Bell Laboratories Cover Sheet for Technical Memorandum

Title- **Struct -**
A Program which Structures Fortran

Date- **December 22, 1975**

TM- **75-1271-12**

Other Keywords-
Structured Programming
Ratfor

ABSTRACT

Fortran programs are often difficult to read because Fortran lacks good constructs for describing flow of control. Programs with many **goto** statements are sometimes incomprehensible. Struct is a program which rewrites Fortran programs using Ratfor control constructs, such as **while** and **if else** statements. These constructs are used by Struct to make loops and branching apparent. The resulting programs appear natural to the reader because Struct follows structuring principles based on normal programming practices. Consequently, the structured programs generated by Struct are dramatically easier to understand than their Fortran counterparts.

Since the structured programs are easier to understand, they are easier to modify and debug. Therefore, Struct is a useful tool for the maintenance of existing Fortran programs. New programs can be written in Ratfor, while old programs can be translated into Ratfor. Thus, all programs can be maintained in the same structured language.

Struct is written in C and currently runs on the PDP-11/45 under UNIX.

Pages Text 10	Other 3	Total 13
No. Figures 0	No. Tables 0	No. Refs. 20

TM-75-1271-12

MC ILROY,M DOUGLAS CM
MH2C526 01/12/76
AUTHOR NAMED ORGANIZATION

E-1932-U (6-73) SEE REVERSE SIDE FOR DISTRIBUTION LIST

December 22, 1975

Bell Laboratories

Subject: **Struct - A Program which Structures Fortran**
Case- 39199 -- File- 39199-11

date: **December 22, 1975**

from: **B. S. Baker**

TM: **75-1271-12**

MEMORANDUM FOR FILE

1. Introduction

Structured programming emphasizes the use of programming language constructs such as **while** loops and **if else** statements, which enable flow of control to be specified clearly. Unfortunately, Fortran lacks these constructs. Yet, Fortran is still heavily used today, for reasons which include efficiency of object code and ease of portability. This paper describes Struct, a program which rewrites Fortran programs in terms of these constructs. The output programs satisfy certain principles of good structuring chosen to ensure that the programs appear natural to the human reader.

Struct is an implementation of the structuring algorithm described in [BAK]. The algorithm itself is not restricted to Fortran as its input language or to any particular output language. It requires only that the flow of control of the input programs be describable by a static flow graph and that all input language statements occur in the output language.

The constructs implemented in Struct include **while** loops, **repeat** (i.e. **do forever**) loops, **if**, **if else**, **next** (which causes a jump to the next iteration of the smallest enclosing loop), and **break** (which causes a jump to the statement following the smallest enclosing loop). The form of the constructs implemented in Struct is based on the Fortran preprocessor language Ratfor [KER].

Struct improves the readability of Fortran programs, often dramatically. Since the structured programs are easier to understand, they are easier to modify, extend, and debug than the original Fortran. Therefore, Struct provides a useful tool for the maintenance of existing Fortran programs. By applying Struct to existing Fortran programs, and writing new programs in Ratfor, all programs may be maintained in the same structured language.

2. Previous approaches to structuring programs

The main goal of Struct is to rewrite existing Fortran programs to make them more readable. Previous approaches to structuring programs have generally concentrated on eliminating **goto** statements, since "structured programming" is often equated with "programming without **goto** statements." However, the known ways of eliminating all **goto** statements can cause programs to become less readable [KN].

One way of eliminating **goto** statements is to add extra variables [AM,BJ,BS,COO,KOS]. The extra variables are used to record information about which statements were executed previously, and the variables are tested to decide what to execute next. Unfortunately, artificial variable names have little mnemonic value and may be confusing when assignments to them are mingled with the main computation of the program.

Another method of eliminating **goto** statements is to modify the program so that **goto** statements may be replaced by other kinds of statements [KOS,PKT]. For example, a **break(i)** statement may be available to cause a jump to the statement following the **i** smallest loops enclosing the **break(i)** statement. By creating dummy loops (i.e. loops where no iteration occurs), the **goto** statements can be replaced by **break(i)** statements. This method makes it difficult to determine where looping really occurs in the program, and it only disguises the **goto** statements as **break** statements in any case.

Another method of avoiding **goto** statements is to copy code which can be reached from several places [AM,BJ,BS,COO,KOS,PKT]. This technique may cause programs to become unduly long, and makes it difficult to observe the identity of the copied statements. An alternative to copying code is to create subroutines out of code segments which can be reached from several places. However, the code segments would not necessarily be semantic entities which deserved to be called subroutines.

Each of the above techniques may be appropriate for some programs. Rather than make a value judgment for each program, Struct does not apply these techniques. Instead, Struct applies structuring principles based on normal programming practice to ensure that the resulting programs appear natural to the reader.

Another approach to structuring Fortran has been taken by de Balbine [BAL74,BAL75]. Like Struct, his "structuring engine" attempts to produce structured programs acceptable to human readers. The "structuring engine" does not create extra variables but sometimes creates subroutines or copies code to guarantee that each block of code has a single entry and single exit. The "structuring engine" is proprietary and no explicit description of the structuring principles or algorithm has been published.

3. Fortran and Ratfor control constructs

Struct accepts a large dialect of Fortran which includes American National Standard Fortran [ANS66, ANS69, ANS71]. It assumes that its input is a syntactically valid Fortran routine. It does not check the program for errors, except for those which make the flow of control impossible to determine, such as a **goto** to a non-existent statement label. It looks for labeled statements and for the following statements which contain labels or affect flow of control (braces enclose optional strings, *label** denotes a list of labels, and ... denotes omitted parts of statements):

assign *label* **to** ...
continue
do *label* ... = ...
end
entry ...
format (...)
... **function** ...
goto (*label**), ... "*computed goto*"
goto ..., (*label**) "*assigned goto*"
goto *label*
if (...) *label, label, label* "*arithmetic if*"
if (...) *statement* "*logical if*"
print *label* {,...}
punch *label* {,...}
read (...{,*label*}{,*opt1*}{,*opt2*}) ...
where *opt1* and *opt2* are **end=***label* or **err=***label*
return
stop

subroutine ...
write (...{,*label*}{,**err**=*label*}) ...

Any statement not recognized as one of the above is treated as straight line code, since it does not affect the structuring process.

Struct rewrites Fortran programs using the following additional Ratfor constructs, where **S**, **S1**, and **S2** are Ratfor statements and **p** is a Fortran logical expression:

1) **if (p) S**

Fortran equivalent:

```
      if (.not.p) goto 10
      S
10    ...
```

2) **if (p) S1 else S2**

Fortran equivalent:

```
      if (.not.p) goto 10
      S1
      goto 20
10    S2
20    ...
```

3) **while(p) S**

Fortran equivalent:

```
10    if (.not.p) goto 20
      S
      goto 10
20    ...
```

4) **repeat S**

Fortran equivalent:

```
10    S
      goto 10
```

5) **do** *control-sequence* **S**

Fortran equivalent:

```
      do 10 control-sequence
      S
10    continue
```

6) **break**
meaning: go to the statement following the smallest enclosing loop,

7) **next**
meaning: goto to the next iteration of the smallest enclosing loop (to the predicate in a **while** loop)

8) braces {} may be used to group multiple Ratfor statements into single Ratfor statements.

Struct applies structuring principles to rewrite a Fortran input program using these Ratfor constructs.

4. Struct and principles of proper structuring

Proper use of control constructs such as **while** loops and **if else** statements should make the flow of control of a program evident from its form. The following eight principles are followed by Struct in order to produce structured programs with clear flow of control.

Principle 1.

Unreachable statements in a Fortran program are deleted during structuring. Structuring preserves the number of occurrences of reachable predicates and segments of straight line code. Their execution order is also preserved to ensure that the structured program is equivalent to the orginal program. No new predicates or straight line code are added.

Principle 2.

Looping constructs reflect iteration in the program. Each statement enclosed by a looping construct such as **while** or **repeat** is reachable from the head of the loop and can lead to an iteration of the loop.

For example, Struct generates

```
repeat {
      n = f(n)
      if (n > 0) break
      }
code segment
return
```

rather than

```
repeat {
        n = f(n)
        if (n > 0) {
                code segment
                return
                }
        }
```

since the latter code violates Principle 2. In this example, this principle prevents the whole program from appearing inside the **repeat** when only two statements are iterated.

Principle 3.

Loops are created only by means of **while** or **repeat**. Each **goto** statement must jump to a statement after it on the page, so that the target of the **goto** is easy to locate.

Principle 4.

A **goto** may not jump into a **then** or **else** clause except from outside a loop containing the clause (see Principle 6 below).

Principle 5.

A **then** or **else** clause must contain as much code as it can without violating Principle 2 or Principle 4.

For example, consider the following code segment.

```
if (p)
        j = f(i)
else
        j = 0
i = j
```

Placing the statement **i = j** inside either the **then** or **else** clause would require a **goto** into the clause. Therefore, this code segment must be written as is to obey Principle 5.

Principle 6.

When a loop is entered in several places, one entry point is selected as the "head" of the loop, and the inside of the loop is structured as if the loop were entered only at its head. The other entry points are then reached by **goto** statements from outside the loop.

It has been shown that jumps into the middle of loops cannot always be avoided without copying code [HU72]. Principle 6 prevents a jump into a loop from destroying the structure of the inside of the loop. For example, consider the following code segment.

```
if (p) goto 10
while(q) {
      if (r)
            while(s) {
            10    j = j+1
                  }
      i = i+1
      }
```

This example contains a jump into an inner loop, and obeys Principle 6. The **while** statements are structured as they would be if both the true and false branches of the **if (p)** went to the outer **while**. As a result, the code contains a jump into the **if (r)** statement from outside the outer **while** (see Principle 4). However, the flow of control within the **while (q)** is clear, and the jump into the inner loop is obvious.

Principle 7.

Every loop construct may be entered at its head, and not just through labeled statements in its body.

Principle 8.

Each statement or predicate may be the head of at most one loop, i.e. the first statement inside a **repeat** statement cannot be another loop construct.

This principle is motivated by the desire to avoid unnecessary complexity in the output. for example, the following code segment violates Principle 8.

```
repeat {
      repeat {
            x = f(x)
            if (p) break
            }
      if (q) break
      }
```

This segment may be rewritten with only one **repeat** to satisfy Principle 8.

```
repeat {
      x = f(x)
      if (p)
            if (q) break
      }
```

Every well-formed Fortran program (i.e. with no loops containing only **goto** and **continue** statements) has at least one corresponding structured program which satisfies these principles. Moreover, if each loop can be entered only at its head, the organization of the structured program into loops and **if else** statements is unique (a more precise version of this statement is

proved in [BAK]). Given a well-formed Fortran program, Struct applies a structuring algorithm based on the above principles to produce a structured program. The algorithm is described briefly in the next section. Struct treats specific Fortran constructs as follows.

Struct transforms all logical and arithmetic **if** statements into statements of the form **if (p) S** or **if (p) S1 else S2**. Predicates are negated when necessary to achieve the **if (p) S** form. Fortran **do** loops are retained as **do** loops. Other loops identified according to the structuring principles are turned into either **while** loops or **repeat** loops according to the following criterion. If a loop begins with a test, and one branch of the test exits from the loop, Struct generates a **while** loop. Otherwise, it generates a **repeat** loop.

Labels occurring in the original program are discarded. New labels are generated as needed, in increasing order on the page to make them easy to locate.

Struct assumes that each comment applies to the line of code following the comment. Comments occurring before a **goto** or **continue** statement are deleted, because **goto** and **continue** statements are not preserved in the output program. Other comments are kept with the following code. **Format** statements are placed at the end of the routine.

Two statements in Fortran which make control flow difficult to follow are the assigned **goto** and the computed **goto**. Other peculiar forms of **goto** in Fortran are the **err=** and **end=** conditions on **read** and **write** statements. These statements are left unchanged except for the substitution of new labels for the original labels.

Struct indents its output to make the scope and nesting of loops and **then** or **else** clauses evident. Struct also replaces Hollerith strings (e.g. **5hhello**) by quoted strings (e.g."**hello**"), and comparison operators (e.g. **.gt.**) by symbols (e.g. >). It applies certain logical identities (e.g. **.not.(c.le.b)** is changed to **c > b**). These cosmetic changes improve the readability of the programs.

The Appendix contains an example of a Fortran program and the structured program generated from it by Struct.

5. The algorithm

This section describes the basic ideas of the structuring algorithm. A full description of the algorithm is given in [BAK].

The first step in analyzing a Fortran program is to obtain a flow graph for the program. Nodes in a flow graph correspond to statements and **if** predicates, while arcs indicate flow of control between statements and predicates. The node corresponding to the first statement in the program is distinguished as the "start" node of the graph.

A loop in a program corresponds to a cycle in the flow graph, i.e. a path which begins and ends at the same node. A node at which the loop may be entered from outside the loop is referred to as an entry point. One entry point of each cycle may be located by means of a depth first search [HU74] starting at the start node of the graph. A depth first search scans the graph by searching arcs from the most recently searched node before arcs from previously searched nodes. The entry point selected by the depth first search becomes the "head" of the loop when the program is written out.

In the structured program a statement should not appear inside a loop unless it can lead to an iteration of the loop. Therefore, for each node, Struct finds the smallest loop in which the node can lead to an iteration, and uses this information to determine what should be written inside each loop. A statement is made to follow any non-enclosing loops from which it can be reached, so that if **goto**'s are needed they will flow downward on the page.

The next step in the analysis is to decide what statements should be written in the **then** or **else** clauses of each **if** statement. Consider the following fragment of code.

```
      if (x.gt.0) goto 10
5     y = -x
      goto 20
10    y = x
20    z = sqrt(y)
```

One might like this fragment to be rewritten as

```
if (x > 0)
      y = x
else
      y = -x
z = sqrt(y)
```

In order to produce the above structuring, it must be ascertained that no other statements jump to **5** or **10**, and that control passes to **20** from both **5** and **10**. That is, statements must be classified according to whether they can be reached only from the "true" ("false") arc of an **if** node, or whether they can be reached from both the "true" and "false" arcs.

This information can be obtained by applying techniques usually used in code optimization. A node *b* **dominates** node *c* if every path from the start node to node *c* passes through node *b* [AU]. Node *b* is the **immediate dominator** of node *c* if node *b* is the closest dominator to *c* in the flow graph. Dominators are used in structuring a program as follows. If a statement can be reached through both the "true" and "false" links of an **if** node, the statement is made to follow the **if** statement which is its immediate dominator. This ensures that any **goto**'s to this node flow downward on the page.

At this point, the basic form of the structured program is determined. Next, Struct determines how each arc in the flowgraph should be written - as a **next**, **break**, **goto**, or with no explicit statement. Finally, labels are added where necessary.

6. Implementation of Struct

Struct is written in the programming language **C**. It consists of about 3000 lines of code, and runs on a PDP-11/45 with 32K 16-bit words of memory. The space and time used are proportional to the square of the length of the input. With 32K words of memory, it is able to structure Fortran programs several hundred lines in length.

7. Evaluation of Struct

The readability of most Fortran programs is improved by Struct, often dramatically. The structured programs usually appear quite natural to the reader. In fact, if a typical Ratfor program is translated into Fortran by the Ratfor preprocessor, and Struct is applied to the resulting program, the output from Struct is usually very similar to the original Ratfor program, except where the original Ratfor program contains Ratfor control structures not implemented in Struct (such as **for** and **until** statements). If each loop can be entered only at its head and the original Ratfor program follows the structuring principles, the structured program generated by Struct is guaranteed to have the same basic form as the original [BAK].

Struct makes the structure of Fortran programs clear, but does not improve the structure of badly-written programs. However, the structure of a Fortran program may be reasonable even when the program looks like a tangled mess of **goto**'s; in such a case, Struct untangles the

program to produce a nice Ratfor program. When the structure of the Fortran program is not good, peculiar flow of control stands out in the Ratfor version and may indicate sections of the program which could be improved by rewriting.

The structured version of a well-written Fortran program generally has few **goto** statements. Many of the **goto** statements which do occur could be replaced by multi-level **break** statements or multi-level **next** statements if these statements were implemented in Struct. Since all **goto** statements generated by Struct flow downward on the page, and labels increase from top to bottom of the page, it is not difficult to determine the target of **goto** statements.

The structured versions of Fortran programs frequently contain many **repeat** statements rather than **while** statements. The reason is that Fortran programmers frequently place tests for exit conditions in the middle or at the end of the loop rather than at the beginning.

Certain types of control flow do not become substantially clearer as a result of structuring. A **goto** statement must be used to enter a loop at a point other than the top of the loop. Ratfor does not have a control construct capable of describing complex ways of merging flow of control. For example, the following code segment cannot be written in a nicer way using the same number of occurrences of the predicates and assignments.

```
      if (p1) {
            if (p2) goto 20
            }
      else if (p3)
            goto 20
10    i = 1
      goto 30
20    i = 2
30    y = f(i)
```

In some cases, copying code or creating subroutines would improve the Ratfor output from Struct. In other cases, improvement would be obtained only by modifying the actual code executed.

Struct is not capable of making semantic decisions. It is not able to identify commonly occurring code sequences such as

```
j = 1
if (p) j = 2
```

in order to replace them by code such as the following.

```
if (p) j = 2
else j = 1
```

To do this, Struct would need to examine the semantics of the statements. Similarly, successive tests such as those occurring in

```
if (p) goto 10
if (q) goto 10
```

are not replaced by a combined test such as **if (p.or.q)** since Struct cannot determine whether it is safe to evaluate q when p is true (in general this is undecidable). The tests could be combined if Ratfor specified evaluation of logical expressions from left to right.

The structuring algorithm used by Struct is not limited to producing the control constructs currently implemented. Other constructs could be added easily. The usefulness of proposed new control structures could be tested by using Struct to demonstrate their application in existing programs. Different ways of writing the same program could be compared by adding options to select the control constructs to be generated. In fact, Struct has been used for some experimentation of this sort.

8. Using Struct.

Struct is implemented on UNIX. The command format is

struct [-s] [-tn] [-cn] [-v] [-en] file

where *n* is a nonnegative integer. The Fortran program specified by *file* is translated into a structured program, written on the standard output. The optional flags may appear in any order and are interpreted as follows:

-s If this flag is set, input is expected to be in standard Fortran card format, i.e. comments are specified by a c, C, or * in column 1, and continuation lines are specified by a nonzero, nonblank character in column 6. If the flag is not present, input is expected to be in free format, i.e. comments are specified by a c, C, or * in column 1, and continuation lines are specified by an & as the first nonblank character on the line.

-tn Make the nonzero integer *n* the lowest valued label in the output program. If the flag is not present, the default value is 10.

-cn Increment successive labels in the output program by the nonzero integer *n*. If the flag is not present, the default value is 10.

-v If this flag is set, the **then** and **else** parts of **if else** statements are interchanged and the predicate is negated.

-en If n is 0, code appears in a loop only if it can lead to an iteration of the loop (Principle 2). If *n* is greater than zero, Struct modifies Principle 2 by allowing certain segments of code to appear within a loop even when they do not lead to an iteration of the loop. The criteria are that the segment must be one of several exits from the iterating code of the loop, must be reachable directly from only one other statement, and must be "small enough". "Small enough" means that the current estimate of the number of statements in the segment must be at most *n* (when this flag is used by Struct, the precise number of statements to be used for the segment hasn't been determined). Values of n under 10 are suggested. If this flag is not present, the default value is 0.

MH-1271-B.S.B.

B. S. Baker

Att.
References
Appendix

References

[AU] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling, Vol. II - Compiling*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[ANS66] *American National Standard Fortran*, American National Standards Institute, New, York, N.Y., 1966.

[ANS69] Clarifications of Fortran Standards - Initial Progress, *Comm. ACM 12* (1969), 289-294.

[ANS71] Clarifications of Fortran Standards - Second Report, *Comm. ACM 14* (1971), 628-642.

[AM] E. Ashcroft and Z. Manna, Translating program schemas to while-schemas, *SIAM J. Comput. 4,2* (1975), 125-146

[BAK] B. S. Baker, An algorithm for structuring programs, to be presented at the *ACM Symposium on Principles of Programming Languages*, January, 1976.

[BAL74] G. de Balbine, Better man power utilization using automatic restructuring, Caine, Farber & Gordon, Inc., 1974.

[BAL75] G. de Balbine, Using the Fortran structuring engine, *Proc. of Comp. Sci. and Stat.: 8th Ann. Symp. on the Interface*, Los Angeles (1975), 297-305.

[BJ] C. Bohm and G. Jacopini, Flow diagrams, Turing machines and languages with only two formation rules, *Comm. ACM 9* (1966), 366-371.

[BS] J. Bruno and K. Steiglitz, The expression of algorithms by charts, *JACM 19* (1972), 517-525.

[COO] D. C. Cooper, Bohm and Jacopini's reduction of flow charts, *Comm. ACM 10* (1967),463.

[DDH] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, New York, 1972.

[DIJ] E. W. Dijkstra, **Goto** statement considered harmful, *Comm. ACM 11* (1968), 147-148.

[HU74] M. S. Hecht and J. D. Ullman, Characterizations of reducible flowgraphs, *JACM 21,3* (1974), 367-375.

[HU72] M. S. Hecht and J. D. Ullman, Flow graph reducibility, *SIAM J. Comput. 1* (1972), 188-202.

[KER] B.W. Kernighan, Ratfor - a preprocessor for a rational Fortran, *Software Practice and Experience 5,4* (1975), 395-406.

[KF] D. E. Knuth and R. W. Floyd, Notes on avoiding "**go to**" statements, *Infor. Proc. Letters 1* (1971), 23-31.

[KN] D.E. Knuth, Structured programming with **goto** statements, *ACM Comp. Surveys 6,4* (1974), 261-302.

[KOS] S. R. Kosaraju, Analysis of structured programs, *J. Comp. Sys. Sci. 9,3* (1974), 232-254.

[PKT] W. W. Peterson, T. Kasami, and N. Tokura, On the capabilities of while, repeat and exit statements, *Comm. ACM 16* (1973), 503-512.

Appendix

A Fortran subroutine (from R. C. Singleton, Algorithm 347, an efficient algorithm for sorting with minimal storage, *Comm. ACM 12,3* (1969), p. 186):

```
      subroutine sort(a,ii,jj)
c sorts array a into increasing order
c from a(ii) to a(jj)
      dimension a(1),iu(16),il(16)
      integer a,t,tt
      m = 1
      i = ii
      j = jj
5     if (i .ge. j) goto 70
10    k = i
      ij = (j+i)/2
      t = a(ij)
      if (a(i) .le. t) goto 20
      a(ij) = a(i)
      a(i) = t
      t=a(ij)
20    l=j
      if (a(j) .ge. t) goto 40
      a(ij) = a(j)
      a(j) = t
      t = a(ij)
      if (a(i) .le. t) goto 40
      a(ij) = a(i)
      a(i) = t
      t = a(ij)
      goto 40
30    a(l) = a(k)
      a(k) = tt
40    l = l-1
      if (a(l) .gt. t) goto 40
      tt = a(l)
50    k=k+1
      if (a(k) .lt. t) goto 50
      if (k .le. l) goto 30
      if (l-i .le. j-k) goto 60
      il(m) = i
      iu(m) = l
      i=k
      m=m+1
      goto 80
60    il(m) = k
      iu(m)=j
      j=l
      m=m+1
      goto 80
70    m=m-1
      if(m.eq. 0) return
      i=il(m)
      j=iu(m)
80    if (j-i .ge. 11) goto 10
      if (i .eq. ii) goto 5
      i=i-1
90    i=i+1
      if (i .eq. j) goto 70
      t = a(i+1)
      if (a(i) .le. t) goto 90
      k=i
100   a(k+1) = a(k)
      k = k-1
      if (t .lt. a(k)) goto 100
      a(k+1) = t
      goto 90
      end
```

The preceding program as structured by Struct:

```
   subroutine sort(a,ii,jj)
# sorts array a into increasing order
# from a(ii) to a(jj)
   dimension a(1),iu(16),il(16)
   integer a,t,tt
   m = 1
   i = ii
   j = jj
   repeat
      {if (i<j)
         go to 10
      repeat
         {m = m-1
         if (m==0)
            return
         i = il(m)
         j = iu(m)
         while (j-i>=11)
            {
         10  k = i
            ij = (j+i)/2
            t = a(ij)
            if (a(i)>t)
               {a(ij) = a(i)
               a(i) = t
               t = a(ij)
               }
            l = j
            if (a(j)<t)
               {a(ij) = a(j)
               a(j) = t
               t = a(ij)
               if (a(i)>t)
                  {a(ij) = a(i)
                  a(i) = t
                  t = a(ij)
                  }
               }
            repeat
               {l = l-1
               if (a(l)<=t)
                  {tt = a(l)
                  repeat
                     {k = k+1
                     if (a(k)>=t)
                        break
                     }
                  if (k>l)
                     break
                  a(l) = a(k)
                  a(k) = tt
                  }
               }
            if (l-i<=j-k)
               {il(m) = k
               iu(m) = j
               j = l
               m = m+1
               }
            else
               {il(m) = i
               iu(m) = l
               i = k
               m = m+1
               }
            }
         if (i==ii)
            break
         i = i-1
         repeat
            {i = i+1
            if (i==j)
               break
            t = a(i+1)
            if (a(i)>t)
               {k = i
               repeat
                  {a(k+1) = a(k)
                  k = k-1
                  if (t>=a(k))
                     break
                  }
               a(k+1) = t
               }
            }
         }
      }
   return
   end
```