracket{3}\underbracket{2}, 4123 \tag{3}$$

and (3) differs from Table 2. However, it stops working for larger n with only $\frac{32}{51}$ and $\frac{115}{176}$ objects generated for $n = 5$ and $n = 6$. This edge labeling also fails on the permutahedron.

Plain changes was previously described greedily as "swap the largest value" while Algorithm J "jumps the largest value". This is equivalent to using Algorithm F with the larger relocated value as the edge label. We cannot use this exact labeling as wiggly flips can move more than one larger value. However, a simple generalization is the *minimum larger value.*

▶ **Definition 6.** *If $\pi, \sigma \in S_n$ and $\mathsf{inv}(\pi) \subsetneq \mathsf{inv}(\sigma)$, then the* minimum larger label *of the edge (π, σ) is the minimum value $v \in [n]$ such that there exists $u \in [n]$ with $(u, v) \in \mathsf{inv}(\sigma) \smallsetminus \mathsf{inv}(\pi)$.*

Definition 6 is illustrated in Figure 7. We ran Algorithm F with these labels and observed that it worked! That is, every wiggly permutation is generated starting from $12\cdots n$. Equivalently, it creates a Hamilton path in the wigglyhedron starting from $12\cdots n$. By the above discussion, the same edge labeling allows Algorithm F to generate the previously discovered Hamilton paths in the permutahedron and associahedron. See Figure 9.

4.2 Zig-Zag Local Recursion

Given a permutation π of $[n]$, we let the parent $\mathsf{p}(\pi)$ be the result of removing n from π to create a permutation of $[n-1]$. In the opposite direction, given a permutation π of $[n-1]$ and an index $i \in [n]$, we let the child $\mathsf{c}_i(\pi)$ be the result of inserting n at position i to create a permutation of $[n-1]$. For example, $\mathsf{c}_3(615243) = 6175243$ and $\mathsf{p}(6175243) = 615243$.

Now we consider the family or hierarchy of wiggly permutation languages $W_0, W_1, W_2, \ldots$. It is easy to see that this family has the following properties.

1. If $\pi \in W_n$ and $n \geq 1$, then $\mathsf{p}(\pi) \in W_{n-1}$.
2. If $\pi \in W_{n-1}$, then $\mathsf{c}_1(\pi) = n \cdot \pi \in W_n$.

In other words, a new wiggly permutation is created by removing the largest value or by inserting a new largest value on the left. Given a family with property 1, we can construct an ordering of each language using a *zig-zag local recursion.* Towards this goal we recall $\overleftarrow{c}(\pi)$ ('zig') and $\overrightarrow{c}(\pi)$ ('zag') from [16]. If π is a permutation of $[n-1]$, then $\overleftarrow{c}(\pi)$ is the list $\mathsf{c}_1(\pi), \mathsf{c}_2(\pi), \ldots, \mathsf{c}_n(\pi)$ except that every permutation that is not in the associated language is skipped over. Similarly, $\overrightarrow{c}(\pi)$ is $\mathsf{c}_n(\pi), \mathsf{c}_{n-1}(\pi), \ldots, \mathsf{c}_1(\pi)$ with invalid entries skipped. Zig-zag local recursion expands each permutation of $[n-1]$ alternately using zigs and zags.

▶ **Definition 7.** *The* wiggly changes *order of W_n uses zig-zag local recursion:* $\mathsf{wiggly}(1) = 1$ *and if* $\mathsf{wiggly}(n-1) = \pi_1, \pi_2, \pi_3, \pi_4, \ldots$ *then* $\mathsf{wiggly}(n) = \overleftarrow{c}(\pi_1), \overrightarrow{c}(\pi_2), \overleftarrow{c}(\pi_3), \overrightarrow{c}(\pi_4), \ldots$.

Table 2 Wiggly changes for $n = 4, 5, 6$ and the edge labels that are greedily selected by Algorithm F to create the next wiggly permutation. Note that the local recursion from $n = 4$ to $n = 5$ is identical to Algorithm J. That is, the largest value 5 jumps into every valid position from the rightmost to leftmost, then vice versa, through successive permutations of $n = 4$. The local recursion from $n = 5$ to $n = 6$ is similar, except that the rightmost allowable position of the largest value 6 is constrained by the position of its partner 5. (This does not interfere with local recursion because if 5 has a smaller symbol to its left and to its right, then the next transition in the $n = 5$ order is a jump of 5. In other words, the larger symbols don't get in the way of transitions made in the $n = 4$ order.) Finally, notice that some jumps of 5 in the $n = 5$ order become hops of 56 in the $n = 6$ order.

1234	4	12345	5	123456	6	641325	5	346152	6	213564	6
1243	4	12354	5	123465	6	641352	6	341652	6	213654	6
1423	4	12534	5	123645	6	413652	6	341562	5	216354	6
4123	3	15234	5	126345	6	413562	5	341256	6	621354	5
4132	4	51234	4	162345	6	564132	6	341265	6	621345	6
1432	4	51243	5	612345	5	654132	4	341625	6	216345	6
1342	3	15243	5	612354	6	651432	6	346125	6	213645	6
3412	4	12543	5	162354	6	561432	5	364125	6	213465	6
4312	2	12435	4	126354	6	156432	6	634125	4	213456	4
4321	4	14235	5	123654	6	165432	6	643125	6	214356	6
3421	3	15423	5	123564	5	615432	5	436125	6	214365	6
2134	4	51423	4	125634	6	614352	6	431625	6	216435	6
2143	4	54123	5	126534	6	164352	6	431265	6	621435	5
4213		41235	3	162534	6	143652	6	431256	5	621543	6
		41325	5	612534	5	143562	5	431562	6	216543	6
		41352	5	615234	6	143256	6	431652	6	215643	5
		54132	4	165234	6	143265	6	436152	6	562143	6
		51432	5	156234	5	143625	6	643152	5	652143	4
		15432	5	561234	6	164325	6	643512	6	654213	6
		14352	5	651234	4	614325	4	436512	6	564213	5
		14325	4	651243	6	613425	6	435612	5	421356	6
		13425	5	561243	5	163425	6	564312	6	421365	6
		13452	5	156243	6	136425	6	654312	2	642135	
		13542	5	165243	6	134625	6	654321	6		
		15342	5	615243	5	134265	6	564321	5		
		51342	3	612543	6	134256	5	435621	6		
		53412	5	162543	6	134562	6	436521	6		
		35412	5	126543	6	134652	6	643521	5		
		34512	5	125643	5	136452	6	643215	6		
		34152	5	124356	6	163452	6	436215	6		
		34125	4	124365	6	613452	5	432165	6		
		43125	5	126435	6	613542	6	432156	4		
		43152	5	162435	6	163542	6	342156	6		
		43512	5	612435	4	136542	6	342165	6		
		54312	2	614235	6	135642	5	346215	6		
		54321	5	164235	6	156342	6	364215	6		
		43521	5	142365	6	165342	6	634215	5		
		43215	4	142356	5	615342	5	634521	6		
		34215	5	156423	6	651342	6	364521	6		
		34521	5	165423	6	561342	3	346521	6		
		35421	5	615423	5	563412	6	345621	5		
		53421	3	651423	6	653412	5	356421	6		
		52134	5	561423	4	635412	6	365421	6		
		21534	5	564123	6	365412	6	635421	5		
		21354	5	654123	5	356412	5	653421	6		
		21345	4	641235	6	345612	6	563421	3		
		21435	5	412365	6	346512	6	562134	6		
		21543	5	412356	3	364512	6	652134	5		
		52143	4	413256	6	634512	5	621534	6		
		54213	5	413265	6	634152	6	216534	6		
		42135		413625	6	364152	6	215634	5		

For example, the order of wiggly permutations is extended from $n = 3$ to $n = 4$ as follows.

$$\begin{aligned}\mathsf{wiggly}(3) &= 123, 132, 312, 321, 213\\ \mathsf{wiggly}(4) &= \overleftarrow{c}(123), \overrightarrow{c}(132), \overleftarrow{c}(312), \overrightarrow{c}(321), \overleftarrow{c}(213)\\ &= \underbrace{1234, 1243, 1423, 4123}_{\overleftarrow{c}(123)}, \underbrace{4132, 1432, 1342}_{\overrightarrow{c}(132)}, \underbrace{3412, 4312}_{\overleftarrow{c}(312)}, \underbrace{4321, 3421}_{\overrightarrow{c}(321)}, \underbrace{2134, 2143, 4213}_{\overleftarrow{c}(213)}\end{aligned}$$

Note that $\overrightarrow{c}(132)$ ends with $13\mathbf{4}2$ (and not $132\mathbf{4}$ as in a zig-zag language) while $\overleftarrow{c}(312)$ begins with $3\mathbf{4}12$. So the transition between these sublists is the hop $\underline{1}\overline{34}2$ to $\overline{34}\underline{1}2$. The next lemma describes the beginning and end of each zig and zag and follows from Definition 3.

▶ **Lemma 8.** *Let* $\pi = \pi_1\pi_2\cdots\pi_{n-1} \in \mathsf{wiggly}(n-1)$ *with* $\pi_m = n-1$. *If* n *is odd, then*

$$\overrightarrow{c}(\pi) = \mathsf{c}_1(\pi), \ldots, \mathsf{c}_n(\pi) \text{ and } \overleftarrow{c}(\pi) = \mathsf{c}_n(\pi), \ldots, \mathsf{c}_1(\pi). \tag{4}$$

In other words, these zigs and zags result in the largest symbol moving from the first position to the last position (or vice versa) as in a zig-zag language. However, if n *is even, then*

$$\overrightarrow{c}(\pi) = \mathsf{c}_1(\pi), \ldots, \mathsf{c}_{m+1}(\pi) \text{ and } \overleftarrow{c}(\pi) = \mathsf{c}_{m+1}(\pi), \ldots, \mathsf{c}_1(\pi). \tag{5}$$

In other words, these zigs and zigs result in the largest symbol moving from the first position to the position immediately to the right of the next largest symbol (or vice versa).

The flip graph of wiggly permutations and wiggly flips is $\mathbb{W}_n$. We now prove Theorem 2.

Proof. We prove the result by induction on $n \geq 1$ with $\mathsf{wiggly}(1) = 1$ as a base case.
Assume that $\mathsf{wiggly}(n-1) = \pi_1, \pi_2, \ldots$ is a Hamilton path in $\mathbb{W}_{n-1}$ and consider

$$\mathsf{wiggly}(n) = \overleftarrow{c}(\pi_1), \overrightarrow{c}(\pi_2), \overleftarrow{c}(\pi_3), \overrightarrow{c}(\pi_4), \ldots.$$

It is clear that $\mathsf{wiggly}(n)$ is an ordering of W_n. The transitions within each $\overleftarrow{c}$ and $\overrightarrow{c}$ sublist are jumps. So we need only consider the transitions between consecutive sublists. Let $\sigma = \pi_i$ and $\tau = \pi_{i+1}$ so that we can index their entries using subscripts.
We first consider transitions between sublists of the form $\overleftarrow{c}(\sigma)$ and $\overrightarrow{c}(\tau)$. By Lemma 8,

$$\overleftarrow{c}(\sigma),\ \overrightarrow{c}(\tau) = \ldots, \mathsf{c}_1(\sigma),\ \mathsf{c}_1(\tau), \ldots = \ldots, n\sigma,\ n\tau, \ldots$$

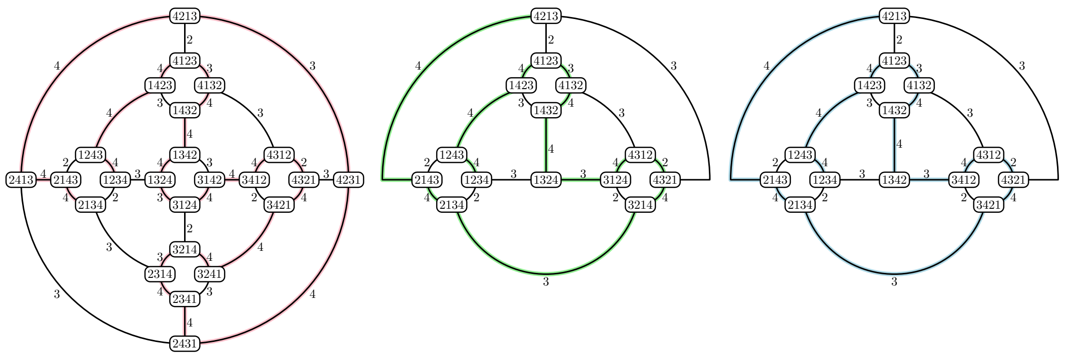
By induction, (σ, τ) is a jump/hop edge of $\mathbb{W}_{n-1}$. Thus, $(n\sigma, n\tau)$ is a jump/hop edge of $\mathbb{W}_n$.
Next consider transitions between sublists of the form $\overrightarrow{c}(\sigma)$ and $\overleftarrow{c}(\tau)$. We consider two cases depending on the parity of n. If n is odd, then by Lemma 8,

$$\overrightarrow{c}(\sigma),\ \overleftarrow{c}(\tau) = \ldots, \mathsf{c}_n(\sigma),\ \mathsf{c}_n(\tau), \ldots = \ldots, \sigma n,\ \tau n, \ldots$$

By induction, (σ, τ) is a jump/hop edge of $\mathbb{W}_{n-1}$. Thus, $(\sigma n, \tau n)$ is a jump/hop edge of $\mathbb{W}_n$. If n is even, then let $\sigma_m = n-1$ and $\tau_{m'} = n-1$. In other words, the largest symbol $n-1$ is at index m in $\sigma \in W_{n-1}$, and at index m' in $\tau \in W_{n-1}$. Thus, by Lemma 8,

$$\overrightarrow{c}(\sigma),\ \overleftarrow{c}(\tau) = \ldots, \mathsf{c}_{m+1}(\sigma),\ \mathsf{c}_{m'+1}(\tau), \ldots.$$

By induction, (σ, τ) is a jump/hop edge of $\mathbb{W}_{n-1}$. Since n is even, we know that $n-1$ is odd. Thus, if $n-1$ moves between σ and τ, then the change between them is a jump of $n-1$. Therefore, the change between $\mathsf{c}_m(\sigma)$ and $\mathsf{c}_{m'}(\tau)$ is a hop of $n-1$. If $n-1$ does not move between σ and τ, then it must be that $m = n-1$ and $m' = n-1$ since the unpartnered odd value $n-1$ can always be inserted into the rightmost position. Therefore, the change between $\mathsf{c}_n(\sigma)$ and $\mathsf{c}_n(\tau)$ in $\mathbb{W}_n$ is the same as the change between σ and τ in $\mathbb{W}_{n-1}$. ◀



(a) S_4 with swaps. **(b)** $\mathsf{Av}_4(231)$ with minimal jumps. **(c)** W_4 with wiggly flips.

Figure 9 Flip graphs including minimum larger value edge labels (Definition 6). The graphs with unlabeled edges and the highlighted Hamilton paths are isomorphic to those in Figures 2 and 6. The paths are generated greedily by Algorithm F. These embeddings show that (b) and (c) are isomorphic (and the edge labels on the Hamilton paths are identical) but this is not true for $n > 4$.

5 Efficient Generation

Now we efficiently generate wiggly changes. Our algorithm differs from the greedy construction as it is *history-free* (i.e., it doesn't remember previous objects). It also iterative, so it differs from a direct translation of the zig-zag local recursion. As is customary in generation algorithms, we store only one object (i.e., one wiggly permutation) and `visit` (i.e., *yield*) it every time it is modified to be the next object in the order. Each successive wiggly permutation is generated in worst-case $\mathcal{O}(n)$-time. Appendix A has a complete Python implementation.

We begin by reviewing a well-known *loopless algorithm* for generating plain changes. This means that each successive permutation is visited with worst-case $\mathcal{O}(1)$-time delay. The approach was devised by Ehrlich [11] and a modern treatment appears in Knuth's Algorithm P (*Plain changes*) and Algorithm H (*Loopless reflected mixed-radix Gray generation*) [20].

5.1 Generating Plain Changes

The main idea is to generate the *inversion word* that encodes each permutation. Inversion words are mixed-radix words with bases $1, 2, \dots, n$ of which there are $n!$. The ith digit of such a word encodes the number of values that are smaller than i and which are inverted with it in its corresponding permutation. The words are generated in standard *reflected Gray code order* (i.e., each digit increments from 0 to its maximum, then decrements from its maximum to 0). Each swap in a permutation changes its inversion word by ± 1 in one digit and vice versa. Pleasingly, this Gray code of inversion words corresponds to plain changes.

To generate the inversion words looplessly we store a direction for each digit, or equivalently, a direction in which each value is moving in the permutation. *Focus pointers* determine which digit changes (i.e., which value is swapped) in constant time; their name is somewhat misleading as they are an array of integers. The inverse of the current permutation is maintained to determine the location of the value to move in worst-case $\mathcal{O}(1)$-time. Our presentation of this classic algorithm appears in Algorithm 1 and it is organized to highlight the similarities with our wiggly changes algorithm. In particular, we use `swapLeft` and `swapRight` functions to update the permutation, its inverse, and its inversion word. The basic approach has previously been modified to generate other permutation Gray codes [17, 13] and Gray codes for other types of permutations [39].

Algorithm 1 Plain changes generated in worst-case $\mathcal{O}(1)$-time delay. Each array has length n (with 1-based indexing) and is traced at the **while** (line 8) for $\mathtt{n} = 4$ in the table below. The operations are swaps $\cdots \underbrace{\pi_{i-1}\overline{\pi_i}} \cdots$ (left) or $\cdots \underbrace{\pi_i\overline{\pi_{i+1}}} \cdots$ (right) with edge label $\mathtt{v}$ as per Definition 6. The table highlights edge labels $\mathtt{v} = \mathtt{fs}[4]$; see Algorithm 2's table for more information on each array.

```
function swapLeft(perm, v, inv, word)
1: i ← inv[v]                    ▷ index of edge label v
2: u ← perm[i − 1]               ▷ smaller value to swap
3: word[v] ← word[v]+1           ▷ new inversion (u, v)
4: perm[i] ← u                   ▷ swap u right
5: perm[i − 1] ← v               ▷ swap v left
6: inv[u] ← i                    ▷ update u's index
7: inv[v] ← i − 1                ▷ update v's index
```

```
function swapRight(perm, v, inv, word)
1: i ← inv[v]                    ▷ index of edge label v
2: u ← perm[i + 1]               ▷ smaller value
3: word[v] ← word[v]−1           ▷ old (u, v)
4: perm[i] ← u                   ▷ swap u left
5: perm[i + 1] ← v               ▷ swap v right
6: inv[u] ← i                    ▷ update u's index
7: inv[v] ← i + 1                ▷ update v's index
```

```
function plain(n)
1:  perm ← [1, 2, ..., n]              ▷ current permutation
2:  inv ← [1, 2, ..., n]               ▷ inverse permutation
3:  word ← [0, 0, ..., 0]              ▷ inversion word
4:  fs ← [1, 2, ..., n]                ▷ focus pointers
5:  dirs ← [−1, −1, ..., −1]           ▷ value directions
6:  visit perm                         ▷ first permutation
7:  v ← fs[n]                          ▷ next edge label (swap value)
8:  while v > 1 do                     ▷ no edges labeled 1
9:    if dirs[v] = −1                  ▷ leftward swap?
10:     swapLeft(perm, v, inv, word)
11:   else                             ▷ rightward swap
12:     swapRight(perm, v, inv, word)
13:   if word[v] = 0 or word[v] = v − 1     ▷ limit?
14:     dirs[v] ← −dirs[v]             ▷ change direction
15:     fs[v] ← fs[v − 1]              ▷ inherit focus pointer
16:     fs[v − 1] ← v − 1              ▷ reset focus pointer
17:   visit perm                       ▷ current permutation
18:   v ← fs[n]                        ▷ next edge label (swap value)
19:   fs[n] ← n                        ▷ reset focus pointer
```

perm	inv	word	fs	dirs
1234	1234	0000	1234	−−−−
1243	1243	0001	1234	−−−−
1423	1342	0002	1234	−−−−
4123	2341	0003	1233	−−−+
4132	2431	0013	1234	−−−+
1432	1432	0012	1234	−−−+
1342	1423	0011	1234	−−−+
1324	1324	0010	1233	−−−−
3124	2314	0020	1224	−−+−
3142	2413	0021	1224	−−+−
3412	3412	0022	1224	−−+−
4313	3421	0023	1232	−−++
4321	4321	0123	1134	−+++
3421	4312	0122	1134	−+++
3241	4213	0121	1134	−+++
3214	3214	0120	1133	−++−
2314	3124	0110	1134	−++−
2341	4123	0111	1134	−++−
2431	4132	0112	1134	−++−
4231	4231	0113	1133	−+++
4213	3241	0103	1214	−+−+
2413	3142	0102	1214	−+−+
2143	2143	0101	1214	−+−+
2134	2134	0100	1231	−+−−

5.2 Generating Wiggly Changes

Generating wiggly changes is similar to generating plain changes. The same concepts allow us to determine the value of the operation to apply (i.e., the next edge label to follow) in worst-case $\mathcal{O}(1)$-time. Moreover, we can update the resulting permutation's inverse and inversion word in worst-case $\mathcal{O}(1)$-time. Pseudocode appears in Algorithm 2 with a full Python implementation in Appendix A. Two specific challenges are highlighted below.

- The framework gives us the value and direction of the next operation. But we must determine if the operation is a jump or a hop. Similarly, we must determine how many smaller values are jumped or hopped over. Fortunately, the necessary information comes directly from whether certain partners are in their upward or downward order as per Definition 3 and Definitions 5–6.
- Each even value must change directions whenever it is swapped to be immediately to the right of its odd partner. This leads to an extra condition in line 17.

Algorithm 2 Wiggly permutations generated in worst-case $\mathcal{O}(n)$-time delay. Each array has length n (with 1-based indexing) and is traced for $\mathtt{n} = 4$ below. The operations are wiggly jumps and hops $\cdots \underbrace{\pi_i \cdots \pi_{j-1}}\overline{\pi_j \cdots \pi_k} \cdots$ (left) or $\cdots \overline{\pi_i \cdots \pi_{j-1}}\underbrace{\pi_j \cdots \pi_k}\cdots$ (right) with edge label v as per Definitions 5–6.

```
function wiggleLeft(perm, v, inv, word, hop)
 1: h ← ⟦hop⟧                       ▷ ⟦True/False⟧ is 1/0
 2: j ← inv[v]                      ▷ index of edge label v
 3:
 4: u ← perm[j−1]                   ▷ last smaller value
 5: i ← j − 1                       ▷ first smaller index?
 6: if u is odd and inv[u] ≥ inv[u+1]
 7:     i ← inv[u + 1]              ▷ correct i to partner
 8: word[v] ← word[v]+j−i           ▷ +inversions
 9: if hop = True                   ▷ partner inversions?
10:     word[v+1] ← word[v+1] + j−i
11: while j > i do                  ▷ shift smaller values
12:     perm[j + h] ← perm[j − 1]   ▷ one or two
13:     inv[perm[j − 1]] ← j + h    ▷ positions
14:     j ← j − 1                   ▷ to the right
15: perm[i] ← v                     ▷ move larger value
16: inv[v] ← i                      ▷ update its inverse
17: if hop = True                   ▷ another larger value?
18:     perm[i+1] ← v+1             ▷ move partner
19:     inv[v+1] ← i+1              ▷ update its inverse
```

```
function wiggleRight(perm, v, inv, word, hop)
 1: h ← ⟦hop⟧                       ▷ ⟦True/False⟧ is 1/0
 2: i ← inv[v]                      ▷ index of edge label v
 3: j ← i + 1 + h                   ▷ first smaller index
 4: u ← perm[j]                     ▷ first smaller value
 5: k ← j                           ▷ last smaller index?
 6: if u is even and inv[u] ≤ inv[u−1]
 7:     k ← inv[u − 1]              ▷ correct k to partner
 8: word[v] ← word[v]−k+j−1         ▷ −inversions
 9: if hop = True                   ▷ partner inversions?
10:     word[v+1] ← word[v+1]−k+j−1
11: while i < k−h do                ▷ shift smaller values
12:     perm[i] ← perm[i + 1 + h]   ▷ one or two
13:     inv[perm[i + 1 + h]] ← i    ▷ positions
14:     i ← i + 1                   ▷ to the left
15: perm[k − h] ← v                 ▷ move larger value
16: inv[v] ← k − h                  ▷ update its inverse
17: if hop = True                   ▷ another larger value?
18:     perm[k] ← v+1               ▷ move partner
19:     inv[v+1] ← k                ▷ update its inverse
```

```
function wiggly(n)
 1: perm ← [1, 2, . . . , n]        ▷ current permutation
 2: inv ← [1, 2, . . . , n]         ▷ inverse permutation
 3: word ← [0, 0, ..., 0]           ▷ inversion word
 4: fs ← [1, 2, . . . , n]          ▷ focus pointers
 5: dirs ← [−1, −1, . . . , −1]     ▷ value directions
 6: visit perm                      ▷ first wiggly permutation
 7: v ← fs[n]                       ▷ next edge label (wiggle value)
 8: while v > 1 do                  ▷ no edges labeled 1
 9:     i ← inv[v]                  ▷ index of v
10:     hop ← v is even and i < n and perm[i+1] = v+1
11:     if dirs[v] = −1             ▷ leftward jump/hop?
12:         wiggleLeft(perm, v, inv, word, hop)
13:     else                        ▷ rightward jump/hop
14:         wiggleRight(perm, v, inv, word, hop)
15:     i ← inv[v]                  ▷ new index of v
16:     pair ← v is even and perm[i−1] = v−1)
17:     if word[v] = 0 or word[v] = v−1 or pair
18:         dirs[v] ← −dirs[v]      ▷ change direction
19:         fs[v] ← fs[v−1]         ▷ inherit focus pointer
20:         fs[v−1] ← v−1           ▷ reset focus pointer
21:     visit perm                  ▷ current wiggly permutation
22:     v ← fs[n]                   ▷ next edge label (wiggle value)
23:     fs[n] ← n                   ▷ reset focus pointer
```

perm	inv	word	fs	dirs
$12\underline{3}\overline{4}$	1234	0000	1234	−−−−
$1\underline{2}\overline{4}3$	1243	0001	1234	−−−−
$\underline{1}\overline{4}23$	1342	0002	1234	−−−−
$41\underline{2}\overline{3}$	2341	0003	1233	−−−+
$\overline{4}\underline{1}32$	2431	0013	1234	−−−+
$1\overline{4}\underline{3}2$	1432	0012	1234	−−−+
$\underline{1}\overline{34}2$	1423	0011	1233	−−−−
$\underline{3}\overline{4}12$	3412	0022	1224	−−+−
$43\underline{1}\overline{2}$	3421	0023	1232	−−++
$\overline{4}\underline{3}21$	4321	0123	1134	−+++
$\overline{34}\underline{21}$	4312	0122	1133	−++−
$21\underline{3}\overline{4}$	2134	0100	1214	−+−−
$\underline{21}\overline{4}3$	2143	0101	1214	−+−−
4213	3241	0103	1231	−+−+

Variable trace at **while** (line 8) in wiggly changes wiggly(4). The edge label of next jump/hop is v = fs[4] in direction dirs[v]. Indexing and direction changes in $\mathcal{O}(1)$-time via inv and word.

The algorithm is loopless except for the time taken to apply each operation. More specifically, each jump and hop can take up to $\Theta(n)$-time because a linear number of symbols may need to be relocated in the array. This leads to the following theorem. In future work we wish to further analyze our Gray code to provide a tighter average or amortized run-time.

▶ **Theorem 9.** *Algorithm 2 generates the wiggly changes Gray code of wiggly permutations of length n with worst-case $\mathcal{O}(n)$-time delay using $\mathcal{O}(n)$ memory.*

References

1 Asilata Bapat and Vincent Pilaud. Wigglyhedra. *Mathematische Zeitschrift*, 310(3):1–41, 2025.

2 David Bevan. Permutation patterns: basic definitions and notation. *arXiv preprint*, 2015. `arXiv:1506.06673`.

3 Samuel Buick, Goertz Madeleine, Amos Lastmann, Kunal Pal, Helen Qian, Sam Tacheny, Aaron Williams, Leah Williams, and Yulin Zhai. Exhaustive generation of pattern-avoiding s-words. In *Proceedings of the 23rd International Conference on Permutation Patterns*, 2025.

4 Bette Bultena and Frank Ruskey. An Eades-McKay algorithm for well-formed parentheses strings. *Information Processing Letters*, 68(5):255–259, 1998. `doi:10.1016/S0020-0190(98)00171-9`.

5 Ben Cameron, Joe Sawada, Wei Therese, and Aaron Williams. Hamiltonicity of k-sided pancake networks with fixed-spin: Efficient generation, ranking, and optimality. *Algorithmica*, 85(3):717–744, 2023. `doi:10.1007/S00453-022-01022-X`.

6 Cesar Ceballos and Viviane Pons. The s-weak order and s-permutahedra I: Combinatorics and lattice structure. *SIAM Journal on Discrete Mathematics*, 38(4):2855–2895, 2024.

7 Peter F Corbett. Rotator graphs: An efficient topology for point-to-point multiprocessor networks. *IEEE Transactions on Parallel & Distributed Systems*, 3(05):622–626, 1992. `doi:10.1109/71.159045`.

8 Emily Downing, Stephanie Einstein, Elizabeth Hartung, and Aaron Williams. Catalan squares and staircases: Relayering and repositioning Gray codes. In *Proceedings of the 35th Canadian Conference on Computational Geometry, CCCG*, pages 273–281, 2023.

9 WMB Dukes, Mark F Flanagan, Toufik Mansour, and Vincent Vajnovszki. Combinatorial Gray codes for classes of pattern avoiding permutations. *Theoretical Computer Science*, 396(1-3):35–49, 2008. `doi:10.1016/J.TCS.2007.12.002`.

10 Stephane Durocher, Pak Ching Li, Debajyoti Mondal, Frank Ruskey, and Aaron Williams. Cool-lex order and k-ary Catalan structures. *Journal of Discrete Algorithms*, 16:287–307, 2012. `doi:10.1016/J.JDA.2012.04.015`.

11 Gideon Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *Journal of the ACM (JACM)*, 20(3):500–513, 1973. `doi:10.1145/321765.321781`.

12 Joseph Gallian. *Contemporary abstract algebra*. Chapman and Hall/CRC, 2021.

13 Pramod Ganapathi and Rezaul Chowdhury. A unified framework to discover permutation generation algorithms. *The Computer Journal*, 66(3):603–614, 2023. `doi:10.1093/COMJNL/BXAB181`.

14 Frank Gray. Pulse code communication. *United States Patent Number 2632058*, 1953.

15 Petr Gregor, Torsten Mütze, and Namrata. Combinatorial generation via permutation languages. VI. Binary trees. *European Journal of Combinatorics*, 122:104020, 2024. `doi:10.1016/J.EJC.2024.104020`.

16 Elizabeth Hartung, Hung P Hoang, Torsten Mütze, and Aaron Williams. Combinatorial generation via permutation languages. I. Fundamentals. *Transactions of the American Mathematical Society*, 375:2255–2291, 2022.

17 Alexander E Holroyd, Frank Ruskey, and Aaron Williams. Shorthand universal cycles for permutations. *Algorithmica*, 64:215–245, 2012. `doi:10.1007/S00453-011-9544-Z`.

18 OEIS Foundation Inc. The On-Line Encyclopedia of Integer Sequences. `https://oeis.org`, 2025.

19 Selmer M Johnson. Generation of permutations by adjacent transposition. *Mathematics of computation*, 17(83):282–285, 1963.

20 Donald E Knuth. *The Art of Computer Programming, Volume 4A, Fascicle 2: Generating All Tuples and Permutations*. Addison-Wesley, 2005.

21 Donald E Knuth. *Art of Computer Programming, Volume 4, Fascicle 4, The: Generating All Trees–History of Combinatorial Generation*. Addison-Wesley Professional, 2013.

22 James Korsh and Seymour Lipschutz. Generating multiset permutations in constant time. *Journal of Algorithms*, 25(2):321–335, 1997. `doi:10.1006/JAGM.1997.0889`.

23 James F Korsh and Paul S LaFollette. Loopless array generation of multiset permutations. *The Computer Journal*, 47(5):612–621, 2004. `doi:10.1093/COMJNL/47.5.612`.

24 Donald L Kreher and Douglas R Stinson. Combinatorial algorithms: generation, enumeration, and search. *ACM SIGACT News*, 30(1):33–35, 1999. `doi:10.1145/309739.309744`.

25 Carsten Lange and Vincent Pilaud. Associahedra via spines. *Combinatorica*, 38(2):443–486, 2018. `doi:10.1007/S00493-015-3248-Y`.

26 Paul Lapey and Aaron Williams. Pop & push: Ordered tree iteration in O(1)-time. In *33rd International Symposium on Algorithms and Computation (ISAAC 2022)*, pages 53–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.ISAAC.2022.53`.

27 Zsuzsanna Lipták, Francesco Masillo, Gonzalo Navarro, and Aaron Williams. Constant time and space updates for the sigma-tau problem. In *International Symposium on String Processing and Information Retrieval*, pages 323–330. Springer, 2023. `doi:10.1007/978-3-031-43980-3_26`.

28 Joan M Lucas, D. Roelants van Baronaigien, and Frank Ruskey. On rotations and the generation of binary trees. *Journal of Algorithms*, 15(3):343–366, 1993. `doi:10.1006/JAGM.1993.1045`.

29 Percy A MacMahon. *Combinatory analysis, volumes I and II*, volume 137. American Mathematical Society, 2001.

30 Arturo Merino and Torsten Mütze. Combinatorial generation via permutation languages. III. Rectangulations. *Discrete & Computational Geometry*, 70(1):51–122, 2023. `doi:10.1007/S00454-022-00393-W`.

31 Arturo Merino, Torsten Mütze, and Aaron Williams. 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)*, pages 22–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.FUN.2022.22`.

32 Arturo Ignacio Merino Figueroa. *Combinatorial generation: greedy approaches and symmetry*. PhD thesis, Technische Universität Berlin, 2023.

33 Torsten Mütze. Proof of the middle levels conjecture. *Proceedings of the London Mathematical Society*, 112(4):677–713, 2016.

34 Torsten Mütze. Combinatorial Gray codes – An updated survey. *The Electronic Journal of Combinatorics*, page 93 pages, 2023.

35 Torsten Mütze. A book proof of the middle levels theorem. *Combinatorica*, 44(1):205–208, 2024. `doi:10.1007/S00493-023-00070-3`.

36 Arnau Padrol, Yann Palu, Vincent Pilaud, and Pierre-Guy Plamondon. Associahedra for finite-type cluster algebras and minimal relations between g-vectors. *Proceedings of the London Mathematical Society*, 127(3):513–588, 2023.

37 Vincent Pilaud, Francisco Santos, and Günter M Ziegler. Celebrating loday's associahedron. *Archiv der Mathematik*, 121(5):559–601, 2023.

38 Alexander Postnikov. Permutohedra, associahedra, and beyond. *International Mathematics Research Notices*, 2009(6):1026–1106, 2009.

39 Yuan Qiu and Aaron Williams. Generating signed permutations by twisting two-sided ribbons. In *Latin American Symposium on Theoretical Informatics*, pages 114–129. Springer, 2024. `doi:10.1007/978-3-031-55598-5_8`.

40 Nathan Reading. Lattice congruences of the weak order. *Order*, 21(4):315–344, 2004. `doi:10.1007/S11083-005-4803-8`.

41 Nathan Reading. Finite Coxeter groups and the weak order. In *Lattice Theory: Special Topics and Applications: Volume 2*, pages 489–561. Springer, 2016.

42 Frank Ruskey. Combinatorial generation. *Working version (1j-CSC 425/520)*, 2003.

43 Frank Ruskey and TC Hu. Generating binary trees lexicographically. *SIAM Journal on Computing*, 6(4):745–758, 1977. `doi:10.1137/0206055`.

44 Frank Ruskey and Andrzej Proskurowski. Generating binary trees by transpositions. *Journal of algorithms*, 11(1):68–84, 1990. `doi:10.1016/0196-6774(90)90030-I`.

45 Frank Ruskey and Aaron Williams. Generating balanced parentheses and binary trees by prefix shifts. In *CATS*, volume 8, page 140, 2008.

46 Frank Ruskey and Aaron Williams. The coolest way to generate combinations. *Discrete Mathematics*, 309(17):5305–5320, 2009. `doi:10.1016/J.DISC.2007.11.048`.

47 Joe Sawada and Aaron Williams. Successor rules for flipping pancakes and burnt pancakes. *Theoretical Computer Science*, 609:60–75, 2016. `doi:10.1016/J.TCS.2015.09.007`.

48 Joe Sawada and Aaron Williams. Solving the sigma-tau problem. *ACM Transactions on Algorithms (TALG)*, 16(1):1–17, 2019. `doi:10.1145/3359589`.

49 Robert Sedgewick. Permutation generation methods. *ACM Computing Surveys (CSUR)*, 9(2):137–164, 1977. `doi:10.1145/356689.356692`.

50 Richard P Stanley. *Catalan numbers*. Cambridge University Press, 2015.

51 Hugo Steinhaus. *One hundred problems in elementary mathematics*. Courier Corp., 1979.

52 Brett Stevens and Aaron Williams. Hamilton cycles in restricted and incomplete rotator graphs. *Journal of Graph Algorithms and Applications*, 16(4):785–810, 2012. `doi:10.7155/JGAA.00278`.

53 Dov Tamari. Monoïdes préordonnés et chaînes de malcev. *Bulletin de la Société mathématique de France*, 82:53–96, 1954.

54 Hale F Trotter. Algorithm 115: perm. *Communications of the ACM*, 5(8):434–435, 1962. `doi:10.1145/368637.368660`.

55 William T Trotter. *Combinatorics and partially ordered sets*. Johns Hopkins Press, 1992.

56 Vincent Vajnovszki and Timothy Walsh. A loop-free two-close Gray-code algorithm for listing k-ary Dyck words. *Journal of Discrete Algorithms*, 4(4):633–648, 2006. `doi:10.1016/J.JDA.2005.07.003`.

57 Aaron Williams. Loopless generation of multiset permutations using a constant number of variables by prefix shifts. In *Proceedings of the twentieth annual ACM-SIAM symposium on discrete algorithms*, pages 987–996. SIAM, 2009. `doi:10.1137/1.9781611973068.107`.

58 Aaron Williams. The greedy Gray code algorithm. In *Workshop on Algorithms and Data Structures*, pages 525–536. Springer, 2013. `doi:10.1007/978-3-642-40104-6_46`.

59 Aaron Michael Williams. *Shift Gray codes*. PhD thesis, University of Victoria, 2009.

60 Shmuel Zaks and Dana Richards. Generating trees and other combinatorial objects lexicographically. *SIAM Journal on Computing*, 8(1):73–81, 1979. `doi:10.1137/0208006`.

A Python Implementation

A full Python implementation appears below. To run it in the terminal copy the top two columns into `wiggly.py`, then use `python3` with a value of `n` as shown. (Note that `pdf` files omit leading spaces, so indents must be restored to `wiggly.py` after copy and pasting.)

```
def wiggleLeft(perm,v,inv,word,hop):
  j = inv[v]
  u = perm[j-1]
  i = j-1
  if u%2 == 1 and inv[u] >= inv[u+1]:
    i = inv[u+1]
  word[v] += j-i
  if hop:
    word[v+1] += j-i
  while j > i:
    perm[j+hop] = perm[j-1]
    inv[perm[j-1]] = j+hop
    j -= 1
  perm[i] = v
  inv[v] = i
  if hop:
    perm[i+1] = v+1
    inv[v+1] = i+1

def wiggleRight(perm,v,inv,word,hop):
  i = inv[v]
  j = i + 1 + hop
  u = perm[j]
  k = j
  if u%2 == 0 and inv[u] <= inv[u-1]:
    k = inv[u-1]
  word[v] -= k-j+1
  if hop:
    word[v+1] -= k-j+1
  while i < k-hop:
    perm[i] = perm[i+1+hop]
    inv[perm[i+1+hop]] = i
    i += 1
  perm[k-hop] = v
  inv[v] = k-hop
  if hop:
    perm[k] = v+1
    inv[v+1] = k
```

```
def wiggly(n):
  perm = list(range(n+1))
  inv = list(range(n+1))
  word = [0] * (n+1)
  fs = list(range(n+1))
  dirs = [-1]*(n+1)
  yield perm
  v = fs[n]
  while v > 1:
    i = inv[v]
    hop = v%2==1 and i<n and perm[i+1]==v+1
    if dirs[v] == -1:
      wiggleLeft(perm, v, inv, word, hop)
    else:
      wiggleRight(perm, v, inv, word, hop)
    i = inv[v]
    pair = (v%2 == 0 and perm[i-1] == v-1)
    if word[v]==0 or word[v]==v-1 or pair:
      dirs[v] *= -1
      fs[v] = fs[v-1]
      fs[v-1] = v-1
    yield perm
    v = fs[n]
    fs[n] = n

if __name__ == "__main__":
  import sys  # Handle command-line arguments
  ok = len(sys.argv) == 2
  ok = sys.argv[1].isdigit() if ok else False
  ok = int(sys.argv[1]) > 0 if ok else False
  if not ok:
    print("usage: %s n" % sys.argv[0])
    exit(0)
  n = int(sys.argv[1])
  total = 0
  for perm in wiggly(n):
    print(*perm[1:], sep="")
    total += 1
  print("total: %d" % total)
```

```
~$ python3 wiggly.py 6
123456
123465
123645
...
642135
total: 176
```