Algorithms Supplement

Previously published algorithms

The following algorithms have recently appeared in the Algorithms Sections of the specified journals.

(*a*) **Communications of the ACM** (May-July 1970)

380 IN-SITU TRANSPOSITION OF A RECTANGULAR MATRIX

Transposes a matrix, assumed stored in a one-dimensional array, by a process of loops.

381 RANDOM VECTORS UNIFORM IN SOLID ANGLE

Generates the components of random unit vectors distributed uniformly in a solid angle.

382 COMBINATIONS OF M OUT OF N OBJECTS

Can be used to generate all combinations of m out of n objects, or to generate all n-length sequences containing m 1's and (n − m) 0's.

383 PERMUTATIONS OF A SET WITH REPETITIONS

A generalisation of Algorithm 382, and of the Trotter-Johnson adjacent-transposition permutation algorithms.

384 EIGENVALUES AND EIGENVECTORS OF A REAL SYMMETRIC MATRIX

Uses a variant of the QR algorithm to evaluate the eigenvalues and, at the user's option, the eigenvectors of a real symmetric matrix.

385 EXPONENTIAL INTEGRAL $E_i(x)$

Evaluates the classical exponential integral

$$E_i(x) \equiv \int_{-\infty}^{x} \frac{e^t}{t}\,dt = -\int_{-x}^{\infty} \frac{e^{-t}}{t}\,dt\,,\ x > 0$$

where the integral is interpreted as the Cauchy principal value.

386 GREATEST COMMON DIVISOR OF n INTEGERS AND MULTIPLIERS

Calculates the greatest common divisor, IGCD, of n integers A(i). Constructs multipliers Z(i) such that

$$IGCD = A(1) \times Z(1) + \ldots + A(n) \times Z(n)\,.$$

The following paper, containing a useful algorithm, has recently appeared in the specified journal.

(*a*) **International Journal for Numerical Methods in Engineering** (October-December 1969)

A SIMPLE MATRIX-VECTOR HANDLING SCHEME FOR THREE-DIMENSIONAL AND SHELL ANALYSIS (Vol. 2, No. 4, pp. 509-522)

New Algorithms

Author's note on Algorithms 63 and 64:

The tree-sort is well known, and a recursive program for it is familiar, e.g. Barron (1968).

However, such programs are usually written in a list processing language, and it may be of interest that substantially the same program can be written in ALGOL 60 (this is Algorithm 63), the list structures being set up by the compiler, handling procedures and their parameters in the usual way.

It is also interesting to consider the storage used by the program. There are, besides the essential array of n keys to be sorted, the two arrays l and r, each of n integers. Besides this there is the storage used in the recursion. How much storage is stacked for each incarnation of a procedure will depend on the compiler, but since the depth of recursion is only the same as the height of the tree, which is of the order $\log_2 n$ (provided the keys are initially in random order) it is possible that this recursive version is more economical of storage than the usual form.

The second algorithm (Algorithm 64) is a non-recursive version: If a recursive program points the way to save some storage, it is usually possible to achieve the same result more efficiently still with a direct program.

The first of these procedures uses the array l to give the result of the sort as a chain. However, if this is then to be used sequentially, e.g. to print out the keys in order, it is quicker to amend the process as follows: All reference to h should be deleted, statements such as $l[h] := 0$ being removed. Instead, in the body of procedure *flatten*, the statements $l[h] := k$; $h := k$ are replaced by a statement causing $a[k]$ to be printed.

To illustrate this, Algorithm 64 is given in the second form, causing print out of the keys instead of a flattened list.

Reference

BARRON, D. W. (1968). *Recursive techniques in programming*, London: Macdonald p. 27.

Algorithm 63

A RECURSIVE TREE SORT

A. D. Woodall
North Staffordshire Polytechnic
Beaconside, Stafford

```
procedure rects (a, l, n); value n; integer n; array a; integer array l;
comment n is the number of items to be sorted, held in the array
a[1 : n]. l is an integer array with subscript bounds 0 to n, l will
be used to provide the left pointers of items on the tree. When the
tree is flattened l will hold the links of the final list, the first item
being a[l[0]], the successor of a[i] being a[l[i]], and the final item
being a[k] where l[k] = 0;
  begin integer t, h; integer array r[1 : n];

  procedure totree (i, j); value j; integer i, j;
  if i = 0 then i: = j else
    begin
    if a[i] > a[j] then totree (l[i], j)
    else totree (r[i], j)
    end totree;

  procedure flatten (k); value k; integer k;
    begin
    if l[k] ≠ 0 then flatten (l[k]);
    l[h]: = k; h: = k;
    if r[k] ≠ 0 then flatten (r[k])
    end flatten;

  h: = 0;
  for t := 1 step 1 until n do l[t] := r[t] := 0;
  for t := 2 step 1 until n do totree (1, t);
  flatten (1);
  l[h] := 0
  end rects;
```

Algorithm 64

A NON-RECURSIVE TREE SORT

A. D. Woodall
North Staffordshire Polytechnic
Beaconside, Stafford

```
procedure ts(n, a, print); value n; integer n; array a; procedure
print;
comment the n items to be sorted are held in the array a[1: n]
which is unchanged after the call of ts. print must be a procedure
which, when called as print (X) will print the value of X on a new
line;
  begin integer k, t, c, h, mh; integer array r, l[1: n];
  comment r and l are the usual left and right pointers. As each
  item is added to the tree, h counts the number of leftwards
  moves it makes, mh keeps the greatest value of h which will
  be used to fix the size of the array of upward pointers needed
  to flatten the tree;
  for k := 1 step 1 until n do l[k] := r[k] := 0;
  mh := 0;
  for k := 2 step 1 until n do
    begin
    t := 1; h := 0;
LOOP: if a[k] < a[t] then
      begin
      h := h + 1;
      if l[t] = 0 then l[t] := k
      else
        begin
        t := l[t];
        goto LOOP
        end
      end
    else
      begin
      if r[t] = 0 then r[t] := k
      else
        begin
        t := r[t];
        goto LOOP
        end
      end;
    if h > mh then mh := h
    end;
    begin integer array up[1: mh];
    comment as the tree is unloaded upward pointers will be set
    in the array up pointing up the left-most remaining side of the
    tree towards the top;
    c := 0; k := 1;
L1: if l[k] ≠ 0 then
      begin
      c := c + 1;
      up[c] := k; k := l[k];
      goto L1
      end;
L2: print (a[k]);
    if r[k] ≠ 0 then
      begin
      k := r[k];
      goto L1
      end;
    if c ≠ 0 then
      begin
      k := up[c];
      c := c - 1; goto L2
      end
    end
  end ts;
```

Algorithm 65

AN IMPROVED CLUSTERING ALGORITHM

A. H. J. Sale
Basser Computing Department
University of Sydney

Author's Note:

This set of routines produces the same type of results as Algorithm 47 (A Clustering Algorithm) in that it will generate and return all subsets of objects from a given set that satisfy the condition that the maximum dissimilarity (or 'distance') between members of the subset is less than the least dissimilarity between any member of the subset and any object not in the subset. Excluded from this definition are sets consisting of single objects only, and the set of all the objects. It differs from Algorithm 47 in its overall design, in its method of returning results, in its use of storage, and in its execution speed. Subsequent paragraphs detail these differences.

The Algorithm given here has been tested on an IBM 7040 under both the IBFTC and WATFOR compilers with both constructed examples and random data for numbers of objects in a set ranging from 10 to 100. In all cases it produced correct results.

The parameters required are fully detailed in the source language comments; however it should be pointed out here that storage of the order of n^2 variables is required for a set of n objects, and the run time should vary approximately as n^3. The following criteria were employed in the design of the program:

1. The vector of distances should not be destroyed by the routine.
2. Since large numbers of objects are to be expected, the execution time should be as short as possible, and as little storage as possible should be used.
3. The results were to be returned to the calling routine, rather than printing them (which makes them unavailable for further processing).
4. The operation of the routines should be clear and simple to understand.

The process used is to search for a smallest dissimilarity between a pair of subsets, then to merge those two into one, simultaneously making the dissimilarities between them inaccessible, but noting the cluster 'diameter'. Then all other subsets are scanned to make a choice between the dissimilarities to each of the (now merged) subsets. Two of the four values are kept: the least and the greatest. The least dissimilarity is needed for subsequent searches; the greatest is needed to determine the subset diameter. During this scan the merged subset diameter is compared with the least external dissimilarities: if it is less than all of these then the subset is a cluster, otherwise another merge must be initiated. To start the process all objects are regarded as subsets with identical maximum and minimum inter-subset dissimilarities.

This algorithm arose out of a certification of Algorithm 47 in which it appeared that it contained several major bottlenecks: the initial sort, the testing of the clustering condition, and final sort, all of which had run-times proportional to n^4.

In storage requirements the new Algorithm appears superior in program size, and certainly in data storage. To verify the expected improvement in execution time a series of examples were run using test data of points with random (x, y) co-

Table 1

Comparative run-times on identical data for Algorithm 47, an improved version of Algorithm 47, and Algorithm 65

NUMBER OF OBJECTS	TIMES RECORDED ON IBM 7040 FOR:		
	ALGORITHM 47	IMPROVED ALGORITHM 47	ALGORITHM 65
10	4·5 sec	2·2 sec	1·2 sec
20	32·9	17·4	4·6
30	177·8	78·6	12·8
40	516·0	228·1	27·7
50	1281·8	528·6	51·0
60	2377·6	1079·7	85·2
70	—	1868·2	132·9
80	—	—	195·9
90	—	—	273·8
100	—	—	373·6

ordinates in a square two-dimensional space. These results are shown in **Table 1**, for numbers of objects from 10 to 100. These results verify the predictions and show a large superiority of the new Algorithm over the original. It should be pointed out that the times will vary slightly with the data: the test data is characterised by few large clusters but many doublets.

To sum up, this new Algorithm seems to be superior in both program and data space utilisation, and in execution time. There seems however to be one place where a version of Algorithm 47 might still be preferable: where the large tables (of size $n \times (n-1)/2$ variables) are sorted and stored on a serial access medium (for example magnetic tape). The reason for this is that Algorithm 47 always runs serially through the tables (wholly or partially) while the new Algorithm requires random access to the tables. A more recent algorithm (Algorithm 52) while slightly different in function points the way to perform the same clustering process with approximately:

(number of different dissimilarities) + $\log_2(n^2)$

passes through tapes holding $2n^2$ variables. This for some cases would produce run-times proportional to n^3; for others proportional to $n^2 \log n^2$. Whether a problem too large to fit in core can be completed in this way in a reasonable time will of course depend on the machine.

Reference

VAN RIJSBERGEN, C. J. (1970). Algorithm 47: A clustering algorithm, *The Computer Journal*, Vol. 13, No. 1, pp. 113-115.

```
      SUBROUTINE CLUST3(KSIZE,KDAT,TABLE,SWITCH,KOUT,KLENG,DIAM,
     1SPACE,KNUM,KLINK,KHEAD,MAXMIN)
C
C     INPUT VARIABLES
C       -UNALTERED BY CLUST3-
C       -MUST NOT BE ALTERED UNTIL -SWITCH- BECOMES .FALSE. -
C       KSIZE
C         NUMBER OF OBJECTS
C       KDAT
C         SIZE OF TABLE (=KSIZE*(KSIZE-1)/2)
C       TABLE
C         VECTOR OF LENGTH KDAT HOLDING DISTANCES BETWEEN
C         OBJECTS. THE MAPPING FUNCTION IS -LOCN-, AND IF
C         I IS GREATER THAN J, THEN THE DISTANCE BETWEEN I AND
C         J IS AT TABLE(((I-1)*(I-2))/2+J)
C
C     CONTROL VARIABLE
C       SWITCH
C         SET BEFORE FIRST CALL TO .FALSE., THE ROUTINE THEN
C         SETS IT .TRUE. AND IT MUST RETAIN THIS VALUE UNTIL THE
C         ROUTINE ITSELF SETS IT .FALSE. BEFORE -RETURN-ING.
C         THIS SIGNIFIES THAT NO MORE CLUSTERS CAN BE FOUND,
C         AND THAT THE OUTPUT RESULTS ARE UNDEFINED
C
C     OUTPUT VARIABLES
C       -SET BY CLUST3-
C       -UNDEFINED AT ENTRY-
C       -MAY BE FREELY ALTERED-
C       KOUT
C         VECTOR OF LENGTH -KSIZE- CONTAINING A SORTED LIST OF
C         -KLENG- OBJECTS FORMING A CLUSTER. THE CONTENTS BEYOND
C         KOUT(KLENG) ARE NOT DEFINED
C       KLENG
C         NUMBER OF OBJECTS IN THE CLUSTER
C       DIAM
C         THE DIAMETER OF THE CLUSTER (MAXIMUM INTRA-CLUSTER
C         DISTANCE)
C       SPACE
C         THE SEPARATION SPACE (DISTANCE FROM THIS CLUSTER TO
C         ITS NEAREST NEIGHBOUR)
C
C     WORKING VARIABLES
C       -SET BY CLUST3-
C       -UNDEFINED AT FIRST ENTRY-
C       -THEY MUST NOT BE ALTERED UNTIL -SWITCH- BECOMES .FALSE. -
C       KNUM
C         THE NUMBER OF SETS AS YET UNCOALESCED
C       KLINK
C         WORK VECTOR OF LENGTH KSIZE, HOLDS LINKED OBJECT
C         INFORMATION
C       KHEAD
C         WORK VECTOR OF LENGTH KSIZE, HOLDS LIST HEADS
C       MAXMIN
C         WORK VECTOR OF LENGTH KDAT, HOLDS POINTERS TO THE
C         MAXIMUM AND MINIMUM INTER-SET DISTANCES. THE TWO
C         POINTERS MAX AND MIN ARE PACKED INTO ONE INTEGER
C
C     SPECIAL COMMENTS
C       INTEGER OVERFLOW MUST NOT OCCUR FOR (KDAT*(KDAT+2))
C       DUE TO THE PACKING INTO MAXMIN. IN CASE OF DIFFICULTY
C       MAXMIN MAY BE SPLIT INTO TWO ARRAYS MMAX AND MMIN, THUS
C       ELIMINATING THE NEED FOR THE VARIABLE -JPACK- AND THE
C       ROUTINE -UNPAK-
C
C     ERROR EXITS
C       NONE, EXCEPT AS EXPLAINED FOR -SWITCH-
C       THE ACTION OF CLUST3 IS UNDEFINED IF THE INPUT VARIABLES
C       OR THE WORKING VARIABLES ARE ALTERED BETWEEN A RETURN
C       FROM CLUST3 WITH -SWITCH- .TRUE. AND A SUBSEQUENT CALL
C       TO CLUST3 WITH -SWITCH- AGAIN .TRUE.
C
C     SPECIFICATIONS
      INTEGER KSIZE,KDAT,KLENG,KNUM
      REAL DIAM,SPACE
      LOGICAL SWITCH
      INTEGER KLINK(KSIZE),KHEAD(KSIZE),KOUT(KSIZE),MAXMIN(KDAT)
      REAL TABLE(KDAT)
C     DECLARATIONS
      INTEGER J,L,M,KT,KSIZET,JPACK
      INTEGER LMAX,LMIN,MAX,MIN,JSET1,JSET2
      REAL RMIN
      LOGICAL TSW
C
C     ROUTINE START POINT
C     TEST TO SEE IF INITIALISING ENTRY TO CLUST3
      IF (SWITCH) GO TO 3
C     INITIALISE THE TABLES AND VARIABLES
      JPACK=KDAT+1
      DO 1 J=1,KSIZE
      KLINK(J)=0
      KHEAD(J)=J
    1 CONTINUE
      DO 2 J=1,KDAT
      MAXMIN(J)=J*JPACK+J
    2 CONTINUE
      KNUM=KSIZE
      SWITCH=.TRUE.
C
C     TEST TO SEE IF TO REJECT THE APPLICATION FOR A CLUSTER
    3 IF (KNUM.GT.2) GO TO 4
C     IF THERE ARE ONLY TWO SETS TO COALESCE, WE CAN ONLY
C     GET THE SET OF ALL OBJECTS, SO GIVE UP
      SWITCH=.FALSE.
      RETURN
C
C     SCAN FOR THE ABSOLUTE MINIMUM INTER-SET DISTANCE
    4 TSW=.TRUE.
C     RUN DOWN THE CLUSTER TABLE
      KSIZET=KSIZE-1
      DO 7 J=1,KSIZET
      IF (KHEAD(J).EQ.0) GO TO 7
C     GOT AN EXISTING CLUSTER
C     IS THERE A HIGHER NUMBERED ONE TOO
      L=J+1
      DO 6 M=L,KSIZE
      IF (KHEAD(M).EQ.0) GO TO 6
C     GOT A PAIR, GET THE LINK POINTERS
      KT=LOCN(J,M)
      CALL UNPAK(LMAX,LMIN,MAXMIN(KT),JPACK)
C     IF ITS THE FIRST PAIR, ACCEPT THE DISTANCES
      IF (TSW) GO TO 5
C     IF THE MIN DISTANCE IS LESS THAN THE PRESUMED MIN, TAKE IT
      IF (TABLE(LMIN).GE.RMIN) GO TO 6
C     KEEP INFORMATION ABOUT THIS PAIR
    5 TSW=.FALSE.
      JSET1=J
      JSET2=M
      RMIN=TABLE(LMIN)
      DIAM=TABLE(LMAX)
    6 CONTINUE
    7 CONTINUE
C     WE NOW HAVE THE CLOSEST PAIR OF CLUSTERS, AND THEIR
C     DIAMETER AS A JOINT CLUSTER
C
C     KEEP THE MAX AND MIN OF THE PAIR DISTANCES
      TSW=.TRUE.
C     RUN THROUGH ALL CLUSTERS, EXCEPT THE TWO SETS FOUND
      DO 9 J=1,KSIZE
      IF (KHEAD(J).EQ.0) GO TO 9
      IF ((J.EQ.JSET1).OR.(J.EQ.JSET2)) GO TO 9
C     GET LOCATIONS OF DISTANCES RELEVANT
      L=LOCN(J,JSET1)
      M=LOCN(J,JSET2)
      CALL UNPAK(LMAX,LMIN,MAXMIN(L),JPACK)
      CALL UNPAK(MAX,MIN,MAXMIN(M),JPACK)
C     KEEP THE LARGEST AND SMALLEST DISTANCE
      IF (TABLE(LMAX).LT.TABLE(MAX)) LMAX=MAX
      IF (TABLE(LMIN).GT.TABLE(MIN)) LMIN=MIN
      MAXMIN(L)=LMAX*JPACK+LMIN
C     ON FIRST PASS KEEP AS ABSOLUTE MIN
      IF (TSW) GO TO 8
C     IS THIS DISTANCE LESS THAN PRESUMED ABSOLUTE MIN
      IF (TABLE(LMIN).GE.RMIN) GO TO 9
    8 RMIN=TABLE(LMIN)
      TSW=.FALSE.
    9 CONTINUE
C     NOW WE HAVE ALL THE DISTANCES FOR JSET1 CORRECT
C
C     JOIN UP ALL THE CLUSTERS
C     AND SIMULTANEOUSLY START BUILDING UP THE OUTPUT VECTOR
      L=KHEAD(JSET1)
      J=1
      KOUT(1)=L
C     RUN ALONG THE LINKS
   10 M=KLINK(L)
      IF (M.EQ.0) GO TO 11
      J=J+1
      KOUT(J)=M
      L=M
      GO TO 10
C     NOW JOIN ON SET 2 BY THE LINK
   11 KLINK(L)=KHEAD(JSET2)
      KHEAD(JSET2)=0
      KNUM=KNUM-1
C     THIS IS THE EARLIEST POINT THAT WE CAN CHECK FOR THE
C     CORRECTNESS OF THE CLUSTER CONDITION
      IF (RMIN.LE.DIAM) GO TO 3
C     GET THEN THE REST OF THE OBJECTS INTO OUTPUT
   12 M=KLINK(L)
      IF (M.EQ.0) GO TO 13
      J=J+1
      KOUT(J)=M
      L=M
      GO TO 12
```

```
C   WE NOW HAVE A CLUSTER, FINISH OFF
 13 KLENG=J
    CALL SORT(KOUT,J)
    SPACE=RMIN
    RETURN
    END
C
    SUBROUTINE SORT(KOUT,KLENG)
    INTEGER KLENG,KOUT(KLENG)
C
C   THIS ROUTINE SORTS THE VECTOR -KOUT- INTO ASCENDING ORDER
C   ELEMENTS KOUT(1) TO KOUT(KLENG) ARE AFFECTED.
C   A SIMPLE BUBBLE SORT IS USED.
C
    INTEGER J,J1,K
    K=KLENG
  1 K=K-1
    IF (K.LE.0) RETURN
    DO 2 J=1,K
    IF (KOUT(J).LE.KOUT(J+1)) GO TO 2
    J1=KOUT(J)
    KOUT(J)=KOUT(J+1)
    KOUT(J+1)=J1
  2 CONTINUE
    GO TO 1
    END
C
    INTEGER FUNCTION LOCN(J,K)
    INTEGER J,K
C
C   LOCN RETURNS THE LOCATION OF THE (J,K) ELEMENT IN THE
C   TRIANGULAR ARRAY STORED IN MAXMIN AND TABLE
C
    IF (J.GT.K) GO TO 1
    LOCN=((K-1)*(K-2))/2+J
    RETURN
  1 LOCN=((J-1)*(J-2))/2+K
    RETURN
    END
C
    SUBROUTINE UNPAK(J,K,L,JPACK)
    INTEGER J,K,L,JPACK
C
C   ROUTINE TO UNPACK TWO INTEGERS FROM ONE.
C     J AND K COME FROM L.
C     JPACK DETERMINES THE PACKING FUNCTION.
C
    J=L/JPACK
    K=L-J*JPACK
    RETURN
    END
```

Note on Algorithm 46

A MODIFIED DAVIDON METHOD FOR FINDING THE MINIMUM OF A FUNCTION, USING DIFFERENCE APPROXIMATION FOR DERIVATIVES

There are three misprints in the procedure DAPODMIN. The beginning of the third last line in procedure *up dot* should be replaced by

$$\textbf{for } j := i \textbf{ step } 1 \ldots$$

and the third last line in procedure *set unit h and H* should be replaced by

$$k := k + n - i + 1$$

The sixth line after the label *SEARCH ALONG S* should read

$$check := check \vee H[i] \leqslant 0;$$

Also I recommend that the lines

$$estd := 2 \times exp(ln(abs(f \times oldg[j]) \times E/H[j] \uparrow 2)/3);$$

in procedure *grad* be replaced by

$$estd := 2 \times abs(f \times oldg[j] \times E/H[j] \uparrow 2) \uparrow (1/3);$$

thus avoiding calculating the logarithm of a number close to 0. An improvement which may reduce slightly the number of function evaluations required to solve a problem may be made to the last section of the linear search. The 8 lines beginning with the line

$$f := fy; fy := fz;$$

should be replaced by

for $i := 1$ **step** 1 **until** n **do**
begin
$x[i] := y[i]; y[i] := z[i]$
end;

$gx := (fy - f)/b;$
$f := fy; fy := fz;$

Shirley A. Lill
Department of Computational Sc.
University of Leeds,
LS2 9JT.

To the Editor
The Computer Journal

Sir,

With reference to Algorithm 50 (this *Journal*, Volume 13, pp. 208-219) I would like to point out that the author's 'Bell's position' (Fig. 3(a), p. 211) for the simplest two move mate position known, is not in fact a legal chess position, i.e. it cannot occur in over-the-board play.

I have constructed a simpler position which gives a two move forced mate in the same manner as Bell's position, and my example can occur as a result of legal chess moves. The position is shown in **Fig. 1,** and a sample game leading to this position (there will doubtless be shorter such games) is given below.

Black

					BB	BR	BK
				BP		BP	
				WP		WP	
					BP		
					BP	BP	
				BP		WP	
				WP		WP	WB
					WB	WK	WR

White

Fig. 1

	White	Black
1.	P—QN3	N—KB3
2.	B—N2	N—K5
3.	B—K5	P—KB4
4.	P—KR4	N—N4
5.	PXN	K—B2
6.	B—R2	P—QR4
7.	N—KB3	R—R3
8.	N—B3	P—R4
9.	P—R3	K—N3
10.	N—K5 ch.	K—R2
11.	N—N4	RPXN
12.	P—N4	R—K3
13.	Q—N1	N—QB3
14.	Q—N2	P—QN3
15.	Q—N1	R—K6
16.	Q—N2	R—N1
17.	P—N6 ch.	K—R1
18.	Q—N1	R—N6
19.	PXR	B—R3
20.	K—B2	N—K4
21.	K—N1	B—Q6

22.	BPXB	Q—R1
23.	P—Q4	Q—B3
24.	PXN	PXP
25.	R—R2	PXP
26.	RXP	Q—N2
27.	P—Q4	P—Q3
28.	P—Q5	PXP
29.	N—N5	Q—N1
30.	R—KB3	P—B4
31.	R—B4	PXR
32.	N—Q4	PXN
33.	QXNP	Q—R1
34.	Q—N3	Q—B1
35.	Q—K3	Q—K3
36.	PXQ	QPXQ

This gives the position in which White has no option but to mate Black by a series of three forced moves. This uses the same theme as in Bell's position.

Yours faithfully,

J. L. BERRY

The National Computing Centre Limited
Manchester M3 3HU
May 1970

Mr. Bell replies:

Ouch! The example was intended to show some actual numbers generated and paths taken by the program. It is illegal because of the pawn structure which Berry has remedied by removing the two (superfluous) middle pawns in the King's file.

I should have defined 'simplest'. What I meant is a position from which the tree structure is minimal, i.e. restricted to one and only one generated (*legal* or *illegal*) move at each play. Berry's position allows both the Black and White King illegal moves and the program would take slightly longer to prove the unique (P × P) solution. It is blocking the Black King that is difficult. Incidentally, if you ignore illegal King moves and always 'queen' a pawn then

BK				
BP				
WK	BP		WP	
	WP			

has the forced sequence

1. P—Q7
2. K—N8
3. P—Q8(Q) ck. mt.

Contributions for the Algorithms Supplement should be sent to
Mrs. M. O. Mutch
University Engineering Department
Control Engineering Group
Mill Lane, Cambridge

Book review

Rank Order Probabilities: Two Sample Normal Shift Alternatives, by Roy C. Milton, 1970; 302 pages. (*Wiley*, 125*s* = £6·25)

One aim of this book was to produce an accurate and comprehensive set of tables of the probabilities of rank order, when small samples are taken from two normal distributions with the same variance but different means. There can be no doubt that the author unequivocally achieves this aim. As regards computational technique, the basic method used to compute this probability (which takes the form of a multi-dimensional integral) is well known, but the use of 'extrapolation to the limit' to improve accuracy is unusual (I suspect), and well worth noting.

The author then uses his basic tables to solve a number of statistical problems. For example, in Chapter 2 he calculates and compares the power functions of some non-parametric two-sample test statistics (like those of Wilcoxon, Fisher-Yates, and Kolmogorov) in the normal case. Of course, there already exists a literature on the efficiency of these tests, but Milton's book certainly seems to contain many new results. For instance, it is shown that the Fisher-Yates test is not always more powerful than the Wilcoxon test and vice versa. Chapter 3 is interesting too since it contains results on the power of some sequential two-sample rank tests. This is a topic that will be quite new to many statisticians.

In my opinion, the author has already put his tables to good use. I would not be surprised if other people (researchers probably) found equally good uses for them in the future.

J. G. FRYER (Exeter)