Generating Permutations of a Bag by Interchanges

CHUN WA KO *

Centrum voor Wiskunde en Informatica
Postbus 4079
1009 AB Amsterdam
the Netherlands
chun@cwi.nl

FRANK RUSKEY †

Department of Computer Science
University of Victoria, P. O. Box 1700
Victoria, B. C. V8W 2Y2 CANADA
fruskey@csr.uvic.ca

Abstract

We present algorithms for generating all permutations of a given bag so that successive permutations differ by the interchange of two elements. One version of the algorithm runs in time linear in the number of permutations.

Keywords: Analysis of algorithms, bag, multiset, permutation, interchange.

*Research supported by a Natural Sciences and Engineering Research Council Postgraduate Scholarship while at the University of Victoria.

†Research supported by the Natural Sciences and Engineering Research Council of Canada under grant A3379.

1 Introduction

A *bag* is a collection of not necessarily distinct elements. Early algorithms for generating all permutations of a bag were developed by Sag [11], Bratley [2], Chase [3], and Hu and Tien [5]. None of these algorithms runs in *constant amortized time* — i.e., time linear in the number of permutations generated. A permutation algorithm has the *interchange* property if successive permutations differ by the interchange of two elements. Ruskey and Roelants' algorithm [10] runs in constant amortized time but does not have the *interchange property*; Chase's algorithm [3] does not have the interchange property but runs in constant amortized time. We present an algorithm that has both properties.

The problem of developing a constant amortized time algorithm for generating bag permutations is given as an exercise by Reingold, Nievergelt, and Deo [9]. However, the solutions manual [4] presents Hu and Tien's algorithm [5] and references two other algorithms, all of which do not run in constant amortized time.

We consider bags over the elements $0, 1, ..., t$. We represent a bag by a sequence $n = \langle n_0, \ldots, n_t \rangle$, where each n_i is the number of occurrences of i in the bag. Also, $\#n$ denotes the size of the bag, or the sum of the n_i. If $n_i > 0$, $n|_i$ denotes sequence n with n_i decremented by one:

$$n|_i \;=\; \langle n_0, \ldots, n_{i-1}, n_i - 1, n_{i+1}, \ldots, n_t \rangle.$$

An *n-permutation* is a permutation of bag n. The number of n-permutations is the bag coefficient $C(n)$.

$$C(n) \;=\; \binom{\#n}{n_0, n_1, \ldots, n_t} = \frac{\#n!}{n_0! n_1! \cdots n_t!}$$

These coefficients satisfy the following recurrence relation.

$$C(n) \;=\; \sum_{j=0}^{t} C(n|_i) \tag{1}$$

2 The Algorithms

The bag permutations can be generated by the recursive algorithm that follows from the familiar proof of recurrence relation (1) for bag coefficients; it classifies the permutations according to their first symbol. This leads to algorithm *GenBag* below. Operator $\oplus$ denotes concatenation. The call $GenBag(n, \epsilon)$, where ϵ denotes the empty sequence, generates all n-permutations.

```
     {Print all sequences πα where α is an n-permutation}
(G1) procedure GenBag ( n : array [0..t] of natural; π : sequence );
(G2) var j : 0..t;
(G3) begin
(G4)     if n_0 = n_1 = ··· = n_t = 0 then Print( π )
(G5)     else for j ∈ {p | 0 ≤ p ≤ t ∧ n_p > 0} do
(G6)         GenBag( n|_j, π ⊕ j )
(G7) end
```

Depending on the order in which the j's are selected by the **for** statement in line (G5), different recursion trees arise, each of which leads to a different sequence of permutations. Selecting the elements in increasing order prints a lexicographic list of the n-permutations. These recursion trees are in one-to-one correspondence with what we call bag trees.

DEFINITION 1 *A labeled ordered tree is an* (n, s)-tree *if its root is labeled* s *and, for each* $n_i > 0$*, its root has a child that is the root of an* $(n|_i, i)$*-tree.*

When we do not wish to specify the label at the root, we write simply n-tree. An n-tree is also called a *bag tree*. We regard bag trees as being embedded in the plane with the root on the left and leaves on the right. The ordering of subtrees is from top to bottom. To each n-tree T there is a corresponding list $L(T)$ of all n-permutations; this list is precisely the list of permutations printed by *GenBag*. Each permutation can be obtained by traversing T from its root to a leaf and recording the labels of the nodes encountered.

DEFINITION 2 *A bag tree* T *is an* interchange tree *if successive permutations in list* $L(T)$ *differ by an interchange of two elements.*

The following examples illustrate the preceding definitions. For $n = \langle 2, 1\rangle$, there are exactly 4 bag trees. The list 001, 100, 010 is not $L(T)$ for any bag tree T. For $n = \langle 2, 2\rangle$, there is a unique bag tree T such that $L(T) =$ 0011, 0110, 0101, 1010, 1001, 1100; however, T is not an interchange tree.

We call a node of a bag tree a *plus-node* if its children are labeled in increasing order and a *minus-node* if they are labeled in decreasing order. A node with only one child is both a plus-node and a minus-node.

DEFINITION 3 *An* alternating tree *is a bag tree in which every internal node is either a plus-node or a minus-node and, at each level of the tree, reading top-to-bottom, plus- and minus-nodes alternate.*

We now prove a sufficient condition for a bag tree to be an interchange tree and use it to derive an interchange algorithm.

THEOREM 1 *Any alternating tree T is also an interchange tree.*

PROOF: Let $\pi = \alpha a\beta c\gamma$ and $\pi' = \alpha b\beta d\delta$ be two successive permutations in $L(T)$, where $a, b, c, d \in \{0, 1, \ldots, t\}$, $a \neq b$, $c \neq d$, and $\alpha, \beta, \gamma, \delta \in \{0, 1, \ldots, t\}^*$. In words, the first symbols that differ are a and b and the next symbols that differ are c and d. The nodes labeled a and b have a common parent; call the tree rooted at their parent an m-tree — thus, like n, m is a sequence that represents a bag. Nodes c and d have different parents; call the tree rooted at c's parent a p-tree and the tree rooted at d's parent a q-tree. There are two symmetric cases, depending upon whether $a < b$ or $a > b$. We discuss only the case $a < b$.

Suppose $a < b$. Since children are labeled monotonically, $m_i = 0$ for all $a < i < b$. This implies $p_i = 0$ and $q_i = 0$ for all $a < i < b$. Since β is common to both π and π', $p_i = q_i$ for $i \neq a$ and $i \neq b$; further, $q_a = p_a + 1$ and $p_b = q_b + 1$. Because T is alternating, the nodes labeled c and d both have either the largest or the smallest labels among their respective siblings.

In the "smallest" case, we have $c = \min\{i \mid p_i > 0\}$ and $d = \min\{i \mid q_i > 0\}$. We argue that $d = a$ — the proof that $c = b$ is similar.

$$
\begin{array}{cl}
 & d \\
= & \quad \langle d \text{ is the smallest sibling} \rangle \\
 & \min\{i \mid 0 \leq i \ \wedge \ q_i > 0\} \\
= & \quad \langle \text{Since } p_i = q_i \text{ for } i < a, \text{ if some such } q_i > 0 \text{ then } c = d; \text{ but } c \neq d \rangle \\
 & \min\{i \mid a \leq i \ \wedge \ q_i > 0\} \\
= & \quad \langle q_a = p_a + 1 > 0 \rangle \\
 & a
\end{array}
$$

The "larger" case is similar and is omitted. Hence, $a = d$ and $b = c$ in either case. Also, $p|_c = q|_d$ and thus, because T is alternating, $\gamma = \delta$. □

An interchange tree need not be an alternating tree, since permuting the symbols $0, 1, \ldots, t$ changes an interchange tree into an interchange tree but may not maintain the alternating-tree property.

We remove some of the non-determinism in algorithm *GenBag* so that its recursion tree is an alternating tree. We introduce a global boolean array $d[0..\#n - 1]$, where $d_i \equiv$ "the current node at level i is a plus-node". By $|\pi|$ we denote the number of elements in sequence π. The qualifier "**by** d_k" in the **for** statement of line (T5) indicates whether the elements of the set are selected in increasing order (d_k is true) or decreasing order (d_k is false). For the initial call $GenAlt(n, \epsilon)$, the sequence produced will depend on the initial value of array d. As the algorithm is executing n is modified; let n' be the initial sequence n.

```
{Array d is global. Print all sequences πα where α is an n-permutation}
procedure GenAlt ( n : array [0..t] of natural; π : sequence );
var j : 0..t; k : 0..#n' − 1;
begin
    if n_0 = n_1 = ··· = n_t = 0 then Print( π ) else begin
        k := |π|;
        for j ∈ {p | 0 ≤ p ≤ t ∧ n_p > 0} by d_k do
            GenAlt( n|_j, π ⊕ j );
        Change( d_k )
    end
end
```

The call *Change*(d_k) negates d_k if there is more than one i such that $n_i > 0$. Otherwise, *Change* can either leave d_k unchanged or negate it arbitrarily; the algorithm is still an interchange algorithm.

Different choices of *Change* give different versions of *GenAlt*. Two particular choices, which we call Version A and Version B, are specified below.

In Version A, *Change*(d_k) is $d_k :=$ **not** d_k.

In Version B, *Change*(d_k) is **if** $\max\{n_1, \ldots, n_t\} > 0$ **then** $d_k :=$ **not** d_k.

Figure 1 shows the recursion tree for Version A with input $n = \langle 2, 1, 1 \rangle$. The $+$'s and $-$'s indicate whether the nodes are plus-nodes or minus-nodes. Figure 2 is the corresponding tree for Version B; the squares indicate nodes at which d_k does not change. In both figures we assume that, for all $k \in 0..n-1$, the initial value of d_k is true.

In Figure 3 we show an implementation of Version A where π, d, and n are global; then the only parameter of the procedure is k, which is the level of the recursion. The initial call is *GenAltA*(0).

3 Efficient Implementation

We now show how to implement *GenAlt* to achieve constant amortized time. Our algorithm, given in Figure 4, follows the general approach of Ruskey and Roelants van Baronaigien [10]. It requires that the symbols have been arranged so that sequence n is in descending order: $n_0 \geq n_1 \geq \cdots \geq n_t$. Next, it requires eliminating certain nodes in the recursion tree. Define an *i-path* as a path in the recursion tree, all of

			$1+$	2	$= 0012$
		$0+$			
			$2-$	1	$= 0021$
			$2+$	$\underline{0}$	$= 0120$
	$0+$	$1-$			
			$0-$	2	$= 0102$
			$0+$	1	$= 0201$
		$2+$			
			$1-$	$\underline{0}$	$= 0210$
$+$					
		$2-$	$\underline{0}+$	$\underline{0}$	$= 1200$
	$1-$		$0-$	2	$= 1002$
		$0+$			
			$2+$	$\underline{0}$	$= 1020$
			$1-$	$\underline{0}$	$= 2010$
		$0-$			
	$2+$		$0+$	1	$= 2001$
		$1+$	$\underline{0}-$	$\underline{0}$	$= 2100$

Figure 1: Recursion Tree (A) for $n = \langle 2, 1, 1 \rangle$.

			$1+$	2	$=$	0012
		$0+$				
			$2-$	1	$=$	0021
			$2\square$	0	$=$	0120
	$0+$	$1-$				
			$0+$	2	$=$	0102
			$0-$	1	$=$	0201
		$2+$				
			$1\square$	0	$=$	0210
$+$						
		$2\square$	$0\square$	0	$=$	1200
	$1-$		$2\square$	0	$=$	1020
		$0-$				
			$0+$	2	$=$	1002
			$0-$	1	$=$	2001
		$0+$				
	$2+$		$1\square$	0	$=$	2010
		$1\square$	$0\square$	0	$=$	2100

Figure 2: Recursion Tree (B) for $n = \langle 2, 1, 1 \rangle$.

$\{$Given $k = \#n' - \#n$,
print all sequences $\pi[0..k-1]\alpha$ where α is an n-permutation.$\}$

procedure *GenAltA*($k : 0..\#n' - 1$);
var $j : 0..t$;
begin
    if $n_0 = n_1 = \cdots = n_t = 0$ **then** Print(π)
    else begin
        for $j \in \{p \mid n_p > 0\}$ **by** d_k **do begin**
            $\pi_k := j$;
            $n_j := n_j - 1$;
            GenAltA($k+1$);
            $n_j := n_j + 1$
        end;
        $d_k :=$ **not** d_k $\{Change(d_k)\}$
    end
end

Figure 3: Version A of Algorithm *GenAlt*.

whose nodes are labeled i and have degree at most one. (The nodes on 0-paths are underlined in Figure 1.) We will remove all 0-paths from the recursion tree.

For example, consider the case $n = \langle m, 1, 1\rangle$. The unpruned recursion tree has $(m+5)(m+3)(m+1)/3$ nodes with $(m+2)(m+1)$ leaves. Deleting 0-paths reduces the number of nodes to $(2m+5)(m+1)$.

There are two subtle, but crucial, differences from *GenAltA*. The first is that the test at line (B4) does not include n_0; this causes 0-paths to be eliminated. The second difference is that $\pi[k..\#n-1]$ contains all zeroes upon a call *GenAltB*(k) (which requires that π contain all zeroes at the call *GenAltB*(0)). This restriction allows us to conclude that if all n_i are 0 except for n_0, then π contains the permutation to be printed. Note that this requirement is satisfied at the recursive call within *GenAltB*. The assignment $\pi_k := 0$ is needed because π should be left unchanged by execution of a call to *GenAltB*.

In order to achieve constant amortized time behavior, maintain the nonzero n_i's (only) as a doubly linked list. Then the test $n_1 = n_2 = \cdots = n_t = 0$ can be done in constant time (it is equivalent to the length of the list being 1), and the loop can be executed in time proportional to the number of iterations, except for recursive calls. Thus the computation time is proportional to the number of recursive calls. We now show that the number of recursive calls is at most four times the number of

{Given $k = \#n' - \#n$ and $\pi[k..\#n' - 1]$ are all zeroes,
print all sequences $\pi[0..k-1]\alpha$ where α is an n-permutation.}

procedure *GenAltB* (k : $0..\#n'$);
var j : $0..t$;
begin
    if $n_1 = n_2 = \cdots = n_t = 0$ **then** Print(π)
    else begin
        for $j \in \{p \mid n_p > 0\}$ **by** d_k **do begin**
            $\pi_k := j$;
            $n_j := n_j - 1$;
            GenAltB($k+1$);
            $n_j := n_j + 1$;
            $\pi_k := 0$
        end;
        $d_k :=$ **not** d_k {$Change(d_k)$}
    end
end

Figure 4: Version B of Algorithm *GenAlt*.

bag permutations printed. This will complete the argument that the algorithm runs in constant amortized time.

The following two bag coefficient identities will prove useful. The first identity is classic and is simply a restatement of (1); the second may be verified by induction on n_0.

$$\sum_{i=0}^{t} \binom{\#n-1}{n_0,\ldots,n_{i-1},n_i-1,n_{i+1},\ldots,n_t} = \binom{\#n}{n_0,n_1,\ldots,n_t} \tag{2}$$

$$\sum_{i=0}^{n_0} \binom{\#n-i}{n_0-i,n_1,\ldots,n_t} = \frac{\#n+1}{n-n_0+1}\binom{\#n}{n_0,n_1,\ldots,n_t} \tag{3}$$

The number of nodes of degree one in the recursion tree T of GenAltB is equal to the number of nodes on i-paths in T, where $i > 0$. The number of nodes at level $\#n - j + 1$ (with the root at level 0) on i-paths in T is the same as the number of bag permutations specified by

$$n_0, n_1, \ldots, n_{i-1}, n_i - j, n_{i+1}, \ldots, n_t.$$

For the example of Figure 2, in Version B the number of nodes on 2-paths at level 4 is 3 (corresponding to 001, 010, and 100), and there are no 2-paths at level 3. Thus, the total number of nodes of degree one is given by the following expression.

$$\begin{aligned}
&\sum_{i=1}^{t}\sum_{j=1}^{n_i} \binom{\#n-j}{n_0,\ldots,n_{i-1},n_i-j,n_{i+1},\ldots,n_t} \\
=\ &\sum_{i=1}^{t} \frac{\#n}{\#n-n_i+1}\binom{\#n-1}{n_0,\ldots,n_{i-1},n_i-1,n_{i+1},\ldots,n_t} \\
\leq\ &\frac{\#n}{\#n-n_1+1}\binom{\#n}{n_0,n_1,\ldots,n_t} \leq 2\binom{\#n}{n_0,n_1,\ldots,n_t}
\end{aligned} \tag{4}$$

All other internal nodes of the tree have degree greater than one, so there cannot be more of them than there are leaf nodes. Thus, the total number of nodes in the tree is at most $4C(n)$.

Finally, we remark that it is also possible to implement Version A in constant amortized time by keeping track of how many 0-paths have been deleted so far at any given level. This does, however, greatly complicate the algorithm. Details may be found in Ko [6].

4 Concluding Remarks

When all n_i equal 1, Versions A and B produce the same list of $\#n!$ permutations, and this list appears to be different than any of those produced by the permutation generation algorithms surveyed in Sedgewick [12] or Lipski [7].

The proof technique used in [1], [9] to show the interchange property is different than that used here. Their proof is inductive and is based on the starting permutation being $0^{n_0}1^{n_1}$ and the ending permutation being $10^{n_0}1^{n_1-1}$. For $t > 1$ the ending permutations are not so easy to specify. For example, 303131122233 and 3033220212 are both ending permutations of (starting from $01^3 2^3 3^5$ and $0^2 1 2^4 3^3$, respectively) Version B.

Finally, a ranking algorithm for Version A, similar to the ranking algorithm presented by Lucas, Roelants van Baronaigien, and Ruskey [8], as well as an unranking algorithm can be developed; details may be obtained from the authors.

5 Acknowledgement

We wish to thank David Gries for his many contributions to the presentation of this paper, for pointing out an error in our original submission, and for reference [7].

References

[1] J.R. Bitner, G. Ehrlich, and E.M. Reingold. Efficient generation of the binary reflected gray code and its applications. *Comm. ACM*, 19:517–521, 1976.

[2] P. Bratley. Algorithm 306: Permutations with repetitions. *CACM*, 10:450, 1967.

[3] P.J. Chase. Algorithm 383: Permutations of a set with repetitions. *CACM*, 13:368–369, 1970.

[4] J.A. Fill and E.M. Reingold. *Solutions Manual for Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.

[5] T.C. Hu and B.N. Tien. Generating permutations with nondistinct items. *AMM*, 83b:629–631, 1976.

[6] C.W. Ko. Generation of permutations of multisets. Master's thesis, U. Victoria, Dept. Computer Science, 1985.

[7] W. Lipski. More on permutation generation methods. *Computing*, 23:357–365, 1979.

[8] J. Lucas, D. Roelant van Baronaigien, and F. Ruskey. On rotations and the generation of binary trees. Technical Report DCS-131-IR, submitted, U. Victoria, 1990.

[9] E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.

[10] F. Ruskey and D. Roelants van Baronaigien. Fast recursive algorithms for generating combinatorial objects. *Congressus Numerantium*, 41:53–62, 1984.

[11] T.W. Sag. Algorithm 242: Permutations of a set with repetitions. *CACM*, 7:585, 1964.

[12] R. Sedgewick. Permutation generation methods. *Computing Surveys*, 9:137–164, 1977.