c. In the algorithm GOMORY there are statements of the form

$$C := entier\ (A/B)$$

where C is an integer variable, and A and B are integer type expressions. In order to prevent roundoff errors the result C should be checked to make sure that

$$C \times B \leq A < C \times B + B$$

and corrected if these inequalities are not satisfied.

The corrections, a, b, c, lead to a program which cannot fail unless the products developed should overflow. However, anyone who wishes to use the algorithm may prefer to do some analysis of the particular division his computer performs and seek an alternative which is not as time-consuming. Many machines have a built-in Euclidean division instruction for integer numbers which would be very useful for Gomory's algorithm. Unfortunately ALGOL translators are not likely to produce this instruction in their object programs since an arithmetical expression A/B is a real type expression by definition.

```
procedure Gomory 1 (m, n) transient: (a) exit: (no solution);
  value m, n;
  integer m, n;
  integer array a;
  label no solution;
comment Gomory 1 algorithm for all-integer programming.
  The objective of this procedure is to determine the integer solution x[1], ··· , x[n−1] of a linear programming problem with integer coefficients only. In other words: The problem is to find
  integer numbers
      x[1], ··· , x[n−1]
  minimizing the objective function
      a[0, 1] × x[1] + ··· + a[0, n−1] × x[n−1]
  under the constraints
      x[1] ≥ 0, ··· , x[n−1] ≥ 0
  and
      a[i, 1] × x[1] + ··· + a[i, n−1] × x[n−1] ≤ a[i, n]
  for i = 1, ··· , m−n+1  (2 ≤ n ≤ m).
    The tableau matrix a used by the procedure consists of m+1 rows and n columns. The components are a[i, j] for i = 0, 1, ··· , m, j = 1, ··· , n.
    The input values for the components are given partly by the problem itself (see above). The remaining components must have been previously assigned in the following manner:
      a[0, n] := 0
  and
      a[i, j] := if i = j + m − n + 1 then −1 else 0
  for i = m−n+2, ··· , m, j = 1, ··· , n. The tableau columns, with the exception of the last column, have to be lexicographically positive.
    The algorithm is finished if all entries in the last column, except the topmost entry, are non-negative. Then −a[0, n] is the value of the objective function. The optimal solution x[1], ··· , x[n−1] is given by the n−1 components a[m−n+2, n], ··· , a[m, n] of the last column of a.
    The exit no solution is used if a row is found which has a negative entry in the last column, but otherwise only non-negative entries;
begin integer i, k, j, l, r, c, t, s, lambda num, lambda denom;
  integer procedure Euclid (u, v);
    value u, v;
    integer u, v;
  begin integer w;
    w := entier (u/v);
L8: if w × v > u then
    begin w := w−1; go to L8 end;
L9: if (w+1) × v ≤ u then
    begin w := w+1; go to L9 end;
    Euclid := w
  end Euclid;
L1: for i := 1 step 1 until m do if a [i, n] < 0 then
    begin r := i; go to L2 end;
  go to end;
L2: for k := 1 step 1 until n−1 do if a[r, k] < 0 then go to L4;
  go to no solution;
L4: l := k;
  for j := k+1 step 1 until n−1 do if a[r, j] < 0 then
    begin i := 0;
L3: if a[i, j] < a[i, l] then l := j else
      if a[i, j] = a[i, l] then
      begin i := i+1; go to L3 end
    end;
  s := 0;
L5: if a[s, l] = 0 then
  begin s := s+1; go to L5 end;
  lambda num := −a[r, l];
  lambda denom := 1;
  for j := 1 step 1 until l−1, l+1 step 1 until n−1 do
    if a[r, j] < 0 then
      begin
      for i := 0 step 1 until s−1 do if a[i, j] ≠ 0 then go to L7;
      t := Euclid (a[s, j], a[s, l]);
      if (t×a[s, l] = a[s, j]) ∧ (t>1) then
      begin i := s;
L6:     i := i+1;
        if t×a[i, l] = a[i, j] then go to L6 else
        if t×a[i, l] > a[i, j] then t := t−1
      end;
      if −a[r, j] × lambda denom > t × lambda num then
      begin lambda num := −a[r, j]; lambda denom := t end;
L7: end;
  for j := 1 step 1 until l−1, l+1 step 1 until n do
    begin c := Euclid (a[r, j] × lambda denom, lambda num);
      if c ≠ 0 then
      for i := 0 step 1 until m do
        a[i, j] := a[i, j] + c × a[i, l]
    end;
  go to L1;
end:
end
```

ALGORITHM 264
INTERPOLATION IN A TABLE [E1]

J. STAFFORD (Recd. 16 Nov. 1964 and 7 June 1965)
Westland Aircraft Ltd., Saunders-Roe Division, East Cowes, Isle of Wight, England

```
real procedure INPOL(T, X, I, N, OUT, XOUT, EXPOL);
  value X, N; array T, X; integer I; integer array N;
  real XOUT, EXPOL; Boolean OUT;
comment Evaluation of a function by polynomial interpolation in a table of values.
  The values may be specified at arbitrary intervals, at nodes of a multidimensional rectangular grid. The interpolation is by Neville's process, repeated in each dimension.
  The given values are arranged in a one-dimensional real array T, as follows. The first value in the table, T[0], is D, the number of independent variables (or dimensions). It will normally be integral (although of type real), but if not then its integral part is taken. T[1], T[2], ··· , T[D] are the numbers of values of X1, X2, ··· , XD, and must be integral. These are followed by
```

$T[1]$ values of $X1$, $T[2]$ values of $X2$, $\cdots$ $T[D]$ values of XD. The values of each of these independent variables must all be distinct and must be arranged in monotonic order. Finally come the $T[1] \times T[2] \times \cdots \times T[D]$ values of the dependent variable $F(X1, X2, \cdots, XD)$, arranged as $T[D]$ sets of $T[D-1]$ sets of $\cdots$ of $T[2]$ sets of $T[1]$ values of F.

The table is represented by a one-dimensional array because it is not feasible to use a general D-dimensional array.

The given values of the independent variables are $X[I]$ $(I=1, 2, \cdots, D)$. $N[I]$ of the tabulated values $X[I]$ are used to interpolate in the Ith dimension. *INPOL* is the required value of the function. The actual parameter corresponding to the formal parameter *EXPOL* should be an expression which provides the value of *INPOL* if any of the $X[I]$ is outside the range covered by the array T. If this occurs *XOUT* is the particular value of $X[I]$ concerned. The variables I, *OUT* and *XOUT* are declared as formal parameters of *INPOL* so that they may be used in the actual parameter corresponding to the formal parameter *EXPOL*.

An example of a call of *INPOL* is $Z :=$ *INPOL*(A, X, K, N, *OUT*, Y, **if** $K=1$ **then** *EXTRAPOLATE* (A, 1, N, *OUT*, Y) **else if** $K=2$ **then** *LIMTAB* (A, 2, *OUT*, Y) **else** $Y-2$). If $X[1]$ is outside the range covered by the array A this statement will use the extrapolatory procedure *EXTRAPOLATE* (given below) to provide a value for *INPOL*. If $X[2]$ is out of range the procedure *LIMTAB* (also given below) will be used to replace the value of $X[2]$ by its value at the nearer edge of the table, before returning to *INPOL* to continue the interpolation. If some other variable ($X[3]$, say) is out of range the value of *INPOL* is taken as $X[3] - 2$.

The procedures *INPOL*, *EXTRAPOLATE* and *LIMTAB* were tested on an ICT Atlas computer. They were also tested on a National-Elliott 803 computer, after being altered to conform to the restrictions of the 803 ALGOL compiler. The tests were for $D = 0$, 1, 2 and 3, and included all special cases;

```
begin integer D, J, K, L, M, Q, XI;

  procedure FORS3(N, P, V, UB);
    value N; integer N; procedure P;
    integer array V, UB;
  comment Nesting of for statements, adapted from procedure
    Fors 1 [Algorithm 137, Comm. ACM 5 (Nov. 1962), 555];
  begin integer J;
    if N = 0 then P else for J := 1 step 1 until UB[N] do
      begin V[N] := J; FORS3(N-1, P, V, UB) end
  end FORS3;

  real procedure NEV(X, AX, SAX, AY, SAY, N);
    value X, SAX, SAY, N; real X; integer SAX, SAY, N;
    array AX, AY;
  comment One-dimensional interpolation by Neville's process.
    N values of the independent variable are used in the
    interpolation, namely, N consecutive elements of array AX
    starting at subscript SAX. The corresponding values of the
    dependent variable are the N consecutive elements of array
    AY starting at subscript SAY. X is the value of the inde-
    pendent variable for which the value of the dependent vari-
    able (namely, NEV) is to be interpolated;
  begin integer I, J, NJ, KI; array F[0: N-1];
    for J := 0 step 1 until N - 1 do F[J] := AY[SAY + J];
    for J := 1 step 1 until N - 1 do
    begin
      NJ := N - J - 1;
      for I := 0 step 1 until NJ do
      begin
        KI := SAX + I;
        F[I] := (F[I+1]-F[I]) × (X-AX[KI])/
          (AX[KI+J]-AX[KI])+F[I]
      end;
    NEV := F[0]
  end NEV;

  D := entier (T[0]);
  comment D = number of dimensions. The special case D = 0
    implies that the tabulated function F is a constant, the value
    of which is T[1]. The same value is taken if D < 0;
  if D < 1 then INPOL := T[1] else
  begin XI := 1;
    for I := 1 step 1 until D do
    begin
      if N[I] < 2 then N[I] := 2;
      if N[I] > T[I] then N[I] := T[I];
      comment Adjustment of number of points used for inter-
        polation. Normally N[I] must be at least 2, and if N[I] < 2
        it is set equal to 2. N[I] also may not exceed the number
        of values of the independent variable in the correspond-
        ing dimension (namely, T[I]), and if it does so it is re-
        duced accordingly.
          The combination of these two tests, in this order,
        permits as a special case one-point interpolation in any
        particular dimension (I, say), if T[I] = 1. This implies
        that the dependent variable is independent of X[I]. If
        this is intended then the actual parameter corresponding
        to the formal parameter EXPOL must be a function
        designator which (if called for) replaces the value of
        XOUT by the single value of the Ith variable from the
        array T. (Procedure LIMTAB may be used for this
        purpose.)
          Since array N is called by value none of these adjust-
        ments affects the values of N[I] in the nonlocal array N;
      XI := XI + N[I]
    end I;
    begin array F[1: XI-N[1]]; integer array V, XINIT,
      YINC [1: D];

      procedure ONEWAY;
        comment Performs an interpolation in the first dimen-
          sion. If this is the last of a set of N[2] such interpola-
          tions, a further interpolation is performed in the second
          dimension, and so on to as many higher levels as neces-
          sary;
        begin F[V[1]] := NEV (X[1], T, XINIT[1], T, Q, L);
          I := 1; M := 0;
          for K := 1 step 1 until D - 1 do
          begin Q := Q + YINC[K];
            if V[K] ≠ N[K] then go to CONTINUE else
            begin M := M + N[K];
              F[M+V[K+1]] := NEV(X[K+1], T, XINIT[K+1],
                F, I, N[K]);
              I := I + N[K]
            end
          end;
        CONTINUE:
        end ONEWAY;

        Q := XI := D + 1; M := 1;
        for I := 1 step 1 until D do
        begin K := XI + T[I] - 1;
          OUT := (X[I] - T[XI]) × (X[I] - T[K]) > 0;
          if OUT then
          begin
            XOUT := X[I]; INPOL := EXPOL; X[I] := XOUT;
            if T[0] ≤ 0
              then begin K := K + T[0]; T[0] := D end
          end;
```

```
comment If X[I] is outside the range covered by the
  table, the extrapolatory expression EXPOL is evalu-
  ated. It is expected that it will often be or contain one
  or more function designators, together with criteria
  for choosing between them, as in the example above.
    EXPOL may incorporate, e.g., any of the following
  alternatives. In the first and third of these the side
  effects are the important ones, the value assigned to
  EXPOL being merely a dummy to conform with Section
  5.4.4 of the Revised Report on Algol 60 [Comm. ACM 6
  (Jan. 1963), 1-17].
  1. EXPOL may be a function designator which uses
     the interpolatory formula to extrapolate by executing
     the statement OUT := false and returning to INPOL.
     The last N[I] values of X[I] are used in the formula,
     but EXPOL may arrange to use the first N[I] values
     instead (which will usually be preferable if X[I] lies
     beyond the lower limit of the table) by executing the
     statement T[0] := N[I] - T[I] (in which the value of
     the local N[I] is to be used if it differs from that of
     the nonlocal N[I]). The procedure EXTRAPOLATE
     (given below) may be used for this purpose.
  2. EXPOL may use some other formula to extrapolate,
     after which it must return to INPOL without altering
     the value of the Boolean variable OUT. If this is all
     that is required the actual parameter corresponding
     to EXPOL may be an ordinary arithmetic expression
     containing no function designators.
  3. EXPOL may be a function designator which con-
     strains X[I] to lie within range by replacing it by the
     value of the Ith variable at the nearer limit of the
     table (or by some other value). In doing this it must
     operate on the value of XOUT and not directly on
     X[I]. The nonlocal array X will not be affected.
     EXPOL must also execute the statement OUT :=
     false before returning to INPOL. The procedure
     LIMTAB (given below) may be used for this purpose.
  4. EXPOL may do something else and continue the
     program without returning to INPOL (e.g., by a go
     to statement referring to a nonlocal label). This
     should be considered an error exit as the value of
     INPOL will be undefined (see Section 5.4.4 of the
     Revised Report on Algol 60);
if OUT then go to B;
comment If OUT = true on exit from INPOL then
  extrapolation has occurred. The converse is not neces-
  sarily true, as it depends on the nature of the actual
  parameter corresponding to the formal parameter
  EXPOL;
J := XI;
L := (J+K) ÷ 2;
if (X[I]-T[J]) × (X[I]-T[L]) > 0 then J := L else
  K := L;
if K - J > 1 then go to A; comment Find X[I] in
  table;
L := K - N[I] ÷ 2;
if L ≤ XI then L := XI else
begin
  K := XI + T[I] - N[I]; if L > K then L := K
end Adjustment near edge of table;
Q := Q + T[I] + (L - XI) × M; XINIT[I] := L;
XI := XI + T[I];
YINC[I] := M × (T[I] - (if I=1 then 0 else N[I]));
M := M × T[I]
end I;
V[D] := 1; L := N[1];
for I := 1 step 1 until D - 1 do N[I] := N[I+1];
    FORS3(D-1, ONEWAY, V, N); INPOL := F[M+1]
  end scope of F
end D ≥ 1;
B:
end INPOL;

real procedure EXTRAPOLATE(T, I, N, OUT, XOUT);
  array T; integer I; integer array N; Boolean OUT;
  real XOUT;
comment This function designator is intended for use in the
  actual parameter corresponding to the formal parameter
  EXPOL in a call of procedure INPOL. The parameters have
  the same significance as in INPOL.
    EXTRAPOLATE arranges for the interpolatory formula to
  be used to extrapolate for the Ith variable, and for the first
  N[I] values of this variable to be used in the formula instead
  of its last N[I] values if it lies beyond the lower limit of the
  table;
begin integer J, K;
  OUT := false; EXTRAPOLATE := 0;
  comment This statement assigns a dummy value to EXTRAP-
    OLATE to conform with Section 5.4.4 of the Revised Report
    on Algol 60;
  J := 1; for K := 0 step 1 until I - 1 do J := J + T[K];
  if T[I] = 1 then XOUT := T[J] else
    if abs(XOUT - T[J]) < abs(XOUT-T[J+T[I]-1]) then
    begin K := N[I];
      if K < 2 then K := 2;
      if K > T[I] then K := T[I];
      T[0] := K - T[I]
    end
end EXTRAPOLATE;

real procedure LIMTAB(T, I, OUT, XOUT);
  array T; integer I; Boolean OUT; real XOUT;
comment This function designator is intended for use in the
  actual parameter corresponding to the formal parameter
  EXPOL in a call of procedure INPOL. The parameters have
  the same significance as in INPOL.
    LIMTAB replaces the value of XOUT, which is outside the
  range of the table, by the value of the Ith variable at the nearer
  edge of the table;
begin integer J, K;
  J := 1; for K := 0 step 1 until I - 1 do J := J + T[K];
  K := J + T[I] - 1;
  LIMTAB := XOUT := if abs(XOUT-T[J]) >
    abs(XOUT-T[K]) then T[K] else T[J];
  comment This statement assigns a dummy value to LIMTAB
    to conform with Section 5.4.4 of the Revised Report on
    Algol 60;
  OUT := false
end LIMTAB
```

ALGORITHM 265
FIND PRECEDENCE FUNCTIONS [L2]

NIKLAUS WIRTH (Recd. 14 Dec. 1964 and 22 Dec. 1964)
Computer Science Dept., Stanford U., Stanford, Calif.

```
procedure Precedence (M, f, g, n, fail);
value n; integer n; integer array M, f, g; label fail;
comment M is a given n × n matrix of integers designating one
  of the four relations <, =, >, ◦. The identifiers ls, eq, gr des-
  ignate variables declared outside the procedure to which distinct
  integers representing the relations <, =, > have been assigned.
  This procedure then determines integers f[1] . . f[n] and g[1]
  . . .g[n] such that for all i, j, f[i] M[i, j] g[j] is true and so that the
```

smallest of these integers is +1. ∘ designates the empty relation, so that $x \circ y$ is true for arbitrary x, y. If M is such that no f and g exist which satisfy all n^2 relations, then control is transferred to the label parameter *fail*. This procedure has been used to determine the precedence functions of symbols in a given precedence grammar (see [FLOYD, R. Syntactic analysis and operator precedence. *J.ACM 10* (1963), 316–333]);

```
begin integer i, j, k, k1, fmin, gmin;
 procedure fixrow (i, l, x); value i, l, x; integer i, l, x;
 begin integer j; f[i] := g[l] :+ x;
   if k = k1 then
   begin if M[i, k] = ls ∧ f[i] ≥ g[k] then go to fail else
     if M[i, k] = eq ∧ f[i] ≠ g[k] then go to fail
   end;
   for j := k1 step −1 until 1 do
   if M[i, j] = ls ∧ f[i] ≥ g[j] then fixcol (i, j, 1) else
   if M[i, j] = eq ∧ f[i] ≠ g[j] then fixcol (i, j, 0)
 end fixrow;
 procedure fixcol (l, j, x); value l, j, x; integer l, j, x;
 begin integer i; g[j] := f[l] + x;
   if k ≠ k1 then
   begin if M[k,j] = gr ∧ f[k] ≤ g[j] then go to fail else
     if M[k,j] = eq ∧ f[k] ≠ g[j] then go to fail
   end;
   for i := k step −1 until 1 do
   if M[i, j] = gr ∧ f[i] ≤ g[j] then fixrow (i, j, 1) else
   if M[i, j] = eq ∧ f[i] ≠ g[j] then fixrow (i, j, 0)
 end fixcol;
 k1 := 0;
 for k := 1 step 1 until n do
 begin fmin := 1;
   for j := 1 step 1 until k1 do
     if M[k, j] = gr ∧ fmin ≤ g[j] then fmin := g[j]+1 else
     if M[k, j] = eq ∧ fmin < g[j] then fmin := g[j];
   f[k] := fmin;
   for j := k1 step −1 until 1 do
     if M[k, j] = ls ∧ fmin ≥ g[j] then fixcol (k, j, 1) else
     if M[k, j] = eq ∧ fmin > g[j] then fixcol (k, j, 0);
   k1 := k1+1; gmin := 1;
   for i := 1 step 1 until k do
     if M[i, k] = ls ∧ f[i] ≥ gmin then gmin := f[i]+1 else
     if M[i, k] = eq ∧ f[i] > gmin then gmin := f[i];
   g[k] := gmin;
   for i := k step −1 until 1 do
     if M[i, k] = gr ∧ f[i] ≤ gmin then fixrow (i, k, 1) else
     if M[i, k] = eq ∧ f[i] < gmin then fixrow (i, k, 0)
 end k
end Precedence
```

ALGORITHM 266
PSEUDO-RANDOM NUMBERS [G5]

M. C. PIKE AND I. D. HILL

(Recd. 15 Feb. 1965 and 6 July 1965)

Medical Research Council, London, England

```
real procedure random (a, b, y);
 real a, b; integer y;
comment random generates a pseudo-random number in the
 open interval (a, b) where a < b. The procedure assumes that
 integer arithmetic up to 3125 × 67108863 = 209715196875 is
 available. The actual parameter corresponding to y must be an
 integer identifier, and at the first call of the procedure its value
 must be an odd integer within the limits 1 to 67108863 inclusive.
 If a correct sequence is to be generated, the value of this inte-
 ger identifier must not be changed between successive calls of
 the procedure;
begin
 y := 3125 × y; y := y − (y÷67108864) × 67108864;
 random := y/67108864.0 × (b−a) + a
end random
```

Coveyou [2] showed that for multiplicative congruential methods of generating pseudorandom numbers, the correlation between successive numbers will be approximately the reciprocal of the multiplying factor. Greenberger [3] showed further that the factor should be considerably less than the square root of the modulus.

(*continued on next page*)

Revised Algorithms Policy • May, 1964

A contribution to the Algorithms department must be in the form of an algorithm, a certification, or a remark. Contributions should be sent in duplicate to the editor, typewritten double-spaced in capital and lower-case letters. Authors should carefully follow the style of this department, with especial attention to indentation and completeness of references. Material to appear in **boldface** type should be underlined in black. Blue underlining may be used to indicate *italic* type, but this is usually best left to the Editor.

An algorithm must be written in the ALGOL 60 Reference Language [*Comm. ACM 6* (Jan. 1963), 1-17], and normally consists of a commented procedure declaration. Each algorithm must be accompanied by a complete driver program in ALGOL 60 which generates test data, calls the procedure, and outputs test answers. Moreover, selected previously obtained test answers should be given in comments in either the driver program or the algorithm. The driver program may be published with the algorithm if it would be of major assistance to a user.

Input and output should be achieved by procedure statements, using one of the following five procedures (whose body is not specified in ALGOL): [see "Report on Input-Output Procedures for ALGOL 60," *Comm, ACM 7* (Oct. 1964), 628–629].

procedure *inreal* (*channel, destination*); **value** *channel*; **integer** *channel*; **real** *destination*; **comment** the number read from channel *channel* is assigned to the variable *destination*; . . . ;

procedure *outreal* (*channel, source*); **value** *channel, source*; **integer** *channel*; **real** *source*; **comment** the value of expression *source* is output to channel *channel*; . . . ;

procedure *ininteger* (*channel, destination*); **value** *channel*; **integer** *channel, destination*; . . . ;

procedure *outinteger* (*channel, source*); **value** *channel, source*; **integer** *channel, source*; . . . ;

procedure *outstring* (*channel, string*); **value** *channel*; **integer** *channel*; **string** *string*; . . . ;

If only one channel is used by the program, it should be designated by 1. Examples:

```
outstring (1, 'x =');  outreal (1, x);
for i := 1 step 1 until n do outreal (1, A[i]);
ininteger (1, digit [17]);
```

It is intended that each published algorithm be a well-organized, clearly commented, syntactically correct, and a substantial contribution to the ALGOL literature. All contributions will be refereed both by human beings and by an ALGOL compiler. Authors should give great attention to the correctness of their programs, since referees cannot be expected to debug them. Because ALGOL compilers are often incomplete, authors are encouraged to indicate in comments whether their algorithms are written in a recognized subset of ALGOL 60 [see "Report on SUBSET ALGOL 60 (IFIP)," *Comm. ACM 7* (Oct, 1964), 626–627].

Certifications and remarks should add new information to that already published. Readers are especially encouraged to test and certify previously uncertified algorithms. Rewritten versions of previously published algorithms will be refereed as new contributions, and should not be imbedded in certifications or remarks.

Galley proofs will be sent to the authors; obviously rapid and careful proofreading is of paramount importance.

Although each algorithm has been tested by its author, no liability is assumed by the contributor, the editor, or the Association for Computing Machinery in connection therewith.

The reproduction of algorithms appearing in this department is explicitly permitted without any charge. When reproduction is for publication purposes, reference must be made to the algorithm author and to the *Communications* issue bearing the algorithm.—G.E.F.

The method of Algorithm 133 [1] satisfies Greenberger's condition, but since the reciprocal of its multiplying factor is as high as 0.2, Coveyou's result shows that it is very unsatisfactory for purposes requiring statistically independent consecutive random numbers.

Algorithms 133 and 266 have both been tested by computing a number of sets of 2000 successive random integers between 0 and 9, dividing each set into 400 groups of 5, and performing the poker test [4]. The results were classified in the following seven categories:

(*i*) all different
(*ii*) 1 pair
(*iii*) 2 pairs
(*iv*) 3 of a kind
(*v*) 3 of a kind and 1 pair
(*vi*) 4 of a kind
(*vii*) 5 of a kind.

The following tables resulted:

ALGORITHM 133

Run	*Starting Value*	(*i*)	(*ii*)	(*iii*)	(*iv*)	(*v*)	(*vi*)	(*vii*)
1	13421773	114	193	42	37	7	7	0
2	22369621	111	181	46	40	14	8	0
3	33554433	130	178	48	28	7	6	3
4	6871947673	118	179	51	35	10	5	2
5	11453246123	128	189	44	28	6	4	1
6	17179869185	135	155	45	52	6	5	2
Expected for each Run		120.96	201.60	43.20	28.80	3.60	1.80	0.04
Total for 6 Runs		736	1075	276	220	50	35	8
Expected for Total		725.76	1209.60	259.20	172.80	21.60	10.80	0.24

ALGORITHM 266

Run	*Starting Value*	(*i*)	(*ii*)	(*iii*)	(*iv*)	(*v*)	(*vi*)	(*vii*)
1	13421773	132	191	35	38	2	2	0
2	22369621	140	187	45	27	0	1	0
3	33554433	129	198	44	25	4	0	0
4	8426219	107	202	50	37	2	2	0
5	42758321	101	207	60	25	5	2	0
6	56237485	118	203	42	34	1	2	0
7	62104023	119	206	41	27	6	1	0
Expected for each Run		120.96	201.60	43.20	28.80	3.60	1.80	0.04
Total for 7 Runs		846	1394	317	213	20	10	0
Expected for Total		846.72	1411.20	302.40	201.60	25.20	12.60	0.28

Combining categories (*vi*) and (*vii*) in each case, the observed totals give χ^2 values (on 5 degrees of freedom) of 159.0 for Algorithm 133, and of 3.28 for Algorithm 266.

REFERENCES:

1. BEHRENZ, P. G. Algorithm 133, Random. *Comm. ACM 5* (NOV. 1962), 553.
2. COVEYOU, R. R. Serial correlation in the generation of pseudo-random numbers. *J. ACM 7*(1960), 72–74.
3. GREENBERGER, M. An a priori determination of serial correlation in computer generated random numbers. *Math. Comput. 15*(1961), 383–389. Correction in *Math. Comput.16*(1962), 126.
4. KENDALL, M. G., AND BABINGTON-SMITH, B. Randomness and random sampling numbers. *J. Royal Statist. Soc. 101* (1938), 147–166.

ALGORITHM 267
RANDOM NORMAL DEVIATE [G5]

M. C. PIKE (Recd. 3 May 1965 and 6 July 1965)
Medical Research Council, London, England


```
procedure RND(x1, x2, Random);
  real procedure Random; real x1, x2;
comment RND uses two calls of the real procedure Random
  which is any pseudo-random number generator which will
  produce at each call a random number lying strictly between 0
  and 1. A suitable procedure is given by Algorithm 266, Pseudo-
  Random Numbers [Comm. ACM 8(Oct. 1965), 605] if one chooses
  a = 0, b = 1 and initializes y to some large odd number, such as
  13421773. RND produces two independent random variables x1
  and x2 each from the normal distribution with mean 0 and
  variance 1. The method used is given by BOX, G.E.P., AND
  MULLER, M.E., A note on the generation of random normal
  deviates. [Ann. Math. Stat. 29 (1958), 610–611];
begin real t;
  x1 := sqrt(−2.0 × ln(Random));
  t := 6.2831853072 × Random;
  comment 6.2831853072 = 2 × pi;
  x2 := x1 × sin(t); x1 := x1 × cos(t)
end RND
```

Algorithm 121, NormDev [*Comm. ACM 5* (Sept. 1962), 482; *8* (Sept. 1965), 556] also produces random normal deviates and Algorithm 200, NORMAL RANDOM [*Comm. ACM 6* (Aug. 1963), 444; *8* (Sept. 1965), 556] produces random deviates with an approximate normal distribution, but the procedure *RND* seems preferable to both of them.

We may compare *NORMAL RANDOM* to *RND* (which is exact) by noting that at recommended minimum *n NORMAL RANDOM* requires 10 calls of *Random* while *RND* gets two independent normal deviates from 2 calls of *Random* and one call each of *sqrt*, *ln*, *sin* and *cos*. Under the stated test conditions a single call of *NORMAL RANDOM* (with $n = 10$) took 20 percent more computing time than a single call of *RND* when the real procedure *Random* was given by Algorithm 266.

To compare *NormDev* to *RND* in the same way, we have first to calculate the expected number of calls of *ln*, *sqrt*, *exp* and *Random* for each call of *NormDev*. This may be done by noting that there is (1) an initial single call of *Random*, then (2) with probability 0.68 a random normal deviate restricted to (0, 1) has to be found and this requires on average 1.36 calls of *Random* and 1.18 calls of *exp*, and (3) with probability 0.32 a random normal deviate restricted to $(1, \infty)$ has to be found and this requires on average 2.04 calls of *Random* and 1.52 calls of each of *ln* and *sqrt*. *NormDev* thus requires on average 2.58 calls of *Random*, 0.80 calls of *exp*, 0.49 calls of *ln* and 0.49 calls of *sqrt*. (Note: *NormDev* requires one further call of *Random* if a signed normal deviate is required.) Under the stated test conditions a single call of *NormDev* took virtually the same amount of computing time as a single call of *RND* when the real procedure *Random* was as above.

(Note: In testing *NormDev* the procedure was speeded up by replacing *A* by 0.6826894 wherever it occurred and removing it from the parameter list. In testing *NORMAL RANDOM Mean, Sigma, n* were replaced by 0, 1.0 and 10 respectively and removed from the parameter list.)