Programming improvements to Fike's algorithm for generating permutations

J. S. Rohl

Department of Computation, The University of Manchester Institute of Science and Technology, PO Box 88, Manchester M60 1QD

In a recent article Fike describes an elegant new algorithm for generating permutations and gives two procedures implementing this algorithm. In this paper we give improved versions with comparative measurements on two machine/compiler systems. The results show that on one of the systems a recursive procedure is the fastest.

(Received July 1975)

In a recent number of this Journal, Fike (1975) has given a new algorithm for generating permutations, which his experiments indicate is comparable with the fastest known methods (see Ord-Smith, 1970 and 1971). In this paper we consider some programming modifications to the procedures Fike presents which improve the performance by a factor of about 2. The timings show that with a compiler which handles procedure calls reasonably, a recursive procedure is the most efficient.

The algorithm

Fike's algorithm is based on the following observations:

1. There are $n!$ permutations of the n marks $1, 2, 3, \ldots, n$.
2. There are $n!$ sequences $(d_2, d_3, \ldots, d_n)$ where d_k denotes an integer such that $1 \leqslant d_k \leqslant k$.
3. These sequences may be trivially generated.
4. There is a one-to-one correspondence between the sequences and the permutations.
5. A permutation may be derived from the corresponding sequence by a simple algorithm: starting with the original arrangement of the marks, interchange the mark in position k with the mark in position d_k for $k = 2, 3, \ldots, n$.

Fig. 1 which is taken directly from Fike's paper illustrates the algorithm, applied to the marks 1, 2, 3 and 4.

At a first glance it seems that $n - 1$ interchanges are required to produce each permutation; but on closer inspection it is clear that if the permutations are generated in the order of Fig. 1 then in most cases two interchanges will suffice. Further, we can arrange that each interchange of the pair is the same, the first producing the required permutation, the second returning to the status quo ready for the next permutation.

Fike's recursive procedure

Fike has given a delightfully elegant recursive procedure for implementing this algorithm. The original is in PL/I but since the author has no PL/I compiler available he has used ALGOL 60 instead. The ALGOL 60 transliteration of Fike's procedure is given in **Fig. 2**, where the place at which a user of the procedure might process the permutation is indicated by the appropriate comment, even though it is syntactically invalid.

Permutation	*Sequence*	*Permutation*	*Sequence*
1234	(2, 3, 4)	2134	(1, 3, 4)
1243	(2, 3, 3)	2143	(1, 3, 3)
1432	(2, 3, 2)	2431	(1, 3, 2)
4231	(2, 3, 1)	4132	(1, 3, 1)
1324	(2, 2, 4)	2314	(1, 2, 4)
1342	(2, 2, 3)	2341	(1, 2, 3)
1423	(2, 2, 2)	2413	(1, 2, 2)
4321	(2, 2, 1)	4312	(1, 2, 1)
3214	(2, 1, 4)	3124	(1, 1, 4)
3241	(2, 1, 3)	3142	(1, 1, 3)
3412	(2, 1, 2)	3421	(1, 1, 2)
4213	(2, 1, 1)	4123	(1, 1, 1)

Fig. 1 The 4! permutations of the integers 1, 2, 3, 4 and the 4! sequences to which they correspond

An improved recursive procedure

Consideration of Fig. 2 leads to the following observations:

1. Suppose we define the level of calling of the procedure as starting at 2; then the level is synonymous with the parameter k. At level k the only element of d that is referenced is $d[k]$, so that the array d as such is redundant. It is sufficient to use a scalar at each level for the appropriate element.
2. At the end of each loop $p[k]$ is assigned a value which is immediately over-written at the start of the next iteration. The assignment then needs only to be done on the last iteration and can therefore be moved outside the loop.

From these observations we produce the improved version of **Fig. 3**.

Fig. 4 gives some measurements taken on two machines, an ICL 1906A using the Manchester ALGOL compiler and a CDC 7600 which is nominally eight times faster using CDC's

```
procedure generate permutations(n);
value n; integer n;
comment This generates the n! permutations of the digits
    1, 2, . . ., n;
 begin
 integer array p[1:n], d[2, n];
 procedure permute(k);
 value k; integer k;
 comment This procedure controls the interchanges at level k;
  begin
  integer temp;
  temp := p[k];
  for d[k] := k step − 1 until 1 do
      begin
      p[k] := p[d[k]];
      p[d[k]] := temp;
      if k < n then permute(k + 1)
      else comment permutation available;
      p[d[k]] := p[k];
      p[k] := temp
      end of loop varying d_k
  end of procedure permute;
 integer i;
 for i := 1 step 1 until n do p[i] := i;
 permute(2)
 end of procedure generate permutations
```

Fig. 2 Fike's original recursive procedure transliterated into ALGOL 60

```
procedure generate permutations(n);
value n; integer n;
comment This generates the n! permutations of the digits
    1, 2, . . ., n;
 begin
 integer array p[1:n];
 procedure permute(k);
 value k; integer k;
 comment This procedure controls the interchanges at level k;
  begin
  integer temp, dk;
  temp := p[k];
  for dk := k step -1 until 1 do
      begin
      p[k] := p[dk];
      p[dk] := temp;
      if k < n then permute(k + 1)
      else comment permutation available;
      p[dk] := p[k]
      end of loop varying d_k;
  p[k] := temp
  end of procedure permute;
 integer i;
 for i := 1 step 1 until n do p[i] := i;
 permute(2)
 end of procedure generate permutations
```

Fig. 3 An improved recursive procedure

ALGOL 4·0 compiler. If we concentrate on the ICL 1900 compiler we see a gain of 43 to 45 per cent, [the author could not afford to run the program for $n = 10$!], the gain improving slowly with increasing n. These figures are subject to a measurement error of some one to two per cent.

The reason is obvious: not only is the number of interchanges important, here $2n! + 2(n-1)! + \ldots 2$ instead of $n! - 1$ for other algorithms, but also the cost of accessing the elements to be interchanged, and the cost of deciding which elements to interchange. [Further, the value of n is unlikely ever to exceed 10 for cost reasons alone]. The improved algorithm reduces the cost of loop control and element accessing leaving the number of interchanges the same. CDC's ALGOL 4·0 shows less improvement, 23 to 25 per cent, because the cost of procedure entry is much more dominant. On the 1906A entry (and exit) to a procedure with one value parameter takes 18·4 μsecs while on the 7600 it takes 31·8 μsecs.

A tuned procedure

It is clear that the improved procedure is a better procedure all round in that it runs faster, occupies less store and has eliminated a redundant concept. If speed is the criterion we can produce a tuned procedure, at the cost of storage, based on the observations:

```
procedure generate permutations(n);
value n; integer n;
comment This generates the n! permutations of the digits
    1, 2, . . ., n;
 begin
 integer array p[1:n];
 procedure permute(k);
 value k; integer k;
 comment This procedure controls the interchanges at level k;
  begin
  integer temp, dk, dn;
  if k = n then
      begin
      comment permutation available;
      temp := p[n];
      for dn := n - 1 step -1 until 1 do
          begin
          p[n] := p[dn];
          p[dn] := temp;
          comment permutation available;
          p[dn] := p[n]
          end of loop varying d_n;
   p[n] := temp
   end of sequence for bottom level
  else begin
      permute(k + 1);
      temp := p[k];
      for dk := k - 1 step -1 until 1 do
        begin
        p[k] := p[dk];
        p[dk] := temp;
        permute(k + 1);
        p[dk] := p[k]
        end of loop varying d_k;
   p[k] := temp
   end of sequence for other levels
  end of procedure permute;
 integer i;
 for i := 1 step 1 until n do p[i] := i;
 permute(2)
 end of procedure generate permutations
```

Fig. 5 A tuned recursive procedure

1. At each level, the interchanges performed at the start and end of each traverse of the loop are redundant since the interchange takes place between $p[k]$ and $p[k]$.
2. The test for the bottom level ($k = n$) takes place inside the loop (where k is constant).

Fig. 5 gives a tuned procedure.

Fig 4 shows times for this procedure, showing a further improvement of approximately 10 per cent over the original. On the 1906A the procedure runs in less than half the time, a gain of 55 to 56 per cent.

	$n = 8$		$n = 7$	
	ICL 1906A Manchester	*CDC 7600 ALGOL* 4·0	*ICL 1906A Manchester*	*CDC 7600 ALGOL* 4·0
Fike's Recursive Procedure	4·53 seconds (1·00)	0·936 seconds (1·00)	0·58 seconds (1·00)	0·126 seconds (1·00)
Improved Procedure	2·51 seconds (0·55)	0·703 seconds (0·75)	0·33 seconds (0·57)	0·098 seconds (0·77)
Tuned Procedure	1·98 seconds (0·44)	0·637 seconds (0·68)	0·26 seconds (0·45)	0·090 seconds (0·71)

Fig. 4 Comparison of the performance of the three recursive procedures on two different machines

Fike's iterative procedure

Fike has given an iterative procedure as well, in which, as he says, the relation to the algorithm is not so obvious. **Fig. 6** gives a transliterated version in ALGOL 60 (which, as it happens, illustrates the weaknesses in the ALGOL 60 looping construct). It is an interesting exercise to derive this procedure from the original recursive one, and we leave that to the reader.

Measurements for this procedure operating under the two systems are given in **Fig. 7.**

Although this non-recursive procedure is faster than the original recursive one (20 per cent on the 1906A, 26 per cent on the 7600) it is slower than either the improved or the tuned recursive algorithms. Perusal of Fig. 6 shows why. Each permutation involves, as well as the double interchange with its heavy dependence of array accessing, the two tests associated with the loops and a further test of k against 1. Since for given values of $(d_2, d_3, \ldots, d_{n-1})$ there are n permutations, we can recover some of the inefficiency by producing these n permutations in a loop. **Fig. 8** gives such a procedure. This is not derived from Fike's original non-recursive procedure, but from the recursive one, through a different route. There is a clear affinity between this and the tuned recursive one.

As Fig. 7 shows this procedure is some 40 per cent better than the original. There are still some unnecessary interchanges though we have not considered whether the procedure could be tuned with any significant gain.

```
procedure generate permutations(n);
value n; integer n;
comment This generates the n! permutations of digits 1, 2, . . ., n;
 begin
 integer array p[1:n], d[2:n];
 integer i, dummy, temp, k;
 Boolean more;
 for i := 1 step 1 until n do p[i] := i;
 for i := 2 step 1 until n do d[i] := i;
 comment first permutation available;
 more := true;
 for dummy := 0 while more do
  begin
  k := n;
  for dummy := 0 while k ≠ 1 ∧ d[k] = 1 do
   begin
   temp := p[k];
   p[k] := p[1];
   p[1] := temp;
   d[k] := k;
   k := k − 1
   end;
  if k = 1 then more := false
  else begin
   temp := p[d[k]];
   p[d[k]] := p[k];
   p[k] := p[d[k] − 1];
   p[d[k] − 1] := temp;
   d[k] := d[k] − 1;
   comment next permutation available;
   end
  end
 end of procedure generate permutations
```

Fig. 6 Fike's non-recursive procedure transliterated in to ALGOL 60

	$n = 8$	
	ICL 1906A Manchester	*CDC 7600 ALGOL* 4·0
Fike's Non-Recursive Procedure	3·62 seconds (1·00)	0·693 seconds (1·00)
Improved Procedure	2·05 seconds (0·57)	0·432 seconds (0·62)
	$n = 7$	
	ICL 1906A Manchester	*CDC 7600 ALGOL* 4·0
Fike's Non-Recursive Procedure	0·46 seconds (1·00)	0·092 seconds (1·00)
Improved Procedure	0·27 seconds (0·57)	0·057 seconds (0·62)

Fig. 7 Comparison of the performance of the two non-recursive procedures on two different machines

Procedure structure

As it stands each procedure generates the permutations of the natural numbers 1, 2, 3, . . . because the array p is initialised to that value on entry. The body of the procedure is, however, invariant to the nature of the marks and so permutations of a

```
procedure generate permutations(n);
value n, integer n;
comment This generates the n! permutations of the digits
     1, 2, . . ., n;
 begin
 integer array p[1:n], d[1:n − 1];
 integer i, temp, dummy, dn, k;
 Boolean more;
 for i := 1 step 1 until n do p[i] := i;
 for i := 1 step 1 until n − 1 do d[i] := i;
 more := true;
 for dummy := 0 while more do
  begin
  comment permutation available;
  temp := p[n];
  for dn := n − 1 step −1 until 1 do
      begin
      p[n] := p[dn];
      p[dn] := temp;
      comment permutation available;
      p[dn] := p[n]
      end of loop producing n permutations;
  p[n] := temp;
  temp: = p [d[n − 1]];
  p[d[n − 1]] := p[n − 1];
  p[n − 1] := temp;
  k := n − 1;
  for dummy := 0 while k > 2 ∧ d[k] = 1 do
      begin
      d[k] := k;
      k := k − 1;
      temp := p[d[k]];
      p[d[k]] := p[k];
      p[k] := temp
      end of loop moving up completed levels;
  if d[k] = 1 then more := false
  else begin
      d[k] := d[k] − 1;
      temp := p[k];
      p[k] := p[d[k]];
      p[d[k]] := temp
      end of sequence at first uncompleted level
 end of loop producing all permutations
end of procedure generate permutations
```

Fig. 8 An improved non-recursive procedure

different set of marks can be produced merely by changing the assignment to p.

If permutations of the natural numbers are required, all three new procedures could be improved slightly. The variable *temp*, is redundant, its value being known. For example, in the recursive procedures on entry to permute at level k, $p[k] = k$.

To return to the general case, the specification of the procedure could have been altered to include p as a parameter. This has not been done because it is not clear that either structure is the best.

The family of procedures described here have a common structure. A procedure is called once and generates all the permutations, these permutations being processed by code embedded within the procedure or by subroutine. The procedures described by Ord-Smith have all been cast in a form in which the permutation procedure is called $n!$ times as a subroutine of the procedure processing the permutations. This suggests that all the procedures might better be recast as co-routines.

Conclusions

In this paper we report the results obtained by accepting an algorithm and taking a programmer's view of the procedure implementing this algorithm. Gains of between 40 to 55 per cent have been obtained. However, when we compare Fig. 4 with Fig. 7 we see that the improvements are very much machine and compiler dependent. If we rank the five procedures for the two systems considered, we produce two different rankings. On the 1906A, the best procedure, marginally, is a recursive one: on the 7600 it is, by over 30 per cent, a non-recursive one.

That the recursive procedure on the 1906A should prove the fastest is of significance in that it supports Fike's observations about his original algorithms on IBM 360. The figures suggest that the much quoted assertion that recursion is inefficient (which the figures for 7600, on their own, would support) is an assertion more about the compiler used than about recursion itself. If procedure entry and exit is handled sensibly, as it is on the 1906A compiler used, then recursion is not expensive, and, where the algorithm involved is essentially recursive, can be used to advantage.

Acknowledgements

I should like to record my thanks to C. T. Fike who, on reading the first draft, pointed out a serious weakness in the procedures, and to Mrs. E. M. J. Chadwick who tested and timed them all.

References

FIKE, C. T. (1975). A permutation generation method, *The Computer Journal*, Vol. 18, pp. 21-22.
ORD-SMITH, R. J. (1970). Generation of permutation sequences: part 1, *The Computer Journal*, Vol. 13, pp. 152-155.
ORD-SMITH, R. J. (1971). Generation of permutation sequences: part 2, *The Computer Journal*, Vol. 14, pp. 136-139.

Book review

The Principles of Systems Programming, by Robert M. Graham, 1975; 422 pages. (*John Wiley*, £8·25)

In writing this book, the author has tackled a formidable task. The phrase 'systems programming' is taken, particularly in the USA, to include almost all programs except those specifically directed towards applications. The book is correspondingly long, (about 180,000 words) and contains several major sections:

An introduction to systems	(30 pages)
Machine and assembly languages, assemblers, macroprocessors and loaders	(92 pages)
Programming languages and compilers	(113 pages)
Operating systems	(130 pages)
Appendices	(40 pages)

The book has many appealing features. A strong unifying thread is the consistent use of INSTRAN, a 'private' high-level language somewhat similar to Pascal or ALGOL 60, throughout most of the main sections. INSTRAN serves to describe all the algorithms used for assembly, loading, compilation, scheduling and peripheral operation, and itself forms the example for an extensive discussion on language translation techniques.

The text is always readable, and markedly fresher at the end than at the beginning of the book, where many of the sentences are cumbersome and repetitious. The best part of the book is the last section on Operating Systems. My guess is that this is what really interests the author. The section begins with an introduction which singles out and describes some of the important characteristics of a modern multi-access system such as its response ratio, file storage, arrangements for privacy and protection, and adaptability to change. Next, there is a good chapter on process control and communication. The writer describes both the classical P and V operations and the alternative 'block', 'wake-up' and message passing mechanisms. The chapter on memory management covers segmentation and paging (without confusing the two) and the organisation of file stores. The next chapter deals with input and output of all kinds, giving several device driver routines in INSTRAN. The section ends with a short chapter on the sharing of information and protection against unauthorised access.

The rest of the book filled me with somewhat less enthusiasm. For all that the text is described as 'machine independent', it is clear that most of the writer's experience was gained on the IBM 360 series and on the various incarnations of MULTICS. Thus, although very little is said about job control languages, those samples which are presented are derived from OS 360. The author sees nothing wrong with the seven job control statements needed to compile and run a simple program in PL/I, and nowhere does he suggest that a JCL might be a genuine programming language instead of a series of rigid primitive commands. The lack of any discussion on this point, and of any suggestion that the user interface might be an important aspect of an operating system, are two unfortunate omissions from the text.

The section on assemblers and loaders again seems to lack breadth. No mention is made of a one-pass assembler, and the various details, which are exceedingly precise, refer quite specifically to the IBM 360 computer. Similarly, the section on compilers to a specific (and not generally known) language, and such general information as it contains is better presented in other books such as those by Gries or Rohl.

The book gives only 19 references, and they are scattered throughout the text instead of being collected together as is normal. However, the names of the authors given—Dijkstra, Wirth, Gries, Brown, Knuth, Denning, etc. are reassuring.

In conclusion, the section on Operating Systems will make good background reading for Computer Science students, and those who teach assembly code programming in the context of the IBM 360 might find that section of the book a possible alternative to the standard IBM documentation as source material. The book will be a useful addition to a college library, but—in view of the price—the student would do well to spend his book allowance elsewhere.

A. J. T. COLIN (Strathclyde)