Functional Programs for Generating Permutations

R. W. Topor
Department of Computer Science, Monash University, Clayton, Victoria 3168, Australia

This paper presents several functional programs for generating permutations represented as linear linked lists. Exact formulas for the storage used by some of these programs are derived and compared with the theoretical lower bound. Two different programs are shown to be optimal with respect to this measure. Problems considered include generation of all $N!$ permutations of N distinct elements, generation of all permutations of length $k \leq N$, and generation of all distinct permutations of a multiset of N elements.

1. INTRODUCTION

The problem of devising algorithms to generate all $N!$ permutations of N elements has attracted much attention in recent years. This attention is due both to the applications to which such algorithms may be put and to the intrinsic interest and complexity of algorithms for such a basic and well-defined problem in combinatorial mathematics. Recent, comprehensive surveys of such algorithms have been provided by Sedgewick[1] and Roy[2] in which they show the common structure of many algorithms, and study the complexity and efficient implementation of some of them.

The methods surveyed by Sedgewick and Roy all assume that individual permutations are represented as arrays of N elements. Often, however, one is interested in representing each permutation as a linear linked list of elements. This change in representation has a considerable effect on the types of algorithms which may be used to generate successive permutations. In particular, algorithms which exchange arbitrary pairs of elements in a permutation are now much less attractive than algorithms which somehow work through the permutation one element at a time.

Many permutation methods are most naturally expressed using recursive procedures. Sedgewick describes how several different methods may be viewed as instances of the same recursive scheme, and shows how these recursive procedures may be transformed into more 'efficient' iterative programs. Recently, however, Backus[3] and others have suggested that a functional style of programming, based entirely on calling and composing functions, has many advantages over the more common imperative style.

The aim of this paper is to study several algorithms for generating lists of permutations under the restrictions that permutations are represented as linear linked lists and that algorithms are presented as sets of mutually recursive function definitions. Thus, these algorithms can be implemented most easily in languages such as Lisp[4] and Backus's functional programming systems. Our wish to use a purely functional programming style implies that all list processing operations must be constructive and that fields of existing list nodes cannot be altered. Since such list processing systems allocate storage (i.e. list nodes) dynamically from a heap or free list, and since $N!$ permutations must be generated, the algorithms studied are going to use many list nodes. Accordingly, we shall define the complexity of these algorithms to be the total number of list nodes they use. This measure has the advantages of being independent of any particular implementation and of being a realistic lower bound on the actual running time. For many of the algorithms studied, we shall derive exact formulas for the number of nodes used as a function of N, the number of elements being permuted. This enables quantitative comparisons to be made between the storage requirements of the different algorithms and the theoretical lower bound.

We also show how the algorithms presented may be modified to solve generalized versions of the basic problem: generating all permutations of length $k \leq N$ from a list of N distinct elements, and generating all distinct permutations of a list which may contain repeated elements. All these algorithms return a list of the desired permutations. We also consider permutation generators: functions which return the next permutation in the list each time they are called. The order in which these various algorithms return or generate permutations is also considered. Programs which use iteration and destructive list processing operations could also be analysed in the same way.

In describing the algorithms we use the following conventions and notation. Formally, an α-list is either null (denoted nil) or has a hd (which is an α) and a tl (which is an α-list). The only lists we shall use are element-lists and permutation-lists. Elements are atomic, and are sometimes assumed to be drawn from a totally ordered set; permutations are represented by element-lists. If x is a list, we test whether it is null by writing null x, and we select its components by writing hd x or tl x. We create a new list, adding the element (or permutation) a at the front of an existing list x, by writing cons(a, x) or $a{:}x$. Physically, lists are represented by pointers to nodes, and each application of cons allocates a new list node from the heap, whose hd is a and whose tl is a pointer to x. Different list nodes may each have tl's which point to the same list. We denote the unit list containing the element a by list(a), the length of a list x by $|x|$, the result of deleting (the first occurrence of) an element a from a list x by $x - a$, and the result of concatenating two lists x and y by $x|y$. As illustrations of the programming style to be used we give the definitions of these functions below.

list (x) = x:nil
$|x|$ = **if** null x **then** 0 **else** 1 + $|$tl $x|$
$x - a$ = **if** hd $x = a$ **then** tl x **else** (hd x):((tl x) $- a$)
$x|y$ = **if** null x **then** y **else** (hd x):((tl x)$|y$)

CCC-0010-4620/82/0025-0257$03.50

This definition of $x - a$ is valid since, whenever $x - a$ is applied in the programs below, a is known to be an element of x.

2. GENERAL ALGORITHMS

In this section we introduce the main algorithms to be studied in the remainder of the paper. Like Roy,[2] we find that all these algorithms can be seen as instances of just two general algorithms which, following his terminology, we call algorithms A and B. A different type of algorithm is presented in section 9.

Assume we are given a list x containing N distinct elements, and are asked to generate all $N!$ permutations of the elements in x. If x is null ($N = 0$), the only permutation is list(x), and we omit this case in the descriptions of the algorithms.

Algorithm A

for each permutation p of (tl x) **do**
  insert (hd x) at each possible position in p

This algorithm is described in Ref. 5 and Ref. 6. It is recursive in that the permutations of (tl x) are themselves generated by the same method. The following algorithm has the same form as algorithm A:

Algorithm Q

for each permutation p of (tl x) **do**
  append (hd x) to p, and
  exchange (hd x) with each element a in p

This algorithm uses exactly the same number of list nodes as algorithm A, but generates permutations in a slightly different order. As it is so similar, we shall not consider it any further.

Algorithm B

for each element a in x **do**
  append a to each permutation of $x - a$.

A version of this algorithm is described in Ref. 7. Since $x - a$ is evaluated constructively, the list x is unchanged after each element a has been considered, and algorithm B is thus an example of a backtrack algorithm. An advantage of backtrack algorithms is that they are easy to modify to avoid considering whole sequences of permutations which do not satisfy some criterion.

Algorithm B appears to differ significantly from algorithm A in that it contains several recursive calls at each level. Hence, it is difficult to say immediately how their complexities compare. We shall see that the complexities of both algorithms, and especially of algorithm B, depend critically on how they are implemented.

Two other backtrack algorithms have the same form as algorithm B:

Algorithm R

for each rotation $a:y$ of x **do**
  append a to each permutation of y

The 'rotations' of a list are the results of repeatedly moving its first element to the end. For example, the rotations of ABCD are ABCD, BCDA, CDAB, and DABC. Algorithm R is strictly less efficient than algorithm B, however they are implemented, and hence is also not considered any further. To see that this is true, note that deleting the kth element of x from x requires $(k - 1)$ list nodes, so deleting each element in turn requires $N(N - 1)/2$ list nodes. On the other hand, each rotation of x except for the identity rotation requires N list nodes, so generating each rotation of x requires $N(N - 1)$ list nodes.

Algorithm S

for each element a in x **do**
  exchange (hd x) with a, and
  append a to each permutation of the tl of the result.

Algorithm S only differs from algorithm B in the position at which (hd x) is replaced after removing a. The number of list nodes required to remove a from x is the same in each algorithm. Only the order in which they generate permutations differs. Since, as we shall see, algorithm B can be made to generate permutations in a very convenient order, we may also ignore algorithm S below. Note, however, the similarity between algorithms Q and S, which perform the same operations in a different order. This suggests it may be possible to derive both algorithms from the abstract specification of the problem by following different execution paths as was done by Clark and Darlington[8] for mergesort and quicksort.

3. A LOWER BOUND

Before implementing algorithms A and B and comparing their complexities, we derive a lower bound on the complexity of any algorithm which computes a list of all permutations of N distinct elements.

Intuitively, it seems that there are $N!$ permutations of length N requiring $N.N!$ nodes, together with an additional $N!$ nodes at the top level of the list containing these permutations, making $(N + 1)!$ nodes in all. However, this ignores the possibility that different permutations may physically use the same nodes. For example, the permutations (A B C) and (B A C) may both end in the same node (C). By sharing sublists in this way, the set of $N!$ permutations may be represented as a tree whose leaves are the individual permutations and whose root is nil.

Let C_N, $N \geq 0$, be the minimum number of list nodes required to represent all $N!$ permutations of N elements. Clearly $C_0 = 0$. For each element a, one node is required to represent z = list(a), and an additional C_{N-1} nodes are required to represent all $(N - 1)!$ permutations of the remaining $N - 1$ elements. Since each of these $(N - 1)!$ permutations can be made to end with a node whose tl is

z, they also represent those permutations of N elements ending in a. Thus we have

$$C_0 = 0$$
$$C_N = N(1 + C_{N-1}), \quad N \geq 1 \qquad (1)$$

If we define

$$e_N = \sum_{k=0}^{N-1} 1/k!, \quad N \geq 0$$

where $e_N \to e$, the base of natural logarithms as $N \to \infty$, then the solution of Eqn (1) is

$$C_N = e_N N!, \quad N \geq 0 \qquad (2)$$

Now, let L_N, $N \geq 0$, be the minimum number of list nodes required to represent a list of all $N!$ permutations of N elements. Since such a list requires $N!$ nodes for its top level, and C_N nodes to represent the permutations, we have

$$L_N = (e_N + 1)N!, \quad N \geq 0 \qquad (3)$$

In later sections we shall present programs which achieve this lower bound.

4. IMPLEMENTATION AND ANALYSIS OF ALGORITHM A

A reasonably efficient implementation of algorithm A can be expressed as follows:

```
permute1(x) =
  if null x then list(x)
  else mapinsert(hd x, permute1(tl x))
mapinsert(a, ps) =
  if null ps then nil
  else insert(a, hd ps, hd ps, mapinsert(a, tl ps))
insert(a, p, q, ps) =
  if null q then put(a, p, q):ps
  else put(a, p, q): insert(a, p, tl q, ps)
put(a, p, q) =
  if p = q then a:q
  else (hd p): put(a, tl p, q)                    (4)
```

In this program, permute1(x) returns a list of all permutations of the elements in the list x. Mapinsert(a, ps) appends the results of inserting a at each position of each permutation in ps. Insert(a, p, q, ps) puts a into p immediately before q (which is a sublist of p) and each subsequent position of p. The fourth argument, ps, of insert is an accumulator which avoids an explicit concatenation operation in mapinsert. Put(a, p, q) constructively puts a into p immediately before q. The list (pointer) comparison $p = q$ could be replaced by integer variables indicating positions in a list, or by comparing the elements of a list, without any change in efficiency.

How many list nodes does permute1 use? Clearly, if $|p| - |q| = m$, then put(a, p, q) uses $m + 1$ nodes. Suppose $|p| = n$ when insert is called initially with $p = q$. Then insert(a, p, q, ps) uses $\sum_{m=0}^{n}(m + 2) = (n + 4)(n + 1)/2$ nodes. Since a list containing the result of inserting a at each position of p when $|p| = n$ requires $(n + 4)(n + 1)/2$ nodes to be represented, this definition of insert is optimal. If ps contain m lists of length n, then mapinsert(a, ps) applies insert m times and thus uses $m(n + 4)(n + 1)/2$ nodes. Finally, let permute1(x) use A_N nodes if $|x| = N$. Then we have

$$A_0 = 1$$
$$A_N = A_{N-1} + (N + 3)N(N - 1)!/2, \quad N \geq 1 \qquad (5)$$

whose solution is

$$A_N = 1 + \tfrac{1}{2}\sum_{k=1}^{N}(k + 3)k!, \quad N \geq 0 \qquad (6)$$

Since $A_N > [(N + 3)/2]N!$, the number of temporary nodes used for each permutation grows slowly, but linearly, with N. It is not surprising that the value of A_N is so much greater than the lower bound L_N as algorithm A offers little opportunity for the sharing necessary to achieve the lower bound.

5. IMPLEMENTATION AND ANALYSIS OF ALGORITHM B

Our first implementation of algorithm B appends a at the start of each permutation of $x - a$:

```
permute2(x) =
  if null x then list(x)
  else mapperm(x, x)
mapperm(x, y) =
  if null y then nil
  else mapcons(hd y, permute2(x − (hd y)),
                                  mapperm (x, tl y))
mapcons(a, ps, qs) =
  if null ps then qs
  else (a: (hd ps)): mapcons(a, tl ps, qs)        (7)
```

In this program, iteration through the elements a of x is done by the function mapperm which appends each element of y in turn to the permutations of the remaining elements of x. Mapcons(a, ps, qs) appends a to each element of ps in turn, using the accumulator qs as in insert to avoid an explicit concatenation operation.

It is easy to show by an inductive argument that the list of permutations returned by permute2 is in lexicographic order. For example, if x = ABC, then permute2(x) = (ABC ACB BAC BCA CAB CBA).

Clearly, if $|ps| = m$, then mapcons(a, ps, qs) uses $2m$ nodes. Suppose $|x| = n$ when mapperm is called initially with $x = y$. Then the n deletion operations use $n(n - 1)/2$ nodes in all. Finally, let permute2(x) use X_N nodes if $|x| = N$. Then we have

$$X_0 = 1$$
$$X_N = NX_{N-1} + N(N - 1)/2 + 2N!, \quad N \geq 1 \qquad (8)$$

whose solution is

$$X_N = (2N + 1 + e_N/2)N! - N/2, \quad N \geq 0 \qquad (9)$$

This result is quite disappointing. Not only is $X_N > A_N$, but also X_N is greater than our first estimate of L_N which ignored the possibility of sharing sublists. Again, this should not be surprising as the use of mapcons ensures there is no sharing in the result returned by permute2.

This inefficiency can be overcome if we implement algorithm B by appending a at the end of each permutation of $x - a$. To do this we rename permute2 as genperm and give it a new argument, p, which is a partial

permutation to which a and the rest of x is to be consed. That is, genperm(x, p, ps) appends all permutations of x in front of p, and conses each resulting permutation to ps. Mapperm remains basically unchanged, except for the introduction of an accumulator, ps. These changes lead to the following implementation of algorithm B:

```
permute3(x) =
  if null x then list(x)
  else genperm(x, nil, nil)
genperm(x, p, ps) =
  if null(tl x) then ((hd x):p):ps
  else mapperm(x, x, p, ps)
mapperm(x, y, p, ps) =
  if null y then ps
  else mapperm(x, tl y, p,
                genperm(x − (hd y), (hd y):p, ps))   (10)
```

The list of permutations returned by permute3 is in reverse lexicographic order. That is, when the list is read backwards, and the permutations are read from right to left, the resulting list is in lexicographic order. For example, if x = ABC, then permute3(x) = (ABC BAC ACB CAB BCA CBA).

Suppose genperm(x, p, ps) uses B_N nodes if $|x| = N$. Then

$$B_1 = 2$$
$$B_N = NB_{N-1} + N(N+1)/2, \quad N \geq 2 \qquad (11)$$

whose solution is

$$B_N = (3e_N/2 + 1)N! - N/2, \quad N \geq 1 \qquad (12)$$

This result is more satisfactory. It indicates that the number of temporary nodes used for each permutation generated is constant, and that permute3 is less than 50% worse than the best possible algorithm. For small values of N, A_N and B_N are very similar.

It is possible, however, to do even better. Permute3 returns a list of permutations which share as many sublists as possible, but does not use the nodes of the lists x − (hd y) in the final result. If we could ensure that whenever we constructed a new list it was used in the result, then the resulting program might be optimal.

Suppose the input list x is $(a_1 a_2 \ldots a_N)$. We can represent the arguments x and p of genperm and mapperm by introducing an integer variable j, $1 \leq j \leq N$, such that $x = (a_1 a_2 \ldots a_j)$ and $p = (a_{j+1} \ldots a_N)$. Similarly, we can represent the argument y of mapperm by another integer variable i, $1 \leq i \leq j$, such that $y = (a_i \ldots a_j)$. If we then define a new function, move(x, i, j), which simultaneously deletes the ith element of x (i.e. hd y) and inserts it after the jth element of x (i.e. at the start of p), we get the following, final implementation of algorithm B:

```
permute4(x) =
  if null x then list(x)
  else genperm(x, |x|, nil)
genperm(x, j, ps) =
  if j = 1 then x:ps
  else mapperm(x, 1, j, ps)
mapperm(x, i, j, ps) =
  if i = j then genperm(x, j − 1, ps)
  else mapperm(x, i + 1, j,
                genperm (move (x, i, j), j − 1, ps))
move (x, i, j) =
  if i = 1 then put (hd x, tl x, j − 1)
  else (hd x): move(tl x, i − 1, j − 1)
put (a, x, j) =
  if j = 0 then a:x
  else (hd x): put (a, tl x, j − 1)                  (13)
```

Because only the representation has been changed, permute4 also returns a list of permutations in reverse lexicographic order.

Clearly, move(x, i, j) uses j nodes. Suppose genperm(x, j, ps) uses Y_j nodes. Then

$$Y_1 = 1$$
$$Y_j = jY_{j-1} + (j-1)j, \quad j \geq 2 \qquad (14)$$

whose solution is

$$Y_N = (e_N + 1)N! - N, \quad N \geq 1 \qquad (15)$$

This appears to be less than the lower bound L_N given in Eqn (3), but the difference is simply that permute4 uses its input list x as one of the resulting permutations, thereby saving N nodes. Thus, we have found a simple, optimal implementation of algorithm B. It appears to be difficult to find such a program without using integer variables because of the need to simultaneously delete a from x and append it to p. Another, quite different, optimal program is presented in section 9.

6. PERMUTATIONS OF LENGTH $k \leq N$

We now consider the slightly more general problem of generating all ${}^N\mathrm{P}_k$ permutations of k elements drawn from a list x of N distinct elements. We solve this problem by modifying both algorithm A and algorithm B, and then showing how the modified algorithms can be implemented. In each case we can assume that if $k = 0$, list(nil) is the result, and that if $k > N$, there are no permutations and nil is the result. Algorithm A may be modified to solve this problem by independently generating all permutations of x which include (hd x) and all those which exclude (hd x). The efficient implementation of this idea, requires us to give mapinsert a third parameter, as we did to insert in Eqn (4). Algorithm A (or permute1) may then be expressed as follows, omitting the definitions of insert and put which remain unchanged:

```
permute1(x, k) =
  if k = 0 then list (nil)
  else if |x| < k then nil
  else mapinsert(hd x, permute1(tl x, k − 1),
                 permute1(tl x, k))
mapinsert(a, ps, qs) =
  if null ps then qs
  else insert(a, hd ps, hd ps, mapinsert(a, tl ps, qs))   (16)
```

The modified version of algorithm B may be expressed as follows:

for each element a of x **do**
  append a to each permutation of $x - a$ of length $k - 1$.

We give below the revised definition of permute2, omitting the definition of mapcons which is unchanged. Permute3 can be revised in an analogous way, but it is

awkward to modify permute4 to solve this problem as it has no explicit pointer to the partial permutation p.

```
permute2(x, k) =
  if k = 0 then list(nil)
  else if |x| < k then nil
  else mapperm(x, x, k)
mapperm(x, y, k) =
  if null y then nil
  else mapcons(hd y, permute2(x − (hd y), k − 1),
                mapperm(x, tl y, k))                    (17)
```

7. PERMUTATIONS WITH REPEATED ELEMENTS

Another related problem is to generate the list of all distinct permutations of x when the list x may contain repeated elements, i.e. when x is a multiset. In this case the number of distinct permutations of x is less than $N!$. For example, if x = AABB, the list of distinct permutations of x is (AABB ABAB ABBA BAAB BABA BBAA). This list can be generated by first generating the list of all possible permutations and then removing all repeated permutations, but this would be grossly inefficient. It is better to modify the basic algorithms to avoid ever generating the same permutation twice.

Algorithm A can be modified to do this as follows:

```
for each distinct permutation p of (tl x) do
  insert (hd x) at each possible position in p
  up to the first occurrence (if any) of (hd x) in p
```

For example, if (hd x) = A, and p = BCAD, the result of inserting (hd x) into p would be the list (ABCAD BACAD BCAAD). The correctness of this algorithm relies on the fact that (hd x) is eventually inserted into all permutations of (tl x). To implement this idea, the only change needed to permute1 is in the definition of the function insert:

```
insert(a, p, q, ps) =
  if null q then put(a, p, q):ps
  else if a = (hd q) then put(a, p, q):ps
  else put(a, p, q):insert(a, p, tl q, ps)              (18)
```

Algorithm B can be modified to solve this problem in the following way:

```
for each distinct element a in x do
  append a to each distinct permutation of x − a
```

To implement this idea we must give mapperm an additional argument which is the set s of elements already appended to each permutation of the remaining elements. Permute2 may then be redefined as follows:

```
permute2(x) =
  if null x then list(x)
  else mapperm(x, x, nil)
mapperm(x, y, s) =
  if null y then nil
  else if (hd y)∈s then mapperm(x, tl y, s)
  else mapcons(hd y, permute2(x − (hd y)),
                mapperm(x, tl y, (hd y):s))             (19)
```

The definitions of permute3 and permute4 may be modified in an analogous way. In these definitions, the expression $a \in s$ is true iff a is an element of the list s, a standard predicate which is easy to define recursively.

Analysing the complexity of these programs to handle lists with repeated elements is a difficult mathematical problem. To solve the problem with both generalizations at once—generate all distinct permutations of k elements drawn from a multiset of N elements—permute2 must simply be modified by making the changes in Eqns (17) and (19) in the one program. The definitions of permute3 and permute4 can be modified similarly. Modifying permute1, however, by simply making the changes in Eqns (16) and (18) does not work. Instead, permute1 must be modified as follows to handle both generalizations at once:

```
permute1(x, k) = genperm(x, k, nil)
genperm(x, k, s) =
  if k = 0 then list(nil)
  else if |x| < k then nil
  else if (hd x)∈s then genperm(tl x, k, s)
  else mapinsert(hd x, genperm(tl x, k − 1, s),
                  genperm(tl x, k, (hd x):s))           (20)
```

Mapinsert must be modified as in Eqn (16), and insert must be modified as in Eqn (18). After making either or both sets of changes to any of these programs, the resulting program still returns a list of permutations in the same order that the original program did.

8. PERMUTATION GENERATORS

The algorithms we have considered so far each return a list of all the (distinct) permutations of the list x. The purpose of generating this list is presumably to allow us to process each permutation in turn. There are, however, two other ways of achieving this goal.

The first way, the more efficient according to Sedgewick, is to treat the permutation generation procedure as the main program, and make it process each permutation as it is generated. Of the programs we have studied, only permute3 and permute4 can be transformed into this sort of a procedure. In the cases of permute1 and permute2, the partially constructed permutations are stored implicitly on the procedural stack, and cannot be accessed at any intermediate stage of the computation. Permute3 and permute4 can, however, be transformed in this way, as their partially constructed permutations are represented explicitly. The resulting program is no longer purely functional as it has the side-effect of processing each permutation in turn. Accordingly, we introduce the imperative features of 'processing' and sequencing into our language. The version of permute3 in Eqn (10) for example, may thus be rewritten as follows:

```
permute3(x) = genperm(x, nil)
genperm(x, p) =
  if null x then process p
  else mapperm(x, x, p)
mapperm(x, y, p) =
  if not (null y) then
        begin
        genperm (x − (hd y), (hd y):p);
        mapperm (x, tl y, p)
        end                                             (21)
```

Clearly, mapperm could also be rewritten using a loop and an assignment statement.

The second way of using each permutation in turn is to write a function which returns a new permutation each time it is called. Functions which return the next item of some sequence each time they are called are sometimes called generators. In our domain, we might call them permutation generators. Each of permute1, permute3 and permute4 can be transformed into a permutation generator which generates the same sequence of permutations using the same sequence of list processing operations. These generators may be derived by transforming the original recursive functions into recursive coroutines,[9] studying how the stack changes between the generation of one permutation and the next, and rewriting the recursive coroutines iteratively. The first of these transformations is straightforward, but the second appears to be quite difficult.

For each program, the resulting generator uses an explicit array containing values of what were local variables of active function calls to represent the state of the computation. In the cases of permute3 and permute4 the array is accessed in a stack discipline, but in the case of permute1, which has a quite different recursion structure, a correspondingly different pattern of array accesses is used.

As we are primarily interested in functional programs here, we omit the derivations and definitions of these generators. The resulting generators are, in any case, much less simple than the other programs we have considered, and less efficient than the generator to be presented in the next section.

9. A FUNCTIONAL PERMUTATION GENERATOR

In this section we return to the domain of functional programming, and give a functional program which, given any permutation p, returns a new permutation which is the successor of p in reverse lexicographic order. For example, the successor of HFEDGCAB in reverse lexicographic order is DEGHFCAB. (The lexicographic successor is harder to generate as it would involve changing the positions of elements at the end of each permutation.) Thus, for this generator, the only state information required is the current permutation itself. The algorithm we shall use is due to Fischer and Krause as described in Ref. 1, p. 153. The basic idea is to find the first sublist r of p whose hd, a, is greater than its predecessor. If there is no such sublist, then the elements are in reverse order and we return NONE since there is no successor. Otherwise, we find the first sublist q of p whose hd in less than a, exchange the hd's of q and r, and reverse the elements of p up to but excluding q. In the above example, if p = HFEDGCAB then r = GCAB and q = FEDGCAB. The following implementation of this algorithm combines the operations of exchanging and reversing in the function next3.

```
nextperm(p) =
  if null p then NONE
  else next2(p, firstup(p))
firstup(p) =
  if null (tl p) then nil
  else if (hd p) < hd(tl p) then tl p
  else firstup(tl p)
next2(p, r) =
  if null r then NONE
  else next3(p, firstless(p, hd r), r)
firstless(p, a) =
  if (hd p) < a then p
  else firstless(tl p, a)
next3(p, q, r) = genrev(p, q, r, (hd q):(tl r))
genrev(p, q, r, s) =
  if p = r then s
  else if p = q then genrev(tl p, q, r, (hd r):s)
  else genrev(tl p, q, r, (hd p):s)                    (22)
```

This program is of interest for several reasons: it generates permutations in a regular order, it is purely functional, and it also turns out to be optimal with respect to its use of storage.

We may analyse the complexity of this program by considering the length of the first (descending) run in a random permutation. A (descending) 'run' in a permutation is simply a maximal descending subsequence of the permutation. For example, the permutation HFEDGCAB has three descending runs, of lengths 4, 3, and 1. Runs are important in the analysis of program (22) as r is the position following the first descending run in p.

Consider a random permutation of length N. Knuth[10] proves that the probability p_m that its first run has length m is given by

$$\begin{aligned} p_m &= 1/m! - 1/(m+1)!, \quad 1 \le m < N \\ p_N &= 1/N! \end{aligned} \tag{23}$$

In our case, p_m is the probability that $|p| - |r| = m$. If r is null, nextperm uses no list nodes; otherwise, if $|p| - |r| = m$, nextperm uses $m + 1$ nodes. Thus, the average number of nodes used by nextperm is given by

$$\sum_{m=1}^{N-1} (m+1)p_m = e_N - 1/(N-1)! \tag{24}$$

and the total number of nodes used in $N!$ successive calls to nextperm, starting with $p = 123 \ldots N$ for example, is

$$e_N N! - N = C_N - N \tag{25}$$

We can use nextperm to define another program which returns a list of all $N!$ permutations of its input list x as follows:

```
permute5(x) =
  if x = NONE then nil
  else x:permute5(nextperm(x))                         (26)
```

The total number of nodes used by this program, including the N nodes of the initial permutation x, is thus

$$e_N N! + N! = L_N$$

Permute5 is thus, like permute4, optimal in its use of storage. Each achieves the lower bound because (i) every list node allocated is used in the result, and (ii) all changes are made at the start of the list, thus maximizing the amount of sharing which occurs. The analysis of permute5 may be a less reliable guide to its running time than the previous analyses were, as it ignores the contributions of the functions firstup and firstless. Permute5 has the nice property that, without any change at all, it also generates all distinct permutations of a list x with repeated elements. Naturally, it cannot be used to generate permutations of length $k \le N$.

10. SUMMARY

We have presented several algorithms for generating lists of permutations represented as linear linked lists, and shown how these algorithms may be implemented in a purely functional way. The implementations of these algorithms have been analysed with respect to the amount of storage used, and shown to compare favourably with the best achievable values. Two of the resulting programs have been shown to be optimal with respect to this measure. We have also shown how the algorithms may be modified to solve several related problems. The various algorithms are characterized by their extreme simplicity and elegance, and could easily be implemented in any real list processing system.

In practice, the following algorithms can be recommended. To generate all permutations of a list of distinct elements, permute3 as defined in Eqn (10) is a short, reasonably efficient solution, and permute4 as defined in Eqn (13) is an optimal solution which, however, requires arithmetic operations. Both these algorithms return a list of permutations in reverse lexicographic order. To generate all permutations of length k from a list which may contain repeated elements, permute1 as modified in Eqns (16), (18) and (20) is a short, reasonably efficient solution. Finally, nextperm, as defined in Eqn (22), generates successive permutations in reverse lexicographic order, assuming only that the elements being permuted are drawn from a totally ordered set.

The following are some obvious questions which remain to be answered. Are there any other, different algorithms for these problems? What are the (space) complexities of the programs to generate permutations of lists with repeated elements? How can one reasonably define and measure the time complexity of these programs? How do they compare with the best iterative algorithms? Are there any general transformations to convert functional programs returning lists of permutations into (iterative) permutation generators? The solution of these problems is the object of continuing research.

Acknowledgment

I am grateful to Dr M. P. Georgeff for showing me how to implement algorithm B optimally.

REFERENCES

1. R. Sedgewick, Permutation Generation Methods, *ACM Computing Surveys* **9** (No. 2), 137–164 (1977).
2. M. K. Roy, Evaluation of Permutation Algorithms, *Comput. J.* **21** (No. 4), 296–301 (1978).
3. J. Backus, Can Programming be Liberated from the von Neuman Style? A Functional Style and its Algebra of Programs, *Communications of the ACM* **21** (No. 8), 613–641 (1978).
4. J. R. Allen, *Anatomy of Lisp*, McGraw-Hill, New York (1978).
5. W. H. Burge, *Recursive Programming Techniques*, p. 148. Addison-Wesley, Reading, Massachusetts (1975).
6. D. E. Knuth, *The Art of Computer Programming*, Vol. 1, 2nd Edn, p. 44. Addison-Wesley, Reading, Massachusetts (1973).
7. A. Nijenhuis and H. S. Wilf, *Combinatorial Algorithms*, 2nd Edn, p. 56. Academic Press, New York (1978).
8. K. L. Clark and J. Darlington, Algorithm Classification through Synthesis, *Comput. J.* **23** (No. 1), 61–65 (1980).
9. O.-J. Dahl and C. A. R. Hoare, Hierarchical Program Structure, in *Structured Programming*, ed. by Dahl *et al.* Academic Press, New York (1972).
10. D. E. Knuth, *The Art of Computer Programming*, Vol. 3. Addison-Wesley, Reading, Massachusetts (1973).

Received August 1981