accuracy. The **real procedure** *gauss* computes the area under the left-hand portion of the normal curve. Algorithm 209 [3] may be used for this purpose. If $f < 0$ or if $df1 < 1$ or if $df2 < 1$ then exit to the label *error* occurs.

National Bureau of Standards formulas 26.6.4, 26.6.5, and 26.6.8 are used for computation of the statistic, and 26.6.15 is used for the approximation [2].

Thanks to Mary E. Rafter for extensive testing of this procedure and to the referee for a number of suggestions.

REFERENCES:

1. DORRER, EGON. Algorithm 322, F-Distribution. *Comm. ACM 11* (Feb. 1968), 116–117.
2. *Handbook of Mathematical Functions.* National Bureau of Standards, Appl. Math. Ser. Vol., 55, Washington, D.C., 1965, pp. 946–947.
3. IBBETSON, D. Algorithm 209, Gauss. *Comm. ACM 6* (Oct. 1963), 616.
4. SNEDECOR, GEORGE W. *Statistical Methods.* Iowa State U. Press, Ames, Iowa, 1956, pp. 244–250;

```
begin
  if df1 < 1 ∨ df2 < 1 ∨ f < 0.0 then go to error;
  if f = 0.0 then prob := 1.0
  else
  begin
    real f1, f2, x, ft, vp;
    f1 := df1; f2 := df2; ft := 0.0;
    x := f2/(f2+f1×f); vp := f1 + f2 − 2.0;
    if 2 × (df1÷2) = df1 ∧ df1 ≦ maxn then
    begin
      real xx; xx := 1.0 − x;
      for f1 := f1 − 2.0 step − 2.0 until 1.0 do
      begin
        vp := vp − 2.0;
        ft := xx × vp/f1 × (1.0+ft)
      end;
      ft := x ↑ (0.5×f2) × (1.0+ft)
    end
    else if 2 × (df2 ÷ 2) = df2 ∧ df2 ≦ maxn then
    begin
      for f2 := f2 − 2.0 step − 2.0 until 1.0 do
      begin
        vp := vp − 2.0;
        ft := x × vp/f2 × (1.0+ft)
      end;
      ft := 1.0 − (1.0−x) ↑ (0.5×f1) × (1.0+ft)
    end
    else if df1 + df2 ≦ maxn then
    begin
      real theta, sth, cth, sts, cts, a, b, xi, gamma;
      theta := arctan(sqrt(f1×f/f2));
      sth := sin(theta); cth := cos(theta);
      sts := sth ↑ 2; cts: = cth ↑ 2;
      a := b := 0.0;
      if df2 > 1 then
      begin
        for f2 := f2 − 2.0 step − 2.0 until 2.0 do
          a := cts × (f2−1.0)/f2 × (1.0+a);
        a := sth × cth × (1.0+a)
      end;
      a := theta + a;
      if df1 > 1 then
      begin
        for f1 := f1 − 2.0 step − 2.0 until 2.0 do
        begin
          vp := vp − 2.0;
          b := sts × vp/f1 × (1.0+b)
        end;
        gamma := 1.0; f2 := 0.5 × df2;
        for xi := 1.0 step 1.0 until f2 do
          gamma := xi × gamma/(xi−0.5);
        b := gamma × sth × cth ↑ df2 × (1.0+b)
      end;
      ft := 1.0 + 0.636619772368 × (b−a);
      comment 0.63661977236758134307553505 ··· = 2.0/π;
    end
    else
    begin
      real cbrf;
      f1 := 2.0/(9.0 × f1); f2 := 2.0/(9.0×f2);
      cbrf := f ↑ 0.333333333333;
      ft := gauss(−((1.0−f2)×cbrf+f1−1.0)/
        sqrt(f2×cbrf ↑ 2+f1))
    end;
    prob := if ft < 0.0 then 0.0 else ft
  end
end Ftest
```

ALGORITHM 347
AN EFFICIENT ALGORITHM FOR SORTING WITH MINIMAL STORAGE [M1]

RICHARD C. SINGLETON* (Recd. 17 Sept. 1968)
Mathematical Statistics and Operations Research Department, Stanford Research Institute, Menlo Park, CA 94025

* This work was supported by Stanford Research Institute with Research and Development funds.

KEY WORDS AND PHRASES: sorting, minimal storage sorting, digital computer sorting
CR CATEGORIES: 5.31

```
procedure SORT(A, i, j);
  value i, j; integer i, j;
  array A;
comment This procedure sorts the elements of array A into
  ascending order, so that
```

$$A[k] \leq A[k+1], \qquad k = i, i+1, \cdots, j-1.$$

The method used is similar to *QUICKERSORT* by R. S. Scowen [5], which in turn is similar to an algorithm given by Hibbard [2, 3] and to Hoare's *QUICKSORT* [4]. *QUICKERSORT* is used as a standard, as it was shown in a recent comparison to be the fastest among four ACM algorithms tested [1]. On the Burroughs B5500 computer, the present algorithm is about 25 percent faster than *QUICKERSORT* when tested on random uniform numbers (see Table I) and about 40 percent faster on numbers in natural order $(1, 2, \cdots, n)$, in reverse order $(n, n-1, \cdots, 1)$, and sorted by halves $(2, 4, \cdots, n, 1, 3, \cdots, n-1)$. *QUICKERSORT* is slow in sorting data with numerous "tied" observations, a problem that can be corrected by changing the code to exchange elements $a[k] \geq t$ in the lower segment with elements $a[q] \leq t$ in the upper segment. This change gives a better split of the original segment, which more than compensates for the additional interchanges.

In the earlier algorithms, an element with value t was selected from the array. Then the array was split into a lower segment with all values less than or equal to t and an upper segment with all values greater than or equal to t, separated by a third segment of length one and value t. The method was then applied

TABLE I. SORTING TIMES IN SECONDS FOR *SORT* AND *QUICKERSORT*, ON THE BURROUGHS B5500 COMPUTER—AVERAGE OF FIVE TRIALS

Original order and number of items	*Algorithm* *SORT*	*QUICKERSORT*
Random uniform:		
500	0.48	0.63
1000	1.02	1.40
Natural order:		
500	0.29	0.48
1000	0.62	1.00
Reverse order:		
500	0.30	0.51
1000	0.63	1.08
Sorted by halves:		
500	0.73	1.15
1000	1.72	2.89
Constant value:		
500	0.43	10.60
1000	0.97	41.65

recursively to the lower and upper segments, continuing until all segments were of length one and the data were sorted. The present method differs slightly—the middle segment is usually missing—since the comparison element with value t is not removed from the array while splitting. A more important difference is that the median of the values of $A[i]$, $A[(i+j)\div 2]$, and $A[j]$ is used for t, yielding a better estimate of the median value for the segment than the single element used in the earlier algorithms. Then while searching for a pair of elements to exchange, the previously sorted data (initially, $A[i] \leq t \leq A[j]$) are used to bound the search, and the index values are compared only when an exchange is about to be made. This leads to a small amount of overshoot in the search, adding to the fixed cost of splitting a segment but lowering the variable cost. The longest segment remaining after splitting a segment of n has length less than or equal to $n - 2$, rather than $n - 1$ as in *QUICKERSORT*.

For efficiency, the upper and lower segments after splitting should be of nearly equal length. Thus t should be close to the median of the data in the segment to be split. For good statistical properties, the median estimate should be based on an odd number of observations. Three gives an improvement over one and the extra effort involved in using five or more observations may be worthwhile on long segments, particularly in the early stages of a sort.

Hibbard [3] suggests using an alternative method, such as Shell's [6], to complete the sort on short sequences. An experimental investigation of this idea using the splitting algorithm adopted here showed no improvement in going beyond the final stage of Shell's algorithm, i.e. the familiar "sinking" method of sorting by interchange of adjacent pairs. The minimum time was obtained by sorting sequences of 11 or fewer items by this method. Again the number of comparisons is reduced by using the data themselves to bound the downward search. This requires

$$A[i-1] \leq A[k], \quad i \leq k \leq j.$$

Thus the initial segment cannot be sorted in this way. The initial segment is treated as a special case and sorted by the splitting algorithm. Because of this feature, the present algorithm lacks the pure recursive structure of the earlier algorithms.

For n elements to be sorted, where $2^k \leq n < 2^{k+1}$, a maximum of k elements each are needed in arrays *IL* and *IU*. On the B5500 computer, single-dimensional arrays have a maximum length of 1023. Thus the array bounds [0:8] suffice.

This algorithm was developed as a FORTRAN subroutine, then translated to ALGOL. The original FORTRAN version follows:

```
      SUBROUTINE SORT(A,II,JJ)
C  SORTS ARRAY A INTO INCREASING ORDER, FROM A(II) TO A(JJ)
C  ORDERING IS BY INTEGER SUBTRACTION, THUS FLOATING POINT
C    NUMBERS MUST BE IN NORMALIZED FORM.
C  ARRAYS IU(K) AND IL(K) PERMIT SORTING UP TO 2**(K+1)-1 ELEMENTS
      DIMENSION A(1),IU(16),IL(16)
      INTEGER A,T,TT
      M=1
      I=II
      J=JJ
    5 IF(I .GE. J) GO TO 70
   10 K=I
      IJ=(J+I)/2
      T=A(IJ)
      IF(A(I) .LE. T) GO TO 20
      A(IJ)=A(I)
      A(I)=T
      T=A(IJ)
   20 L=J
      IF(A(J) .GE. T) GO TO 40
      A(IJ)=A(J)
      A(J)=T
      T=A(IJ)
      IF(A(I) .LE. T) GO TO 40
      A(IJ)=A(I)
      A(I)=T
      T=A(IJ)
      GO TO 40
   30 A(L)=A(K)
      A(K)=TT
   40 L=L-1
      IF(A(L) .GT. T) GO TO 40
      TT=A(L)
   50 K=K+1
      IF(A(K) .LT. T) GO TO 50
      IF(K .LE. L) GO TO 30
      IF(L-I .LE. J-K) GO TO 60
      IL(M)=I
      IU(M)=L
      I=K
      M=M+1
      GO TO 80
   60 IL(M)=K
      IU(M)=J
      J=L
      M=M+1
      GO TO 80
   70 M=M-1
      IF(M .EQ. 0) RETURN
      I=IL(M)
      J=IU(M)
   80 IF(J-I .GE. 11) GO TO 10
      IF(I .EQ. II) GO TO 5
      I=I-1
   90 I=I+1
      IF(I .EQ. J) GO TO 70
      T=A(I+1)
      IF(A(I) .LE. T) GO TO 90
      K=I
  100 A(K+1)=A(K)
      K=K-1
      IF(T .LT. A(K)) GO TO 100
      A(K+1)=T
      GO TO 90
      END
```

This FORTRAN subroutine was tested on a CDC 6400 computer. For random uniform numbers, sorting times divided by $n \log_2 n$ were nearly constant at 20.2×10^{-6} for $100 \leq n \leq 10{,}000$, with a time of 0.202 seconds for 1000 items. This subroutine was also hand-compiled for the same computer to produce a more efficient machine code. In this version the constant of proportionality was 5.2×10^{-6}, with a time of 0.052 seconds for 1000 items. In both cases, integer comparisons were used to order normalized floating-point numbers.

REFERENCES:

1. BLAIR, CHARLES R. Certification of algorithm 271. *Comm. ACM 9* (May 1966), 354.
2. HIBBARD, THOMAS N. Some combinatorial properties of certain trees with applications to searching and sorting. *J. ACM 9* (Jan. 1962), 13–28.
3. HIBBARD, THOMAS N. An empirical study of minimal storage sorting. *Comm. ACM 6* (May 1963), 206–213.
4. HOARE, C. A. R. Algorithms 63, Partition, and 64, Quicksort. *Comm. ACM 4* (July 1961), 321.
5. SCOWEN, R. S. Algorithm 271, Quickersort. *Comm. ACM 8* (Nov. 1965), 669.
6. SHELL, D. L. A high speed sorting procedure. *Comm. ACM 2* (July 1959), 30–32;

```
begin
  real t, tt;
  integer ii, ij, k, L, m;
  integer array IL, IU[0:8];
  m := 0; ii := i; go to L4;
L1: ij := (i+j) ÷ 2; t := A[ij]; k := i; L := j;
  if A[i] > t then
    begin A[ij] := A[i]; A[i] := t; t := A[ij] end;
  if A[j] < t then
  begin
    A[ij] := A[j]; A[j] := t; t := A[ij];
    if A[i] > t then
      begin A[ij] := A[i]; A[i] := t; t := A[ij] end
  end;
L2: L := L - 1;
  if A[L] > t then go to L2;
    tt := A[L];
L3: k := k + 1;
  if A[k] < t then go to L3;
  if k ≤ L then
    begin A[L] := A[k]; A[k] := tt; go to L2 end;
  if L - i > j - k then
    begin IL[m] := i; IU[m] := L; i := k end
  else
    begin IL[m] := k; IU[m] := j; j := L end;
  m := m + 1;
L4: if j - i > 10 then go to L1;
    if i = ii then
    begin if i < j then go to L1 end;
    for i := i + 1 step 1 until j do
    begin
    t := A[i]; k := i - 1;
    if A[k] > t then
    begin
L5: A[k+1] := A[k]; k := k - 1;
    if A[k] > t then go to L5;
    A[k+1] := t
    end
  end;
  m := m - 1; if m ≥ 0 then
    begin i := IL[m]; j := IU[m]; go to L4 end
  end SORT
```

REMARK ON ALGORITHM 329 [G6] DISTRIBUTION OF INDISTINGUISHABLE OBJECTS INTO DISTINGUISHABLE SLOTS [Robert R. Fenichel, *Comm. ACM 11* (June 1968), 430]

M. Gray (Recd. 20 Sept. 1968)
Computing Science Department, University of Adelaide, South Australia

As the procedure stands it is incorrect. Preceding

end skip 99,189,198, etc.

the following statement should be inserted:

if *q*[*k*] ≠ 0 **then** *LeftmostZero* := *k* + 1

Thus the compound statement becomes:

```
begin
  LeftmostZero := LeftmostZero -1;
  q[k] := q[LeftmostZero] - 1;
  q[LeftmostZero] := 0;
  q[LeftmostZero-1] := q[LeftmostZero-1] + 1;
  if q[k] ≠ 0 then LeftmostZero := k + 1
end skip 99, 189, 198, etc.
```

REMARK ON ALGORITH 339 [C6] AN ALGOL PROCEDURE FOR THE FAST FOURIER TRANSFORM WITH ARBITRARY FACTORS [Richard C. Singleton, *Comm. ACM 11* (Nov. 1968), 776]

Richard C. Singleton (Recd. 27 Nov. 1968)
Stanford Research Institute, Menlo Park, CA 94025

KEY WORDS AND PHRASES: fast Fourier transform, complex Fourier transform, multivariate Fourier transform, Fourier series, harmonic analysis, spectral analysis, orthogonal polynomials, orthogonal transformation, virtual core memory, permutation
CR CATEGORIES. 3.15, 3.83, 5.12, 5.14

On page 778, column 2, the 7th and 6th lines from the bottom should be corrected to read:

```
LJ: jj := C[i-2] + jj; if jj ≧ C[i-1] then
      begin i := i - 1; jj := jj - C[i]; go to LJ end;
```

On page 779, column 1, the 9th and 8th lines from the bottom should be corrected to read:

```
LX: jj := D[k+1] + jj; if jj ≧ D[k] then
      begin jj := jj - D[k]; k := k + 1; go to LX end;
```

In both cases *jj* was originally used as the controlled variable of a for clause and thus was undefined after exit; the corrections preserve the value of *jj* for later use.

If the user prefers to compute constants with library functions, line 5 in column 2 on page 777 may be replaced by:

```
rad := 8.0 × arctan(1.0); c30 := sqrt(0.75);
```

Algorithms 338 [*Comm. ACM 11* (Nov. 1968), 773] and 339 were punched from the printed page and tested on the CDC 6400 Algol compiler. After changing a colon to a semicolon at the end of line 37 in column 2 on page 775, the test results agreed with those obtained earlier with this compiler.

When computing a single-variate Fourier transform of real data, procedure *REALTRAN* may be used with procedure *FFT* (Algorithm 339) to reduce computing time. Two versions of *REALTRAN* have been given (Algorithms 338 and 345 [*Comm. ACM 12* (Mar. 1969), 179–184]); the first version is the faster of the two, but the second should be used if arithmetic results for real quantities are truncated rather than rounded.

In describing the evaluation of a real Fourier series, in the middle of column 2 on page 776, the necessary steps of reversing the signs of the *B* array values both before and after calling *FFT* were omitted. The correct steps, including scaling, are as follows:

```
REALTRAN(A B, n, true);
for j := n - 1 step -1 until 0 do B[j] := -B[j];
FFT(A, B, n, n, n);
for j := n - 1 step -1 until 0 do
  begin A[j] := 0.5 × A[j]; B[j] := -0.5 × B[j] end;
```

The policy concerning the contributions of algorithms to *Communications of the ACM* appears, most recently, in the January 1969 issue, page 39. A contribution should be in the form of an algorithm, a certification, or a remark. An algorithm must normally be written in the ALGOL 60 Reference Language or in USASI Standard FORTRAN or Basic FORTRAN.