Reversing is Not Inherent in Lexicographical Permutation Generation

J. S. Rohl*

TR 89-1019
June 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*On sabbatical leave from The University of Western Australia.

Reversing is not inherent in lexicographical permutation generation

J.S.Rohl[1]
Department of Computer Science
Cornell University
Ithaca, New York, 14853-7501

Abstract
In his comprehensive 1977 survey of permutation generation methods, Sedgewick[4] stated that "(reversing) seems to be inherent in lexicographical (permutation) generation". It is the purpose of this paper to give an algorithm which does not use reversing and to show its relationship to the classical reversing algorithm of Ord-Smith[3]. We also give a number of related algorithms to illustrate the flexibility of the new algorithm.

[1] On sabbatical leave from The University of Western Australia

1. Introduction

In his comprehensive 1977 survey of permutation generation methods, Sedgewick[4] stated that "(reversing) seems to be inherent in lexicographical (permutation) generation". It is the purpose of this paper to give an algorithm which does not use reversing and to show its relationship to the classical reversing algorithm of Ord-Smith[3]. In recent times a number of authors have given a number of "non-reversing" algorithms but these have also used auxiliary data structures: two arrays in the case of Hitotumatu and Noshita[1], which was reduced to a single array by Rohl[6]; and a linked-list by Irving[2]. However, Sedgewick was considering the classical problem of *in situ* permutations so that these algorithms do not qualify.

2. The algorithm

The following algorithm, Fig 1, does qualify.

```
procedure Lex (p: perm; n: range);
{Pre : p[1..n] contains the marks in lexicographical order.}
{Post : All permutations have been processed in lexicographical order.}

  procedure Choose (k: range);
  {Pre : p[k..n] is in lexicographical order.}
  {Post : All permutations with prefix p[1..k] have been processed lexicographically
        and p[k..n] is (back) in lexicographical order.}
    var i: range;
  begin
    for i := k to n do
      begin
        Swap(p[i], p[k]);
        if k = n then ProcessPerm(p, n) else Choose(k+1);
      end;
    Rotate(p, k, n)
  end;{of procedure "Choose"}

begin
  Choose(1)
end;{of procedure "Lex"}
```

Fig 1. The non-reversing lexicographical algorithm

It uses the definitions:

```
type
   range = 1..maxrange;
   mark = {some unspecified enumerated or subrange type};
   perm = array[range] of mark
```

and calls *ProcessPerm* with all $n!$ permutations of the marks in p in lexicographical order.

The algorithm is based on the choosing paradigm of Rohl[5]. A call *Choose* can be thought of as "make all choices for the k^{th} element, and for each choice in turn call *Choose(k+1)* to make all the choices for the $k+1^{th}$ element etc.".

It should be noted that there are many variants of this procedure: the one given here is the shortest, and very nearly the slowest. Notice, first, that on the initial iteration of the loop at each level, the swapping operation is essentially a null operation, and can be avoided by taking it out of the loop. Notice, secondly, that unless the procedure is being adapted for back-tracking purposes, the recursion can terminate when $n-1$ marks have been chosen, because there is only 1 choice for the n^{th} mark and it is already in *p[n]*. The same observations apply to Ord-Smith's algorithm as we shall derive it, so we will be comparing like with like. It is one of the problems with comparing permutation algorithms that they can be fine-tuned almost without limit!

3. Proof

Let us prove that *Choose(i)* is correct, assuming that when called its precondition is true, and that on exit from all the calls *Choose(i+1)* the post-condition of that procedure (ie that p has the same value as it did at the call) is true. We illustrate with an example. Suppose that *Choose(4)* is to be elaborated with p having the value *DBFACEG*. This call then has to process all permutations with prefix *DBF*. The first group starts with *DBFA*. *A* is already in position, and the swap instruction has no effect since it swaps *p[4]* and *p[4]*. *Choose(5)* is called with its precondition true, because *CEG* is in lexicographical order, and all permutations with the prefix *DBFA* are processed. On exit from *Choose(5)*, with p restored, the loop is entered, *p[4]* and *p[5]* are swapped, making *p=DBFCAEG*, and *Choose(5)* is called with its precondition true. Successive traverses of the loop set p to *DBFEACG* and *DBFGACE*. Thus the first conjunct of the post-condition is true, since all permutations with the *DBFA*, *DBFC*, *DBFE* and *DBFG* prefixes, and therefore with the *DBF* prefix, have been processed in lexicographical order. The second conjunct is assured by the rotating of the last 4 elements of *DBFGACE*, giving *DBFACEG*, which is, of course, the original value of *p*.

It is clear that on the initial call of *Choose(1)* its pre-condition is satisfied and that *Choose(n)* exits with its post-condition satisfied.

4. The relationship to Ord-Smith's algorithm

There is a sense in which all lexicographical algorithms are the same; they produce the same permutations in precisely the same order. The only differences lie in the control used to achieve this end. Therefore it is natural to ask how this algorithm relates to Ord-Smith's. Let us therefore transform the algorithm to produce his.

The last statement in any recursive procedure (here the *Rotate*) can always be removed and a modified version instead placed immediately after all calls for the procedure, including the initial call. The modification is necessary to take account of the changed values of any parameters between the two levels and to ensure that the statement is not obeyed at the bottom level. In this case

```
Rotate(p, k, n)        ⇒        if k < n then Rotate(p, k+1, n)
```

except in the body of the main procedure, where *k* is undefined, and we need only *Rotate(p, 1, n)*.

Look now at the loop. The conditional *Rotate* statement we have just introduced is obeyed on each traverse. We can modify the loop so that it is obeyed on all iterations but the last, and compensate by adding it explicitly after the loop, using the transformation:

```
for i := a to b do              for i := a to b do
  begin                         begin
    S1;              ⇒            S1;
    S2                            if i <> b then S2
  end                           end;
                                S2
```

The procedure *Choose* now looks like this:

```
procedure Choose (k: range);
{Pre : p[k..n] is in lexicographical order.}
{Post : All permutations with prefix p[1..k] have been processed lexicographically
        and p[k..n] has suffered a right shift.}
   var i: range;
begin
   for i := k to n do
      begin
         Swap(p[i], p[k]);
         if k = n then ProcessPerm(p, n) else Choose(k+1);
         if i <> n then
            if k < n then Rotate(p, k+1, n)
      end;
   if k < n then Rotate(p, k+1, n)
end;{of procedure "Choose"}
```

Note that this procedure has the same structure as it had originally, and we can repeat the pair of transformations. In the first phase, when moving

```
if k < n then Rotate(p, k+1, n)
```

into the loop the transformation produces

```
if k < n-1 then
   if k+1 < n then Rotate(p, k+2, n)
```

which of course can be reduced to

```
if k < n-1 then Rotate(p, k+2, n)
```

In the second phase, the relevant sequence inside the loop is

```
if k < n-1 then Rotate(p, k+2, n);
if i <> n then
   if k < n then Rotate(p, k+1, n)
```

During the transformation some simplifications can be made. Specifically, when making the sequence conditional we use the transformation:

```
if q then                                  if q then
   begin                                      begin
      S1;                 ⇒                      S1;
      if q then                                  S2
         S2                                   end
```

and when the copy is made outside the loop, where i≠n, we use the transformation:

```
if q then                                     if q then
   begin                  ⇒                      S1;
      S1;
      if false then S2
   end
```

After the second round of transformation *Choose* now looks like this:

```
procedure Choose (k: range);
{Pre : p[k..n] is in lexicographical order.}
{Post : All permutations with prefix p[1..k] have been processed lexicographically
        and p[k..n] has suffered a right shift,
        and then p[k+1..n] has suffered a right shift.}
   var i: range;
begin
   for i := k to n do
      begin
         Swap(p[i], p[k]);
         if k = n then ProcessPerm(p, n) else Choose(k+1);
         if i <> n then
            begin
               if k < n-1 then Rotate(p, k+2, n);
               if k < n then Rotate(p, k+1, n)
            end
         end;
   if k < n-1 then Rotate(p, k+2, n);
end;{of procedure "Choose"}
```

We can repeat the processes, symbolically at least, *n-1* times. It is clear that the process produces a sequence of *Rotate* statements:

```
if k < 2 then Rotate(p, n-1, n);
  ....
if k < n-1 then Rotate(p, k+2, n);
if k < n then Rotate(p, k+1, n).
```

after the calls of *Choose*. These can be expressed as a loop:

```
for j := n-1 downto k+1 do
   Rotate(p, j, n)
```

The resulting procedure is given in Fig 2.

```
procedure Lex (p: perm; n: range);
{Pre : p[1..n] contains the marks in lexicographical order.}
{Post : All permutations have been processed in lexicographical order.}
   var j: range;

   procedure Choose (k: range);
   {Pre : p[k..n] is in lexicographical order.}
   {Post : All permutations with prefix p[1..k] have been processed lexicographically}
   {          and p[k..n] has suffered a right shift,}
   {          and then p[k+1..n] has suffered a right shift}
   {          and so on.}
      var i, j: range;
   begin
      for i := k to n do
         begin
            Swap(p[i],  p[k]);
            if k = n then ProcessPerm(p, n) else Choose(k + 1);
            if i <> n then
               for j := n - 1 downto k + 1 do
                  Rotate(p, j, n)
         end
   end;{of procedure "Choose"}
```

```
begin
   Choose(1);
   for j := n - 1 downto 1 do
      Rotate(p, j, n)
end;{of procedure "Lex"}
```

Fig 2. The transformed lexicographical algorithm

Now, the key observation, the loop of *Rotates* reverses the elements of p between the $k+1^{\text{th}}$ and n^{th} positions! Replacing it produces the procedure of Fig 3, which is a recursive rendering of Ord-Smith's procedure.

```
procedure OrdSmith (p: perm; n: range);
{Pre : p[1..n] contains the marks in lexicographical order.}
{Post : All permutations have been processed in lexicographical order.}

   procedure Choose (k: range);
   {Pre : p[k..n] is in lexicographical order.}
   {Post : All permutations with prefix p[1..k] have been processed lexicographically
            and p[k..n] is in reverse lexicographical order.}
      var i: range;
   begin
      for i := k to n do
         begin
            Swap(p[i], p[k]);
            if k = n then ProcessPerm(p, n) else Choose(k+1);
            if i <> n then Reverse(p, k+1, n)
         end
   end;{of procedure "Choose"}

begin
   Choose(1);
   Reverse(p, 1, n)
end;{of procedure "OrdSmith"}
```

Fig 3. Ord-Smith's lexicographical algorithm

Note the post-condition on the procedure *Choose*. Note, too, the *Reverse* in the main procedure can be eliminated as p is a value parameter. If, however, p were called as a variable it might be useful to retain it because it restores p to its original value.

5. Backtracking

To adapt the new procedure to backtracking is trivial, because the nett result on p of any call of *Choose* is nil. Suppose that we have a predicate $q(p, k)$ which is true iff the permutation whose prefix is $p[1..k]$ can lead to a solution. We simply make

```
if k=n then ProcessPerm else Choose(k+1)
```

conditional on $q(p, k)$. Figure 4 gives a procedure for generating the derangements of n marks. These are permutations in which the mark i does not appear in the i^{th} position.

```
procedure Derangements (p: perm; n: range);
{Pre : p[1..n] contains the marks in lexicographical order.}
{Post : All derangements have been processed in lexicographical order.}

  procedure Choose (k: range);
  {Pre : p[k..n] is in lexicographical order.}
  {Post : All derangements with prefix p[1..k] have been processed lexicographically
        and p[k..n] is (back) in lexicographical order.}
    var i: range;
  begin
    for i := k to n do
      begin
        Swap(p[i], p[k]);
        if p[k] <> k then
          if k = n then ProcessPerm(p, n) else Choose(k+1);
      end;
    Rotate(p, k, n)
  end;{of procedure "Choose"}

begin
  Choose(1)
end;{of procedure "Derangements"}
```

Fig 4. Lexicographical derangements algorithm

To adapt Ord-Smith's algorithm to such tasks is a little more onerous because any call of *Choose(k)* reverses *p[k+1..n]* as a side-effect. Referring to Fig 2, we see that the call of *Choose* is immediately followed by a conditional reverse, whose effect must be simulated. The loop becomes:

```
for i := k to n do
  begin
    Swap(p[i], p[k]);
    if p[k] <> k then
      begin
        if k = n then ProcessPerm(p, n) else Choose(k+1);
        if i <> n then Reverse(p, k+1, n)
      end
    else
      if i = n then Reverse(p, k+1, n)
  end
```

6. Generating the *r*-permutations of *n*

One important problem associated with permutations is the production of the so-called r-permutations of n. These are permutations consisting of r marks, rather than n, chosen from the n. This is easily accomplished with the lex procedure. First, the parameter list must be expanded to cater for the value of r. Second, the procedure is modified so that the recursion terminates when *k* is equal to *r* rather than when it is equal to *n*, and of course *ProcessPerm* is called with its second parameter also equal to *r*. Figure 5 gives the procedure.

```
procedure RPermutationsOfN (p: perm; r, n: range);
{Pre : p[1..n] contains the marks in lexicographical order.}
{Post : All r-permutations have been processed in lexicographical order.}

  procedure Choose (k: range);
  {Pre : p[k..n] is in lexicographical order.}
  {Post : All r-permutations with prefix p[1..k] have been processed lexicographically
          and p[k..n] is (back) in lexicographical order.}
    var i : range;
  begin
    for i := k to n do
      begin
```

```
      Swap(p[i], p[k]);
      if k = r then ProcessPerm(p, r) else Choose(k+1);
    end;
   Rotate(p, k, n)
  end;{of procedure "Choose"}

begin
  Choose(1)
end;{of procedure "RPermutationsOfN"}
```

Fig 5. The non-reversing lexicographical algorithm for generating *r*-permutations of *n*

Once again Ord-Smith's algorithm can be adapted, but the same problems arise, as arose with the back-tracking procedures. However, in this case, since the conditions involved do not include elements of *p*, simplifications can be made to produce the inner loop:

```
for i := k to n do
  begin
    Swap(p[i], p[k]);
    if k = r then ProcessPerm(p, r) else Choose(k+1);
    if (i = n) = (k = r) then Reverse(p, k+1, n)
  end;
```

7 Permutations with repeated marks

Another related problem is that of generating permutations of marks some of which may be repeated. The procedures of Fig 1 and Fig 2 produce the full n! permutations, with many repetitions. It is usually desirable that these duplicates be eliminated, so that the permutations generated for $p = AAB$ are AAB, ABA and BAA. This is easily accomplished, by what amounts to a backtracking process: at each level we simply skip an element which has already been chosen. It will always be the one chosen immediately prior to the one being considered. A simple condition

```
(i = k) or (p[k] <> p[i])
```

determines whether the mark in *p[k]* is a valid choice. The procedure is given in Fig 6.

```
procedure RepeatedMarks (p: perm; n: range);
{Pre : p[1..n] contains the marks in lexicographical order.}
{Post : All distinct permutations have been processed in lexicographical order.}

  procedure Choose (k: range);
  {Pre : p[k..n] is in lexicographical order.}
  {Post : All permutations with prefix p[1..k] have been processed lexicographically
        and p[k..n] is (back) in lexicographical order.}
    var i : range;
  begin
    for i := k to n do
      begin
        Swap(p[i], p[k]);
        if (i = k) or (p[k] <> p[i]) then
          if k = n then ProcessPerm(p, n) else Choose(k+1);
      end;
    Rotate(p, k, n)
end;{of procedure "Choose"}

begin
  Choose(1)
end;{of procedure "RepeatedMarks"}
```

Fig 6. The non-reversing lexicographical algorithm which deals with repetitions

To adapt the Ord-Smith version also requires similar treatment to that required for backtracking.

8. Some observations

The title of this paper is somewhat facetious. Of course, this algorithm does reverse the elements of p. However, it does so, not with a *Reverse* statement but with a series of distributed *Rotate* statements, and this is its advantage. A call to *Choose* has a null effect on p, and therefore can be skipped if required. When the *Rotates* are combined into a *Reverse*, a call to *Choose* reverses p, and that effect must be simulated. This gives complexity to the resulting procedure. The reader is referred to Wilson [7] for further evidence of this.

References

1. Hitotumatu,H. and Noshita,K. (1979) *A technique for implementing backtrack algorithms and its applications*, Inform. Process. Lett. 8(4) 174-175.

2. Irving,R.W. (1984) Permutation Backtracking in Lexicographical Order, Comput. J. 27(4) 373-374.

3. Ord-Smith, R.J.(1968) *Generation of permutations in lexicographical order*, Comm. A.C.M. 11(2) 117.

4. Sedgewick, R. (1977) *Permutation Generation Methods*, Comp. Surveys 9(2), 136-164.

5. Rohl, J.S. (1978) *Generating permutations by choosing*, Comput.J. 21(4) 302-305.

6. Rohl, J.S. (1983) *A faster lexicographical N-queens algorithm*, Inform. Process. Lett. 17(5) 231-233.

7. Wilson, J.M. (1983) *Interrupted permutations in lexicographic order*, Comput. J. 26(1) 93.