

# Algorithms

J. G. HERRIOT, Editor

## ALGORITHM 268

### ALGOL 60 REFERENCE LANGUAGE EDITOR [R2]

W. M. McKEEMAN\* (Recd. 9 Dec. 1964, 23 Feb. 1965 and 17 May 1965)

Computer Science Department, Stanford University, Stanford, California

\*Supported in part by the Office of Naval Research under Contract Nonr 225(37), NR 044-211.

The author expresses his thanks to the referee for several valuable suggestions.

```

procedure Algoedit(characterset, linelimit);
string characterset;
integer linelimit;
comment If this procedure is presented an ALGOL 60 program
or procedure in the form of a sequence of basic symbols, it will
transmit to the output medium a copy of the text with indentations
between each begin-end pair and some rearrangement of the
blank spaces within the text. This procedure is an example of its
own output. It is used to edit ALGOL 60 text that is difficult to read
because, for example, the ALGOL has been transcribed from
printed documents, or written by inexperienced programmers, or
stored in compressed form (i.e., with all redundant blank spaces
removed). The integer "—1" will represent the nonbasic symbol
"carriage return", "—2" will represent an end-of-file mark, other
symbols will have the integer value corresponding to their position
in the parametric string "characterset". The string must contain
exactly the 116 basic symbols of ALGOL 60. The parameter "line-
limit" sets an upper bound on the number of basic symbols that
the user wishes to appear on a line of output. The identifiers
"lsq" and "rsq" will be used in place of strings of length one whose
only elements are " " and " ", respectively;
begin integer array spacesbefore, spacesafter[1 : 116],
    buffer[1 : linelimit];
    integer tabstop, symbol, i, symbolcount, level;
    Boolean newline;
    integer procedure val(s);
    string s;
    comment The value of this procedure is the integer
    corresponding to the position in the string "characterset"
    of the symbol in the string "s". The body of the
    procedure must be expressed in code;
    procedure get(symbol);
    integer symbol;
    begin insymbol(2, characterset, symbol);
        if symbol = — 2 then go to eof
    end get;
    procedure send(symbol);
    integer symbol;
    begin comment "send" must not break identifiers
        across lines or insert spurious characters into
        strings;
        integer i, u, v;
        if symbol = — 1  $\vee$  symbolcount  $\geq$  linelimit
        then
            begin v := tabstop;
                if newline then go to E;

```

```

if level  $\neq$  0 then
    begin comment Inside a string;
        for i := 1 step 1 until
            symbolcount do outsymbol(1,
                characterset, buffer[i]);
        outsymbol(1, characterset, — 1);
        v := 0
    end else
        begin u := symbolcount;
            newline := true;
            if symbol = — 1 then go to D;
            comment Find a convenient place to
            break the line;
            for u := symbolcount — 1 step —
                1 until 1 do if buffer[u + 1] =
                    val('u')  $\vee$  buffer[u] = val(rsq) then
                        go to D;
            u := symbolcount;
            comment Send the line;
            D : for i := 1 step 1 until u do
                outsymbol(1, characterset, buffer[i]);
                outsymbol(1, characterset, — 1);
            comment Find a non-blank character
            to start the next line;
            for i := u + 1 step 1 until
                symbolcount do if buffer[i]  $\neq$  val('u')
                    then go to F;
            go to G;
            comment Move a new line to the
            head of the buffer area;
            F : for i := i step 1 until
                symbolcount do
                    begin v := v + 1;
                        newline := false;
                        buffer[v] := buffer[i]
                    end;
            comment Insert blanks for tab stops;
            G : for i := 1 step 1 until
                tabstop do buffer[i] := val('u')
        end;
        E : symbolcount := v
    end;
comment Now we can put the new symbol in the
buffer array;
if symbol  $\neq$  — 1  $\wedge$   $\neg$  (newline  $\wedge$  symbol
= val('u')) then
    begin symbolcount := symbolcount + 1;
        newline := false;
        buffer[symbolcount] := symbol
    end
end send;
for symbol := 1 step 1 until 116 do
    spacesbefore[symbol] := spacesafter[symbol] := 0;
    for symbol := val('+'), val('—'), val('¬'), val(':'),
        val('='), val('<'), val('≤'), val('='), val('≠'),
        val('≥'), val('>') do spacesbefore[symbol] :=
        spacesafter[symbol] := 1;

```

```

for symbol := val('∧'), val('∨'), val('⊃'), val('≡'),
val('then'), val('else'), val('step'), val('until'),
val('while'), val('do') do spacesbefore[symbol] :=
spacesafter[symbol] := 2;
for symbol := val('go to'), val('begin'), val('if'),
val('for'), val('procedure'), val('value'), val('own'),
val('real'), val('Boolean'), val('integer'), val('array'),
val('switch'), val('label'), val('string'), val(',') do
spacesafter[symbol] := 2;
level := symbolcount := tabstop := 0;
newline := true;
nextsymbol := deblank : get(symbol);
scanned : if symbol = val('␣') ∨ symbol = - 1
then go to deblank;
if symbol = val('begin') then send( - 1) else
if symbol = val('end') then
begin tabstop := tabstop - 5;
send( - 1)
end;
for i := 1 step 1 until spacesbefore[symbol] do
send(val('␣'));
send(symbol);
for i := 1 step 1 until spacesafter[symbol] do
send(val('␣'));
if symbol = val('comment') then
begin comment Pass comments on unchanged;
for i := 1 while symbol ≠ val(',') do
begin get(symbol);
send(symbol)
end
end else if symbol = val('end') then
begin comment "end" comments;
for i := 1 while symbol ≠ val(',') do
begin get(symbol);
if symbol = val('else') ∨ symbol =
val('end') then go to scanned;
send(symbol)
end
end else if symbol = val('lsq') then
begin comment Pass strings on unchanged;
level := 1;
for i := 1 while level ≠ 0 do
begin get(symbol);
send(symbol);
if symbol = val('lsq') then level := level
+ 1 else if symbol = val('rsq')
then level := level - 1
end
end;
if symbol = val('begin') then tabstop := tabstop + 5
else if symbol = val(',') then send( - 1);
go to nextsymbol;
eof : send( - 1);
outsymbol(1, character set, - 2)
end Algotedit

```

## ALGORITHM 269

### DETERMINANT EVALUATION [F3]

JAROSLAV PFANN AND JOSEF STRAKA

(Recd. 10 Sept. 1964 and 29 Dec. 1964)

Institute of Nuclear Research, Řež by Prague, Czechoslovakia

**real procedure determinant** ( $A$ ,  $n$ ); **array**  $A$ ; **integer**  $n$ ;  
**comment** This procedure evaluates a determinant by triangularization with searching for pivot in row and with scaling of

the rows of the matrix before the triangularization. This was done as in procedure *EQUILIBRATE* of the Algorithm 135 [Comm. ACM 5 (Nov. 1962), 553];

```

begin real product, temp; integer i, j, r, s;
array mult[1:n];
procedure EQUILIBRATE( $A$ ,  $n$ , mult);
integer  $n$ ; array  $A$ , mult;
begin integer  $i, j$ ; real  $mx$ ;
for  $i := 1$  step 1 until  $n$  do
begin  $mx := 0.0$ ;
for  $j := 1$  step 1 until  $n$  do
if  $abs(A[i, j]) > mx$  then  $mx := abs(A[i, j])$ ;
if  $mx = 0.0$  then
begin determinant := 0; go to RETURN end;
mult[i] :=  $mx$ ; comment := base  $\uparrow$   $ex$  for exact scaling;
if  $mx \neq 1.0$  then
for  $j := 1$  step 1 until  $n$  do  $A[i, j] := A[i, j]/mx$ ;
end
end EQUILIBRATE;
EQUILIBRATE( $A$ ,  $n$ , mult);
product := 1;
for  $r := 1$  step 1 until  $n-1$  do
begin  $s := r$ ; temp :=  $abs(A[r, r])$ ;
for  $j := r+1$  step 1 until  $n$  do
if temp <  $abs(A[r, j])$  then
begin temp :=  $abs(A[r, j])$ ;  $s := j$  end;
if temp = 0 then begin determinant := 0; go to RETURN
end;
if  $s \neq r$  then
begin product := - product;
for  $i := r$  step 1 until  $n$  do
begin temp :=  $A[i, r]$ ;  $A[i, r] := A[i, s]$ ;
 $A[i, s] := temp$ 
end
end;
product := product  $\times A[r, r]$ ;
comment Be on guard against overflow or underflow here;
for  $i := r+1$  step 1 until  $n$  do
begin temp :=  $A[i, r]/A[r, r]$ ;
for  $j := r+1$  step 1 until  $n$  do
 $A[i, j] := A[i, j] - A[r, j] \times temp$ 
end
end;
temp := product  $\times A[n, n]$ ;
for  $r := 1$  step 1 until  $n$  do temp := temp  $\times$  mult[r];
comment Again danger of overflow or underflow;
determinant := temp;
RETURN;
end determinant

```

**REFERENCE:**

McKEEMAN, W. M. Algorithm 135—Crout with equilibration and iteration. *Comm. ACM* 5 (Nov. 1962), 553.

## ALGORITHM 270

### FINDING EIGENVECTORS BY GAUSSIAN ELIMINATION [F2]

ALBERT NEWHOUSE (Recd. 3 May 1965 and 16 July 1965)  
University of Houston, Houston, Texas

**procedure NULLSPACE** ( $n$ ,  $a$ ,  $ec$ ,  $eps$ ); **value**  $n$ ,  $eps$ ; **integer**  $n$ ,  $ec$ ; **real**  $eps$ ; **array**  $a$ ;

**comment** *NULLSPACE* computes the vectors  $x$  of order  $n$  such that  $xa = z$ , where  $a$  is an  $n \times n$  matrix,  $z$  is the zero-vector of order  $n$ ,  $eps$  is a small positive number such that if the maximum pivot element is numerically less than  $eps$  the procedure considers it zero. The  $ec$  vectors  $x$  are to be found in the first  $ec$  rows of the matrix  $a$  upon exit from this procedure;

**comment** In finding the eigenvectors  $x$  of an  $n \times n$  matrix  $B$  after having found the eigenvalues  $\lambda$  of  $B$  by any of the many available methods, it is often desirable to start from the original matrix  $B$  and not from its transform from which the  $\lambda$ 's were obtained. Whereas the resulting eigenvectors will still be influenced by errors in the  $\lambda$ 's, the eigenvectors would not be influenced by errors in the transformed matrix.

Since  $M - B = A$  is a singular matrix of rank  $r$  the problem is to find  $ec = n - r$  vectors  $x$  which form a basis of the left null space of  $A$ .

Note: If the right null space is desired the matrix  $A$  should be transposed.

The following algorithm finds these  $n - r$  linearly independent vectors by the Gauss-Jordan elimination in place using the maximal available element for the pivot. The process will terminate after  $r$  steps, since the maximal available elements for pivoting are then equal to zero.

Now, replacing these zero pivot elements by unity, the rows of the matrix, from which no nonzero element has been chosen, are the basis of the null space of  $A$ , that is, if  $x$  is such a row then  $xA = z$ , the zero vector of order  $n$ .

The proof for this is established by the fact that the elimination amounts to premultiplying  $B$  by a matrix  $A'$ , a product of elementary matrices, such that  $A'A$  is a matrix with ones on  $r$  of the diagonal positions and zeros everywhere else.

Test results. A version of this procedure acceptable to the IBM 7094 (ALCOR-ILLINOIS 7090 ALGOL Compiler) was tested.

With  $eps = 10^{-6}$  the results for the  $5 \times 5$  matrix

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

showed the dimension of the null space as 3 having as a basis

$$x_1 = (-.75, 1.00, 0.00, 0.00, -.25)$$

$$x_2 = (-.50, 0.00, 1.00, 0.00, -.50)$$

$$x_3 = (-.25, 0.00, 0.00, 1.00, -.75)$$

exact to 6 decimal places;

```

begin integer array r, c[1:n]; integer i, j, k, m, jj, kk, t;
real max, temp;
for i := 1 step 1 until n do r[i] := c[i] := 0;
for m := 1 step 1 until n do
  begin max := 0;
    for k := 1 step 1 until n do
      begin if r[k]  $\neq$  0 then go to L else
        for j := 1 step 1 until n do
          if c[j] = 0  $\wedge$  abs(a[k, j]) > max then
            begin kk := k; jj := j; max := abs(a[k, j])
          end j loop;
        end k loop;
      if max < eps then go to SORT;
      c[jj] := kk; r[kk] := jj; temp := 1/a[kk, jj]; a[kk, jj] := 1;
      for j := 1 step 1 until n do a[kk, j] := a[kk, j]  $\times$  temp;
      for k := 1 step 1 until kk - 1, kk + 1 step 1 until n do
        begin temp := a[k, jj]; a[k, jj] := 0;
          for j := 1 step 1 until n do
            begin
              a[k, j] := a[k, j] - temp  $\times$  a[kk, j];
              if abs(a[k, j]) < eps then a[k, j] := 0
            end;
          end k loop;
        end m loop;
    SORT: for j := 1 step 1 until n do
      begin
        REPEAT: if c[j]  $\neq$  0  $\wedge$  j  $\neq$  c[j] then

```

**begin**

**for** k := 1 **step** 1 **until** n **do**

**if** r[k] = 0 **then**

**begin** temp := a[k, j];

a[k, j] := a[k, c[j]]; a[k, c[j]] := temp

**end** k loop;

t := c[j]; c[j] := c[t]; c[t] := t; **go to** REPEAT

**end**;

**end** conditional and j loop;

ec := 0;

**for** k := 1 **step** 1 **until** n **do**

**if** r[k] = 0 **then**

**begin** ec := ec + 1; a[k, k] := 1;

**if** ec  $\neq$  k **then**

**begin**

**for** j := 1 **step** 1 **until** n **do** a[ec, j] := a[k, j]

**end**;

**end** conditional and k loop;

**comment** The first ec rows of the matrix a are the vectors which are orthogonal to the columns of the matrix a;

**end** NULLSPACE

## ALGORITHM 271

### QUICKSORT [M1]

R. S. SCOWEN\* (Recd. 22 Mar. 1965 and 30 June 1965)  
National Physical Laboratory, Teddington, England

\*The work described below was started while the author was at English Electric Co. Ltd, completed as part of the research programme of the National Physical Laboratory and is published by permission of the Director of the Laboratory.

**procedure** quicksort(a, j);

**value** j; **integer** j; **array** a;

**begin** **integer** i, k, q, m, p; **real** t, x; **integer** array ut, lt[1:ln(abs(j)+2)/ln(2)+0.01];

**comment** The procedure sorts the elements of the array a[1:j] into ascending order. It uses a method similar to that of QUICKSORT by C. A. R. Hoare [1], i.e., by continually splitting the array into parts such that all elements of one part are less than all elements of the other, with a third part in the middle consisting of a single element. I am grateful to the referee for pointing out that QUICKSORT also bears a marked resemblance to sorting algorithms proposed by T. N. Hibbard [2, 3]. In particular, the elimination of explicit recursion by choosing the shortest sub-sequence for the secondary sort was introduced by Hibbard in [2].

An element with value  $t$  is chosen arbitrarily (in QUICKSORT the middle element is chosen, in QUICKSORT a random element is chosen).  $i$  and  $j$  give the lower and upper limits of the segment being split. After the split has taken place a value  $q$  will have been found such that  $a[q] = t$  and  $a[I] \leq t \leq a[J]$  for all  $I, J$  such that  $i \leq I < q < J \leq j$ . The program then performs operations on the two segments  $a[i:q-1]$  and  $a[q+1:j]$  as follows. The smaller segment is split and the position of the larger segment is stored in the  $lt$  and  $ut$  arrays ( $lt$  and  $ut$  are mnemonics for lower temporary and upper temporary). If the segment to be split has two or fewer elements it is sorted and another segment obtained from the  $lt$  and  $ut$  arrays. When no more segments remain, the array is completely sorted.

#### REFERENCES:

1. HOARE, C. A. R. Algorithms 63 and 64. *Comm. ACM* 4 (July 1961), 321.
2. HIBBARD, THOMAS N. Some combinatorial properties of certain trees with applications to searching and sorting. *J. ACM* 9 (Jan. 1962), 13.
3. —. An empirical study of minimal storage sorting. *Comm. ACM* 6 (May 1963), 206-213;

```

    i := m := 1;
N: if j-i > 1 then
    begin comment This segment has more than two elements,
        so split it;
        p := (j+i) ÷ 2;
        comment p is the position of an arbitrary element in the
            segment a[i:j]. The best possible value of p would be one
            which splits the segment into two halves of equal size, thus
            if the array (segment) is roughly sorted, the middle ele-
            ment is an excellent choice. If the array is completely
            random the middle element is as good as any other.
            If however the array a[1:j] is such that the parts a[1:j÷2]
            and a[j÷2+1:j] are both sorted the middle element could
            be very bad. Accordingly in some circumstances
            p := (i+j) ÷ 2 should be replaced by p := (i+3×j) ÷ 4
            or p := RANDOM(i, j) as in QUICKSORT;
        t := a[p];
        a[p] := a[i];
        q := j;
        for k := i + 1 step 1 until q do
            begin comment Search for an element a[k] > t starting
                from the beginning of the segment;
                if a[k] > t then
                    begin comment Such an a[k] has been found;
                        for q := q step -1 until k do
                            begin comment Now search for a[q] < t starting from
                                the end of the segment;
                                if a[q] < t then
                                    begin comment a[q] has been found, so exchange
                                        a[q] and a[k];
                                        x := a[k];
                                        a[k] := a[q];
                                        a[q] := x;
                                        q := q-1;
                                        comment Search for another pair to exchange;
                                        go to L
                                    end
                                end for q;
                                q := k - 1;
                                comment q was undefined according to Para. 4.6.5 of
                                    the Revised ALGOL 60 Report [Comm. ACM 6 (Jan.
                                    1963), 1-17];
                                go to M
                            end;
                        end for k;
                        comment We reach the label M when the search going up-
                            wards meets the search coming down;
M: a[i] := a[q];
    a[q] := t;
    comment The segment has been split into the three parts
        (the middle part has only one element), now store the
        position of the largest segment in the lt and ut arrays and
        reset i and j to give the position of the next largest segment;
    if 2 × q > i + j then
        begin
            lt[m] := i;
            ut[m] := q-1;
            i := q+1
        end
    else
        begin
            lt[m] := q+1;
            ut[m] := j;
            j := q-1
        end
    end;
    comment Update m and split this new smaller segment;
    m := m+1;
    go to N
end

```

```

    else if i ≥ j then
        begin comment This segment has less than two elements;
            go to P
        end
    else
        begin comment This is the case when the segment has just
            two elements, so sort a[i] and a[j] where j = i + 1;
            if a[i] > a[j] then
                begin
                    x := a[i];
                    a[i] := a[j];
                    a[j] := x
                end;
            comment If the lt and ut arrays contain more segments
                to be sorted then repeat the process by splitting the smallest
                of these. If no more segments remain the array has been
                completely sorted;
P: m := m-1;
    if m > 0 then
        begin
            i := lt[m];
            j := ut[m];
            go to N
        end
    end
end quicksort

```

#### REMARK ON ALGORITHM 250 [G6] INVERSE PERMUTATION

[B. H. Boonstra, *Comm. ACM* 8 (Feb. 1965), 104]  
C. W. MEDLOCK (Recd. 12 Apr. 1965 and 14 July 1965)  
IBM Corp., Programming Systems, Poughkeepsie, N.Y.

Several simplifications may be made to the subject algorithm to permit more efficient operation.

1. On many compilers, the procedure would be more efficient if the outer loop were written as a **for** loop.

2. The initialization of the vector *P* to negative values may be omitted by reversing the interpretation of positive and negative values. As revised, *P*[*i*] contains a negative number if it contains the inverse value and *i* is less than the current value of the parameter *n*. *P*[*i*] contains a positive value in all other cases. This allows the **for** loop labeled *tag* to be eliminated.

3. The variable *first* may be eliminated by declaring the parameter *n* as a value parameter, and utilizing it as the controlled variable of the outer loop.

The author wishes to thank the referee for valuable suggestions.

The revised algorithm then reads:

```

procedure inversepermutation (P) of natural numbers up to: (n);
    value n; integer n; integer array P;
    comment Given a permutation P(i) of the numbers i = 1(1)n,
        the inverse permutation is computed in situ;
    begin integer i, j, k;
    for n := n step -1 until 1 do
        begin i := P[n];
            if i < 0 then P[n] := -i
            else if i ≠ n then
                begin k := n;
                    loop: j := P[i]; P[i] := -k;
                        if j = n then P[n] := i
                        else
                            begin k := i; i := j; go to loop
                            end
                        end
                    end
                end
            end
        end
    end inversepermutation

```