Maximize the Rightmost Digit: Gray Codes for Restricted Growth Strings

Yuan (Friedrich) Qiu[1], Joe Sawada[2], and Aaron Williams[1][0000−0001−6816−4368]

[1] Williams College, Williamstown MA 01267, USA

[2] University of Guelph, Guelph ON N1G 2W1, Canada

yq1@williams.edu, jsawada@uoguelph.edu, aaron.williams@williams.edu

Abstract. The term *restricted growth string* typically refers to strings of non-negative integers $a_1a_2\cdots a_n$ (with $a_1 = 0$) in which the next symbol is at most one more than the maximum of the previous symbols: $0 \leq a_i \leq \max(a_1 \cdots a_{i-1}) + 1$ for $2 \leq i \leq n$. These strings are counted by the Bell numbers $\mathcal{B}_n$ (Oeis A000110) and encode set partitions. Kerr showed that the following algorithm generates a Gray code starting from 0^n: greedily maximize the rightmost possible digit that creates a new string. For example, the result is 000, 001, 011, 012, 010 for $n = 3$; the last transition causes the rightmost digit to decrease to 0 because that is the largest value for that digit that creates a new string. Kerr's algorithm is a special case of more general results for $\mathbf{e}$-restricted and st-restricted strings by Mansour and Vajnovszki (and Nassar), although those authors did not describe their results greedily. We show that the same greedy max-right algorithm generates restricted growth strings parameterized by $(s, f, \mathbf{c})$: $0 \leq a_1 \leq s{-}1$ and $0 \leq a_i \leq f(a_1a_2\cdots a_{i-1}) + c_i$ where f is any function with $f \geq 0$ and $c_i \geq 1$ are constants for each digit. The resulting Gray codes change a single digit by -1 or -2 (cyclically). Special cases include the binary reflected Gray code ($s = 2$, $f = 0$, $\mathbf{c} = 1^n$) and the aforementioned results. We also consider restricted growth string counted by the k-Catalan numbers and provide loopless algorithms for generating these k-Catalan strings and Bell strings.

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

1 Introduction

This paper efficiently orders and generates restricted growth strings. We first describe two common types of restricted growth strings and their significance.

1.1 Bell and Catalan Strings

The term *restricted growth string* is often defined as a string of integers (called *digits*) of the form $a_1a_2\cdots a_n$ that satisfies the following conditions,

$$a_1 = 0 \text{ and } 0 \leq a_i \leq max(a_1a_2\cdots a_{i-1}) + 1 \text{ for } 2 \leq i \leq n. \tag{1}$$

In other words, the first digit is 0, and each subsequent digit is at least 0 and at most one more than the maximum of the previous digits. For example, 0102 is a restricted growth string of length $n = 4$, but 0103 is not. The number of restricted growth strings of length $n \geq 0$ is the n^{th} Bell number $\mathcal{B}_n$ (OEIS A000110): 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, ...

Since they are enumerated by the Bell numbers, we refer to this type of restricted growth string as *Bell strings*. Bell strings provide a convenient representation for the set partitions of $[n] = \{1, 2, \ldots, n\}$, which are also Bell objects. The standard bijection puts i into the a_i^{th} part, as shown below for $n \leq 3$ [36].

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

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

Perhaps the most well-known ordering of set partitions was created by Knuth and presented by Kaye [15]. Later work by Ruskey and Savage [29] adapted the approach to Bell strings. A student project by Kerr [16] provided an alternate ordering of Bell strings (see [26]) that uses a greedy approach [40]. This paper generalizes Kerr's result from Bell strings to other restricted growth strings.

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

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

Here the bound on a_i uses the previous digit a_{i-1} rather than all previous digits. For example, 0102 is not valid since (2) is not satisfied for $i = 4$ as $2 > 0 + 1$. These *Catalan strings* of length $n \geq 0$ are counted by the n^{th} Catalan number $\mathcal{C}_n$ (OEIS A000108): 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, ...

We let $\mathbf{B}(n)$ and $\mathbf{C}(n)$ be the sets of Bell and Catalan strings of length n, respectively. Figure 3 has lists of $\mathbf{B}(n)$ and $\mathbf{C}(n)$ for $n \leq 5$. In particular, the reader can confirm that $\mathbf{C}(n) \subseteq \mathbf{B}(n)$ and in particular $\mathbf{B}(4) \setminus \mathbf{C}(4) = \{0102\}$.

Catalan strings provide an alternate representation for the large number of other Catalan objects counted by $\mathcal{C}_n$ [37]. We will also provide a simple generalization to k-Catalan strings $\mathbf{C}_k(n)$ in Section 2. There are several other types of strings counted by Catalan and k-Catalan numbers (e.g., see [42, 41]).

1.2 Generalized Restricted Growth Strings

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

$$0 \leq a_1 \leq s - 1 \text{ and } 0 \leq a_i \leq f(a_1 a_2 \cdots a_{i-1}) + c_i \text{ for } 2 \leq i \leq n \tag{3}$$

with $s \geq 1$, $f \geq 0$, and $c_i \geq 1$. In other words, the *starting digit* a_1 has s possible values $0 \leq a_1 \leq s$. Then each subsequent digit a_i is a non-negative integer limited by the sum of a *function* f that maps the previous digits $a_1 a_2 \cdots a_{i-1}$ to a non-negative integer and a positive integer *constant* c_i that depends only on the index i. (For notational convenience, we write $c = w$ and $f = w$ when $c_i = w$ and $f(a_1 a_2 \cdots a_{i-1}) = w$ for all $2 \leq i \leq n$, respectively.)

Our generalization captures a wide variety of previously studied strings as seen in Table 1. In particular, st-restricted strings are considered by Mansour and Vajnovszki [20] and Sabri and Vajnovszki [30]. These strings start with $a_1 = 0$ and then bound each successive digit by a *prefix statistic* (e.g., number of ascents): $0 \leq a_i \leq \mathsf{st}(a_1 a_2 \cdots a_{i-1}) + 1$. By carefully tailoring the statistic they can also model our $(s, f, \mathbf{c})$-restricted growth strings. Both [19] and [20] use the greedy max-right strategy discussed in Section 3, although they do not observe this interpretation. For example, $\mathsf{succ}_{1,m}$ and $\mathsf{succ}_{2,m}$ in [19] mirror our g_0 and g_1 expansions (see Section 4).

	Type	Start s	Function $f(a_1 a_2 \cdots a_{i-1})$	Constant c_i	References
(a)	binary strings	2	0	1	
(b)	k-ary strings	k	0	$k-1$	
(c)	mixed-radix strings	b_1	0	$b_i - 1$	
(d)	Bell strings	1	$\max(a_1, a_2, \ldots, a_{i-1})$	1	
(e)	RGS of order d	1	$\max(a_1, a_2, \ldots, a_{i-1})$	d	[19]
(f)	$\mathbf{e}$-restricted growth functions	1	$\max(a_1, a_2, \ldots, a_{i-1})$	e_i	[19]
(g)	restricted growth tails	k	$\max(a_1, a_2, \ldots, a_{i-1}, k)$	1	[29]
(h)	Catalan strings	1	a_{i-1}	1	
(i)	k-Catalan strings	1	a_{i-1}	$k-1$	
(j)	ascent sequences	1	$\lvert\{j \mid 2 \leq j < i, a_{j-1} < a_j\}\rvert$	1	[3, 20]
(k)	subexcedent sequences	1	i	0	[11, 20]
(l)	st-restricted strings	1	$\mathsf{st}(a_1 a_2 \cdots a_{i-1})$	1	[20, 30]

Table 1: Types of $(s, f, \mathbf{c})$-restricted growth strings. Note that names differ across the literature (e.g., [1] uses (e) max-increment, (i) increment-i, (k) K-increment).

1.3 Goals and Results

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

An initial roadblock is that Bell strings do not have a ± 1 Gray code when $n \equiv 4, 6, 7, 9 \pmod{12}$ [10, 28]. In other words, it is not possible to order the strings in an arbitrary $\mathbf{B}(n)$ so that consecutive strings differ in only one digit and only by ± 1. However, Ehrlich [10] constructed a Gray code for $\mathbf{B}(n)$ in which a single digit changes by ± 1 when considered cyclically[3] and provided a loopless implementation. On the other hand, Ruskey [28] created a CAT algorithm that allows ± 1 and ± 2 non-cyclically[4]. Li and Sawada provided a Gray code for $\mathbf{B}(n)$ as part of their *reflectable languages* framework [18], and their special values $x = 0$ and $y = 1$ arise naturally in our results.

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

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

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

1.4 Outline

Section 2 discusses k-Catalan strings and proves that they are an example of $(s, f, \mathbf{c})$-restricted growth strings. Section 3 discusses Gray codes and combinatorial generation. Section 4 provides our Gray codes for $(s, f, \mathbf{c})$-restricted growth strings. Section 5 provides new loopless algorithms for mixed-radix, k-Catalan, and Bell strings. An online version of the paper includes appendices with additional figures and Python code.

2 k-Catalan Strings

In this section we provide a natural generalization of Catalan strings. Our generalization replaces the $+1$ in (2) with $+(k-1)$ (for any fixed $k \geq 2$) as follows:

$$a_1 = 0 \text{ and } 0 \leq a_i \leq a_{i-1} + (k-1) \text{ for } 2 \leq i \leq n. \tag{4}$$

[3] Ehrlich uses a flexible notion of cyclic ± 1: $a_i = 0 \leftrightarrow a_i = m$ and $a_i = 0 \leftrightarrow a_i = m+1$ are allowed when $m = max(a_1 a_2 \cdots a_{i-1})$. We would consider the former case as ± 2.

[4] This order was also mentioned in a paper by Ruskey and Savage [29], however, the two descriptions are not equivalent.

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

$$\mathbf{C}_3(3) = \{000, 001, 002, 010, 011, 012, 013, 020, 021, 022, 023, 024\}. \quad (5)$$

We prove that these sets are counted by the k-Catalan numbers $\mathcal{C}_{k,n}$. Other objects counted by $\mathcal{C}_{k,n}$ are found in OEIS sequences A000108, A001764, A002293, A002294, A002295 for $2 \leq k \leq 6$. For example, the $|\mathbf{C}_3(3)| = 12$ strings in (5) are in bijection with the ternary trees with 3 internal nodes (OEIS A001764).

Standard Catalan strings $\mathbf{C}(n)$ are obtained from (4) with $k = 2$, and Catalan numbers are also called 2-Catalan numbers (i.e., $\mathcal{C}_n = \mathcal{C}_{2,n}$). We prove that $\mathbf{C}_k(n)$ are an example of $(s, f, \mathbf{c})$-restricted growth strings (and st-restricted strings).

Theorem 1 ([1][5]). $|\mathbf{C}_k(n)| = \mathcal{C}_{k,n}$ *for all* $n \geq 0$ *and* $k \geq 2$.

Proof. We prove that the members of $\mathbf{C}_k(n)$ are in bijective correspondence with the k-ary trees with n internal nodes, which are known to be counted by $\mathcal{C}_{k,n}$. The proof is by induction on n for a fixed k and is illustrated by Figure 1.

There is a single k-ary tree with one internal node and $\mathbf{C}_k(1) = \{0\}$, so the result is true for $n = 1$. Suppose the result holds for $n = t$. Now we extend the bijection to strings and trees with $n = t + 1$. By (4) each string in $\mathbf{C}_k(t)$ that ends with digit d is the prefix of $d + (k - 1)$ distinct strings in $\mathbf{C}_k(t + 1)$. Next consider a k-ary tree with t internal nodes and label them by a preorder traversal. Consider the location of the node x that is last in preorder; it is a leaf with label t. To grow this tree without changing the preorder traversal we can add a new leaf as a child of x or as a last child of any node on the path from the root to x that doesn't already have a kth child. Thus, if x had been added in the dth rightmost location, then the new node can added in $d + (k - 1)$ locations. So we can extend the bijection with the new node's position as a 0-based value. □

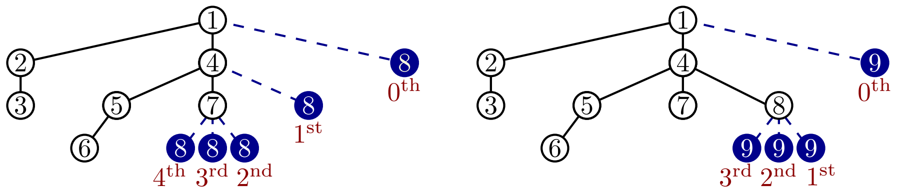
Theorem 2. *The set of k-Catalan strings $\mathbf{C}_k(n)$ are an example of $(s, f, \mathbf{c})$-restricted growth strings (as well as* st*-restricted strings).*

Proof. We claim this is true from $s = 1$, $f(a_1 a_2 \cdots a_{i-1}) = a_{i-1}$, and $c_i = k - 1$ for all $2 \leq i \leq n$. This follows from (3) as these choices force $0 \leq a_1 = s - 1 = 0$ (i.e., $a_1 = 0$) and the following bound for $2 \leq i \leq n$ that matches (4),

$$0 \leq a_i \leq f(a_1 a_2 \cdots a_{i-1}) + c_i = a_{i-1} + (k - 1). \quad (6)$$

Similarly, $\mathbf{C}_k(n)$ are st-restricted strings [20] using statistic $a_{i-1} + (k - 1)$. □

[5] This result was proven independently by the authors, however, a later literature review found that it was previously observed by Arndt [1] (Ch. 15.5). Arndt refers to k-Catalan strings as *i-increment RGS* and gives a bijection with k-ary Dyck words.



(a) The 3-ary tree with $n = 7$ nodes whose 3-ary Catalan string is $023135\underline{2}$. The location of node ⑦ is encoded as the last digit $\underline{2}$, so there are $\underline{2} + k = 5$ locations where a new leaf can be added and be last in preorder. Correspondingly, the digit d following $\underline{2}$ in a 3-ary Catalan string is one of the 5 values satisfying $0 \leq d \leq 4 = \underline{2} + (k-1)$.

(b) The 3-ary tree with $n = 8$ nodes whose 3-ary Catalan string is $0231352\underline{1}$. It is (a) with a leaf in the 1^{st} position. So there are $\underline{1} + k = 4$ locations where a new leaf could be added and be last in preorder. Correspondingly, the digit d following $\underline{1}$ in a 3-ary Catalan string is one of the 4 values satisfying $0 \leq d \leq 3 = \underline{1} + (k-1)$.

Figure 1: The bijection between k-ary trees and k-Catalan words from Theorem 1. The ith digit encodes how far from the right the node ⓘ in preorder is located.

3 Gray Codes and Combinatorial Generation

The term *Gray code* refers to an exhaustive list of some combinatorial object (parameterized by size) in which successive objects differ in some (small) way. They are named after the famous order of n-bit binary strings with Hamming distance one (i.e., a single bit's value is complemented or *flipped*) in Frank Gray's 1954 patent titled *Pulse Code Communication* [12]. The order is referred to as the *binary reflected Gray code (brgc)* and it appears below for $n = 3$, with overlines denoting the bit that changes to create the next string.

$$\mathbf{brgc}(3) = 00\overline{0}, 0\overline{0}1, 01\overline{1}, \overline{0}10, 11\overline{0}, 1\overline{1}1, 10\overline{1}, 100 \tag{7}$$

$$\mathbf{plain}(3) = 1\overleftarrow{23}, \overleftarrow{13}2, 3\overleftarrow{12}, \overrightarrow{32}1, 2\overrightarrow{31}, 213 \tag{8}$$

Plain changes predates the binary reflected Gray code by hundred years and is illustrated for $n = 3$ in (8). In this order, consecutive permutations of $[n] = \{1, 2, \ldots, n\}$ differ by a *swap* (i.e., adjacent entries are transposed) with the arrows in (8) showing a larger value "jumping over" a smaller value. The order was performed by bell-ringers in the 1600s [38], and is known as the *Steinhaus-Johnson-Trotter algorithm* due to multiple rediscoveries in the mid-20th century.

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

Countless Gray codes have been constructed over the years. Academic surveys have been written by Savage [31] and more recently Mütze [26], with Ruskey [28] and Knuth [17] devoting extensive textbook coverage to the subject. In fact, one of the issues facing this research area is the sheer breadth of results and the recursive 'tricks' that have been used to obtain them. For an interactive introduction to the area, we recommend the *combinatorial object server* `combos.org`.

3.1 Greedy Gray Code Algorithm

This decade has seen the introduction of the *greedy Gray code algorithm* [40]. The algorithm eschews recursive schemes to focus on a simple idea: build an order one object at a time by prioritizing the possible changes. For example, $\mathbf{brgc}(n)$ can be constructed starting from 0^n (where exponentiation denotes concatenation) by greedily flipping the rightmost possible bit. Similarly, plain changes starts at $12\cdots n$ and then greedily swaps the largest possible value[6]. To clarify these descriptions, consider the partial orders below.

$$\mathbf{brgc}(3) = 00\overline{0}, 0\overline{0}1, 01\overline{1}, 010, \ldots ? \tag{9}$$

$$\mathbf{plain}(3) = 12\overleftarrow{3}, 1\overleftarrow{3}2, 312, \ldots ? \tag{10}$$

Which binary string should follow 010 in (9)? Flipping the rightmost bit gives $01\overline{0} = 011$ which is already in (9). Similarly, flipping the middle bit would repeat $0\overline{1}0 = 000$. But flipping the leftmost bit gives a new string $\overline{0}10 = 110$, so it is next in the order. In (10) we cannot swap 3 to the right as it would recreate $\overrightarrow{3}12 = 132$, nor can it swap left as it is in the leftmost position. Thus, our highest priority change is to swap the next largest value 2 to the left to create $31\overleftarrow{2} = 321$.

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

The greedy Gray code algorithm has also led to new results. In particular, the greedy description of plain change order was the impetus for the *permutation language* series [13, 14, 22, 7, 6], as well as new Gray codes for signed permutations [27] and Catalan objects [9]. Similarly, our new results generalize the binary reflected Gray code and other greedy generalizations of the 'original' Gray code include [24] and [23]. Greedy Gray code results also exist for de Bruijn sequences [21] and universal cycles [34, 8], colored permutations [5], ballot sequences [39], and spanning trees [4, 2]. Greedy Gray codes can often be translated into efficient history-free algorithms (c.f., [32, 33]) but they typically do not produce sublist Gray codes (e.g., see [30, 35]). The simplicity of the greedy approach belies the complexity of the general underlying problem [25].

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

3.2 Four Greedy Definitions of the Binary Reflected Gray Code

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

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

Figure 2a illustrates the four interpretations for **brgc**(4). In the figure, we use the symbols $^{-}$ for complement, ± 1 for increment / decrement, $\oplus$ for cyclic increment, and max for maximum possible value. While the four algorithms give the same order for binary strings, we will see that the last three produce different orders for other sets of strings; we henceforth ignore the first algorithm as complements cannot be applied to non-binary digits. Eventually, we will see that max approach has a particular advantage for restricted growth strings, as observed in [19, 18].

3.3 Three Greedy Gray Codes for Mixed-Radix Strings

Let $b_1, b_2, \cdots, b_n$ be a list of positive integers called *bases*. A *mixed-radix string* with these bases is any $a_1 a_2 \cdots a_n$ with $1 \leq a_i \leq b_i - 1$ for all i. In other words, each b_i provides the number of values that the i^{th} digit can hold. Figure 2 illustrates how three of the greedy approaches mentioned in Section 3.2 generate Gray codes for the strings with bases $1, 2, 3, 4$. In each case, the reader's attention should be drawn to the different patterns created in the rightmost digit.

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

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

4 Main Result

Bell strings do not have ± 1 Gray codes (see Section 1.3), so greedily incrementing or decrementing the rightmost digit will not generate them. Similarly, greedily performing a cyclic increment of the rightmost possible digit does not work for $n \geq 7$ regardless of the start string. So of the greedy strategies discussed in Section 3.2, only maximizing the rightmost possible digit has the potential to generate all $(s, f, \mathbf{c})$-restricted growth strings. Now we prove that this is the case.

brgc(4)	$b_4b_3b_2b_1$	$\overline{}$	$\pm$	$\oplus$	max
	0000	$\overline{1}$	$+_1$	$\oplus_1$	$\max_1$
	0001	$\overline{2}$	$+_2$	$\oplus_2$	$\max_2$
	0011	$\overline{1}$	$+_1$	$\oplus_1$	$\max_1$
	0010	$\overline{3}$	$-_3$	$\oplus_3$	$\max_3$
	0110	$\overline{1}$	$+_1$	$\oplus_1$	$\max_1$
	0111	$\overline{2}$	$-_2$	$\oplus_2$	$\max_2$
	0101	$\overline{1}$	$-_1$	$\oplus_1$	$\max_1$
	0100	$\overline{4}$	$-_4$	$\oplus_4$	$\max_4$
	1100	$\overline{1}$	$+_1$	$\oplus_1$	$\max_1$
	1101	$\overline{2}$	$+_2$	$\oplus_2$	$\max_2$
	1111	$\overline{1}$	$-_1$	$\oplus_1$	$\max_1$
	1110	$\overline{3}$	$-_3$	$\oplus_3$	$\max_3$
	1010	$\overline{1}$	$+_1$	$\oplus_1$	$\max_1$
	1011	$\overline{2}$	$-_2$	$\oplus_2$	$\max_2$
	1001	$\overline{1}$	$-_1$	$\oplus_1$	$\max_1$
	1000				

(a) Greedily generating the binary reflected Gray code **brgc**(4) via complements ($\overline{}$), increments/decrements ($\pm$), cyclic increments ($\oplus$), or digit maximizing (max). The complement operation specifies the bit index to change and is specific to binary strings.

sgc$(4, \pm)$	$a_3a_2a_1$	$\pm$
	0000	$+_4$
	0001	$+_4$
	0002	$+_4$
	0003	$+_3$
	0013	$-_4$
	0012	$-_4$
	0011	$-_4$
	0010	$+_3$
	0020	$+_4$
	0021	$+_4$
	0022	$+_4$
	0023	$+_2$
	0123	$-_4$
	0122	$-_4$
	0121	$-_4$
	0120	$-_3$
	0110	$+_4$
	0111	$+_4$
	0112	$+_4$
	0113	$-_3$
	0103	$-_4$
	0102	$-_4$
	0101	$-_4$
	0100	

(b) Increment/decrement.

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

(c) Cyclic increments.

sgc$(4, \max)$	$a_3a_2a_1$	max
	0000	$\max_4$
	0003	$\max_4$
	0002	$\max_4$
	0001	$\max_3$
	0011	$\max_4$
	0013	$\max_4$
	0012	$\max_4$
	0010	$\max_3$
	0020	$\max_4$
	0023	$\max_4$
	0022	$\max_4$
	0021	$\max_2$
	0121	$\max_4$
	0123	$\max_4$
	0122	$\max_4$
	0120	$\max_3$
	0110	$\max_4$
	0113	$\max_4$
	0112	$\max_4$
	0111	$\max_3$
	0101	$\max_4$
	0103	$\max_4$
	0102	$\max_4$
	0100	

(d) Maximize digit.

Figure 2: (a) Four greedy interpretations of **brgc**(4). Each one greedily applies an operation (or operations) to the rightmost possible digit. Three of these greedy approaches also generate mixed-radix strings as seen for bases $1, 2, 3, 4$ in (b)–(d).

Theorem 3. *The greedy max-right algorithm starting from* 0^n *generates all* $(s, f, \mathbf{c})$*-restricted growth strings of length* n*, and successive strings differ by* -1 *or* -2 *in one digit where the subtractions are taken cyclically relative to* (3)*.*

Proof. Recall from (3) that $a_1 a_2 \cdots a_n$ is an $(s, f, \mathbf{c})$-restricted growth string if

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

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

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

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

$$\begin{aligned} &g_0(x_1),\ g_1(x_2),\ g_0(x_1),\ g_1(x_2),\ \ldots,\ g_0(x_p) \text{ if } p \text{ is odd} \\ &g_0(x_1),\ g_1(x_2),\ g_0(x_1),\ g_1(x_2),\ \ldots,\ g_1(x_p) \text{ if } p \text{ is even} \end{aligned} \tag{11}$$

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

$$\begin{aligned} g_0(x_i) &= x_i \cdot 0,\ x_i \cdot m,\ x_i \cdot (m-1),\ \ldots,\ x_i \cdot 2,\ x_i \cdot 1 \\ g_1(x_i) &= x_i \cdot 1,\ x_i \cdot m,\ x_i \cdot (m-1),\ \ldots,\ x_i \cdot 2,\ x_i \cdot 0 \end{aligned} \tag{12}$$

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

- The greedy max-right algorithm does indeed generate the list in (11).
- The list in (11) includes all $(s, f, \mathbf{c})$ strings of length $k + 1$.
- Successive strings in (11) differ in a single digit by -1 or -2 (cyclically).

To prove the first point, note that the greedy max-right algorithm prefers to change the rightmost digit to the maximum possible value that results in a new string. Therefore, if $x_i \cdot 0$ is the first string to be created with prefix x_i, then the algorithm will proceed by generating the list $g_0(x_i)$. Similarly, if $x_i \cdot 1$ is the first string to be created with prefix x_i, then the algorithm will proceed by generating the list $g_1(x_i)$. In both cases, all of the strings with prefix x_i are generated in succession. Therefore, when the expansion of x_i is completed, the algorithm will then set the rightmost possible digit in x_i to the maximum possible value. By induction, this means that the prefix x_i will be replaced by x_{i+1} by the next change. Finally, note that the sublist $g_0(x_i)$ ends with $x_i \cdot 1$, so the aforementioned change will result in $x_{i+1} \cdot 1$, which is the first string of $g_1(x_{i+1})$. Similarly, the sublist $g_1(x_i)$ ends with $x_i \cdot 0$, so the aforementioned change will result in $x_{i+1} \cdot 0$, which is the first string of $g_0(x_{i+1})$. Hence, the expanded sublists alternate as per (11).

The second point follows from the fact that a digit's valid values are between 0 and m inclusively. The third point follows from (12) and induction. □

5 Loopless Algorithms

In this section we provide loopless algorithms for generating multi-radix strings, k-Catalan strings, and Bell strings according to our max-right Gray codes. This improves upon the excellent CAT implementations that follow from [20]. Here we generate the strings in reverse (i.e., right-to-left) to simplify the indexing. When the next string is ready we **yield** it and continue running. For each string, except the first, we also **yield** the index of the digit that was changed to create it.

5.1 Loopless Mixed-Radix Algorithms

The **MixedRadix** function in Algorithm 1 provides the well-known loopless algorithm for generating a mixed-radix Gray code using increment/decrement (i.e., ± 1) changes (see Knuth's description in [17]). The modified function **MixedRadixMax** in Algorithm 1 instead implements our mixed-radix Gray code using max changes. In this implementation, s_i keeps track of the starting value of the corresponding i-th digit: 0 or 1 (as per Section 4).

5.2 Loopless k-Catalan Strings

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

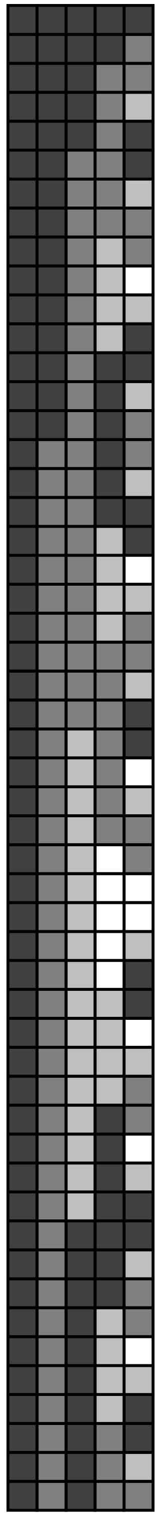
Theorem 4. CatalanStrings(n) *in Algorithm 2 looplessly generates the max-right Gray code for k-Catalan strings of length n.*

5.3 Loopless Bell Strings

In our loopless implementation of the max-right Gray code for Bell strings[7], the concept of bases is not directly used, since computing the base of any digit (which is the maximum of previous digits plus 1) is a worst-case $\Theta(n)$ operation. Instead, we store the first positions of digits equal to successively larger values

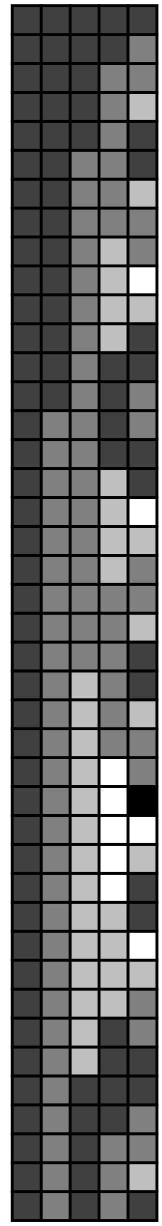
[7] Lines 14–15 of **BellStrings**(**n**) are missing in a published version of this document.

0	00	000	0000	00000
	01	001	0001	00001
		011	0011	00011
		012	0012	00012
		010	0010	00010
			0110	00110
			0112	00112
			0111	00111
			0121	00121
			0123	00123
			0122	00122
			0120	00120
			0100	00100
			0102	00102
			0101	00101
				01101
				01102
				01100
				01120
				01123
				01122
				01121
				01111
				01112
				01110
				01210
				01213
				01212
				01211
				01231
				01234
				01233
				01232
				01230
				01220
				01223
				01222
				01221
				01201
				01203
				01202
				01200
				01000
				01002
				01001
				01021
				01023
				01022
				01020
				01010
				01012
				01011



(a) Bell strings $\mathbf{B}(n)$ for $n \leq 5$.

0	00	000	0000	00000
	01	001	0001	00001
		011	0011	00011
		012	0012	00012
		010	0010	00010
			0110	00110
			0112	00112
			0111	00111
			0121	00121
			0123	00123
			0122	00122
			0120	00120
			0100	00100
			0101	00101
				01101
				01100
				01120
				01123
				01122
				01121
				01111
				01112
				01110
				01210
				01212
				01211
				01231
				01234
				01233
				01232
				01230
				01220
				01223
				01222
				01221
				01201
				01200
				01000
				01001
				01011
				01012
				01010



(b) Catalan strings $\mathbf{C}(n)$ for $n \leq 5$.

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

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

```
MixedRadix(b)
a_1 a_2 ··· a_n       ← 0 0 ··· 0
f_1 f_2 ··· f_{n+1}   ← 1 2 ··· n+1
d_1 d_2 ··· d_n       ← 1 1 ··· 1
yield a
while f_1 ≤ n
    j ← f_1
    f_1 ← 1
    a_j ← a_j + d_j
    yield j, a
    if a_j ∈ {b_j − 1, 0} then
        d_j ← −d_j
        f_j ← f_{j+1}
        f_{j+1} ← j + 1
```

```
MixedRadixMax(b)
a_1 a_2 ··· a_n       ← 0 0 ··· 0
f_1 f_2 ··· f_{n+1}   ← 1 2 ··· n+1
s_1 s_2 ··· s_n       ← 0 0 ··· 0
yield a
while f_1 ≤ n
    j ← f_1
    f_1 ← 1
    if a_j = s_j then
        if b_j = 2 and s_j = 1 then a_j ← 0
        else a_j ← b_j − 1
    else if a_j = 2 and s_j = 1 then a_j ← 0
    else a_j −= 1
    yield j, a
    if a_j = 1 − s_j then
        s_j ← a_j
        f_j ← f_{j+1}
        f_{j+1} ← j + 1
```

≥ 2 (i.e., $2, 3, 4, \ldots$) in a stack **S**. If the stack is non-empty, then its size allows us to determine a digit's maximum value. If the stack is empty, then the maximum is typically 2, since the earlier digits are comprised of 0s and 1s by (12). One exception is that these digits are all 0s precisely when the digit is being changed for the first time. To track this special case, we store whether or not a digit has ever been changed in a Boolean list **v**. Collectively, this additional information allows us to determine the maximum value for a digit in worst-case $\mathcal{O}(1)$-time.

Theorem 5. **BellStrings**(n) *in Algorithm 2 looplessly generates the max-right Gray code for Bell strings of length* n*.*

6 Final Remarks

We provided Gray codes for restricted growth strings parameterized by $(s, f, \mathbf{c})$. The orders change one digit by -1 or -2 (cyclically) and are generated from 0^n by a simple greedy rule. Our greedy max-right algorithms are not efficient, but the orders can be efficiently generated by other means. We showed this with loopless algorithms for mixed-radix strings, k-Catalan strings, and Bell strings.

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

```
CatalanStrings(n, k)
a_1 a_2 ··· a_n ← 0 0 ··· 0
f_1 f_2 ··· f_{n+1} ← 1 2 ··· n+1
s_1 s_2 ··· s_n ← 0 0 ··· 0
yield a
while f_1 < n
    j ← f_1
    f_1 ← 1
    if a_j = s_j then
        if a_j, a_{j+1}=1, 0 and k=2 then a_j←0
        else a_j ← a_{j+1} + k − 1
    else if a_j = 2 and s_j = 1 then a_j ← 0
    else a_j −= 1
    yield j, a
    if a_j = 1 − s_j then
        s_j ← a_j
        f_j ← f_{j+1}
        f_{j+1} ← j + 1
```

```
BellStrings(n)
a_1 a_2 ··· a_n ← 0 0 ··· 0
f_1 f_2 ··· f_{n+1} ← 1 2 ··· n+1
s_1 s_2 ··· s_n ← 0 0 ··· 0
S ← empty
v_1 ··· v_n ← True ··· True
yield a
while f_1 < n
    j ← f_1
    f_1 ← 1
    if a_j = s_j then
        if v_j then m ← 0; v_j ← False
        else if S is empty then m ← 1
        else pos ← top(S); m ← a_pos
        a_j ← m + 1
        if m ≠ 0 then S ← j
    else if a_j = 2 and s_j = 1 then
        a_j ← 0
        if top(S) = j then pop(S)
    else
        a_j −= 1
        if top(S) = j then pop(S)
    yield j, a
    if a_j = 1 − s_j then
        s_j ← a_j
        f_j ← f_{j+1}
        f_{j+1} ← j + 1
```

References

1. Arndt, J.: Matters Computational: ideas, algorithms, source code. Springer Science & Business Media (2010)
2. Behrooznia, N., Mütze, T.: Listing spanning trees of outerplanar graphs by pivot exchanges. arXiv preprint arXiv:2409.15793 (2024)
3. Bousquet-Mélou, M., Claesson, A., Dukes, M., Kitaev, S.: (2+ 2)-free posets, ascent sequences and pattern avoiding permutations. Journal of Combinatorial Theory, Series A **117**(7), 884–909 (2010)

4. Cameron, B., Grubb, A., Sawada, J.: Pivot Gray codes for the spanning trees of a graph ft. the fan. Graphs and Combinatorics **40**(4), 78 (2024)
5. Cameron, B., Sawada, J., Therese, W., Williams, A.: Hamiltonicity of k-sided pancake networks with fixed-spin: Efficient generation, ranking, and optimality. Algorithmica **85**(3), 717–744 (2023)
6. Cardinal, J., Hoang, H.P., Merino, A., Mička, O., Mütze, T.: Combinatorial generation via permutation languages. V. Acyclic orientations. SIAM Journal on Discrete Mathematics **37**(3), 1509–1547 (2023)
7. Cardinal, J., Merino, A., Mütze, T.: Efficient generation of elimination trees and graph associahedra. In: Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 2128–2140. SIAM (2022)
8. DiMuro, J.: Classifying rotationally-closed languages having greedy universal cycles. The Electronic Journal of Combinatorics pp. P1–35 (2019)
9. Downing, E., Einstein, S., Hartung, E., Williams, A.: Catalan squares and staircases: Relayering and repositioning Gray codes. In: Proceedings of the 35th Canadian Conference on Computational Geometry, CCCG (2023)
10. Ehrlich, G.: Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. Journal of the ACM (JACM) **20**(3), 500–513 (1973)
11. Foata, D., Han, G.N.: New permutation coding and equidistribution of set-valued statistics. Theoretical computer science **410**(38-40), 3743–3750 (2009)
12. Gray, F.: Pulse code communication. United States Patent Number 2632058 (1953)
13. Hartung, E., Hoang, H., Mütze, T., Williams, A.: Combinatorial generation via permutation languages. I. Fundamentals. Transactions of the American Mathematical Society **375**(4), 2255–2291 (2022)
14. Hoang, H.P., Mütze, T.: Combinatorial generation via permutation languages. II. lattice congruences. Israel Journal of Mathematics **244**(1), 359–417 (2021)
15. Kaye, R.: A Gray code for set partitions. Information Processing Letters **5**, 171–173 (1976)
16. Kerr, K.: Successor rule for a restricted growth string Gray code. upublished manuscript (2015)
17. Knuth, D.E.: The art of computer programming, Volume 4A: Combinatorial algorithms, Part 1. Pearson Education India (2011)
18. Li, Y., Sawada, J.: Gray codes for reflectable languages. Information processing letters **109**(5), 296–300 (2009)
19. Mansour, T., Nassar, G., Vajnovszki, V.: Loop-free Gray code algorithm for the e-restricted growth functions. Information Processing Letters **111**(11), 541–544 (2011)
20. Mansour, T., Vajnovszki, V.: Efficient generation of restricted growth words. Information Processing Letters **113**(17), 613–616 (2013)
21. Martin, M.: A problem in arrangements. Bulletin of the American Mathematical Society **40**(12), 859–864 (1934)
22. Merino, A., Mütze, T.: Combinatorial generation via permutation languages. III. Rectangulations. Discrete & Computational Geometry pp. 1–72 (2022)
23. Merino, A., Mütze, T.: Traversing combinatorial 0/1-polytopes via optimization. SIAM Journal on Computing **53**(5), 1257–1292 (2024)
24. Merino, A., Mutze, T., Williams, A.: All your bases are belong to us: Listing all bases of a matroid by greedy exchanges. In: 11th International Conference on Fun with Algorithms (FUN 2022). vol. 226, p. 22. Schloss Dagstuhl—Leibniz-Zentrum für Informatik (2022)

25. Merino, A., Namrata, Williams, A.: On the hardness of Gray code problems for combinatorial objects. In: International Conference and Workshops on Algorithms and Computation. pp. 103–117. Springer (2024)
26. Mütze, T.: Combinatorial Gray codes-an updated survey. The Electronic Journal of Combinatorics **30**(3), DS26 (2022)
27. Qiu, Y., Williams, A.: Generating signed permutations by twisting two-sided ribbons. In: Latin American Symposium on Theoretical Informatics. pp. 114–129. Springer (2024)
28. Ruskey, F.: Combinatorial generation. Preliminary working draft. University of Victoria, Victoria, BC, Canada **11**, 20 (2003)
29. Ruskey, F., Savage, C.D.: Gray codes for set partitions and restricted growth tails. Australas. J Comb. **10**, 85–96 (1994)
30. Sabri, A., Vajnovszki, V.: Two reflected Gray code-based orders on some restricted growth sequences. The Computer Journal **58**(5), 1099–1111 (2015)
31. Savage, C.: A survey of combinatorial Gray codes. SIAM review **39**(4), 605–629 (1997)
32. Sawada, J., Williams, A.: Greedy flipping of pancakes and burnt pancakes. Discrete Applied Mathematics **210**, 61–74 (2016)
33. Sawada, J., Williams, A.: Successor rules for flipping pancakes and burnt pancakes. Theoretical Computer Science **609**, 60–75 (2016)
34. Sawada, J., Williams, A., Wong, D.: Generalizing the classic greedy and necklace constructions of de Bruijn sequences and universal cycles. The electronic journal of combinatorics pp. P1–24 (2016)
35. Sawada, J., Williams, A., Wong, D.: Flip-swap languages in binary reflected Gray code order. Theoretical Computer Science **933**, 138–148 (2022)
36. Stanley, R.P.: Enumerative combinatorics volume 1 second edition. Cambridge studies in advanced mathematics (2011)
37. Stanley, R.P.: Catalan numbers. Cambridge University Press (2015)
38. Stedman, F.: Campanalogia: or the Art of Ringing Improved, With plain and easie rules to guide the Practitioner in the Ringing all kinds of Changes, To Which is added, great variety of New Peals. London (1677)
39. Vajnovszki, V., Wong, D.: Greedy Gray codes for Dyck words and ballot sequences. In: International Computing and Combinatorics Conference. pp. 29–40. Springer (2023)
40. Williams, A.: The greedy Gray code algorithm. In: Algorithms and Data Structures: 13th International Symposium, WADS 2013, London, ON, Canada, August 12-14, 2013. Proceedings 13. pp. 525–536. Springer (2013)
41. Williams, A.: Pattern avoidance for k-Catalan sequences. In: Proceedings of the 21st International Conference on Permutation Patterns. pp. 147–149 (2023)
42. Zaks, S.: Lexicographic generation of ordered trees. Theoretical Computer Science **10**(1), 63–82 (1980)

A Ternary String Gray Codes

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

$\mathbf{tgc}(3, \pm)$	$a_3a_2a_1$	$\pm$
	000	$+_3$
	001	$+_3$
	002	$+_2$
	012	$-_3$
	011	$-_3$
	010	$+_2$
	020	$+_3$
	021	$+_3$
	022	$+_1$
	122	$-_3$
	121	$-_3$
	120	$-_2$
	110	$+_3$
	111	$+_3$
	112	$-_2$
	102	$-_3$
	101	$-_3$
	100	$+_1$
	200	$+_3$
	201	$+_3$
	202	$+_2$
	212	$-_3$
	211	$-_3$
	210	$+_2$
	220	$+_3$
	221	$+_3$
	222	

(a) Increment/decrement.

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

(b) Cyclic increment.

$\mathbf{tgc}(3, \max)$	$a_3a_2a_1$	max
	000	$\max_3$
	002	$\max_3$
	001	$\max_2$
	021	$\max_3$
	022	$\max_3$
	020	$\max_2$
	010	$\max_3$
	012	$\max_3$
	011	$\max_1$
	211	$\max_3$
	212	$\max_3$
	210	$\max_2$
	220	$\max_3$
	222	$\max_3$
	221	$\max_2$
	201	$\max_3$
	202	$\max_3$
	200	$\max_1$
	100	$\max_3$
	102	$\max_3$
	101	$\max_2$
	121	$\max_3$
	122	$\max_3$
	120	$\max_2$
	110	$\max_3$
	112	$\max_3$
	111	

(c) Maximize digit.

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

B Python Code

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

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

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

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

n = 5
k = 3
total = 0
for word in looplessCatalanStrings(n,k):
  total += 1
  print(*word, sep="")
print("\ntotal: %d" % total)
```

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

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

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

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

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

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

C Preorder Preserving Gray Code for k-ary Trees

Our Gray code for k-Catalan strings appears to give a preorder preserving Gray coded for k-ary trees. This is illustrated in Figure 5 and will be investigated in future work.

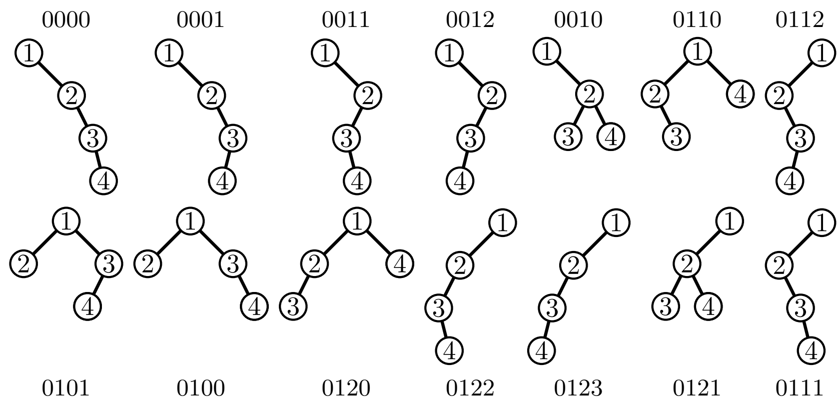


Figure 5: A preorder preserving order of binary trees with $n = 4$ nodes corresponding to the list in Figure 3b. The order is read in boustrophedon order (i.e., left-to-right on the top row and right-to-left on the bottom row).