The Greedy Gray Code Algorithm

Aaron Williams

Department of Mathematics and Statistics, McGill University
haron@uvic.ca

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

1 Introduction

Let $\mathbb{B}(n)$ be the set of n-bit binary strings. The *binary reflected Gray code* $\mathbf{Gray}(n)$ orders $\mathbb{B}(n)$ so that successive strings have Hamming distance one (i.e., they differ in one bit). For example, the order for $n = 3$ appears below, with overlines denoting the change that creates the next string

$$\mathbf{Gray}(3) = 00\overline{0},\ 0\overline{0}1,\ 01\overline{1},\ \overline{0}10,\ 11\overline{0},\ 1\overline{1}1,\ 10\overline{1},\ 100. \tag{1}$$

The term *reflected* indicates how the order is created: $\mathbf{Gray}(n)$ prefixes 0 to each string of $\mathbf{Gray}(n-1)$, and then prefixes 1 to the strings of $\mathbf{Gray}(n-1)$ in reflected order. For example, the top and bottom rows below are $0 \cdot \mathbf{Gray}(3)$ and $1 \cdot \mathsf{reflect}(\mathbf{Gray}(3))$, respectively, where $\cdot$ denotes concatenation

$$\begin{aligned}\mathbf{Gray}(4) = {} & 000\overline{0}, 00\overline{0}1, 001\overline{1}, 0\overline{0}10, 011\overline{0}, 01\overline{1}1, 010\overline{1}, \overline{0}100,\\ & 110\overline{0}, 11\overline{0}1, 111\overline{1}, 1\overline{1}10, 101\overline{0}, 10\overline{1}1, 100\overline{1}, 1000.\end{aligned}$$

We can express the construction recursively as $\mathbf{Gray}(1) = 0, 1$ and for $n > 1$,

$$\mathbf{Gray}(n) = 0 \cdot \mathbf{Gray}(n-1),\ 1 \cdot \mathsf{reflect}(\mathbf{Gray}(n-1)), \tag{2}$$

where the comma appends the two lists. The above definition uses *global recursion* since it refers to the entire $\mathbf{Gray}(n-1)$ list as one unit. We can instead define the order using *local recursion* by referring to the individual strings in $\mathbf{Gray}(n-1)$. If $\mathbf{Gray}(n-1) = \mathbf{b_1}, \mathbf{b_2}, \ldots, \mathbf{b_{k-1}}, \mathbf{b_k}$ for $k = 2^{n-1}$, then

$$\mathbf{Gray}(n) = \mathbf{b_1} \cdot 0, \mathbf{b_1} \cdot 1, \mathbf{b_2} \cdot 1, \mathbf{b_2} \cdot 0, ..., \mathbf{b_{k-1}} \cdot 0, \mathbf{b_{k-1}} \cdot 1, \mathbf{b_k} \cdot 1, \mathbf{b_k} \cdot 0 \tag{3}$$

where $\mathbf{Gray}(1) = 0, 1$. In other words, $\mathbf{Gray}(n)$ can be created by alternately suffixing 0 then 1, and 1 then 0, to successive strings in $\mathbf{Gray}(n-1)$.

Since Frank Gray was granted U.S. Patent 2,632,058 in 1953 [4], his order has been used in numerous applications, with rotary encoders providing a prominent example [12]. The term *Gray code* now refers to minimal change orders of

F. Dehne, R. Solis-Oba, and J.-R. Sack (Eds.): WADS 2013, LNCS 8037, pp. 525–536, 2013.

combinatorial objects. Gray codes are related to efficient algorithms for exhaustively generating combinatorial objects. Knuth recently surveyed combinatorial generation in *The Art of Computer Programming* [6], and included separate subsections on generating tuples, permutations, combinations, partitions, and trees. Although the research area is quite diverse, it is fair to say that it has been dominated by the ideas of recursion and reflection. To demonstrate, we next recount a classic Gray code for combinations and a classic Gray code for permutations.

In the 1980s, Eades and McKay [3] followed Gray's approach to order the k-combinations of an n element set, which can be represented by $\mathbb{B}(n,k)$ the n-bit binary strings with fixed *weight* (i.e., number of 1s) equal to k. The Eades-McKay Gray code is defined using recursion and reflection as follows

$$\mathbf{EM}(n,k) = \mathbf{EM}(n-1,k)\cdot 0,\ \mathsf{reflect}(\mathbf{EM}(n-2,k-1))\cdot 01,\ \mathbf{EM}(n-2,k-2)\cdot 11$$

with $\mathbf{EM}(n,0) = 0^n$, $\mathbf{EM}(n,n) = 1^n$, and $\mathbf{EM}(n,1) = 10^{n-1}, 010^{n-2}, \ldots, 0^{n-1}1$, where exponentiation denotes repetition. For example,

$$\mathbf{EM}(5,3) = \underbrace{11\overline{10}0,1\overline{10}10,\overline{10}110,011\overline{10}}_{\mathbf{EM}(4,3)\cdot 0},\ \underbrace{\overline{01}101,1\overline{01}01,1\overline{1}0\overline{0}1}_{\mathsf{reflect}(\mathbf{EM}(3,2))\cdot 01},\ \underbrace{\overline{10}011,0\overline{10}11,00111}_{\mathbf{EM}(3,1)\cdot 11.} \tag{4}$$

In this order, successive strings differ by a *homogeneous transposition*, meaning that a 1 and 0 can only be interchanged if the intermediate symbols are all 0s. In other words, substrings of the form $00\cdots01$ and $100\cdots0$ can be interchanged. Thus, a single s_i changes when the elements of the combination are represented as $1 \le s_1 < s_2 < \cdots < s_k \le n$. For this reason, the order allows pianists to practice all k-note chords while moving only a single finger between chords [3].

Let $\mathbb{P}(n)$ be the permutations of $[n] = \{1,2,\ldots,n\}$ in one-line notation. For example, $\mathbb{P}(3) = \{123, 213, 213, 231, 312, 321\}$. In the 1960s, researchers considered permutation Gray codes using *adjacent transpositions* (or *swaps*), meaning that two symbols can only be interchanged if they are next to each other. Johnson, Trotter, and Steinhaus all approached the problem using local recursion, and they all rediscovered an order known in the 17th century as *plain changes* [2]. To explain the order, let $\mathsf{zig}(\mathbf{p})$ be the list obtained from $\mathbf{p}$ by applying the following swaps: $(n\ n{-}1)$, $(n{-}1\ n{-}2)$, ..., $(1\ 2)$. For example, $\mathsf{zig}(1234) = 1\overleftarrow{234}$, $1\overleftarrow{243}$, $\overleftarrow{1423}$, 4123 where the arrow shows the movement of 4. Similarly, let $\mathsf{zag}(\mathbf{p})$ apply the following swaps: $(1\ 2)$, $(2\ 3)$, ..., $(n\ n{-}1)$. Notice that zigs and zags only change the relative order of the last and first symbols, respectively. Thus, we can define a Gray code as follows: If $\mathbf{Plain}(n-1) = \mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_{(n-1)!}}$, then

$$\mathbf{Plain}(n) = \mathsf{zig}(\mathbf{p_1}\cdot n),\ \mathsf{zag}(n\cdot \mathbf{p_2}), ..., \mathsf{zig}(\mathbf{p_{(n-1)!-1}}\cdot n),\ \mathsf{zag}(n\cdot \mathbf{p_{(n-1)!}}) \tag{5}$$

where $\mathbf{Plain}(2) = 12, 21$. For example, the following order is $\mathsf{zig}(12)\cdot 3, \mathsf{zag}(21)\cdot 3$

$$\mathbf{Plain}(3) = 123, 132, 312, 321, 231, 231. \tag{6}$$

In this article we propose an alternate method for understanding the aforementioned Gray codes and many others. To illustrate the idea, consider the following

method for building a list $\mathcal{L}$ of unique strings in $\mathbb{B}(n)$: Initialize $\mathcal{L}$ to contain 0^n, then repeatedly extend $\mathcal{L}$ by complementing the rightmost possible bit in its last string. For example, if the current list is $\mathcal{L} = 000, 001, 011, 010$ then we examine its last string 010. The rightmost bit cannot be complemented since $01\overline{0} = 011$ is already in $\mathcal{L}$. Similarly, the middle bit cannot be complemented since $0\overline{1}0 = 000 \in \mathcal{L}$. However, the leftmost bit can be complemented since $\overline{0}10 = 110 \notin \mathcal{L}$. Thus, 110 is added to the end of $\mathcal{L}$. The complete list for $n = 3$ is in (1). More generally, we prove that the method always creates $\mathbf{Gray}(n)$.

As a second example, initialize $\mathcal{L}$ to contain 1^k0^{n-k}, then repeatedly extend $\mathcal{L}$ by homogeneously transposing the leftmost possible 1 into the leftmost possible position. For example, if $\mathcal{L} = 11100, 11010, 10110, 01110$ then we examine 01110. The leftmost 1 could be homogeneously transposed into the first position, however $\overline{01}110 = 10110 \in \mathcal{L}$. The middle 1 cannot be homogeneously transposed since it is bordered by 1s. The rightmost 1 can be homogeneously transposed into the last position and $011\overline{10} = 01101 \notin \mathcal{L}$. Thus, 01101 is added to the end of $\mathcal{L}$. The complete list for $n = 5$ and $k = 3$ is in (4). More generally, we prove that the method always creates $\mathbb{B}(n, k)$, and for odd k the order is $\mathbf{EM}(n, k)$.

As a third example, initialize $\mathcal{L}$ to contain $12\cdots n$, then extend $\mathcal{L}$ by swapping the largest possible symbol once to the left or right. For example, if $\mathcal{L} = 123, 132, 312$ then we examine 312. The 3 cannot swap left since it is in the leftmost position. Similarly, 3 cannot swap right since $3\overleftrightarrow{12} = 132 \in \mathcal{L}$. However, 2 can swap left since $3\overleftrightarrow{12} = 321 \notin \mathcal{L}$. Thus, 321 is added to $\mathcal{L}$. The complete list for $n = 3$ is in (6). More generally, the method always creates $\mathbf{Plain}(n)$.

Our "greedy Gray code algorithm" is defined in Section 3 and reinterprets many classic Gray codes. Section 4 discusses these results on binary strings:

1. The binary reflected Gray code complements the rightmost possible bit;
2. Lexicographic order complements the shortest possible suffix;
3. The de Bruijn sequence by Martin [8] shifts in the lowest possible bit.

Section 5 discusses these results for permutations:

4. The plain change order adjacently transposes the largest possible symbol;
5. The pancake flipping order by Zaks [13] reverses the shortest possible prefix;
6. Corbett's rotator graph order [1] rotates the prefix with the first possible length in $n, 2, n-1, 3, \ldots$.

Section 6 discusses the following additional results:

7. The Eades-McKay order of combinations homogeneously transposes the leftmost possible 1 into the leftmost possible position when the weight is odd.
8. The Lucas, van Baronaigien and Ruskey order of binary trees [7] rotates the edge with the largest inorder label.
9. Kaye's set partition order [5] moves the largest possible symbol into the leftmost possible subset.

In addition, Section 2 provides an application for our greedy reinterpretations. We conclude the introduction with several clarifications.

- This greedy method is not entirely new. For example, the de Bruijn sequence we discuss here was first defined greedily by Martin in 1934 [8]. However, the

term 'greedy' is not common in the literature, nor is it featured in Knuth's 400 page treatise on combinatorial generation [6].
- The greedy method is not suitable for efficiently generating Gray codes since it may have to 'remember' an exponential number of objects relative to their size. However, it can provide a simpler description for previously created Gray codes and a simpler method for discovering new Gray codes (see [9]).
- Recursive constructions are often general results. For example, any swap Gray code of $\mathbb{P}(n-1)$ provides a swap Gray code of $\mathbb{P}(n)$ using (5). In contrast, our greedy method gives only one order. However, this order may illuminate a general recursive principle that leads to an efficient generation algorithm.

In general, the author views the greedy Gray code algorithm as a simple and unified "first step" in understanding and discovering Gray codes.

2 Network Application

Gray codes give Hamilton paths and cycles in well-studied graphs, such as the n-cube (binary reflected Gray code), the permutohedron (plain changes), the rotator graph (Corbett's order), and the pancake graph (Zaks's order). These graphs are used as network topologies, where vertices are computers and two vertices can communicate if they are adjacent (see Siegel [10]). In this section, we illustrate how our greedy algorithms can send messages through these networks.

The *pancake graph* has vertices $\mathbb{P}(n)$ and edges between pairs of vertices that differ by a prefix-reversal. Figure 1 a) illustrates the graph for $n = 4$. In Section 5 we will see that a Hamilton path can be created in this graph from 1234 by repeatedly reversing the shortest possible prefix that gives a new permutation. The order of vertices visited on this Hamilton path is illustrated by Figure 1 b).

Suppose each vertex sets a flag if it has seen a particular message, and each vertex can query the flags of its neighbors. Also assume that the neighbors of a vertex are 'prioritized' by increasing prefix-reversal lengths. Given this scenario, we claim that a message m will propagate from an initial vertex to all other vertices in the pancake graph so long as each vertex runs the following algorithm:

> When a vertex receives m, it sets its flag and passes m to its highest-priority neighbor whose flag is not set.

For example, consider vertex 3214 in Figure 1 b), which is the sixth vertex to receive the message. Once it receives the message, it cannot pass it to its highest-priority neighbor $\overleftrightarrow{23}14$ since this vertex has already seen the message, and similarly it cannot pass it to $\overleftrightarrow{123}4$. However, it can pass the message to its lowest-priority neighbor $\overleftrightarrow{4123}$, and at this point its algorithm terminates.

To clarify an important point, we mention that the pancake graph is vertex-transitive, and that our greedy prefix-reversal algorithm generates $\mathbb{P}(n)$ for any initial permutation. Thus, our approach works regardless of where the message originates. Furthermore, the same arguments apply for our greedy algorithms in the n-cube, permutohedron, and rotator graph. In particular, this approach in the rotator graph is much more efficient than the table approach in Corbett [1].

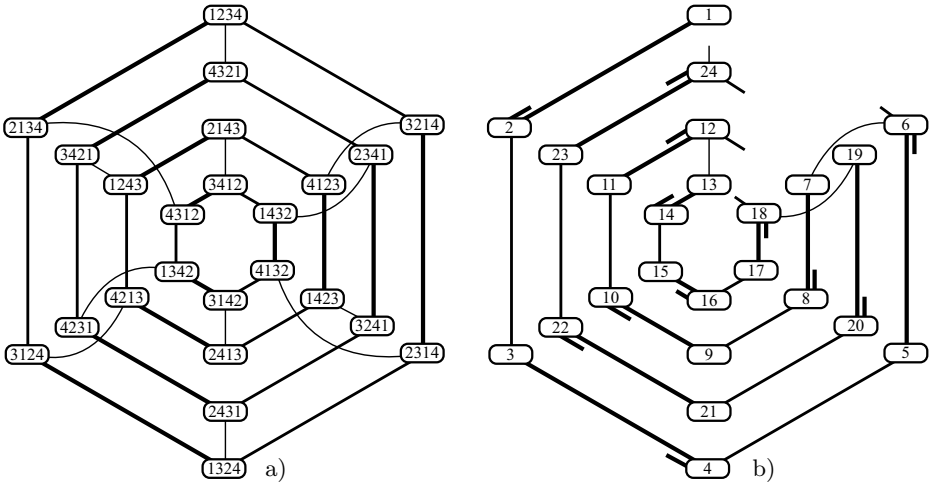

Fig. 1. a) The pancake network for $n = 4$ in which thick, medium, and thin edges are used for prefix-reversals of length two, three, and four, respectively. b) The Hamilton path obtained by greedily reversing the shortest possible prefix starting from 1234, where each partial edge shows a prefix-reversal leading to a previously visited vertex.

3 Greedy Gray Code Algorithm

The *greedy Gray code algorithm* takes as input an object $\mathbf{x} \in \mathbf{X}$ 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 $1 \leq i \leq k$. The algorithm outputs a *greedy object list* $\mathcal{L}$ of distinct objects in $\mathbf{X}$. The list initially contains $\mathbf{x}$, and then is repeatedly extended by one object as follows: If $\mathbf{x}$ is the last object in $\mathcal{L}$, and i is the minimum value such that $o_i(\mathbf{x})$ is not already in $\mathcal{L}$, then $o_i(\mathbf{x})$ is added to the end of $\mathcal{L}$. $\mathsf{Greedy}_{\mathcal{O}}(\mathbf{x})$ is *successful* if it generates $\mathbf{X}$. In other words, success occurs if every object of the same type as $\mathbf{x}$ is in $\mathcal{L}$.

$\mathsf{Greedy}_{\mathcal{O}}(\mathbf{x})$
1: Initialize list $\mathcal{L}$ to contain the single object $\mathbf{x}$.
2: Let $\mathbf{x}$ be the last object in list $\mathcal{L}$.
3: Let i be minimum such that $o_i(\mathbf{x})$ is not in $\mathcal{L}$. If i does not exist, then return.
4: Add the new object $o_i(\mathbf{x})$ to the end of $\mathcal{L}$.
5: Return to line 2.

Given a prioritized list of operations $\mathcal{O} = o_1, o_2, \ldots$ and an index list $\mathcal{I} = i_1, i_2, \ldots$, we generate a list of objects as follows. Let $\mathsf{Apply}_{\mathcal{O}}(\mathbf{x_1}, \mathcal{I})$ be the list $\mathbf{x_1}, \mathbf{x_2}, \ldots$ where $\mathbf{x_{i+1}} = o_{i_k}(\mathbf{x_i})$ for $k = 1, 2, \ldots$. That is, the i_kth operation creates the $(k+1)$st object from the kth object.

4 Binary Strings

In this section, we give greedy interpretations to three orders of binary strings. Throughout this section we index the bits of a binary string from right-to-left. Thus, if $\mathbf{b} \in \mathbb{B}(n)$, then $\mathbf{b} = b_n b_{n-1} \cdots b_1$ are its individual bits. The *ith bit* of $\mathbf{b}$

uses this right-to-left indexing, so the first bit is the rightmost. A draft of this paper illustrates each order in Table 1 (see the author's website).

4.1 Binary Reflected Gray Code

We first prove the greedy interpretation of the binary reflected Gray code using local recursion. Let bit_i be the operation that complements the ith bit of a binary string. That is, $\mathsf{bit}_i(\mathbf{b}) = b_n \cdots b_{i+1}\overline{b_i}b_{i-1} \cdots b_1$. We prioritize the bit complements from right-to-left in $\mathsf{Bit}_n^{\uparrow} = \mathsf{bit}_1, \mathsf{bit}_2, \ldots, \mathsf{bit}_n$. (In general, we use lowercase for individual operations and uppercase for prioritized lists of operations, with $^{\uparrow}$ and $^{\downarrow}$ for lists with increasing and decreasing subscripts, respectively.)

Theorem 1. *The greedy Gray code algorithm that complements the rightmost possible bit generates the reflected Gray code. That is,* $\mathsf{Greedy}_{\mathsf{Bit}_n^{\uparrow}}(0^n) = \mathbf{Gray}(n)$.

Proof. The proof is by induction on n with $\mathsf{Greedy}_{\mathsf{Bit}_1^{\uparrow}}(0^1) = 0, 1 = \mathbf{Gray}(1)$ for the base case. Inductively assume that

$$\mathsf{Greedy}_{\mathsf{Bit}_{m-1}^{\uparrow}}(0^{m-1}) = \mathbf{b_1}, \mathbf{b_2}, \ldots, \mathbf{b_{2^{m-1}}} = \mathbf{Gray}(m{-}1).$$

In particular, $\mathbf{b_1} = 0^{m-1}$ and $\mathbf{b_2} = 0^{m-2}1$. The first four strings generated by $\mathsf{Greedy}_{\mathsf{Bit}_m^{\uparrow}}(0^m)$ are $0^m, 0^{m-1}1, 0^{m-2}11, 0^{m-2}10 = \mathbf{b_1} \cdot 0, \mathbf{b_1} \cdot 1, \mathbf{b_2} \cdot 1, \mathbf{b_2} \cdot 0$. More generally, suppose $\mathsf{Greedy}_{\mathsf{Bit}_m^{\uparrow}}(0^m)$ begins

$$\mathbf{b_1} \cdot 0, \mathbf{b_1} \cdot 1, \mathbf{b_2} \cdot 1, \mathbf{b_2} \cdot 0, \ldots, \mathbf{b_{2i-1}} \cdot 0, \mathbf{b_{2i-1}} \cdot 1, \mathbf{b_{2i}} \cdot 1, \mathbf{b_{2i}} \cdot 0 \tag{7}$$

for some fixed $1 \leq i < 2^{m-1}$. The algorithm cannot apply bit_m to the last string in (7) since $\mathbf{b_{2i}} \cdot \overline{0} = \mathbf{b_{2i}} \cdot 1$ is the second-last string in (7). Therefore, the algorithm can only apply bit_j for some $j < m$. Thus, the next string (if any) generated by the algorithm will end with 0. Observe that the strings ending with 0 in (7) are precisely $\mathbf{b_1} \cdot 0, \mathbf{b_2} \cdot 0, \ldots, \mathbf{b_{2i}} \cdot 0$. Since $\mathsf{Greedy}_{\mathsf{Bit}_{m-1}^{\uparrow}}(0^{m-1})$ begins by generating $\mathbf{b_1}, \mathbf{b_2}, \ldots, \mathbf{b_{2i}}, \mathbf{b_{2i+1}}$, we know $\mathsf{Greedy}_{\mathsf{Bit}_m^{\uparrow}}(0^m)$ behaves accordingly. Thus, $\mathsf{Greedy}_{\mathsf{Bit}_m^{\uparrow}}(0^m)$ follows $\mathbf{b_{2i}} \cdot 0$ by generating $\mathbf{b_{2i+1}} \cdot 0$. Furthermore, the string generated after $\mathbf{b_{2i+1}} \cdot 0$ is $\mathbf{b_{2i+1}} \cdot 1$ since bit_m is the highest priority operation. Therefore, (7) is true when $i{+}1$ replaces i. Hence, by repeating this argument (7) is true for $i = 2^{m-1}$. Therefore, $\mathsf{Greedy}_{\mathsf{Bit}_m^{\uparrow}}(0^m)$ and $\mathbf{Gray}(m)$ share the same recursive structure by (3) and (7), which completes the induction. □

4.2 Lexicographic Order

We next give a greedy interpretation to $\mathbf{Lex}(n)$, the *lexicographic order of* $\mathbb{B}(n)$ in which successive strings have decimal value $0, 1, 2, \ldots, 2^n{-}1$. For example,

$$\mathbf{Lex}(3) = 00\overline{0}, 0\overline{01}, 01\overline{0}, \overline{011}, 10\overline{0}, 1\overline{01}, 11\overline{0}, 111.$$

Notice that each successive string is obtained by a *suffix complement* suff_i which complements the rightmost i bits. That is, $\mathsf{suff}_i(\mathbf{b}) = b_n b_{n-1} \cdots b_{i+1}\overline{b_i b_{i-1} \cdots b_1}$

for $\mathbf{b} \in \mathbb{B}(n)$. We prioritize by shortest suffix in $\mathsf{Suff}_n^{\uparrow} = \mathsf{suff}_1, \mathsf{suff}_2, \ldots, \mathsf{suff}_n$. Lexicographic order has the same global recursive definition as the binary reflected Gray code, without the reflection. That is, $\mathbf{Lex}(1) = 0, 1$ and for $n > 1$,

$$\mathbf{Lex}(n) = 0 \cdot \mathbf{Lex}(n{-}1),\ 1 \cdot \mathbf{Lex}(n{-}1). \tag{8}$$

For example, the order below is $0 \cdot \mathbf{Lex}(3)$ followed by $1 \cdot \mathbf{Lex}(3)$

$$\begin{aligned}\mathbf{Lex}(4) = {} & 000\overline{0}, 00\overline{01}, 001\overline{0}, 0\overline{011}, 010\overline{0}, 01\overline{01}, 011\overline{0}, \overline{0111},\\ & 100\overline{0}, 10\overline{01}, 101\overline{0}, 1\overline{011}, 110\overline{0}, 11\overline{01}, 111\overline{0}, 1111.\end{aligned}$$

Theorem 2. *The greedy Gray code algorithm that complements the shortest possible suffix generates lexicographic order. That is,* $\mathsf{Greedy}_{\mathsf{Suff}_n^{\uparrow}}(0^n) = \mathbf{Lex}(n)$.

Proof. Our proof is by induction, with $\mathsf{Greedy}_{\mathsf{Suff}_1^{\uparrow}}(0^1) = \overline{0}, 1 = \mathbf{Lex}(1)$ as the base case. Inductively assume $\mathsf{Greedy}_{\mathsf{Suff}_{m-1}^{\uparrow}}(0^{m-1}) = \mathbf{Lex}(m{-}1)$. Now consider $\mathsf{Greedy}_{\mathsf{Suff}_m^{\uparrow}}(0^m)$. Since suff_m is the lowest-priority operation, the algorithm will begin by creating as many strings as possible using $\mathsf{Suff}_{m-1}^{\uparrow}$. By induction, this produces $0 \cdot \mathbf{Lex}(m{-}1)$, whose last string is $0 \cdot 1^{m-1}$. The greedy algorithm must then apply suff_m to this string to create $\overline{0 \cdot 1^{m-1}} = 1 \cdot 0^{m-1}$ since every string beginning with 0 has already been generated. Now the algorithm again proceeds by creating as many strings as possible using $\mathsf{Suff}_{m-1}^{\uparrow}$ starting from $1 \cdot 0^{m-1}$. By induction, this produces $1 \cdot \mathbf{Lex}(m{-}1)$. Thus, $\mathsf{Greedy}_{\mathsf{Suff}_m^{\uparrow}}(0^m) = 0 \cdot \mathbf{Lex}(m{-}1),\ 1 \cdot \mathbf{Lex}(m{-}1)$, and so our result is true by (8). □

4.3 de Bruijn Sequences

A *de Bruijn sequence* is binary string of length 2^n that contains every string in $\mathbb{B}(n)$ exactly once as a circular substring of length n. Martin [8] showed that a de Bruijn sequence $\mathbf{dB}(n)$ can be built one bit at a time by starting from 0^n and greedily suffixing the largest possible next bit 1 or 0, subject to the condition that the resulting sequence does not contain any substring twice[1]. For example, if the algorithm for $n = 4$ has currently built 00001111011, then Martin's algorithm will not append 1 since the resulting sequence of bits 000<u>01111</u><u>0111</u> would contain two copies of 0111. Thus, it would append 0. The result of Martin's algorithm for $n = 4$ is $\mathbf{dB}(4) = 0000111101100101$. A de Bruijn sequence is *decoded* by listing its successive substrings of length n. For example,

$$\begin{aligned}\mathsf{decode}(\mathbf{dB}(4)) &= \mathsf{decode}(0000111101100101)\\ &= 0000, 0001, 0011, 0111, 1111, 1110, 1101, 1011,\\ &\quad 0110, 1100, 1001, 0010, 0101, 1010, 0100, 1000\end{aligned}$$

where the final three substrings "wrap around". Successive decoded substrings always differ by a 1-*shift* or a 0-*shift*, meaning that $b_n b_{n-1} \cdots b_1$ is replaced by

[1] Martin constructs a sequence of length $2^n{+}n{-}1$ starting from $0^{n-1}1$ whose 2^n non-circular substrings are $\mathbb{B}(n)$. This sequence ends with 0^n, so it is equivalent to $\mathbf{dB}(n)$.

$b_{n-1}b_{n-2}\cdots b_1 1$ or $b_{n-1}b_{n-2}\cdots b_1 0$, respectively. We denote these two operations by shift_1 and shift_0, respectively, and prioritize them as $\mathsf{Shift}_2^{\downarrow} = \mathsf{shift}_1, \mathsf{shift}_0$. This allows us to reinterpret Martin's result using the greedy Gray code algorithm.

Theorem 3 ([8]). *The greedy Gray code algorithm that shifts in the largest possible bit generates decoded strings in Martin's de Bruijn sequence. That is,* $\mathsf{Greedy}_{\mathsf{Shift}_2^{\downarrow}}(0^n) = \mathsf{decode}(\mathbf{dB}(n))$.

We mention that Theorem 2 depends on the initial string. For example, 000 and 001 are the only suitable choices for generating $\mathbb{B}(3)$ in this way.

5 Permutations

In this section, we give greedy interpretations to three permutation orders. Throughout this section we index the symbols of a permutation from left-to-right. Thus, if $\mathbf{p} \in \mathbb{P}(n)$, then $\mathbf{p} = p_1 p_2 \cdots p_n$. A draft of this paper illustrates each order in Table 2 (see the author's website).

5.1 Plain Change Order

The *transposition* $(i\ j)$ interchanges the values in positions i and j of a string. A *swap* is a transposition of the form $(i\ i{+}1)$. Swaps are also known as *adjacent transpositions*. When considering permutations, we can indicate a specific swap by indicating a value and a direction, instead of a pair of positions. Let swap_{-v} and swap_{+v} be the operations that swap value v one position to the left, or right, respectively. For example, $\mathsf{swap}_{-2}(7654321) = 765\overleftarrow{432}1 = 7654231$ is a *left swap* of 2, and $\mathsf{swap}_{+2}(7654321) = 7654\overrightarrow{321} = 7654312$ is a *right swap* of 2. If $\mathbf{p} = p_1 p_2 \cdots p_n \in \mathbb{P}(n)$, then $\mathsf{swap}_{+p_n}(\mathbf{p}) = \mathsf{swap}_{-p_1}(\mathbf{p}) = \mathbf{p}$. In other words, left swapping the first value does not change a permutation, nor does right swapping the last value. We prioritize our swaps by decreasing values, and right before left,

$$\mathsf{Swap}_n^{\downarrow} = \mathsf{swap}_{+n}, \mathsf{swap}_{-n}, \ldots, \mathsf{swap}_{+2}, \mathsf{swap}_{-2}, \mathsf{swap}_{+1}, \mathsf{swap}_{-1}.$$

Note: The relative priorities of swap_{+i} and swap_{-i} do not affect the proof of Theorem 4, so we say that the swaps are prioritized by decreasing value.

Theorem 4. *The greedy Gray code algorithm that swaps the largest possible value generates the plain change order. That is,* $\mathsf{Greedy}_{\mathsf{Swap}_n^{\downarrow}}(12\cdots n) = \mathbf{Plain}(n)$.

Proof. The proof is by induction on n with $\mathsf{Greedy}_{\mathsf{Swap}_2^{\downarrow}}(12) = \overleftarrow{12}, 21 = \mathbf{Plain}(2)$ as the base case. Inductively assume

$$\mathsf{Greedy}_{\mathsf{Swap}_{m-1}^{\downarrow}}(12\cdots m{-}1) = \mathbf{p_1}, \ldots, \mathbf{p_{(m-1)!}} = \mathbf{Plain}(m{-}1).$$

In particular, $\mathbf{p_1} = 12\cdots m{-}1$ and $\mathbf{p_2} = 12\cdots m{-}3\, m{-}1\, m{-}2$. The first m strings generated by $\mathsf{Greedy}_{\mathsf{Swap}_m^{\downarrow}}(12\cdots m)$ are

$$1\,2\cdots m{-}2\,\overleftarrow{m{-}1\,m},\ 1\,2\cdots\overleftarrow{m{-}2\,m}\,m{-}1,\ \ldots,\ m\,1\,2\cdots m{-}2\,m{-}1\ = \mathsf{zig}(\mathbf{p_1}\cdot m)$$

by repeatedly applying swap_{-m}. At this point, the algorithm cannot swap m so it swaps $m{-}1$ to create $m\,1\,2\cdots\overleftarrow{m{-}2\,m{-}1} = m\,1\,2\cdots m{-}3\,m{-}1\,m{-}2$. This is followed by repeatedly applying swap_{+m} as below

$$\overrightarrow{m}\,1\,2\cdots m{-}3\,m{-}1\,m{-}2, 1\,\overrightarrow{m}\,2\cdots m{-}3\,m{-}1\,m{-}2, ..., 1\,2\cdots m{-}3\,m{-}1\,m{-}2\,m = \mathsf{zag}(m{\cdot}\mathbf{p_2}).$$

More generally, suppose $\mathsf{Greedy}_{\mathsf{Swap}_m^{\downarrow}}(12\cdots m)$ begins

$$\mathsf{zig}(\mathbf{p_1}\cdot m), \mathsf{zag}(m\cdot\mathbf{p_2}), \mathsf{zig}(\mathbf{p_3}\cdot m), \mathsf{zag}(m\cdot\mathbf{p_4}), ..., \mathsf{zig}(\mathbf{p_{2i-1}}\cdot m), \mathsf{zag}(m\cdot\mathbf{p_{2i}}) \quad (9)$$

for some fixed $1 \le i < (m-1)!$. Notice that the last string in (9) is $\mathbf{p_{2i}}\cdot m$. The algorithm cannot apply swap_{+m} to $\mathbf{p_{2i}}\cdot m$ since m is in the rightmost position. Similarly, the algorithm cannot apply swap_{+m} since that would result in the second-last string in (9). Thus, the next string (if any) generated by the algorithm will begin with m. Observe that the strings beginning with m in (9) are precisely $m\cdot\mathbf{p_1}, m\cdot\mathbf{p_2}, \ldots, m\cdot\mathbf{p_{2i}}$. Since $\mathsf{Greedy}_{\mathsf{Swap}_{m-1}^{\downarrow}}(12\cdots m-1)$ begins by generating $\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_{2i}}, \mathbf{p_{2i+1}}$, we know $\mathsf{Greedy}_{\mathsf{Swap}_m^{\downarrow}}(12\cdots m)$ behaves accordingly. Thus, $\mathsf{Greedy}_{\mathsf{Swap}_m^{\downarrow}}(12\cdots m)$ follows $m\cdot\mathbf{p_{2i}}$ with $m\cdot\mathbf{p_{2i+1}}$. At this point, $m\cdot\mathbf{p_{2i+1}}$ is the first generated string in which the symbols of $[m{-}1]$ are in the relative order given by $\mathbf{p_{2i+1}}$. Thus, the algorithm continues by generating $\mathsf{zig}(m\cdot\mathbf{p_{2i+1}})$ since swap_{+m} is the highest priority operation. This ends with $\mathbf{p_{2i+1}}\cdot m$, and for similar reasons, the algorithm follows this by $\mathsf{zag}(\mathbf{p_{2i+1}}\cdot\mathbf{m})$. Therefore, (9) is true when $i{+}1$ replaces i. Hence, (9) is true for $i = (m-1)!$. Therefore, $\mathsf{Greedy}_{\mathsf{Swap}_m^{\downarrow}}(12\cdots m)$ and $\mathbf{Plain}(m)$ share the same recursive structure by (5) and (9), which completes the induction. □

5.2 Zaks's Pancake Order

Let rev_i be the operation that reverses the first i symbols of a string. Thus, if $\mathbf{p} = p_1p_2\cdots p_n$, then $\mathsf{rev}_i(\mathbf{p}) = p_ip_{i-1}\cdots p_1p_{i+1}p_{i+2}\cdots p_n$. This operation is known as a *prefix-reversal* or a *flip*. The term 'flip' comes from the *pancake problem*: Given a stack of n pancakes of distinct sizes, what is the minimum number of times a waiter must flip over some number of pancakes at the top of the stack in order to sort the pancakes from smallest to largest?

Zaks [13] considered the problem of creating all possible stacks (or permutations) using flips. As he writes, "The poor waiter will be able to generate, in n! such steps, all possible n! stacks". Zaks used global recursion to create his order. For example, $\mathbf{Pan}(3) = \overleftarrow{12}3, \overleftarrow{21}3, \overleftarrow{31}2, \overleftarrow{132}, \overleftarrow{23}1, 321$ and $\mathbf{Pan}(4)$ repeats this four times below, with prefix-reversals of length three in between

$$\overleftarrow{12}34,\ \overleftarrow{21}34,\ \overleftarrow{31}24,\ \overleftarrow{132}4,\ \overleftarrow{23}14,\ \overleftarrow{3214},\ \overleftarrow{41}23,\ \overleftarrow{142}3,\ \overleftarrow{24}13,\ \overleftarrow{4213},\ \overleftarrow{12}43,\ \overleftarrow{2143},$$
$$\overleftarrow{34}12,\ \overleftarrow{4312},\ \overleftarrow{13}42,\ \overleftarrow{3142},\ \overleftarrow{41}32,\ \overleftarrow{1432},\ \overleftarrow{23}41,\ \overleftarrow{3241},\ \overleftarrow{42}31,\ \overleftarrow{2431},\ \overleftarrow{34}21,\ 4321.$$

Theorem 5. *The greedy Gray code algorithm that reverses the shortest possible prefix generates Zaks's order. That is,* $\mathsf{Greedy}_{\mathsf{Rev}_n^{\uparrow}}(12\cdots n) = \mathbf{Pan}(n)$.

A new pancake order $\mathbf{Pan}'(n)$ is generated by greedily reversing the longest possible prefix, as prioritized by $\mathsf{Rev}_n^{\downarrow}$. The reader can refer to the recent article by the author and Sawada [9] for these results.

Theorem 6 ([9]). *The greedy Gray code algorithm that reverses the longest possible prefix generates all permutations. That is,* $\mathsf{Greedy}_{\mathsf{Rev}_n^{\uparrow}}(12\cdots n) = \mathbf{Pan}'(n)$.

5.3 Corbett's Rotator Order

It is easy to show that $\mathbb{P}(n)$ is not generated by greedily rotating the shortest possible prefix, or the longest possible prefix, for $n \geq 4$. However, we will see that $\mathbb{P}(n)$ can be generated by prioritizing the rotations in a different way. In fact, the Gray code will equal an order given by Corbett in the context of the interconnection network known as the *rotator graph* (see Corbett [1]).

Corbett's order $\mathbf{Rotator}(n)$ is generated with the help of an index sequence $\mathbf{Rotator}''(n)$. The index sequence is defined as follows: $\mathbf{Rotator}''(2) = 2$ and if $\mathbf{Rotator}''(n-1) = r_1, r_2, \ldots, r_{(n-1)!-1}$ then $\mathbf{Rotator}''(n)$ appears below

$$n, ..., n, n+1-r_1,\ n, ..., n, n+1-r_2,\ \ldots,\ n, ..., n, n+1-r_{(n-1)!-1},\ n, ..., n.$$

where each $n, ..., n$ denotes n repeated $n-1$ times. For example, $\mathbf{Rotator}''(3) = 3, 3, 2, 3, 3$ and so $\mathbf{Rotator}''(4) =$ 4,4,4,2,4,4,4,2,4,4,4,3,4,4,4,2,4,4,4,2,4,4,4. Corbett's order is obtained by applying the sequence as rotations starting from $n\,n{-}1\cdots 1 \in \mathbb{P}(n)$. That is, $\mathbf{Rotator}(n) = \mathsf{Apply}_{\mathbf{Rotator}''(n)}(n\,n{-}1\cdots 1, \mathsf{Rot}_n^{\uparrow})$, where $\mathsf{Rot}_n^{\uparrow} = \mathsf{rot}_1, \mathsf{rot}_2, \ldots, \mathsf{rot}_n$ and rot_1 is included for convenience. For example, the orders for $n = 3$ and $n = 4$ appear below.

$\mathbf{Rotator}(3)$	$\mathbf{Rotator}(4)$
$\overrightarrow{321}, \overrightarrow{213}, \overrightarrow{132},$	$\overrightarrow{4321}, \overrightarrow{3214}, \overrightarrow{2143}, \overrightarrow{1432}, \overrightarrow{4132}, \overrightarrow{1324}, \overrightarrow{3241}, \overrightarrow{2413}, \overrightarrow{4213}, \overrightarrow{2134}, \overrightarrow{1342}, \overrightarrow{3421},$
$\overrightarrow{312}, \overrightarrow{123}, 231$	$\overrightarrow{4231}, \overrightarrow{2314}, \overrightarrow{3142}, \overrightarrow{1423}, \overrightarrow{4123}, \overrightarrow{1234}, \overrightarrow{2341}, \overrightarrow{3412}, \overrightarrow{4312}, \overrightarrow{3124}, \overrightarrow{1243}, 2431.$

Understanding the correctness of Corbett's construction is somewhat tricky, and we refer the reader to [1] and Stevens and Williams [11]. On the other hand, it has a relatively simple greedy interpretation. We prioritize the prefix rotations by *interleaving* the longest and shortest as follows

$$\mathsf{Rot}_n^{\updownarrow} = \mathsf{rot}_n, \mathsf{rot}_2, \mathsf{rot}_{n-1}, \mathsf{rot}_3, \ldots, \mathsf{rot}_{\lceil \frac{n+1}{2} \rceil}.$$

Due to space restrictions, we omit the proof of Theorem 7.

Theorem 7. *The greedy Gray code algorithm that rotates prefixes with interleaved longest and shortest lengths generates Corbett's order of permutations. That is,* $\mathbf{Rotator}(n) = \mathsf{Greedy}_{\mathsf{Rot}_n^{\updownarrow}}(12\cdots n)$.

6 Additional Results

In this section, we describe greedy interpretations of additional Gray codes. Formal proofs will appear in the full article. A draft of this paper illustrates each order in Table 3 (see the author's website).

A *k-combination of* $[n]$ is a subset of size k, which we represent by its selected elements $1 \le s_1 < s_2 < \cdots < s_n \le n$, or by its incidence vector in $\mathbb{B}(n,k)$ with bitwise indexing from left-to-right. A *homogeneous transposition* $\mathsf{homo}_{i,j}$ transposes the bits in positions i and j only if the bits have opposite values and the intermediate symbols are all 0s. Thus, for a given $\mathbf{b} = b_1 b_2 \cdots b_n \in \mathbb{B}(n,k)$

$$\mathsf{homo}_{i,j}(\mathbf{b}) = \begin{cases} b_1 \cdots b_{i-1}\overline{b_i}b_{i+1}\cdots b_{j-1}\overline{b_j}b_{j+1}\cdots b_n & \text{if } i < j \\ b_1 \cdots b_{j-1}\overline{b_j}b_{j+1}\cdots b_{i-1}\overline{b_i}b_{i+1}\cdots b_n & \text{if } j < i \end{cases}$$

so long as $\{b_i, b_j\} = \{0,1\}$ and $b_{i+1}\cdots b_{j-1} = 0^{j-i-1}$; otherwise, $\mathsf{homo}_{i,j}(\mathbf{b}) = \mathbf{b}$. In particular, $\mathsf{homo}_{i,i}(\mathbf{b}) = \mathbf{b}$. We prioritize the homogeneous transpositions for a given combination with $1 \le s_1 < s_2 < \cdots < s_n \le n$ as follows

$$\begin{aligned} \mathsf{Homo}_n = {} & \mathsf{homo}_{s_1,1},\ \mathsf{homo}_{s_1,2},\ \ldots,\ \mathsf{homo}_{s_1,s_2-1}, \\ & \mathsf{homo}_{s_2,s_1+1},\ \mathsf{homo}_{s_1,s_1+2},\ \ldots,\ \mathsf{homo}_{s_1,s_3-1},\ \ldots, \\ & \mathsf{homo}_{s_n,s_{n-1}+1},\ \mathsf{homo}_{s_1,s_{n-1}+2},\ \ldots,\ \mathsf{homo}_{s_1,n}. \end{aligned} \tag{10}$$

Theorem 8. *The greedy Gray code algorithm that homogeneously transposes the leftmost possible* 1 *into the leftmost possible position generates all combinations. That is,* $\mathsf{Greedy}_{\mathsf{Homo}_n}(1^k 0^{n-k})$ *generates* $\mathbb{B}(n,k)$*. Furthermore, the order is* $\mathbf{EM}(n,k)$ *when* k *is odd.*

Let $\mathbb{T}(n)$ be the set of binary trees with n vertices, which is enumerated by the nth Catalan number. When modifying a binary search tree, we can use *edge rotations* to keep the tree in balance (see [6]). In the 1990s, Ruskey, van Baronaigien and Lucas [7] showed how to recursively construct a Gray code of $\mathbb{T}(n)$, in which successive trees differ by a single edge rotation. In their Gray code $\mathbf{Tree}(n)$, they let the *label* of each vertex be its order during an inorder traversal. To describe our greedy interpretation of their Gray code, we *label* an edge between vertices with label u and label v as $\max(uv, vu)$. Given these labels, let $\mathsf{edge}_{i,j}$ denote the operation of rotating the edge with label ij, where $\mathsf{edge}_{i,j}$ has no effect if there is no such edge in the tree. We prioritize the edge rotations by lexicographically largest label as follows

$$\begin{aligned} \mathsf{Edge}_n^{\downarrow} = {} & \mathsf{edge}_{n,n-1},\ \mathsf{edge}_{n,n-2},\ \ldots,\ \mathsf{edge}_{n,1}, \\ & \mathsf{edge}_{n-1,n-2},\ \mathsf{edge}_{n-1,n-3},\ \ldots,\ \mathsf{edge}_{n-1,1},\ \ldots, \\ & \mathsf{edge}_{3,2},\ \mathsf{edge}_{3,1} \\ & \mathsf{edge}_{2,1}. \end{aligned}$$

Theorem 9. *The greedy Gray code algorithm that rotates the edge with the largest possible label generates the Ruskey, van Baronaigien, and Lucas Gray code for binary trees. That is,* $\mathsf{Greedy}_{\mathsf{Edge}_n^{\downarrow}}(1^n 0^n) = \mathbf{Tree}(n)$*, where* $1^n 0^n$ *denotes the binary tree that is a left path from the root.*

A *set partition of* $[n]$ is a collection of disjoint non-empty subsets $S_1, S_2, ..., S_k \subseteq [n]$ with $S_1 \cup S_2 \cup \cdots \cup S_k = [n]$. The disjoint sets are numbered by their minimum elements, so S_1 is the set containing value 1, and S_2 is the set containing the minimum value that is not in S_1, and so on. Let $\mathbb{S}(n)$ denote the set partitions of $[n]$, which is enumerated by the nth Bell number. For example, the following is a set partition of [6] with three subsets $S_1 = \{1, 2, 5\}, S_2 = \{3, 6\}$, and $S_3 = \{4\}$: $(\{1, 2, 5\},\ \{3, 6\},\ \{4\}) \in \mathbb{S}(6)$.

The operation $\mathsf{move}_{i,j}$ *moves* the value i into the jth subset. If j is the only value in its subset, then the operation removes the subset $\{i\}$, and if j is greater than the number of subsets then the operation creates a new subset $\{i\}$. Kaye [5] provided a Gray code for $\mathbb{S}(n)$ in which successive partitions differ by a single move. We denote this Gray code by $\mathbf{Kaye}(n)$, and then show that it has a simple greedy interpretation which prioritizes the operations as follows

$$\begin{aligned}\mathsf{Move}_n = {} & \mathsf{move}_{n,1},\ \mathsf{move}_{n,2}, ..., \mathsf{move}_{n,n}, \\ & \mathsf{move}_{n-1,1},\ \mathsf{move}_{n-1,2}, ..., \mathsf{move}_{n-1,n},\ \ldots, \\ & \mathsf{move}_{1,1},\ \mathsf{move}_{1,2}, ..., \mathsf{move}_{1,n}.\end{aligned}$$

Theorem 10. *The greedy Gray code algorithm that moves the largest possible value into the leftmost possible subset generates Kaye's set partition Gray code. That is,* $\mathbf{Kaye}(n) = \mathsf{Greedy}_{\mathsf{Move}_n}(\{1, 2, \ldots, n\})$.

References

1. Corbett, P.: Rotator graphs: An efficient topology for point-to-point multiprocessor networks. IEEE Trans. on Parallel and Distributed Systems 3, 622–626 (1992)
2. Duckworth, R., Stedman, F.: Tintinnalogia (1668)
3. Eades, P., McKay, B.: An algorithm for generating subsets of fixed size with a strong minimal change property. Inf. Proc. Letters 19, 131–133 (1984)
4. Gray, F.: Pulse code communication. U.S. Patent 2,632,058 (1947)
5. Kaye, R.: A Gray code for set partitions. Information Processing Letters 5(6), 171–173 (1976)
6. Knuth, D.E.: The Art of Computer Programming. Combinatorial Algorithms, Part 1, vol. 4. Addison-Wesley (2010)
7. Lucas, J.M., van Baronaigien, D.R., Ruskey, F.: On rotations and the generation of binary trees. Journal of Algorithms 15, 343–366 (1993)
8. Martin, M.H.: A problem in arrangements. Bull. Amer. Math. Soc. 40, 859–864 (1934)
9. Sawada, J., Williams, A.: Greedy pancake flipping. In: Latin-American Algorithms, Graphs and Optimization Symposium, LAGOS 2013 (accepted, 2013)
10. Siegel, J.: Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies. McGraw-Hill (1990)
11. Stevens, B., Williams, A.: Hamilton cycles in restricted rotator graphs. In: Iliopoulos, C.S., Smyth, W.F. (eds.) IWOCA 2011. LNCS, vol. 7056, pp. 324–336. Springer, Heidelberg (2011)
12. Wikipedia. Rotary encoder, `http://en.wikipedia.org/wiki/Rotary_encoder`
13. Zaks, S.: A new algorithm for generation of permutations. BIT Numerical Mathematics 24(2), 196–204 (1984)