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 while 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. For longer lists, however, the quicksort algorithm is acknowledged to be an efficient internal sorting method. 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.

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 KEY 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

```fortran
DO 11 = 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.
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
care.

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, with all
variables declared explicitly at the beginning of the routines
and indentation used to reveal the structure of the sub-
routines. 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 mini-
umum 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 incon-
venient 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 \( N/2 \) 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 begin-
ing 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 com-
pared. 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

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

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

Adv Eng Software, 1984, Vol 6, No 4 199
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 log2 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 procedure 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 unsorted 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 in 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 N2 (and for the VAX 11/780 was approximately 3.8 x 10-6 N2 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 log2 N (2.1 x 10-8 N log2 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 log2 N, but because the quicksort routine is not used on short sub-lists, it is consistently

<table>
<thead>
<tr>
<th>N</th>
<th>Insertion (ISORT)</th>
<th>Quicksort (QSORT)</th>
<th>Hybrid (HSORT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.00052</td>
<td>0.00126</td>
<td>0.0062</td>
</tr>
<tr>
<td>20</td>
<td>0.00187</td>
<td>0.00189</td>
<td>0.00169</td>
</tr>
<tr>
<td>50</td>
<td>0.00994</td>
<td>0.00609</td>
<td>0.00474</td>
</tr>
<tr>
<td>100</td>
<td>0.0367</td>
<td>0.0138</td>
<td>0.0105</td>
</tr>
<tr>
<td>200</td>
<td>0.156</td>
<td>0.0325</td>
<td>0.0289</td>
</tr>
<tr>
<td>500</td>
<td>0.883</td>
<td>0.0906</td>
<td>0.0762</td>
</tr>
<tr>
<td>1000</td>
<td>3.70</td>
<td>0.200</td>
<td>0.170</td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td>0.442</td>
<td>0.389</td>
</tr>
<tr>
<td>5000</td>
<td></td>
<td>1.31</td>
<td>1.12</td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td>2.93</td>
<td>2.66</td>
</tr>
</tbody>
</table>
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

2. Dromey, R. G. *How to Solve it by Computer*, Prentice-Hall, New Jersey, 1982

See overleaf for Appendixes with program listings.
APPENDIX 2

Quicksort routine

SUBROUTINE QUOR(LIST, KEY)
C ORDER INTEGERS STORED IN 'LIST' IN ASCENDING SEQUENCE OF THEIR KEY
C VALUES STORED IN 'KEY'
C
C INPUT PARAMETERS:
C   N POSITIVE INTEGER GIVING LENGTHS 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   N UNCHANGED
C   LIST A LIST OF LENGTH N OF INTEGERS SORTED IN ASCENDING SEQUENCE OF THEIR DOUBLE PRECISION KEYS
C   KEY UNCHANGED
C
C NOTES
C USES QUICKSORT ALGORITHM, EFFICIENT ONLY FOR N VALUES GREATER THAN ABOUT 12 (ALTHOUGH MAY BE SYSTEM DEPENDENT)
C ROUTINE SORTS LISTS OF LENGTH 2**MxN

INTEGER N
INTEGER LIST(*)
DOUBLE PRECISION KEY(*)

C N INTEGER MAXKEY
PARAMETER MAXKEY = 32
INTEGER LL, LR, NW, NL, LTEMP, STKTOP
INTEGER LTEMPSTOP(MAXKEY), NSTACK(MAXKEY)
DOUBLE PRECISION GUESS

C B) Routines
C
C QUICKSORT ROUTINE
C
C ORDER INTEGERS STORED IN 'LIST' IN ASCENDING SEQUENCE OF THEIR KEY
C VALUES STORED IN 'KEY'
C
C INPUT PARAMETERS:
C   N POSITIVE INTEGER GIVING LENGTHS 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   N UNCHANGED
C   LIST A LIST OF LENGTH N OF INTEGERS SORTED IN ASCENDING SEQUENCE OF THEIR DOUBLE PRECISION KEYS
C   KEY UNCHANGED
C
C NOTES
C USES QUICKSORT ALGORITHM, EFFICIENT ONLY FOR N VALUES GREATER THAN ABOUT 12 (ALTHOUGH MAY BE SYSTEM DEPENDENT)
C FOR SEQUENTIAL LIST SHORTER THAN 2**M, USE QUICKSORT ROUTINE, WHICH IS EFFICIENT, BUT MAY BE SYSTEM DEPENDENT
C
C ROUTINE SORTS LISTS OF LENGTH 2**M

C QUICKSORT ROUTINE
C
C ORDER INTEGERS STORED IN 'LIST' IN ASCENDING SEQUENCE OF THEIR KEY
C VALUES STORED IN 'KEY'
C
C INPUT PARAMETERS:
C   N POSITIVE INTEGER GIVING LENGTHS 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   N UNCHANGED
C   LIST A LIST OF LENGTH N OF INTEGERS SORTED IN ASCENDING SEQUENCE OF THEIR DOUBLE PRECISION KEYS
C   KEY UNCHANGED
C
C NOTES
C USES QUICKSORT ALGORITHM, EFFICIENT FOR N VALUES GREATER THAN ABOUT 12 (ALTHOUGH MAY BE SYSTEM DEPENDENT)
C FOR SEQUENTIAL LIST SHORTER THAN 2**M, USE QUICKSORT ROUTINE, WHICH IS EFFICIENT, BUT MAY BE SYSTEM DEPENDENT
C
C ROUTINE SORTS LISTS OF LENGTH 2**M

APPENDIX 3

Hybrid sort routine

SUBROUTINE HORS(LIST, KEY)
C ORDER INTEGERS STORED IN 'LIST' IN ASCENDING SEQUENCE OF THEIR KEY
C VALUES STORED IN 'KEY'
C
C INPUT PARAMETERS:
C   N POSITIVE INTEGER GIVING LENGTHS 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   N UNCHANGED
C   LIST A LIST OF LENGTH N OF INTEGERS SORTED IN ASCENDING SEQUENCE OF THEIR DOUBLE PRECISION KEYS
C   KEY UNCHANGED
C
C NOTES
C USES QUICKSORT ALGORITHM, EXCEPT FOR SEGMENTS OF THE LIST SHORTER THAN 2**M, WHICH IS EFFICIENT, BUT MAY BE SYSTEM DEPENDENT
C
C ROUTINE SORTS LISTS OF LENGTH 2**M