CHAPTER 1

Generating All and Random Instances of a Combinatorial Object

IVAN STOJMENOVIC

1.1 LISTING ALL INSTANCES OF A COMBINATORIAL OBJECT

The design of algorithms to generate combinatorial objects has long fascinated mathematicians and computer scientists. Some of the earliest papers on the interplay between mathematics and computer science are devoted to combinatorial algorithms. Because of its many applications in science and engineering, the subject continues to receive much attention. In general, a list of all combinatorial objects of a given type might be used to search for a counterexample to some conjecture, or to test and analyze an algorithm for its correctness or computational complexity.

This branch of computer science can be defined as follows: Given a combinatorial object, design an efficient algorithm for generating all instances of that object. For example, an algorithm may be sought to generate all n-permutations. Other combinatorial objects include combinations, derangements, partitions, variations, trees, and so on.

When analyzing the efficiency of an algorithm, we distinguish between the cost of *generating* and cost of *listing* all instances of a combinatorial object. By generating we mean producing all instances of a combinatorial object, without actually outputting them. Some properties of objects can be tested dynamically, without the need to check each element of a new instance. In case of listing, the output of each object is required. The lower bound for producing all instances of a combinatorial object depends on whether generating or listing is required. In the case of generating, the time required to "create" the instances of an object, without actually producing the elements of each instance as output, is counted. Thus, for example, an optimal sequential algorithm in this sense would generate all n-permutations in $\theta(n!)$ time, that is, time linear in the number of instances. In the case of listing, the time to actually "output" each instance in full is counted. For instance, an optimal sequential algorithm generates all n-permutations in $\theta(nn!)$ time, since it takes $\theta(n)$ time to produce a string.

Handbook of Applied Algorithms: Solving Scientific, Engineering and Practical Problems
Edited by Amiya Nayak and Ivan Stojmenović Copyright © 2008 John Wiley & Sons, Inc.

COPYRIGHTED MATERIAL

Let P be the number of all instances of a combinatorial object, and N be the average size of an instance. The *delay* when generating these instances is the time needed to produce the next instance from the current one. We list some desirable properties of generating or listing all instances of a combinatorial object.

Property 1. *The algorithm* lists *a*ll instances in asymptotically optimal time, *that is*, *in time* $O(NP)$.

Property 2. *The algorithm* generates *all instances with* constant average delay. *In other words*, *the algorithm takes* $O(P)$ *time to generate all instances. We say that a generating algorithm has constant average delay if the time to generate all instances is* $O(P)$; *that is*, *the ratio* T/P *of the time* T *needed to generate all instances and the number of generated instances* P *is bounded by a constant.*

Property 3. *The algorithm generates all instances with* constant (worst case) delay. *That is*, *the time to generate the next instance from the current one is bounded by a constant. Constant delay algorithms are also called* loopless *algorithms, as the code for updating given instance contains no* (*repeat*, *while*, *or for*) *loops.*

Obviously, an algorithm satisfying Property 3 also satisfies Property 2. However, in some cases, an algorithm having constant delay property is considerably more sophisticated than the one satisfying merely constant average delay property. Moreover, sometimes an algorithm having constant delay property may need more time to generate all instances of the same object than an algorithm having only constant average delay property. Therefore, it makes sense to consider Property 3 independently of Property 2.

Property 4. *The algorithm* does not use large integers *in generating all instances of an object. In some papers*, *the time needed to* "*deal*" *with large integers is not properly counted in.*

Property 5. *The algorithm is the* fastest *known algorithm for generating all instances of given combinatorial object. Several papers deal with comparing actual (not asymptotic) times needed to generate all instances of given combinatorial object, in order to pronounce a* "*winner*," *that is*, *to extract the one that needs the least time. Here*, *the fastest algorithm may depend on the choice of computer. Some computers support fast recursion giving the recursive algorithm advantage over iterative one. Therefore*, *the ratio of the time needed for particular instructions over other instructions may affect the choice of the fastest algorithm.*

We introduce the *lexicographic order* among sequences. Let $a = a_1, a_2, \ldots, a_p$ and $b = b_1, b_2, \ldots, b_q$ be two sequences. Then a precedes $b (a < b)$ in lexicographic order if and only if there exists i such that $a_j = b_j$ for $j < i$ and either $p = i + 1 < q$ or $a_i < b_i$. The lexicographic order corresponds to dictionary order. For example, $112 < 221$ (where $i = 1$ from the definition).

For example, the lexicographic order of subsets of $\{1, 2, 3\}$ in the set representation is Ø, $\{1\}$, $\{1, 2\}$, $\{1, 2, 3\}$, $\{1, 3\}$, $\{2\}$, $\{2, 3\}$, $\{3\}$. In binary notation, the order of subsets is somewhat different: 000, 001, 010, 011, 100, 101, 110, 111, which correspond to subsets Ø, $\{3\}$, $\{2\}$, $\{2, 3\}$, $\{1\}$, $\{1, 3\}$, $\{1, 2\}$, $\{1, 2, 3\}$, respectively. Clearly the lexicographic order of instances depends on their representation. Different notations may lead to different listing order of same instances.

Algorithms can be classified into *recursive* or *iterative,* depending on whether or not they use recursion. The iterative algorithms usually have advantage of giving easy control over generating the next instance from the current one, which is often a desirable characteristic. Also some programming languages do not support recursion. In this chapter we consider only iterative algorithms, believing in their advantage over recursive ones.

Almost all sequential generation algorithms rely on one of the following three ideas:

1. *Unranking*, which defines a bijective function from consecutive integers to instances of combinatorial objects. Most algorithms in this group do not satisfy Property 4.
2. *Lexicographic updating,* which finds the rightmost element of an instance that needs "updating" or moving to a new position.
3. *Minimal change,* which generates instances of a combinatorial object by making as little as possible changes between two consecutive objects. This method can be further specified as follows:
   - Gray code generation, where changes made are theoretically minimal possible.
   - Transpositions, where instances are generated by exchanging pairs of (not necessarily adjacent) elements.
   - Adjacent interchange, where instances are generated by exchanging pairs of adjacent elements.

The algorithms for generating combinatorial objects can thus be classified into those following lexicographic order and those following a minimal change order. Both orders have advantages, and the choice depends on the application. Unranking algorithms usually follow lexicographic order but they can follow minimal change one (normally with more complex ranking and unranking functions).

Many problems require an exhaustive search to be solved. For example, finding all possible placements of queens on chessboard so that they do not attack each other, finding a path in a maze, choosing packages to fill a knapsack with given capacity optimally, satisfy a logic formula, and so on. There exist a number of such problems for which polynomial time (or quick) solutions are not known, leaving only a kind of exhaustive search as the method to solve them.

Since the number of candidates for a solution is often exponential to input size, systematic search strategies should be used to enhance the efficiency of exhaustive search. One such strategy is the *backtrack*. Backtrack, in general, works on partial solutions to a problem. The solution is extended to a larger partial solution if there is a hope to reach a complete solution. This is called an extend phase. If an extension of the current solution is not possible, or a complete solution is reached and another one is sought, it backtracks to a shorter partial solution and tries again. This is called a reduce phase. Backtrack strategy is normally related to the lexicographic order of instances of a combinatorial object. A very general form of backtrack method is as follows:

```
initialize;
repeat
    if current partial solution is extendable then extend else reduce;
    if current solution is acceptable then report it;
until search is over
```

This form may not cover all the ways by which the strategy is applied, and, in the sequel, some modifications may appear. In all cases, the central place in the method is finding an efficient test as to whether current solution is extendable. The backtrack method will be applied in this chapter to generate all subsets, combinations, and other combinatorial objects in lexicographic order.

Various algorithms for generating all instances of a combinatorial object can be found in the journal *Communications of ACM* (between 1960 and 1975) and later in *ACM Transactions of Mathematical Software* and *Collected Algorithms from ACM*, in addition to hundreds of other journal publications. The generation of ranking and unranking combinatorial objects has been surveyed in several books [6,14,21,25,30,35,40].

1.2 LISTING SUBSETS AND INTEGER COMPOSITIONS

Without loss of generality, the combinatorial objects are assumed to be taken from the set $\{1, 2, \ldots, n\}$, which is also called n-set. We consider here the problem of generating subsets in their set representation. Every subset [or (n,n)-subset] is represented in the set notation by a sequence $x_1, x_2, \ldots, x_r$, $1 \leq r \leq n$, $1 \leq x_1 < x_2 < \ldots < x_r \leq n$. An (m,n)-subset is a subset with exactly m elements.

Ehrlich [11] described a loopless procedure for generating subsets of an n-set. An algorithm for generating all (m,n)-subsets in the lexicographic order is given in the work by Nijenhius and Wilf [25]. Semba [33] improved the efficiency of the algorithm; the algorithm is modified in the work by Stojmenović and Miyakawa [37] and presented in Pascal-like notation without goto statements. We present here the algorithm from the work by Stojmenović and Miyakawa [37]. The generation goes in the following manner (e.g., let $n = 5$):

1	12	123	1234	12345
			1235	
		124	1245	
		125		
	13	134	1345	
		135		
	14	145		
	15			
2	23	234	2345	
		235		
	24	245		
	25			
3	34	345		
	35			
4	45			
5.				

The algorithm is in *extend* phase when it goes from left to right staying in the same row. If the last element of a subset is n, the algorithm shifts to the next row. We call this the *reduce* phase.

```
read( n); r ← 0; x_r ← 0;
repeat
        if x_r<n  then  extend  else  reduce;
        print out x_1, x_2, ..., x_r
until x_1 = n
extend ≡ {x_{r+1} ← x_r + 1; r ← r + 1}
reduce ≡ {r ← r − 1; x_r ← x_r + 1}.
```

The algorithm is loopless, that is, has constant delay. To generate (*m,n*)-subsets, the **if** instruction in the algorithm should be changed to

```
if x_r<n and r<m  then {x_{r+1} ← x_r + 1; r ← r + 1} (* extend *)
        else  if x_r<n  then  x_r ← x_r + 1  (*cut *)
                        else {r ← r − 1; x_r ← x_r + 1} (* reduce *).
```

The new *cut* phase will be used when the algorithm goes from one subset to a subset in a lower row, skipping several subsets (having more than m elements). For example, for $m = 3$ and $n = 5$, the first three columns of the last table of subsets are

(3,5)-subsets. This illustrates the backtrack process applied on all subsets to extract (*m,n*)-subsets.

We now present the algorithm for generating variations. A (*m,n*)-variation out of $\{p_1, p_2, \ldots, p_n\}$ can be represented as a sequence $c_1c_2\ldots c_m$, where $p_1 \le c_i \le p_n$. Let $z_1z_2\ldots z_m$ be the corresponding array of indices, that is, $c_i = p_{z_i}$, $1 \le i \le m$. The next variation can be determined by a backtrack search that finds an element c_t with the greatest possible index *t* such that $z_t < n$, therefore increasable (the index *t* is called the turning point). The value of z_t is increased by 1 while the new value of z_i for $i \ge t$ is 1. The algorithm is as follows.

```
for i ← 0 to m do z_i ← 1 ;
repeat
        print out p_{z_i}, 1 ≤ i ≤ m ;
        t ← m ;
        while z_t = n do t ← t − 1 ;
        z_t ← z_t + 1;
        for i ← t + 1 to m do z_i ← 1
until t = 0.
```

We now prove that the algorithm has constant average delay property. Every step will be assigned to the current value of *t*; in this way the time complexity *T* is subdivided into *m* portions $T_1, T_2, \ldots, T_m$. In the process of a backtrack search and the update of elements, every portion T_i for $t \le i \le m$ increases by a constant amount. After the update, *i*th element does not change (moreover, the backtrack search does not reach it) during the next n^{m-i} variations (i.e., T_i does not increase). Therefore, on average, T_i increases by $O(1/n^{m-i})$. It follows that the average delay is, up to a constant,

$$\sum_{i=1}^{m} \frac{1}{n^{m-1}} = \frac{1}{n^m}\frac{n^{m+1}-1}{n-1} = O(1).$$

Subsets may be also represented in binary notation, where each “1” corresponds to the element from the subset. For example, subset {1,3,4} for $n = 5$ is represented as 11010. Thus, subsets correspond to integers written in the binary number system (i.e., counters) and to bitstrings, giving all possible information contents in a computer memory. A simple recursive algorithm for generating bitstrings is given in the work by Parberry [28]. A call to bitstring (*n*) produces all bitstrings of length *n* as follows:

```
procedure bitstring( m);
        if m = 0 then print out c_i;
          else c_m ← 0; bitstring(m − 1 );
            c_m ← 1; bitstring(m − 1 ) .
```

Given an integer n, it is possible to represent it as the sum of one or more positive integers (called parts) a_i that is, $n = x_1 + x_2 + \cdots + x_m$. This representation is called an *integer partition* if the order of parts is of no consequence. Thus, two partitions of an integer n are distinct if they differ with respect to the x_i they contain. For example, there are seven distinct partitions of the integer $5:\ 5,\ 4+1,\ 3+2,\ 3+1+1,\ 2+2+1,\ 2+1+1+1,\ 1+1+1+1+1$. If the order of parts is important then the representation of n as a sum of some positive integers is called *integer composition*. For example, integer compositions of 5 are the following:

$$5,\ 4+1,\ 1+4,\ 3+2,\ 2+3,\ 3+1+1,\ 1+3+1,\ 1+1+3,\ 2+2+1,$$
$$2+1+2,\ 1+2+2,\ 2+1+1+1,\ 1+2+1+1,\ 1+1+2+1,$$
$$1+1+1+2,\ 1+1+1+1+1.$$

Compositions of an integer n into m parts are representations of n in the form of the sum of exactly m positive integers. These compositions can be written in the form $x_1 + \cdots + x_m = n$, where $x_1 \geq 0, \ldots, x_m \geq 0$. We will establish the correspondence between integer compositions and either combinations or subsets, depending on whether or not the number of parts is fixed.

Consider a composition of $n = x_1 + \cdots + x_m$, where m is fixed or not fixed. Let $y_1, \ldots, y_m$ be the following sequence: $y_i = x_1 + \cdots + x_i,\ 1 \leq i \leq m$. Clearly, $y_m = n$. The sequence $y_1, y_2, \ldots, y_{m-1}$ is a subset of $\{1, 2, \ldots, n-1\}$. If the number of parts m is not fixed then compositions of n into any number of parts correspond to subsets of $\{1, 2, \ldots, n-1\}$. The number of such compositions is in this case $CM(n) = 2^{n-1}$. If the number of parts m is fixed then the sequence $y_1, \ldots, y_{m-1}$ is a combinations of $m-1$ out of $n-1$ elements from $\{1, \ldots, n-1\}$, and the number of compositions in question is $CO(m, n) = C(m-1, n-1)$. Each sequence $x_1 \ldots x_m$ can easily be obtained from $y_1, \ldots, y_m$ since $x_i = y_i - y_{i-1}$ (with $y_0 = 0$).

To design a loopless algorithm for generating integer compositions of n, one can use this relation between compositions of n and subsets of $\{1, 2, \ldots, n-1\}$, and the subset generation algorithm above.

1.3 LISTING COMBINATIONS

A (m,n)-combination out of $\{p_1, p_2, \ldots, p_n\}$ can be represented as a sequence $c_1, c_2, \ldots, c_m$, where $p_1 \leq c_1 < c_2 < \cdots < c_m \leq p_n$. Let $z_1, z_2, \ldots, z_m$ be the corresponding array of indices, that is, $c_i = p_{z_i}, 1 \leq i \leq m$. Then $1 \leq z_1 < z_2 < \cdots < z_m \leq n$, and therefore $z_i \leq n - m + i$ for $1 \leq i \leq m$. The number of (m,n)-combinations is binomial coefficient $C(m, n) = n!/(m!(n-m)!)$. In this section, we investigate generating the $C(m,n)$ (m,n)-combinations, in lexicographically ascending order. Various sequential algorithms have been given for this problem.

Comparisons of combination generation techniques are given in the works by Akl [1] and Payne and Ives [29]. Akl [1] reports algorithm by Misfud [23] to be the fastest while Semba [34] improved the speed of algorithm [23].

The sequential algorithm [23] for generating (m,n)-combinations determines the next combination by a backtrack search that finds an element c_t with the greatest possible index t such that $z_t < n - m + t$, therefore increasable (the index t is called the turning point). The new value of z_i for $i \geq t$ is $z_t + i - t + 1$.

The average delay of the algorithm is $O(n/(n-m))$ [34]. The delay is constant whenever $m = o(n)$. On the contrary, the average delay may be nonconstant in some cases (e.g., when $n - m = O(\sqrt{n})$). Semba [34] modified the algorithm by noting that there is no need to search for the turning point as it can be updated directly from one combination to another, and that there is no need to update the elements with indices between t and m if they do not change from one combination to another. If $z_t < n - m + t - 1$ then all elements in the next combination will be less that their appropriate maximal values and the turning point of the next combination will be index m. In this case, a total of $d = m - t + 1$ elements change their value in the next combination. Otherwise, that is, when $z_t = n - m + t - 1$, the new value for the turning point element becomes its maximal possible value $n - m + t$, elements between t and m remain unchanged (with their maximal possible values), and the turning point for the next combination is the element with index $t - 1$. Only one element is checked in this case. The following table gives values of t and d for (4,6)-combinations.

1234	1235	1236	1245	1246	1256	1345	1346	1356	1456	2345	2346	2356	2456	3456
$t = 4$	4	3	4	3	2	4	3	2	1	4	3	2	1	0
$d = 1$	1	2	1	1	3	1	1	1	4	1	1	1	1	

The algorithm [34] is coded in FORTRAN language using goto statements. Here we code it in PASCAL-like style.

```
z0 ← 1; t ← m;
for i ← 1 to m do z_i ← i;
repeat
      print out p_{z_i}, 1 ≤ i ≤ m;
      z_t ← z_t + 1;
      if z_t = n − m + t    then t ← t − 1
                            else for i = t + 1 to m do z_i ← z_t + i − t; t ← m
until t = 0.
```

The algorithm always does one examination to determine the turning point. We now determine the average number d of changed elements. For a fixed t, the number of (m,n)-combinations that have t as the turning point with $z_t < n - m + t - 1$ is $C(t, n - m + t - 2)$. This follows because $z_i = n - m + i$ when $i > t$ for each of these combinations while $z_1, z_2, \ldots, z_t$ can be any $(t, n - m + t - 2)$ -combination. The turning point element is always updated. In addition, $m - t$ elements whenever $z_t < n - m + t - 1$, which happens $C(t, n - m + t - 2)$ times. Therefore, the

total number of updated elements (in addition to the turning point) to generate all combinations is

$$\sum_{t=1}^{m}(m-t)C(t,n-m+t-2)=\sum_{j=0}^{m-1}jC(n-j-2,n-m-2)$$
$$=\frac{m}{n-m}C(n-m-1,n-1)-m$$
$$=\frac{m}{n}C(m,n)-m.$$

Thus, the algorithms updates, on the average, less than $m/n+1<2$ elements and therefore the average delay is constant for any m and $n(m\le n)$.

1.4 LISTING PERMUTATIONS

A sequence $p_1, p_2, \ldots, p_n$ of mutually distinct elements is a *permutation* of $S=\{s_1, s_2, \ldots, s_n\}$ if and only if $\{p_1, p_2, \ldots, p_n\}=\{s_1, s_2, \ldots, s_n\}=S$. In other words, an n-permutation is an ordering, or arrangement, of n given elements. For example, there are six permutations of the set $\{A, B, C\}$. These are ABC, ACB, BAC, BCA, CAB, and CBA.

Many algorithms have been published for generating permutations. Surveys and bibliographies on the generation of permutations can be found in the Ord-Smith [27] and Sedgewick [31] [27,31]. Lexicographic generation presented below is credited to L.L. Fisher and K.C. Krause in 1812 by Reingold et al. [30].

Following the backtrack method, permutations can be generated in lexicographic order as follows. The next permutation of $x_1x_2\ldots x_n$ is determined by scanning from right to left, looking for the rightmost place where $x_i<x_{i+1}$ (called again the turning point). By another scan, the smallest element x_j that is still greater than x_i is found and interchanged with x_i. Finally, the elements $x_{i+1}, \ldots, x_n$ (which are in decreasing order) are reversed. For example, for permutation 3, 9, 4, 8, 7, 6, 5, 2, 1, the turning point $x_3=4$ is interchanged with $x_7=5$ and 8, 7, 6, 4, 2, 1 is reversed to give the new permutation 3, 9, 5, 1, 2, 4, 6, 7, 8. The following algorithm is the implementation of the method for generating permutations of $\{p_1, p_2, \ldots, p_n\}$. The algorithm updates the indices z_i (such that $x_i=p_{z_i}$,), $1\le i\le n$.

for $i \leftarrow 0$ **to** n **do** $z_i \leftarrow i$;
$i \leftarrow 1$;
while $i \neq 0$ **do** {
    print out p_{z_i}, $1 \le i \le n$;
    $i \leftarrow n-1$;
    while $z_i \ge z_{i+1}$ **do** $i \leftarrow i-1$;
    $j \leftarrow n$;
    while $z_i \ge z_j$ **do** $j \leftarrow j-1$;

$ch \leftarrow z_i$; $z_i \leftarrow z_j$; $z_j \leftarrow ch$;
$v \leftarrow n$; $u \leftarrow i+1$;
while $v>u$ **do** $\{ch \leftarrow z_v$; $z_v \leftarrow z_u$; $z_u \leftarrow ch$; $v \leftarrow v-1$;
$u \leftarrow u+1\}\}$.

We prove that the algorithm has constant average delay property. The time complexity of the algorithm is clearly proportional to the number of tests $z_i \geq z_{i+1}$ in the first *while* inside loop. If *i*th element is the turning point, the array $z_{i+1}, \ldots, z_n$ is decreasing and it takes $(n-1)$ tests to reach z_i. The array $z_1 z_2 \ldots z_i$ is a (*m,n*)-permutation. It can be uniquely completed to *n*-permutation $z_1 z_2 \ldots z_n$ such that $z_{i+1} > \cdots > z_n$. Although only these permutations for which $z_i < z_{i+1}$ are valid for z_i to be the turning point, we relax the condition and artificially increase the number of tests in order to simplify the proof. Therefore for each i, $1 \leq i \leq n-1$ there are at most $P(i,n) = n(n-1)\cdots(n-i+1)$ arrays such that z_i is the turning point of *n*-permutation $z_1 z_2 \ldots z_n$. Since each of them requires $n-i$ tests, the total number of tests is at most $\sum_{i=1}^{n-1} P(i,n)(n-i) = \sum_{i=1}^{n-1} (n(n-1)\cdots(n-i+1)(n-i)) = \sum_{i=1}^{n-1} n!/(n-i-1)! = n! \sum_{j=0}^{n-2} 1/j!$. Since $j! = 2 \cdot 3 \cdots j > 2 \times 2 \cdots \times 2 = 2^{j-1}$, the average number of tests is $<2 + \sum_{j=2}^{n-2} 1/(2^{j-1}) = 2 + 1/2 + 1/4 + \ldots < 3$. Therefore the algorithm has constant delay property. It is proved [27] that the algorithm performs about $1.5n!$ interchanges.

The algorithm can be used to generate the permutations with repetitions. Let $n_1, n_2, \ldots, n_k$ be the multiplicities of elements $p_1, p_2, \ldots, p_k$, respectively, such that the total number of elements is $n_1 + n_2 + \cdots + n_k = n$. The above algorithm uses no arithmetic with indices z_i and we can observe that the same algorithm generates permutations with repetitions if the initialization step (the first instruction, i.e., **for** loop) is replaced by the following instructions that find the first permutation with repetitions.

$n \leftarrow 0$; $z_0 \leftarrow 0$;
for $i \leftarrow 1$ **to** k **do**
for $j \leftarrow 1$ **to** n_i **do** $\{n \leftarrow n+1$; $z_n \leftarrow j\}$;

Permutations of combinations (or (*m,n*)-permutations) can be found by generating all (*m,n*)-combinations and finding all (*m,m*)-permutations for each (*m,n*)-combination. The algorithm is then obtained by combining combination and permutation generating algorithms. In the standard representation of (*m,n*)-permutations as an array $x_1 x_2 \ldots x_m$, the order of instances is not lexicographic. Let $c_1 c_2 \ldots c_m$ be the corresponding combination for permutation $x_1 x_2, \ldots, x_m$, that is, $c_1 < c_2 < \cdots < c_m$ and $\{c_1, c_2, \ldots, c_m\} = \{x_1, x_2, \ldots, x_m\}$. Then we can observe that the obtained order of generating (*m,n*)-permutations is lexicographic if they are represented as an array of $2m$ elements $c_1 c_2 \ldots c_m x_1 x_2 \ldots x_m$, composed of corresponding (*m,n*)-combination followed by the (*m,n*)-permutation. In other words, the order is lexicographic if corresponding combinations are compared before comparing permutations.

1.5 LISTING EQUIVALENCE RELATIONS OR SET PARTITIONS

An equivalence relation of the set $Z = \{p_1, \ldots, p_n\}$ consists of classes $\pi_1, \pi_2, \ldots, \pi_k$ such that the intersection of every two classes is empty and their union is equal to Z. Equivalence relations are often referred to as set partitions. For example, let $Z = \{A, B, C\}$. Then there are four equivalence relations of Z : $\{\{A, B, C\}\}$, $\{\{A, B\}\{C\}\}$, $\{\{A, C\}\{B\}\}$, $\{\{A\}, \{B, C\}\}$, and $\{\{A\}, \{B\}, \{C\}\}$.

Equivalence relations of Z can be conveniently represented by codewords $c_1c_2 \ldots c_n$ such that $c_i = j$ if and only if element p_i is in class π_j. Because equivalence classes may be numbered in various ways ($k!$ ways), such codeword representation is not unique. For example, set partition $\{\{A, B\}\{C\}\}$ is represented with codeword 112 while the same partition $\{\{C\}\{A, B\}\}$ is coded as 221.

In order to obtain a unique codeword representation for given equivalence relation, we choose lexicographically minimal one among all possible codewords. Clearly $c_1 = 1$ since we can choose π_1 to be the class containing p_1. All elements that are in π_1 are also coded with 1. The class containing element that is not in π_1 and has the minimal possible index is π_2 and so on. For example, let $\{\{C, D, E\}, \{B\}, \{A, F\}\}$ be a set partition of $\{A, B, C, D, E, F\}$. The first equivalence class is $\{A, F\}$, the second is $\{B\}$, and the third is $\{C, D, E\}$. The corresponding codeword is 123331.

A codeword $c_1 \ldots c_n$ represents an equivalence relation of the set Z if and only if $c_1 = 1$ and $1 \leq c_r \leq g_{r-1} + 1$ for $2 \leq r \leq n$, where $c_i = j$ if i is in π_j, and $g_r = \max(c_1, \ldots, c_r)$ for $1 \leq r \leq n$. This follows from the definition of lexicographically minimal codeword. Element p_t is either one of the equivalence classes with some other element $p_i (i<t)$ in which case c_t receives one of existing codes assigned to elements $p_1, p_2, \ldots, p_{t-1}$ or in none of previous classes, in which case it starts a new class with index one higher than previously maximal index.

Sequential algorithms [9,12,25,32] generate set partitions represented by codewords in lexicographic order. The next equivalence relation is found from the current one by a backtracking or recursive procedure in all known sequential generating techniques that maintain the lexicographic order of elements; in both cases an increasable element (one for which $x_j \leq g_j - 1$ is satisfied) with the largest possible index t is found ($t \leq n - 2$); we call this element the *turning point*. For example, the turning point of the equivalence relation 1123 is the second element ($t = 2$).

A list of codewords and corresponding partitions for $n = 4$ and $Z = \{A, B, C, D\}$ is, in lexicographic order, as follows:

1111 = {{A, B, C, D}}, 1112 = {{A, B, C}, {D}}, 1121 = {{A, B, D}, {C}},
1122 = {{A, B}, {C, D}}, 1123 = {{A, B}, {C}, {D}},
1211 = {{A, C, D}, {B}}, 1212 = {{A, C}, {B, D}},
1213 = {{A, C}, {B}, {D}}, 1221 = {{A, D}, {B, C}},
1222 = {{A}, {B, C, D}}, 1223 = {{A}, {B, C}, {D}}, 1231 = {{A, D}, {B}, {C}},
1232 = {{A}, {B, D}, {C}}, 1233 = {{A}, {B}, {C, D}}, 1234 = {{A}, {B}, {C}, {D}}.

We present an iterative algorithm from the work by Djokić et al. [9] for generating all set partitions in the codeword representation. The algorithm follows backtrack method for finding the largest r having an increasable c_r, that is, $c_r < g_{r-1} + 1$.

```
program setpart( n);
r ← 1; c1 ← 1; j ← 0; b0 ← 1; n1 ← n − 1;
repeat
    while r<n1 do {r ← r + 1; cr ← 1; j ← j + 1; bj ← r};
    for i ← 1 to n − j do {cn ← i; print out c1, c2, . . . , cn};
    r ← bj; cr ← cr + 1;
    if cr>r − j then j ← j − 1
until r = 1
```

In the presented iterative algorithm b_j is the position where current position r should backtrack after generating all codewords beginning with $c_1, c_2, \ldots, c_{n-1}$. Thus the backtrack is applied on $n-1$ elements of codeword while direct generation of the last element in its range speeds the algorithm up significantly (in most set partitions the last element in the codeword is increasable). An element of b is defined whenever $g_r = g_{r-1}$, which is recognized by either $c_r = 1$ or $c_r > r - j$ in the algorithm. It is easy to see that the relation $r = g_{r-1} + j$ holds whenever j is defined. For example, for the codeword $c = 111211342$ we have $g = 111222344$ and $b = 23569$. Array b has $n - g_n = 9 - 4 = 5$ elements.

In the algorithm, backtrack is done on array b and finds the increasable element in constant time; however, updating array b for future backtrack calls is not a constant time operation (**while** loop in the program). The number of backtrack calls is B_{n-1} (recall that B_n is the number of set partitions over n elements).

The algorithm has been compared with other algorithms that perform the same generation and it was shown to be the fastest known iterative algorithm. A recursive algorithm is proposed in the work by Er [12]. The iterative algorithm is faster than recursive one on some architectures and slower on other [9].

The constant average time property of the algorithm can be shown as in the work by Semba [32]. The backtrack step returns to position r exactly $B_r - B_{r-1}$ times, and each time it takes $n - r + 1$ for update (**while** loop), for $2 \leq r \leq n-1$. Therefore, up to a constant, the backtrack steps require $(B_2 - B_1)(n-1) + (B_3 - B_2)(n-2) + \cdots + (B_{n-1} - B_{n-2})2 < B_2 + B_3 + \cdots + B_{n-2} + 2B_{n-1}$. The update of nth element is performed $B_n - B_{n-1}$ times. Since $B_{i+1} > 2B_i$, the average delay, up to a constant, is bounded by

$$\frac{B_n + B_{n-1} + \cdots + B_2}{B_n} < 1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{n-2}} < 2.$$

1.6 GENERATING INTEGER COMPOSITIONS AND PARTITIONS

Given an integer n, it is possible to represent it as the sum of one or more positive integers (called *parts*) x_i, that is, $n = x_1 + x_2 + \cdots + x_m$. This representation is called

an *integer partition* if the order of parts is of no consequence. Thus, two partitions of an integer n are distinct if they differ with respect to the x_i they contain. For example, there are seven distinct partitions of the integer 5:

$$5,\ 4+1,\ 3+2,\ 3+1+1,\ 2+2+1,\ 2+1+1+1,\ 1+1+1+1+1.$$

In the *standard* representation, a partition of n is given by a sequence $x_1, \ldots, x_m$, where $x_1 \geq x_2 \geq \cdots \geq x_m$, and $x_1 + x_2 + \cdots + x_m = n$. In the sequel x will denote an arbitrary partition and m will denote the number of parts of x (m is not fixed). It is sometimes more convenient to use a *multiplicity* representation for partitions in terms of a list of the distinct parts of the partition and their respective multiplicities. Let $y_1 > \cdots > y_d$ be all distinct parts in a partitions, and $c_1, \ldots, c_d$ their respective (positive) multiplicities. Clearly $c_1 y_1 + \cdots + c_d y_d = n$.

We first describe an algorithm for generating integer compositions of n into any number of parts and in lexicographic order. For example, compositions of 4 in lexicographic order are the following: $1+1+1+1,\ 1+1+2,\ 1+2+1,\ 1+3,\ 2+1+1,\ 2+2,\ 3+1,\ 4$. Let $x_1 \ldots x_m$, where $x_1 + x_2 + \cdots + x_m = n$ be a composition. The next composition, following lexicographic order, is $x_1, \ldots, x_{m-1} + 1, 1, \ldots, 1(x_m - 1$ 1s). In other words, the next to last part is increased by one and the $x_m - 1$, 1s are added to complete the next composition. This can be coded as follows:

```
program composition( n);
m ← 1; x_1 ← n;
repeat
    for j ← 1 to m do print out x_1, x_2, ..., x_m;
    m ← m − 1; x_m ← x_m + 1;
    for j ← 1 to x_{m+1} − 1 do {m ← m + 1; x_m ← 1}
until m = n.
```

In antilexicographic order, a partition is derived from the previous one by subtracting 1 from the rightmost part greater than 1, and distributing the remainder as quickly as possible. For example, the partitions following $9+7+6+1+1+1+1+1+1$ is $9+7+5+5+2$. In standard representation and antilexicographic order, the next partition is determined from current one $x_1 x_2 \ldots x_m$ in the following way. Let h be the number of parts of x greater than 1, that is, $x_i > 1$ for $1 \leq i \leq h$, and $x_i = 1$ for $h < i \leq m$. If $x_m > 1$ (or $h = m$) then the next partition is $x_1, x_2, \ldots, x_{m-1}, x_m - 1, 1$. Otherwise (i.e., $h < m$), the next partition is obtained by replacing $x_h, x_{h+1} = 1, \ldots, x_m = 1$ with $(x_h - 1), (x_h - 1), \ldots, (x_h - 1), d$, containing c elements, where $0 < d \leq x_h - 1$ and $(x_h - 1)(c - 1) + d = x_h + m - h$.

We describe two algorithms from the work by Zoghbi and Stojmenovic [43] for generating integer partitions in standard representation and prove that they have constant average delay property. The first algorithm, named ZS1, generates partitions in antilexicographic order while the second, named ZS2, uses lexicographic order.

Recall that h is the index of the last part of partition, which is greater than 1 while m is the number of parts. The major idea in algorithm ZS1 is coming from the

observation on the distribution of x_h. An empirical and theoretical study shows that $x_h = 2$ has growing frequency; it appears in 66 percent of cases for $n = 30$ and in 78 percent of partitions for $n = 90$ and appears to be increasing with n. Each partition of n containing a part of size 2 becomes, after deleting the part, a partition of $n - 2$ (and *vice versa*). Therefore the number of partitions of n containing at least one part of size 2 is $P(n - 2)$. The ratio $P(n - 2)/P(n)$ approaches 1 with increasing n. Thus, almost all partitions contain at least one part of size 2. This special case is treated separately, and we will prove that it suffices to argue the constant average delay of algorithm ZS1. Moreover, since more than 15 instructions in known algorithms that were used for all cases are replaced by 4 instructions in cases of at least one part of size 2 (which happens almost always), the speed up of about four times is expected even before experimental measurements. The case $x_h > 2$ is coded in a similar manner as earlier algorithm, except that assignments of parts that are supposed to receive value 1 is avoided by an initialization step that assigns 1 to each part and observation that inactive parts (these with index $> m$) are always left at value 1. The new algorithm is obtained when the above observation is applied to known algorithms and can be coded as follows.

```
Algorithm ZS1
for i ← 1 to n do x_i ← 1;
x_1 ← n; m ← 1; h ← 1; output x_1;
while x_1 ≠ 1 do {
    if x_h = 2 then {m ← m + 1; x_h ← 1; h ← h − 1}
    else {r ← x_h − 1; t ← m − h + 1; x_h ← r;
            while t ≥ r do {h ← h + 1; x_h ← r; t ← t − r}
            if t = 0 then m ← h
            else m ← h + 1
                if t>1 then {h ← h + 1; x_h ← t}}
    output x_1, x_2, . . . , x_m}}.
```

We now describe the method for generating partitions in lexicographic order and standard representation of partitions. Each partition of n containing two parts of size 1 (i.e., $m - h > 1$) becomes, after deleting these parts, a partition of $n - 2$ (and *vice versa*). Therefore the number of integer partitions containing at least two parts of size 1 is $P(n - 2)$, as in the case of previous algorithm. The coding in this case is made simpler, in fact with constant delay, by replacing first two parts of size 1 by one part of size 2. The position h of last part > 1 is always maintained. Otherwise, to find the next partition in the lexicographic order, an algorithm will do a backward search to find the first part that can be increased. The last part x_m cannot be increased. The next to last part x_{m-1} can be increased only if $x_{m-2} > x_{m-1}$. The element that will be increased is x_j where $x_{j-1} > x_j$ and $x_j = x_{j+1} = \ldots = x_{m-1}$. The jth part becomes $x_j + 1$, h receives value j, and appropriate number of parts equal to 1 is added to complete the sum to n. For example, in the partition $5 + 5 + 5 + 4 + 4 + 4 + 1$ the leftmost 4 is increased, and the next partition is $5 + 5 + 5 + 5 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$. The following is a code of appropriate algorithm ZS2:

Algorithm ZS2

for $i \leftarrow 1$ **to** n **do** $x_i \leftarrow 1$; output $x_i,\ i = 1, 2, \ldots, n$;
$x_0 \leftarrow 1$; $x_1 \leftarrow 2$; $h \leftarrow 1$; $m \leftarrow n - 1$; output $x_i,\ i = 1, 2, \ldots, m$;
while $x_1 \neq n$ **do** {
    if $m - h > 1$ **then** {$h \leftarrow h + 1$; $x_h \leftarrow 2$; $m \leftarrow m - 1$}
        else {$j \leftarrow m - 2$;
            while $x_j = x_{m-1}$ **do** {$x_j \leftarrow 1$; $j \leftarrow j - 1$};
            $h \leftarrow j + 1$; $x_h \leftarrow x_{m-1} + 1$;
            $r \leftarrow x_m + x_{m-1}(m - h - 1)$; $x_m \leftarrow 1$;
            if $m - h > 1$ **then** $x_{m-1} \leftarrow 1$;
            $m \leftarrow h + r - 1$;
    output $x_1,\ x_2, \ldots, x_m$}.

We now prove the constant average delay property of algorithms ZS1 and ZS2.

Theorem 1 *Algorithms ZS1 and ZS2 generate unrestricted integer partitions in standard representation with constant average delay, exclusive of the output.*

Proof. Consider part $x_i \geq 3$ in the current partition. It received its value after a backtracking search (starting from last part) was performed to find an index $j \leq i$, called the turning point, that should change its value by 1 (increase/decrease for lexicographic/antilexicographic order) and to update values x_i for $j \leq i$. The time to perform both backtracking searches is $O(r_j)$, where $r_j = n - x_1 - x_2 - \cdots - x_j$ is the remainder to distribute after first j parts are fixed. We decide to charge the cost of the backtrack search evenly to all "swept" parts, such that each of them receives constant $O(1)$ time. Part x_i will be changed only after a similar backtracking step "swept" over ith part or recognized ith part as the turning point (note that ith part is the turning point in at least one of the two backtracking steps). There are $\mathrm{RP}(r_i, x_i)$ such partitions that keep all x_j intact. For $x_i \geq 3$ the number of such partitions, is $\geq r_i^2/12$. Therefore the average number of operations that are performed by such part i during the "run" of $\mathrm{RP}(r_i, x_i)$, including the change of its value, is $O(1)/RP(r_i, x_i) \leq O(1)/r_i^2 = O(1/r_i^2) < q_i/r_i^2$, where q_i is a constant. Thus the average number of operations for all parts of size ≥ 3 is $\leq q_1/r_1^2 + q_2/r_2^2 + \cdots + q_s/r_s^2 \leq q(1/r_1^2 + \cdots + 1/r_s^2) < q(1/n^2 + 1/(n-1)^2 + \cdots + 1/1^2) < 2q$ (the last inequality can be obtained easily by applying integral operation on the last sum), which is a constant. The case that was not counted in is when $x_i \leq 2$. However, in this case both algorithms ZS1 and ZS2 perform constant number of steps altogether on all such parts. Therefore the algorithm has overall constant time average delay. ■

The performance evaluation of known integer partition generation methods is performed in the work by Zoghbi and Stojmenovic [43]. The results show clearly that both algorithms ZS1 and ZS2 are superior to all other known algorithms that generate partitions in the standard representation. Moreover, both algorithms SZ1 and ZS2 were even faster than any algorithm for generating integer partitions in the multiplicity representation.

1.7 LISTING *t*-ARY TREES

The t-ary trees are data structures consisting of a finite set of n nodes, which either is empty ($n = 0$) or consists of a root and t disjoint children. Each child is a t-ary subtree, recursively defined. A node is the parent of another node if the latter is a child of the former. For $t = 2$, one gets the special case of rooted binary trees, where each node has a left and a right child, where each child is either empty or is a binary tree. A computer representation of t-ary trees with n nodes is achieved by an array of n records, each record consisting of several data fields, t pointers to children and a pointer to the parent. All pointers to empty trees are nil. The number of t-ary trees with n nodes is $B(n, t) = (tn)!/(n!(tn - n)!)/((t - 1)n + 1)$ (cf. [19,42]).

If the data fields are disregarded, the combinatorial problem of generating binary and, in general, t-ary trees is concerned with generating all different shapes of t-ary trees with n nodes in some order. The lexicographic order of trees refers to the lexicographic order of the corresponding tree sequences. There are over 30 ingenious generating algorithms for generating binary and t-ary trees. In most references, tree sequences are generated in lexicographic order. Each of these generation algorithms causes trees to be generated in a particular order. Almost all known sequential algorithms generate tree sequences, and the inclusion of parent–child relations requires adding a decoding procedure, usually at a cost of greatly complicating the algorithm and/or invalidating the run time analysis. Exceptions are the works by Akl et al. [4] and Lucas et al. [22].

Parent array notation [4] provides a simple sequential algorithm that extends trivially to add parent–children relations. Consider a left-to-right breadth first search (BFS) labeling of a given tree. All nodes are labeled by consecutive integers $1, 2, \ldots, n$ such that nodes on a lower level are labeled before those on a higher level, while nodes on the same level are labeled from left to right. Children are ordered as $L = 1, \ldots, t$. Parent array $p_1, \ldots, p_n$ can be defined as follows: $p_1 = 1$, $p_i = t(j - 1) + L + 1$ if i is the Lth child of node j, $2 \leq i \leq n$, and it has property $p_{i-1} < p_i \leq ti - t + 1$ for $2 \leq i \leq n$. For example, the binary tree on Figure 1.1 has parent array 1, 3, 4, 5, 7, 8; the 3-ary tree on Figure 1.1 has parent array 1, 2, 3, 4, 8, 10, 18.

The algorithm [4] for generating all parent arrays is extended from the work by Zaks [42] to include parent–children relations (the same sequence in the works by Zaks [42] and Akl et al. [4] refers to different trees). The Lth children of node i is denoted by $child_{i,L}$ (it is 0 if no such child exist) while $parent_i$ denotes the parent

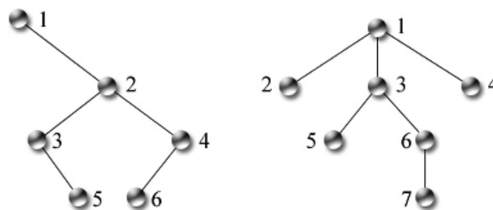

FIGURE 1.1 Binary tree 1, 3, 4, 5, 7, 8 and ternary tree 1, 2, 3, 4, 8, 10, 18.

node of i. Integer division is used throughout the algorithm. The algorithm generates tree sequences in lexicographic order.

for $i \leftarrow 1$ **to** n **do**
  for $L \leftarrow 1$ **to** t **do** $child_{i,L} \leftarrow 0$;
for $i \leftarrow 1$ **to** n **do** $\{p_i \leftarrow i;\ parent_i \leftarrow (i-2)/t+1;$
    $L \leftarrow p_i - 1 - t(parent_i - 1);\ child_{(i-2)/t+1,L} \leftarrow i\}$
repeat
  report t-ary tree;
  $j \leftarrow n$;
  while $p_j < 2j-1$ and $j>1$ **do** $\{i \leftarrow parent_j;$
    $L \leftarrow p_i - 1 - t(i-1);\ child_{i,L} \leftarrow 0;\ j \leftarrow j-1\}$
  $p_j \leftarrow p_j + 1$;
  for $i \leftarrow j+1$ **to** n **do** $p_i \leftarrow p_{i-1} + 1$;
  for $i \leftarrow j$ **to** n **do** $\{k \leftarrow (p_i - 2)/t + 1;\ parent_i \leftarrow k;$
    $L \leftarrow p_i - 1 - t(k-1);\ child_{k,L} \leftarrow i\}$
until $p_1 = 2$.

Consider now generating t-ary trees in the children array notation. A tree is represented using a children array $c_1 c_2, c_3, \ldots, c_{tn}$ as follows:

- The jth children of node i is stored in $c_{(i-1)t+j+1}$ for $1 \leq i \leq n-1$ and $1 \leq j \leq t$; missing children are denoted by 0. The array is, for convenience, completed with $c_1 = 1$ and $c_{(n-1)t+2} = \cdots = c_{nt} = 0$ (node n has no children).

For example, the children array notations for trees in FIGURE 1.1 are 102340560000 and 12340005060000007000. Here we give a simple algorithm to generate children array tree sequences, for the case of t-ary trees (generalized from the work by Akl et al. [4] that gives corresponding generation of binary trees).

The rightmost element of array c that can be occupied by an integer $j>0$, representing node j, is obtained when j is tth child of node $j-1$, that is, it is $c_{(j-1)t+1}$. We say that an integer j is mobile if it is not in $c_{(j-1)t+1}$ and all (nonzero) integers to its right occupy their rightmost positions. A simple sequential algorithm that uses this notation to generate all t-ary trees with n nodes is given below. If numerical order $0 < 1 < \cdots < n$ is assumed, the algorithm generates children array sequences in antilexicographic order. Alternatively, the order may be interpreted as lexicographic if $0, 1, \cdots, n$ are treated as symbols, ordered as "1" < "2" < ... < "n" < "0". Numeric lexicographic order may be obtained if 0 is replaced by a number larger than n (the algorithm should always report that number instead of 0).

for $i \leftarrow 1$ **to** n **do** $c_i \leftarrow i$;
for $i \leftarrow n+1$ **to** tn **do** $c_i \leftarrow 0$;
repeat
  print out $c_1, \ldots, c_{tn}$;
  $i \leftarrow (n-1)t$;

while $\left(c_i = 0 \text{ or } c_i = \frac{k-1}{t} + 1\right)$ and $(i > 1)$ **do** $i \leftarrow i - 1$;
$c_{i+1} \leftarrow c_i$;
$c_i \leftarrow 0$;
for $k \leftarrow 1$ **to** $n - c_{i+1}$ **do** $c_{i+k+1} \leftarrow c_{i+k} + 1$;
for $k \leftarrow i + n - c_{i+1} + 2$ **to** $(n-1)t + 1$ **do** $c_k \leftarrow 0$
until $i = 1$.

We leave as an exercise to design an algorithm to generate well-formed parenthesis sequences. This can be done by using the relation between well-formed parenthesis sequences and binary trees in the children representation, and applying the algorithm given in this section.

An algorithm for generating B-trees is described in the work by Gupta et al. [16]. It is based on backtrack search, and produces B-trees with worst case delay proportional to the output size. The order of generating B-trees becomes lexicographic if B-trees are coded as a B-tree sequence, defined in [5]. The algorithm [16] has constant expected delay in producing next B-tree, exclusive of the output, which is proven in the work by Belbaraka and Stojmenovic [5]. Using a decoding procedure, an algorithm that generates the B-tree data structure (meaning that the parent–children links are established) from given B-tree sequence can be designed, with constant average delay.

1.8 LISTING SUBSETS AND BITSTRINGS IN A GRAY CODE ORDER

It is sometimes desirable to generate all instances of a combinatorial object in such a way that successive instances differ as little as possible. An order of all instances that minimizes the difference between any two neighboring instances is called *minimal change order*. Often the generation of objects in minimal change order requires complicated and/or computationally expensive procedures. When new instances are generated with the least possible changes (by a single insertion of an element, single deletion or single replacement of one element by another, interchange of two elements, updating two elements only, etc.), corresponding sequences of all instances of a combinatorial objects are refereed to as *Gray codes*. In addition, the same property must be preserved when going from the last to first sequence. In most cases, there is no difference between minimal change and Gray code orders. They may differ when for a given combinatorial object there is no known algorithm to list all instances in Gray code order. The best existing algorithm (e.g., one in which two instances differ at two positions whereas instances may differ in one position only) then is referred to achieving minimal change order but not in Gray code order.

We describe a procedure for generating subsets in binary notation, which is equivalent to generating all bitstrings of given length. It is based on a backtrack method and sequence comparison rule. Let $e_1 = 0$ and $e_i = x_1 + x_2 + \cdots + x_{i-1}$ for $1 < i \leq n$. Then the sequence that follows $x_1 x_2 \ldots x_n$ is $x_1 x_2 \ldots x_{i-1} x'_i x_{i+1} \ldots x_n$, where i is the largest index such that $e_i + x_i$ is even and $'$ is complement function

$(0' = 1,\ 1' = 0$; also $x' = x + 1 \bmod 2)$.

```
read( n);
for i ← 0  to  n  do {x_i ← 0; e_i ← 0};
repeat
        print out x_1, x_2, ..., x_n;
        i ← n;
        while x_i + e_i is odd  do i ← i − 1;
        x_i ← x'_i;
        for j ← i + 1  to  n  do e_j ← e'_j
until i = 0 .
```

The procedure has $O(n)$ worst case delay and uses no large integers. We will prove that it generates Gray code sequences with constant average delay. The element x_i changes 2^{i-1} times in the algorithm, and each time it makes $n - i + 1$ steps back and forth to update x_i. Since the time for each step is bounded by a constant, the time to generate all Gray code sequences is $\sum_{i=1}^{n} c2^{i-1}(n - i + 1)$. The average delay is obtained when the last number is divided by the number of generated sequences 2^n, and is therefore

$$c\sum_{i=1}^{n} 2^{-n+i-1}(n - i + 1) = c\sum_{i=1}^{n} 2^{-i}i = c\left(2 - \frac{n}{2^n} - \frac{1}{2^{n-1}}\right) < 2c.$$

An algorithm for generating subsets in the binary notation in the binary reflected Gray code that has constant delay in the worst case is described in the work by Reingold et al. [30]. Efficient loopless algorithms for generating k-ary trees are described in the Xiang et al. [41].

1.9 GENERATING PERMUTATIONS IN A MINIMAL CHANGE ORDER

In this section we consider generating the permutations of $\{p_1, p_2, \ldots, p_n\}$ $(p_1 < \cdots < p_n)$ in a minimum change order. We present one that is based on the idea of adjacent transpositions, and is independently proposed by Johnson [18] and Trotter [39]. It is then simplified by Even [14]. In the work by Even [14], a method by Ehrlich is presented, which has constant delay. The algorithm presented here is a further modification of the technique, also having constant delay, and suitable as a basis for a parallel algorithm [36].

The algorithm is based on the idea of generating the permutations of $\{p_1, p_2, \ldots, p_n\}$ from the permutations of $\{p_1, p_2, \ldots, p_{n-1}\}$ by taking each such permutation and inserting p_n in all n possible positions of it. For example, taking the permutation $p_1\, p_2 \ldots p_{n-1}$ of $\{p_1, p_2, \ldots, p_{n-1}\}$ we get n permutations of $\{p_1, p_2, \ldots, p_n\}$ as follows:

$$
\begin{array}{l}
p_1\ p_2 \cdots p_{n-2}\ p_{n-1}\ p_n \\
p_1\ p_2 \cdots p_{n-2}\ p_n\ \ p_{n-1} \\
p_1\ p_2 \cdots p_n\ \ p_{n-2}\ p_{n-1} \\
\quad\vdots \\
p_n\ p_1 \cdots p_{n-3}\ p_{n-2}\ p_{n-1}.
\end{array}
$$

The nth element sweeps from one end of the $(n-1)$ -permutation to the other by a sequence of adjacent swaps, producing a new n-permutation each time. Each time the nth element arrives at one end, a new $(n-1)$ -permutation is needed. The $(n-1)$-permutations are produced by placing the $(n-1)$th element at each possible position within an $(n-2)$ -permutation. That is, by applying the algorithm recursively to the $(n-1)$ elements.

The first permutation of the set $\{p_1, p_2, \ldots, p_n\}$ is $p_1, p_2, \ldots, p_n$. Assign a *direction* to every element, denoted by an arrow above the element. Initially all arrows point to the left. Thus if the permutations of $\{p_1, p_2, p_3, p_4\}$ are to be generated, we would have

$$\overleftarrow{p}_1\ \overleftarrow{p}_2\ \overleftarrow{p}_3\ \overleftarrow{p}_4.$$

Now an element is said to be *mobile* if its direction points to a smaller adjacent neighbor. In the above example, p_2, p_3 and p_4 are mobile, while in

$$\vec{p}_3\ \overleftarrow{p}_2\ \overleftarrow{p}_1\ \vec{p}_4$$

only p_3 is mobile. The algorithm is as follows:

While there are mobile elements **do**
(i) find the largest mobile element; call it p_m
(ii) reverse the direction of all elements larger than p_m
(iii) switch p_m with the adjacent neighbor to which its direction points
endwhile.

The straightforward implementation of the algorithm leads to an algorithm that exhibits a linear time delay. The algorithm is modified to achieve a constant delay. After initial permutation, the following steps are then repeated until termination:

1. Move element p_n to the left, by repeatedly exchanging it with its left neighbor, and do (i) and (ii) in the process.
2. Generate the next permutation of $\{p_1, p_2, \ldots, p_{n-1}\}$ (i.e., do step (iii)).
3. Move element p_n to the right, by repeatedly exchanging it with its right neighbor, and do (i) and (ii) in the process.

4. Generate the next permutation of $\{p_1, p_2, \ldots, p_{n-1}\}$ (i.e., do step (iii)).

For example, permutations of {1, 2, 3, 4} are generated in the following order:

1234, 1243, 1423, 4123	move element 4 to the left
4132	132 is the next permutation of 123, with 3 moving to the left
1432, 1342, 1324	move 4 to the right
3124	312 is the next permutation following 132, with 3 moving to the left
3142, 3412, 4312	4 moves to the left
4321	321 is the next permutation following 312; 2 in 12 moves to the left
3421, 3241, 3214	4 moves to the right
2314	231 follows 321, where 3 moves to the right
2341, 2431, 4231	4 moves to the left
4213	213 follows 231, 3 moved to the right
2413, 2143, 2134	4 moves to the right.

The constant delay is achieved by observing that the mobility of p_n has a regular pattern (moves $n-1$ times and then some other element moves once). It takes $n-1$ steps to move p_n to the left or right while (i), (ii), and (iii) together take $O(n)$ time. Therefore, if steps (i), (ii), and (iii) are performed after p_n has already finished moving in a given direction, the algorithm will have constant average delay. If the work in steps (i) and (ii) [step (iii) requires constant time] is evenly distributed between consecutive permutations, the algorithm will achieve constant worst case delay. More precisely, finding largest mobile element takes $n-1$ steps, updating directions takes also $n-1$ steps. Thus it suffices to perform two such steps per move of element p_n to achieve constant delay per permutation.

The current permutation is denoted $d_1, d_2, \ldots, d_n$. The direction is stored in a variable a, where $a_i = -1$ for left and $a_i = 1$ for right direction. When two elements are interchanged, their directions are also interchanged implicitly. The algorithm terminates when no mobile element is found. For algorithm conciseness, we assume that two more elements p_0 and p_{n+1} are added such that $p_0 < p_1 < \ldots < p_n < p_{n+1}$. Variable i is used to move p_n from right to left ($i = n, n-1, \ldots, 2$) or from left to right ($i = 1, 2, \ldots, n-1$). The work in steps (i) and (ii) is done by two "sweeping" variables l (from left to right) and r (from right to left). They update the largest mobile elements dlm and drm, respectively, and their indices lm and rm, respectively, that they detect in the sweep. When they "meet" ($l = r$ or $l = r-1$) the largest mobile element dlm and its index lm is decided, and the information is broadcast (when $l>r$) to all other elements who use it to update their directions. Obviously the

sweep of variable *i* coincides with either the sweep of *l* or sweep of *r*. For clarity, the code below considers these three sweeps separately. The algorithm works correctly for $n>2$.

```
procedure output;
        { for s ← 1 to n do write( d[s]); writeln}

procedure exchange ( c, b: integer);
        { ch ← d[c + b]; d[c + b] ← d[c]; d[c] ← ch; ch ← a[c + b];
a[c + b] ← a[c]; a[c] ← ch };

procedure updatelm; {
        l ← l + 1; if (d[l] = p_n) or (d[l + dir] = p_n) then l ← l + 1;
        if l > r then {
                if d[l − 1] ≠ p_n then l1 ← l − 1 else l1 ← l − 2;
                if d[l + 1] ≠ p_n then l2 ← l + 1 else l2 ← l + 2;
                if (((a[l] = −1) and (d[l1] < d[l])) or ((a[l] = 1) and
(d[l2] < d[l]))) and (d[l]>dlm)
                        then {lm ← l; dlm ← d[l]};};
        if ((l = r) or (l = r − 1)) and (drm>dlm) then {lm ← rm;
dlm ← drm};
        if (l>r) and (d[r]>dlm) then a[r] ← −a[r];
        r ← r − 1; if (d[r] = p_n) or (d[r + dir] = p_n) then r ← r − 1;
        if l < r then {
                if d[r − 1] ≠ p_n then l1 ← r − 1 else l1 ← r − 2;
                if d[r − 1] ≠ p_n then l2 ← r + 1 else l2 ← r + 2;
                if (((a[r] = −1) and (d[l1] < d[r])) or
((a[r] = 1) and (d[l2] < d[r]))) and (d[r]>drm)
                        then { rm ← r; drm ← d[r] }; };
                if ((l = r) or (l = r − 1)) and (drm>dlm) then
{ lm ← rm; dlm ← drm };
                if (lεr) and (d[r]>dlm) then a[r] − a[r];
                exchange( i, dir);
                if i + dir = lm then lm ← i;
                if i + dir = rm then rm ← i;
                output; };
read( n); for i ← 0 to n + 1 do read p_i;
d[0] ← p_{n+1}; d[n + 1] ← p_{n+1}; d[n + 2] ← p_0;
for i ← 1 to n do { d[i] ← p_i; a[i] ← −1};
repeat
                output;
                l ← 1; r ← n + 1; lm ← n + 2; dlm ← p_0; rm ← n + 2;
drm ← p_0; dir ← −1;
                for i ← n downto 2 do updatelm;
                exchange (lm, a[lm]);
```

output;
$l \leftarrow 1$; r $\leftarrow$ n + 1; $lm \leftarrow n+2$; $dlm \leftarrow p_0$;
$drm \leftarrow p_0$; $rm \leftarrow n+2$; $dir \leftarrow 1$;
for $i \leftarrow 1$ **to** $n-1$ **do** updatelm;
exchange $(lm,\ a[lm])$;
until $lm = n+2$.

1.10 RANKING AND UNRANKING OF COMBINATORIAL OBJECTS

Once the objects are ordered, it is possible to establish the relations between integers $1, 2, \ldots, N$ and all instances of a combinatorial object, where N is the total number of instances under consideration. The mapping of all instances of a combinatorial object into integers is called *ranking*. For example, let *f(X)* be ranking procedure for subsets of the set $\{1, 2, 3\}$. Then, in lexicographic order, $f() = 1$, $f(\{1\}) = 2$, $f(\{1, 2\}) = 3$, $f(\{1, 2, 3\}) = 4$, $f(\{1, 3\}) = 5$, $f(\{2\}) = 6$, $f(\{2, 3\}) = 7$ and $f(\{3\}) = 8$. The inverse of ranking, called *unranking*, is mapping of integers $1, 2, \ldots, N$ to corresponding instances. For instance, $f^{-1}(4) = \{1, 2, 3\}$ in the last example.

The objects can be enumerated in a systematic manner, for some combinatorial classes, so that one can easily construct the sth element in the enumeration. In such cases, an unbiased generator could be obtained by generating a random number s in the appropriate range *(1,N)* and constructing the sth object. In practice, random number procedures generate a number r in interval [0,1); then $s = \lceil rN \rceil$ is required integer.

Ranking and unranking functions exist for almost every kind of combinatorial objects, which has been studied in literature. They also exist for some objects listed in minimal change order. The minimal change order has more use when all instances are to be generated since in this case either the time needed to generate is less or the minimal change order of generating is important characteristics of some applications. In case of generating an instance at random, the unranking functions for minimal change order is usually more sophisticated than the corresponding one following lexicographic order. We use only lexicographic order in ranking and unranking functions presented in this chapter.

In most cases combinatorial objects of given kind are represented as integer sequences. Let $a_1 a_2 \ldots a_m$ be such a sequence. Typically each element a_i has its range that depends on the choice of elements $a_1, a_2, \ldots, a_{i-1}$. For example, if $a_1 a_2 \ldots a_m$ represents a (*m*,*n*)-combination out of $\{1, 2, \ldots, n\}$ then $1 \leq a_1 \leq n - m + 1$, $a_1 < a_2 \leq n - m + 2, \ldots,\ a_{m-1} < a_m \leq n$. Therefore element a_i has $n - m + 1 - a_{i-1}$ different choices.

Let $N(a_1, a_2, \ldots, a_i)$ be the number of combinatorial objects of given kind whose representation starts with $a_1 a_2 \ldots a_i$. For instance, in the set of (4,6)-combinations we have $N(2, 3) = 3$ since 23 can be completed to (4,6)-combination in three ways: 2345, 2346, and 2356.

To find the rank of an object $a_1a_2 \dots a_m$, one should find the number of objects preceding it. It can be found by the following function:

function rank($a_1, a_2, \dots, a_m$)
 $rank \leftarrow 1$;
  for $i \leftarrow 1$ **to** m **do**
    for each $x < a_i$
      $rank \leftarrow rank + N(a_1, a_2, \dots, a_{i-1}, x)$.

Obviously in the last **for** loop only such values x for which $a_1a_2 \dots a_{i-1}x$ can be completed to represent an instance of a combinatorial object should be considered (otherwise adding 0 to the rank does not change its value). We now consider a general procedure for unranking. It is the inverse of ranking function and can be calculated as follows.

procedure unrank ($rank, n, a_1, a_2, \dots, a_m$)
 $i \leftarrow 0$;
 repeat
   $i \leftarrow i + 1$;
   $x \leftarrow$ first possible value;
   while $N(a_1, a_2, \dots, a_{i-1}, x) \leq rank$ **do**
     $\{rank \leftarrow rank - N(a_1, a_2, \dots, a_{i-1}, x)$;
     $x \leftarrow$ next possible value$\}$;
   $a_i \leftarrow x$
 until $rank = 0$;
$a_1a_2 \dots a_m \leftarrow$ lexicographically first object starting by $a_1a_2 \dots a_i$.

We now present ranking and unranking functions for several combinatorial objects. In case of ranking combinations out of $\{1, 2, \dots, n\}$, x is ranged between $a_{i-1} + 1$ and $a_i - 1$. Any (m, n)-combination that starts with $a_1a_2 \dots a_{i-1}x$ is in fact a $(m - i, n - x)$- combination. The number of such combinations is $C(m - i, n - x)$. Thus the ranking algorithm for combinations out of $\{1, 2, \dots, n\}$ can be written as follows ($a_0 = 0$ in the algorithm):

function rankcomb ($a_1, a_2, \dots, a_m$)
 $rank \leftarrow 1$;
 for $i \leftarrow 1$ **to** m **do**
   for $x \leftarrow a_{i-1} + 1$ **to** $a_i - 1$ **do**
     $rank \leftarrow rank + C(m - i, n - x)$.

In lexicographic order, $C(4, 6) = 15$ (4,6)-combinations are listed as 1234, 1235, 1236, 1245, 1246, 1256, 1345, 1346, 1356, 1456, 2345, 2346, 2356, 2456, 3456. The rank of 2346 is determined as $1 + C(4 - 1, 6 - 1) + C(4 - 4, 6 - 5) = 1 + 10 + 1 = 12$ where last two summands correspond to combinations that start with 1 and 2345, respectively. Let us consider a larger example. The rank of 3578 in

(4,9)-combinations is $1 + C(4-1, 9-1) + C(4-1, 9-2) + C(4-2, 9-4) + C(4-3, 9-6) = 104$ where four summands correspond to combinations starting with 1, 2, 34, and 356, respectively.

A simpler formula is given in the work by Lehmer [21]: the rank of combination $a_1a_2 \ldots a_m$ is $C(m, n) - \sum_{j=1}^{m} C(j, n-1-a_{m-j+1})$. It comes from the count of the number of combinations that follow $a_1a_2 \ldots a_m$ in lexicographic order. These are all combinations of j out of elements $\{a_{m-j+1}+1,\ a_{m-j+1}+2, \ldots, a_n\}$, for all $j,\ 1 \leq j \leq m$. In the last example, combinations that follow 3578 are all combinations of 4 out of {4, 5, 6, 7, 8, 9}, combinations with first element 3 and three others taken from {6, 7, 8, 9}, combinations which start with 35 and having two more elements out of set {8, 9} and combination 3579.

The function calculates the rank in two nested **for** loops while the formula would require one **for** loop. Therefore general solutions are not necessarily best in the particular case. The following unranking procedure for combinations follows from general method.

```
procedure unrankcomb (rank, n, a1, a2, ..., am)
    i ← 0; a0 ← 0;
    repeat
        i ← i + 1;
        x ← a(i−1) + 1;
        while C(m − i, n − x) ≤ rank do
                {rank ← rank − C(m − i, n − x); x ← x + 1};
        ai ← x
    until rank = 0;
    for j = i + 1 to m do aj ← n − m + j.
```

What is 104th (4,9)-combination? There are $C(3, 8) = 56$ (4,9)-combinations starting with a 1 followed by $C(3, 7) = 35$ starting with 2 and $C(3, 6) = 20$ starting with 3. Since $56 + 35 \leq 104$ but $56 + 35 + 20 > 104$ the requested combination begins with a 3, and the problem is reduced to finding $104 - 56 - 35 = 13$th (3,6)-combination. There are $C(2, 5) = 10$ combinations starting with 34 and $C(2, 4) = 6$ starting with a 5. Since $13 > 10$ but $13 < 10 + 6$ the second element in combination is 5, and we need to find $13 - 10 = 3$rd (2,4)-combination out of {6, 7, 8, 9}, which is 78, resulting in combination 3578 as the 104th (4,9)-combination.

We also consider the ranking of subsets. The subsets in the set and in the binary representation are listed in different lexicographic orders. In binary representation, the ranking corresponds to finding decimal equivalent of an integer in binary system. Therefore the rank of a subset $b_1, b_2, \ldots, b_n$ is $b_n + 2b_{n-1} + 4b_{n-2} + \cdots + 2^{n-1}b_1$. For example, the rank of 100101 is $1 + 4 + 32 = 37$. The ranks are here between *0* and $2^n - 1$ since in many applications empty subset (here with rank 0) is not taken into consideration. The ranking functions can be generalized to variations out of $\{0, 1, \ldots, m-1\}$ by simply replacing all "2" by "m" in the rank expression. It corresponds to decimal equivalent of a corresponding number in number system with base m.

Similarly, the unranking of subsets in binary representation is equivalent to converting a decimal number to binary one, and can be achieved by the following procedure that uses the mod or remainder function. The value rank mod 2 is 0 or 1, depending whether rank is even or odd, respectively. It can be generalized for m-variations if all "2" are replaced by "m".

function unranksetb($n,\ a_1a_2\ldots a_m$)
  $rank \leftarrow m;\ a_0 \leftarrow 0;$
  for $i \leftarrow m$ **downto** 1 **do**
    $\{b_i \leftarrow rank \bmod 2;\ rank \leftarrow rank - b_i 2^{n-i}\}.$

In the set representation, the rank of n-subset $a_1a_2\ldots a_m$ is found by the following function from the work by Djokić et al. [10].

function rankset($n,\ a_1a_2\ldots a_m$)
  $rank \leftarrow m;\ a_0 \leftarrow 0;$
  for $i \leftarrow 1$ **to** $m-1$ **do**
    for $j \leftarrow a_i + 1$ **to** $a_{i+1} - 1$ **do**
      $rank \leftarrow rank + 2^{n-j}.$

The unranking function [10] gives n-subset with given rank in both representations but the resulting binary string $b_1b_2\ldots b_n$ is assigned its rank in the lexicographic order of the set representation of subsets.

function unranksets($rank,\ n,\ a_1a_2\ldots a_m$)
  $m \leftarrow 0;\ k \leftarrow 1;$ **for** $i \leftarrow 1$ **to** n **do** $b_i \leftarrow 0$;
  repeat
    if $rank \leq 2^{n-k}$ **then** $\{b_k \leftarrow 1;\ m \leftarrow m+1;\ a_m \leftarrow k\};$
    $rank \leftarrow rank - (1-b_k)2^{n-k} - b_k;$
    $k \leftarrow k+1$
  until $k>n$ or $rank = 0$.

As noted in the work by Djokić [10], the rank of a subset $a_1a_2\ldots a_m$ among all $(m,\ n)$-subsets is given by

$$\text{ranks}(a_1a_2\ldots a_m) = \text{rankcomb}(a_1a_2\ldots a_m) + \text{rankcomb}(a_1a_2\ldots a_{m-1}) + \cdots$$
$$+ \text{rankcomb}(a_1a_2) + \text{rankcomb}(a_1).$$

Let $L(m,\ n) = C(1, n) + C(2,\ n) + \cdots + C(n,\ m)$ be the number of $(m,\ n)$-subsets. The following unranking algorithm [10] returns the subset $a_1a_2\ldots a_m$ with given rank.

function unranklim ($rank,\ n,\ m,\ a_1a_2\ldots a_r$)
  $r \leftarrow 0;\ i \leftarrow 1;$
  repeat
    $s \leftarrow t - 1 - L(m-r-1,\ n-i);$

```
            if s>0 then t ← s else {r ← r + 1; a_r ← i; rank ← rank − 1};
          i ← i + 1
      until i = n + 1 or rank = 0.
```

Note that the (*m, n*)-subsets in lexicographic order also coincide with a minimal change order of them. This is a rare case. Usually it is trivial to show that lexicographic order of instances of an object is not a minimal change order.

Ranking and unranking functions for integer compositions can be described by using the relation between compositions and either subsets or combinations (discussed above).

A ranking algorithm for n-permutations is as follows [21]:

```
function rankperm(a_1 a_2 ... a_n )
     rank ← 1 ;
     for i ← 1 to n do
               rank ← rank + k(n − i)! where k = |{1, 2, ..., a_i − 1}\
{a_1, a_2, ..., a_{i−1}}|.
```

For example, the rank of permutation 35142 is $1 + 2 \times 4! + 3 \times 3! + 1 \times 1! = 68$ where permutations starting with 1, 2, 31, 32, 34, and 3512 should be taken into account. The unranking algorithm for permutations is as follows [21]. Integer division is used (i.e., $13/5 = 2$).

```
procedure unrankperm(rank, n, a_1 a_2 ... a_n)
    for i ← 1 to n do {
              ⌊k ← (rank − 1)/(n − i)!⌋;
              a_i ← kth element of {1, 2, ..., n}\{a_1, a_2, ..., a_{i−1}};
              rank ← rank − (k − 1)(n − i)!}.
```

The number of instances of a combinatorial object is usually exponential in size of objects. The ranks, being large integers, may need $O(n)$ or similar number of memory location to be stored and also $O(n)$ time for the manipulation with them. Avoiding large integers is a desirable property in random generation in some cases. The following two sections offer two such approaches.

1.11 RANKING AND UNRANKING OF SUBSETS AND VARIATIONS IN GRAY CODES

In a Gray code (or minimal change) order, instances of a combinatorial object are listed such that successive instances differ as little as possible. In this section we study Gray codes of subsets in binary representation. Gray code order of subsets is an ordered cyclic sequence of 2^n n-bit strings (or codewords) such that successive codewords differ by the complementation of a single bit. If the codewords are considered to be

vertices of an n-dimensional binary cube, it is easy to conclude that Gray code order of subsets corresponds to a Hamiltonian path in the binary cube. We will occasionally refer in the sequel to nodes of binary cubes instead of subsets. Although a binary cube may have various Hamiltonian paths, we will define only one such path, called the binary-reflected Gray code [17] that has a number of advantages, for example, easy generation and traversing a subcube in full before going to other subcube. The (binary reflected) Gray code order of nodes of n-dimensional binary cube can be defined in the following way:

- For $n = 1$ the nodes are numbered $g(0) = 0$ and $g(1) = 1$, in this order,
- If $g(0), g(1), \ldots, g(2^n - 1)$ is the Gray code order of nodes of an n-dimensional binary cube, then $g(0) = 0g(0),\ g(1) = 0g(1), \ldots,\ g(2^n - 1) = 0g(2^n - 1),\ g(2^n) = 1g(2^n - 1),\ g(2^n + 1) = 1g(2^n - 2),\ \ldots,\ g(2^{n+1} - 2) = 1g(1),$ $g(2^{n+1} - 1) = 1g(0)$ is a Gray code order of nodes of a $(n + 1)$-dimensional binary cube.

As an example, for $n = 3$ the order is $g(0) = 000,\ g(1) = 001,\ g(2) = 011,\ g(3) = 010,\ g(4) = 110,\ g(5) = 111,\ g(6) = 101,\ g(7) = 100$. First, let us see how two nodes u and v can be compared in Gray code order. We assume that a node x is represented by a bitstring $x_1 \geq x_2 \ldots x_n$. This corresponds to decimal node *address* $x = 2^{n-1}x_1 + 2^{n-2}x_2 + \cdots + 2x_{n-1} + x_n$ where $0 \leq x \leq 2^n - 1$. Let i be the most significant (or leftmost) bit where u and v differ, that is, $u[l] = v[l]$ for $l < i$ and $u[i] \neq v[i]$. Then $u < v$ if and only if $u[1] + u[2] + \cdots + u[i]$ is an even number. For instance, $11100 < 10100 < 10110$.

The above comparison method gives a way to find Gray code address t of a node u (satisfying $g(t) = u$), using the following simple procedure; it ranks the Gray code sequences.

```
procedure rank_GC(n, u, t);
    sum ← 0; t ← 0;
    for l ← 1 to n do {
        sum ← sum + u[l];
        if sum is odd then t ← t + 2^(n−l) }.
```

The inverse operation, finding the binary address u of node having Gray code address t $(0 \leq t \leq 2^n - 1)$, can be performed by the following procedure; it unranks the Gray code sequences.

```
procedure unrank_GC( n,u,t);
    sum ← 0; q ← t; size ← 2^n;
    for l ← 1 to n do {
        size ← size/2;
        if q ≥ size then {q ← q − size; s ← 1} else s ← 0 ;
        if sum + s is even then u[l] ← 0 else u[l] ← 1;
        sum ← sum + u[l]}.
```

The important property of the Gray code order is that corresponding nodes of a binary cube define an edge of the binary cube whenever they are neighbors in the Gray code order (this property is not valid for the lexicographic order $0,\ 1,\ 2, \ldots,\ 2^n - 1$ of binary addresses).

The reflected Gray code order for subsets has been generalized for variations [7,15]. Gray codes of variations have application in analog to digital conversion of data.

We establish a n-ary reflected Gray code order of variations as follows. Let $x = x_1 \geq x_2 \ldots x_m$ and $y = y_1 y_2 \ldots y_m$ be two variations. Then $x < y$ iff there exist i, $0 \leq i \leq m$, such that $x_j = y_j$ for $j < i$ and either $x_1 + x_2 + \ldots + x_{i-1}$ is even and $x_i < y_i$ or $x_1 + x_2 + \cdots + x_{i-1}$ is odd and $x_i > y_i$. We now prove that the order is a minimal change order. Let x and y be two consecutive variations in given order, $x < y$, and let $x_j = y_j$ for $j < i$ and $x_i \neq y_i$. There are two cases. If $x_i < y_i$ then $X_i = x_1 + x_2 + \cdots + x_{i-1}$ is even and $y_i = x_i + 1$. Thus X_{i+1} and Y_{i+1} have different parity, since $Y_{i+1} = X_{i+1} + 1$. It means that either $x_{i+1} = y_{i+1} = 0$ or $x_{i+1} = y_{i+1} = n - 1$ (the $(i + 1)$th element in x is the maximum at that position while the $(i + 1)$ –the element in y is the minimum at given position, and they are the same because of different parity checks). Similarly we conclude $Y_j = X_j + 1$ and $x_j = y_j$ for all $j > i + 1$. The case $x_i > y_i$ can be analyzed in analogous way, leading to the same conclusion.

As an example, 3-ary reflected Gray code order of variations out of {0, 1, 2} is as follows (the variations are ordered columnwise):

000 122 200
001 121 201
002 120 202
012 110 212
011 111 211
010 112 210
020 102 220
021 101 221
022 100 222.

It is easy to check that, at position $i(1 \leq i \leq m)$, each element repeats n^{m-i} times. The repetition goes as follows, in a cyclic manner: 0 repeats n^{m-i} times, 1 repeats n^{m-i} times, ..., $n - 1$ repeats n^{m-i} times, and then these repetitions occur in reverse order, that is $n - 1$ repeats n^{m-i} times, ..., 0 repeats n^{m-i} times.

Ranking and unranking procedures for variations in the n-ary reflected Gray code are described in the work by Flores [15].

1.12 GENERATING COMBINATORIAL OBJECTS AT RANDOM

In many cases (e.g., in probabilistic algorithms), it is useful to have means of generating elements from a class of combinatorial objects uniformly at random (an unbiased generator). Instead of testing new hypothesis on all objects of given kind, which may be time consuming, several objects chosen at random can be used for testing, and likelihood of hypothesis can be established with some certainty. There are several ways of choosing a random object of given kind. All known ways are based on the correspondence between integer or real number(s) and combinatorial objects. This means that objects should be ordered in a certain fashion. We already described two general ways for choosing a combinatorial object at random. We now describe one more way, by using *random number series*. This method uses a series of random numbers in order to avoid large integers in generating a random instance of an object. Most known techniques in fact generate a series of random numbers. This section will present methods for generating random permutations and integer partitions. A random subset can easily be generated by flipping coin for each of its elements.

1.12.1 Random Permutation and Combination

There exist a very simple idea of generating a random permutation of $A = \{a_1, \ldots, a_n\}$. One can generate an array $x_1, x_2, \ldots, x_n$ of random numbers, sort them, and obtain the destination indices for each element of A in a random permutation. The first m elements of the array can be used to determine a random (m, n)-combination (the problem of generating combinations at random is sometimes called random sampling). Although very simple, the algorithm has $O(n \log n)$ time complexity [if random number generation is allowed at most $O(\log n)$ time]. We therefore describe an alternative solution that leads to a linear time performance. Such techniques for generating permutations of $A = \{a_1, \ldots, a_n\}$ at random first appeared in the works by the Durstenfeld [8] and Hoses [24], and repeated in the works by Nijeshius [25] and Reingold [30]. The algorithm uses a function random (x) that generates a random number x from interval (0,1), and is as follows.

> **for** $i \leftarrow 1$ **to** $n-1$ **do** {
> random(x_i);
> $c_i \lfloor x_i(n-i+1) \rfloor + 1$;
> $j \leftarrow i - 1 + c_i$;
> exchange a_i with a_j }.

As an example, we consider generating a permutation of $\{a, b, c, d, e, f\}$ at random. Random number $x_1 = 0.7$ will choose $\lfloor 6 \times 0.7 \rfloor + 1 = 5$th element e as the first element in a random permutation, and decides the other elements considering the set $\{b, c, d, a, f\}$ (e exchanged with a). The process is repeated: another random number, say $x_2 = 0.45$, chooses $\lfloor 5 \times 0.45 \rfloor + 1 = 3$rd element d from $\{b, c, d, a, f\}$ to be the

second element in a random permutation, and b and d are exchanged. Thus, random permutation begins with e, d, and the other elements are decided by continuing same process on the set $\{c, b, a, f\}$.

Assuming that random number generator takes constant time, the algorithm runs in linear time. The same algorithm can be used to generate combinations at random. The first m iterations of the **for** loop determine (after sorting, if such output is preferable) a combination of m out of n elements.

Uniformly distributed permutations cannot be generated by sampling a finite portion of a random sequence and the standard method [8] does not preserve randomness of the x-values due to computer truncations. Truncation problems appear with other methods as well.

1.12.2 Random Integer Partition

We now present an algorithm from the work by Nijenhius and Wilf [26] that generates a random integer partition. It uses the distribution of the number of partitions *RP(n,m)* of n into parts not greater than m.

First, we determine the first part. An example of generating random partition of 12 will be easier to follow than to show formulas. Suppose a random number generator gives us $r_1 = 0.58$. There are 77 partitions of 12. In lexicographic order, the random number should point to $0.58 \times 77 = 44.66$th integer partition. We want to avoid rounding and unranking here. Thus, we merely determine the largest part such. Looking at the distribution *RP(12,m)* of partitions of 12 (Section 1.2), we see that all integer partitions with ranks between 35 and 47 have the largest part equal to 5. What else we need in a random partition of 12? We need a random partition of $12 - 5 = 7$ such that its largest part is 5 (the second part cannot be larger than the first part). There are $\text{RP}(7, 5) = 13$ such partitions. Let the second random number be $r_2 = 0.78$. The corresponding partition of 7 has the rank $0.78 \times 13 = 10.14$. Partitions of 7 ranked between 9 and 11 have the largest part equal to 4. It remains to find a random partition of $7 - 4 = 3$ with largest part 4 (which in this case is not a real restriction). There are $\text{RP}(3, 3) = 3$ partitions as candidates let $r_3 = 0.20$. Then $0.20 \times 3 = 0.6$ points to the third (and remaining) parts of size 1. However, since the random number is taken from open interval (0,1), in our scheme the partition $n = n$ will never be chosen unless some modification to our scheme is made. Among few possibilities, we choose that the value < 1 as the rank actually points to the available partition with the maximal rank. Thus, we decide to choose partition $3 = 3$, and the random partition of 12 that we obtained is $12 = 5 + 4 + 3$.

An algorithm for generating random rooted trees with prescribed degrees (where the number of nodes of each down degree is specified in advance) is described in the work by Atkinson [3]. A linear time algorithm to generate binary trees uniformly at random, without dealing with large integers is given in the work by Korsch [20]. An algorithm for generating valid parenthesis strings (each open parenthesis has its matching closed one and vice versa) uniformly at random is described in the work

by Arnold and Sleep [2]. It can be modified to generate binary trees in the bitstring notation at random.

1.13 UNRANKING WITHOUT LARGE INTEGERS

Following the work by Stojmenovic [38], this section describes functions mapping the interval $[0\ldots1)$ into the set of combinatorial objects of certain kind, for example, permutations, combinations, binary and t-ary trees, subsets, variations, combinations with repetitions, permutations of combinations, and compositions of integers. These mappings can be used for generating these objects at random, with equal probability of each object to be chosen. The novelty of the technique is that it avoids the use of very large integers and applies the random number generator only once. The advantage of the method is that it can be applied for both random object generation and dividing all objects into desirable sized groups.

We restrict ourselves to generating only one random number to obtain a random instance of a combinatorial object but request no manipulation with large integers. Once a random number g in [0,1) is taken, it is mapped into the set of instances of given combinatorial object by a function $f(g)$ in the following way. Let N be the number of all instances of a combinatorial object. The algorithm finds the instance x such that the ratio of the number of instances that precede x and the total number of instances is $\leq g$. In other words, it finds the instance $f(g)$ with the ordinal number $\lfloor gN \rfloor + 1$. In all cases that will be considered in this section, each instance of given combinatorial object may be represented as a sequence $x_1 \ldots x_m$, where x_i may have integer values between 0 and n (m and n are two fixed numbers), subject to constraints that depend on particular case.

Suppose that the first $k-1$ elements in given instance are fixed, that is, $x_i = a_i$, $1 \leq i < k$. We call them $(k-1)$ -fixed instances. Let $a'_1 < \cdots < a'_h$ be all possible values of x_k of a given $(k-1)$ -fixed instance. By $S(k,u)$, $S(k, \leq u)$, and $S(k, \geq u)$, we denote the ratio of the number of $(k-1)$ -fixed instances for which $x_k = a'_u (x_k \leq a'_u$, and $x_k \geq a'_u$ respectively) and the number of $(k-1)$ -fixed instances. In other words, these are the probabilities (under uniform distribution) that an instance for which $x_i = a_i,\ 1 \leq i < k$, has the value in variable x_k which is $= a'_u$, $\leq a'_u$, and $\geq a'_u$, respectively.

Clearly, $S(k,u) = S(k, \leq u) - S(k, \leq u-1)$ and $S(k, \geq u) = 1 - S(k, \leq u-1)$. Thus

$$\frac{S(k,u)}{S(k,\geq u)} = \frac{S(k,\leq u) - S(k,\leq u-1)}{1 - S(k,\leq u-1)}.$$

Therefore

$$S(k,\leq u) = S(k,\leq u-1) + (1 - S(k,\leq u-1))\frac{S(k,u)}{S(k,\geq u)}.$$

Our method is based on the last equation. The large numbers can be avoided in cases when $S(k,u)/S(k,\geq u)$ is explicitly found and is not a very large integer. This

condition is satisfied for combinations, permutations, t-ary trees, variations, subsets, and other combinatorial objects.

Given g from $[0, \ldots, 1)$, let l be chosen such that $S(1, \leq u-1) < g \leq S(1, \leq u)$. Then $x_1 = a'_u$ and the first element of combinatorial object ranked g is decided. To decide the second element, the interval $[S(1, \leq u-1) \ldots S(1, \leq u))$ containing g can be linearly mapped to interval $[0 \ldots 1)$ to give the new value of g as follows:

$$g \leftarrow \frac{g - S(1, \leq u-1)}{S(1, \leq u) - S(1, \leq u-1)}.$$

The search for the second element proceeds with the new value of g. Similarly the third, ..., mth elements are found. The algorithm can be written formally as follows, where $p\prime$ and p stand for $S(k, \leq u-1)$ and $S(k, \leq u)$, respectively.

procedure object(m, n, g);
  $p\prime \leftarrow 0$;
  for $k \leftarrow 1$ **to** m **do** {
    $u \leftarrow 1$;
    $p \leftarrow S(k, 1)$;
    while $p \leq g$ **do** {
      $p\prime \leftarrow p$;
      $u \leftarrow u + 1$;
      $p \leftarrow p\prime + (1 - p\prime)\dfrac{S(k, u)}{S(k, \geq u)}$ }
      $x_k \leftarrow a'_u$;
      $g \leftarrow \dfrac{g - p\prime}{p - p\prime}$ }.

Therefore the technique does not involve large integers iff $S(k, u)/S(k, \geq u)$ is not a large integer for any k and u in the appropriate ranges (note that $S(k, \geq 1) = 1$).

The method gives theoretically correct result. However, in practice the random number g and intermediate values of p are all truncated. This may result in computational imprecision for larger values of m or n. The instance of a combinatorial object obtained by a computer implementation of above procedure may differ from the theoretically expected one. However, the same problem is present with other known methods (as noted in the previous section) and thus this method is comparable with others in that sense. Next, in applications, randomness is practically preserved despite computational errors.

1.13.1 Mapping $[0 \ldots 1)$ Into the Set of Combinations

Each (m, n)-combination is specified as an integer sequence $x_1, \ldots, x_m$ such that $1 \leq x_1 < \cdots < x_m \leq n$. The mapping $f(g)$ is based on the following lemma. Recall that (k-1)-fixed combinations are specified by $x_i = a_i$, $1 \leq i < k$. Clearly, possible values for x_k are $a'_1 = a_{k-1} + 1, a'_2 = a_{k-1} + 2, \ldots, a'_h = n$ (thus $h = n - a_{k-1}$).

Lemma 1. The ratio of the number of $(k-1)$-fixed (m,n)-combinations for which $x_k = j$ and the number of $(k-1)$-fixed combinations for which $x_k \geq j$ is $(m-k+1)/(n-j+1)$ whenever $j > a_{k-1}$.

Proof. Let $y_{k-i} = x_i - j,\ k < i \leq n$. The $(k-1)$-fixed (m,n)-combinations for which $x_k = j$ correspond to $(m-k, n-j)$ -combinations $y_1, \ldots, y_{m-k}$, and their number is $C(m-k, n-j)$. Now let $y_{k-i+1} = x_i - j + 1, k \leq i \leq n$. The $(k-1)$-fixed combinations for which $x_k \geq j$ correspond to $(m-k+1, n-j+1)$ -combinations $y_1 \ldots y_{m-k+1}$, and their number is $C(m-k+1, n-j+1)$. The ratio in question is

$$\frac{C(m-k, n-j)}{C(m-k+1, n-j+1)} = \frac{m-k+1}{n-j+1}. \blacksquare$$

Using the notation introduced in former section for any combinatorial objects, let $u = j - a_{k-1}$. Then, from Lemma 1 it follows that

$$\frac{S(k,u)}{S(k, \geq u)} = \frac{m-k+1}{n-u-a_{k-1}+1}$$

for the case of (m,n)-combinations, and we arrive at the following procedure that finds the (m,n)-combination with ordinal number $\lfloor gC(m,n) \rfloor + 1$. The procedure uses variable j instead of u, for simplicity.

```
procedure combination( m,n,g);
  j ← 0; p' ← 0;
  for k ← 1 to m do {
        j ← j + 1;
        p ← (m − k + 1)/(n − j + 1);
        while p ≤ g do {
              p' ← p;
              j ← j + 1;
              p ← p' + (1 − p')(m − k + 1)/(n − j + 1) }
     x_k ← j;
     g ← (g − p')/(p − p') }.
```

A random sample of size m out of the set of n objects, that is, a random (m,n)-combination can be found by choosing a real number g in $[0, \ldots, 1)$ and applying the map $f(g) = \text{combination}(m,n,g)$.

Each time the procedure combination (m,n,g) enters for or while loop, the index j increases by 1; since j has n as upper limit, the time complexity of the algorithm is $O(n)$, that is, linear in n. Using the correspondences established in Chapter 1, the same procedure may be applied to the case of combinations with repetitions and compositions of n into m parts.

1.13.2 Random Permutation

Using the definitions and obvious properties of permutations, we conclude that, after choosing $k-1$ beginning elements in a permutation, each of the remaining $n-k+1$ elements has equal chance to be selected next. The list of unselected elements is kept in an array *remlist*. This greatly simplifies the procedure that determines the permutation $x_1 \dots x_n$ with index $\lfloor gP(n) \rfloor + 1$.

procedure permutation(n,g);
for $i \leftarrow 1$ **to** n **do** $remlist_i \leftarrow i$;
for $k \leftarrow 1$ **to** n **do** {
  $u \leftarrow \lfloor g(n-k+1) \rfloor + 1$;
  $x_k \leftarrow remlist_u$;
  for $i \leftarrow u$ **to** $n-k$ **do** $remlist_i \leftarrow remlist_{i+1}$;
  $g \leftarrow g(n-k+1) - u + 1$}.

The procedure is based on the same choose and exchange idea as the one used in the previous section but requires one random number generator instead of a series of n generators. Because the lexicographic order of permutations and the ordering of real numbers in $[0 \dots 1)$ coincide, the list of remaining elements is kept sorted, which causes higher time complexity $O(n^2)$ of the algorithm.

Consider an example. Let $n = 8$ and $g = 0.1818$. Then $\lfloor 0.1818 * 8! \rfloor + 1 = 7331$ and the first element of 7331st 8-permutation is $u = \lfloor 0.1818 \times 8 \rfloor + 1 = 2$; the remaining list is 1,3,4,5,6,7,8 ($7331 - 1 \times 5040 = 2291$; this step is for verification only, and is not part of the procedure). The new value of g is $g = 0.1818 \times 8 - 2 + 1 = 0.4544$, and new u is $u = \lfloor 0.4544 \times 7 \rfloor + 1 = 4$; the second element is 4th one in the remaining list, which is 5; the remaining list is 1,3,4,6,7,8. Next update is $g = 0.4544 \times 7 - 3 = 0.1808$ and $u = \lfloor 0.1808 \times 6 \rfloor + 1 = 2$; the 3rd element is the 2nd in the remaining list, that is, 3; the remaining list is 1,4,6,7,8. The new iteration is $g = 0.1808 \times 6 - 1 = 0.0848$ and $u = \lfloor 0.0848 \times 5 \rfloor + 1 = 1$; the 4th element is 1st in the remaining list, that is, 1; the remaining list is 4,6,7,8. Further, $g = 0.0848 \times 5 = 0.424$ and $u = \lfloor 0.424 \times 4 \rfloor + 1 = 2$; the 5th element is 2nd in the remaining list, that is, 6; the new remaining list is 4,7,8. The next values of g and u are $g = 0.424 \times 4 - 1 = 0.696$ and $u = \lfloor 0.696 \times 3 \rfloor + 1 = 3$; the 6th element is 3rd in the remaining list, that is, 8; the remaining list is 4,7. Finally, $g = 0.696 \times 3 - 2 = 0.088$ and $u = \lfloor 0.088 \times 2 \rfloor + 1 = 1$; the 7th element is 1st in the remaining list, that is, 4; now 7 is left, which is the last, 8th element. Therefore, the required permutation is 2,5,3,1,6,8,4,7.

All (m,n)-permutations can be obtained by taking all combinations and listing permutations for each combination. Such an order that is not lexicographic one, and (m,n)-permutations are in this case refereed to as the permutations of combinations. Permutation of combinations with given ordinal number can be obtained by running the procedure combination first, and continuing the procedure permutation afterwards, with the new value of g that is determined at the end of the procedure combination.

1.13.3 Random *t*-Ary Tree

The method requires to determine $S(k, 1)$, $S(k, u)$, and $S(k, \geq u)$. Each element b_k has two possible values, that is, $b_k = a'_1 = 0$ or $b_k = a'_2 = 1$; thus it is sufficient to find *S(k,1)* and $S(k, \geq 1)$. $S(k, \geq 1)$ is clearly equal to 1. Let the sequence $b_k \ldots b_{tn}$ contains q ones, the number of such sequences is $D(k-1, q)$. Furthermore, *D(k,q)* of these sequences satisfy $b_k = 0$. Then

$$S(k, 1) = \frac{D(k, q)}{D(k-1, q)} = \frac{(t(n-q)-k+1)(tn-k-q+2)}{(t(n-q)-k+2)(tn-k+1)}.$$

This leads to the following simple algorithm that finds the t-ary tree *f(g)* with the ordinal number $\lfloor gB(t, n) \rfloor + 1$.

procedure tree(t, n, g);
 $p\prime \leftarrow 0$; $q \leftarrow n$;
 for $k \leftarrow 1$ to tn do {
  $b_k \leftarrow 0$;
  $p \leftarrow \dfrac{(t(n-q)-k+1)(tn-k-q+2)}{(t(n-q)-k+2)(tn-k+1)}$;
  if $p \leq g$ **then** {
   $p\prime \leftarrow p$;
   $b_k \leftarrow 1$;
   $q \leftarrow q - 1$;
   $p \leftarrow 1$ }
  $g \leftarrow \dfrac{g-p}{p-p\prime}$ }

The time complexity of the above procedure is clearly linear, that is, $O(tn)$.

1.13.4 Random Subset and Variation

There is a fairly simple mapping procedure for subsets in binary representation. Let $g = 0.\ a_1 \ldots a_n a_{n+1} \ldots$ be number g written in the binary numbering system. Then the subset with ordinal number $\lfloor gS(n) \rfloor + 1$ is coded as $a_1 \ldots a_n$. Using a relation between subsets and compositions of n into any number of parts, described procedure can be also used to find the composition with ordinal number $\lfloor gCM(n) \rfloor + 1$.

A mapping procedure for variations is a generalization of the one used for subsets. Suppose that the variations are taken out of the set $\{0, 1, \ldots, n-1\}$. Let $g = 0.\ a_1 a_2 \ldots a_m a_{m+1} \ldots$ be the number g written in the number system with the base n, that is, $0 \leq a_i \leq n-1$ for $1 \leq i \leq m$. Then the variation indexed $\lfloor gV(m, n) \rfloor + 1$ is coded as $a_1 a_2 \ldots a_m$.

If variations are ordered in the n-ary reflected Gray code then the variation indexed $\lfloor gV(m, n) \rfloor + 1$ is coded as $b_1 b_2 \ldots b_m$, where $b_1 = a_1$, $b_i = a_i$ if $a_1 + a_2 + \cdots + a_{i-1}$ is even and $b_i = n - 1 - a_i$ otherwise ($2 \leq i \leq m$).

REFERENCES

1. Akl SG. A comparison of combination generation methods. ACM Trans Math Software 1981;7(1):42–45.
2. Arnold DB. Sleep MR. Uniform random generation of balanced parenthesis strings. ACM Trans Prog Lang Syst 1980;2(1):122–128.
3. Atkinson M. Uniform generation of rooted ordered trees with prescribed degrees. Comput J 1993;36(6):593–594.
4. Akl SG, Olariu S, Stojmenovic I. A new BFS parent array encoding of t-ary trees, Comput Artif Intell 2000;19:445–455.
5. Belbaraka M, Stojmenovic I. On generating B-trees with constant average delay and in lexicographic order. Inform Process Lett 1994;49(1):27–32.
6. Brualdi RA. Introductory Combinatorics. North Holland; 1977.
7. Cohn M. Affine m-ary gray codes, Inform Control 1963;6:70–78.
8. Durstenfeld R. Random permutation (algorithm 235). Commun ACM 1964;7:420.
9. Djokić B, Miyakawa M, Sekiguchi S, Semba I, Stojmenović I. A fast iterative algorithm for generating set partitions. Comput J 1989;32(3):281–282.
10. Djokić B, Miyakawa M, Sekiguchi S, Semba I, Stojmenović I. Parallel algorithms for generating subsets and set partitions. In: Asano T, Ibaraki T, Imai H, Nishizeki T, editors. Proceedings of the SIGAL International Symposium on Algorithms; August 1990; Tokyo, Japan. Lecture Notes in Computer Science. Volume 450. p 76–85.
11. Ehrlich G. Loopless algorithms for generating permutations, combinations and other combinatorial configurations. J ACM 1973;20(3):500–513.
12. Er MC. Fast algorithm for generating set partitions. Comput J 1988;31(3):283–284.
13. Er MC. Lexicographic listing and ranking t-ary trees. Comp J 1987;30(6):569–572.
14. Even S. Algorithmic Combinatorics. New York: Macmillan; 1973.
15. Flores I. Reflected number systems. IRE Trans Electron Comput 1956;EC-5:79–82.
16. Gupta UI, Lee DT, Wong CK. Ranking and unranking of B-trees. J Algor 1983;4: 51–60.
17. Heath FG. Origins of the binary code. Sci Am 1972;227(2):76–83.
18. Johnson SM. Generation of permutations by adjacent transposition, Math Comput 1963;282–285.
19. Knuth DE. The Art of Computer Programming, Volume 1: Fundamental Algorithms. Reading, MA: Addison-Wesley; 1968.
20. Korsch JF. Counting and randomly generating binary trees. Inform Process Lett 1993;45:291–294.
21. Lehmer DH. The machine tools of combinatorics. In: Beckenbach E, editor. Applied Combinatorial Mathematics. Chapter 1. New York: Wiley; 1964. p 5–31.
22. Lucas J, Roelants van Baronaigien D, Ruskey F. On rotations and the generation of binary trees. J Algor 1993;15:343–366.
23. Misfud CJ, Combination in lexicographic order (Algorithm 154). Commun ACM 1963;6(3):103.
24. Moses LE, Oakford RV. Tables of Random Permutations. Stanford: Stanford University Press; 1963.

25. Nijenhius A, Wilf H. Combinatorial Algorithms. Academic Press; 1978.
26. Nijenhius A, Wilf HS. A method and two algorithms on the theory of partitions. J Comb Theor A 1975;18:219–222.
27. Ord-Smith RJ. Generation of permutation sequences. Comput J 1970;13:152–155 and 1971;14:136–139.
28. Parberry I. Problems on Algorithms. Prentice Hall; 1995.
29. Payne WH, Ives FM. Combination generators. ACM Transac Math Software 1979;5(2):163–172.
30. Reingold EM, Nievergelt J, Deo N. Combinatorial Algorithms. Englewood Cliffs, NJ: Prentice Hall; 1977.
31. Sedgewick R. Permutation generation methods. Comput Survey 1977;9(2):137–164.
32. Semba I. An efficient algorithm for generating all partitions of the set $\{1, \ldots, n\}$. J Inform Process 1984;7:41–42.
33. Semba I. An efficient algorithm for generating all k-subsets ($1 \leq k \leq m \leq n$) of the set $\{1, 2, \ldots, n\}$ in lexicographic order. J Algor 1984;5:281–283.
34. Semba I. A note on enumerating combinations in lexicographic order. J Inform Process 1981;4(1):35–37.
35. Skiena S. Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Addison-Wesley; 1990.
36. Stojmenovic I. Listing combinatorial objects in parallel. Int J Parallel Emergent Distrib Syst 2006;21(2):127–146.
37. Stojmenović I, Miyakawa M. Applications of a subset generating algorithm to base enumeration, knapsack and minimal covering problems. Comput J 1988;31(1):65–70.
38. Stojmenović I. On random and adaptive parallel generation of combinatorial objects. Int J Comput Math 1992;42:125–135.
39. Trotter HF, Algorithm 115. Commun ACM 1962;5:434–435.
40. Wells MB, Elements of Combinatorial Computing. Pergamon Press; 1971.
41. Xiang L, Ushijima K, Tang C. Efficient loopless generation of Gray codes for k-ary trees. Inform Process Lett 2000;76:169–174.
42. Zaks S. Lexicographic generation of ordered trees. Theor Comput Sci 1980;10:63–82.
43. Zoghbi A, Stojmenović I. Fast algorithms for generating integer partitions. Int J Comput Math 1998;70:319–332.