Technical Note: Efficient sorting routines in FORTRAN 77

G. T HOULSBY and S. W. SLOAN

Department of Engineering Science, Parks Road, University of Oxford, Oxford OX1 3PJ, England

A problem faced which arises in many computing applications is that of sorting the entries in a list into ascending sequence of their values. Efficient algorithms for sorting long lists usually require recursive techniques and are not straightforward to program in a non-recursive language such as FORTRAN 77 This paper describes three FORTRAN 77 implementations of the insertion sort, quicksort and modified quicksort algorithms. The first algorithm is suitable for short lists whilst the second two are suitable for longer lists. A detailed description of the removal of recursion from the quicksort algorithm by the use of stacks is presented. All of the implementations are believed to be efficient, and should prove useful in engineering applications. Some timing statistics are given to illustrate the utility of the subroutines presented.

INTRODUCTION

The problem of sorting lists of numbers into ascending sequence is of considerable importance in computing, and a significant amount of literature has been published on the topic Of the many internal sorting algorithms that have been devised, it is generally accepted that the most efficient method for short lists is the insertion sort [1-3] For longer lists, however, the quicksort algorithm is acknowledged to be an efficient internal sorting method [1-5] Other internal sorting methods (e g shellsort) are not discussed as they are rarely as efficient as the techniques presented here In this paper it is not intended to review the merits of various procedures, which have been amply dealt with elsewhere, but rather to describe FORTRAN 77 implementations of the insertion sort and quicksort algorithms A third algorithm, which combines the best features of the insertion sort and quicksort methods, is also discussed

The quicksort algorithm is by its nature recursive, and is often explicitly programmed using recursive subroutine calls Since this is not allowed in FORTRAN 77 an alternative method must be used, and, in the algorithm described here, explicit recursion is avoided by use of a stack of pointers Even in a language which allows recursion the use of a stack is not avoided, it is simply that the stack is handled by the compiler rather than the programmer For the quicksort algorithm this has the disadvantage in the quicksort application that in the worst case the height of the internal stack can be proportional to the length of the list to be sorted, and overflow problems could be encountered By programming the implementation of the stack explicitly it is possible to avoid this worst case, and ensure that the stack height is proportional only to the logarithm of the list length

A third algorithm which makes use of the best features of the insertion sort and quicksort is also included, and this perhaps requires some discussion Quicksort works by successively dividing the list into two sub-lists, one list containing items larger than or equal to some cut-off value and the other containing items smaller than or equal to the cut-off value The algorithm is then re-applied to each sub-list until each sub-list is a single item This process becomes rather inefficient once the sub-lists become short, and the overall efficiency of the algorithm can be improved by leaving sub-lists shorter than a certain length (say 12 items) unsorted This results in a list which is almost sorted, in that each item is close (i e within 12 places) of its final position in the fully sorted list This partially sorted list is then sorted by a second pass with the insertion sort algorithm, which is very efficient for an almost sorted list This approach (called here a 'hybrid' sort), in which a single pass of the insertion sort is used to post-process the almost sorted list, is slightly more efficient than the alternative of applying the insertion sort separately to each short sub-list [3]

The sorting routines described in this paper all use the method of sorting a pointer to a series of keys The key values to be sorted are stored in a vector KEY, and the pointers in an INTEGER vector LIST, which is of length N Each of the N entries in LIST gives the number of an item in KEY, e g if LIST(3) = 6 then the third item in the list of keys to be sorted is KEY(6) After sorting, the entries in LIST are re-ordered to point to the items in KEY in an ascending sequence Note that KEY will usually, but not always, be of length N In principal KEY could be shorter than N (and more than one item in LIST point to the same item in KEY) or longer, in which case some items would not be referenced The advantage of the above method of sorting is that only the pointers are changed, and not the items in the actual list of keys

If KEY is of length N and all the items are to be sorted, then the subroutines given here will usually be preceded by the code fragment

```
      DO 1 I = 1, N
         LIST(I) = I
    1 CONTINUE
```

which simply initialises the list of pointers If the keys themselves are to be re-ordered then the subroutines would be followed by the code fragment

Accepted June 1984 Written discussion closes December 1984

0141-1195/84/040198-06 $2 00
© 1984 CML Publications

```
      DO 2 I = 1, N
         TEMP(I) = KEY(I)
    2 CONTINUE
      DO 3 I = 1, N
         KEY (I) = TEMP(LIST(I))
    3 CONTINUE
```

where TEMP is a temporary array or the same length and type of KEY

All three subroutines are given in versions to sort keys of type DOUBLE PRECISION To change to REAL or INTEGER keys, all that is required is to change each declaration of DOUBLE PRECISION to REAL or INTEGER The routines may also easily be modified to sort CHARACTER variables, although some attention is then necessary to ensure that appropriate lengths of character strings are compared If sorting in descending rather than ascending order is required it is recommended that the present routines are used, followed by a simple post-processing to reverse the order Internal alterations to the routines to achieve a reverse order (e.g changing all occurrences of GT to LT etc) should be treated with caution

All three routines have been rigorously tested using lists of random numbers, lists already in correct sequence, lists in reverse sequence, lists containing repeated identical entries and very short lists (of one or two entries) They are believed to work in all these circumstances, but of course cannot in any way be guaranteed by the authors The code has been written in standard FORTRAN 77,[6] with all variables declared explicitly at the beginning of the routines and indentation used to reveal the structure of the subroutines Some CONTINUE statements are included for clarity of presentation, and may be removed for optimum efficiency if a non-optimising compiler is used The authors would welcome any comments on both the clarity and efficiency of the routines

DESCRIPTION OF FORTRAN 77 CODE

The FORTRAN 77 code for the insertion sort, quicksort and hybrid sort is given in Appendices 1, 2 and 3, and descriptions of all the variables and parameters used in the code are given in Table 1 Some additional information is given in the comment statements at the beginning of each subroutine

Insertion sort routine

The insertion sort routine begins by finding the minimum value of KEY and placing this at the beginning of the list This pre-processing is necessary to provide a stopping point for the backward search which is carried out in the main part of the insertion sort It can be avoided only by adding a sentinel element, which must be set to a large negative number, at the beginning of the list of keys as the zeroth element This procedure can, however, be inconvenient in many applications Another alternative is to put an extra conditional statement in the backward search loop, so that the search stops once the beginning of the list is reached This latter procedure is less efficient since the extra statement is executed approximately $N^2/2$ times whilst the loop to search for the minimum value is executed only $N-1$ times

The main body of the insertion sort is carried out as follows After placing the minimum item at the beginning of the list, each of the remaining items from 2 to N is added in turn Consider the addition of item I ($2 < I < N$) The item is first placed at position I, and its value compared with the value of item I − 1 If it is less than item I − 1 then these two items are swapped, and the new item I − 1 (the original item I) now compared with item I − 2 The backward search continues until the newly added item is greater than or equal to the item with which it is compared At this stage the search terminates and a new item I + 1 is added to the list in a similar manner The process continues until all items on the list have been added

Table 1 Variables and parameters used in subroutines

Variable/ Parameter	Type	Routines*	Description
N	INTEGER	I, Q, H	Number of values to be sorted
LIST	INTEGER array (of length N)	I, Q, H	List of pointers to entries in array KEY, input in any order (e g LIST (I) = I) and output pointing to entries in KEY in ascending order
KEY	DOUBLE PRECISION array	I, Q, H	Values to be sorted in ascending order
I	INTEGER	I, H	DO loop count
J	INTEGER	I, H	Temporary used in backward search
K	INTEGER	I, H	Temporary store for LIST element
VALUE	DOUBLE PRECISION	I, H	Temporary value of KEY element
MAXSTK	INTEGER	Q, H	Parameter giving maximum stack size (set to 32 in the version given, which is adequate for N up to approximately 10^9)
LL	INTEGER	Q, H	Pointer to left end of sub-list
LR	INTEGER	Q, H	Pointer to right end of sub-list
LM	INTEGER	Q, H	Pointer to middle of sub-list
NL	INTEGER	Q, H	Pointer to left key to be exchanged
NR	INTEGER	Q, H	Pointer to right key to be exchanged
LTEMP	INTEGER	Q, H	Temporary used in list swapping
STKTOP	INTEGER	Q, H	Current size of stacks
LSTACK	INTEGER array (of length MAXSTK)	Q, H	Stack of pointers to left ends of sub-lists
RSTACK	INTEGER array (of length MAXSTK)	Q, H	Stack of pointers to right ends of sub-lists
GUESS	DOUBLE PRECISION	Q, H	Guess of median value for sub-list
NCUT	INTEGER	H	Cut-off length of sub-list below which quicksort is not used

* I = ISORT, Q = QSORT, H = HSORT

Quicksort routine

Two main aspects of the quicksort routine must be described, firstly the implementation of the stack, and secondly the details of the algorithm for swapping items and forming new sub-lists

The stack system is for convenience implemented in the form of two stacks LSTACK and RSTACK, which contain pointers to the left- and right-hand ends of each sub-list to be processed The INTEGER variable STKTOP points to the top of the stack The variables LL and LR are used as pointers to the left- and right-hand ends of the sub-list currently being processed When the routine is entered LL is set to 1 and LR to N so that the current sub-list is the whole list, and STKTOP is set to zero If LL is less than LR (i e the sub-list is longer than a single item) then the sub-list is processed in the IF-block following label 10 During this process the sub-list is broken into two new sub-lists In the lines following 'SELECT SUB-LIST TO BE PROCESSED NEXT', LL and LR are reset to point to the *shorter* of the new sub-lists, and pointers to the *longer* sub-list are put on the stack It is this system of always processing the shorter sub-list next which ensures that the height of the stack need only be of the order of $\log_2 N$ In the last eight lines of the program the stack is examined and if there are any sub-lists stacked then LL and LR are set accordingly, the stack height decremented and the main body of the routine re-entered

The system for swapping items and forming new sub-lists is as follows Movable pointers NL and NR are first set to point to the ends of the sub-list LL to LR, and LM is set to point approximately to the middle of the list The value of GUESS is then set to that of the item in the LM position (This procedre has the advantage that it capitalises on any partial ordering of the list) Items equal or less than GUESS will be end up in the first sub-list and those equal or greater in the second sub-list (Note that this means that an item equal to GUESS can appear in *either* sub-list)

In the IF-blocks after labels 20 and 30, NL and NR are respectively incremented and decremented until items are found which should be in the other sub-list Provided that NL is less than NR − 1, the swapping is then carried out and the search continued for new items to be swapped As this process continues NL and NR approach each other and will eventually cross Once NL is greater than NR − 2, the crossing of the pointers is handled in the lines following the comment 'DEAL WITH CROSSING OF POINTERS'

It is only possible to reach this point with NL equal to NR − 1, NR or NR + 1 In the first instance the items must be swapped, NL is incremented to point to the left-hand end of the new *right* sub-list and NR decremented to point to the right hand end of the new *left* sub-list The second instance is only possible with the NL (and NR) item equal to GUESS, and in this case NL and NR are incremented and decremented to point to the ends of sub-lists as above, but excluding the NL (or NR) item which has now been sorted to its final position In the third case no action is necessary as NL and NR already point to the appropriate items

The shorter of the sub-lists is then selected to be processed next by resetting the appropriate LL or LR value, and the pointers to the ends of the longer sub-list put on the stack and the stack top pointer incremented

Hybrid sort routine

The hybrid routine takes advantage of the best features of both the insertion and quicksort routines The first part of the routine is identical to the quicksort routine described above, except that when checking the length of each sub-list the quicksort algorithm is only used if the length of the list is greater than NCUT, whereas in quicksort it is used for any sub-list of length greater than one item The net result is that after applying the first part of the routine to the list it will consist of a partially sorted list containing unsorted segments of length not greater than NCUT

The routine then takes a second pass through the list, this time applying the insertion sort routine described above, which acts efficiently on the partially sorted list In sorting the small unordered sub-lists of approximate length NCUT, the backward search in the insertion sort is never required to step through more than NCUT items For values of NCUT of the order of 12, this operation is usually faster than dividing the short lists further and applying quicksort to each of them The choice of NCUT as 12 is somewhat arbitrary, as the improvement over the quicksort algorithm is approximately the same for NCUT values from about 10 to 30 The best value of NCUT is close to, but not necessarily the same as, the length of list N for which the quicksort and insertion sort take the same time

Note also that, in the pre-processing stage for the insertion sort, it is only necessary to search for the minimum in the first NCUT items, since the list has already been partially ordered (For lists in which $N <$ NCUT a search through only N items is required)

TIMING STATISTICS

Table 2 gives the timing statistics for the three subroutines to sort N randomly generated numbers The figures (which are averages for many runs) are in CPU seconds on a VAX 11/780 machine and the code was compiled using an optimising compiler These statistics would, of course, vary from one machine to another, but the general trends would be expected to be the same Although the VAX 11/780 is not a particularly fast machine, the quicksort and hybrid sort implementations are capable of sorting 10 000 items in less than 3 s

As expected, the time required for the insertion sort increases approximately as N^2 (and for the VAX 11/780 was approximately $3\ 8 \times 10^{-6} N^2$ CPU seconds) Although this is rather faster than the quicksort routine for very short lists, it is less efficient than quicksort once N is greater than about 20 The CPU time for the simple quicksort routine is approximately proportional to $N \log_2 N$ ($2\ 1 \times 10^{-5} N \log_2 N$ CPU seconds on the VAX 11/780) The hybrid sorting method takes advantage of the best features of both the insertion sort and the quicksort routines The time required increases approximately as $N \log_2 N$, but because the quicksort routine is not used on short sub-lists, it is consistently

Table 2 Timing statistics on VAX 11/780 (CPU seconds)

N	Insertion (ISORT)	Quicksort (QSORT)	Hybrid (HSORT)
10	0 00052	0 00126	0 00062
20	0 00187	0 00189	0 00169
50	0 00994	0 00609	0 00474
100	0 0367	0 0138	0 0105
200	0 156	0 0325	0 0289
500	0 883	0 0906	0 0762
1000	3 70	0 200	0 170
2000	–	0 442	0 389
5000	–	1 31	1 12
10 000	–	2 93	2 66

faster than the quicksort routine The proportional time saving decreases as N increases, but the absolute time saving over the quicksort routine increases in proportion to N (for N greater than about 50) and is given by about $2.9 \times 10^{-5} N$ for the VAX 11/780

CONCLUSIONS

Three FORTRAN 77 algorithms have been presented for sorting lists into ascending order The insertion sort routine is very simple to program and is efficient for lists up to a length of the order of 20 It may also be chosen for longer lists when ease of programming is more important than speed of computation The quicksort routine is more complex, and in a FORTRAN 77 implementation requires the use of a stack to avoid the need for recursion It is efficient for lists longer than about 20 items and would be the usual choice for sorting lists of widely varying lengths The third routine, called here a hybrid sort, works essentially in the same way as quicksort, but improves efficiency slightly by using an insertion sort for short sub-lists If efficiency is of overriding importance, this third routine would be the choice for most applications involving long lists

REFERENCES

1 Bentley, J L *Writing Efficient Programs*, Prentice-Hall, New Jersey, 1982
2 Dromey, R G *How to Solve it by Computer*, Prentice-Hall, New Jersey, 1982
3 Sedgewick, R *Algorithms*, Addison Wesley, Reading, Massachusetts, 1983
4 Aho, A V , Hopcroft, J E and Ullman, J D *Data Structures and Algorithms*, Addison Wesley, Reading, Massachusetts, 1983
5 Lewis, T G and Smith, M. Z *Applying Data Structures*, 2nd Edition, Houghton Mifflin, Boston, 1982
6 Balfour, A and Marwick, D H *Programming in Standard FORTRAN 77*, Heinemann, London, 1979

See overleaf for Appendixes with program listings

```
C     SUBROUTINE ISORT(N,LIST,KEY)
C     ORDER INTEGERS STORED IN 'LIST' IN ASCENDING SEQUENCE OF THEIR KEY
C     VALUES STORED IN 'KEY'
C
C     INPUT PARAMETERS:
C
C      N      POSITIVE INTEGER GIVING LENGTH OF LIST
C     'LIST' A LIST OF LENGTH 'N' OF INTEGERS
C      KEY   A LIST OF LENGTH 'N' OF DOUBLE PRECISION KEYS
C
C     OUTPUT PARAMETERS
C
C     'N'     UNCHANGED
C      LIST  A LIST OF OF LENGTH 'N' OF INTEGERS SORTED IN ASCENDING
C            SEQUENCE OF THEIR DOUBLE PRECISION KEYS
C      KEY   UNCHANGED
C
C     NOTES
C
C     USES INSERTION SORT - EFFICIENT ONLY FOR 'N' VALUES LESS THAN
C     ABOUT 12 (ALTHOUGH MAY BE SYSTEM DEPENDENT)
C*********************************************************************
      INTEGER N
      INTEGER LIST(*)
      DOUBLE PRECISION KEY(*)
C
      INTEGER I  J  K
      DOUBLE PRECISION VALUE
C
C     SET FIRST POINTER TO MINIMUM KEY
C
      J = 1
      K = LIST(1)
      VALUE = KEY(K)
      DO 10 I = 2  N
         IF (KEY(LIST(I))  LT  VALUE) THEN
            J = I
            VALUE = KEY(LIST(I))
         END IF
10    CONTINUE
      LIST(1) = LIST(J)
      LIST(J) = K
C
C     CARRY OUT INSERTION SORT
C
      DO 30 I = 2  N
         J = I
         K = LIST(I)
         VALUE = KEY(K)
20       CONTINUE
         IF (VALUE  LT  KEY(LIST(J-1))) THEN
            LIST(J) = LIST(J-1)
            J = J - 1
            GOTO 20
         END IF
         LIST(J) = K
30    CONTINUE
      RETURN
      END
```

APPENDIX 2

Quicksort routine

```
      SUBROUTINE QSORT(N,LIST,KEY)
C*********************************************************************
C     SUBROUTINE QSORT
C     ORDER INTEGERS STORED IN 'LIST' IN ASCENDING SEQUENCE OF THEIR KEY
C     VALUES STORED IN 'KEY
C
C     INPUT PARAMETERS:
C
C      N      POSITIVE INTEGER GIVING LENGTH OF LIST
C     'LIST' A LIST OF LENGTH 'N' OF INTEGERS
C      KEY   A LIST OF LENGTH 'N' OF DOUBLE PRECISION KEYS
C
C     OUTPUT PARAMETERS
C
C     'N      UNCHANGED
C      LIST  A LIST OF OF LENGTH 'N' OF INTEGERS SORTED IN ASCENDING
C            SEQUENCE OF THEIR DOUBLE PRECISION KEYS
C      KEY   UNCHANGED
C
C     NOTES
C
C     USES QUICKSORT ALGORITHM EFFICIENT FOR 'N' VALUES GREATER THAN
C     ABOUT 12 (ALTHOUGH MAY BE SYSTEM DEPENDENT)
C
C     ROUTINE SORTS LISTS UP TO LENGTH 2**MAXSTK
C*********************************************************************
      INTEGER N
      INTEGER LIST(*)
      DOUBLE PRECISION KEY(*)
C
      INTEGER MAXSTK
      PARAMETER(MAXSTK = 32)
      INTEGER LL, LR, LM, NL, NR, LTEMP, STKTOP
      INTEGER LSTACK(MAXSTK), RSTACK(MAXSTK)
      DOUBLE PRECISION GUESS
C
C     INITIALISE
C
      STKTOP = 0
      LL = 1
      LR = N
C
C     CHOOSE MIDDLE OF SEGMENT
C
      LM = (LL+LR)/2
      GUESS = KEY(LIST(LM))
C
C     FIND KEYS FOR EXCHANGE
C
20    CONTINUE
      IF (KEY(LIST(NL))  LT  GUESS) THEN
         NL = NL + 1
         GOTO 20
      END IF
30    CONTINUE
      IF (GUESS  LT  KEY(LIST(NR))) THEN
         NR = NR - 1
         GOTO 30
      END IF
      IF (NL  LT  (NR-1)) THEN
         LTEMP = LIST(NL)
         LIST(NL) = LIST(NR)
         LIST(NR) = LTEMP
         NL = NL + 1
         NR = NR - 1
         GOTO 20
      END IF
C
C     DEAL WITH CROSSING OF POINTERS
C
      IF (NL  LE  NR) THEN
         IF (NL  LT  NR) THEN
            LTEMP = LIST(NL)
            LIST(NL) = LIST(NR)
            LIST(NR) = LTEMP
         END IF
         NL = NL + 1
         NR = NR - 1
      END IF
C
C     SELECT SUB-LIST TO BE PROCESSED NEXT
C
      STKTOP = STKTOP + 1
      IF (NR  LT  LM) THEN
         LSTACK(STKTOP) = NL
         RSTACK(STKTOP) = LR
         LR = NR
      ELSE
         LSTACK(STKTOP) = LL
         RSTACK(STKTOP) = NR
         LL = NL
      END IF
      GOTO 10
   END IF
C
C  PROCESS ANY STACKED SUB-LISTS
C
   IF (STKTOP  NE  0) THEN
      LL = LSTACK(STKTOP)
      LR = RSTACK(STKTOP)
      STKTOP = STKTOP - 1
      GOTO 10
   END IF
   RETURN
   END
```

APPENDIX 3

Hybrid sort routine

```
      SUBROUTINE HSORT(N,LIST,KEY)
C*********************************************************************
C     SUBROUTINE HSORT
C     ORDER INTEGERS STORED IN  LIST  IN ASCENDING SEQUENCE OF THEIR KEY
C     VALUES STORED IN 'KEY
C
C     INPUT PARAMETERS:
C
C     'N      POSITIVE INTEGER GIVING LENGTH OF LIST
C     'LIST' A LIST OF LENGTH 'N' OF INTEGERS
C     'KEY'  A LIST OF LENGTH 'N' OF DOUBLE PRECISION KEYS
C
C     OUTPUT PARAMETERS
C
C      N      UNCHANGED
C      LIST  A LIST OF OF LENGTH 'N' OF INTEGERS SORTED IN ASCENDING
C            SEQUENCE OF THEIR DOUBLE PRECISION KEYS
C      KEY   UNCHANGED
C
C     NOTES
C
C     USES QUICKSORT ALGORITHM, EXCEPT FOR SEGMENTS OF THE LIST SHORTER
C     THAN  NCUT  FOR WHICH A SECOND PASS INSERTION SORT IS USED  'NCUT
C     SHOULD BE SET TO AN INTEGER OF THE ORDER OF 12 FOR MAXIMUM
C     EFFICIENCY, BUT MAY BE SYSTEM DEPENDENT
C
C     ROUTINE SORTS LISTS UP TO LENGTH 2**MAXSTK
C*********************************************************************
      INTEGER N
      INTEGER LIST(*)
      DOUBLE PRECISION KEY(*)
C
```

```
      INTEGER MAXSTK, NCUT
      PARAMETER(MAXSTK =
     +          NCUT + 1)
      INTEGER I1, LR, LM, NL, NR, LTEMP, STKTOP
      INTEGER LSTACK(MAXSTK), RSTACK(MAXSTK)
      DOUBLE PRECISION GUESS, VALUE

      I1 =
      LR = N
      STKTOP = 0
10    CONTINUE
      IF ((LR-I1) .GE. N) THEN

C        ONLY USE QUICKSORT IF THE LIST IS LONGER THAN N

         NL = I1
         NR = LR
         LM = (NL+NR) / 2
         GUESS = KEY(LIST(LM))
C
C        FIND KEYS FOR EXCHANGE
C
20       CONTINUE
         IF (KEY(LIST(NL)) .LT. GUESS) THEN
            NL = NL + 1
            GOTO 20
         END IF
30       CONTINUE
         IF (GUESS .LT. KEY(LIST(NR))) THEN
            NR = NR - 1
            GOTO 30
         END IF
         IF (NL .LT. NR) THEN
            LTEMP = LIST(NL)
            LIST(NL) = LIST(NR)
            LIST(NR) = LTEMP
            NL = NL + 1
            NR = NR - 1
            GOTO 20
         END IF
C
C        DEAL WITH CROSSING OF POINTERS
C
         IF (NL .LE. NR) THEN
            IF (NL .LT. NR) THEN
               LTEMP = LIST(NL)
               LIST(NL) = LIST(NR)
               LIST(NR) = LTEMP
            END IF
            NL = NL + 1
            NR = NR - 1
         END IF
C
```

```
         SELECT SUB-LIST TO BE PROCESSED NEXT

            IF (NL .GE. STKTOP) THEN
               IF (NR .GT. LM) THEN
                  LSTACK(STKTOP) = NL
                  RSTACK(STKTOP) = NR
                  I1 = NR
               ELSE
                  LSTACK(STKTOP) = NL
                  RSTACK(STKTOP) = NR
                  LR = NL
               END IF
               GOTO 10
            ELSE
               IF (STKTOP .EQ. 0) THEN
                  GOTO 10
               ELSE
                  I1 = LSTACK(STKTOP)
                  LR = RSTACK(STKTOP)
                  STKTOP = STKTOP - 1
                  GOTO 10
               END IF
            END IF

         END PASS - INSERTION SORT ON SUBLISTS WHICH HAVE MINIMUM KEY

         I1 = 1
         K = LIST(1)
         VALUE = KEY(K)
         DO 40 I = 2, MIN(N,NCUT)
            IF (KEY(LIST(I)) .LT. VALUE) THEN
               J = 1
               VALUE = KEY(LIST(I))
            END IF
40       CONTINUE
         LIST(1) = LIST(J)
         LIST(J) = K

         CARRY OUT INSERTION SORT

         DO 60 I = 2, N
            J = I
            K = LIST(I)
            VALUE = KEY(K)
50          CONTINUE
            IF (VALUE .LT. KEY(LIST(J-1))) THEN
               LIST(J) = LIST(J-1)
               J = J-1
               GOTO 50
            END IF
            LIST(J) = K
60       CONTINUE
         RETURN
         END
```