Algorithms Supplement

Previously published algorithms

The following Algorithms have been published in the *Communications of the Association for Computing Machinery* during the period May–June 1967.

301 AIRY FUNCTION

Evaluates the real Airy functions and their derivatives by solution of the differential equation $y'' = xy$.

302 TRANSPOSE VECTOR STORED ARRAY

Performs an in-situ transposition of an $m \times n$ array $A[1:m, 1:n]$ stored by rows in the vector $a[1:m \times n]$.

303 AN ADAPTIVE QUADRATURE PROCEDURE WITH RANDOM PANEL SIZES

Approximates the quadrature of the function fx on the interval $a < x < b$ to an estimated accuracy by sampling the function fx at appropriate points until the estimated error is less than the estimated accuracy.

304 NORMAL CURVE INTEGRAL

Calculates the tail area of the standardized normal curve.

The following papers have been published in *Nordisk Tidskrift for Informationsbehandling* in the January 1967 issue.

(*a*) COMPUTER CARTOGRAPHY-POINT-IN-POLYGON PROGRAMS.
(*b*) REMARKS ON "GARBAGE COLLECTION" USING A TWO-LEVEL STORAGE.

Algorithms

Author's Note on Algorithms 22, 23, 24

In a recent paper, T. A. J. Nicholson (1966) describes a fast algorithm for finding the shortest route between two points in a connected network and compares this with other methods. Of the three procedures given below, *minpath* implements Nicholson's algorithm and provides one and only one solution for a given pair of nodes, whereas *netpaths* and *shortpath* are associated procedures which together may be used to find the shortest path between any specified node and all others. The original source of *netpaths* and *shortpath* is not known to the author though the essential method is that described in Wilson.

References

NICHOLSON, T. A. J. (1966). Finding the shortest route between two points in a network, *The Computer Journal*, Vol. 9, pp. 275–280.

WILSON, R. C. Example Problem 61, *The Use of Computers in Industrial Engineering Education.* Ann Arbor: College of Engineering, The University of Michigan.

Algorithm 22

SHORTEST PATH BETWEEN START NODE AND END NODE OF A NETWORK

J. Boothroyd,
Hydro-University Computing Centre,
University of Tasmania.


```
integer procedure minpath (d, n, sn, en, route); value n, sn, en;
integer n, sn, en; integer array d, route;
comment yields the value of the shortest path between start
node sn and end node en of a connected n-node network having
up to n × (n−1) directed links. d[1 : n, 1 : n] is the cost, or
distance, matrix with elements d[i, j] containing the cost
(distance) of the ij directed link between nodes i and j.
The diagonal elements of d and all d[p, q] elements associated
with pq directed open links between nodes p and q should
contain M = n × max (d[i, j]) i.e. n times the maximum con-
nected link value.
As this algorithm requires the diagonal elements to be zero the
procedure clears these after entry and restores them again
before exit.
The array route[1 : n] contains, in its first m positions, the
numbers of the m(≤ n) nodes in the connected chain forming the
shortest path. The remaining elements of route are set to zero;

begin integer i, j, k, gp, fp, si, ti, mins, mint, sum, x, y, max,
    dmi, m, min, imin;
  integer array p, q, s, t, f, g[1 : n];
  procedure smin;
  comment finds mins and stores in stack f[1 : fp] all values
  of m such that s[m] = mins (s[i] > x);
  begin si := s[i];
    if si > x then
    begin if si < mins then
      begin fp := 1; mins := si;
        f[fp] := i
      end
      else
      if si = mins then
      begin fp := fp + 1;
        f[fp] := i
      end
    end
  end smin;
  procedure tmin;
  comment finds mint and stores in stack g[1 : gp] all values
  of m such that t[m] = mint (t[i] > y);
  begin ti := t[i];
    if ti > y then
    begin if ti < mint then
      begin gp := 1; mint := ti;
        g[gp] := i
      end
      else
      if ti = mint then
      begin gp := gp + 1;
        g[gp] := i
      end
    end
  end tmin;
```

```
    comment pick up max and initialize x, y, s, p, q, t and the
    diagonal of d;
    max := d[1, 1]; x := y := 0;
    for i := 1 step 1 until n do
    begin d[i, i] := 0;
      s[i] := d[sn, i]; t[i] := d[i, en];
      p[i] := sn; q[i] := en
    end initialization;
    comment find the initial values of mins and mint with
    corresponding m values for both s[1 : n] and t[1 : n];
    fp := gp := 0; mint := mins := max;
    for i := 1 step 1 until n do
    begin smin;
      tmin
    end;
    comment the algorithm proper begins;
iterate: if mins ≤ mint then
    begin comment reset s[1 : n];
      x := mins;
      for fp := fp step −1 until 1 do
      begin m := f[fp];
        for i := 1 step 1 until n do
        begin dmi := d[m, i];
          sum := mins + dmi;
          if s[i] > sum then
          begin s[i] := sum;
            p[i] := m
          end
        end
      end;
      comment find new mins and m values for s[1 : n];
      mins := max; fp := 0;
      for i := 1 step 1 until n do smin
    end
    else
    begin comment reset t[1 : n];
      y := mint;
      for gp := gp step −1 until 1 do
      begin m := g[gp];
        for i := 1 step 1 until n do
        begin dmi := d[i, m];
          sum := mint + dmi;
          if t[i] > sum then
          begin t[i] := sum;
            q[i] := m
          end
        end
      end;
      comment find new mint and m values for t[1 : n];
      mint := max; gp := 0;
      for i := 1 step 1 until n do tmin
    end;
    comment compute convergence criterion;
    min := max + max;
    for i := 1 step 1 until n do
      begin sum := s[i] + t[i];
      if sum < min then
      begin min := sum;
        imin := i
      end
    end;
    if min > mins + mint then goto iterate;
    comment the two ends of one shortest route (there may be
    others equally short) meet in node imin. Now to unravel the
    route;
    j := route[n] := imin;
    if imin ≠ sn then
    begin k := n − 1;
      for i := p[j] while i ≠ sn do
      begin j := route[k] := i;
        k := k − 1
      end
    end
    else
    k := n;
    route[1] := sn; j := k + 1; k := 2;
    for j := j step 1 until n do
    begin route[k] := route[j];
      k := k + 1
    end;
    if imin ≠ en then
    begin j := imin;
      for i := q[j] while i ≠ en do
      begin j := route[k] := i;
        k := k + 1
      end;
      route[k] := en
    end;
    for k := k + 1 step 1 until n do route[k] := 0;
    comment restore the diagonal of d;
    for i := 1 step 1 until n do d[i, i] := max;
    minpath := s[imin] + t[imin]
end minpath
```

Algorithm 23

SHORTEST PATH BETWEEN START NODE AND ALL OTHER NODES OF A NETWORK

J. Boothroyd,
Hydro-University Computing Centre,
University of Tasmania.

```
procedure netpaths (d, n, sn, precede, mincost); value n, sn;
integer array d, precede, mincost; integer n, sn;
comment yields in mincost[i] of mincost[1 : n] the value of
the shortest path from node sn to all other nodes i, i = 1, 2 . . . n,
in a connected n-node network having up to n × (n − 1)
directed links. d[1 : n, 1 : n] is the cost, or distance, matrix
with elements d[i, j] containing the cost (distance) of the ij
directed link between nodes i and j. The diagonal elements and
elements d[p, q] associated with pq directed open links between
nodes p and q should contain M=n × max (d[i, j]) i.e. n times
the maximum connected link value.
The array precede[1 : n] is a chained list of node numbers
such that precede[i] contains the node number preceding node i
on the shortest route. This array may subsequently be used
by procedure shortpath to evaluate the list of nodes on the
shortest route from sn to any specified end node;

begin integer i, j, mini, jcost, M;
  integer array scan[1 : n];
  M := d[1, 1];
  for i := 1 step 1 until n do
  begin scan[i] := precede[i] :=0;
    mincost [i] := M
  end;
  mincost[sn] := 0; scan[sn] := 1;
iterate: for i := 1 step 1 until n do
  if scan[i] ≠ 0 then
```

```
    begin mini := mincost[i];
      for j := 1 step 1 until n do
      begin jcost := d[i,j] + mini;
        if jcost < mincost[j] then
        begin mincost[j] := jcost;
          scan[j] := 1;
          precede[j] := i
        end
      end;
      scan[i] := 0; goto iterate
    end
end netpaths
```

Algorithm 24

THE LIST OF NODES ON THE SHORTEST PATH FROM START NODE TO END NODE OF A NETWORK

J. Boothroyd,
Hydro-University Computing Centre,
University of Tasmania.

```
procedure shortpath (n, sn, en, precede, route); value n, sn, en;
integer n, sn, en; integer array precede, route;
comment evaluates in the first m(≤n) positions of route[1 : n]
the list of nodes on the shortest path from start node sn to end
node en in an n-node connected network. The remaining
elements of route are set to zero.
Information necessary for determining the path must be
supplied in precede[1 : n] in the form obtained by previous
use of procedure netpaths;
begin integer i, j, k;
  j := route[n] := en; k := n − 1;
  for i := precede[j] while i ≠ sn do
  begin j := route[k] := i;
    k := k − 1
  end;
  route[1] := sn; j := k + 1; k := 2;
  for j := j step 1 until n do
  begin route[k] := route[j];
    k := k + 1
  end;
  for j := k step 1 until n do route[k] := 0;
end shortpath
```

Author's Note on Algorithms 25, 26, 27

Some justification is surely needed for the publication of yet another sorting procedure using the method of partition on the rank of selected elements. Hibbard (1963) describes the essential process in his Program B and notes its similarity to Hoare's (1961) Quicksort in which the method is implemented as a recursive ALGOL procedure.

With the publication of the non-recursive implementation Quickersort (Scowen, 1965) it might be supposed that the final word has been said. However, the efficiency of an ALGOL procedure is a function of both the method and its implementation and *partsort*, given below, appears on test to be not less than 15% faster than Quickersort. This has been achieved largely by minimizing array access.

Other tests (Blair, 1965) show the general superiority of this method for internal sorting and it has been chosen as the basis for the procedure *keysort*, also given below.

An understanding of the operation of *keysort* is more easily had if details of the procedure on which it is based are available. This offers a further excuse for the publication of *partsort*.

[In procedures *partsort* and *keysort*, for sorting small numbers of elements and at the expense of extra storage, increased efficiency may be had by avoiding one block entry as follows:—

delete lines 4 and 5 of the procedure body,
i.e., **begin comment** – – –
– – – **do** $k := k + 1$
alter line 7 to read **integer array** *f,g* [1 : size]
one line from end: delete **end**

—Referee];

References

HIBBARD, T. N. (1963). An Empirical Study of Minimal Storage Sorting, *Communications of the Association for Computing Machinery*, Vol. 6, p. 207.

HOARE, C. A. R. (1961). Algorithm 63, Partition and Algorithm 64, Quicksort, *Communications of the Association for Computing Machinery*, Vol. 4, pp. 321–2.

SCOWEN, R. S. (1965). Algorithm 271, Quickersort, *Communications of the Association for Computing Machinery*, Vol. 8, p. 669.

BLAIR, C. R. (1965). Certification of Algorithm 271, *Communications of the Association for Computing Machinery*, Vol. 9, p. 354.

Algorithm 25

SORT A SECTION OF THE ELEMENTS OF AN ARRAY BY DETERMINING THE RANK OF EACH ELEMENT

J. Boothroyd,
Hydro-University Computing Centre,
University of Tasmania.

```
procedure partsort (a, m, n); value m, n; integer m, n; array a;
comment sorts the elements a[m] through a[n], m < n, of
array a by determining the rank of each element. The rank
of an element d, say, is that index position such that no element
of lower index has a value greater than d, and no element of
higher index has a value less than d. Once the rank of d is
established and d is placed in its ranking position it partitions
the set into three subsets, itself and two others on either side
each of which may be similarly treated in turn. Choice of d
is arbitrary but affects the efficiency of the algorithm according
to the initial ordering of the unsorted elements. This procedure
chooses the first element of each subset and indicates how, by
a trivial change, the approximately centre element may be
chosen. Other implementations choose the last element or
some random element.
  The arrays f, g are stacks, with stack pointer k (in the inner
block). The lower and upper bounds of subsets as yet unsorted
are stored in f and g respectively. The bounds of f and g are
computed in the outer block;

begin integer size, i, k;
  size := n − m + 1;
  if size ≥ 2 then
  begin comment compute size of address stacks f, g;
    k := 0;
    for i := 1, i + i while i < size do k := k + 1;
    begin integer j, p; real d, aj, ai;
      integer array f, g[1 : k];
      k := 1;
```

```
    comment deal with subsets of order 2 separately;
  loop: if size = 2 then
    begin ai := a[m]; aj := a[n];
      if ai > aj then
      begin a[m] := aj;
        a[n] := ai
      end;
      comment extract the bounds of the next subset;
    next: k := k - 1; if k = 0 then goto exit;
      m := f[k]; n := g[k]
    end
    else
    begin i := m; j := n;
      comment choose the first element as d and determine
      its rank. To select the approximately centre element
      as d replace the next statement by the statements:—
      p := (i + j) ÷ 2   d := a[p]   a[p] :=a[i];
      d := a[i];
    L: for aj := a[j] while i ≠ j do
      begin comment j indexes a high to low scan;
        if aj < d then
        begin a[i] := aj; i := i + 1;
          for ai := a[i] while i ≠ j do
          begin comment i indexes a low to high scan;
            if ai > d then
            begin a[j] := ai; j := j - 1;
              goto L
            end;
            i := i + 1
          end;
          goto partition
        end;
        j := j - 1
      end;
      comment i is the rank of d and a[i] is vacant so;
    partition: a[i] := d;
      j := i - m; p := n - i;
      comment choose the smaller subset for treatment,
      store the bounds of the larger subset unless the
      smaller subset is of order one in which case deal with
      the larger subset immediately;
      if j < p then
      begin if j > 1 then
        begin f[k] := i + 1; g[k] := n;
          n := i - 1; k := k + 1
        end
        else
        m := i + 1
      end
      else
      begin if p > 1 then
        begin f[k] := m; g[k] := i - 1;
          m := i + 1; k := k + 1
        end
        else
        n := i - 1
      end
    end;
    size := n - m + 1;
    goto if size < 2 then
    next
    else
    loop;
  exit:
    end
  end
end partsort
```

Algorithm 26

ORDER THE SUBSCRIPTS OF AN ARRAY SECTION ACCORDING TO THE MAGNITUDES OF THE ELEMENTS

J. Boothroyd,
Hydro-University Computing Centre,
University of Tasmania.

```
procedure keysort (a, r, m, n); value m, n; integer m, n; array a;
integer array r;
comment effects a re-ordering of the integers m through n in
r[m] through r[n] so that a[r[m]] ⩽ a[r[m+1]] ⩽ ... ⩽
a[r[n]], i.e. the elements of r[m : n] are re-ordered to indicate
an ordering by magnitude of the elements in a[m : n]. The
bounds of a and r may, of course, extend beyond m and n on
either side. This procedure is essentially the same as pro-
cedure partsort (Algorithm 25) in which indirect addressing is
used to effect a re-ordering of the ranking index vector r rather
than a re-ordering of a itself. This procedure is useful in cases
where several arrays a, b, c ... are to be sorted on the magnitude
of elements in one of these, the key array. The resulting rank
index vector may be used subsequently by procedure permvector
(Algorithm 27) to re-order all these arrays if necessary.
Other uses of r for indirect addressing purposes are obvious;

begin integer size, i, k;
  size := n - m + 1;
  if size ⩾ 2 then
  begin comment compute size of address arrays;
    k := 0;
    for i := 1, i + i while i < size do k := k + 1;
    begin integer j, p, ri, rj, rm, rn; real d;
      integer array f, g[1 : k];
      comment initialize rank index vector;
      for i := m step 1 until n do r[i] := i;
      k := 1;
      comment deal with subsets of order 2 separately;
    loop: if size = 2 then
      begin rm := r[m]; rn := r[n];
        if a[rm] > a[rn] then
        begin r[m] := rn;
          r[n] := rm
        end;
        comment extract the bounds of the next subset;
      next: k := k - 1; if k = 0 then goto exit;
        m := f[k]; n := g[k]
      end
      else
      begin i := m; j := n;
        comment choose the first element as d and determine
        its rank. To select the approximately centre element
        as d replace the next statement by the statements:—
        p := (i + j) ÷ 2   rm := r[p]   r[p] := r[m];
        rm := r[m]; d := a[rm];
      L: for rj := r[j] while i ≠ j do
        begin comment j indexes a high to low scan;
          if a[rj] < d then
          begin r[i] := rj; i := i + 1;
            for ri := r[i] while i ≠ j do
            begin comment i indexes a low to high scan;
              if a[ri] > d then
```

```
            begin r[j] := ri; j := j - 1;
              goto L
            end;
            i := i + 1
          end;
          goto partition
        end;
        j := j - 1
      end;
      comment i is the (indirect addressed) rank of d,
      referenced by rm and r[i] is vacant so;
    partition: r[i] := rm;
      j := i - m; p := n - i;
      comment choose the smaller subset for treatment,
      store the bounds of the larger subset unless the smaller
      subset is of order one in which case deal with the
      larger subset immediately;
      if j < p then
      begin if j > 1 then
        begin f[k] := i + 1; g[k] := n;
          n := i - 1; k := k + 1
        end
        else
        m := i + 1
      end
      else
      begin if p > 1 then
        begin f[k] := m; g[k] := i - 1;
          m:= i + 1; k := k + 1
        end
        else
        n := i - 1
      end
    end;
    size := n - m + 1;
    goto if size < 2 then
    next
    else
    loop;
  exit:
  end
 end
end keysort
```

Algorithm 27

REARRANGE THE ELEMENTS OF AN ARRAY SECTION ACCORDING TO A PERMUTATION OF THE SUBSCRIPTS

J. Boothroyd,
Hydro-University Computing Centre,
University of Tasmania.

```
procedure permvector (a, r, m, n); value m, n;
integer m, n; array a; integer array r;
comment rearranges the elements of the sector a[m] through
a[n], m < n, of array a so that a[i] := a[r[i]],
i = m, m + 1, m + 2, . . ., n. The index vector r is intact on
exit;
begin integer i, k, m1; real w;
  m1 := m + 1;
  for i := n step - 1 until m1 do
  begin k := r[i];
    L: if k ≠ i then
      begin if k > i then
        begin k := r[k];
          goto L
        end;
        w := a[i]; a[i] := a[k]; a[k] := w
      end
    end
end permvector
```

Authors' Note on Algorithms 28, 29, 30.

Combinatorial problems involving permutations not unreasonably take a long time ($10! \simeq 3\cdot6_{10}6$, $20! \simeq 2\cdot4_{10}18$). It is essential therefore that procedures for generating all permutations of n marks should be as efficient as possible.

The efficiency of an ALGOL procedure depends on the method and its implementation. Three procedures are given below which implement known methods in new ways, with considerably improved performance.

Algorithm 28, *NEXTPERM*, generates distinct permutations in lexicographic order and uses the same method as that of Mok–Kong Shen (1963).

Algorithm 29, *vectorperm*, generates permutations in non-lexicographic order, is suitable for $n > 1$ and implements a method described by Mark B. Wells (1961). This is an inherently efficient process which, by the nature of the sequence of transpositions used, is particularly adapted to efficient implementation as shown in Algorithm 30, suitable only for $n \geqslant 5$. A further 14% improvement may be had by implementing Algorithm 30 as a parameterless procedure and by making extensive use of global variables and letting the control program handle any necessary initializations.

The techniques of Algorithm 30 are also applicable to *NEXTPERM* and result in a 16% reduction in running time. These changes are however left as an exercise and challenge to the interested user.

Algorithms 29 and 30 are equivalent procedures for $n \geqslant 5$, have been given the same identifier and identical parameter lists. Each has run under the control of the same driver program with identical results.

Running times, in seconds on an ELLIOTT 503, are given below for each of the following procedures:—

(*a*) Algorithm 30, below
(*b*) Algorithm 28, below
(*c*) Algorithm 29, below
(*d*) ACM202 (Mok-Kong Shen, 1963)
(*e*) ACM86 (Peck and Schrack, 1962)

	$n=6$	$n=7$	$n=8$
(*a*)	1·0	6·0	44·2
(*b*)	1·6	10·2	81·0
(*c*)	2·0	12·2	95·4
(*d*)	3·0	21·0	167
(*e*)	3·6	23·0	180

References

SHEN, MOK-KONG (1963). Algorithm 202, Generation of Permutations in Lexicographic Order, *Communications of the Association for Computing Machinery*, Vol. 6, p. 517.

WELLS, MARK B. (1961). Generation of Permutations by Transposition, *Mathematics of Computation*, Vol 15, p. 192.

PECK, J. E. L., and SCHRACK, G. F. (1962). Algorithm 86, Permute, *Communications of the Association for Computing Machinery*, Vol. 5, p. 208.

Algorithm 28

PERMUTATIONS OF THE ELEMENTS OF A VECTOR IN LEXICOGRAPHIC ORDER

J. P. N. Phillips,
Department of Psychology,
University of Hull.

```
Boolean procedure NEXTPERM (PERM, A, B);
value A, B; integer A, B; integer array PERM;
comment NEXTPERM takes as data the integer array seg-
ment PERM[A] to PERM[B]. If A ≥ B, or if PERM[A] to
PERM[B] (not all, or even any, of which need be distinct) are
in non-increasing order, i.e. if there is no next permutation in
lexical order, then NEXTPERM becomes false and the seg-
ment is left unaltered, otherwise PERM[A] to PERM[B] are
rearranged into the next lexical permutation and NEXTPERM
becomes true;
begin integer i, j, k, pi, pj, pk, pb;
  NEXTPERM := true; j := B − 1;
  if j < A then
  begin NEXTPERM := false;
    goto exit
  end;
  pb := PERM[B]; pj := PERM[j];
  if pj < pb then
  begin PERM[B] := pj;
    PERM[j] := pb
  end
  else
  begin i := B − 2;
    if i < A then
    begin NEXTPERM := false;
      goto exit
    end;
    pi := PERM[i];
    if pi < pj then
    begin if pb > pi then
      begin PERM[i] := pb;
        PERM[j] := pi; PERM[B] := pj
      end
      else
      begin PERM[i] := pj;
        PERM[j] := pb; PERM[B] := pi
      end
    end
    else
    begin for j := B − 3 step − 1 until A do
      begin pj := PERM[j];
        if pj < pi then goto swap;
        i := j; pi := pj
      end;
      NEXTPERM := false
    end;
    goto exit;
  swap: k := B;
    for pk := PERM[k] while pk ≤ pj do k: = k − 1;
    PERM[k] := pj; PERM[j] := pk;
    k := (B + j) ÷ 2; j := B;
    for i := i step 1 until k do
    begin pi := PERM[i]; PERM[i] := PERM[j];
      PERM[j] := pi; j := j − 1
    end
  end;
exit:
end NEXTPERM
```

Acknowledgement: Thanks are due to the referee for useful comment and helpful suggestions.

Algorithm 29

PERMUTATION OF THE ELEMENTS OF A VECTOR

J. Boothroyd,
Hydro-University Computing Centre,
University of Tasmania.

```
procedure vectorperm (m, d, n, mode, endperm); value n, mode;
integer n, mode; array m; integer array d; label endperm;
comment generates, at each entry, one new permutation of the
n marks m[1], . . ., m[n] in m[1 : n]. The permutation is con-
trolled by a variable radix counter d[2 : n] with digit positions
d[2], d[3], . . . , d[n] in which the subscript value denotes the
radix. Starting with d = (0, 0, . . ., 0) one is added to the
counter at each entry to the procedure. One and only one
digit position increases in value and all digit positions below
this are reset to zero. Denoting by k that digit position which
increases the transposition rules are:—
    (k odd) or (k even and d[k] ≤ 2) exchange m[k], m[k −1]
    k even and 2 < d[k] < k   exchange m[k], m[k − d[k]].
A call of vectorperm with mode = 1 initializes the counter
preparatory to further calls with mode = 2. After n factorial
permutations have been generated d resets to zero and the
procedure exits to endperm.
  The essential algorithm is that of Mark B. Wells (1961)
though the transposition rules given above are much simplified
compared with those in (Mark B. Wells, 1961);
begin integer k, j, kless1, dk; real temp;
  switch s := set, run;
  goto s[mode];
set: for k := 2 step 1 until n do d[k] := 0; goto exit;
run: j := − 1; kless1 := 1;
  for k := 2 step 1 until n do
  begin dk := d[k];
    if dk ≠ kless1 then goto swap;
    d[k] :=0; j := − j;
    kless1 := k
  end;
  goto endperm;
swap: dk := d[k] := dk + 1;
  if j ≠ 1 ∧ dk > 2 then kless1 := k − dk;
  temp := m[k]; m[k] := m[kless1]; m[kless1] := temp;
exit:
end vectorperm
```

Algorithm 30

FAST PERMUTATION OF THE ELEMENTS OF A VECTOR

J. Boothroyd,
Hydro-University Computing Centre,
University of Tasmania.

```
procedure vectorperm (m, d, n, mode, endperm); value n, mode;
integer n, mode; integer array d; array m; label endperm;
comment a highly efficient implementation of Algorithm 29,
suitable only for n ≥ 5. The improvement in efficiency results
from capitalizing on the fact that 23 successive entries to the
```

procedure affect two elements in the subset $m[1], \ldots, m[4]$, *and array access is minimized by using, on one entry, elements accessed in the immediately preceding entry. The parameters are the same as those of Algorithm 29 though the bounds of d may be changed to* $d[5:n]$;

```
begin integer j, k, kless1, dk; real mk;
  own real m1, m2, m3, m4; own integer i;
  switch s := s1, s2, s1, s2, s1, s3, s1, s2, s1, s2, s1, s3,
    s1, s2, s1, s2, s1, s4, s1, s2, s1, s2, s1, s5, set, run;
  switch ss := ss1, ss2, ss3, ss4;
  goto s[24 + mode];
set: for k := 5 step 1 until n do d[k] := 0;
  m1 := m[1]; m2 := m[2]; m3 := m[3]; m4 := m[4];
  i := 0; goto exit;
run: i :=i + 1; goto s[i];
s1: mk :=m1; m1 := m[1] := m2; m2 := m[2] := mk;
  goto exit;
s2: mk := m2; m2 := m[2] := m3; m3 := m[3] := mk;
  goto exit;
s3: mk := m3; m3 := m[3] := m4; m4 := m[4] := mk;
  goto exit;
s4: mk := m4; m4 := m[4] := m1; m1 := m[1] := mk;
  goto exit;
s5: j := 1; kless1 := 4; i := 0;
  for k := 5 step 1 until n do
  begin dk := d[k];
    if dk ≠ kless1 then goto swap;
    kless1 := k; d[k] := 0; j := −j
  end;
  goto endperm;
swap: dk := d[k] := dk + 1;
  if j ≠ 1 ∧ dk > 2 then kless1 := k − dk;
  mk := m[k]; m[k] := m[kless1]; m[kless1] := mk;
  goto if kless1 ⩽ 4 then ss[kless1] else exit;
ss1: m1 := mk; goto exit;
ss2: m2 := mk; goto exit;
ss3: m3 := mk; goto exit;
ss4: m4 := mk;
exit:
end vectorperm
```

Contributions to the Algorithms Supplement should be sent to

P. Hammersley
University Mathematical Laboratory
Corn Exchange Street
Cambridge

Published Quarterly by

The British Computer Society, 23 Dorset Square, LONDON, N.W.1, England.

The Computer Journal is registered at Stationers' Hall, London (certificate No. 24690, June 1966).

Subscription price per volume £4 10s. (U.S. $12.60). Single Copies 25s. (U.S. $3.50)

All inquiries should be sent to the Secretary at the above address.

Communications: Papers submitted for publication should be sent to E. N. Mutch, The University Mathematical Laboratory, Corn Exchange Street, Cambridge. Intending authors should first apply for *Notes on the Submission of Papers*, as the onus of preparing papers in a form suitable for sending to press lies in the first place with the authors.