МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2026
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2026
Proceedings of the Fifty-Fifth Spring Conference
of the Union of Bulgarian Mathematicians
Tryavna, Bulgaria, April 5–9, 2026

THE FOUR BASIC FUNCTIONS IN THE LEXICOGRAPHIC GENERATION OF THE SUBSETS OF A GIVEN SET

Valentin Bakoev
Faculty of Mathematics and Informatics, St. Cyril and St. Methodius University
of Veliko Tarnovo, Bulgaria
e-mail: v.bakoev@ts.uni-vt.bg

Here we apply a novel approach to generate subsets of a given set in lexicographic order and on this basis derive alternative functions for computing the successor and predecessor of a given subset. We also derive functions for ranking a given subset and unranking a given integer in a subset. We provide recursive and non-recursive implementations of these four basic functions. They run in linear time with respect to the cardinality of the given set and are easy to understand and use.

Keywords: subset, lexicographic order, characteristic vector, successor, predecessor, ranking, unranking

ЧЕТИРИТЕ ОСНОВНИ ФУНКЦИИ ПРИ ЛЕКСИКОГРАФСКО ГЕНЕРИРАНЕ НА ПОДМНОЖЕСТВАТА НА ДАДЕНО МНОЖЕСТВО

Валентин Бакоев
Факултет „Математика и информатика“, ВТУ „Св. св. Кирил и Методий“,
Велико Търново, България
e-mail: v.bakoev@ts.uni-vt.bg

Тук прилагаме нов подход за генериране на подмножествата на дадено множество в лексикографска наредба и на тази основа извеждаме алтернативни функции за изчисляване на наследника и предшественика на дадено подмножество. Също така успяхме да изведем функции за номериране на дадено подмножество и деномериране на дадено цяло число в подмножество. Предоставяме рекурсивни и нерекурсивни реализации на тези четири основни функции. Те се изпълняват в линейно време спрямо мощността на даденото множество и са лесни за разбиране и използване.

https://doi.org/10.55630/mem.2026.55.087-097

2020 Mathematics Subject Classification: 68R05, 06A05, 11B37, 68Q25.

* Partially supported by the Bulgarian National Science Fund grant number KP-06-H62/2/13.12.2022.

Ключови думи: подмножество, лексикографска наредба, характеристичен вектор, следващо подмножество, предишно подмножество, номериране, деномериране

1 Introduction

The problem of generating all subsets of a given set has important applications and therefore occupies a well-deserved place in books on combinatorial algorithms [9, 11, 8, 12, 2, 7] and others. The most popular are two orders of the generated subsets: the lexicographic order and the minimum-change (Gray code) order. Moreover, the corresponding generation algorithms may operate either byte-wise or bitwise. When subsets are generated bitwise, i.e., via their characteristic vectors, it is very natural to use the lexicographic order of these vectors. That is why S. Skiena [13, p. 454] (as well as many other authors) notes: "To generate all subsets in order, simply count from 0 to $2^n - 1$." and use the least significant n bits of the binary representation of each such integer as the characteristic vector of a subset. However, this simplest method **does not imply** a lexicographic order of the subsets themselves.

Various types of algorithms that generate all subsets of a given set in lexicographic order can be found in [9, 1, 2, 3, 14] and others. The functions for computing the successor and predecessor, as well as for ranking and unranking, play an important role in the generation of combinatorial objects [7, 12, 8, 2] and others. If such functions are created, they are always considered when generating given combinatorial objects. The four basic functions used in generating combinatorial objects and the relationships between them are illustrated in Figure 1, where c denotes the **c**urrent object and n denotes the **n**ext one.

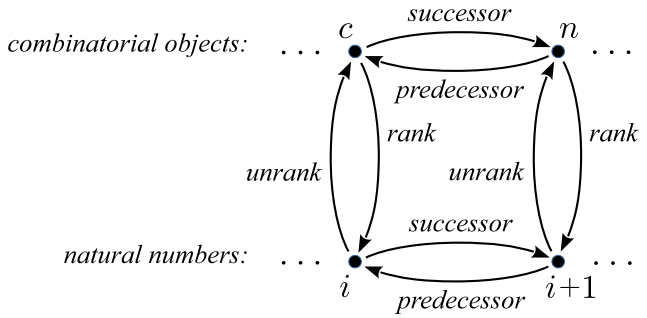

Figure 1: The four basic functions in generating combinatorial objects.

We **note** that these functions depend on the chosen order—for different orders of the same set of combinatorial objects, the corresponding functions differ from each other. Regarding their utility, Frank Ruskey notes: "One of the primary uses of ranking algorithms is that they provide *perfect hashing functions* for a class of combinatorial objects", and also: "Ranking (and unranking) is generally only possible for some of the elementary combinatorial objects" [12, p. 5].

In [4] and especially in [6] we study three basic combinatorial objects and the relationship between them according to Figure 2. We implement a new approach that, for

a given integer n, generates a sequence of serial numbers such that their binary representation, as characteristic vectors of length n, determine the lexicographic order of the subsets of a given set of n elements.

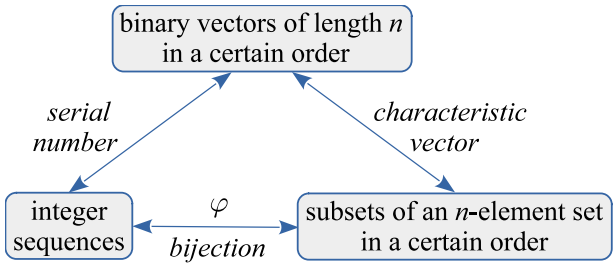

Figure 2: Basic combinatorial objects and the relationships between them.

In the monograph [6] we extended the study from [4], following the approach illustrated in Figure 1. It is written in Bulgarian, published in a small edition and therefore cannot be known to a wide audience, unlike the present work. Since the discussed problem is important and not well known, and the obtained results are new, we believe that they deserve to be presented and popularized, which is our main goal. Therefore, here we present **alternative algorithms** for the functions that compute the **successor** and **predecessor** of a given subset. We also derive **original formulas** and provide algorithms for **ranking** and **unranking** functions.

This article is organized as follows. In Section 2, we give some basic concepts and preliminary results. In Section 3, we present alternative functions for computing the subset that is the successor or predecessor of a subset defined by a given term of the integer sequence $s(n)$. The rank function (which ranks a subset given by a term of $s(n)$) and the unrank function (which unranks a natural number in a term of $s(n)$) are presented in Section 4. The last section contains some concluding remarks.

2 Basic notions and preliminary results

Here we summarize the basic notions and preliminary results from [4].

Let $\mathbb{N} = \{0, 1, 2, \dots\}$ be the set of natural numbers, and $\mathbb{N}^+ = \mathbb{N}\backslash\{0\}$. For $n \in \mathbb{N}^+$, the set of all n-dimensional binary vectors is denoted by $\{0,1\}^n = \{(x_1, x_2, \dots, x_n) | \, x_i \in \{0,1\}, \text{ for } i = 1, 2, \dots, n\}$ and is called the *n-dimensional Boolean cube* (*hypercube*). Furthermore, we define $\{0,1\}^0 = \{()\}$, where $()$ denotes the *empty vector* (with 0 coordinates). Therefore $|\{0,1\}^n| = |\{0,1\}|^n = 2^n$.

Let $\alpha = (a_1, a_2, \dots, a_n)$ and $\beta = (b_1, b_2, \dots, b_n)$ be arbitrary vectors of $\{0,1\}^n$. The natural number $\#\alpha = \sum_{i=1}^{n} a_i.2^{n-i}$ is called the *serial number* of the vector α. It is the natural number with the n-digit binary representation $a_1 a_2 \dots a_n$.

The vector α *precedes lexicographically* β, denoted by $\alpha \leq \beta$, when either $\alpha = \beta$, or $\exists i, 1 \leq i \leq n$, such that $a_i < b_i$ and $a_j = b_j$, for all $j < i$. The relation $R_{\leq} \subseteq \{0,1\}^n \times \{0,1\}^n$, called *lexicographically precedes*, is defined as $(\alpha, \beta) \in R_{\leq}$ when $\alpha \leq \beta$. Obviously, it is reflexive, transitive and *strictly antisymmetric*, that is, for any $\alpha, \beta \in$

$\{0,1\}^n$, with $\alpha \neq \beta$, either $\alpha \leq \beta$, or $\beta \leq \alpha$ holds. In this case, α and β are said to be *lexicographically comparable*. Hence $R_{\leq}$ is a *total order* in $\{0,1\}^n$.

Let $n \in \mathbb{N}^+$ and $U = \{x_1, x_2, \ldots, x_n\}$ be an universal set, whose elements are ordered according to the total order relation[1] $R_{\prec}$: $x_1 \prec x_2 \prec \cdots \prec x_n$. Let $A = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$ and $B = \{x_{j_1}, x_{j_2}, \ldots, x_{j_m}\}$ are subsets of U_n whose elements are also ordered according to $R_{\prec}$. Assume that $|A| = k \leq m = |B|$—otherwise we rename the subsets. We say that A *lexicographically precedes* B and write $A \preceq B$ when either $A = \emptyset$, or $A = B$, or there exists an integer $r, 1 \leq r \leq k$, such that $x_{i_r} \prec x_{j_r}$ and $x_{i_p} = x_{j_p}$, for all $p < r$.

Usually, $\mathcal{P}(U)$ denotes the power set of U. Obviously, the relation *lexicographically precedes* $R_{\preceq}$, defined on $\mathcal{P}(U)$, is reflexive, transitive and *strictly antisymmetric* (for every $X, Y \subseteq U$, $X \neq Y$, either $X \preceq Y$, or $Y \preceq X$ holds, and then X and Y are said to be *lexicographically comparable*). Hence $R_{\preceq}$ is a *total order* in $\mathcal{P}(U)$.

Definition 1. Let $U = \{x_1, x_2, \ldots, x_n\}$ be a given set, $n \in \mathbb{N}^+$, and $X \subseteq U_n$. The vector $\alpha = (a_1, a_2, \ldots, a_n) \in \{0,1\}^n$ defined as:

$$a_i = \begin{cases} 0, \text{ if } x_i \notin X, \\ 1, \text{ if } x_i \in X, \end{cases}$$

for $i = 1, 2, \ldots, n$, is called the *characteristic vector* of the subset X.

Theorem 2. *Let U be a given n-element set, $n \in \mathbb{N}$. The function $f : \mathcal{P}(U) \to \{0,1\}^n$ is defined as $f(X) = \alpha$, where $\alpha \in \{0,1\}^n$ is the characteristic vector of X, for any $X \in \mathcal{P}(U)$. Then f is a bijection.*

For a given integer $n \in \mathbb{N}$, the notation $U_n = \{x_n, x_{n-1}, \ldots, x_2, x_1\}$, where $x_n \prec x_{n-1} \prec \cdots \prec x_2 \prec x_1$, is more convenient further on. Thus $U_0 = \emptyset$ and $U_n = \{x_n\} \cup U_{n-1}$.

The *sequence $p(n)$ of all subsets of U_n*, which is an **ordering** of the elements of $\mathcal{P}(U_n)$, is defined recursively as follows:

$$p(n) = \begin{cases} \emptyset, \text{ if } n = 0, \\ \emptyset, (\{x_n\} \cup p(n-1)), \widetilde{p}(n-1), \quad \text{if } n > 0, \end{cases} \tag{1}$$

where $\{x_n\} \cup p(n-1)$ means that the new element x_n for U_n is appended to each subset of $p(n-1)$, and $\widetilde{p}(n-1)$ means $p(n-1)$ without the empty set, i.e., $p(n-1) \backslash \emptyset$.

Theorem 3. *Let $n \in \mathbb{N}$ and $U_n = \{x_n, x_{n-1}, \ldots, x_2, x_1\}$ be a given set and $R_{\prec}$ be a total order relation such that $x_n \prec x_{n-1} \prec \cdots \prec x_2 \prec x_1$. Let the sequence $p(n)$ of all subsets of U_n be obtained according to formula (1). Then the subsets of $p(n)$ are in lexicographic order.*

In [4], recursive definitions are also given for: (a) the *sequence $c(n)$ of characteristic vectors* corresponding to the subsets of $p(n)$, and (b) the *sequence $s(n)$ of serial numbers* corresponding to the characteristic vectors of $c(n)$. Thus:

$$c(n) = \begin{cases} (), \text{ if } n = 0, \\ \mathbf{0}^n, (1, c(n-1)), (0, \widetilde{c}(n-1)), \quad \text{if } n > 0, \end{cases} \tag{2}$$

where $\mathbf{0}^n$ denotes the zero vector with n coordinates, $(1, c(n-1))$ denotes all vectors of $c(n-1)$ prefixed by 1, and $(0, \widetilde{c}(n-1))$ denotes all vectors of $c(n-1)$ except the vector

[1] Such as alphabetical order, index order, ASCII, Unicode, etc.

$\mathbf{0}^{n-1}$ prefixed by 0. And for the sequence $s(n)$:

$$s(n) = \begin{cases} 0, \text{ if } n = 0, \\ 0, (2^{n-1} + s(n-1)), \widetilde{s}(n-1), \text{ if } n > 0, \end{cases} \tag{3}$$

where $2^{n-1} + s(n-1)$ means that the integer 2^{n-1} is added to each term of $s(n-1)$ and $\widetilde{s}(n-1)$ means $s(n-1)$ without the term 0. Therefore:

Theorem 4. *The thus defined sequences $c(n)$ and $s(n)$ determine an ordering of the characteristic vectors and their serial numbers such that the corresponding subsets of U_n (resp. of $p(n)$) are in lexicographic order.*

Example 5. Let $U_3 = \{x, y, z\}$. Following the given definitions, we construct the corresponding subsets and sequences as follows:

- $U_0 = \emptyset$. Then $p(0) = \emptyset$; $c(0) = ()$, and $s(0) = 0$.
- $U_1 = \{z\}$. Then $p(1) = \emptyset, \{z\}$; $c(1) = (0), (1)$, and $s(1) = 0, 1$.
- $U_2 = \{y, z\}$. Then $p(2) = \emptyset, \{y\}, \{y, z\}, \{z\}$; $c(2) = (0, 0), (1, 0), (1, 1), (0, 1)$, and $s(2) = 0, 2, 3, 1$.
- $U_3 = \{x, y, z\}$. Then $p(3) = \emptyset, \{x\}, \{x, y\}, \{x, y, z\}, \{x, z\}, \{y\}, \{y, z\}, \{z\}$; $c(3) = (0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 0, 1), (0, 1, 0), (0, 1, 1), (0, 0, 1)$, and $s(3) = 0, 4, 6, 7, 5, 2, 3, 1$.

Based on these definitions and statements, the following algorithm was created. It sequentially generates the sequences $s(1), s(2), \ldots, s(n)$ for a given input n and implements formula (3) by performing **two main steps**. The first step is called **copy**—it copies the terms of $\widetilde{s}(i-1)$ into the same array `s` after the element with index `len`, which is the length of $s(i-1) = 2^{n-1}$. The second step is called **increase and shift**—it increases all terms of $s(i-1)$ by 2^{i-1} and shifts them to the right by 1 position in the array `s`. In doing so, `s[0]` retains its value 0. Here is its code in the C programming language.

Algorithm 1: Computes the sequence $s(n)$ for a given input $n > 0$

```
const int num = 26; // max. |U| - according to the English alphabet
const int max_len = 1 << num;  // max. |P(U)|=2^{26}
int s[max_len]; // to store s(n)
// ...
void Gen_Subsets_Lex (int n) { // 0 < n <= num
   s[0]= 0;  s[1]= 1;  // initialization: s(1)
   int len= 2;  // current length
   for (int i= 2; i <= n; i++) { // main loop - generation of s(i)
      for (int j= 1; j < len; j++) { // I step: copy s(i-1)
         s[j + len]= s[j];
      }
      for (int j= len; j > 0; j--) { // II step: increase the
         s[j]= s[j-1] + len;         // terms of s(i-1) by 2^{i-1}
      }                              // and shift them right by 1
      len *= 2;
   }
}
```

The results of the algorithm execution for $n \leq 5$ are shown in Table 1 in the form of an irregular triangle $T(n, k)$.

Table 1: The sequences $s(n)$, computed by Algorithm 1, for $n = 0, 1, \ldots, 5$.

n	$T(n,k)$, for $k =$ 0, 1, 2, 3, 4, 5, 6, 7, ...
0	0
1	0, 1
2	0, 2, 3, 1
3	0, 4, 6, 7, 5, 2, 3, 1
4	0, 8, 12, 14, 15, 13, 10, 11, 9, 4, 6, 7, 5, 2, 3, 1
5	0, 16, 24, 28, 30, 31, 29, 26, 27, 25, 20, 22, 23, 21, 18, 19, 17, 8, 12, 14, 15, 13, 10, 11, 9, 4, 6, 7, 5, 2, 3, 1

The time complexity and space complexity of the Algorithm 1 are of the type $\Theta(2^n)$, which is exponential with respect to the size of the input, but linear with respect to the size of the output. So it is a *Constant Amortized Time* (*CAT*) algorithm [12]. All of this was used in the creation of the sequence A356120 in the OEIS [10].

3 Successor and predecessor functions

Algorithms for computing the successor and predecessor of a given subset are discussed in [9, 14, 2, 3, 5]. Starting from a given subset $A \subseteq U_n$ and using them, one can generate all or some of all subsets of a given set U_n in lexicographic order.

Once the sequence $s(n)$ is generated, it is trivial to compute the successor or predecessor of its kth term, as well as to rank this term or unrank the corresponding integer. So we will derive the four basic functions **in the case when** $s(n)$ **is not available**, but using its properties presented above. Here we mean the successor/predecessor of a given term of $s(n)$. If we need a true subset, we need to start with a given subset and go through its characteristic vector, its serial number (which is a term of $s(n)$), its successor/predecessor, and back to its corresponding subset.

3.1 Successor function

Given integers n and t, where t is a **t**erm of $s(n)$, i.e., $0 \le t < 2^n$. We want to derive a recurrence formula for the **successor** (or next term) of t in the sequence $s(n)$. So, if $t = 1$, it is the last term of $s(n)$ and has no successor. Otherwise, we reason as follows:

1) If $t < 2^{n-1}$, then t is the result of the first step (copy) of Algorithm 1 and hence its successor is the same as for the sequence $s(n-1)$.

2) If $2^{n-1} \le t < 2^n$, then t is the result of the second step of the algorithm. To obtain its successor, we get the successor of $t - 2^{n-1}$ for the sequence $s(n-1)$ and then increase it by 2^{n-1}.

It remains to consider the boundary conditions. When on the ith iteration of the main loop: (1) $t = 0$ occurs, then t it is followed by 2^{i-1}—see Table 1. And (2) $t = 2^{i-1} + 1$ occurs (concerns the terms where the two subsequences are concatenated), the next term

is 2^{i-2}. Thus we obtain the formula:

$$succ(n,t) = \begin{cases} 2^{n-1}, \text{ if } t = 0, \\ 2^{n-2}, \text{ if } t = 2^{n-1}+1, \\ succ(n-1,t), \text{ if } t < 2^{n-1}, \\ 2^{n-1} + succ(n-1, t-2^{n-1}), \text{ if } t \geq 2^{n-1}. \end{cases} \tag{4}$$

Equality (4) implies the correctness of the following algorithm, which computes the successor of t.

Algorithm 2: Recurfive function for computing the successor of t in the sequence $s(n)$

```
int successor_rec (int n, int t) { // t >= 1 in the first call
   int hlen= 1 << (n - 1);  // = 2^{n-1} = half the length of s(n)
   if (0 == t)  return  hlen;
   if (t == hlen + 1)  return  hlen >> 1; // i.e. hlen/2;
   if (t < hlen)  return  successor_rec (n - 1, t);
   else  return  hlen + successor_rec (n - 1, t - hlen);
}
```

The time complexity of this algorithm is $O(n)$, since the value of n decreases by 1 in each recursive call, and the value of t decreases by 2^{n-1} when $2^{n-1} \leq t < 2^n$. Thus, the value of t is remains correct, i.e., in the interval $[0, 2^{n-1}]$. We note that a small optimization is possible if we pass `hlen` as a parameter, instead of `n`.

Here is the code of the non-recursive version of the same function. It again implements formula (4) with the same time complexity $O(n)$.

Algorithm 3: Function for computing the successor of t in $s(n)$

```
int successor (int n, int t) {
   int hlen= 1 << (n-1);  int succ= 0;
   while (t >= 0) {
      if (0 == t) { succ += hlen;  break; }
      if (t >= hlen) {
         if (t == hlen+1) { succ += hlen >> 1;  break; }
         else { succ += hlen;  t -= hlen; }
      }
      hlen >>= 1;
   }
   return succ;
}
```

3.2 Predecessor function

Now, for given integers $n \in \mathbb{N}^+$ and t, $0 \leq t < 2^n$, we want to derive a recurrence formula for the **predecessor** (or previous term) of t in the sequence $s(n)$. If $t = 0$, this is the first term of $s(n)$ and has no predecessor. Otherwise, we reason analogously:

1) If $t < 2^{n-1}$, then t is the result of the first step of the Algorithm 1 and therefore its predecessor is the same as in the sequence $s(n-1)$.

2) If $t \geq 2^{n-1}$, then t is the result of the second step of the algorithm. Therefore, its predecessor is like that in the sequence $s(n-1)$, i.e., that of $t - 2^{n-1}$, but increased by 2^{n-1}.

Boundary conditions—when on the i iteration of the main loop: (1) $t = 2^{i-1}$ occurs, then its predecessor is 0, see Table 1. And (2) $t = 2^{i-2}$ occurs (concerns the terms where the two subsequences are concatenated), then its predecessor is $2^{i-1}+1$. Thus we obtain a formula analogous to (4), the difference being in the boundary conditions.

$$pred\,(n,t) = \begin{cases} 0, \text{ if } t = 2^{n-1}, \\ 2^{n-1}+1, \text{ if } t = 2^{n-2}, \\ pred\,(n-1,t), \text{ if } t < 2^{n-1}, \\ 2^{n-1} + pred\,(n-1, t-2^{n-1}), \text{ if } t > 2^{n-1}, \end{cases} \tag{5}$$

Here is the code of a recursive function that implements formula (5) for given parameters $n \in \mathbb{N}^+$ and $t, 0 < t < 2^n$. We present the slightly optimized version (as noted above) that uses the parameter `hlen` instead of `n` and therefore, on the first call, the value of the argument corresponding to `hlen` should be 2^{n-1}.

Algorithm 4: Recursive function for computing the predecessor of t in $s\,(n)$

```
int predecessor_rec (int hlen, int t) {
   if (t == hlen)  return  0;
   if (t == hlen/2)  return  hlen + 1;
   if (t < hlen)  return  predecessor_rec (hlen >> 1, t);
   else  return  hlen + predecessor_rec (hlen >> 1, t - hlen);
}
```

Formula (5) implies the correctness of this algorithm. Its time complexity is $O(n)$ with the same arguments as for the Algorithm 2. And here is the non-recursive version.

Algorithm 5: Function for computing the predecessor of t in $s\,(n)$

```
int predecessor (int n, int t) {
   int hlen= 1 << (n - 1);
   int p= 0; // predecessor
   while (t > 0) {
      if (t == hlen/2) { p += hlen + 1;  break; }
      if (t >= hlen) {
         if (t == hlen)  break;
         else { p += hlen;  t -= hlen; }
      }
      hlen >>= 1;
   }
   return p;
}
```

4 Ranking and unranking functions

After an extensive search, we found only one source [2], where ranking and unranking functions are mentioned, but referring to reverse lexicographic order of subsets. There are no algorithms, explanations, correctness, etc., just their code (in the C++ language) is presented in less than a page. Both functions are related to the sequence A108918 in the OEIS, called "Reversed binary words in reversed lexicographic order" [10].

4.1 Ranking function

The *ordinal* (or *sequential*) *number* of a subset is the position of that subset in the sequence $p(n)$—it is the same as the position of its characteristic vector in $c(n)$, or the position of its serial number in $s(n)$. The ordinal number is an integer, which is the result of the corresponding *ranking function* applied to it. **Note** that it changes depending on the selected order of the subsets, while the serial number of the corresponding characteristic vector is unique, like the serial number of a device.

Let $n \in \mathbb{N}^+$ and $t, 0 \leq t < 2^n$, be an arbitrary term of $s(n)$. We want to compute the ordinal number of t in $s(n)$, which is its rank, denoted by r, or more precisely by $r(n,t)$. So, we are looking for r, $0 \leq r < 2^n$, such that $s(n,r) = t$. Our reasons are analogous to the previous ones: if $t = 0$, then $r = 0$, otherwise there are two possibilities for t:

1) If $t < 2^{n-1}$, then t is the result of the first step (copy) of Algorithm 1. Then its rank in the sequence $s(n-1)$ is $r - 2^{n-1}$, and then $s(n,r) = t = s(n-1, r-2^{n-1})$. Hence $r(n,t) = 2^{n-1}$ plus the rank of t in $s(n-1)$, which is $r(n,t) = 2^{n-1} + r(n-1,t)$.

2) If $t \geq 2^{n-1}$, then t is the result of the second step of the algorithm. Hence the integer 2^{n-1} is added to the element with rank $(r-1)$ in $s(n-1)$, and thus the result is t. Then $s(n,r) = t = s(n-1, r-1) + 2^{n-1}$, whence $r(n,t)$ is 1 plus the rank of $(t - 2^{n-1})$ in $s(n-1)$, which is $r(n,t) = 1 + r(n-1, t-2^{n-1})$.

Therefore, the recursive definition of r for given parameters n and t is:

$$r(n,t) = \begin{cases} 0, \text{ if } t = 0, \\ r(n-1,t) + 2^{n-1}, \text{ if } t < 2^{n-1}, \\ 1 + r(n-1, t-2^{n-1}), \text{ if } t \geq 2^{n-1}. \end{cases} \tag{6}$$

Here is the code of the ranking function that implements formula (6) and therefore this function is correct.

Algorithm 6: Recursive ranking function for the sequence $s(n)$

```
int rank_term_rec (int n, int t) {
    if (0 == t)  return  0;
    int hlen = 1 << (n-1);
    if (t < hlen)  return  hlen + rank_term_rec (n - 1, t);
    else  return  1 + rank_term_rec (n - 1, t - hlen);
}
```

The time complexity of Algorithm 6 is again $O(n)$ for the same reasons as the previous functions. The small optimization by passing `hlen` as a parameter instead of `n` is obvious. And here is the C code of the non-recursive ranking function.

Algorithm 7: Ranking function for the sequence $s(n)$

```
int rank_term (int n, int t) {
    int hlen= 1 << (n-1);  int r= 0;
    while (t) {
        if (t < hlen)  r += hlen;
        else { r++;  t -= hlen; }
        hlen >>= 1;
    }
    return r;
}
```

4.2 Unranking function

Now we need to solve the inverse problem of the ranking problem: for given integers n and r, $0 \le r < 2^n$, find the element t of $s(n)$ with rank r, i.e., such that $s(n,r) = t$. The reasons are analogous to those for deriving formula (6). If $r = 0$, then $t = 0$, otherwise:

1) If $r \le 2^{n-1}$, then t is the result of the second step of Algorithm 1, i.e., the integer 2^{n-1} is added to the element with rank $(r-1)$ in $s(n-1)$. Therefore, $s(n,r) = t = 2^{n-1} + s(n-1, r-1)$.

2) If $r > 2^{n-1}$, then t is simply copied from $s(n-1)$ to position r after executing the first step of the algorithm. Hence the rank of t in $s(n-1)$ is $r - 2^{n-1}$, i.e., $s(n,r) = t = s(n-1, r-2^{n-1})$.

So, the recursive definition of t as a function of the parameters n and r is:

$$t(n,r) = \begin{cases} 0, \text{ if } r = 0, \\ t(n-1, r-1) + 2^{n-1}, & \text{if } r \le 2^{n-1}, \\ t(n-1, r-2^{n-1}), & \text{if } r > 2^{n-1}. \end{cases} \tag{7}$$

The following two algorithms implement formula (7), which determines their correctness. Here is the C code of a recursive and a non-recursive function that unranks the integer r in a term t of $s(n)$. On the first call of the recursive function, the value of the argument corresponding to `hlen` must be 2^{n-1}.

Algorithm 8: Recursive unranking function for the sequence $s(n)$

```
int unrank_rec (int hlen, int r) {
   if (r <= hlen) {
      if (0 == r)  return 0;
      else  return hlen + unrank_rec (hlen/2,r - 1);
   }
   else  return unrank_rec (hlen/2, r - hlen);
}
```

Algorithm 9: Unranking function for the sequence $s(n)$

```
int unrank (int n, int r) {
   int hlen= 1 << (n-1);
   int t = 0;
   while (r) {
      if (r <= hlen) { t += hlen;  r--; }
      else  r -= hlen;
      hlen >>= 1;
   }
   return t;
}
```

Obviously, the time complexity the last two algorithms is $O(n)$.

5 Conclusions

Here we have considered a different approach to generating subsets of a given set in lexicographic order, which has advantages and disadvantages compared to the usual approach [6]. This approach presents a new perspective on such generation and helped us to

derive alternative successor and predecessor functions, as well as to derive original ranking and unranking functions for the lexicographic order. These four basic functions run in linear time $O(n)$ and are easy to understand. In addition to the various applications (e.g. those in [14]), ranking and unranking functions can be useful in compact representation of subsets, in generating random subsets, etc. We now consider the next step—using the presented ranking and unranking functions to derive other similar functions that operate directly on the subsets themselves, without going through their characteristic vectors.

The proposed approach can be applied to the generation of other combinatorial objects and to the derivation of their four basic functions.

References

[1] A. Aho, J. Hopcroft, J. Ullman, The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Company, 1974.

[2] J. Arndt, MATTERS COMPUTATIONAL: Ideas, Algorithms, Source Code, Springer, 2011.

[3] J. Arndt, Subset-lex: did we miss an order? https://arxiv.org/abs/1405.6503. Last accessed 12 Feb. 2026.

[4] V. Bakoev, An Algorithm for Generating All Subsets in Lexicographic Order, 2022 International Conference Automatics and Informatics (ICAI), Varna, Bulgaria, (2022), 271–275.

[5] V. Bakoev, Generating sets in lexicographic order and with minimal change, St. Cyril and St. Methodius Univ. Publ. House, Veliko Tarnovo, 2023. (monograph, in Bulgarian)

[6] V. Bakoev, Binary vectors: orderings, integer sequences, and generating of sets, St. Cyril and St. Methodius Univ. Publ. House, Veliko Tarnovo, 2023. (monograph, in Bulgarian)

[7] D. Knuth, The art of computer programming, Volume 4A: Combinatorial Algorithms, Part 1. Addison-Wesley, 2011.

[8] D. Kreher, D. Stinson, Combinatorial algorithms: generation, enumeration and search, CRC Press, 1999.

[9] A. Nijenhuis, H. Wilf, Combinatorial Algorithms for Computers and Calculators, 2nd ed., Academic Press, 1978.

[10] OEIS Foundation Inc., *The On-line Encyclopedia of Integer Sequences*. https://oeis.org/. Last accessed 12 Feb. 2026

[11] E. Reingold, J. Nievergelt, N. Deo, Combinatorial algorithms, Theory and practice. New Jersey, Prentice-Hall, 1977.

[12] F. Ruskey, Combinatorial Generation. Working Version (1j-CSC 425/520), 2003. Available at http://page.math.tu-berlin.de/∼felsner/ SemWS17-18/Ruskey-Comb-Gen.pdf. Last accessed 12 Feb. 2026

[13] S. Skiena, The Algorithm Design Manual. 2nd ed., Springer-Verlag London Limited, 2008.

[14] I. Stojmenović, M. Miyakawa, Applications of a subset generating algorithm to base enumeration, knapsack and minimal covering problems. *The Computer Journal*, Vol. 31, No. 1, (1988), 65–70.