An $O(1)$ Time Algorithm for Generating Multiset Permutations

Tadao Takaoka

Department of Computer Science, University of Canterbury
Christchurch, New Zealand
E-mail: tad@cosc.canterbury.ac.nz

Abstract. We design an algorithm that generates multiset permutations in $O(1)$ time from permutation to permutations, using only data structures of arrays. The previous $O(1)$ time algorithm used pointers, causing $O(n)$ time to access an element in a permutation, where n is the size of permutations. The central idea in our algorithm is tree traversal. We associate permutations with the leaves of a tree. By traversing this tree, going up and down and making changes when necessary, we spend $O(1)$ time from permutation to permutation. Permutations are generated in a one-dimensional array.

1 Introduction

Algorithms for generating combinatorial objects, such as (multiset) permutations, (multiset) combinations, well-formed parenthesis strings are a well studied area and many results are documented in Nijenhuis and Wilf [6], and Reingold, Nievergelt, and Deo [8], etc.

Let n be the size of the objects to be generated. The most primitive algorithms are recursive ones for generating those objects in lexicographic order, causing $O(n)$ changes from object to object, and thus $O(n)$ time. To overcome this drawback, many algorithms were invented, which generate objects with a constant number of changes, $O(1)$ changes, from object to object. This idea of generating combinatorial objects with $O(1)$ changes is named "combinatorial Gray codes", and a good survey is given in [11]. In many cases, these changes are made by swappings of two elements, that is, two changes. It is still easy to design recursive algorithms for combinatorial generation with $O(1)$ changes, since we can control the paths of the tree of recursive calls and thus we can rather easily identify changing places. Note that combinatorial objects correspond to the leaves of the tree, meaning that it takes $O(n)$ time from object to object as the height of the tree is n. Further to overcome this shortcoming, several attempts were made to design iterative algorithms, which are called loopless algorithms in some ltera iterature, removing recursion, so that $O(1)$ time is achieved from object to object. At this stage, we need some care in defining the $O(1)$ time from object to object. In Korsh and Lipschutz [3], $O(1)$ time was achieved to generate multiset permutations, whose algorithm is a refinement of that by Hu and Tien [1]. In this algorithm, multiset permutations are given one after another in a linked list. The operations on the list are manipulated by pointers, involving shift operations in $O(1)$ time. For example, the list $(1, 1, 1, 2, 2, 2)$ with $n = 6$ can be converted to $(2, 2, 2, 1, 1, 1)$ in $O(1)$ time by changing pointers. We assume that the above conversion takes $O(n)$ time in this paper, and we claim that multiset permutations can be generated in $O(1)$ time using arrays, not pointers.

This kind of strict requirement for $O(1)$ time was demonstrated in the recent development in parenthesis strings generation. An $O(1)$ change algorithm was developed in Ruskey and Proskurowski [10] and an $O(1)$ time algorithm with pointer structures was achieved in Roelants van Baronaigien [9], and they challenged the readers, asking whether there could be $O(1)$ algorithms with arrays, whereby stricter $O(1)$ time could be achieved. This problem was recently solved by three independent works of Mikawa and Takaoka [5], Vajnowski [13], and Walsh [14]. Note that we can access any element of a combinatorial object in $O(1)$ time in array implementation, whereas we need $O(n)$ time in linked list implementation, as we must traverse the pointer structure. The algorithm by Ko and Ruskey [2] generates multiset permutations with swappings of two elements, but not with $O(1)$ time from permutation to permutation.

The main idea of $O(1)$ time for multiset permutation generation in this paper is tree traversal. The generation tree for a set of permutations, arranged in some order, on the given multiset is a tree whose paths to the leaves correspond to the permutations. Basically we traverse the tree in movements of (up, cross, down). The move "up" is to go up the tree from a node to one of its ancestors. The move "cross" is to move from a node to its adjacent sibling, causing a swapping with the element at that level and the one at a level closer to the leaf. The move "down" is to go down from a node to one of its descendants, which we call the landing point. The landing point has no sibling and the path to the leaf has no branching, causing a straight line. It is important that we avoid traversing this straight line node by node. The core part of the algorithm is centered on how to compute the positions to which we go up and down, and where we should perform swappings. Although the use of tree structure for combinatorial generation was originated in Lucas [4] and Zerling [15], and well known, the technique of tree traversal in this paper is new.

Since the final algorithm is rather complicated, we go through a stepwise refinement process, going from simple structures to details. In Section 2, we define the generation tree and design a recursive algorithm that traverses this tree to generate multiset permutations. We give a formal proof of the recursive algorithm. In Section 3, we design an iterative algorithm based on the recursive algorithm. We first describe an informal framework for an iterative algorithm, and translate the recursive algorithm into an iterative one guided by the framework. The resulting iterative algorithm generates multiset permutations in $O(1)$ time in a one-dimensional array. As additional data structures, we use a few more arrays, causing $O(kn)$ space requirement, where k is the number of distinct elements in the multiset. In Section 4, we give details of some informal descriptions in Section 3. In Section 5, we give concluding remarks. We give a full Pascal code as an appendix at the end for the readers' inspection.

2 Permutation tree and recursive algorithm

We denote a multiset by [...] and ordinary set by $\{...\}$. Those notations identify operations such as set union and set subtraction when the same symbols are used on sets and multisets. We convert a multi-set S to the set $set(S)$ by removing repetition of each element. If $S = [1, 1, 2]$, for example, $set(S) = \{1, 2\}$. Let a multiset $S = [1, ..., 1, 2, ..., 2, ..., k, ..., k]$ be defined by $(m_1, m_2, ..., m_k)$, where m_i is the multiplicity of i. Let P be a set of all multiset permutations on S arranged in some order. Since S is the base multiset for P, we use the notation $base(P)$ $= S$. We use word "permutation" for "multiset permutation" for simplicity. Let $N = n!/(m_1!...m_k!)$. Then we have $|P| = N$. Let $x \in P$ be a permutation given by $x = a_1a_2...a_n$. We construct the permutation tree of P, $T(P)$, in such a way that each $x \in p$ is associated with a path from the root to a leaf. Since the path from the root to a leaf is unique in a tree, x will also correspond to the leaf at the end of the path. If x' is the next permutation of x in P, we correspond x' to the next leaf of that for x. Let x' be given by $x' = a_1...a_ia'_{i+1}...a'_n$. That is, x' shares some prefix (possibly empty) with x. Then the paths to the two adjacent leaves x and x' share the path corresponding to $a_1...a_i$.

Example 1. Let S be given by $(m_1, m_2, m_3) = (1, 2, 2)$. We give P and $T(P)$ in the next page.

In this example, we assume we give permutations in P in this order. The number shown by (i) to the right side of each permutation is to indicate the ith permutation. This list of permutations also gives the shape of the tree $T(P)$. The root at level 0 has three branches leading to sibling nodes at level 1 with labels 1, 2, and 3. Then the node at level 1 with label 1 has two branches leading to sibling nodes with labels 2 and 3, etc. We have $5!/(1!2!2!) = 30$ members in P. We draw the tree horizontally, rather than vertically, for notational convenience.

We use a list $nodes[i]$ of elements from the set $set(S)$ of a multiset S to keep track of siblings at level i, We define two types of operation with notation $\Leftarrow$. Operation $c \Leftarrow nodes[i]$ means that the first element of $nodes[i]$ is moved to a single variable c. Operation $nodes[i] \Leftarrow c$ means that c is appended to the end of $nodes[i]$. The history of variable c keeps track of all elements in $nodes[i]$. For a list L, $set(L)$ is the set made of elements taken from L. $Next(L)$ is the second element of L. We identify nodes of the tree by array elements of a whenever possible from context. A recursive algorithm is given below.

ALGORITHM 1 *Recursive algorithm*

```
procedure generate(i);
var t, s;
begin
```

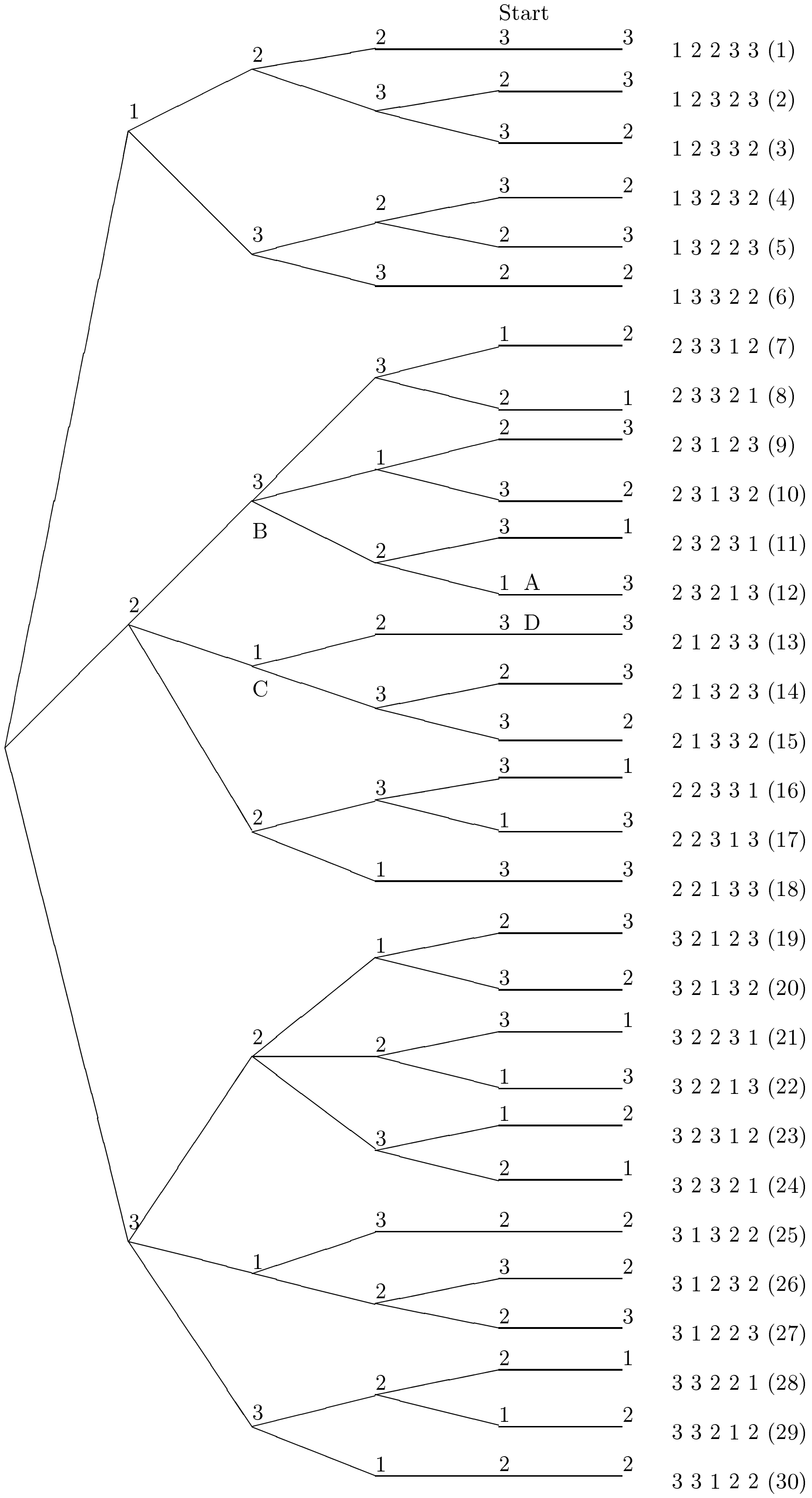

Fig. 1. Generation tree for permutations on [1,2,2,3,3]

```
4.     $nodes[i] := (a[i])$;
5.     if $i \leq n$ then
6.     repeat
7.         $generate(i+1)$;
8.         Let $s$ be the leftmost position of $next(nodes[i])$ in $a$ such that $i < s$
9.         if $a[i]$ is not a last child then $swap(a[i], a[s])$;
10.        if $a[i-1]$ is a first child then
11.            if $a[i] \neq a[i-1]$ then $nodes[i-1] \Leftarrow a[i]$;
12.        $t \Leftarrow nodes[i]$
13.    until $nodes[i] = \emptyset$
14. end;
15. begin { main program }
16.     Let $a = [1, ..., 1, 2, ..., 2, ..., k, ..., k]$;
17.     $generate(1)$
18. end.
```

Let $tail(a)$ be the consecutive portion of the tail part of a such that all elements in $tail(a)$ are equal to $a[n]$. Let Q be a set of all permutations generated from $S - [a_1, ..., a_i]$. Then the notation $a_1...a_iQ$ means the set of permutations made by concatenating $a_1...a_i$ with all members of Q. We use notations a_i and $a[i]$ interchangeably to denote the i-th element of array a. We state the following obvious lemmas.

Lemma 1. *Let P be the set of permutations on the multiset S of size n and $first(P) = \{x_1 | x_1x_2...x_n \in P\}$. Then $first(P) = set(S)$.*

Lemma 2. *Let S be a multiset and P be the set of permutations on S. Then $P = b_1Q_1 \cup ... \cup b_lQ_l$, where $set(S) = \{b_1, ..., b_l\}$ and Q_j is the set of permutations on the multiset $S - [b_j]$.*

Theorem 1. *Algorithm 1 generates all permutations on S.*

Proof. We show by backward induction that $generate(i)$ generates the set P of all permutations on $[a[i], ..., a[n]]$. The case of $i = n$ is obvious. Suppose the theorem is true for $i+1$. Then observe that the first call of $generate(i+1)$ in $generate(i)$ will generate all permutations on $[a[i+1], ..., a[n]]$ by induction, which we denote by Q. From Lemma 1, it holds that $first(Q) = set(a[i+1], ..., a[n])$. From lines 10-11 of the program we have $set(nodes[i]) = \{a[i]\} \cup first(Q) = set[a[i], ..., a[n]]$ immediately after the first call of $generate(i+1)$. Let $set[a[i], ..., a[n]] = \{b_1, ..., b_l\}$ for some l such that $b_1 = a[i]$ at the beginning of $generate(i)$. Then we are generating $a[1]...a[i-1]b_jQ_j$ for $j = 1, ..., l$, where $base(Q_j) = base(Q_{j-1}) - [b_j] \cup [b_{j-1}]$ for $j > 1$, and $Q_1 = Q$. Since we swap $a[i]$ and $a[s]$ at the end of each call of $generate(i+1)$, l different multisets are given in $(a[i+1], ..., , a[n])$ as $base(Q_j)$ before calls of $generate(i+1)$. From Lemma 2, we conclude that the set P is generated by calling $generate(i+1)$ with all b_j given in t.

Example 2. Let $i = 1$, and suppose we start from $a = [1, 2, 2, 3, 3]$. Then we have $base(Q) = [2, 2, 3, 3]$, and $first(Q) = \{2, 3\}$. Since these elements are appended to $nodes[i] = (1)$, we have $nodes(1) = (1, 2, 3)$, which forms the $first(P)$, where P is the entire set of permutations.

Note that the choice of position s at line 9 can be arbitrary as long as we choose a position s such that $next[nodes[i]] = a[s]$ and $i < s$. Two consecutive permutations before and after *swap* are different only at i and s such that $i < s$. In this context, we say i is the difference point and s is the solution point. In Example 1, the permutations on [1, 2, 2, 3, 3] are generated by this algorithm.

3 $O(1)$ implementation

Algorithm 1 takes $O(n)$ time from permutation to permutation due to its recursive structure. In this section we avoid this $O(n)$ overhead time for traversing the tree. By using some data structures, we jump from node to node in the permutation tree.

When we first call $generate(1)$, it will go down to level n and come back to level $i = n - tail(a) + 1$ without doing any substantial work, since all nodes on this path are last children. At this level the algorithm append $a[i] = k$ to $nodes[i - 1]$ and comes to level $i = n - tail(a)$, that is, i is decreased by 1. Then it swaps $a[i]$ and $a[i + 1]$, add new $a[i]$ to $next[i - 1]$, and go down to level n. When the algorithm traverses the tree downwards and upwards, there are many steps that can be avoided. Specifically we can start from level $i = n - tail(a) + 1$. After we perform swapping, we can come down straight to level $i = n - tail(a) + 1$ with the new $tail(a)$. We keep two arrays *up* and *down* to navigate our traversal in the tree; $up[i]$ tells where to go up from level i and $down[i]$ tells where to go down from level i. Level $up[i]$ is the level where we hit a non-last child when we traverse the tree from level i. The formal definition of $down[i]$ is given later.

When we perform $swap(a[i], a[s])$, we need the information of s at hand without computing the leftmost position of $next(nodes[i])$ to the right of i. Obtaining this information is carried out when we come to a last child at level i by updating $s[up[i]]$ by i if $a[i] = next(nodes[up[i]])$ for the first time. For this purpose, the variable s is given by array s to keep the information of s for each level.

Example 3. In Figure 1, we can start from the point *Start*. Suppose we reached the point A after several steps. We have $up[4] = 2$, which we inherited from $up[3]$. Since $next(nodes[2]) = 1$, we set $s[up[4]] = 4$. We make transition $A \rightarrow B \rightarrow C \rightarrow D$. When we cross from B to C, we swap $a[2]$ and $a[4]$ and come to the landing point D.

We translate Algorithm 1 into the following informal iterative algorithm for traversing the tree, resulting in Algorithm 3.

Algorithm 2 *Informal iterative tree traversal*

```
initialize a to be the first permutation on S;
initialize $up[i]$ and $down[i]$ to $i$ for $i = 0, ..., n$;
initialize $nodes[i]$ to $(a[i])$ for $i = 1, ..., n$;
$i := n - |tail(a)| + 1$;
repeat
    if $nodes[i-1]$ has not been updated by $a[i-1]$'s children then update it;
    output(a);
    if $a[i]$ is not a last child then $swap(a[i], a[s[i]])$; {action cross}
    update $nodes[i-1]$;
    if $a[i]$ is a last child then begin
        $up[i] := up[i-1]$; $up[i-1] := i-1$;
        update $s[up[i]]$;
        update $down[up[i]]$;
        if $i = n - |tail(a)| + 1$ {$a[i], ..., a[n]$ form a straight line}
            then $i := up[i]$; {going up}
            else $i := down[i]$; {going down}
    end
    else {$a[i]$ is not a last child}
        $i := down[i]$ {going down}
until $i = 0$ {root level}.
```

As we cross from a node to the next, swapping two array elements, $tail(a)$ grows or shrinks. For the computation of $tail(a)$, which, in turn, gives the information of *down*, we use array *run*. Array *run* is to keep track of the length of consecutive array elements that are equal to $a[i]$ when we traverse the path of last children starting at $a[up[i]]$. Array *run* is computed by increasing $run[up[i]]$ by 1 when we hit $a[up[i]] = a[i]$ on the path, and reset to 0 otherwise. These values of *run* are used to compute $tail(a)$ after we perform the swap operation, whereby we can compute the values of *down*. Specifically we can set $down[up[i]] := i - run[up[i]]$ if $a[up[i]] = a[n]$ and $down[i] = i + 1$. Note that $down[i] = i + 1$ means $a[i+1] = ... = a[n]$, since the landing point is the left end of $tail(a)$. In other cases, $down[up[i]]$ is set to $i + 1$ or i depending on the situation at level i, as described in the line-by-line explanation.

The Boolean value of $mark[i]$ is to show that the value of $down[i]$ has been set and prevent further modification.

If we hit a non-last child we always go down guided by $down[i]$. If we hit a last child, we may go down or go up, if $i < down[i]$ or $i = down[i]$ respectively.

When we go up to the ancestor, the path to the node on which we stand consists of last children. We call this path the current path. When we go down from a node to a descendant, the path from the node to the descendant consists of first children. We call this path the opposite path. Most of the work in the algorithm is to prepare the necessary environment for the opposite path when we are traversing the current path. We jump over the opposite path from the left end to the landing point, whereas we traverse the current path node by node. Whenever we come to a node, the necessary information for the next action must be ready. We leave the details of the data structure $nodes[i]$ in Section 4.

Example 4. In Fig.2, $run(up[i]) = 2$ for two b's between d and c on the current path. We swap b at level $up[i]$ and c at level i and go down to level $down[up[i]]$, that is, the leftmost position of $tail(a)$ on the opposite path, which consists of five b's.

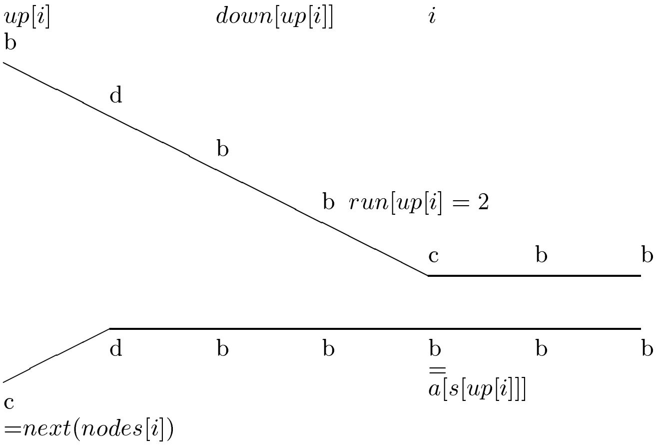

Fig. 2. Illustration of run

Algorithm 3 *Iterative algorithm for multiset permutations.*

```
$a := [1, ..., 1, 2, ..., 2, ..., k, ..., k]$;
for $i := 1$ to $n$ do begin $nodes[i] := (a[i]); s[i] := 0;$
                $mark[i] := false; up[i] := i$ end;
$i := n - m[k] + 1$;
$up[0] := 0$;
repeat
  if $a[i-1]$ is a first child and $nodes[i-1]$ has not been updated by its children
    then if $a[i] \neq a[i-1]$ then $nodes[i-1] \Leftarrow a[i]$;
    if $|nodes[i]| > 1$ then begin
      $swap(a[i], a[s[i]])$;
      $mark[i] := false$;
      $nodes[s[i]] := (a[s[i]])$;
      remove first of $nodes[i]$;
      if $a[i-1]$ is a first child then
```

```
14.         if $a[i] \neq a[i-1]$ then $nodes[i-1] \Leftarrow a[i]$;
15.      $s[i] := 0$
16.   end;
17.   $run[i] := 0$;
18.   if $a[i]$ is a last child then begin
19.      $up[i] := up[i-1]$; $up[i-1] := i-1$;
20.      {Compute run}
21.      if $a[up[i]] = a[i]$ and $up[i] < i$ then
22.         $run[up[i]] := run[up[i]] + 1$
23.      else begin $temp := run[up[i]]$; $run[up[i]] := 0$ end;
24.      if $a[i] = next(nodes[up[i]])$ then begin
25.         if $s[up[i]] = 0$ begin
26.            if $i = down[i] - 1$ and $a[up[i]] = a[n]$ then begin
27.               $down[up[i]] := i - temp$;
28.               $mark[up[i]] := true$;
29.            end else $down[up[i]] := i + 1$;
30.            $s[up[i]] := i$
31.         end
32.         else begin { $s[up[i]] \neq 0$ }
33.            $nodes[i] := (a[i])$;
34.            if $mark[up[i]] = false$ then $down[up[i]] := i$
35.         end
36.      else begin { $a[i] \neq next(nodes[up[i]]$ }
37.         $nodes[i] := (a[i])$;
38.         if $mark[up[i]] = false$ then $down[up[i]] := i$
39.      end;
40.      if $i < down[i]$ then begin {Going down}
41.         $mark[i] := false$;
42.         $i := down[i]$;
43.         $nodes[i] := (a[i])$;
44.      end else
45.      begin {Going up}
46.         $i1 := i$;
47.         $i := up[i]$;
48.         $up[i1] := i1$;
49.      end
50.   end else
51.   begin {$a[i]$ is not a last child, going down}
52.      $i := down[i]$;
53.      $nodes[i] := (a[i])$;
54.   end;
55. until $i = 0$;
```

Lines 1-4: Initialization.

Lines 6-7: While processing level i in the first subtree, prepare $nodes[i-1]$ of

the upper level.
Line 8: If current node $a[i]$ is not a last child,
Line 9: swap $a[i]$ and $a[s[i]]$.
Line 10: Immediately after swapping at level i, the value of $down[i]$ needs to be updated for later use, signalling $mark[i] = false$. The current value of $down[i]$ is good for going down.
Line 11: Set $nodes[s[i]] = (a[s[i]])$.
Lines 13-14: Update $nodes[i-1]$ by adding $a[i]$, if $a[i] \neq a[i-1]$.
Line 15: Set $s[i]$ to 0 to indicate that the solution point for level i has not been set.
Line 17: Set $run[i]$ to 0.
Line 18: If $a[i]$ is a last child, we do the following.
Line 19; The value of $up[i-1]$ propagates downwards to level i. After that reset $up[i-1]$ to $i-1$.
Lines 21-22: Extend the run length $run[up[i]]$ for level $up[i]$.
Line 23: Reset the run length for level $up[i]$ to 0.
Line 24: If $a[i] = next(nodes[up[i]]$, we do the following.
Line 25: If $s[up[i]]$ has not been computed,
Line 26: If the current position is at the left end of $tail(a)$ and $a[up[i]] = a[n]$,
Line 27: Set $down[up[i]] := i - run[up[i]]$, since $down[up[i]] = n - |tail(a)| + 1$ and $|tail(a)| = n - i + run(up[i]) + 1$ for a on the opposite path. The situation is illustrated in Example 2.
Line 28: Set $mark[up[i]] = true$ to indicate that $down[up[i]]$ has been finalized on the current path.
Line 29: If the condition at line 26 does not hold, the next landing point is at least $i+1$.
Line 30: Set $s[up[i]]$ to i.
Line 32: If $s[up[i]] \neq 0$,
Line 33: Prepare $nodes[i]$ on the opposite path by $(a[i])$, since label $a[i]$ will the same on the opposite path.
Line 34: If $down[up[i]]$ has not been finalized, we set $down[up[i]]$ to i, meaning that we come down from $up[i]$ to level at least i.
Lines 36: If $a[i] \neq next(nodes[i])$,
Line 37: Prepare $nodes[i]$ on the opposite path by $(a[i])$.
Line 38: If $down[up[i]]$ has not been finalized, we set $down[up[i]]$ to i, meaning that we come down from $up[i]$ to level at least i.
Lines 40-44: If we are not at the left end of $tail(a)$, that is, $i < down[i]$, we go down.
Line 41: Set $mark[i]$ to false for level i of the opposite path.
Line 43: Set $nodes[i]$ to list $(a[i])$.
Lines 45-49: If we are at the left end of $tail(a)$, we go up, and reset $up[i]$ to i using the old i.
Lines 51-54: If $a[i]$ is not a last child, we go down, and set $nodes[i]$ to list $(a[i])$ for level i of the opposite path, using the new i.

4 Detailed implementation

In this section, we implement informal descriptions given in Algorithm 3. For simplicity, we use a two-dimensional array $nodes[1..n, 1..k]$ for the lists, causing $O(kn)$ space requirement. The notation $nodes[i, c[i]]$ gives the $c[i]$-th value of t in Algorithm 1, that is, we maintain a pointer for the i-th list by array element $c[i]$. Since we skip all nodes on the opposite path when we go down, we need some care to maintain $nodes[i]$ properly. We use a Boolean array element $start[i-1] = true$ to show that $a[i-1]$ itself is a first child, and that $nodes[i-1]$ needs to be updated by $a[i-1]$'s first child. The array element $bound[i]$ is the current size of the list $nodes[i]$. The explanation of detailed code lines follows.

- Lines 6-7:

```
if $(start[i-1] = true)$ and $(up[i-1] = i-1)$ then begin
  $c[i-1] := 1$; $bound[i-1] := 1$; $start[i-1] := false$;
  if $a[i] \neq a[i-1]$ then begin
    $bound[i-1] := bound[i-1] + 1$; $nodes[i-1, bound[i-1]] := a[i]$
  end;
end;
```

We need the condition $up[i-1] = i-1$ in addition to $start[i-1] = true$ to judge whether $nodes[i-1]$ needs to be updated by $a[i-1]$'s first child, since we needed to prepare the environment for the opposite path by setting $start[i] := true$ at a last child when i was $i-1$, and thus we can not judge whether $a[i-1]$ is a first child or a last child just by $start[i-1]$. Fortunately in this case we have $up[i-1] \neq i-1$, since we perform $up[i] := up[i-1]$ for a last child.

- Line 8: *if* $c[i] < bound[i]$ *then begin*
- Line 11: $nodes[s[i], 1] := a[s[i]]$;
- Line 12: $c[i] := c[i] + 1$;
- Lines 13-14:

```
if $c[i-1] = 1$ then
  if $a[i] \neq a[i-1]$ begin
    $bound[i-1] := bound[i-1] + 1$; $nodes[i-1, bound[i-1]] := a[i]$
  end;
```

- We need the statement $start[i] := true$; between lines 9 and 10
- Line 18: *if* $c[i] = bound[i]$ *then begin*
- We need the statements $start[i] := true$; $bound[i] := 1$; between lines 19 and 20.
- Through out the algorithm we replace $nodes[i] := (a[i])$ by a pair of statements $c[i] := 1$; $nodes[i, c[i]] := a[i]$;
- Also we need the statement *if* $start[i] = true$ *then* $c[i] := 1$ after we go up to level i.

5 Concluding remarks

We developed an $O(1)$ time algorithm for generating multiset permutations. The main idea is tree traversal and identification of swapping positions. This technique is general enough to solve other combinatorial generation problems. In fact, this technique stemmed from that used in generation of parenthesis strings in [5]. The author succeeded in designing $O(1)$ time generation algorithms for other combinatorial objects, such as in-place combinations, reported in [12].

The key point is the computation of *up*, *down*, and s, the solution point, in which *up* is very much standard in almost all kinds of combinatorial objects. If we always go down to leaves, we need not worry about *down*. This happens with more regular structures, such as binary reflected Gray codes, ordinary permutations, and parenthesis strings, where we can concentrate on the computation of s. Multiset combinations and permutations have more irregular structures, that is, straight lines at some places, which require the computation of *down*, in addition to that of s. There are still many kinds of combinatorial objects, for which only O(1) change algorithms are known. The present technique will bring about O(1) time algorithms for those objects.

The space requirement for the algorithm is $O(kn)$. It is open whether this can be optimized to $O(n)$.

References

1. Hu, T.C. and B.N. Tien, Generating permutations with nondistinct items, Amer. Math. Monthly, 83 (1976) 193-196
2. Ko, C.W. and F. Ruskey, Generating permutations of a bag by interchanges, Info. Proc. Lett., 41 (1992) 263-269
3. Korsh, J. and S. Lipshutz, Generating multiset permutations in constant time, Jour. Algorithms, 25 (1997) 321-335
4. Lucas, J., The rotation graph of binary trees is Hamiltonian, Jour. Algorithms, 8 (1987) 503-535
5. Mikawa, K. and T. Takaoka, Generation of parenthesis strings by transpositions, Proc. the Computing: The Australasian Theory Symposium (CATS '97) (1997) 51-58
6. Nijenhuis, A. and H.S. Wilf, Combinatorial Mathematics, Academic Press (1975)
7. Proskurowski, A. and F. Ruskey, Generating binary trees by transpositions, Jour. Algorithms, 11 (1990) 68-84
8. Reingold, E.M., J. Nievergelt, and N. Deo, Combinatorial Algorithms, Prentice-Hall (1977)
9. Roelants van Baronaigien, D., A loopless algorithm for generating binary tree sequences, Info. Proc. Lett., 39 (1991) 189-194.
10. Ruskey, F. and D. Roelants van Baronaigien, Fast recursive algorithms for generating combinatorial objects, Congr. Numer., 41 (1984) 53-62
11. Savage, C, A survey of combinatorial Gray codes, SIAM Review, 39 (1997) 605-629
12. Takaoka, T., $O(1)$ Time Algorithms for combinatorial generation by tree traversal, Computer Journal (to appear)(1999)
13. Vajnovski, V., On the loopless generation of binary tree sequences, Info. Proc. Lett., 68 (1998) 113-117
14. Walsh, T.R., Generation of well-formed parenthesis strings in constant worst-case time, Jour. Algorithms, 29 (1998) 165-173
15. Zerling, D., Generating binary trees by rotations, JACM, 32 (1985) 694-701

Appendix. Pascal program for multiset permutations

```
program ex(input,output);
var first,i,j,k,kk,i1,n,temp,count:integer;
    a,m,up,down,s,start,run,mark,c,bound:array[0..20] of integer;
    nodes:array[1..10,1..10] of integer;
procedure out;
var k:integer;
begin
  for k:=1 to n do write(a[k]:2);
  writeln;
end;
procedure swap(i,j:integer);
var w:integer;
begin
  w:=a[i]; a[i]:=a[j]; a[j]:=w;
  count:=count+1;
  out;
end;
procedure perm;
begin
  repeat
    if (start[i-1]=1) and (up[i-1]=i-1) then begin
      c[i-1]:=1; bound[i-1]:=1;
      start[i-1]:=0;
      if a[i]<>a[i-1] then begin
        bound[i-1]:=bound[i-1]+1;
        nodes[i-1,bound[i-1]]:=a[i];
      end;
    end;
    if c[i]<bound[i] then begin
      start[i]:=0;
      swap(i,s[i]);
      mark[i]:=0;
      nodes[s[i],1]:=a[s[i]];
      c[i]:=c[i]+1;
      if (c[i-1]=1) and (a[i]<>a[i-1]) then begin
        bound[i-1]:=bound[i-1]+1; nodes[i-1,bound[i-1]]:=a[i];
      end;
      s[i]:=0;
    end;
    run[i]:=0;
    if c[i]=bound[i] then begin
      up[i]:=up[i-1]; up[i-1]:=i-1;
      start[i]:=1; bound[i]:=1;
      {*** run ***}
```

```
      if (a[up[i]]=a[i]) and (up[i]<i)
        then run[up[i]]:=run[up[i]]+1
        else begin temp:=run[up[i]]; run[up[i]]:=0 end;
      if a[i]=nodes[up[i],c[up[i]]+1] then
        begin
          if s[up[i]]=0 then begin
            {*** down ***}
            if (i=down[i]-1) and (a[up[i]]=a[n]) or (i=n)
            then begin
              down[up[i]]:=i-temp;
              mark[up[i]]:=1
            end
            else down[up[i]]:=i+1;
            s[up[i]]:=i;
          end else
          begin {s[up[i]]<>0}
            nodes[i,1]:=a[i];
            if (mark[up[i]]=0) then down[up[i]]:=i;
          end
        end
      else begin {a[i]<>nodes[up[i],c[up[i]]+1]}
        nodes[i,1]:=a[i];
        if mark[up[i]]=0 then down[up[i]]:=i
      end;
      if i<down[i] then begin
        mark[i]:=0;
        i:=down[i];
        c[i]:=1;
        nodes[i,1]:=a[i];
      end else
      begin
        i1:=i;
        i:=up[i];
        up[i1]:=i1;
        if start[i]=1 then c[i]:=1
      end;
    end else
    begin {c[i]<>bound[i]}
      i:=down[i];
      c[i]:=1; nodes[i,1]:=a[i];
    end;
  until i=0
end;
begin
  write('input k ');
```

```
  readln(kk);
  writeln('input multiplicities m[1], ..., m[k]');
  for i:=1 to kk do read(m[i]);
  readln;
  first:=0;
  for i:=1 to kk do begin
    for j:=1 to m[i] do a[first+j]:=i;
    first:=first+m[i]
  end;
  n:=first;
  count:=1;
  i:=n-m[kk];
  up[0]:=0;
  for k:=1 to n do begin nodes[k,1]:=a[k]; up[k]:=k; s[k]:=0 end;
  for k:=1 to n do mark[k]:=0;
  for k:=1 to n do begin c[k]:=1; bound[k]:=1; start[k]:=1; end;
  i:=i+1;
  out;
  perm;
  write('count ',count:3);
  readln
end.
```