On Lexicographic and Colexicographic Orders and the Mirror (Left-Recursive) Reflected Gray Code for *m*-ary Vectors

Valentin Penev Bakoev *

Posted Date: 22 December 2025

doi: 10.20944/preprints202512.1941.v1

Keywords: combinatorial generation; *m*-ary vectors generation; lexicographic order; colexicographic order; m-ary reflected Gray code; mirror m-ary reflectedGray code; ranking; unranking; successor; predecessor

 doi:10.20944/preprints202512.1941.v1

Article

On Lexicographic and Colexicographic Orders and the Mirror (Left-Recursive) Reflected Gray Code for m-ary Vectors

Valentin Bakoev

St. Cyril and St. Methodius University of Veliko Tarnovo, Bulgaria; v.bakoev@ts.uni-vt.bg

Abstract

In this paper, we investigate the lexicographic and colexicographic orderings of m-ary vectors of length n, as well as the mirror (left-recursive) reflected Gray code, complementing the classical m-ary reflected Gray code. We present efficient algorithms for generating vectors in each of these orders, each achieving constant amortized time per vector. Additionally, we propose algorithms implementing the four fundamental functions in generating combinatorial objects—successor, predecessor, rank, and unrank—each with time complexity $\Theta(n)$. The properties and the relationships between these orderings and the set of integers $\{0, 1, \ldots, m^n - 1\}$ are examined in detail. We define explicit transformations between the different orders and illustrate them as a digraph very close to the complete symmetric digraph. In this way, we provide a unified framework for understanding ranking, unranking, and order conversion. Our approach, based on emulating the execution of nested loops, proves to be powerful and flexible, leading to elegant and efficient algorithms that can be extended to other combinatorial generation problems. The mirror m-ary Gray code introduced here has potential applications in coding theory and related areas. By providing an alternative perspective on m-ary Gray codes, we aim to inspire further research and applications in combinatorial generation and coding.

Keywords: combinatorial generation; m-ary vectors generation; lexicographic order; colexicographic order; m-ary reflected Gray code; mirror m-ary reflected Gray code; ranking; unranking; successor; predecessor

1. Introduction

The generation of combinatorial objects in various orderings is a fundamental and widely studied topic in the literature on combinatorial algorithms [1–6] and many others. The most well-known types of orderings of generated objects are the lexicographic order and the minimum-change order, known as Gray code order, each having numerous varieties [1–3,6–9], etc. Due to their wide range of application, which are difficult to enumerate exhaustively, these orderings are of great importance for both theory and practice.

The lexicographic order of all m-ary vectors of length n is a classical concept. Considering these vectors as n-digit numbers in a number system with radix m, their lexicographic order corresponds to the increasing numerical order of the numbers when read from left to right. The *colexicographic order* of these numbers (resp. vectors) is obtained by sorting them based on their digits (resp. coordinates) from right to left, i.e., from the least significant to the most significant digit [6,9]. In [3], Section 7.1.3. (Bit reversal, p. 144), Knuth uses the term *left-right mirror image* in the context: *For our next trick, let's change* $x = (x_{63}, x_{62}, \ldots, x_1, x_0)_2$ *to its left-right mirror image,* $x^R = (x_0, x_1, \ldots, x_{62}, x_{63})_2 \ldots$ As we will see, each vector in the colexicographic order is the left-right mirror image of the corresponding vector in the lexicographic order, and vice versa. Based on this relationship and additional arguments, we propose that the classical m-ary reflected Gray code be divided into two different types. For $m = 2$, these are the binary reflected Gray code and the mirror (left-recursive) binary Gray code [10], while for $m > 2$, the

 doi:10.20944/preprints202512.1941.v1

standard type is presented in [11]. In this work, we introduce and examine the second type, referred to as the *mirror (left-recursive) m-ary reflected Gray code*. We propose algorithms for generating m-ary vectors of length n in lexicographic and colexicographic orders, as well as in the mirror m-ary reflected Gray code, and discuss their recursive implementations, highlighting both advantages and limitations. Furthermore, we present a general approach for generating these vectors by emulating the execution of n nested loops, which improves the clarity and adaptability of the algorithms. Corresponding C code is provided for straightforward use.

The functions for computing the predecessor and successor, as well as for ranking and unranking, play an important role in the generation of combinatorial objects [1,3,4,6]. Most of the authors always consider these functions (if they are created) when generating given combinatorial objects. For all orderings considered here, we propose the four basic functions, illustrated in Figure 1, with different algorithms for different orderings.

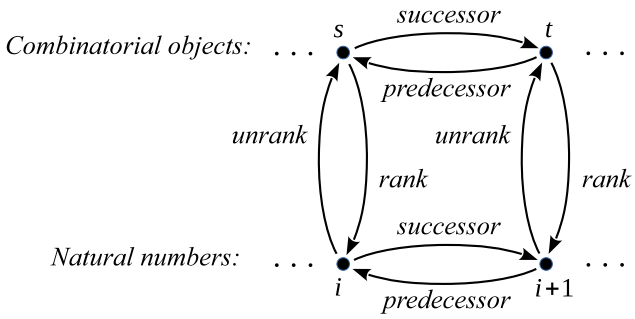

Figure 1. The four basic function used in generating combinatorial objects.

The proposed algorithms are designed for $m \geq 2$, considering that for $m = 2$ more efficient implementations using bitwise representations and operations are possible.

This article is organized as follows. Section 2 presents the generation of all m-ary vectors in lexicographic order, including the basic concepts, their properties, and the corresponding algorithms. Section 3 deals with the colexicographic order of the same vectors and has a similar structure, adding Remark 1 and Theorem 3, which describe the relationship between lexicographic and colexicographic order. Section 4 poses the problem of the two versions of the m-ary Gray reflection code that are used in the literature but are not distinguished. We present several arguments supporting the need to differentiate these versions and assign them distinct names. We then describe the generation of m-ary vectors in the Gray reflection code, together with the successor and predecessor functions, after deriving the main concepts and comparing them with the standard m-ary Gray code. Section 5 summarizes the previous sections by presenting the relationships between the considered orders and the set $J_{m^n} = \{0, 1, \ldots, m^n - 1\}$, the transformations between each pair of them, and the corresponding ranking and unranking functions. All of these are illustrated in Figure 3 as a digraph very close to the complete symmetric digraph $\overleftrightarrow{K}_5$, i.e., missing exactly two arcs. Section 6 contains several concluding remarks. Most of the results obtained here are original, while the rest are similar to known results.

2. Lexicographic Order of the m-ary Vectors

From now on, we talk about given positive integers n and $m > 1$. We consider the set of integers $J_m = \{0, 1, \ldots, m-1\}$ and its n-th Cartesian power $J_m^n = \{(x_1, x_2, \ldots, x_n) | \, x_i \in J_m, \text{ for } i = 1, 2, \ldots, n\}$, which is the set of all m-ary vectors of length n. There are m^n such vectors, i.e., $|J_m^n| = |J_m|^n = m^n$.

2.1. Definitions, Properties and Preliminary Notes

For convenience, we define J_m^n recursively as follows.

 doi:10.20944/preprints202512.1941.v1

Definition 1. *The set of all m-ary vectors of length n is defined by:*

$$J_m^n = \begin{cases} \{(0), (1), \ldots, (m-1)\}, \text{ if } n = 1, \\ \{(0, J_m^{n-1}), (1, J_m^{n-1}), \ldots, (m-1, J_m^{n-1})\}, \text{ if } n > 1\,. \end{cases}$$

where (k, J_m^{n-1}) *denotes the set of all vectors of* J_m^{n-1} *prepended by k, for* $0 \leq k \leq m-1$.

If necessary, Definition 1 can start with $J_m^0 = \{()\}$, which is the set of the empty vector (with 0 coordinates) corresponding to the empty word/string. When a matrix representation is needed, the vectors can be arranged as rows of a matrix of size $m^n \times n$.

Let $\alpha = (a_1, a_2, \ldots, a_n)$ and $\beta = (b_1, b_2, \ldots, b_n)$ be arbitrary vectors of J_m^n. The non-negative integer $\sharp\alpha = \sum_{i=1}^{n} a_i.m^{n-i}$ is called the *serial number* of α, i.e., the conversion of the n-digit radix-m number $a_1a_2\ldots a_n$ into decimal. Usually, the serial number differs from the *rank* $r(\alpha)$, which is the position of α in the sequence of vectors according to a chosen ordering, and is also called the *sequential, ordinal number* of α. The serial number is unique (like the serial number of a device), while the rank of the vector changes depending on the selected order of the vectors.

The vector α *precedes lexicographically* the vector β and is denoted by $\alpha \leq_l \beta$ if $\alpha = \beta$ or there exists an integer k, $1 \leq k \leq n$, such that $a_k < b_k$ and $a_i = b_i$ for all $i < k$. The relation $R_{\leq_l}$ on J_m^n is defined as follows: for arbitrary vectors α and $\beta \in J_m^n$, $(\alpha, \beta) \in R_{\leq_l}$ whenever $\alpha \leq_l \beta$. It is obvious that $R_{\leq_l}$ is reflexive, strictly antisymmetric (i.e., for any $\alpha, \beta \in A^n$, $\alpha \neq \beta$, either $\alpha \leq_l \beta$ or $\beta \leq_l \alpha$ and then α and β are *lexicographically comparable*) and transitive. Therefore, $R_{\leq_l}$ is a relation of *total order* on J_m^n and all vectors of J_m^n can be uniquely ordered according to $R_{\leq_l}$ in lexicographical order.

Theorem 1. *Let the vectors of* J_m^n *be defined as in Definition 1. Then:*

a) they are in lexicographic order;

b) their serial numbers form the sequence $0, 1, 2, \ldots, m^n - 1$*;*

c) $\sharp\alpha = r(\alpha)$ *for each vector* α *in the lexicographic order of* J_m^n*.*

The theorem can be proven by induction on n following Definition 1 (see the proof of Theorem 2). Its statement is illustrated in Table 1 for $m = n = 3$.

Table 1. The vectors of J_3^3 in lexicographic order and their serial numbers equal to their rank.

$\sharp\alpha = r(\alpha)$	α		$\sharp\alpha = r(\alpha)$	α		$\sharp\alpha = r(\alpha)$	α
0	(0,0,0)		9	(1,0,0)		18	(2,0,0)
1	(0,0,1)		10	(1,0,1)		19	(2,0,1)
2	(0,0,2)		11	(1,0,2)		20	(2,0,2)
3	(0,1,0)		12	(1,1,0)		21	(2,1,0)
4	(0,1,1)		13	(1,1,1)		22	(2,1,1)
5	(0,1,2)		14	(1,1,2)		23	(2,1,2)
6	(0,2,0)		15	(1,2,0)		24	(2,2,0)
7	(0,2,1)		16	(1,2,1)		25	(2,2,1)
8	(0,2,2)		17	(1,2,2)		26	(2,2,2)

2.2. Generation of the Vectors of J_m^n *in Lexicographic Order*

There are at least two ways to generate all vectors of J_m^n in lexicographic order. Following Theorem 1, the first involves converting each integer from 0 to $m^n - 1$ into a radix-m number and, if necessary, prepending leading zeros to obtain an n-digit representation. This operation (function) is

 doi:10.20944/preprints202512.1941.v1

known as *unranking*, and its inverse operation (function) is referred to as *ranking*. Both operations are based on the *Horner rule*:

$$r(\alpha) = \sharp\alpha = \sum_{i=1}^{n} a_i.m^{n-i} = ((\dots((a_1.m) + a_2).m + \dots).m + a_{n-1}).m + a_n\,.$$

In the algorithms below, the vectors are stored in global one-dimensional arrays: `a` for the lexicographic order, `c` for the colexicographic order, and `g` for the Gray code. They are defined appropriately so that their zeroth or last element can be used as a sentinel or left unused for better efficiency. Thus, the coordinates of the vectors occupy the elements with indices $1, 2, \dots, n$.

Algorithm 1: Rank and unrank functions for J_m^n in lexicographic order

```
// ranking the vector from the array a
int lex_rank (int a[], int n, int m) {
  int k= a[1];
  for (int i= 2; i <= n; i++)  k= k*m + a[i];  // Horner rule
  return k; // the rank
}
// unranking the integer k to lexicographic order,
// the result is in the array a
void unrank_in_lex (int k, int n, int m, int a[]) {
  int i= n;
  while (k > 0) { // or while (k)
    a[i--]= k % m;
    k= k/m;
  }
  while (i)  a[i--]= 0; // prepending leading zeros
}
```

Obviously, each of these functions has a time complexity $\Theta(n)$, although the operations integer division and computing the reminder modulo m are time-consuming operations when m is not a power of 2.

The second way to generate the vectors of J_m^n in lexicographic order is by using of n nested loops. This is a special case of the *general problem* of generating all n-tuples in a mixed radix system[1], which is considered by Knuth [3] [Sect. 7.2.1.1. Mixed-radix generation, Algorithm M] and Arndt [1] [Chapter 9. Mixed radix numbers]. The values of the loop variables in the body of the innermost loop form the current vector of length n—the idea is shown in the next code fragment.

Algorithm 2: Example with n nested loops

```
for (int i1= 0; i1 < m; i1++) {
   a[1]= i1;
   for (int i2= 0; i2 < m; i2++) {
      a[2]= i2;
      ...       ...      ...
               for (int in= 0; in < m; in++) {
                  a[n]= in;
                  visit (a, n);  // using the current vector
               }
      ...       ...      ...
   }
}
```

[1] This problem is equivalent with generating all: (1) generalized characteristic vectors of the subsets (submultisets) of a given multiset [2]; (2) n-tuples of the Cartesian product of n sets.

 doi:10.20944/preprints202512.1941.v1

Writing code with n nested loops is hard-coding and bad practice. So *we will emulate the execution of n nested loops*, where each has an initial value of 0, a final value of $m-1$, and a step of 1. The emulation algorithm uses the array `a`, whose elements contain the values of the loops variables (see the code above) and are the coordinates of the current vector represented by `a`. As in the generation of other combinatorial objects in [1–3,12,13], it is convenient to formulate and use a *rule of succession* for the vectors of J_m^n in lexicographic order: To obtain the next vector after the current one in the array `a`:

1) Scan the elements of the array `a` from right to left (i.e., `a[n]`, `a[n-1]`, etc.), looking for the first element that has not reached its final value $m-1$. During the scan, set each scanned element to 0.

2) If such an element is found, increase its value by 1. Otherwise, terminate, as the last vector was in the array `a`.

We examine the emulation algorithm in detail, as it is the basis for the development of the other algorithms presented here. They all use the `printn` function instead of the more general `visit` function used in [1–3]. It prints the current vector with its ordinal number (rank) in the generated sequence, using the global variable `ord_num`. Here is their C code.

Algorithm 3: Generation of the vectors of J_m^n in lexicographic order by emulating the execution of n nested loops

```
void printn (int b[], int n) {
   cout << ord_num << ":\t";
   ord_num++;
   for (int i= 1; i <= n; i++)   cout << b[i] << ", ";
   cout << endl;
}
// ...
void lex_gen (int a[], int n, int m) {
   for (int i= 1; i <= n; i++)  a[i]= 0;  // initialization
   a[0]= -1; // sentinel
   int f_v= m-1;  // final value
   int k;
   do {
      printn (a, n);
      k= n;  // rule of succession starts here
      while (a[k] == f_v)  a[k--]= 0;
      a[k]++;
   } while (k > 0);
}
```

The correctness of this algorithm can be easily proven by induction on n. Regarding its time complexity: obviously, the algorithm performs $O(n)$ operations to generate each vector after the current one and therefore a total of $O(n.m^n)$ operations. Let us estimate its time complexity more precisely. To generate all vectors, the `while` loop of the algorithm checks m^n times the last element `a[n]`, m^{n-1} times the penultimate element `a[n-1]` and so on, m^1 times the first element `a[1]`. In total, the `while` loop performs $m^n + m^{n-1} + \cdots + m^1 = m.(m^n-1)/(m-1) = \Theta(m^n)$ comparisons and the same number of assignments and subtractions. In the body of the main loop `do ... while`, other $m^n = \Theta(m^n)$ comparisons, assignments, and additions are performed (except for the operations in the function `printn`). Therefore, the total time complexity of the algorithm is $\Theta(m^n) + \Theta(m^n) = \Theta(m^n)$ to generate all m^n vectors. Thus, the average time to generate one vector is constant and the algorithm is of the type *Constant Amortized Time (CAT) algorithm* [6]. The space complexity is $\Theta(n)$ and therefore it and the time complexity are of the same type as of a loopless algorithm.

Here is the code for the corresponding recursive algorithm.

Algorithm 4: Recursive generation of the vectors of J_m^n in lexicographic order

```
void lex_gen_rec (int k) {
```

 doi:10.20944/preprints202512.1941.v1


```
   if (k > n) printn (a, n);
   else {
      for (int i= 0; i < m; i++) {
         a[k]= i;
         lex_gen_rec (k+1);
      }
   }
}
```

This function must be invoked using `lex_gen_rec(1)`. So it starts by executing the first (outermost) loop—see Algorithm 2—and each subsequent function call continues with the execution of the next (second, third, etc.) loop, with the n-th call executing the n-th (innermost) loop, and with the $(n+1)$-st call reaching the bottom of the recursion. This fully corresponds to Definition 1 and therefore, Algorithm 4 is correct.

As we said, the function `lex_gen_rec` makes additional calls to reach the bottom of the recursion. Table 2 is part of a larger table and shows the trend of the percentage ratio between the number of recursive calls and the number of vectors generated, depending on the values of n and m. Despite the additional execution time of the recursive calls (each of them for constant time), despite the additional calls, Algorithm 4 is also a CAT algorithm. It is slower, but shorter and probably clearer than the non-recursive Algorithm 3.

Table 2. The percentage ratio between the number of recursive calls and the number of vectors generated.

n \ m	1	2	3	4	5	6	7	8	9	10
...					...					...
5	600	196.87	149.79	133.30	124.99	119.99	116.67	114.28	112.50	111,11
6	700	198.44	149.93	133.32	124.99	120	116.67	114.29	112.50	111,11
7	800	199.22	149.98	133.33	125	120	116.67	114.29	112.50	111,11
8	900	199.60	149.99	133.33	125	120	116.67	114.29	112.50	111,11
...					...					...

2.3. Successor and Predecessor Functions in the Lexicographic Order

Using the the rule of succession and the code of Algorithm 3, writing the functions for computing the *successor* and *predecessor* of the current vector becomes straightforward. Both of the following functions modify the vector stored in the array `a` to compute the corresponding next or previous vector in the same array. The function `lex_successor` implements the rule of succession. The function `lex_predecessor` works in the opposite direction: it scans the elements of `a` from right to left, looking for an element with a positive value (since 0 is the minimum). During this scan, scanned elements are reset from 0 to $m-1$. When such an element is found, its value is decremented by 1. Finally, note that the last vector $(m-1, m-1, \ldots, m-1)$ has no successor, and the first vector $(0, 0, \ldots, 0)$ has no predecessor. Similar functions for mixed radix numbers are described in [1] [Sect. 9.1].

Algorithm 5: Successor and predecessor functions for the vectors of J_m^n in lexicographic order

```
bool lex_successor (int a[], int n, int m) {
   a[0] = -1;   // sentinel
   int f_v= m-1;  // final value
   int k= n;
   while (a[k] == f_v)   a[k--]= 0;
   if (0 == k)   return false;  // no successor
   else   a[k]++;
   return true;
}
```

 doi:10.20944/preprints202512.1941.v1


```
bool lex_predecessor (int a[], int n, int m) {
   a[0] = -1;   // a sentinel
   int f_v= m-1;
   int k= n;
   while (0 == a[k])   a[k--]= f_v;
   if (0 == k)   return false;  // no predecessor
   else   a[k]--;
   return true;
}
```

It is obvious that each of the two functions has running time $O(n)$.

3. Colexicographic Order of the Vectors of J_m^n

Of the various orders associated with the lexicographic order, the colexicographic (or colex) order is *the most prominent and useful alternative to the lexicographic ordering* [4]. It is used in generating combinations, permutations, partitions, bitstrings, related structures such as necklaces and Lyndon words, etc. [1,3,4,6,14] and others. Arndt notes: *The co-lexicographic (or simply colex) order is obtained by sorting with respect to the reversed strings* and *The sequence for co-lexicographic (or colex) order is such that the sets, when written reversed, are ordered lexicographically* [1] [p. 172, p. 177]. Ruskey [6] [p. 59] introduces the **Colex Superiority Principle**: *Try to use colex order (right-to-left array filling) whenever possible. It will make your programs shorter, faster, and more elegant and natural*. That is why we are consider it here. To the best of our knowledge, there is no prior publication that treats the complete generation of all m-ary vectors of length n in colexicographic order, including explicit algorithms for ranking/unranking functions. The present work appears to be the first to consider this case systematically.

3.1. Definitions, Properties and Preliminary Notes

Definition 2. *The set of all m-ary vectors of length n is defined by:*

$$J_m^n = \begin{cases} \{(0),(1),\ldots,(m-1)\}, \text{ if } n=1, \\ \{(J_m^{n-1},0),(J_m^{n-1},1),\ldots,(J_m^{n-1},m-1)\}, \text{ if } n>1\,. \end{cases}$$

where (J_m^{n-1},k) denotes the set of all vectors of J_m^{n-1} suffixed by k, for $0 \le k \le m-1$.

Analogously to the lexicographic order, the vector $\alpha = (a_1, a_2, \ldots, a_n)$ *precedes colexicographically* the vector $\beta = (b_1, b_2, \ldots, b_n)$ and is denoted by $\alpha \le_c \beta$, if $\alpha = \beta$ or there exists an integer k, $1 \le k \le n$, such that $a_k < b_k$ and $a_i = b_i$ for all $i > k$. The corresponding relation $R_{\le_c}$ is defined on J_m^n analogously: for arbitrary vectors α and $\beta \in J_m^n$, $(\alpha,\beta) \in R_{\le_c}$ whenever $\alpha \le_c \beta$. Then $R_{\le_c}$ is also reflexive, strictly antisymmetric and transitive and so it is a relation of total order on J_m^n, i.e., all vectors of J_m^n can be uniquely ordered according to $R_{\le_c}$ in colexicographic order.

Theorem 2. *Let the vectors of J_m^n be defined as in Definition 2. Then:*

a) they are in colexicographic order;

b) the sequence of their serial numbers, denoted by $\sharp J_m^n$, is:

$$\sharp J_m^n = \begin{cases} 0,1,2,\ldots,m-1, \text{ if } n=1, \\ (\sharp J_m^{n-1}).m,\ (\sharp J_m^{n-1}).m+1,\ \ldots,\ (\sharp J_m^{n-1}).m+(m-1), \text{ if } n>1, \end{cases}$$

where $(\sharp J_m^{n-1}).m + k$ means that the serial numbers of all vectors from J_m^{n-1} are multiplied by m, then k is added to each of them, for $0 \le k \le m-1$.

Proof. We will prove the theorem by mathematical induction on n, following Definition 2.

1) For $n = 1$, both statements are obviously true.

2) Let us assume that for $n > 1$ and for the vectors of J_m^{n-1}:

 doi:10.20944/preprints202512.1941.v1

a) they are in colexicographic order;

b) $(\sharp J_m^{n-2}).m$, $(\sharp J_m^{n-2}).m+1$, ..., $(\sharp J_m^{n-2}).m+(m-1)$ is the sequence of their serial numbers.

3) For the vectors of J_m^n it follows that:

a) They are in colexicographic order, due to the inductive hypothesis (the vectors of J_m^{n-1} are in colexicographic order) and each vector of the type (J_m^{n-1}, i) precedes colexicographically each vector of the type (J_m^{n-1}, j) when $0 \le i \le j \le m-1$.

b) They are obtained by taking the vectors of J_m^{n-1} exactly m times and appending a new coordinate to each copy. In other words, $J_m^n = \{(J_m^{n-1}, 0), (J_m^{n-1}, 1), \ldots, (J_m^{n-1}, m-1)\}$. This means that the vectors of (J_m^{n-1}, k) are obtained by shifting the vectors of J_m^{n-1} by one position to the left, and the new coordinate k occupies the n-th position in each of them, for $k = 0, 1, \ldots, m-1$. According to the definition of the serial number, this shift corresponds to multiplication by m. Thus, and in accordance with the inductive proposition, the serial numbers $(\sharp J_m^{n-1}).m + k$ are obtained from the corresponding serial numbers $\sharp J_m^{n-2}$ multiplied by m and then adding k to each of them, for $k = 0, 1, \ldots, m-1$. Hence:

$$\sharp J_m^n = (\sharp J_m^{n-1}).m,\ (\sharp J_m^{n-1}).m+1,\ \ldots,\ (\sharp J_m^{n-1}).m+(m-1)\,.$$

Thus, the theorem holds. □

The statements of the theorem are illustrated in Table 3 and in Figure 2 for $m = n = 3$.

Table 3. The vectors of J_3^3 in colexicographic order, their ranks and serial numbers.

$r(\alpha)$	α	$\sharp\alpha$		$r(\alpha)$	α	$\sharp\alpha$		$r(\alpha)$	α	$\sharp\alpha$
0	(0,0,0)	0		9	(0,0,1)	1		18	(0,0,2)	2
1	(1,0,0)	9		10	(1,0,1)	10		19	(1,0,2)	11
2	(2,0,0)	18		11	(2,0,1)	19		20	(2,0,2)	20
3	(0,1,0)	3		12	(0,1,1)	4		21	(0,1,2)	5
4	(1,1,0)	12		13	(1,1,1)	13		22	(1,1,2)	14
5	(2,1,0)	21		14	(2,1,1)	22		23	(2,1,2)	23
6	(0,2,0)	6		15	(0,2,1)	7		24	(0,2,2)	8
7	(1,2,0)	15		16	(1,2,1)	16		25	(1,2,2)	17
8	(2,2,0)	24		17	(2,2,1)	25		26	(2,2,2)	26

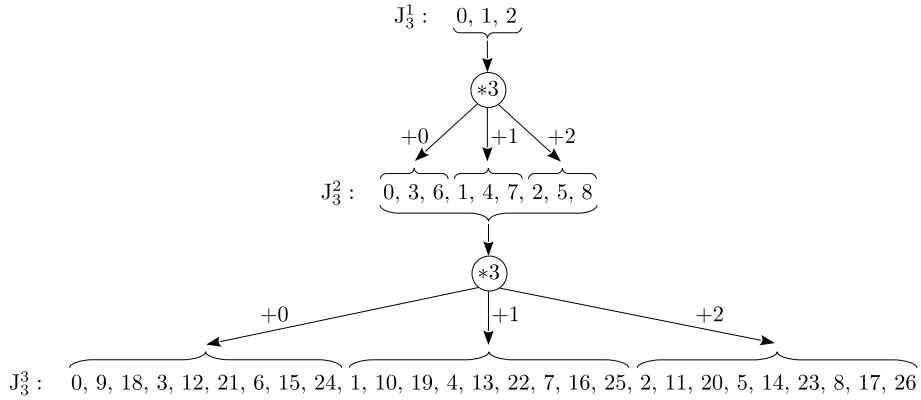

Figure 2. The sequence of serial numbers of the vectors of J_3^3 and its derivation according to Theorem 2.

We have developed an algorithm that generates the sequence $\sharp J_m^n$ of the serial numbers of the vectors of J_m^n. Its time complexity is $\Theta(m^n)$, i.e., it is a CAT algorithm. However, its space complexity is of the same type. To obtain the vectors of J_m^n in colexicographic order, it is only necessary to transform each of its terms into an m-ary vector with n coordinates. For example, the function `unrank_in_colex` (see Algorithm 6) can be used for this purpose, in $\Theta(n)$ time per term. But the overall time and space complexity of this approach is not acceptable.

 doi:10.20944/preprints202512.1941.v1

Generating the vectors in lexicographic order and then applying the coordinate-reversal function (see Remark 1) gives the same $\Theta(n.m^n)$ time complexity but only $\Theta(n)$ space complexity.

Remark 1. *Before considering the algorithms for the colexicographic order, it is important to note:*

1. ***Relationship between the lexicographic and colexicographic order**: The colexicographic order can be obtained from the lexicographic order and vice versa, by reflecting the coordinate order of the vectors in the respective ordering. That is, $(a_1, a_2, \ldots, a_n) \leq_c (b_1, b_2, \ldots, b_n)$, iff $(a_n, a_{n-1}, \ldots, a_1) \leq_l (b_n, b_{n-1}, \ldots, b_1)$ [6].*
2. *We define the **coordinate-reversal** function φ such that $\varphi : J_m^n \to J_m^n$, and for any vector $\alpha = (a_1, a_2, \ldots, a_n)$, $\varphi(\alpha) = \alpha^R = (a_n, \ldots, a_2, a_1)$. Thus, the coordinates of α^R are left-right **mirror image** of those of α and vice versa [3] [p. 144]. Obviously φ is a bijection and also $\varphi(\varphi(\alpha)) = \varphi(\alpha^R) = \alpha$ and therefore φ is an involution.*
3. *The **fixed points** of the function φ are the palindromes, i.e., $\alpha = \alpha^R$. Their number is $m^{\lceil n/2 \rceil}$.*
4. *Comparing Definition 1 and Definition 2 and in accordance with Theory of Formal Languages, we note that Definition 1 uses **right recursion** (or substitution—the words are derived from left to the right and therefore the rightmost symbols change the fastest, compared to the leftmost symbols, which change the slowest), while Definition 2 uses **left recursion**.*

From the statements in Remark 1 it follows immediately:

Theorem 3. *If the vectors of J_m^n are listed in lexicographic order and we apply the coordinate-reversal function φ to each vector, then the resulting sequence is the colexicographic order of these vectors, and conversely. Moreover:*

1. *For every vector α in lexicographic order, $r(\alpha) = r(\alpha^R)$, where α^R is the corresponding vector in colexicographic order. Also, $\sharp\alpha = \sharp\alpha^R$ if and only if $\alpha = \alpha^R$, that is, α is a palindrome.*
2. *Let L and C be $m^n \times n$ matrices representing the vectors of J_m^n, listed in lexicographic and colexicographic order, respectively. Then the i-th column of L coincides with the $(n-i+1)$-st column of C, for $i = 1, 2, \ldots, n$.*

The assertion of the Theorem 3 is illustrated in Table 1 and Table 3, for $m = n = 3$. The last statement of the theorem means that the coordinate-reversal function can be extended to a *column-reversal* function for these matrices.

3.2. Generation of the Vectors of J_m^n in Colexicographic Order

We now present the algorithms for colexicographic order. They follow Definition 2, Theorem 2 and Remark 1 and are analogous to the corresponding algorithms for lexicographic order. Hence their correctness can be proven in the same way. Their time complexity is of the same type as that of the corresponding algorithms for lexicographic order. So, instead of analogous reasoning, we provide only brief comments.

We start with the functions for ranking a vector $\gamma = (c_1, c_2, \ldots, c_n) \in J_m^n$ and unranking an integer k, $0 \leq k < m^n$, into a vector of J_m^n in colexicographic order. They use the following *reverse version* of the Horner rule:

$$r(\gamma) = \sum_{i=1}^{n} c_i.m^i = ((\ldots((c_n.m) + c_{n-1}).m + \ldots) + c_2).m + c_1\,.$$

Algorithm 6: Ranking and unranking functions for the vectors of J_m^n in colexicographic order

```
// ranking the vector from the array c - colex order
int colex_rank (int c[], int n, int m) {
   int k= c[n];
   for (int i= n-1; i > 0; i--)  k= k*m + c[i];  // reversed Horner rule
   return k; // the rank
}
```

 doi:10.20944/preprints202512.1941.v1


```
// unranking the integer k in colex order, the result is in the array c
void unrank_in_colex (int k, int n, int m, int c[]) {
   int i= 1;
   while (k > 0) {
      c[i++]= k % m;
      k= k/m;
   }
   while (i <= n)  c[i++]= 0; // adding trailing zeros
}
```

Another way to generate is to simply change the assignments in Algorithm 2: `a[n]= i1; a[n-1]= i2; ..., a[1]= in`. This makes the following algorithms more understandable.

Algorithm 7: Generation of the vectors of J_m^n in colexicographic order by emulating the execution of n nested loops

```
void colex_gen (int c[], int n, int m) {
   for (int i= 1; i <= n; i++)   c[i]= 0; // initialization
   c[n+1]= -1; // sentinel
   int f_v= m-1; // final value
   int k;
   do {
      printn (c, n);
      k = 1;
      while (c[k] == f_v) c[k++]= 0;
      c[k] += 1;
   } while (k <= n);
}
```

Algorithm 8: Recursive generation of the vectors of J_m^n in colexicographic order

```
void colex_gen_rec (int k) {
   if (k < 1)   printn (a, n);
   else {
      for (int i= 0; i < m; i++) {
         a[k]= i;
         colex_gen_rec (k-1);
      }
   }
} // call this function by colex_gen_rec (n);
```

Everything mentioned in the comparison of recursive and non-recursive algorithms for generating in lexicographic order (including Table 2) applies equally to the corresponding algorithms for colexicographic order.

3.3. Successor and Predecessor Functions in the Colexicographic Order

Algorithm 9: Successor and predecessor functions for the vectors of J_m^n in colexicographic order

```
// successor - for colex order
 bool colex_successor (int c[], int n, int m) {
    c[n+1] = -1;   // sentinel
    int f_v= m-1;
    int k= 1;
    while (c[k] == f_v)  c[k++]= 0;
    if (k == n+1)  return  false;  // no successor
    else  c[k] += 1;
    return  true;
```

 doi:10.20944/preprints202512.1941.v1


```
}
// predecessor - for colex order
bool colex_predecessor (int c[], int n, int m) {
    c[n+1] = -1;   // sentinel
    int f_v= m-1;
    int k= 1;
    while (0 == c[k])
        c[k++]= f_v;
    if (k == n+1)  return  false;  // no predecessor
    else  c[k] -= 1;
    return  true;
}
```

4. Mirror *m*-Ary Reflected Gray Code

We recommend that the reader refer to the open access article [11], as we cannot repeat its content regarding *m*-ary reflected Gray code here.

4.1. About the Two Versions of the m-ary Reflected Gray Code

This section is a natural extension of [11] and a generalization of [10]. In the first article, we examined in detail the *m*-ary reflected Gray code, as well as the *m*-ary modular (or shifted) Gray code. In the second article, we introduced and investigated a version of the Binary Reflected Gray Code (BRGC), called the *mirror (left-recursive) binary Gray code*, and compared it with the standard BRGC. We also argued for treating these two codes as distinct. Most of these arguments remain valid for the *m*-ary reflected Gray code, and additional sources and reasons support this distinction. Taken together, they convincingly show that these two versions of the *m*-ary reflected Gray code should be considered distinct and, consequently, should have different names. Therefore, we refer to the new version as the *mirror m-ary reflected Gray code*, or more precisely, the *left-recursive* (*L-R*) *m-ary reflected Gray code*, and to the standard version as the *m-ary reflected Gray code*, or more precisely, the *right-recursive* (*R-R*) *m-ary reflected Gray code*—see Definition 3 and the following paragraph. The terms *left-recursive* and *right-recursive* (proposed by Krassimir Manev) accurately characterize the two versions, since the order of the coordinates of any vector in one version is the mirror image of the order of the coordinates of the corresponding vector in the other version. Hence, the term *mirror* may refer to either of them. Below, we present the main arguments supporting this distinction.

1. **Relationship between the lexicographic and colexicographic order.** The connection between the lexicographic and colexicographic order of the *m*-ary vectors is given by the coordinate-reversal function φ defined in Remark 1. Although these orders appear closely related, they are distinct and have separate names. The same relationship exists between the *m*-ary reflected Gray code and its mirror version. This justifies distinguishing them and calling them by different names.
2. **Equivalence versus identity.** Some sources (e.g., [15], ([16]) [Definition 2.1.1]) state that Gray codes obtained from each other by permuting coordinates are called equivalent. However, equivalence does not imply identity; it merely shows a structural relationship between the codes.
3. **Ambiguities in the literature.** Many authors, when discussing an *m*-ary reflected Gray code, actually talk about a mirror *m*-ary reflected Gray code, or talk about both codes using the same name. When readers are not careful or do not distinguish between these codes, they can confuse them. For example:
    (a) In [5] the authors first give a right-recursive definition of BRGC and show an example of it. They also define the transition sequence, where the coordinates are numbered from right to left, and emphasize the connection between it and the generation of the codewords. They also give a left-recursive definition of BRGC, but treat the two codes as the same code. In [12,13], the authors define and use only the left-recursive (mirror) BRGC and show the generated codewords. They formulate and prove the rule of succession for this

 doi:10.20944/preprints202512.1941.v1

code—which coordinate must be changed in the current codeword to obtain the next codeword, in fact the next term of the transition sequence. The same, but more thoroughly and in an optimized way, is done in [3] [Algorithm G].
We can summarize: all these algorithms generate BRGC when the coordinates are numbered from right to left, for example $(g_n, \ldots, g_2, g_1)$. For each $i, 1 \leq i \leq n$, the i-th coordinate of the current vector is stored in element `g[i]` of array `g`. Thus, the algorithms output BRGC if the array `g` is printed from right to left, otherwise they output the vectors of the mirror BRGC.

(b) In [6], Ruskey defines recursively (starting from the empty string) the standard BRGC and relates it to a Hamiltonian cycle on the Boolean cube, the Towers of Hanoi problem, transition sequence and the generation of BRGC from it, as well as the ranking and unranking functions for BRGC. He then presents a left-recursive definition (via formula (5.3), p. 120), which corresponds to the mirror BRGC. Ruskey explicitly notes: *Note that we are appending rather than prepending. In developing elegant natural algorithms this has the same advantage that colex order had over lex order in the previous chapter*. Thus, the algorithms he proposes (indirect and direct) generate the mirror BRGC.

(c) In [17], the author discusses and illustrates (in Table 1) the usual (right-recursive) k-ary reflected Gray code. He derives a non-recursive algorithm that generates its codewords and outputs them from right to left. In this way, the algorithm reverses the coordinates of the generated codewords and actually generates the left-recursive k-ary reflected Gray code. In [18], the author recursively defines the usual N-ary reflected Gray code and proposes Algorithm 1, which generates its codewords. Instead of printing them, the algorithm outputs the message "codeword available". If we print them, we observe that they are the codewords of the left-recursive N-ary reflected Gray code. The same can be seen in [19], where the authors use the same definition and algorithm.

4. **Algorithmic differences.** The algorithms for generating the vectors of the two versions of m-ary reflected Gray code are similar, but have subtle differences, as we will see later. The same applies to the four basic functions for these versions.

4.2. Definitions, Properties and Preliminary Notes

The Gray codes under consideration are defined as follows.

Definition 3. *The sequence of vectors of J_m^n, ordered in mirror m-ary reflected Gray code, is denoted by $\widetilde{G}_m(n)$ and defined as:*

If $n = 1$ then $\widetilde{G}_m(1) = ((0), (1), \ldots, (m-1))$.

If $n > 1$ then $\widetilde{G}_m(n) = ((\widetilde{G}_m(n-1), 0), (\widetilde{G}_m^r(n-1), 1), (\widetilde{G}_m(n-1), 2), \ldots, (\widetilde{G}_m^*(n-1), m-1))$, where:

a) $\widetilde{G}_m(n-1)$ is the m-ary reflected Gray code of length $n-1$;

b) $\widetilde{G}_m^r(n-1)$ is again the m-ary reflected Gray code of length $n-1$, but its vectors are reflected, i.e., taken in reverse order;

c) $(\widetilde{G}_m^*(n-1), k)$ denotes $(\widetilde{G}_m(n-1), k)$ if k is even, or $(\widetilde{G}_m^r(n-1), k)$ if k is odd.

The sequence of vectors of J_m^n, ordered in an m-ary reflected Gray code, is denoted by $G_m(n)$. The definition of $G_m(n)$ is similar to Definition 3, with the only difference that the new coordinate is added at the beginning of the vectors, instead of at the end—for example, $(k, G_m^*(n-1))$, which means $(k, G_m(n-1))$ when k is even, or $(k, G_m^r(n-1))$ when k is odd, for $k = 0, 1, \ldots, m-1$.

The definitions of $G_m(n)$ and $\widetilde{G}_m(n)$ are illustrated in Tables 4 and 5 for $m = n = 3$.

After these definitions, we note that everything said in Remark 1 is valid for the m-ary Gray code and the mirror m-ary Gray code, and we will omit an analogous remark. The following assertion corresponds to Theorem 3.

 doi:10.20944/preprints202512.1941.v1

Table 4. The vectors of $G_3(3)$, their ranks and serial numbers.

$r(\alpha)$	α	$\sharp\alpha$		$r(\alpha)$	α	$\sharp\alpha$		$r(\alpha)$	α	$\sharp\alpha$
0	(0,0,0)	0		9	(1,2,2)	17		18	(2,0,0)	18
1	(0,0,1)	1		10	(1,2,1)	16		19	(2,0,1)	19
2	(0,0,2)	2		11	(1,2,0)	15		20	(2,0,2)	20
3	(0,1,2)	5		12	(1,1,0)	12		21	(2,1,2)	23
4	(0,1,1)	4		13	(1,1,1)	13		22	(2,1,1)	22
5	(0,1,0)	3		14	(1,1,2)	14		23	(2,1,0)	21
6	(0,2,0)	6		15	(1,0,2)	11		24	(2,2,0)	24
7	(0,2,1)	7		16	(1,0,1)	10		25	(2,2,1)	25
8	(0,2,2)	8		17	(1,0,0)	9		26	(2,2,2)	26

Table 5. The vectors of $\widetilde{G}_3(3)$, their ranks and serial numbers.

$r(\alpha)$	α	$\sharp\alpha$		$r(\alpha)$	α	$\sharp\alpha$		$r(\alpha)$	α	$\sharp\alpha$
0	(0,0,0)	0		9	(2,2,1)	25		18	(0,0,2)	2
1	(1,0,0)	9		10	(1,2,1)	16		19	(1,0,2)	11
2	(2,0,0)	18		11	(0,2,1)	7		20	(2,0,2)	20
3	(2,1,0)	21		12	(0,1,1)	4		21	(2,1,2)	23
4	(1,1,0)	12		13	(1,1,1)	13		22	(1,1,2)	14
5	(0,1,0)	3		14	(2,1,1)	22		23	(0,1,2)	5
6	(0,2,0)	6		15	(2,0,1)	19		24	(0,2,2)	8
7	(1,2,0)	15		16	(1,0,1)	10		25	(1,2,2)	17
8	(2,2,0)	24		17	(0,0,1)	1		26	(2,2,2)	26

Theorem 4. *If the vectors of J_m^n are listed in reflected Gray code and we apply the coordinate reversal function φ to each vector, then the resulting sequence is the mirror Gray code of these vectors and vice versa. Moreover:*

1. *For every vector $\alpha \in G_m(n)$, we have $r(\alpha) = r(\alpha^R)$, where α^R is the corresponding vector in $\widetilde{G}_m(n)$. Also, $\sharp\alpha = \sharp\alpha^R$ if and only if $\alpha = \alpha^R$, that is, α is a palindrome.*
2. *Let M and $\widetilde{M}$ be $m^n \times n$ matrices representing the vectors of $G_m(n)$, and $\widetilde{G}_m(n)$, respectively. Then the i-th column of M coincides with the $(n-i+1)$-st column of $\widetilde{M}$, for $i = 1, 2, \ldots, n$.*

The assertion of the theorem is illustrated in Table 4 and Table 5 for $m = n = 3$.

Theorem 1 in [18] states that the *m-ary reflected Gray code is a cyclic code when its radix m is even, and is not cyclic otherwise*. This statements is also true for the mirror m-ary Gray code, since the two codes are related via the coordinate-reversal function.

Theorem 1 (or Theorem 2 in [18]) is important for all algorithms working with m-ary reflected Gray code in [11] and so we give its statement.

Theorem 1. [11] *Let $v = (v_1, v_2, \ldots, v_n)$ be a codeword of the m-ary reflected Gray code $G_m(n)$ for $m \geq 2$ and $n \geq 2$. When $S_i = \sum_{j=1}^{i-1} v_j$ is even, then v_i, and for $1 < i \leq n$, is in a subsequence of digits in ascending order; otherwise, it is in a subsequence of digits in descending order.*

We propose an analogous statement for the mirror m-ary reflected Gray code, which will be important for the following algorithms. It follows directly from this theorem and the last statement of Theorem 4, applied to an arbitrary row of the matrix M and its corresponding row of the matrix $\widetilde{M}$.

Theorem 5. *Let $\alpha = (a_1, a_2, \ldots, a_n)$ be an arbitrary vector from $\widetilde{G}_m(n)$. When $S_i = \sum_{j=i+1}^{n} a_j$ is even, then the coordinate a_i belongs to a subsequence of i-th coordinates in ascending order; otherwise, it belongs to a subsequence of i-th coordinates in descending order, for $1 \leq i < n$.*

4.3. Generation of All Vectors of $\widetilde{G}_m(n)$

We start with an algorithm that generates this Gray code via n **nested loops**. It is easy to reverse the loops in Algorithm 3 in [11]: the first (outermost) with the n-th (innermost), the second with the

 doi:10.20944/preprints202512.1941.v1

$(n-1)$-st, etc. However we propose another version based on Theorem 5 and closer to its recursive version, which is Algorithm 11. We illustrate the idea of such an algorithm by an example with $n = 4$ nested loops—see Algorithm 10.

4.3.1. Nested Loops Algorithm.

The algorithm for generating $\widetilde{G}_m(n)$ via n nested loops uses two arrays, `g` and `steps`, each of length n, storing respectively the current vector and the steps for changing the elements of `g`. When `steps[i] = 1`, the element (coordinate) `g[i]` is increased by one, and when `steps[i] = -1`, the element `g[i]` is decreased by one, for $i = 1, 2, \ldots, n$. The variable `sum` maintains the sum of coordinates in accordance with Theorem 5, i.e., in the i-th loop `sum= g[n]+g[n-1]+ ... + g[i]`, which determines the value of step[i-1], for $i = 2, 3, \ldots, n$. Note the role of the operators `if (ik < m-1) g[k] += steps[k];` at the end of the body of the k-th loop, for $k = 1, 2, \ldots, n$. When the final value $m-1$ of `ik` is reached, the value of `g[k]` (which is 0 or $m-1$) remains unchanged until some other coordinate changes. Other details can be seen in the following example, as well as in the comments to the following algorithms for generating $\widetilde{G}_m(n)$.

Algorithm 10: Generation of $\widetilde{G}_m(n)$ via n nested loops—an example with $n = 4$

```
void four_nested_loops (int m) {
   for (int i= 1; i <= 4; i++) { // initialization
      g[i]= 0;  steps[i]= 1;
   }
   int sum;
   for (int i4= 0; i4 < m; i4++) {
      sum = g[4];
      if (sum & 1) steps[3]= -1;
      else steps[3]= 1;
      for (int i3= 0; i3 < m; i3++) {
         sum = g[4] + g[3];
         if (sum & 1) steps[2]= -1;
         else steps[2]= 1;
         for (int i2= 0; i2 < m; i2++) {
            sum = g[4] + g[3] + g[2];
            if (sum & 1) steps[1]= -1;
            else steps[1]= 1;
            for (int i1= 0; i1 < m; i1++) {
               printn (g, 4);
               if (i1 < m-1) g[1] += steps[1];
            }
            if (i2 < m-1) g[2] += steps[2];
         }
         if (i3 < m-1) g[3] += steps[3];
      }
      if (i4 < m-1) g[4] += steps[4];
   }
}
```

Proof of the correctness. We will prove that Algorithm 10 with n nested loops generates the vectors of $\widetilde{G}_m(n)$ by mathematical induction on n. The idea of the proof is visible in Algorithm 10. It is clear that for $n = 1$, the algorithm is correct. Assume it is correct for $n > 1$ (for example, for $n = 3$) nested loops. When we add the outermost $(n+1)$-st loop (for instance, the fourth), it executes all n nested loops m times. In each iteration, it sequentially assigns the values $0, 1, \ldots, m-1$ to the last element `g[n+1]`, and for each such assignment, the nested loops are executed. Each of these values contributes to the value of the variable `sum` and:

 doi:10.20944/preprints202512.1941.v1

1) When `g[n+1]` is even, the algorithm operates according to Theorem 5 and Definition 3, i.e., with the same step values as for generating the vectors of $\widetilde{G}_m(n)$. By the inductive hypothesis, this subset of vectors of $\widetilde{G}_m(n+1)$ is generated correctly.

2) When `g[n+1]` is odd, the algorithm again works according to Theorem 5 and Definition 3, i.e., with the opposite step values compared to those for generating the vectors of $\widetilde{G}_m(n)$. In this way, it generates the vectors of $\widetilde{G}^r_m(n)$ and by the inductive hypothesis this subset is also generated correctly.

Therefore, all vectors of $\widetilde{G}_m(n+1)$ are generated correctly. □

4.3.2. Recursive Algorithms

As we said:

1) In [6], Ruskey proposes analogous algorithms that generate the mirror BRGC. He calls Algorithm 5.2 *indirect*, since the recursion is indirect—the two recursive functions call each other (as at Algorithm 2 in [11]). He proposes Algorithm 5.3, called *direct*—with one function that calls recursively itself. Ruskey formulates the **Direct Gray Code Algorithm Superiority Principle**: *Try to develop direct algorithms whenever possible. It will make your programs more flexible and easier to analyze*.

2) The indirect Algorithm 1 in [18,19] generates recursively the mirror m-ary reflected Gray code.

So we will consider a direct recursive algorithm, analogous to Algorithm 4 in [11]. This algorithm recursively implements the execution of n nested loops—see and compare with Algorithm 10. It supports and passes the variable `p` (means *parity*), which is analogous to the variable `sum`, namely `p` is `sum & 1`. It determines the value of the variable `step` ($+1$ or -1), which changes `g[h]`. Here is its code.

Algorithm 11: Recursive generation of the vectors of $\widetilde{G}_m(n)$ for given n and m

```
void M_Gray_refl_gen_rec (int h, int p, int n, int m) {
// parity: p=0 - even, ascending order, p=1 - odd, descending order
   if (h <= 0) return;
   M_Gray_refl_gen_rec (h - 1, (p + g[h]) & 1, n, m);
   int step= 1 - 2*p;  // step = +1 when p=0, or step = -1 when p=1
   for (int i = 1; i < m ; i++) {
      g[h] = g[h] + step;
      printn (g, n);
      M_Gray_refl_gen_rec (h - 1, (p + g[h]) & 1, n, m);
   }
}   // call  M_Gray_refl_gen_rec (n, 0, n, m);  where necessary
```

The correctness of Algorithm 11 can be proven analogously to that of Algorithm 10. Regarding its implementation, we can repeat the same thing that we said about Algorithm 4. Experimental results show that Algorithm 11 performs additional recursive calls and the same data in Table 2 are valid for it as well.

4.3.3. Nested Loops Emulation Algorithm

A short and simple non-recursive algorithm for generating the sequence $G_m(n)$ or $\widetilde{G}_m(n)$—depending on the output—is presented in [17]. It is derived after the proof of three theorems.

Here we propose a similar algorithm derived in a simpler way, we will **emulate the execution of the loops** in the algorithm for generating $\widetilde{G}_m(n)$ via n nested loops. The reflection in it, as in Algorithm 10, is performed according to the variable `sum`, so that in two successive executions of the same nested loop, the corresponding element of the array `g` is increased in the first execution and decreased in the second or vice versa. In other words, it is as if we have two alternative loops: the first, whose variable increases from 0 to $m-1$ with a step of $+1$, and the second, whose variable decreases from $m-1$ to 0 with a step of -1.

Instead of the variable `sum`, when emulating the execution of n nested loops, we will use a one-dimensional integer array `final_vals`, whose elements represent the final values of the loops: they are set to $m-1$ when the variable of the corresponding loop is incremented, or to 0 otherwise. Then the elements of `steps` take the values $+1$ or -1 respectively, used to change the i-th coordinate `g[i]`

 doi:10.20944/preprints202512.1941.v1

by that value, for $i = 1, 2, \ldots, n$. Here it is also convenient to formulate and use the **rule of succession** for the mirror Gray code $\widetilde{G}_m(n)$: To obtain the next vector after the given one in the array g:

1) Scan the elements of g from left to right (i.e., `g[1]`, `g[2]`, etc.), looking for the first element that has not reached its final value. During the scan, swap the values: `steps[i]= -steps[i]` and those of their corresponding elements: if `steps[i]=1`, then `final_vals[i]=m-1`, otherwise `final_vals[i]=0`.

2) If the scan stops at an index $k \leq n$, assign `g[k]= g[k] + steps[k]`. Otherwise, terminate, as the last vector was in the array g.

Note that this rule applies under the assumption that the arrays `steps` and `final_vals` contain the correct values corresponding to the elements of g. However, this need not always be the case, so we consider two situations, here and in Section 4.4.

First case: the arrays `steps` and `final_vals` contain the correct values. In this situation, we can construct the algorithm by sequentially applying the rule of succession. It starts with the all-zero vector and then all elements of the array `steps` should be set to $+1$, and these of the array `final_vals` should be set to $m - 1$. The corresponding C code is given below.

Algorithm 12: Non-recursive generation of the vectors of $\widetilde{G}_m(n)$ for given n and m

```
void M_Gray_refl_gen (int n, int m) {
   int max_fv= m-1; // maximum final value
   for (int i = 0; i <= n; i++) {  // initialization
      g[i]= 0;
      steps[i]= 1;
      final_vals[i]= max_fv;
   }
   int k;
   do {
      printn (g, n);
      k= 1;
      while (g[k] == final_vals[k]) {
         steps[k]= -steps[k];
         if (steps[k] < 0)  final_vals[k]= 0;
         else  final_vals[k]= max_fv;
         k++;
      }
      g[k] += steps[k];
   } while (k <= n);
}
```

As can be seen, the algorithm continues until all elements of the array g reach their final values. Then value of k is $n + 1$ and the main loop `do ... while` ends. The (inner) `while` loop implements the rule of succession. Although its correctness seems obvious, it can be proven rigorously by induction on n, similar to the correctness of the algorithm for generating $\widetilde{G}_m(n)$ via n nested loops.

The time complexity of the algorithm 12 is of the same type as that of Algorithm 3—they are similar and its exact estimate can be derived in the same way. The `while` loop is executed a total of $(m^n + m^{n-1} + \cdots + m^1) = (m^{n+1} - m)/(m - 1) = \Theta(m^n)$ times. For each of its executions, it performs a constant number of operations—comparisons, assignments, etc. Thus, the `while` loop performs a total of $\Theta(m^n)$ operations. In the body of the main loop `do ... while`, another $m^n = \Theta(m^n)$ operations are executed (excluding the operations in the `printn` function). Therefore, the total time complexity of Algorithm 12 is again $\Theta(m^n)$ and is therefore a CAT algorithm.

The results of the execution of Algorithm 11 and Algorithm 12, for $m = n = 3$ are the same, they are shown in Table 5.

4.3.4. Generation of $\widetilde{G}_m(n)$ via the Transition Sequence

This topic is discussed relatively rarely in the literature, especially for non-binary Gray codes. The ***transition sequence*** for an m-ary Gray code of length n is defined as *the ordered sequence of position changes*

 doi:10.20944/preprints202512.1941.v1

from one word to the next [5,19], or in a similar manner [16,20,21]. Following [20], we denoted it by $T_m(n)$ in [11], although other authors use different (often closely related) notations. Recursive definitions of $T_m(n)$ are given in [16,19–21] and others. For m-ary reflected Gray codes, the transition sequence contains both positive and negative integers and is therefore called a *mixed-sign transition sequence*. Each integer specifies which coordinate should be increased or decreased by 1, depending on whether the integer is positive or negative, respectively [11,19]. In almost all sources, the codeword coordinates are numbered from right to left, for example $\gamma = (g_n, g_{n-1}, \ldots, g_1)$. We **note** that in the mirror m-ary Gray code the coordinates are numbered from left to right, that is, $(g_1, g_2, \ldots, g_n) = \gamma^R = \varphi(\gamma)$, which is the mirror image of γ. Therefore, $G_m(n)$ and $\widetilde{G}_m(n)$ have the same transition sequence.

For example, the transition sequence of $\widetilde{G}_3(3)$ is:

$$\begin{aligned} T_3(3) =& 1,1,2,-1,-1,2,1,1,3,-1,-1,-2,1,1,-2,-1,-1,3, \\ & 1,1,2,-1,-1,2,1,1 \end{aligned}$$

—see which coordinates change in Table 5. To generate $G_3(3)$, we can use the same transition sequence, but the changes must be applied to the coordinates when they are numbered from right to left, see Table 4.

4.4. Successor and Predecessor Functions in the Mirror Gray Code

Now let us consider the **second case** of the **rule of succession**—when the correct values corresponding to the elements of g are not stored in the arrays `steps` and `final_vals`. In this situation, the task is as follows: "Given an arbitrary vector $g \in \widetilde{G}_m(n)$, represented by the array g. Compute (if it exists) the successor and predecessor of g". This task admits a natural generalization: for a given integer $k > 0$, find the next k successors of g (if they exist or until the last vector is reached) in the sequence $\widetilde{G}_m(n)$. Such a generalized task has applications to parallelizing the generation of all vectors of $\widetilde{G}_m(n)$.

We cannot directly apply the ideas in the predecessor/successor functions for lexicographic order—see Algortithm 5. We need to know which coordinates of g are increasing and which are decreasing, and then we will know their final values. Following Theorem 5, we can compute the exact values for the arrays `steps` and `final_vals` that correspond to a given vector $g \in \widetilde{G}_m(n)$ as if g had just been generated by Algorithm 12. This is the first step and its code is given below.

Algorithm 13: Computing the exact values in the arrays `steps` and `final_vals` for $g \in \widetilde{G}_m(n)$ stored in the array g

```
void compute_steps_and_final_vals (int g[], int n, int m) {
   steps [n]= 1;  final_vals [n]= m-1;
   int sum= g[n];
   for (int i= n-1; i > 0; i--) {
      if (sum & 1) {
         steps[i]= -1;  final_vals[i]= 0;
      }
      else {
         steps[i]= 1;  final_vals[i]= m-1;
      }
      sum += g[i];
  }
}
```

The second step is to compute the **successor** of a given vector $g \in \widetilde{G}_m(n)$. We can use a small part of the code of Algorithm 12, namely the `while (...)` loop, but simplified, since the values corresponding to g are already stored in the arrays `steps` and `final_vals`. They do not need to be changed for only one successor. However, if more successors are needed, then the code of Algorithm 12 can be adapted for this purpose. Instead of initialization, the function `compute_steps_and_final_vals`

 doi:10.20944/preprints202512.1941.v1

should be included, as well as a counter of generated vectors. Together with the variable `k`, they will control the main loop `do ... while`.

When computing the **predecessor** of a given vector $g \in \widetilde{G}_m(n)$, we need the initial values for each coordinate, not its final value. So, if any final value is $m-1$, the corresponding initial value is 0, and vice versa. Thus, the initial value of `g[k]` is: `m - 1 - final_vals[k]`, for $k = 1, 2, \ldots, n$. Here is the C code of these two functions.

Algorithm 14: Successor and predecessor functions in mirror m-ary reflected Gray code, for $g \in \widetilde{G}_m(n)$ stored in the array g

```
bool MGray_successor (int g[], int n, int m) {
// computes the successor of the vector in the array g
   compute_steps_and_final_vals (g, n, m);
   int k= 1;
   while (g[k] == final_vals[k])   k++;
   if (k <= n) {
      g[k] += steps[k];  // g stores the successor
      return true;
   }
   else return false;  // the last vector has no successor
}
bool MGray_predecessor (int g[], int n, int m) {
// computes the predecessor of the vector in the array g
   compute_steps_and_final_vals (g, n, m);
   int k= 1;
   while (g[k] == m - 1 - final_vals[k])   k++;
   if (k <= n) {
      g[k] -= steps[k];
      return  true;
   }
   else  return  false;
}
```

Since the function `compute_steps_and_final_vals` performs $\Theta(n)$ operations, each of the functions `MGray_successor` and `MGray_predecessor` performs a total of $\Theta(n) + O(n) = \Theta(n)$ operations.

The ranking and unranking functions are considered in the next section.

5. Transformations Between the Orderings

Here we discuss the relationship between the considered orders, which essentially means the transformations between them. For greater clarity, they are illustrated as a directed graph (digraph) in Figure 3.

Theorem 1 states that when the vectors of J_m^n are in lexicographic order, the serial number of each vector is equal to the rank of that vector, and thus they form the sequence $0, 1, 2, \ldots, m^n - 1$. The transformations between a number k in a radix-m number system and the corresponding vector α, whose coordinates are the digits of k, are based on the Horner rule and are referred to as ranking (of α) and unranking (of k). In Figure 3 they are labeled as r_1 and u_1, respectively.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 December 2025 doi:10.20944/preprints202512.1941.v1

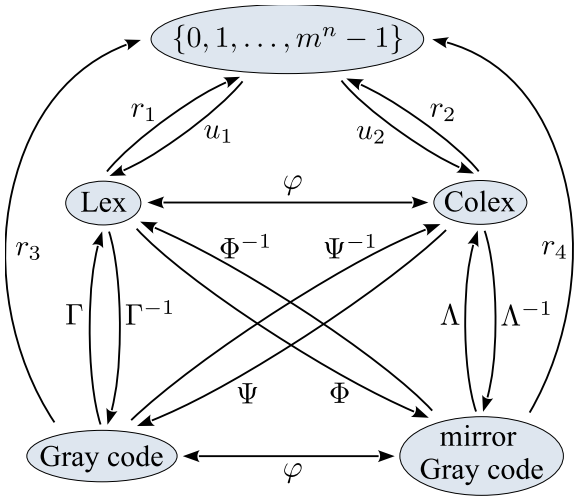

Figure 3. Transformations between the orderings of m-ary vectors, their ranking and unranking.

The basic equalities for transformations between the m-ary vectors in lexicographic order and the vectors of the m-ary reflected Gray code, are considered in [3,15,17,21–23]. In [11] we proved and used the following two statements.

Theorem 2 [11]. Let $g = (g_1, \ldots, g_n) \in G_m(n)$ be a codeword in the reflected Gray code and a_g be its sequence number. If $a_g = a_1 m^{n-1} + a_2 m^{n-2} + \cdots + a_n$, $0 \le a_i \le m-1$, $i = 1, \ldots, n$, then

$$a_i = \begin{cases} g_i, \text{ if } S_i \text{ is even,} \\ m - 1 - g_i, \text{ if } S_i \text{ is odd,} \end{cases} \tag{1}$$

where $S_1 = 0$ and $S_i = \sum_{j=1}^{i-1} g_j$, for $2 \le i \le n$.

Corollary 1 [11]. Let $a = a_1 m^{n-1} + a_2 m^{n-2} + \cdots + a_n$, $0 \le a_i \le m-1$, $i = 1, \ldots, n$, be a nonnegative integer, $0 \le a \le m^n - 1$, and $g = (g_1, g_2, \ldots, g_n)$ be the codeword in $G_m(n)$ in the a-th row according to Definition 2. Then

$$g_i = \begin{cases} a_i, \text{ if } S_i \text{ is even,} \\ m - 1 - a_i, \text{ if } S_i \text{ is odd.} \end{cases} \tag{2}$$

where $S_1 = 0$ and $S_i = \sum_{j=1}^{i-1} g_j$, for $2 \le i \le n$.

These two statements define transformations of the vectors of $G_m(n)$ in their lexicographic order and vice versa. In Figure 3 these transformations are denoted by Γ and Γ^{-1}, respectively. This is followed by ranking a vector in lexicographic order to the corresponding integer or vice versa. Thus, as a composition of two transformations: $r_1 \circ \Gamma$ and $\Gamma^{-1} \circ u_1$, the ranking and unranking functions for the m-ary reflected Gray code are obtained—see Algorithm 7 in [11].

The transformations between the lexicographic and colexicographic orders of m-ary vectors, as well as between the m-ary reflected Gray code and its mirror version, are defined by the **coordinate-reversal function** φ. As we have shown, it is an involution and is therefore drawn as a double arrow in Figure 3. Applying this function and Theorem 5 to the statements of Theorem 2 and Corollary 1 of [11], we obtain the following transformations between the colexicographic order of the m-ary vectors and the mirror m-ary reflected Gray code.

Theorem 6. *Let $g = (g_1, \ldots, g_n) \in \widetilde{G}_m(n)$. Its corresponding vector $c_g = (c_1, c_2, \ldots, c_n)$ in the colexicographic order of J_m^n is defined by:*

$$c_n = g_n \text{ and } c_i = \begin{cases} g_i, \text{ if } S_i \text{ is even,} \\ m - 1 - g_i, \text{ if } S_i \text{ is odd,} \end{cases} \tag{3}$$

 doi:10.20944/preprints202512.1941.v1

where $S_i = \sum_{j=i+1}^{n} g_j$, *for* $i = n-1, n-2, \ldots, 1$.

Corollary 1. *Let* $c = (c_1, c_2, \ldots, c_n)$ *be an arbitrary vector in the colexicographic order of* J_m^n. *Its corresponding vector* $g_c = (g_1, g_2, \ldots, g_n)$ *in* $\widetilde{G}_m(n)$ *is defined by:*

$$g_n = c_n \text{ and } g_i = \begin{cases} c_i, \text{ if } S_i \text{ is even,} \\ m-1-c_i, \text{ if } S_i \text{ is odd.} \end{cases} \tag{4}$$

where $S_i = \sum_{j=i+1}^{n} g_j$, *for* $i = n-1, n-2, \ldots, 1$.

These transformations are denoted by Λ and Λ^{-1} in Figure 3. The next step is to apply the ranking and unranking functions in colexicographic order (see Algorithm 6), based on the reversed Horner rule. They are labeled r_2 and u_2, respectively, in Figure 3. Thus we obtain the **ranking** and **unranking functions** in the mirror Gray code as compositions $r_2 \circ \Lambda$ and $\Lambda^{-1} \circ u_2$, respectively.

However, the computation of the rank can be simplified. The transformation from the mirror Gray code to colexicographic order and the computation of the rank in colexicographic order traverse the coordinates from right to left: n-th, $(n-1)$-st, ..., first. Therefore, they can be combined, as shown in Algorithm 15. An analogous case occurs for the ranking function in the Gray code—see Algorithm 1 and Algorithm 7 in [11]. In both cases, the ranking can be performed without explicitly transforming to colexicographic (or lexicographic) order, since the array c (or the array a in Algorithm 7 of [11]) can be replaced by a single variable—see and compare the last two functions in Algorithm 16. Hence, the corresponding arcs are labeled as r_4 and r_3 in the digraph in Figure 3.

For the unranking in the mirror Gray code the calculations are performed in opposite directions. Therefore, the unrank in colexicographic order is first performed, followed by a transformation from the colexicographic order to the mirror Gray code, which is the composition $\Lambda^{-1} \circ u_2$. The same applies to the unrank in lexicographic order, followed by a transformation to the Gray code, i.e., $\Gamma^{-1} \circ u_1$ (see Algorithm 7 in [11]).

Here is the code of the corresponding functions.

Algorithm 15: Transformations between mirror Gray code and colexicographic order, ranking and unranking in the mirror Gray code

```
// transformation from mirror Gray code to colex order
// and simultaneous computing of rank
int MGray_to_CoLex_and_rank (int g[], int n, int m) {
   int c[max_n];
   c[n]= g[n];
   int r= c[n];  // for computing the rank
   int sum= g[n];
   for (int i= n-1; i > 0; i--) {
      if (sum & 1) c[i]= m - 1 - g[i];
      else c[i]= g[i];
      sum += g[i];
      r= r*m + c[i];  // computing the rank
   }
   return  r; // the rank
}
// --- Unranking ---
void Colex_to_MGray (int c[], int g[], int n, int m) {
   g[n]= c[n];
   int sum= g[n];
   for (int i= n-1; i > 0 ; i--) {
      if (sum & 1) g[i]= m - 1 - c[i];
      else g[i]= c[i];
```

 doi:10.20944/preprints202512.1941.v1


```
      sum += g[i];
   }
}
void unrank_CoLex_to_MGray (int k, int n, int m, int g[]) {
   int c[max_n];
   unrank_in_colex (k, n, m, c);
   Colex_to_MGray (c, g, n, m);
}
```

Let us consider the transformations between the lexicographic order and the mirror Gray code, and vice versa, see Figure 3. The first transformation is the composition $\Lambda^{-1} \circ \varphi$, or equivalently $\varphi \circ \Gamma^{-1}$, and its inverse is $\varphi \circ \Lambda$, or equivalently $\Gamma \circ \varphi$. We note that the coordinate-reversal function φ can be implemented while performing any of the transformations Λ, Λ^{-1}, Γ, or Γ^{-1}; that is, these operations can be combined in the same manner as in the ranking function for the mirror Gray code. Consequently, each of them can be regarded as a separate transformation rather than as an explicit composition. Hence, the corresponding two arcs are added to the digraph in Figure 3 and are denoted by Φ and Φ^{-1}.

The code of these transformations is given below. We recommend that the reader compare it with the code of the corresponding functions for colexicographic order in Algorithm 15. In addition, we propose an alternative ranking function for the mirror Gray code that follows the steps of the transformation from the mirror Gray code to lexicographic order.

Algorithm 16: Transformations between lexicographic order and mirror Gray code

```
// from lexicographic order to mirror Gray code
void Lex_to_MGray (int a[], int g[], int n, int m) {
   g[n]= a[1];
   int sum= g[n];
   for (int i= n-1; i > 0 ; i--) {
      if (sum & 1) g[i]= m - 1 - a[n-i+1];
      else g[i]= a[n-i+1];
      sum += g[i];
   }
}
// from mirror Gray code to lexicographic order
void MGray_to_Lex (int g[], int a[], int n, int m) {
   a[1]= g[n];
   int sum= g[n];
   for (int i= n-1; i > 0; i--) {
      if (sum & 1) a[n-i+1]= m - 1 - g[i];
      else a[n-i+1]= g[i];
      sum += g[i];
   }
}
// computing the rank for the mirror Gray code
// by MGray_to_Lex transformation
int rank_MGray_by_Lex (int g[], int n, int m) {
   int a= g[n],  r= g[n],  sum= g[n];
   for (int i= n-1; i > 0; i--) {
      if (sum & 1) a= m - 1 - g[i];
      else a= g[i];
      sum += g[i];
      r= r*m + a;
   }
   return r; // the rank
}
```

 doi:10.20944/preprints202512.1941.v1

We omit the consideration of the transformations between the colexicographic order and the reflected Gray code, since they are analogous to the transformations just considered. For the same reasons, they are included in the digraph of Figure 3 and are denoted as Ψ and Ψ^{-1}. The code of the functions that implement them can be easily created based on the code proposed here and in Algorithm 7 in [11].

In this way, we have summarized the results of the previous sections and introduced additional relationships and transformations between the considered orders. These are illustrated in Figure 3 as a digraph that is very close to the complete symmetric digraph $\overleftrightarrow{K}_5$, with exactly two edges removed. The missing edges (should have been labeled u_3 and u_4) correspond to the unranking of a number from the set $J_{m^n} = \{0, 1, \ldots, m^n - 1\}$ into an m-ary vector of length n in Gray code and in mirror Gray code. By contrast, the opposite edges, corresponding to the ranking of a vector in these orders, should be considered doubled, since ranking can be performed in two different ways. All proposed algorithms for transforming a vector from one order to another, as well as for ranking and unranking a vector or number, have the same time complexity $\Theta(n)$. Furthermore, the fixed points in all these transformations are the palindromes.

6. Conclusions

Here we have examined in detail the lexicographic and colexicographic orderings of the vectors of J_m^n, as well as their mirror reflected Gray code; the standard reflected Gray code is considered in [11]. We have proposed various algorithms for generating the vectors of J_m^n in each of these orderings, each of which is a CAT algorithm. For each of these orderings, we have also proposed algorithms that implement the four basic functions for generating combinatorial objects, each with a time complexity of $\Theta(n)$. The connections between each pair of these orderings, as well as with the set of integers $\{0, 1, \ldots, m^n - 1\}$, have been thoroughly investigated. The transformations between them are illustrated in Figure 3 as an almost complete symmetric digraph $\overleftrightarrow{K}_5$.

We have shown that the approach of emulating the execution of nested loops is powerful enough and leads to the creation of efficient algorithms. Our experience shows that it can be successfully applied to the generation of other combinatorial objects as well.

The mirror m-ary Gray code can have the same applications as those discussed in [11], and especially in coding theory. We hope that the remaining considerations here will also find useful applications.

Some readers may prefer only the standard m-ary reflected Gray code and not accept its mirror version. We hope that we have at least offered a different perspective and stimulated further thought in this direction.

Funding: This research was partially supported by Bulgarian National Science Fund grant number KP-06-H62/2/13.12.2022.

References

1. Arndt, J. *Matters Computational: Ideas, Algorithms, Source Code*; Springer, **2011**.
2. Arndt, J. *Subset-lex: did we miss an order?* **2014**. [Accessed 30.10.2025]. Available at: https://doi.org/10.48550/arXiv.1405.6503
3. Knuth, D. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*; Addison-Wesley: *Boston, MA, USA*, **2014**.
4. Kreher, D.; Stinson, D. *Combinatorial Algorithms: Generation, Enumeration and Search*; CRC Press: *Cambridge, MA, USA*, **1999**.
5. Reingold, E.; Nievergelt, J.; Deo, N. *Combinatorial algorithms. Theory and practice*; Prentice-Hall: *New Jersey (NJ)*, **1977**.
6. Ruskey, F. Combinatorial Generation. Working Version (1j-CSC 425/ 520). In Preliminary Working Draft; *University of Victoria: Victoria, BC, Canada*, **2003**. [Accessed 23.11.2025]. Available at: http://page.math.tu-berlin.de/~felsner/SemWS17-18/Ruskey-Comb-Gen.pdf

7. Mütze, T. Combinatorial Gray Codes—An Updated Survey. *Electron. J. Comb.* **2023**, *30*. [Accessed 30.10.2025]. Available at: https://www.combinatorics.org/ojs/index.php/eljc/article/view/ds26/pdf
8. Savage, C. A Survey of Combinatorial Gray Codes, *SIAM Review*, **1997**, *39*, 605–629.
9. OEIS Foundation Inc., *The On-Line Encyclopedia of Integer Sequences. Orderings*. [Accessed 30.10.2025]. Available at: https://oeis.org/wiki/Orderings
10. Bakoev, V. Mirror (Left-recursive) Binary Gray Code, *Mathematics and Informatics*, **2023**, *66*, No. 6, 559–578.
11. Bouyuklieva, S.; Bouyukliev, I.; Bakoev, V.; Pashinska-Gadzheva, M. Generating m-ary Gray Codes and Related Algorithms, *Algorithms* **2024**, *17(7)*, 311.
12. Lipski, W. *Kombinatoryka dla Programistów (Combinatorics for Programmers)*; Wydawnictwa Naukowo-Techniczne: Warszawa, Poland, **1982**, **1989**; ISBN 83-204-1023-1. (In Polish, Russian translation—Mir, Moskva, 1988)
13. Nijenhuis, A.; Wilf, H. Combinatorial Algorithms for Computers and Calculators, (1st ed., 1975), 2nd ed., Academic Press, **1978**.
14. Sawada, J.; Williams, A.; Wong, D. Necklaces and Lyndon words in colexicographic and binary reflected Gray code order, *Journal of Discrete Algorithms*, **2017**, *46-47*, 25–35.
15. Cohn, M. Affine m-ary gray codes, *Information and Control*, **1963**, *6*, Is. 1, 70–78,
16. Suparta, I.N. Counting sequences, Gray codes and Lexicodes, Dissertation at Delft University of Technology, **2006**. Available at https://theses.eurasip.org/theses/113/counting-sequences-gray-codes-and-lexicodes/download/. Last visited: 3.06.2024.
17. Guan, D.J. Generalized Gray Codes with Applications, *Proc. Natl. Sci. Counc. ROC(A)* **1998**, *22*, 841–848.
18. Er, M.C. On Generating the N-ary Reflected Gray Codes. *I*EEE Trans. Comput. **1984**, *c*-33, 739–741.
19. Gulliver, T.A.; Bhargava; V.K.; Stein, J.M. Q-ary Gray codes and weight distributions, *Applied Mathematics and Computation* **1999**, *103*, 97–109.
20. Kapralov, S. Bounds, constructions and classification of optimal codes. Doctor Math. Sci. Dissertation, Technical University, Gabrovo, Bulgaria, **2004**. (in Bulgarian)
21. Sharma, B.D.; Khanna, R.K. On *m*-ary Gray codes. *I*nf. Sci. **1978**, *15*, 31–43.
22. Flores, I. Reflected Number Systems, *IRE Transactions on Electronic Computers*, June **1956**, *Vol. EC-5, No. 2*, 79–82.
23. Mambou, E.N.; Swart, T.G. A Construction for Balancing Non-Binary Sequences Based on Gray Code Prefixes, *IEEE Trans. Inf. Theory*, Aug. **2018**, *Vol. 64, No. 8*, 5961–5969.