SERIAL AND UNSERIAL COMBINATORIAL FAMILY

GLAUCIO G. DE M. MELO AND EMERSON A. DE O. LIMA

Abstract. This article presents the *Serial and Unserial Methods* (SUM). The algorithms showed here is strongly related to the first part of a classical reference in combinatorics, the *Combinatorial Algorithms for computers and calculators*, from Albert Nijenhuis and Herbert Wilf. The *Serial Method* proposal is to obtain the output of an specific kind of combinatorial family from its position on the list of all combinatorial possibilities. The *Unserial Method* is the inverted step of Serial Method, getting the serial number from the combinatorial family given as input. The *serial number* is denoted here as the position of the combinatorial family on the list.

Introduction

In your classical reference of combinatorial algorithms[1], A. Nijenhuis and H. Wilf presents the *Next* algorithms, and the proposal of this methods is to obtain the next output of a combinatorial family from the actual one. Getting an example from Nijenhuis-Wilf's reference, the *Next Permutation*[1] algorithm generates de whole list of permutations without considering the list's position of each permutation contained in the list. This is a fast way to obtain all the permutations or a set of permutations from an specific one. This work showed here is a set of algorithms which are concentrated to get an specific combinatorial family on the list. The Serial and Unserial combinatorial family showed in this article is the permutation, composition, partition of an n-set, k-subsets and subsets. Each kind of combinatorial family exposed in this article is divided on the sections below.

Serial Permutation Method (SPM)

Basic Concepts. The permutation algorithms are classified in two main groups: the one that creates a set of permutations from the identity permutation and the other that produces a set of permutations by means of simple changes between the vector's elements, creating a new permutation from the actual one. The algorithm *Next Permutation for N letters* [1] belongs to the second group, creating a complete set of permutations through successive algorithm invocation, getting the next permutation until reaches the last vector of the list. The algorithm *Next Permutation* is powerful creating the next permutation vector using only local information. This guarantees that the next vector will be different from anyone else that was defined before.

There is a specific problem on this creating process:

> *Is it possible to get directly a permutation vector located on a specific position on the list of vectors, excluding the alternative of the Next*

Key words and phrases. Combinatorial Algorithms, Complexity, Combinatorial Optimization, Combinatorial Family.

Permutation algorithm's successive execution until gets the desired vector?

The *Serial Permutation Method* (SPM) answers this question, being able to process the same list of permutation that the *Next Permutation* algorithm does, with a difference: the SPM only needs the serial number and the permutation vector's size to process the desired output. It is also possible to invert this process from the SPM, getting the serial number through the permutation vector.

SPM Construction. The SPM was developed from the observation of the *Next Permutation's* data outputs. This algorithm has an auxiliary variable called offset vector[1], which is defined below.

Consider p the permutation vector with n elements and let d be the offset vector with $n-1$ elements, the d vector is defined by[1]:

$$d_i = |j : j \leq i,\ p_j > p_{i+1}|\ i = 1, 2, ..., n \tag{1}$$

That is explained:

How many numbers are there biggest than p_{i+1} between the beginning of the permutation vector and the i index of the d vector?

Being $n!$ the complete list of the created vectors, we have in Table 1 an example of output from the p permutation vector related to its serial, for $n = 5$ and the $serial = 32$.

Table 1. Input and output processing that are common in the *Next Permutation* algorithm.

Input	Output
$n = 5$	$p = (3, 5, 1, 2, 4)$
$Serial = 32$	$d = (0, 2, 2, 1)$

The SPM is subdivided in two steps, and the data input of the second step corresponds to the data output of the first step:

(1) Given a serial number and the size of the permutation vector, determine the offset vector;
(2) Given the offset vector, determine the permutation vector.

Algorithm to Attain the Offset Vector. To solve the first SPM step it is necessary an algorithm to get the offset vector, which is called *Serial Offset Algorithm* (SOA). The algorithm does a calculus that reflects the pattern of the offset vectors' creation throughout the whole list of permutation, using only local information. In Table 2, we can visualize the pattern of the offset vector in the whole list of permutation, where the detached column will serve as a guide to the determination that the SOA decodes. Let d be the offset vector with n elements. Each d vector's column showed in Table 2 has the following properties:

- Let k be the value of the current column ($k = 1, 2, ..., n$), the elements of the d_k column ($0 \leq d_k \leq k$) repeat $k!$ times until the element of the column gets the k value. This can be visualized on the first part of the detached column in Table 2, illustrated in blue, which the number of detached elements in blue or red is defined by $(k+1)!$

TABLE 2. Full list of permutations, with $n = 4$

Serial	p	d
1	(1,2,3,4)	(0,0,0)
2	(2,1,3,4)	(1,0,0)
3	(3,1,2,4)	(1,1,0)
4	(1,3,2,4)	(0,1,0)
5	(2,3,1,4)	(0,2,0)
6	(3,2,1,4)	(1,2,0)
7	(4,2,1,3)	(1,2,1)
8	(2,4,1,3)	(0,2,1)
9	(1,4,2,3)	(0,1,1)
10	(4,1,2,3)	(1,1,1)
11	(2,1,4,3)	(1,0,1)
12	(1,2,4,3)	(0,0,1)
13	(1,3,4,2)	(0,0,2)
14	(3,1,4,2)	(1,0,2)
15	(4,1,3,2)	(1,1,2)
16	(1,4,3,2)	(0,1,2)
17	(3,4,1,2)	(0,2,2)
18	(4,3,1,2)	(1,2,2)
19	(4,3,2,1)	(1,2,3)
20	(3,4,2,1)	(0,2,3)
21	(2,4,3,1)	(0,1,3)
22	(4,2,3,1)	(1,1,3)
23	(3,2,4,1)	(1,0,3)
24	(2,3,4,1)	(0,0,3)

- When $d_k = k$ in $k!$ times, the following elements from the list are put on the inverted form, like showed in red in Table 2. The direct and inverted list intercalate themselves until complete all the positions of the column.

Exists throughout each offset vector column an intercalation between the two kinds of lists (direct and inverted), we can consider this fact as an element of parity in the list, considering as an even the list in its direct form and as odd the list in its inverted form. To determinate the relation between the serial number and the parity of the list, follows:

$$f = \left\lfloor \frac{s-1}{(k+1)!} \right\rfloor \; mod \; 2 \tag{2}$$

Where:

- f: Determines if the list is direct ($f = 0$) or inverted ($f = 1$);
- s: Serial number;
- k: Index of the offset vector;
- $\lfloor x \rfloor$: Floor function. It returns the biggest integer value smaller than x;
- *mod*: An operation that returns a division's rest.

If the list had been direct, we attribute to the elements of the offset vector:

$$d_k = \left\lfloor \frac{[(s-1)\ mod\ (k+1)!]}{k!} \right\rfloor \tag{3}$$

If the list had been inverted, we attribute the complement which would be the direct list:

$$d_k = k - \left\lfloor \frac{[(s-1)\ mod\ (k+1)!]}{k!} \right\rfloor \tag{4}$$

Each attribution is made through a loop that go through the offset vector. The equations 3 and 4 can be joined in a unique form, but is a inefficient way to compute the equation:

$$d_k = f.k + (-1)^f.\left\lfloor \frac{[(s-1)\ mod\ (k+1)!]}{k!} \right\rfloor \tag{5}$$

Where f is defined on equation 2.

Serial Offset Algorithm (SOA)
Routine Specifications:

- n: Size of the offset vector;
- i: Index of the offset vector;
- d: Offset vector, alternating its indices from $0..n-1$;
- s: Permutation's serial.

Routine:

```
For $i \leftarrow 1$ to $n$ do
    If $\lfloor(s-1)\ /\ (i+1)!\rfloor\ mod\ 2 = 1$
        $d_{i-1} \leftarrow i - \lfloor((s-1)\ mod\ (i+1)!)\ /\ i!)\rfloor$
    Else
        $d_{i-1} \leftarrow \lfloor((s-1)\ mod\ (i+1)!)\ /\ i!)\rfloor$
End For
return $d$.
```

Algorithm to attain the permutation vector from the offset vector. After the SOA is computed, the SPM is concluded with the *Permutation Algorithm by Offset* (PAO). The PAO does the SPM's second step, returning the desired output. A relevant topic for the PAO construction is to find the decoding process of the permutation vector, with only the offset vector being the input. We know that the offset vector maps the elements of the permutation vector. In the decoding, we have:

Let p be the permutation vector and d the offset vector:

$$p = (p_1, p_2, ..., p_n) \quad d = (d_1, d_2, ..., d_{n-1}) \tag{6}$$

And d already has its values determining by the SOA. This is how the decoding process is made: we know that d_1 has its values discretely included between 0 and 1. Obviously, 0 and 1 are the unique possible elements for d_1. From this information, we can conclude:

$$d_1 = \begin{cases} 0, & \text{if } p_1 < p_2 \\ 1, & \text{if } p_2 < p_1 \end{cases}$$

We do not consider the possibility of equality between the elements of the permutation vector, because we know that there is no repeated elements on the vector, and if we organize it, the difference between them will be only one unit. Like d_1, the element d_2 has its values included between 0 and 2. In this and in other cases, we analyze the current inequality from the inequality that was created previously. For d_2, we have:

$$d_1 = \begin{cases} 0, \quad d_2 = \begin{cases} 0, & \text{if } p_1 < p_2 < p_3 \\ 1, & \text{if } p_1 < p_3 < p_2 \\ 2, & \text{if } p_3 < p_1 < p_2 \end{cases} \\ \\ 1, \quad d_2 = \begin{cases} 0, & \text{if } p_2 < p_1 < p_3 \\ 1, & \text{if } p_2 < p_3 < p_1 \\ 2, & \text{if } p_3 < p_2 < p_1 \end{cases} \end{cases}$$

We can conclude that as the value of d_2 increases, p_3 "slides" on the left through the inequality. In general, we have:

$$d_k \in \{0, 1, ..., k\} \quad \{p_i < p_j < ... < p_k < ... < p_t\} \quad i, j, t \neq k \tag{7}$$

With p_k in the inequality with d_k positions, counted from right to left, because the order of the elements is ascendent. Finished the offset vector's raster, we will get a set of inequalities that informs the order of the permutation elements. We have, for example:

Given a $d = (0, 2, 2)$ offset vector, the inequality for a p permutation vector is:

$$p_3 < p_4 < p_1 < p_2 \tag{8}$$

Being the last inequality at 8 the final disposition between the elements of the vector. After the mapping of the permutation's elements was done, we can say that each element of the vector ordered in 8 corresponds to the elements of the identity permutation. This fact classifies the SPM in the first group described in the basic concepts of this section. On the 8 inequality, we have:

$$(p_3 = 1) < (p_4 = 2) < (p_1 = 3) < (p_2 = 4) \tag{9}$$

The next step to get the final output is to arrange each one of the elements ordered on its own positions. As the p vector is ordered like $(p_1, p_2, ..., p_n)$, we have:

$$p = (p_1, p_2, p_3, p_4) \quad \Rightarrow \quad p = (3, 4, 1, 2) \tag{10}$$

So we can get the SPM's final output. In implementation terms, to compute the input built from the permutation vector (from right to left) the corresponding indices of the vector were used in relation to its complement. We will notice at the implementation that in the insertion moment of the current element, if it replaces another, the elements of inequality located on the left will "slide" to the left side, allocating space for the current element. The "slides to the left" operation is implemented on the subroutine *push*.

Permutation Algorithm by Offset (PAO)
Routine Specifications:

- n: Size of the permutation vector;
- i, j: Indices of the algorithm's vectors;
- p: Permutation vector, alternating its elements on $0..n-1$;
- r: Vector which will keep the element's position before being ordered.

Routine:

```
  $r_{n-1} \leftarrow 1$
  For $i \leftarrow 1$ to $n-1$ do
    If $r_{n-1-d_{i-1}} = 0$
      $r_{n-1-d_{i-1}} \leftarrow i+1$
    Else
      $r_{n-1-d_{i-1}} \leftarrow push$
  End For
  For $i \leftarrow 0$ to $n-1$ do
    $p_{r_i-1} \leftarrow i+1$
  End For
return $p$.
```

Subroutine Push

```
  For $j \leftarrow (n-1)-i$ to $j < (n-1)-d_{i-1}$ do
    $p_j \leftarrow p_{j+1}$
  End For
return $i+1$.
```

The Inverted Process of SPM (Unserial Method)

We can describe now the inverted process of the SPM's. We have the permutation vector as the input, and the desired output is the correspondent serial number. The ingenuous process to get the serial number for the permutation vector is the raster of the permutation list, comparing the vectors one by one, until gets the equivalent vector computed on the input, being the returned value the loop's index that does this raster. However, we can find the serial value inverting the SPM's steps:

(1) Given the permutation vector, find the offset vector;
(2) Given the offset vector, find the serial number.

Like the SPM, the second process depends on the first, with the first step data output corresponding to the second step data input.

Algorithm to attain the offset vector from the permutation vector. We will call this algorithm as the *Offset Algorithm by Permutation* (OAP). It uses on details the definition of the offset vector[1] (see equation 1). Two nested loops add the value of each element of the offset vector.

Offset Algorithm by Permutation (OAP)
Routine Specifications:

- n: Size of the permutation vector;
- i, j: Indices of algorithms's vectors;
- p: Permutation vector alternating its elements on $0..n-1$;
- d: Offset vector alternating its elements on $0..n-2$.

```
Routine:
    For $i \leftarrow 0$ to $n-2$ do
        For $j \leftarrow 0$ to $i$ do
            If $p_j > p_{i+1}$
                $d_i \leftarrow d_i + 1$
            End If
        End For
    End For
return $d$.
```

Algorithm to attain the serial number from the offset vector. Like the OAP, the *Serial Algorithm by Offset* (SAO) also uses definitions already showed at this section. We know that the value attributed for the elements of the offset vector, defined on 3 and 4 can be ordered depending on the serial. Given the variables:

- $d = (d_1, d_2, ..., d_k, ..., d_n)$: Offset vector;
- d_k: Offset vector's element;
- s: Permutation's serial number;
- k: Index of the offset vector;
- q: Quotient of the division between $s-1$ and $(k+1)!$;
- $\lfloor x \rfloor$: It returns the biggest integer value smaller than x.

For direct list, we have:

$$s - 1 = d_k.k! + \lfloor q.(k+1)! \rfloor \tag{11}$$

For inverted lists, we have:

$$s - 1 = (k - d_k).k! + \lfloor q.(k+1)! \rfloor \tag{12}$$

An important question for the algorithm implementation is how to find the variable's q value. To solve this question we have to consider that d_n belongs to a column that has only one list on a direct disposition (see Table 2, last d vector column). With this information, we conclude that the quotient for this column corresponds to a value between zero (closed) and one (open), making:

$$\lfloor q.(k+1)! \rfloor \tag{13}$$

Corresponds to zero. With this, 11 and 12 is equivalent to:

$$s - 1 = d_k.k! \tag{14}$$

For direct lists, and

$$s - 1 = (k - d_k).k! \tag{15}$$

For inverted lists.

The strategy of implementation to get the serial is done by incremental mode, working the current information being based on past information, doing a raster on the offset vector from right to left. This is particularly useful on this case, considering the next value that will converge to the final serial that belongs to a group related with the previous serial elements. Each quotient q during the raster will corresponds to zero, because we are considering the division in relation to the gotten serial. Another relevant question is how to classify if the previous element of the offset vector it is contained in a direct or inverted list. Looking at Table 2, we can make easily an equivalence grade, showed in Table 3.

Exemplifying the attainment serial process from the offset vector, we have:

TABLE 3. Next element status list of the offset vector

Previous Value - Parity			Next Value - Parity
Element	List		List
Even	Direct	$\Longrightarrow$	Direct
Even	Inverted	$\Longrightarrow$	Inverted
Odd	Direct	$\Longrightarrow$	Inverted
Odd	Inverted	$\Longrightarrow$	Direct

Let $d = (1, 0, 3)$ be the offset vector and s the permutation serial. As the raster is done from right to left, we work first with the value 3. As was already said, we have as an initial input a direct list which 3 is contained. For direct lists, we use an equation showed on 14:

$$s \leftarrow 3.3! \Rightarrow 18 \tag{16}$$

After that, we observer the next element of the offset vector. As the previous element is in a direct list and it is odd, the element 0 will be in a inverted list. Using 15, we have:

$$s \leftarrow s + (2 - 0).2! \Rightarrow 18 + 4 \Rightarrow 22 \tag{17}$$

Finally, observing the last element of the offset vector, we have the previous element which is in a inverted list and it is even. Looking at Table 3, we evidence that the element 1 is in a inverted list. Thus, we have:

$$s \leftarrow s + (1 - 1).1! \Rightarrow 22 + 0 \Rightarrow 22. \tag{18}$$

After we have calculated the serial in a incremental mode, we add 1 to the serial, because the equations 14 and 15 depend on $s - 1$. Thus, the serial value characterizes it between $(1, n!)$. Where n is the size of the permutation value given as an input:

$$s \leftarrow s + 1 \Rightarrow 22 + 1 \Rightarrow 23. \tag{19}$$

The serial 23 is the corresponding serial to the offset vector $(1, 0, 3)$ which corresponds to the permutation vector $(3, 2, 4, 1)$. On the algorithm's implementation showed here, the increment of one unit to the serial was made in the beginning, before get into the loop. A boolean variable was specified to determine if the list which belongs the elements is direct or inverted. The conditions that determine if the list is direct or inverted is optimized from four to two conditions. The algorithm returns a non-negative integer value, corresponding to the serial number required.

Serial Algorithm by Offset (SAO)
Routine Specifications:

- s: Serial value;
- n: Size of the offset vector;
- i: Index of the offset vector;
- d: Offset vector, alternating its elements on $0..n-1$;
- *direct*: Boolean variable. It determines if the list is direct or not.

Routine:

```
$s \leftarrow [\ d_{n-1}.(n-1)!\ ] + 1$
$direct \leftarrow true$
```

```
    For $i \leftarrow n-1$ to 1 do
        If $(d_i = even\ and\ direct = true)\ or\ (d_i = odd\ and\ direct = false)$
            $direct \leftarrow true$
            $s \leftarrow s + d_{i-1}.(i-1)!$
        End If
        Else
        If $(d_i = odd\ and\ direct = true)\ or\ (d_i = even\ and\ direct = false)$
            $direct \leftarrow false$
            $s \leftarrow s + (i - d_{i-1}).(i-1)!$
        End If
    End For
return s.
```

Serial Composition Method

Basic Concepts. On the combinatorial family, the composition of an integer n in k parts is defined by:

$$n = r_1 + r_2 + \ldots + r_k \quad r_i \geq 0 \quad i = 1..k \tag{20}$$

Where the order of the elements is important on the compositions generations. The *Next Composition Algorithm* [1] does the task of making composition, obtaining the next composition starting from the one before, interactively working to reach until the last composition of the list. The proposal of the described method on this section is to obtain the vector from the list position, with no need of processing the compositions one by one until getting the expecting vector.

Construction of the SCM. The SCM was built from the repetition patterns which are present on the composition vector throughout its list. It is possible to see that this repetition is done in specific positions on the compositions lists: they can be obtained through calculus. This feature of the composition vector makes possible the cast among the specific positions on the list. This way, it decreases the number of needing interactions to find the vector of a specific position. The calculus of the specific positions is defined below.

Used definitions on the built of SCM. It is known that the total number of compositions of n in k parts is defined by[1]:

$$J(n,k) = \left(\begin{array}{c} n+k-1 \\ n \end{array} \right) \tag{21}$$

It is also know that:

$$\sum_{i=0}^{k} \left(\begin{array}{c} n-i \\ k-i \end{array} \right) = \left(\begin{array}{c} n+1 \\ k \end{array} \right) \tag{22}$$

Associating 21 and 22 we have:

$$\sum_{i=0}^{n} \left(\begin{array}{c} n+k-2-i \\ n-i \end{array} \right) = \left(\begin{array}{c} n+k-1 \\ n \end{array} \right) \tag{23}$$

For a c composition vector of a k length from the L composition list, the modification of k-tuple element in L is determined by the index of the partial sum defined on the equation 23. This way, it is possible to know when a component of c stops to repeat its current element to get modifications. Taking as a basis equation 23, we can see this index modification with the definition of the M matrix, exposed in 24.

$$M(n,k) = \begin{pmatrix} \binom{n+k-2}{n} & \binom{n+k-3}{n-1} & \cdots & \binom{k-2}{0} \\ \binom{n+k-3}{n} & \binom{n+k-4}{n-1} & \cdots & \binom{k-3}{0} \\ \vdots & \vdots & \ddots & \vdots \\ \binom{n}{n} & \binom{n-1}{n-1} & \cdots & \binom{0}{0} \end{pmatrix} \tag{24}$$

- Each line of M that corresponds to the elements of the sum on equation 23;
- From one line to another, the superior element of the binomial is decreased in one unity;
- M possui $k-1$ lines and $n+1$ columns.

Now let's see bellow how M is used to represent the SCM execution.

Description of the *M* Matrix Raster. Initially, we have as initial information the serial number of the composition on the list, to obtain the composition vector. Let's suppose s is the serial related to the input variable of the method to a composition vector with a k length. The elements of the first line of M are added until this sum is over $s-1$. When it's found a value in the sum which is over $s-1$, the corresponding value of the column which the added element immediately before the actual one is found is given to the last element of the composition vector found. After finding the last element (the attributions of each element of the composition vector are done from the end to the beginning until the second element of the composition vector), the next element, that means the element before the last one of the composition vector is found in M going down in diagonal to the next line of M, re-starting the counting of indices starting from the actual column. This is done in a successive way, until the sum of the elements which were visited in M is $s-1$. Let's see an example of this strategy in a matrix $M(7,5)$ with serial $s-1 = 282$, seen in 25. The result of the raster is the composition vector $c(283) = \{1, 0, 2, 1, 3\}$. It is known that the first element of the vector does not need to be calculated using M, once it can be obtained from the complement of the sum of the elements already found through the raster done in M. There is in 25 on the first row with the partial sum until the third element (we allocate the number 3 on the last position of the list), the second row with only one element of the partial sum (we allocate the number 1 on the position before the last one of the list), and so on. The sum of the underlined numbers in 25 converged to $s-1$.

$$M(7,5) = \begin{pmatrix} \underline{\mathbf{120}} & \underline{\mathbf{84}} & \underline{\mathbf{56}} & 35 & 20 & 10 & 4 & 1 \\ 36 & 28 & 21 & \underline{\mathbf{15}} & 10 & 6 & 3 & 1 \\ 8 & 7 & 6 & 5 & \underline{\mathbf{4}} & \underline{\mathbf{3}} & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{25}$$

Description of the SCM. Related to implementation, the algorithm proposed here to the SCM is abstract to the structure of M matrix, defined above. The loop is traced as defined, applying on the first element of the composition vector the corresponding value of the complement of the sum of the elements yet to come. On the subroutine *element* there is the procedure to obtain each element of the composition vector.

Serial Composition Method (SCM)
Algorithm specifications:

- n: Number of the composition (Composition of n elements in k parts);
- k: Parts of the composition;
- s: Serial of the composition;
- a: Auxiliary variable which makes the increment to the convergence of the serial number;
- x, y: Auxiliary variables, for the definition of new binomial indices;
- $C_{i,j}$: Combination of i elements j by j;
- z: Value of the complement, used to apply the value to the first element of the composition vector.

```
Routine:
  $z \leftarrow a \leftarrow 0$
  $x \leftarrow n + k - 2$
  $y \leftarrow n$
  For $i \leftarrow 0$ to $k - 2$ do
    $c_{k-1-i} \leftarrow element$
    $z \leftarrow z + c_{k-1-i}$
  End For
  $c_0 \leftarrow n - z$
return $c$.
Subroutine element
  For $j \leftarrow 0$ to $n - 1$ do
    If $a + C_{x-i-j,y-j} \leq s - 1$
      $a \leftarrow a + C_{x-i-j,y-j}$
    End If
    Else
      $x \leftarrow x - j$
      $y \leftarrow y - j$
      return $j$
    End Else.
  End For
return $n$.
End subroutine element.
```

The inverse process of SCM (Unserial Method)

Related to the inverse process of the SCM, the actual view is the composition serial number obtaining, having as input data the composition vector of a number n in k parts. In this case, each component of the composition vector is seen as part of superior intervals of a nested loops, which makes the raster on the composition vector. The inverse process has concise data input related to the data input of the SCM, once the inverse process will not work under the value convergence of a serial number method: everything is already well defined on the composition vector components, known that is only necessary to process the loops with related interactions to each component of the composition vector give as input. As to the SCM, the proposed algorithm for the inverse process to the SCM works only with indices referred to the defined matrix on SCM. The use of the indices abstracting from the matrix structure decreases the memory use and processes only what is needed for the calculus of the trace to be run on M matrix.

Serial Composition Method (Inverse process)
Algorithm specifications:

- n: Number of the composition (Composition of n elements in k parts);
- k: Parts of the composition;
- s: Serial of the composition;
- x, y: Auxiliary variables, for the definition of new binomial indices;
- $C_{i,j}$: Combination of i elements j by j.

Routine:
```
$x \leftarrow n + k - 2$
$y \leftarrow n$
$s \leftarrow 1$
For $i \leftarrow k - 1$ to 1 do
    For $j \leftarrow 0$ to $c_i - 1$ do
        $s \leftarrow s + C_{x-j-[(k-1)-i],y-j}$
    End For
    $x \leftarrow x - c_i$
    $y \leftarrow y - c_i$
End For
return s.
```

Serial Partition of an n-Set Method

Basic Concepts. For the sets partitions, we consider a family of subsets $T_1, T_2, \ldots, T_k$ contained in a set $S = \{1, 2, \ldots, n\}$ that satisfy the conditions[1]:

$$T_i \cap T_j = \oslash \quad (i \neq j) \tag{26}$$

$$\bigcup_{i=1}^{k} T_i = S \tag{27}$$

$$T_i = \oslash \quad (i = 1, 2, \ldots, k) \tag{28}$$

We don't consider the order of the elements contained on each one of S subsets. The algorithm *Next Partition of an n-Set*[1] finds the next partition set from the current one, working only with local information. The section's proposal is to obtain a partition of S related to its position on the list of partition, without considering the information about the partition vector.

SPSM Construction. The SPSM was built from the analysis of the data output of the *Next Partition of an n-Set* algorithm. Each index that denotes the owned elements to a subset has a pattern model and the search for these indices is made by a combinatorial structure that take as basis a tree that represents all the partitions of a set with n elements, called Bell Tree. The number of nodes of this structure grows quickly when n increases. So, the solution for such problem is to define a new structure that does a mapping of the tree specific positions. As a reference we took a matrix structure to keep those positions, called D Matrix. The Bell Tree and D Matrix is defined below.

Bell Trees. We can define a Bell Number as a number of partitions possibilities of a set with n elements. Such number is defined by:

$$B_n = \sum_{k=1}^{n} \left\{ {n \atop k} \right\} \tag{29}$$

Where $\left\{ {n \atop k} \right\}$ is the Stirling number of second kind, which is defined by:

$$\left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{i=0}^{k-1} (-1)^i \binom{k}{i} (k-i)^n \tag{30}$$

With the definitions 29 and 30 above, this subsection proposes to show a combinatorial structure which is called here *Bell Tree* and the use of this structure to solve the *Partition of an n-Set* problem. It was used as basis the definition of a partition tree, shown in the algorithm *Next Partition of an n-Set*[1], associating the tree nodes with the equations which its terms are defined on the tree construction properties.

Properties of Bell Trees. Let n be the number of elements of a set that will be partitioned and B the Bell Numbers that denotes a set of terms that can form a mathematical expression in a node of the tree, the structure here defined adopt the partition tree structure with n-Sets and added to it others properties. Follows below some properties of the structure:

- The Tree has n levels;
- The Tree nodes are non-negative integers numbers which is formed by expressions that evolves exclusively Bell Numbers and multiplicative integer constants;
- Let S be a subset contained in B which denotes the number of distinct terms of an expression with a node N, we determine the number of descendants produced by N from the number of elements of S increased one unit;
- The number R of the nodes on a Bell Tree corresponds to:

$$R = \sum_{j=1}^{n} B_j \tag{31}$$

- Let k be the number of descendants of the actual node N, the value for all its $k-1$ are defined by:

Let E_w be a mathematical expression with w terms from the ascendant node, the descendant node corresponds to E_{w-1}. To illustrate such fact, if we have a ascendant node represented by the expression:

$$E_w = (B_w - B_{w-1}) - 2(B_{w-1} - B_{w-2}) \tag{32}$$

Then, the descendants nodes will correspond to the expression:

$$E_{w-1} = (B_{w-1} - B_{w-2}) - 2(B_{w-2} - B_{w-3}) \tag{33}$$

For the k-tuple descendant of N, we have the expression:

$$E_k = E_w - (k-1)E_{w-1} \tag{34}$$

That it is equivalent to say that the k-tuple descendant is equal to the ascendant node value minus $k-1$ times the value of the others descendant nodes.

We can now make a tree that follows this formation law. For $n = 5$, we have a tree illustrated on Figure 1. The three first levels expressions of this tree are in Table 4, which the number of nodes are entered from top to bottom and from left to right.

Table 4. Nodes values of the Tree illustrated on Figure 1

Nodes	Expressions
1	B_5
2	B_4
3	$B_5 - B_4$
4	B_3
5	$B_4 - B_3$
6	$B_4 - B_3$
7	$B_4 - B_3$
8	$(B_5 - B_4) - 2(B_4 - B_3)$

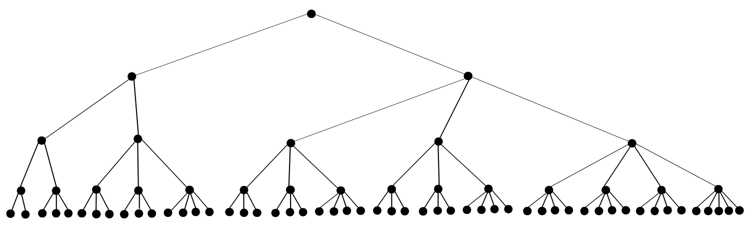

Figure 1. Bell Tree with level 5

Definition of D Matrix. The D Matrix is a superior triangular matrix with n x n dimension. Its first column is made by the Bell Numbers

$$D_{v,0} = B_{n-v} \tag{35}$$

and the others columns are defined by the equation below:

$$D_{u,v} = D_{u,v-1} - v.D_{u+1,v-1} \tag{36}$$

Where u indicates the line and v the column on D. The matrix showed on 37 represents the Matrix D for $n = 6$.

$$D = \begin{pmatrix} 203 & 151 & 77 & 26 & 6 & 1 \\ 52 & 37 & 17 & 5 & 1 & \\ 15 & 10 & 4 & 1 & & \\ 5 & 3 & 1 & & & \\ 2 & 1 & & & & \\ 1 & & & & & \end{pmatrix} \tag{37}$$

The D Matrix fulfilling. Before process the SPSM, it is necessary fill in the components of D Matrix to speed up the attainment calculus of the search on the Bell Tree. The use of D Matrix abstracts the construction of the whole Bell Tree, processing only the necessary to the SPSM use. If the tree had been built completely, we would have the number of nodes R equivalent to the equation 31. Whereas on the D Matrix , we have

$$R = \frac{n^2 + n}{2} \tag{38}$$

Algorithm for D Matrix fulfilling
Algorithm Specifications:

- D: Matrix that keep the specific positions of the Bell Tree;
- i, j: Indices for the raster of D Matrix;
- B_n: Bell Number.

Routine:

```
For $i \leftarrow 0$ to $n - 1$ do
    $D_i, 0 \leftarrow B_{n-i}$
End For
For $i \leftarrow 1$ to $n - 1$ do
    For $j \leftarrow 0$ to $n - i - 1$
        $D_{j,i} \leftarrow D_{j,i-1} - i.D_{j+1,i-1}$
    End For
End For.
```

SPSM's Specifications. After that the D Matrix has its values filled in, the SPSM can be invoked. We have an external loop that runs all the partition vector, attributing for each component by means of the sub-routine *element* that does a raster on the current tree level (the structure was abstracted from the method). On this raster, we have a condition that examines if the variable used for the indication of extrapolation surpassed the serial number token as an input. If it exceed this value, we go down a tree level; if not, we add the descendants' current value to the control variable for later verification of extrapolation, token as an input. If the condition had not been satisfied for the whole loop, it means that the search on the tree's level arrived to the last descendant of this one, indicating that the search will be expanded to the last descendant of the current tree's level.

Serial Partition of an n-Set Method
Algorithm Specifications:

- p: Partition vector;
- D: Matrix that keep the specific positions of the tree;
- i, j: Indices for the raster of D Matrix;
- k: Index for the raster on the p vector;
- t: Index that determines the element of each component of partition vector;
- a: Number that does the convergence for the serial number given as input;
- s: Partition's serial.

```
Routine:
  $i \leftarrow j \leftarrow a \leftarrow 0$
  For $k \leftarrow 0$ to $n-1$ do
    $p_k \leftarrow element$
  End For
return p.

Sub-Routine $element$
  For $t \leftarrow 0$ to $j$ do
    If $a + D_{i,j} \geq s$
      $i \leftarrow i+1$
      return t.
    End If
    Else
      $a \leftarrow a + D_{i,j}$
    End Else
  End For
  $j \leftarrow j+1$
return $j$.
```

Algorithm for stylized output of the partition vectors. We know that the output gotten on both SPSM and Next Partition of an n-Set corresponds to the indices of each subset of the partition on the set. For the partition vector on position 26, we have the output $(0, 1, 1, 0, 0)$ and the stylized output corresponds to $(1, 4, 5)(2, 3)$, indicating that $1, 4$ and 5 is on the first subset (indicated as 0), with 2 and 3 on the second subset (indicated as 1 on the partition vector). For show the stylized output, it was used a *string* vector to get the elements adequately on each subset which it belongs. Next, the algorithm does the output treatment and the all elements of each subset be ordered related to the last elements of partition as an input for the stylized output of the n-sets' partitions method.

Algorithm for stylized output data
Algorithm Specifications:

- s: *Strings* vector that does the mapping of the elements referring to the partition vector's indices;
- p: Partition vector;
- i: Indices for the raster of s vector;
- r: Final output on *string* form;
- n: Size of the partition vector;
- $+$: Operator that denotes a concatenation between *strings*;
- k: *String* that get the current element of the stylized output;
- $length(w)$: Function that returns the size of the *string* w;
- $substring(w, ini, sup)$: Function that returns a w *substring* of the interval between ini and sup.

Routine:

```
For $i \leftarrow 0$ to $n-1$ do
    $s_{p_i} \leftarrow s_{p_i} + (i+1) + ","$
End For
For $i \leftarrow 0$ to $n-1$ do
    If $length(s_i) > 1$
        $k \leftarrow substring(s_i, 0, length(s_i) - 1)$
    End If
    $s_i \leftarrow "(" + k + ")"$
    If $s_i = "()"$
        $r \leftarrow r + s_i$
    End If
End For
return r.
```

The Inverted Process of SPSM (Unserial Method)

For the inverted process of SPSM, we have a partition vector as input and the serial number as output. Such process is made taking each partition vector's component as the number of loops that can does a sum of each specific position referring to the current node of the tree mapping through the D Matrix.

Inverted Process of SPSM Specifications. Initially, we have the indices attribution referring to the D Matrix. The u index has the attribution equals to 1 because it is not necessary do the mapping on the first component of the D Matrix, once it always begins with zero for being the main tree descendant. After that, we have an external loop that runs the whole partition vector given as input, with another loop that does the sum to the main positions' serial mapped on D Matrix. After the sum attributions to the serial, we have the checking if the actual component should go down a line on D Matrix (equivalent to go down a level on the tree). If it should go down a level, the line is added by one unit. If not, the column on D is added by one unit, indicating the search to the descendants on the current node reached its last one and the search be doing on the other tree ramification.

Inverted Process of SPSM
Algorithm Specifications:

- p: Partition vector, given as input;
- i, j: Indices referring to the raster on p vector;

- D: Matrix of specific positions on the Bell Tree;
- u, v: Indices referring to the mapping on D;
- s: Partition's serial, that is the result expected from de algorithm.

Routine:

```
  $i \leftarrow j \leftarrow v \leftarrow 0$
  $s \leftarrow u \leftarrow 1$
  For $i \leftarrow 1$ to $n-1$ do
      For $j \leftarrow 0$ to $p_i - 1$ do
          $s \leftarrow s + D_{u,v}$
      End For
      If $j \leq p_{i-1}$
          $u \leftarrow u + 1$
      End if
      Else
          $v \leftarrow v + 1$
      End Else
  End For
return s.
```

Serial k-Subset of an n-Set Method

Basic Concepts. We call $\binom{n}{k}$ the number of possibilities to combine n things on k different parts. On analyzed literature[1], we have two ways to do this work on a sequential form. The first method builds the k-subsets in a lexicographic order and the second method obtains the next subset from its predecessor, subtracting one element of the set and adding on other element of the subset.

The proposal of this section is presents a method that, put in action iteratively, it creates a list in a lexicographic order given as input the position on the list of possible combinations. This strategy is more efficient when we desire to get a combination on a specific position inside the whole list of combinations.

Getting the combinations on lexicographic order. The algorithm *Next k-subset of an n-Set*[1] is able to create on a simple way the combinations in lexicographic order in a non-recursive way. The recursive model of this algorithm will be shown on this section optionally. The current combination of the recursive model is showed through the *showOutput* method, that is indicating here an output model of generic data for a combination that is going to have as output elements that can vary between 0 and $n-1$. The call to begin the routine must be done on a *Combine*(0) way, taking as stopped of the recurrence, the moment that the parameter did k recurrences. Next, we have a recursive model of the algorithm, without taking longer on its construction.

Recursive k-subset of an n-Set Algorithm
Algorithm Specifications:

- k: Dimension of the subset;
- s: Subset vector;
- n: Set's cardinality;
- y: Vector that determines the changes of n set elements in subset k.

```
Routine:
  $Combine(i)$
    For $s_i \leftarrow sum(i,i,0)$ to $n-(k-i)$ do
      If $i \neq k-1$
        $Combine(i+1)$
      Else
        $showOutput$
      End If-Else
    End For
    $y_{i+1} \leftarrow 0$
    $y_i \leftarrow y_i + 1$
  End Combine.

  Subroutine $sum(w,j,z)$
    For $i \leftarrow 0$ to $j$ do
      $z \leftarrow z + y_i$
    End For
    return $z+w$.
  End $sum$.
```

SKSM Construction. To build the SKSM, we observe the data output of the subsets' list created by the *Next k-subset of an n-Set* algorithm, characterizing the repetition's pattern in the elements of subset. In this case, the pattern can be delineated under a tree model. The formation law of this tree is showed below.

Definition of the Binomial Tree. The structure that represents the repetition's pattern is characterized as a tree formed exclusively by binomial coefficients. Let $\binom{n}{k}_w$ be the current node of the tree with w label, its descendants are defined by

$$\binom{n}{k}_w \rightarrow \begin{cases} \binom{n-1}{k-1}_{w+1} \\ \binom{n-2}{k-1}_{w+2} \\ \vdots \\ \binom{n-k+1}{k-1}_{w+n-k+1} \end{cases} \tag{39}$$

Each ascendant will have $n+k-1$ descendants, and the repetition's pattern of combinations are analyzed through the insertion of labels in each node of the tree. For each label of the current node, the descendants nodes enter its labels in relation to the ascendant node, indicating the value from each element of the subset founded on the tree.

SKSM Specifications. The SKSM abstracts the construction of the tree, doing the search calculating only the binomial coefficients and its relations with the nodes' labels that were visited on the search. The method does an external loop attributing to each element of the subset the result of the subroutine *element*. The subroutine *element* does a search on the tree's nodes. The abstraction of the tree is made through the indices changes of the binomial coefficient attributed on x and y. The method is described below.

Serial k-Subset of an n-Set Method
Algorithm Specifications:

- p: k-dimensional Subset;
- n: Cardinality of the Set;
- a: Auxiliary variable that is used to check the stopped of the method;
- x, y: Indices of the binomial coefficients functioning as the formation law of the Binomial Tree;
- r: Variable that controls the labels of Binomial Tree;
- s: Serial of p subset;
- $C_{n,k}$: $\binom{n}{k}$.

```
Routine:
  $x \leftarrow n$
  $y \leftarrow k - 1$
  $a \leftarrow r \leftarrow 0$
  For $i \leftarrow 0$ to $k - 1$ do
    $p_i \leftarrow element$
  End For
return $p$.
End SKSM Routine.

Subroutine $element$
  For $j \leftarrow 1$ to $x - y + 1$ do
    If $a + C_{x-j,y} < s$
      $a \leftarrow a + C_{x-j,y}$
    Else
      $x \leftarrow x - j$
      $y \leftarrow y - 1$
      $r \leftarrow r + j$
      return $r$.
    End If-Else
  End For
return $r$.
End Subroutine $element$.
```

The Inverted process of SKSM (Unserial Method)

As many others serial methods from others combinatorial problems with same importance, the SKSM also has its inverted process. What is giving as input is the vector that represents the k-dimensional subset with the cardinality of the set, called n, getting as output its position (i.e. the serial number) on the subsets list in a lexicographic order.

SKSM Inverse Process Specifications. On the inverted SKSM process, it is taken as the maximum limit of the internal loop is the difference between the elements of the subset. This will determine how much the loop will repeat related to the elements of the subset. The indices of the binomial coefficient do an appropriate control for the adding among the nodes be made correctly. The inverted process also abstract the tree's structure, getting the result by means of local information related to the subset given as input.

Serial k-Subset of an n-Set Method (Inverted Process)
Algorithm Specifications:

- p: k-dimensional Subset;
- n: Cardinality of the Set;
- x, y: Indices of the Binomial Coefficients;
- r: Variable that will control the internal loop of the method;
- s: Serial of p subset;
- $C_{n,k}$: $\binom{n}{k}$.

Routine:

```
  $x \leftarrow n$
  $y \leftarrow k-1$
  $s \leftarrow 1$
  $r \leftarrow 0$
  For $i \leftarrow 0$ to $k-1$ do
    For $j \leftarrow 1$ to $p_i - r - 1$ do
      $s \leftarrow s + C_{x-j,y}$
    End For
    $x \leftarrow x - (p_i - r)$
    $y \leftarrow y - 1$
    $r \leftarrow p_i$
  End For
return s.
```

Serial Subset Method

Basic Concepts. We have 2^n possible configurations to get the subsets of a $\{1, 2, \ldots, n\}$ set. The disposition of the subset's elements can be represented by a flag which activates its insertion into the subset. For example, the configuration $\{1, 0, 1, 1, 0\}$ represents the subset $\{1, 3, 4\}$. The *Next Subset of an n-Set*[1] algorithm does this job in a sequential form, returning the next subset from the actual one.

This section proposes to present a method that returns a subset of an n-Set from its serial number. The inverted process for this method is also showed, getting the serial number from the subset given as input.

SSM Construction. The SSM was built from the analysis of repetition's pattern that the subsets presents in the complete list of subsets. Similarly to the *Serial Permutation Method* related to repetition's pattern of the offset vector, the SSM has a regular repetition pattern, being able to get each subset's component through a closed equation.

Let p be a subset of a n-Set, each subset's component with index $k = 0, 1, \ldots, n-1$ related to a serial number $s = 1, 2, \ldots, 2^n$ is defined by

$$p_k = \left\lfloor \frac{(s - 1 + 2^k)\ mod\ 2^{k+2}}{2^{k+1}} \right\rfloor \tag{40}$$

The equation 40 doesn't present restrictions in the pattern's representation. Each component $p_k \in \{0, 1\}$, and 2^k in $(s-1+2^k)$ is related with a cyclic structure which the numbers are presented at the list. This peculiarity is easily identified on output data of *Next Subset of an n-Set*[1] algorithm and the method here proposed.

SSM Specifications. About the SSM's specifications, we have a loop that do an association between the equation 40 and the structure that represents the subset. The method returns a subset from the serial number given as input, where the number 1 on the output data indicates which subset's component will be activated, as explained in the basic concepts on this section.

Serial Subset Method
Algorithm Specifications:

- p: Subset of n-Set;
- n: Cardinality of the Set;
- s: Serial of p subset;

```
Routine:
  For $i \leftarrow 0$ to $n-1$ do
    $p_i \leftarrow \lfloor((s-1+2^i)\ mod\ 2^{i+2})\ /\ 2^{i+1}\rfloor$
  End For
return $p$.
End Routine.
```

The Inverted Method of SSM (Unserial Method)

In this section the inverse process of the SSM will be shown. The input data is the subset (the elements of the set that will be activated) and the output data is the corresponding position in the list of subsets. In the case at hands, we determine which it is the moment to walk through specific positions of the identified repetition's pattern in subsets. The inverted process for subsets also have similarities with the inverted process of *Serial Permutation Method.* The list of subsets, from right to left, presents a sequence of 0's and 1's, in this order. From there, the list for analysis of the subset's next component can be inverted in the case of this subset's element be equals to 1. In this case, we cannot advance in the list for convergence of the serial's desired value. Otherwise, the serial value is modified on iterative form among the specific position of the list until reach the subset's serial.

SSM Unserial Method Specifications. In the SSM's specification, it was defined a logical variable which defines if the list of current subset's component is direct or inverted, defining as $true$ for direct lists and $false$ for inverted lists, starting this variable as $true$ because the components' disposition has been analysed from right to left. The initial serial's value is 1. After that, we have a decreased loop that does the verifications in each component of the subset for determines if it advances or not for the convergence of the final result. It was observed that such condition in the way as was constructed can be represented by *eXclusive OR* logical connective, usually denoted by *xor* operator. When the *xor* is satisfied, the logical variable is activated as $true$, indicating that the next component is included on a direct list. Otherwise, the serial advances other specific position and the list is configured as inverted.

Inverted Process of the SSM (Unserial Method)
Algorithm Specifications:

- p: Subset of n-Set;
- n: Cardinality of the Set;
- s: Serial of p subset;
- d: Logical variable, which indicates if the list of subsets is on direct or inverted order.

```
Routine:
  $s \leftarrow 1$
  $d \leftarrow true$
  For $i \leftarrow n - 1$ down to 0 do
    If $(p_i = 1 \ xor \ d) = true$
      $d \leftarrow true$
    Else
      $d \leftarrow false$
      $s \leftarrow s + 2^i$
    End If-Else
  End For
return $s$.
End Routine.
```

Conclusion

(This section is not ready).

References

[1] WILF, Herbert S., NIJENHUIS, A. Combinatorial Algorithms for computers and calculators. Academic Press, INC, 1978.

(Melo) Departamento de Estatística e Informática - UNICAP
E-mail address, Melo: glaucio.melo@gmail.com

(Oliveira-Lima) Departamento de Estatística e Informática - UNICAP
E-mail address, Oliveira-Lima: eal@dei.unicap.br