ON RANDOM AND ADAPTIVE PARALLEL GENERATION OF COMBINATORIAL OBJECTS

IVAN STOJMENOVić

Computer Science Department, University of Ottawa, Ottawa, Ontario, Canada K1N 9B4

(Received 30 April 1990)

This paper describes functions mapping the interval $[0..1)$ into the set of combinatorial objects of certain kind, e.g. permutations, combinations, binary and $t$-ary trees, subsets, variations, combinations with repetitions, permutations of combinations and composition 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 at the same time (known methods either use counters that are exponential in the size of objects or make use of a series of random numbers). The advantage of the new method is that it can be applied for both random object generation and dividing all objects into desirable sized groups. The latter is exploited in designing adaptive algorithm for generating all combinations, permutations, $t$-ary trees, or variations, on a parallel model of computation, where an algorithm runs adaptively if it can be implemented on a model with arbitrary number of available processors.

KEY WORDS: Random generation, combinatorial objects, lexicographic order, adaptive parallel algorithms.


1 INTRODUCTION

In many cases (for example, in probabilistic algorithms), it is useful to have a means of generating elements from a class of combinatorial objects uniformly at random (an unbiased generator). For some combinatorial classes the objects can be enumerated in a systematic manner so that one can easily construct the $s$-th element in the enumeration. In such case an unbiased generator could be obtained by generating a random number $s$ in the appropriate range and constructing the $s$-th object. Such methods exist for generating combinations, permutations, trees etc. [17, 18, 19, 29, 33, 36]. This technique is not practical however, because the number $s$ can be exponential in the size of objects. Difficulties occur with both manipulation of large integers and random number generation.

1 This research is supported by National Science and Engineering Research Council of Canada.
It is therefore preferable to avoid large integers in a random generating method. Such techniques exist for generating permutations at random, first appeared by Durstenfeld [10], and repeated in [21, 29, 33]. The algorithm is: Given the objects \( a_1, \ldots, a_n \), and random values \( x_i \) in \([0, 1)\), \( 1 \leq i \leq n - 1 \), compute \( c_i = \lfloor x_i(n - i + 1) \rfloor + 1 \) and exchange \( a_i \) with \( a_j \) where \( j = i - 1 + c_i \). The same algorithm can be used to generate combinations at random. Random trees are generated in [26, 27], without dealing with large integers. A random subset can easily be generated by flipping coin for each of its elements. Random combinatorial objects are studied also in [7, 14, 22, 35, 28, 30, 31].

All known techniques generate a series of random numbers. In [16] it is noted that uniformly-distributed permutations cannot be generated by sampling a finite portion of a random sequence; and that standard method [10] does not preserve randomness of the \( x \)-values due to computer truncations. Truncation problems appear with other methods as well.

In this paper we restrict ourselves to generating only one random number to obtain a random combinatorial object of certain kind (as in [17, 18, 19]) but request no manipulation with large integers. Once a random number \( g \) in \([0, 1)\) is taken, it is mapped into the set of combinatorial objects by a function \( f(g) \) in the following way. Let the number of combinatorial objects be \( R(m, n) \). For example, \( R(m, n) = C(m, n) = n!/(m!(n - m)!) \) in case of combinations of \( m \) out of \( n \) elements. If combinatorial objects are specified by one parameter \( n \) only, then \( R(m, n) = R(n) \), as in case of permutations of \( n \) elements \( (R(n) = n!) \), binary trees with \( n \) nodes \( (R(n) = C(2n, n)/(n + 1)) \), and subsets of the set of \( n \) elements \( (R(n) = 2^n) \).

In most cases, objects are ordered by a lexicographic order, i.e. if \( A = (a_1, a_2, \ldots, a_m) \) and \( B = (b_1, b_2, \ldots, b_m) \) are two objects, then \( A \) precedes \( B \) lexicographically if and only if, for some \( j \geq 1 \), \( a_i = b_i \) when \( i < j \), and \( a_j < b_j \).

Our algorithms find the combinatorial object \( x \) such that the ratio of the number of objects that precede \( x \) and the total number of objects is \( \leq g \). In other words, it finds the object \( f(g) \) with the ordinal number \( \lfloor gR(m, n) \rfloor + 1 \). We also describe an application of our technique in the design of an adaptive parallel algorithm for generating all combinatorial objects of given kind.

2 MAPPING \([0..1)\) TO COMBINATORIAL OBJECTS

In all cases that will be considered in this paper, each combinatorial object of given kind may be represented as a sequence \( x_1 \ldots x_m \), where \( m \) is fixed, and \( x_i \) may have integer values between 0 and \( n \), subject to constraints that depend on particular case.

Suppose that the first \( k - 1 \) elements in given object are fixed, i.e. \( x_i = a_i \), \( 1 \leq i < k \). We call these objects \((k - 1)\)-fixed. Let \( a'_1 < \cdots < a'_k \) be all possible values of \( x_k \) of a given \((k - 1)\)-fixed object. by \( S(k, l), S(k, \leq l), \) and \( S(k, \geq l) \) we denote the ratio of the number of \((k - 1)\)-fixed combinatorial objects for which \( x_k = a'_i \) \((x_k \leq a'_i \), and \( x_k \geq a'_i \), respectively) and the number of \((k - 1)\)-fixed objects. In other words, these are the probabilities (under uniform distribution) that a combinatorial object for which \( x_i = a_i \), \( 1 \leq i < k \), has the value in variable \( x_k \) which is \( a'_i \), \( \leq a'_i \), and \( \geq a'_i \), respectively.
Clearly $S(k, l) = S(k, \leq l) - S(k, \leq l - 1)$ and $S(k, \geq l) = 1 - S(k, \leq l - 1)$. Thus $S(k, l)/S(k, \geq l) = (S(k, \leq l) - S(k, \leq l - 1))/(1 - S(k, \leq l - 1))$. Therefore

$$S(k, \leq l) = S(k, \leq l - 1) + (1 - S(k, \leq l - 1))S(k, l)/S(k, \geq l).$$

Our method is based on the last equation. The large numbers can be avoided in cases when $S(k, l)/S(k, \geq l)$ 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 $[0, 1)$, let $l$ be chosen such that $S(1, \leq l - 1) < g < S(1, \leq l)$. Then $x_1 = a'_i$ and the first element of combinatorial object ranked $g$ is decided. To decide the second element, the interval $[S(1, \leq l - 1) \ldots S(1, \leq l)]$ containing $g$ can be linearly mapped to interval $[0 \ldots 1)$ to give the new value of $g$ as follows: $g \leftarrow (g - S(1, \leq l - 1))/(S(1, \leq l) - S(1, \leq l - 1))$. The search for the second element proceeds with the new value of $g$. Similarly the third, . . . , $m$-th elements are found. The algorithm can be written formally as follows, where $p'$ and $p$ stand for $S(k, \leq l - 1)$ and $S(k, \leq l)$, respectively.

```plaintext
procedure object(m, n, g);
    p' ← 0;
    for k ← 1 to m do {
        l ← 1;
        p ← S(k, 1);
        while p ≤ g do {
            p' ← p;
            l ← l + 1;
            p ← p' + (1 - p')S(k, l)/S(k, \geq l); 
        }
        x_k ← a'_i;
        g ← (g - p')/(p - p');
    }
```

Therefore the technique does not involve large integers iff $S(k, l)/S(k, \geq l)$ is not a large integer for any $k$ and $l$ 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 imprecisions for larger values of $m$ or $n$. The combinatorial object obtained by a computer implementation of above procedure may differ from the theoretically expected object. However, the same problem is present with other known methods including [21, 29, 33] (as discussed in [16]) and thus our method is comparable with others in that sense. Next, in applications, randomness is practically preserved despite computational errors. In case of designing adaptive parallel algorithms for generating combinatorial objects, these errors do affect the asymptotic behavior of the algorithm (it will be discussed later).

3 MAPPING $[0..1)$ INTO THE SET OF COMBINATIONS

In this section we investigate combinations of $m$ elements chosen from the set \{\(c_1, \ldots, c_n\). Without loss of generality, we assume $c_i = i$. We call these combinations
(m, n)-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 number of such combinations is
C(m, n) = n!/(m!(n - m)!).

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_n = 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-1} = 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-1+1} = x_i - j + 1, k \leq i \leq n. The (k - 1)-fixed com-
binations 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 C(m - k, n - j)/C(m - k + 1, n - j + 1) = (m - k + 1)/(n - j + 1).

Using the notation introduced in former section for any combinatorial objects,
let l = j - a_k - 1. Then, from Lemma 1 it follows that S(k, l)/S(k, \geq l) =
(m - k + 1)/(n - l - 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
\lceil gC(m, n) \rceil + 1. The procedure uses variable j instead of l, for simplicity.

**procedure** combination (m, n, g);
\n\begin{align*}
& \quad j \leftarrow 0; p' \leftarrow 0;
& \quad \textbf{for} k \leftarrow 1 \textbf{ to } m \textbf{ do } \{ \\
& \quad \quad j \leftarrow j + 1; \\
& \quad \quad p \leftarrow (m - k + 1)/(n - j + 1); \\
& \quad \quad \textbf{while} p \leq g \textbf{ do } \{ \\
& \quad \quad \quad p' \leftarrow p; \\
& \quad \quad \quad j \leftarrow j + 1; \\
& \quad \quad \quad p \leftarrow p' + (1 - p')(m - k + 1)/(n - j + 1) \\
& \quad \quad \quad x_k \leftarrow j; \\
& \quad \quad \quad g \leftarrow (g - p')/(p - p') \}
& \}
\end{align*}

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

4 COMBINATIONS WITH REPETITIONS

In [13] a correspondence between (m, n)-combinations and combinations with
repetitions of m out of n - m + 1 elements (where multiple choice of the same element
is possible) is established in the following way. Let z_i = x_i - i + 1. Then
PARALLEL COMBINATORIAL ALGORITHMS

1 ≤ z₁ ≤ z₂ ≤ ⋅⋅⋅ ≤ zₘ ≤ n - m + 1, and z₁, z₂, . . . , zₘ is a combination with repetitions of m out of n - m + 1 elements; we call it (m, n + 1) - r_combination. The number of (p, q) - r_combinations is R(p, q) = C(p + q - 1, p). Because of the simple relation, any algorithm that generates (m, n + m - 1)-combinations may be used to generate (m, n) - r_combinations, and vice versa. The only difference is in the output. Therefore, the procedure combination(m, n, g) can be used to generate at random combination with repetitions (by replacing instruction x₀ ← j with zₖ ← j - k + 1).

5 COMPOSITIONS OF n INTO m PARTS

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 x₁ + ⋅⋅⋅ + xₘ = n, where x₁ ≥ 0, . . . , xₘ ≥ 0. Sequential algorithms for generating such compositions appear in [29, 33]. There are two cases:

a) xᵢ > 0 for each i. Let y₁, . . . , yₘ be the following sequence: yᵢ = x₁ + ⋅⋅⋅ + xᵢ. The sequence y₁, . . . , yₘ is a combination of m out of n elements such that yₘ = n; in other words, it is a combination of m - 1 out of n - 1 elements from {1, . . . , n - 1}, and the number of compositions in question is CO(m, n) = C(m - 1, k - 1) [29, 33]. Each sequence x₁ . . . xₘ can easily be obtained from y₁ . . . yₘ since xᵢ = yᵢ - yᵢ₋₁ (with y₀ = 0).

b) xᵢ ≥ 0. By adding 1 to each part we get compositions of the number n + k and proceed as in case a).

Therefore the procedure combination described in former section can also be used to find the composition with ordinal number ⌊gCO(m, n)⌋ + 1.

6 RANDOM PERMUTATION

In this section we consider permutations of n elements. Without loss of generality we assume permutations over the set {1, . . . , n}. The number of permutations of n elements is P(n) = n!.

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 which determines the permutation x₁ . . . xₙ with index ⌊gP(n)⌋ + 1.

procedure permutation (n, g);
for j ← 1 to n do remlist.i ← i;
for k ← 1 to n do {
    l ← ⌊g(n - k + 1)⌋ + 1;
    xₖ ← remlist.l;
    for i ← l to n - k do remlist.i ← remlist.(i + 1);
    g ← g(n - k + 1) - l - 1
}
The procedure is based on the same choose and exchange idea as the one used in [10] 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 \([0 \ldots 1)\) coincide, the list of remaining elements is kept sorted which causes higher time complexity \( O(n^2) \) of the algorithm.

7 PERMUTATIONS OF COMBINATIONS

Permutations of combinations of the set \( \{1, 2, \ldots, n\} \) are sequences \( x_1 \ldots x_m \) such that \( x_i \neq x_j \) for \( i \neq j \) and \( 1 \leq x_i \leq n \) for \( 1 \leq i \leq m \). They can be obtained by taking all combinations and listing permutations for each combination. We will consider such an order which is lexicographic one. The number of permutations of combinations of \( m \) out of \( n \) elements is \( PC(m, n) = \frac{n!}{(n-m)!} \).

The permutation of combinations with ordinal number \( [gPC(m, n)] + 1 \) 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.

8 RANDOM \( t \)-ARY TREE

A \( t \)-ary tree is an ordered tree in which each node has at most \( t \) children. We adopt the binary and \( t \)-ary tree representation which are independently introduced in [32, 40] and used in [12]. A \( t \)-ary tree \( T \) having \( n \) nodes can be represented by a bitstring \( b_1b_2 \ldots b_n \) with \( (t-1)n \) zeros and \( n \) ones that satisfies the prefix property: the number of 0's is always less than or equal to \( (t-1) \) times the number of 1's while scanning from \( b_1 \) to \( b_n \) (i.e. \( b_1 + \cdots + b_i \geq i/t, 1 \leq i \leq tn \)). Here each 1 corresponds to a node of \( T \) while 0's correspond to the leaves of the extended tree \( e(T) \) obtained by replacing empty children of \( T \) with real nodes. Any such string encodes the extended \( t \)-ary tree, in preorder traversal. For example, binary trees with three nodes correspond to bitstrings 101010, 101100, 110010, 110100, and 111000, respectively (in lexicographic order).

A tree is regular if each node has either 0 or \( t \) children. A \( 1-1 \) correspondence between regular trees with \( tn + 1 \) nodes and \( t \)-ary trees with \( n \) nodes can be established by simply matching \( t \)-ary tree \( T \) with extended tree \( e(T) \). Given a regular tree \( P \), let \( r_p \) be the number of children of the root of \( P \), i.e. \( r_p = t \) or \( r_p = 0 \). Let \( P_1, \ldots, P_t \) denote the subtrees of \( T \). Two regular trees \( P \) and \( Q \) are in \( B \)-order, \( P < Q \), if:

1) \( r_p < r_Q \),
2) \( r_p = r_Q \), \( P_j = Q_j \) for \( j < i \), and \( P_i < Q_i \) for some \( i \leq t \).

Two \( t \)-ary trees \( T' \) and \( T'' \) are in \( B \)-order, \( T' < T'' \) if \( e(T') < e(T'') \). The sequential algorithms [12, 34, 39, 40] generate \( t \)-ary trees in \( B \)-order. In [40] it is proved that \( B \)-order of trees matches the lexicographic order of bitstring sequences \( b_1 \ldots b_n \).
We use the following lemma in our algorithm.

**Lemma 2** [12]  The number of bitstring sequences $b_{i+1} \ldots b_n$ containing $q$ ones and $tn - q - i$ zeros such that there exist a sequence $b_1 \ldots b_m$ satisfying the prefix property is $D(i, q) = (tn - q - i + 1)/(tn - i - q + 1) C(tn - i, q)$.

For instance, for $i = 0$ and $q = n$ one gets the number of $t$-ary trees with $n$ nodes $B(t, n) = C(tn, n)/((t - 1)n + 1)$.

Our method requires to determine $S(k, 1)$, $S(k, l)$, and $S(k, > l)$. Each element $b_k$ has two possible values, i.e. $b_k = a'_1 = 0$ or $b_k = a'_2 = 1$; thus it is sufficient to find $S(k, 1)$ and $S(k, > l)$. $S(k, > 1)$ is clearly equal to 1. To find $S(k, 1)$, note that the $(k - 1)$-fixed sequences $b_1 \ldots b_n$ satisfy the property in lemma for appropriate choice of $q$. Let the sequence $b_1 \ldots b_n$ contains $q$ ones; the number of such sequences is $D(k - 1, q)$. Furthermore, $D(k, q)$ of these sequences satisfy $b_0 = 0$. Then $S(k, 1) = D(k, q)/D(k - 1, q) = ((tn - q - k + 1)(tn - k - q + 2))/((tn - q) - k + 2)(tn - k + 1))$. This leads to the following simple algorithm which finds the $t$-ary tree $f(g)$ with the ordinal number $\lfloor gB(t, n) \rfloor + 1$.

**procedure** tree(t, n, g);

$p' \leftarrow 0; q \leftarrow n;$

for $k \leftarrow 1$ to $tn$ do {

$b_k \leftarrow 0;$

$p \leftarrow ((tn - q - k + 1)(tn - k - q + 2))/((tn - q) - k + 2)(tn - k + 1));$

if $p \leq g$ then {

$p' \leftarrow p;$

$b_k \leftarrow 1;$

$q \leftarrow q - 1;$

$p \leftarrow 1;$

$g \leftarrow (g - p')/(p - p')$

}

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

The algorithms [7, 27] generate a random binary tree without the use of large integers but use a series of random numbers; therefore they cannot be used to subdivide trees into groups (for example, to enable adaptive parallel algorithms). [7] also uses bitstring notation for binary trees (in fact, they solve the problem of random generation of parenthesis strings) and the algorithm is based on a similar property (for the special case of binary trees) as in this paper. This property is generalized here for $t$-ary trees. [27] uses a different representation of binary trees. In [26] a random $t$-ary tree generator is presented which also does not use large integers and is generalization of the algorithm [27].

9 RANDOM SUBSET

Any subset $S$ of a set of $n$ elements can be coded as a binary sequence $s_1 \ldots s_n$ where $s_i = 1$ if and only if $i$-th elements belongs to $S$. The number of subsets of a $n$-element set is $S(n) = 2^n$.

There is a fairly simple mapping procedure for subsets. 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 \( a_1 \ldots a_n \).

10 COMPOSITIONS OF \( n \) INTO ANY NUMBER OF PARTS

The sequence \( y_1 \ldots y_m \) and its relation with \( x_1 \ldots x_m \) are the same as defined in Section 5; however \( m \) is not fixed and thus compositions of \( n \) into any number of parts correspond to subsets of \( \{1, 2, \ldots, n - 1\} \) \([29]\). The number of such compositions is \( CM(n) = 2^{n-1} \).

Therefore the procedure described in former section can be also used to find the composition with ordinal number \( \lfloor gCM(n) \rfloor + 1 \).

11 RANDOM VARIATION

A variation of \( m \) out of \( n \) elements is any sequence \( x_1 \ldots x_m \) such that \( 1 < x_i < n \) for \( 1 < i < m \). Obviously there are \( V(m, n) = n^m \) such variations.

A mapping procedure for variations is a generalization of the one used for subsets. Let \( g = 0.a_1a_2 \ldots a_ma_{m+1} \ldots \) be the number \( g \) written in the number system with the base \( n \), i.e. \( 0 < a_i < n - 1 \) for \( 1 < i < m \). Then the variation indexed \( \lfloor gV(m, n) \rfloor + 1 \) is coded as \( (a_1 + 1)(a_2 + 1) \ldots (a_m + 1) \).

12 ADAPTIVE PARALLEL GENERATION ALGORITHMS

Recently, a number of parallel algorithms for generating combinatorial objects were proposed \([1, 2, 3, 4, 5, 6, 8, 9, 11, 15, 23, 24, 25, 37, 38]\) and others. They use various models and methods, but most of them are not adaptive, or are made adaptive at the cost of involving very large integers like \( n! \). Here an algorithm is \textit{adaptive} if it can run on a parallel model of computation with an arbitrary number \( k \) of processors.

Two general approaches for designing parallel generation algorithms are known in literature. In the first approach used there are arbitrary number of \( k \) processors available; each of them produces an interval of \( S/k \) objects, where \( S \) is total number of objects to be generated (for example, 1024 subsets of 10-element set are produced by 4 processors generating subsets 1–256, 257–512, 513–768, and 769–1024, respectively). The best known technique to follow this approach is to apply a sequential algorithm on each interval (i.e. for each processor), and is used in \([2]\) for generating permutations and combinations.

In the second approach \( m \) processors produce a \( m \)-element combinatorial object \( a_1a_2 \ldots a_m \) such that processor \( i \) is responsible for producing element \( a_i \). For example, subset \( \{2, 3, 5\} \) is produced in the following way: processor 1 produces 2; processor 2 produces 3; processor 3 produces 5. Using this approach, algorithms are designed to generate combinations \( m \) out of \( n \) elements in \([1, 4, 8, 24, 38]\), permutations of \( n \) elements \([5, 23]\), derangements \([3]\), subsets, equivalence relations and variations \([37]\),
and binary and t-ary trees [6]. In these papers, the algorithms are made adaptive by combining second approach with first one. There are two cases to consider:

i) \( k < m \): here each processor will do the job of \( m/k \) processors in the original algorithm (with \( m/k \) rounded appropriately if not an integer, so that the last processor does slightly less work);

ii) \( k \geq m \), and \( r = k/m \) (integer division): here the array is divided into \( r \) groups of \( m \) processors each, such that each group produces an interval of consecutive objects (if \( k/n \) is not an integer then \( k - rm \) processors may either be left without any job which will not change asymptotic behavior of the adaptive algorithm, or assume the appropriate portion of objects to generate in the way given in (i)).

Therefore in both approaches for generating combinatorial objects the set of all objects is divided into \( r \) (approximately) equal sets or groups. Each of the \( r \) groups will produce (approximately) \( \lfloor R(m, n)/r \rfloor \) objects. One can use a numbering system to find the initial and final object in each group. However, all known numbering systems ([17, 18, 19] and others) use large integers (up to \( R(m, n) \)) and are, for practical purposes, inefficient. All algorithms for generating combinatorial objects refer to such numbering systems (except [5] that has a method which does not deal with large integers).

In this paper we suggested a new numbering system for some kinds of combinatorial objects which do not deal with large integers. The new system finds, for given \( g \) such that \( 1 \leq g \leq n \), the combinatorial object \( x = x_1 \ldots x_m \) such that the ratio of the number of objects preceeding \( x \) and the total number of objects is \( \leq g \). In other words, it finds the object with the ordinal number \( \lfloor gR(m, n) \rfloor + 1 \).

The group \( j \) (\( 1 \leq j \leq r \)) will produce objects numbered from \( \lfloor (j - 1)R(m, n)/r \rfloor + 1 \) to \( \lfloor jR(m, n)/r \rfloor \). Thus we apply the mentioned algorithm for \( g = j/r \), \( 0 \leq j \leq r \), to find the beginning and the end of each group. Note that with such division and truncations during computation, the number of objects in each group to be generated is not strictly equal, but is balanced asymptotically.

The above procedure is sequential and is supposed to be done by one processor in each of the groups (in the second approach; in the first approach only one processor is responsible for each group). To avoid the need for more than constant space for this processor, one can decide that the processor which normally should produce the last element in any object (for a given group) will find the initial object as described. As soon as new element of the initial object is found, all currently known elements are shifted toward the first processor in given group. In this way there is no need to store the full object by the chosen processor. In case of permutations there is a list of the remaining elements. The list can be stored one element per processor, and the desired element can be extracted by a shift operation. This will require \( O(n) \) time per element, or \( O(n^2) \) time to find the initial permutation for a given group.

Using described method and results in previous sections, the algorithms [1, 2, 4, 5, 6, 24, 37, 38] can be made adaptive without using calculations with large integers. In all cases except [5] it is the only known way of making these algorithms adaptive. Note that [5] describes a procedure for adaptive generation of permutations that includes a subdivision step which is dependent on the generating algorithm while this paper presents a general procedure.
13 CONCLUSION

We were able to describe a general procedure which can be used to generate a random combination, permutation, binary and r-ary tree, subset, or variation without using large integers and by applying only one random number generator. The procedure is applicable in cases when there were suitable formulas for the number of objects in which some of them are already fixed. The linear time complexity is achieved in all cases except those involving permutations. It remains open problem to describe corresponding mapping procedures for other combinatorial objects for which our approach does not give satisfactory results. These are, for example, equivalence relations (set partitions), integer partitions, derangements etc. Also, our method may not apply if other ordering of objects studied here is considered. One example are subsets coded as combinations rather than bitstrings, ordered in lexicographic order.

References


