## ALGORITHM 230
## MATRIX PERMUTATION

J. BOOTHROYD   (Recd 18 Nov. 1963)

English Electric-Leo Computers, Kidsgrove, Stoke-on-Trent, England

**procedure**   $matrixperm(a,b,j,k,s,d,n,p)$;   **value** $n$;   **real** $a,b$;   **integer array** $s,d$;   **integer** $j,k,n,p$;

**comment**   a procedure using Jensen's device which exchanges rows or columns of a matrix to achieve a rearrangement specified by the permutation vectors $s,d[1:n]$. Elements of $s$ specify the original source locations while elements of $d$ specify the desired destination locations. Normally $a$ and $b$ will be called as subscripted variables of the same array. The parameters $j,k$ nominate the subscripts of the dimension affected by the permutation, $p$ is the Jensen parameter. As an example of the use of this procedure, suppose $r,c[1:n]$ to contain the row and column subscripts of the successive matrix pivots used in a matrix inversion of an array $a[1:n,1:n]$; i.e. $r[1]$, $c[1]$ are the relative subscripts of the first pivot $r[2]$, $c[2]$ those of the second pivot and so on. The two calls

$$matrixperm\ (a[j,p],\ a[k,p],\ j,k,r,c,n,p)$$
$$\text{and } matrixperm\ (a[p,j],\ a[p,k],\ j,k,c,r,n,p)$$

will perform the required rearrangement of rows and columns respectively;

**begin integer array** $tag$, $loc[1:n]$;   **integer** $i,t$;   **real** $w$;

**comment**   set up initial vector tag number and address arrays;

**for** $i := 1$ **step** 1 **until** $n$ **do** $tag[i] := loc[i] := i$;

**comment**   start permutation;

**for** $i := 1$ **step** 1 **until** $n$ **do**
   **begin** $t := s[i]$;   $j := loc[t]$;   $k := d[i]$;
    **if** $j \neq k$ **then begin for** $p := 1$ **step** 1 **until** $n$ **do**
          **begin** $w := a$;   $a := b$;   $b := w$ **end**;
          $tag[j] := tag[k]$;   $tag[k] := t$;
          $loc[t] := loc[tag[j]]$;   $loc[tag[j]] := j$
        **end** $jk$ conditional
   **end** $i$ loop
**end** $matrixperm$

## ALGORITHM 231
## MATRIX INVERSION

J. BOOTHROYD   (Recd 18 Nov. 1963)

English Electric-Leo Computers, Kidsgrove, Stoke-on-Trent, England

**procedure**   $matrixinvert\ (a,n,eps,singular)$;   **value** $n,eps$;   **array** $a$;   **integer** $n$;   **real** $eps$;   **label** $singular$;

**comment**   inverts a matrix in its own space using the Gauss-Jordan method with complete matrix pivoting. I.e., at each stage the pivot has the largest absolute value of any element in the remaining matrix. The coordinates of the successive matrix pivots used at each stage of the reduction are recorded in the successive element positions of the row and column index vectors $r$ and $c$. These are later called upon by the procedure $matrixperm$ which rearranges the rows and columns of the

matrix. If the matrix is singular the procedure exits to an appropriate label in the main program;

**begin   integer** $i,j,k,l,pivi,pivj,p$;   **real** $pivot$;   **integer array** $r,c[1:n]$;

**comment**   set row and column index vectors;

**for** $i := 1$ **step** 1 **until** $n$ **do** $r[i] := c[i] := i$;

**comment**   find initial pivot;   $pivi := pivj := 1$;

**for** $i := 1$ **step** 1 **until** $n$ **do for** $j := 1$ **step** 1 **until** $n$ **do**
   **if** $abs\ (a[i,j]) > abs\ (a[pivi,pivj])$ **then begin** $pivi := i$;
   $pivj := j$ **end**;

**comment**   start reduction;

**for** $i := 1$ **step** 1 **until** $n$ **do**
**begin** $l := r[i]$;   $r[i] := r[pivi]$;   $r[pivi] := l$;   $l := c[i]$;
   $c[i] := c[pivj]$;   $c[pivj] := l$;
   **if** $eps > abs\ (a[r[i],c[i]])$ **then**
    **begin comment**   here include an appropriate output procedure to record $i$ and the current values of $r[1:n]$ and $c[1:n]$;   **go to** $singular$ **end**;
   **for** $j := n$ **step** $-1$ **until** $i+1$, $i-1$ **step** $-1$ **until** 1 **do** $a[r[i],c[j]]$
    $:= a[r[i],c[j]]/a[r[i],c[i]]$;   $a[r[i],c[i]] := 1/a[r[i],c[i]]$;
   $pivot := 0$;
   **for** $k := 1$ **step** 1 **until** $i-1$, $i+1$ **step** 1 **until** $n$ **do**
   **begin for** $j := n$ **step** $-1$ **until** $i+1$, $i-1$ **step** $-1$ **until** 1 **do**
    **begin** $a[r[k],c[j]] := a[r[k],c[j]] - a[r[i],c[j]] \times a[r[k],c[i]]$;
      **if** $k>i \wedge j>i \wedge abs\ (a[r[k],c[j]]) > abs(pivot)$ **then**
        **begin** $pivi := k$;   $pivj := j$;
          $pivot := a[r[k],c[j]]$ **end** conditional
    **end** $j$loop;
     $a[r[k],c[i]] := -a[r[i],c[i]] \times a[r[k],c[i]]$
   **end** $k$loop
**end**   $i$loop and reduction;

**comment**   rearrange rows; $matrixperm\ (a[j,p],a[k,p],j,k,r,c,n,p)$;

**comment**   rearrange columns;

  $matrixperm\ (a[p,j],a[p,k],j,k,c,r,n,p)$

**end**   $matrixinvert$

[EDITOR'S NOTE. On many compilers $matrixinvert$ would run much faster if the subscripted variables $r[i]$, $c[i]$, $r[k]$ were replaced by simple integer variables $ri$, $ci$, $rk$, respectively, inside the $j$ loop.—G.E.F.]

## ALGORITHM 232
## HEAPSORT

J. W. J. WILLIAMS   (Recd 1 Oct. 1963 and, revised, 15 Feb. 1964)

Elliott Bros. (London) Ltd., Borehamwood, Herts, England

**comment**   The following procedures are related to $TREESORT$ [R. W. Floyd, Alg. 113, *Comm. ACM* 5 (Aug. 1962), 434, and A. F. Kaupe, Jr., Alg. 143 and 144, *Comm. ACM* 5 (Dec. 1962), 604] but avoid the use of pointers and so preserve storage space. All the procedures operate on single word items, stored as elements 1 to $n$ of the array $A$. The elements are normally so arranged that $A[i] \leq A[j]$ for $2 \leq j \leq n$, $i = j \div 2$. Such an arrange-

ment will be called a heap. $A[1]$ is always the least element of the heap.

The procedure *SETHEAP* arranges $n$ elements as a heap, *INHEAP* adds a new element to an existing heap, *OUTHEAP* extracts the least element from a heap, and *SWOPHEAP* is effectively the result of *INHEAP* followed by *OUTHEAP*. In all cases the array $A$ contains elements arranged as a heap on exit.

*SWOPHEAP* is essentially the same as the tournament sort described by K. E. Iverson—*A Programming Language*, 1962, pp. 223-226—which is a top to bottom method, but it uses an improved storage allocation and initialisation. *INHEAP* resembles *TREESORT* in being a bottom to top method. *HEAPSORT* can thus be considered as a marriage of these two methods.

The procedures may be used for replacement-selection sorting, for sorting the elements of an array, or for choosing the current minimum of any set of items to which new items are added from time to time. The procedures are the more useful because the active elements of the array are maintained densely packed, as elements $A[1]$ to $A[n]$;

**procedure** *SWOPHEAP* $(A,n,in,out)$;
  **value** $in,n$;  **integer** $n$;  **real** $in,out$;  **real array** $A$;
  **comment** *SWOPHEAP* is given an array $A$, elements $A[1]$ to $A[n]$ forming a heap, $n \geq 0$. *SWOPHEAP* effectively adds the element $in$ to the heap, extracts and assigns to $out$ the value of the least member of the resulting set, and leaves the remaining elements in a heap of the original size. In this process elements 1 to $(n+1)$ of the array $A$ may be disturbed. The maximum number of repetitions of the cycle labeled *scan* is $log_2 n$;
  **begin integer** $i,j$;  **real** $temp, temp\ 1$;
    **if** $in \leq A[1]$ **then** $out := in$ **else**
    **begin** $i := 1$;
      $A[n+1] := in$;  **comment** this last statement is only necessary in case $j = n$ at some stage, or $n = 0$;
      $out := A[1]$;
    $scan$:  $j := i+i$;
      **if** $j \leq n$ **then**
      **begin** $temp := A[j]$;
        $temp\ 1 := A[j+1]$;
        **if** $temp\ 1 < temp$ **then**
        **begin** $temp := temp\ 1$;
          $j := j+1$
        **end**;
        **if** $temp < in$ **then**
        **begin** $A[i] := temp$;
          $i := j$;
          **go to** $scan$
        **end**
      **end**;
      $A[i] := in$
    **end**
  **end** *SWOPHEAP*;
**procedure** *INHEAP* $(A, n, in)$;
  **value** $in$;  **integer** $n$;  **real** $in$;  **real array** $A$;
  **comment** *INHEAP* is given an array $A$, elements $A[1]$ to $A[n]$ forming a heap and $n \geq 0$. *INHEAP* adds the element $in$ to the heap and adjusts $n$ accordingly. The cycle labeled *scan* may be repeated $log_2 n$ times, but on average is repeated twice only;
  **begin integer** $i,j$;
    $i := n := n+1$;
  $scan$: **if** $i > 1$ **then**
    **begin** $j := i \div 2$;
      **if** $in < A[j]$ **then**
      **begin** $A[i] := A[j]$;
        $i := j$;
        **go to** $scan$

$\quad$ **end**
$\quad$ **end**;
$\quad A[i] := in$
**end** *INHEAP*;
**procedure** *OUTHEAP* $(A,n,out)$;
  **integer** $n$;  **real** $out$;  **real array** $A$;
  **comment**  given array $A$, elements 1 to $n$ of which form a heap, $n \geq 1$, *OUTHEAP* assigns to $out$ the value of $A[1]$, the least member of the heap, and rearranges the remaining members as elements 1 to $n-1$ of $A$. Also, $n$ is adjusted accordingly;
  **begin** *SWOPHEAP* $(A,n-1, A[n],out)$;
    $n := n-1$
  **end** *OUTHEAP*;
**procedure** *SETHEAP* $(A,n)$;
  **value** $n$;  **integer** $n$;  **real array** $A$;
  **comment**  *SETHEAP* rearranges the elements $A[1]$ to $A[n]$ to form a heap;
  **begin integer** $j$;
    $j := 1$;
  $L$:  *INHEAP* $(A,j,A[j+1])$;
    **if** $j < n$ **then go to** $L$
  **end** *SETHEAP*

## ALGORITHM 233
## SIMPSON'S RULE FOR MULTIPLE
##     INTEGRATION

FRANK OLYNYK*    (Recd 24 Dec. 1963)
Case Institute of Technology, Cleveland, Ohio

**real procedure** *Simps* $(X, x1, x2, delta, f)$;
  **value** $x1, x2, delta$;  **real** $X, x1, x2, delta, f$;
  **comment**  This procedure calculates a single integral by Simpson's rule in such a way that it can be called recursively for the evaluation of an iterated integral. $x1$ and $x2$ are the lower and upper limits, respectively, which may be any mathematically meaningful expressions. Hence in using *Simps* for multiple integration the region is not limited to rectangular boxes. The algorithm terminates when two successive evaluations pass the test involving *delta*. The formal parameter $f$ stands for the expression to be integrated.

As an example of the use of *Simps*,

$$\int_0^1 dx \int_0^{(1-x^2)^{\frac{1}{2}}} g(x, y)\ dy$$

would be evaluated by
*Simps* $(x, 0, 1, delta, Simps(y, 0, sqrt(1 - x \uparrow 2), delta2, g(x, y)))$.
$\quad$ *Simps* has been written and run in ALGOL 60 on the Univac 1107 at Case Institute.

$\quad$ [EDITOR'S NOTE. Experience of W. McKeeman suggests the wisdom of choosing $delta2 < delta$.—G.E.F.];
**begin**
  **Boolean** *turing*;  **real** $z1, z2, z3, h, k$;
  $turing := $ **false**;
  **if** $x1 = x2$ **then begin** $z1 := 0$;  **go to** $box2$ **end**;
  **if** $x1 > x2$ **then begin** $h := x1$;  $x1 := x2$;  $x2 := h$;
    $turing := $ **true end**;
  $X := x1$;  $z1 := f$;  $X := x2$;  $z3 := z1 := z1 + f$;
  $k := x2 - x1$;
$box$:
  $z2 := 0$;  $h := k/2$;
  **for** $X := x1 + h$ **step** $k$ **until** $x2$ **do**   $z2 := z2 + f$;
  $z1 := z1 + 4 \times z2$;
  **if** $h \times abs((z1 - 2 \times z3)/($**if** $z1 = 0$ **then** $1.0$ **else** $z1)) < delta$
    **then go to** $box2$
  **else** $z3 := z1$;
  $z1 := z1 - 2 \times z2$;

```
        k := h;
        go to box;
box2:
        if turing then h := -h;
        Simps := h × z1/3
end Simps
```

## CERTIFICATION OF ALGORITHM 40
## CRITICAL PATH SCHEDULING [B. Leavenworth, Comm. ACM 4 (Mar. 1961), 152; 4 (Sep. 1961), 392; 5 (Oct. 1962), 513]

IRVIN A. HOFFMAN   (Recd 7 Feb. 1964)
Woodward Governor Co., Rockford, Ill.

The Critical Path Scheduling algorithm was coded in FAST for the NCR315. The modifications suggested by Alexander [Comm. ACM 4 (Sept. 1961)] were included. Results were correct in all tested cases. However, the example of the $I$, $J$ vectors given in the comment is incorrect, as it would cause the exit $out3 - I_k$ missing.

[EDITOR'S NOTE.   There are also two semicolons which should be removed from the comment of Algorithm 40.—G.E.F.]

## CERTIFICATION OF ALGORITHM 201
## SHELLSORT [J. BOOTHROYD, Comm. ACM 6 (Aug. 1963), 445]

M. A. BATTY   (Recd 27 Jan. 1964)
English Electric Co., Whetstone, Nr. Leicester, England

This algorithm has been tested successfully using the DEUCE ALGOL Compiler. When the first statement of the algorithm was replaced by the statement
$$m := n;$$
to implement Shell's original choice of $m_1 := n/2$, a slight increase in sorting time was observed with most of the cases tested.

## REMARK ON ALGORITHM 214
## q-BESSEL FUNCTIONS $I_n(t)$ [J. M. S. Simões Pereira, Comm. ACM 6 (Nov. 1963), 662]

J. M. S. SIMÕES PEREIRA   (Recd 6 Jan 1964)
Gulbenkian Scientific Computing Center, Lisbon, Portugal

Corrections:
1.  Insert a dummy statement labeled $C$ just before the final end.
2.  Add a statement go to $C$ just before the label $B$.
3.  Add a colon in the clause for $k := 1$ step 1 until $j$ do . . .

## CERTIFICATION OF AND REMARK ON ALGORITHM 220
## GAUSS-SEIDEL [P. W. Shantz, Comm. ACM 6 (Dec. 1963), 739]

A. P. BATSON   (Recd 6 Jan. 1964)
University of Virginia, Charlottesville, Va.
NIKLAUS WIRTH   (Recd 6 Jan. 1964)
Computer Science Div., Stanford U., Stanford, Calif.

[EDITOR'S NOTE.   Two substantially equivalent contributions were received on the same day, and so the editor has merged them.—G.E.F.]

The following errors were detected.
1. The procedure cannot communicate the solution to the outside block unless $X$ (or $Y$) is made a parameter of the procedure.

2. The identifier GAUSS-SEIDEL may not contain a hyphen.
3. In the fourth line after the label START change $y[i]$ to $Y[i]$.

With the above errors corrected, GAUSS SEIDEL was successfully run on the Stanford 7090 computer in Wirth's Extended ALGOL, and on the Virginia ALGOL compiler for the Burroughs 205.

The following improvements would be desirable.
1. Avoid repeated reference to the subscripted variable $Y[i]$ inside the $j$ loop.
2. Permit the user to initialize the array $X$ to an appropriate value at the start of the iteration.
3. Modify tol to be a relative error, rather than an absolute error.
4. Incorporate a guard against nonconvergence.