A WORKED EXAMPLE IN UNSTRUCTURED SYSTEMATIC PROGRAMMING

by

M.H. van Emden

Research Report CS-76-27

Department of Computer Science

University of Waterloo
Waterloo, Ontario, Canada

July, 1976

A WORKED EXAMPLE IN UNSTRUCTURED SYSTEMATIC PROGRAMMING

by

M.H. van Emden

Abstract:

A FORTRAN program, which is both efficient and proved correct, for the quicksort algorithm is obtained by means of Hoare's method of data abstraction and a certain method due to Dijkstra. According to this method, a proof with Floyd's assertions is constructed as much as possible before the program is written. In this paper, Dijkstra's method is realized by means of 'flowgraphs', a form of program which can be interpreted as the verification conditions of its own correctness proof and which can also be translated in a systematic manner to FORTRAN code.

Key Words & Phrases

programming method, data abstraction, flowgraphs, program correctness, assertions, verification conditions, quicksort

CR Categories

4.0, 5.24

---

*) Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

1. Introduction

"Unstructured Systematic Programming" [2] described the method of programming with flowgraphs and explained by means of a small example how this method can be used to obtain in a systematic fashion a correct FORTRAN program. Because the example was so small, it did not show clearly that the method described is useful in practice. The present paper, which is entirely devoted to the systematic development of an efficient sorting algorithm, is intended as a companion of, and a sequel to, "Unstructured Systematic Programming".

In the systematic development of the sorting algorithm not only the method of flowgraphs is useful, but also the method of "data abstraction", which can best be described by using the following quote from Hoare [4]:

> "In the development of programs by stepwise refinement, the programmer is encouraged to postpone the decision on the representation of his data until after he has designed his algorithm, and has expressed it as an "abstract" program operating on "abstract" data. He then chooses for the abstract data some convenient and efficient representation in the store of a computer; and finally programs the primitive operations required by his abstract program in terms of this concrete representation".

Hoare's paper is devoted to an automatic method of accomplishing the transition between an abstract and a concrete program and to a proof of correctness of the concrete program. In the development of the sorting algorithm I do not apply Hoare's automatic method for effecting the transition.

A few remarks about notation. The brackets [ ] are used not only in literature references but also to surround the listing of a sequence. Braces { } are used around the assertions in a verification condition and also in the usual way for denoting a set. An expression consisting of vertical bars | | around the name of a set or sequence denotes the number of elements in that set or sequence.

2. Two abstract sorting algorithms

The quicksort algorithm [3] is usually regarded as a function from an array to an array. However, the unconstrained accessibility, which distinguishes arrays from sequences in general, is not required for the quicksort algorithm: it may be programmed so as to access the elements sequentially. This fact suggests an opportunity for decomposing the task of programming quicksort by an application of the method of data abstraction: the first subtask is to write an abstract version of the algorithm to sort from a sequence to a sequence; the second subtask is to represent the sequence in storage.

But why should the *input* to an abstract sorting algorithm be a sequence? There, surely, order is irrelevant: from an abstract point of view, sorting is a function from a set to a sequence. According to the method of data abstraction, the representation of a set is considered only in a later stage of program development. The choice of representation of a set as a sequence is only one among several possibilities. Again, the representation of a sequence as an array is one possibility among several.

By regarding sorting as a function from a set to a sequence we have succeeded in discarding details of representation which are irrelevant in an abstract specification of sorting. But we have lost the advantage that input and output are of the same type. If the sorting algorithm is to have the simple form of an iteration of one operation then it is necessary to have the input and output for that operation of the same type. This same type must be such that both sets and sequences are special cases of it.

Let us suppose that both the input and the output for a sorting algorithm are a *set of sequences*. The input would be a set of n one-element sequences because nothing is ordered yet. The output would be a set containing one sequence which has n elements in order. For example:

$$\{[c], [a], [b], [d]\} \longrightarrow \{[a,b,c,d]\}$$

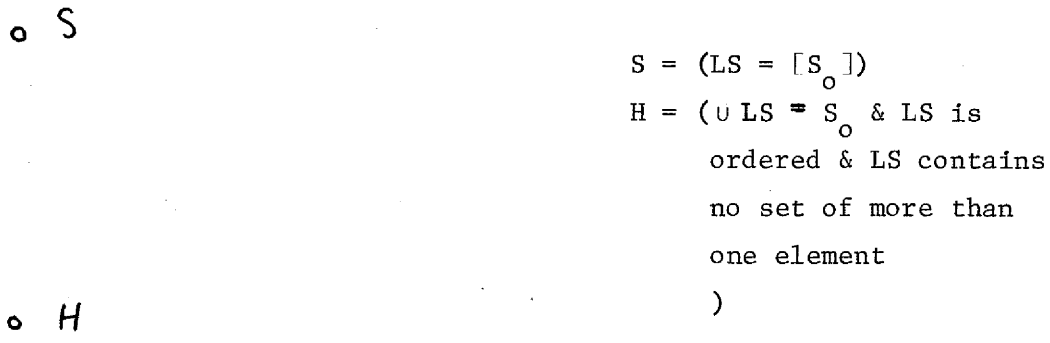
The sorting algorithm proceeds here from a set of many sequences to a set of one sequence; such an algorithm I call a *mergesort*.

According to the other possibility both the input and the output are a *sequence of sets*. The input would be a one-element sequence of the set containing all objects to be sorted. The output would be a sequence of one-element sets in order. For example:

$$[\{c,a,b,d\}] \longrightarrow [\{a\},\{b\},\{c\},\{d\}]$$

Because the sorting algorithm proceeds here from a sequence of one set to a sequence of many sets, I call it a *splitsort*.

We are now in a position to begin a first approximation to a splitsort algorithm. According to the method of flowgraphs there is at each stage during the development of an algorithm a partially correct flowgraph. Each successive flowgraph is obtained from the previous one by accretion of nodes or of arcs or by a change of some assertion. Each command is a binary relation over the set of states, which will be specified for the initial version to consist of the values of one variable LS. The values of LS are sequences of sets. The following flowgraph, which expresses only the input - output specification, is the first approximation.



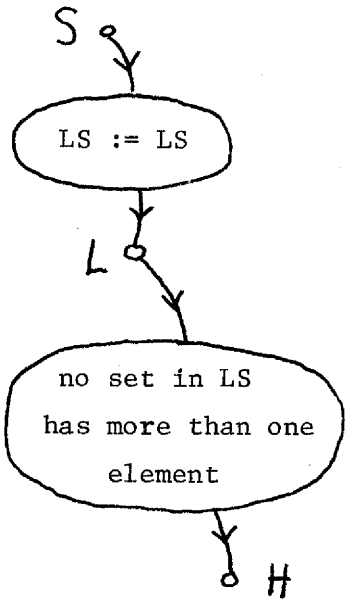
Flowgraph 2.1

The assertion "LS is ordered" means that if a set S_1 **precedes** a set S_2 in LS then $S_1 \leq S_2$, which means that $x \in S_1$ and $y \in S_2$ imply that $x \leq y$. This relation among sets generalizes the given order among elements in the sense that it is analogous for the case of unit sets.

For further development of Flowgraph 2.1 I assume that no useful command X is available which satisfies $\{S\}\ X\ \{H\}$. Then an assertion L and commands X_1 and X_2 have to be found such that $\{S\}\ X_1\ \{L\}$ and $\{L\}\ X_2\ \{H\}$. It is straightforward to find X_1 and X_2 if L is such that both S and H are special cases of it, such as, for example,

$$L = (\cup LS = S_o \ \&\ LS \text{ is ordered})$$

It will make things much easier later on if the possibility of an empty set in LS is ruled out once and for all, as in the next approximation (see Flowgraph 2.2).



$S = (LS = [S_o] \ \&\ S_o \text{ nonempty})$

$L = (\cup LS = S_o \ \&\ LS \text{ is ordered}$
& LS contains no empty
set & LS is disjoint
)

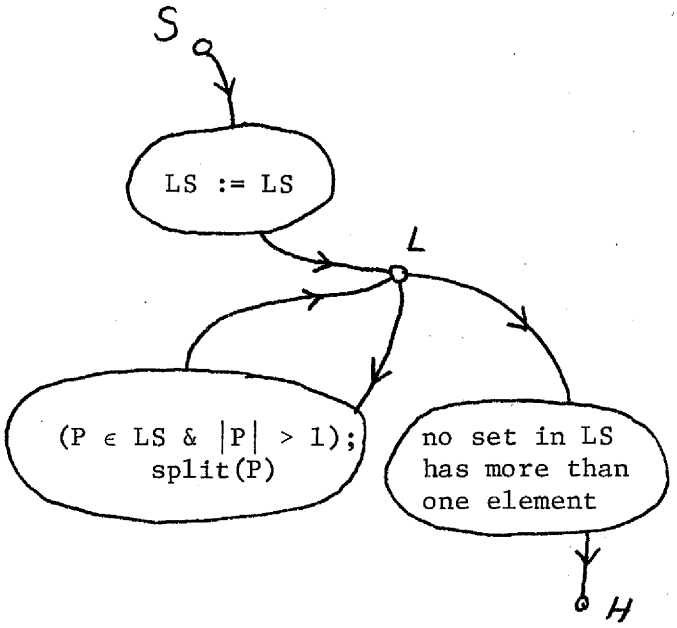
H = (L & LS contains no set of
more element)

$\{S\}$ LS := LS $\{L\}$

$\{L\}$ no set in LS has more than
one element $\{H\}$

Flowgraph 2.2

In Flowgraph 2.2 we only have a terminated computation if $|S_o| = 1$, because there is yet no command for changing LS. Suppose now that there is available a command, split(P), which replaces in LS a set P with more than one element by a partition of P into three sets: X, a unit set Y, and Z (the replacement to be in that order), such that $X \leq Y \leq Z$. However, when X or Z is empty, then P is replaced by Y,Z (in that order) or by X,Y (in that order). Because the command split(P) preserves the truth of the assertion L, we obtain Flowgraph 2.3 as the next version of splitsort.



S = (LS = [S_o] & S_o nonempty)
L = (∪ LS = S_o & LS is ordered
& LS contains no empty set
& LS is disjoint)
H = (L & LS contains no set of
more than one element)

{ S} LS := LS { L}
{ L} (P ∈ LS & |P| > 1);split(P) { L}
{ L} no set in LS has more than one element { H}

Flowgraph 2.3

The validity of the verification conditions in Flowgraph 2.3 implies partial correctness. The absence of blocked computations follows from the complementarity of the guards "P ∈ LS & |P|> 1" and "no set in LS has more than one element" and from the (hereby assumed) termination of split(P), under the given guard. The absence of infinite computations follows from the fact that every infinite path from S has infinitely many occurrences of split(P) which decreases $|S_o|-|LS|$ by at least one. This quantity is integer, bounded from below, and is increased nowhere.

Now an abstract splitsort algorithm has been completed. It may be anticipated that a concrete algorithm based on it will not be efficient, because a splittable set will have to be selected from among possibly many sets in LS. Yet the basic idea in Flowgraph 2.3 can be elaborated, still at the abstract level, so as to avoid search for a splittable set.

3. Improvement of the abstract algorithm

The most obvious defect of flowgraph 2.3 is that LS has to be searched for a splittable set. The defect can be remedied by introducing a set, the Waiting Set, of all splittable sets in LS. The operation split would then take its argument from the Waiting Set and would return to it any result, which is itself splittable. But split is immediately repeated, so that we find ourselves delving into the Waiting Set to get an element just after having put into it the results of split.

This inefficiency can be avoided by introducing another set, say RS (the Ready Set), of at most one element, to hold the next set to be split. Then, if at least one of the results of split is itself splittable, such a one goes into RS to be ready for the next activation of split. If both results are splittable, then the other goes into the Waiting Set to be split at some later time. It may happen that no result is splittable, but then the Waiting Set may be nonempty and RS can be filled from it before the next activation of split. Finally, if the Waiting Set is also empty, then no splittable set is left in LS, that is, sorting is completed.

The advantage of having the Waiting Set is that it does not need to be searched, because each of its elements has to be split sooner or later. *But* with the right choice of an element to be split one can achieve a great saving in the size of the Waiting Set: if at each time the smallest element is transferred to RS for splitting, then we can expect large sets in the Waiting Set, and hence a small number of them.

Can efficiency of retrieval (just take for splitting *any* element of the Waiting Set) be combined with efficiency in the sense that the Waiting Set remain small? The combination is possible because if the smallest set is selected for transfer to RS, the next set (if any) to be stored, being a result of splitting, is smaller still. This shows that we get the best of both worlds by making the Waiting Set into a stack, because then automatically smaller sets will be above larger ones. In fact, the verification conditions will only prove that the sets

w_1 (the bottom),..., w_k (the top)

of the stack satisfy (from now on the *stack* will refer to the Waiting Set)

$$|w_i| \leq 2^{-i+1} |S_o| \quad \text{for } i = 1,...,k$$

and that the set r in a nonempty RS satisfies

$$|r| \leq 2^{-k} |S_o|$$

Because $|w_k| \geq 2$ we find that $k \leq \log_2|S_o|$, which bounds the height of the stack for a given size $|S_o|$ of the set to be sorted. The use of the stack and of RS is summarized in the assertion W shown with flowgraph 3.1.

In flowgraph 3.1 the state has as components the variables LS, WS, RS, X, Y, and Z. The differences between flowgraphs 3.1 and 2.3 arise from the use of the stack and of RS: the results X and Z of splitting have to be compared in size and placed on the stack or in RS according to the result of that comparison and, of course, only if the set has more than one element.

I assume that the validity of the verification conditions in flowgraph 3.1 requires no further explanation, except possibly those that contain commands changing the stack. Take for example

$$\{Q_4\}\ |Z| \geq |X|;\ \text{push}(Z);\ RS := \{X\}\ \{Q_1\}$$

Both Q_4 and Q_1 imply W, which must therefore remain invariant under the above commands. Let us verify this. X and Z are the results of splitting r, the contents of RS before splitting. Initially W holds: $|r| \leq 2^{-k} |S_o|$; hence, $|Z| \leq 2^{-k} |S_o|$ and for X, being the smaller result of splitting, we have $|X| \leq 2^{-k-1} |S_o|$. When now Z is pushed onto the stack, k, the size of the stack, is increased by one, and we have ($|Z| =$) $|w_k| \leq 2^{-k+1} |S_o|$, as required by assertion W. When X is made equal to r, the set in a nonempty RS, we have $|r| \leq 2^{-k} |S_o|$, as required by assertion W.

The absence of infinite and of blocked computations is proved in essentially the same way as for flowgraph 2.3.

$\{ S\}$ stack := empty; RS := $\{ S_o\}$

$\{ Q_1 \stackrel{def}{=}$ (L & W
& (stack $\cup$ RS) contains all nonunits of LS
)
$\}$

$\{ Q_1\}$ split $\{ Q_2 \stackrel{def}{=}$ (L & W
& (stack $\cup \{ X,Z\}$) contains all nonunits of LS
)
$\}$

$\{ Q_2\}$ $|X| > 1$ $\{ Q_4 \stackrel{def}{=} Q_2$ & $(|X| > 1)\}$
$\{ Q_2\}$ $|X| \leq 1$ $\{ Q_3 \stackrel{def}{=} Q_2$ & $(|X| \leq 1)\}$
$\{ Q_3\}$ $|Z| \leq 1$ $\{ Q_6 \stackrel{def}{=}$ (L & W & (stack contains all nonunits of LS))$\}$
$\{ Q_3\}$ $|Z| > 1$; RS := $\{ Z\}$ $\{ Q_1\}$
$\{ Q_4\}$ $|Z| < |X|$ $\{ Q_5 \stackrel{def}{=} Q_4$ &$(|Z| < |X|)\}$
$\{ Q_4\}$ $|Z| \geq |X|$; push (Z); RS := $\{ X\}$ $\{ Q_1\}$
$\{ Q_5\}$ $|Z| > 1$; push (X); RS := $\{ Z\}$ $\{ Q_1\}$
$\{ Q_5\}$ $|Z| \leq 1$; RS := $\{ X\}$ $\{ Q_1\}$
$\{ Q_6\}$ stack is empty $\{ H\}$
$\{ Q_6\}$ stack is nonempty; RS := { the result of popping the stack} $\{ Q_1\}$

where

$S = (LS = [S_o]$ & $1 < |S_o|)$

$L = (\cup LS = S_o$ & the sets of LS are mutually disjoint
& LS is ordered & LS contains no empty set
)

$W = (|w_i| \leq 2^{-i+1} |S_o|$ for i = 1,...,k
& $|r| \leq 2^{-k} |S_o|$, **where** k = |stack| and r is the set in RS
)

H = (L & (LS contains no set of more than one element))

Flowgraph 3.1

4. The choice of storage representation

The abstract splitsort algorithm of flowgraph 3.1 can now be turned into a concrete algorithm by determining storage representations for sets, sequences, and for the stack. The command 'split', which is not elaborated in flowgraph 3.1, will be developed from the start in concrete terms.

A *set* is stored as a sequence of contiguous elements of an array A (a "segment" of A). A *sequence of sets* is stored as a sequence of segments contiguous in A. A set

$$\{A(min),A(min+1),\ldots,A(max)\}$$

of contiguous array elements can be represented by a pair (min,max) of indexes, of the leftmost and of the rightmost element. The *stack of sets* is stored as two arrays, LEFT and RIGHT and an index PRT: the i-th set (i = 1,...,PTR) of the stack is the set

$$\{A(LEFT(i)),A(LEFT(i)+1),\ldots,A(RIGHT(i))\}$$

The set in a nonempty RS is represented by the pair (M,N) of indexes. The results X,Y, and Z of split can be stored in the segment A(M),...,A(N) in such a way that they are represented as (M,J-1),(J,J), and (J+1,N), respectively, for some J such that $M \leq J \leq N$.

Given the above decisions about storage representation, the translation of the commands in flowgraph 3.1 to FORTRAN should hardly need any comment. For example, the fact that the set X is stored as

$$A(M),\ldots,A(J-1)$$

implies that the guard $|X| \leq 1$ in flowgraph 3.1 translates to $J-M \leq 1$. For example, push(Z) translates to the FORTRAN statements

```
PTR = PTR + 1
LEFT(PTR) = J + 1
RIGHT(PTR) = N
```

For example, RS := { the result of popping the stack} translates to

```
M = LEFT(PTR)
N = RIGHT(PTR)
PTR = PTR - 1
```

For the sake of brevity and readability I have not yet followed FORTRAN syntax for the commands in the verification conditions of Flowgraph 4.1. The guards and assignments by themselves follow the style of Algol, embellished with parallel assignments. The semantics of the commands which are compositions of guards or assignments is given by relational composition, as described in the semantics of flowgraphs [2].

Now, what about translating the verification conditions? For a correctness proof of Flowgraph 4.1 we have a choice between two methods. According to one possibility, we translate each abstract assertion into its equivalent in terms of the storage representation, in such a way that, if the verification condition holds true before translation (of both commands and assertions), then it holds true afterwards. The difficulty with this method is that the representation has to be specified much more precisely than has been done here and that even then it may not be easy or possible to prove that the truth of a verification condition always survives the translation process.

Suppose we would not be successful in overcoming this difficulty. We still have a program (in the form of Flowgraph 4.1) but we do not have assertions to go into the curly brackets to make the arcs into valid verification conditions. We would have the notoriously difficult problem of finding assertions to prove an existing program correct, if we did not, as we do here, have very clear hints as to what the assertions should be. According to the second method, which I adopt here, only the commands are translated, as shown above, and the abstract assertions are only used as hints for the required matching concrete assertions. Then the verification conditions, with newly found concrete assertions, are tried for validity.

The following hints are sufficient. The facts that $\cup LS = S_o$ and that the sets of LS are mutually disjoint mean that the array A is a permutation of A_o, the value of A in the initial state. This concrete assertion will be called P'.

We also need the concrete version of the assertion that a certain set B contains exactly the nonunit sets of LS. A segment in LS of adjacent unit sets corresponds to a segment in A that is already sorted, that is, each of its elements has already found its definitive place in the sorted permutation. A nonunit set in LS corresponds to a segment in A that still has to be sorted, but is such that the place of each of its elements in the sorted permutation is also within the segment. Hence, the segment only needs to be sorted locally.

Thus, the nonunit sets of LS correspond to the set V of segments of A that still have to be sorted: TBS(V) asserts that the elements of V are those that still have to be sorted. With π being a permutation that rearranges A into sorted order, we can define

$$\begin{aligned} \text{TBS}(V) = \ & (\text{index } i \text{ not in any segment of } V \rightarrow \\ & \pi(i) = i \\ & \& \ (\text{index } i \in \text{segment } s \ \& \ s \in V \rightarrow \pi(i) \in s) \\ &) \end{aligned}$$

For example, instead of the abstract assertion that the union of the stack and RS is the set of nonunit sets in LS, we will use the concrete assertion

TBS({(LEFT(1), RIGHT(1)),...,(LEFT(PTR),RIGHT(PTR)),(M,N)})

It may be verified that the concrete assertions satisfy the verification conditions in Flowgraph 4.1.

The assertion S' requires some explanation. In the abstract algorithm there was no bound on the stack size. But the arrays LEFT and RIGHT have to have a dimension, say d. In the previous section we found that $k \leq \log_2 |S_o|$, where k is the size of the stack at a particular moment. If we assume $\log_2 |S_o| \leq d$, then we can be sure that $k \leq d$, that is, that we can never have stack overflow. A generous bound for the size of S_o is obtained when we take d = 50.

```
{ S'} PTR,M,N := 0,1,N_o  { Q'_1}
{ Q'_1} split  { Q'_2}
{ Q'_2} J - M > 1  { Q'_4}
{ Q'_2} J - M ≤ 1  { Q'_3}
{ Q'_3} N - J ≤ 1  { Q'_6}
{ Q'_3} N - J > 1; M := J + 1  { Q'_1}
{ Q'_4} N - J ≮ J - M  { Q'_5}
{ Q'_4} N - J ≥ J - M; PTR := PTR + 1
                      ; LEFT(PTR),RIGHT(PTR) := J + 1,N
                      ; N := J - 1                          { Q'_1}
{ Q'_5} N - J > 1; PTR := PTR + 1
                  ; LEFT(PTR),RIGHT(PTR) := M,J - 1
                  ; M := J + 1                              { Q'_1}
{ Q'_5} N - J ≤ 1; N := J - 1  { Q'_1}
{ Q'_6} PTR ≤ 0  { H'}
{ Q'_6} PTR > 0; M,N := LEFT(PTR),RIGHT(PTR)
                ; PTR := PTR - 1                            { Q'_1}
```

$S' = (M < N < M + 2^{50})$

$Q'_1 = (P' \ \&$
$TBS(\{ (LEFT(1),RIGHT(1)),\ldots,(LEFT(PTR),RIGHT(PTR)),(M,N)\}))$

$Q'_2 = (P' \ \&$
$TBS(\{ (LEFT(1),RIGHT(1)),\ldots,(LEFT(PTR),RIGHT(PTR),(M,J-1),(J+1,N)\}))$

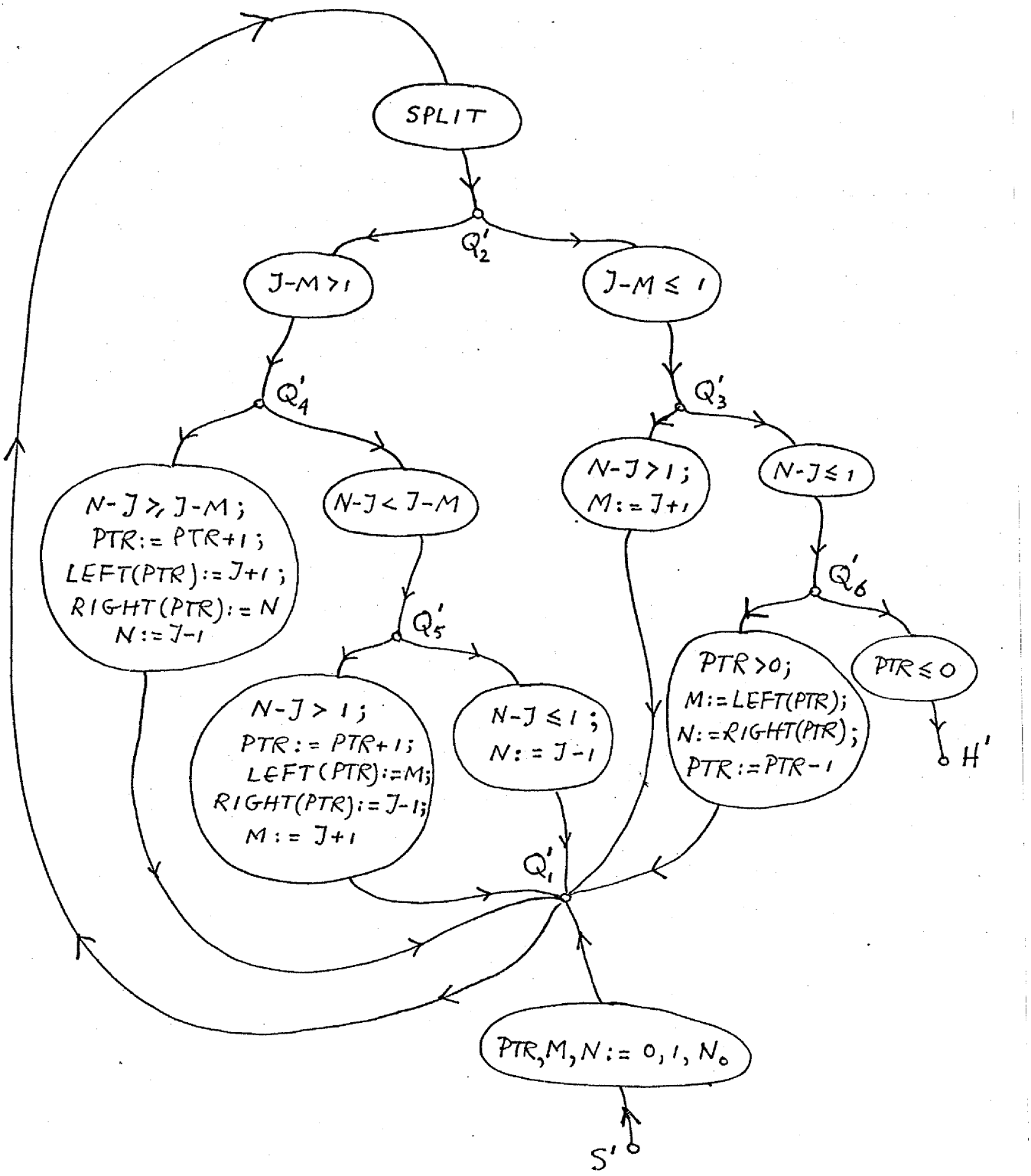
$Q'_3 = (Q'_2 \ \& \ (J-M \leq 1))$

$Q'_4 = (Q'_2 \ \& \ (J-M \not> 1))$

$Q'_5 = (Q'_4 \ \& \ (N-J < J-M))$

$Q'_6 = (P' \ \&$
$TBS(\{ (LEFT(1),RIGHT(1)),\ldots,(LEFT(PTR),RIGHT(PTR))\}))$

Flowgraph 4.1



Pictorial representation of the verification conditions in flowgraph 4.1

For a proof of total correctness we can find a counter by analogy with the counter $|S_o| - |LS|$ used in flowgraph 3.1. This counter expressed how much of LS still has to be sorted. Analogously, we find that the absence of infinite computations in the flowgraph 4.1 may be proved by observing that

$$N - M + 1 + \sum_{i=1}^{PTR} (RIGHT(i) - LEFT(i) + 1)$$

is always integer, is bounded from below, is increased nowhere and decreases along each of the paths from Q_1' to Q_1'; and that any infinite path from S' must have infinitely many occurrences of at least one of these. For a proof of the absence of blocked computations the termination of the command split will have to be investigated.

5. The algorithm for splitting

The only command in flowgraph 4.1 which is not yet immediately translatable to FORTRAN is the one in

$$\{ Q_1' \} \quad \text{split} \quad \{ Q_2' \}$$

The translation must therefore be regarded as a programming problem in its own right with Q_1' as input specification and Q_2' as output specification. In the same way as before, a set of verification conditions will be obtained, which can be regarded as a flowgraph for the splitting algorithm. These verification conditions replace in flowgraph 4.1 the one shown above.

'Split' only affects the segment (M,N) so that instead of Q_1' and Q_2' as defined before, we can take

$$Q_1' = P' \ \& \ (M < N)$$

$$Q_2' = P' \ \& \ M \leq i < J \ \text{ implies } A(i) \leq A(J)$$
$$\& \ J < i \leq N \ \text{ implies } A(J) \leq A(i)$$

For the latter assertion, and others in this section, the following shorthand will be used:

$$Q_2' = P' \ \& \ A(M..J-1) \leq A(J) \leq A(J+1..N)$$

It is advantageous to apply split only if $M+2 < N$ and to take a shortcut to assertion Q_6' (where (M,N) no longer needs to be sorted) by a complete sort of (M,N) when $M+2 \geq N$. Hence we will use

$$\{ Q_1' \} \ A(M) \leq A(N) \ \{ R_1' \stackrel{\text{def}}{=} Q_1' \ \& \ A(M) \leq A(N) \}$$

$$\{ Q_1' \} \ A(M) > A(N); \ \text{swap } (A,M,N) \ \{ R_1' \}$$

$$\{ R_1' \} \ M+1 \geq N \ \{ Q_6' \}$$

where swap(A,M,N) interchanges A(M) and A(N). In case $M+1 < N$, where we have A(M) and A(N) already in the right order, it takes only two additional conditional swaps to order A(M),A((M+N)/2), and A(N):

$\{R_1'\}$ M+1 < N; MN := (M+N)/2 $\{R_2' \stackrel{def}{=} R_1' \;\&\; (M+1 < N) \;\&\; (MN = (M+N)/2)\}$

$\{R_2'\}$ A(M) ≤ A(MN) $\{R_3' \stackrel{def}{=} R_2' \;\&\; (A(M) \le A(MN))\}$

$\{R_2'\}$ A(M) > A(MN); swap (A,M,MN) $\{R_4'\}$

$\{R_3'\}$ A(MN) ≤ A(N) $\{R_4' \stackrel{def}{=} R_3' \;\&\; (A(MN) \le A(N))\}$

$\{R_3'\}$ A(MN) > A(N); swap (A,MN,N) $\{R_4'\}$

$\{R_4'\}$ M+2 ≥ N $\{Q_6'\}$

Only in the now remaining case, where M+2 < N, will 'split' be applied. A(M),A(MN), and A(N) are already in sorted order. We can choose A(MN) to be the element that will be in A(J) when split is finished. If we assume the initial contents of A(M),...,A(N) to be a random permutation, then A(MN) is the median of a random sample of size 3 out of A(M),...,A(N). Such a choice of A(J) has been shown [1] to increase the efficiency of quicksort by about 14% (averaged over equally probable permutations, asymptotically for large arrays) compared to the alternative where A(J) is a uniformly distributed random choice.

For the assertions used with split we are only concerned with A(M),...,A(N). Hence we take for the start assertion

$$R_4' = (P' \;\&\; M+2 < N \;\&\; A(M) \le A(MN) \le A(N))$$

and for the halt assertion

$$Q_2' = P' \;\&\; A(M..J-1) \le A(J) \le A(J+1..N) \;\&\; M \le J \le N$$

At least one intermediate assertion, say R_5', is required. Because we hope to find commands C_1 and C_2 such that $\{R_4'\}\ C_1\ \{R_5'\}$ and $\{R_5'\}\ C_2\ \{Q_2'\}$, we look for an R_5' such that both R_4' and Q_2' are special cases of it. I choose an R_5' that implies

$$P' \;\&\; A(M..I-1) \le A(MN) \le A(J+1..N)$$

It is convenient to move the content of A(MN) to a place where it is guaranteed to stay, so that we can find it later on. Thus I arrive at

$\{R_4'\}$ M+2 < N; AMN := A(MN); A(MN) := A(M+1); A(M+1) := AMN ; I,J := M+2, N-1 $\{R_5'\}$

with

$$R_5' \stackrel{def}{=} P' \;\&\; A(M..I-1) \le AMN \le A(J+1..N)$$
$$\&\; M+1 < I \le N \;\&\; M \le J < N \;\&\; M+2 < N$$
$$\&\; AMN = A(M+1)$$

One of the extra assumptions necessary to conclude Q_2' from R_5' is that $I \ge J$. The fact that in R_5' possibly ("typically") $I < J$ suggests J-I as counter within 'split'. A way to decrease the counter is to increase I or to decrease J; hence:

$$\{R_5'\}\ A(I) < AMN;\ I := I+1\ \{R_5'\}$$

$$\{R_5'\}\ A(J) > AMN;\ J := J-1\ \{R_5'\}$$

The validity of these verification conditions illustrates the well-known technique of the "sentinel". If $A(I) < AMN$, then we must have $A(I) < A(N)$ and, hence, $I < N$; therefore, $I \le N$ is still true after $I := I+1$. Here A(N) is the sentinel that prevents I from leaving the sequence to be split: an explicit test $I < N$ is superfluous. This fact is exploited by Knuth [5] in his "fast inner loop" for quicksort, and also by Singleton in his algorithm [7] which not only incorporates the fast inner loop but also the use of the median of a sample of 3 as separating element in split.

We must now take care of the situation

$$R_6' \stackrel{def}{=} R_5' \;\&\; A(I) \ge AMN \ge A(J)$$

If we also know that $I < J$, then R_5' can be made true again after decreasing the counter:

$$\{R_5'\}\ A(I) \ge AMN \ge A(J)\ \{R_6'\}$$

$$\{R_6'\}\ I < J;\ \text{swap}\ (A,I,J);\ I,J := I+1,\ J-1\ \{R_5'\}$$

If, on the other hand, $I \ge J$, then we are almost done, because R_6' & $I \ge J$ implies

$$A(M..J-1) \le AMN \le A(J+1..N) \;\&\; AMN \ge A(J) \;\&\; AMN = A(M+1)$$

Hence we can finish splitting with

$$\{R_6'\}\ \ I \ge J;\ \text{swap}\ (A,J,M+1)\ \{Q_2'\}$$

The counter J-I is bounded below, always integer, is never increased, and is decreased along the prickly arcs in flowgraph 5.1. Hence there can be no infinite computations. The absence of blocked computations depends on the fact, which can be also observed in flowgraph 5.1, that for each node except Q_2' and Q_6' each state is in at least one of the guards beginning a command on an outgoing arc and that the remaining commands terminate for all inputs which they encounter in computations of the flowgraph.

$\{ Q_1' \}$ A(M) $\leq$ A(N) $\{ R_1' \stackrel{def}{=} Q_1' \ \&\ (A(M) \leq A(N)) \}$

$\{ Q_1' \}$ A(M) > A(N); swap (A,M,N) $\{ R_1' \}$

$\{ R_1' \}$ M+1 $\geq$ N $\{ Q_6' \}$

$\{ R_1' \}$ M+1 < N; MN := (M+N)/2 $\{ R_2' \stackrel{def}{=} R_1' \ \&\ (M+1 < N) \ \&\ (MN = (M+N)/2) \}$

$\{ R_2' \}$ A(M) $\leq$ A(MN) $\{ R_3' \stackrel{def}{=} R_2' \ \&\ (A(M) \leq A(MN)) \}$

$\{ R_2' \}$ A(M) > A(MN); swap (A,M,MN) $\{ R_4' \}$

$\{ R_3' \}$ A(MN) $\leq$ A(N) $\{ R_4' \stackrel{def}{=} R_3' \ \&\ (A(MN) \leq A(N)) \}$

$\{ R_3' \}$ A(MN) > A(N); swap (A,MN,N) $\{ R_4' \}$

$\{ R_4' \}$ M+2 $\geq$ N $\{ Q_6' \}$

$\{ R_4' \}$ M+2 < N; AMN := A(MN); A(MN) := A(M+1); A(M+1) := AMN ;I,J := M+2, N-1 $\{ R_5' \}$

$\{ R_5' \}$ A(I) < AMN; I := I+1 $\{ R_5' \}$

$\{ R_5' \}$ A(J) > AMN; J := J-1 $\{ R_5' \}$

$\{ R_5' \}$ A(I) $\geq$ AMN $\geq$ A(J) $\{ R_6' \stackrel{def}{=} R_5' \ \&\ (A(I) \geq AMN \geq A(J)) \}$

$\{ R_6' \}$ I < J; swap (A,I,J); I,J := I+1, J-1 $\{ R_5' \}$

$\{ R_6' \}$ I $\geq$ J; swap (A,M+1,J) $\{ Q_2' \}$

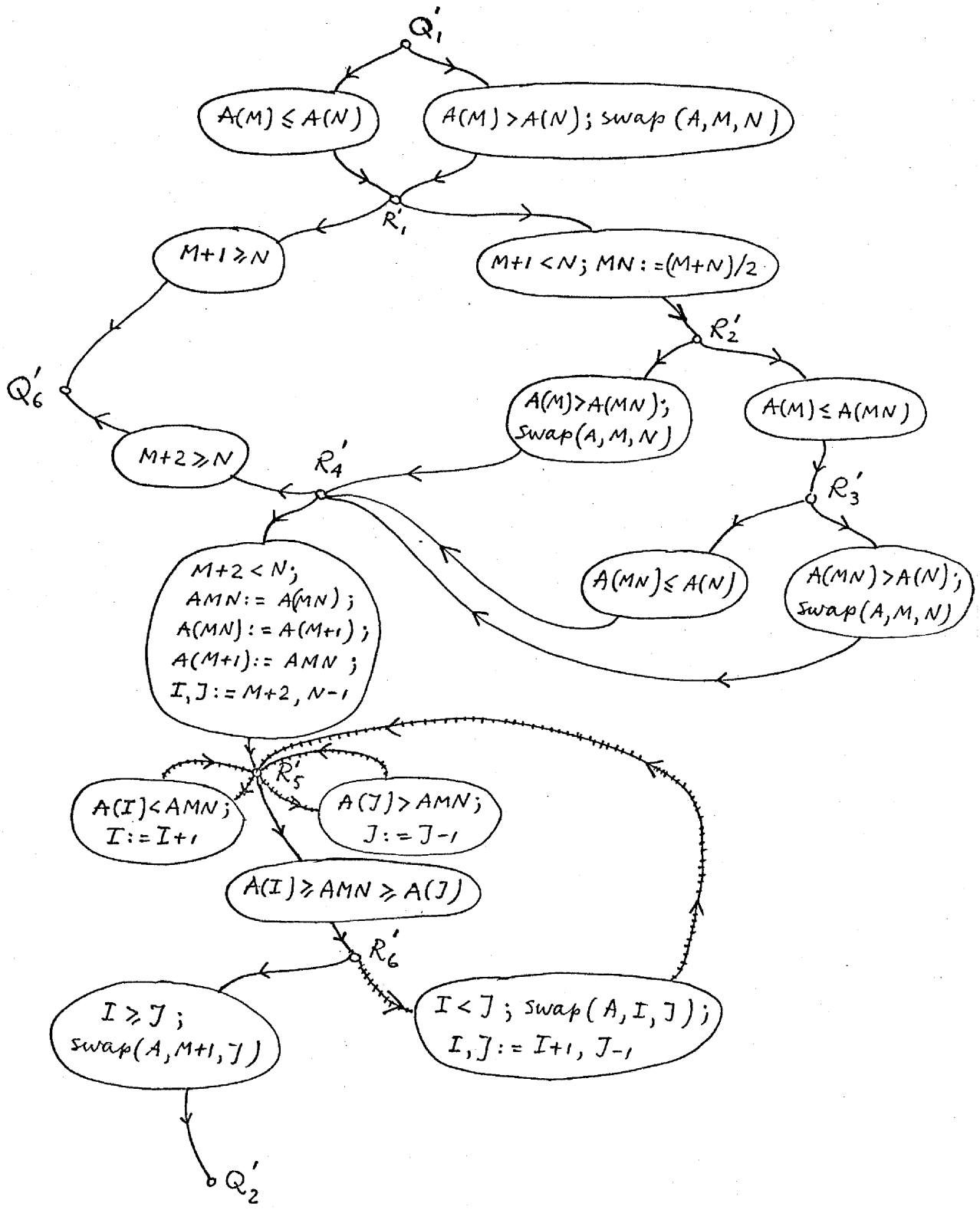
where

Q_1' = (P' & M < N)

R_5' = (P' & A(M..I-1) $\leq$AMN $\leq$ A(J+1..N)
& M+1 < I $\leq$ N & M $\leq$ J < N & M+2 < N
& AMN = A(M+1)
)

Q_2' = (P' & A(M..J-1) $\leq$ A(J) $\leq$ A(J+1..N)
& M $\leq$ J $\leq$ N
)

Flowgraph 5.1



Pictorial representation of the verification conditions in flowgraph 5.1

6. The final result

The final result may now be obtained by the following steps:

a) Flowgraph 5.1 replaces the line

$\{Q_1'\}$ split $\{Q_2'\}$

in flowgraph 4.1.

b) The individual commands of the resulting flowgraph are translated to FORTRAN.

c) The result is translated to FORTRAN according to the description in [2].

d) The most obvious optimizations are carried out in the result, with the program fragment below as final result.

The resulting algorithm is much like Singleton's in [7], which is based on Scowen's modification [6] of Hoare's original "quicksort" [3]. Singleton also used the median of a sample of three as bounding element in split.

```
     10 PTR=0
        M=1
        N=NO
C    ASSERTION Q1
     20 IF (A(M).LE.A(N)) GOTO 30
        SWAP = A(M)
        A(M) = A(N)
        A(N) = SWAP
C    ASSERTION R1
     30 IF (M+1.GE.N) GOTO 130
        MN=(M+N)/2
        IF (A(M).LE.A(MN)) GOTO 40
        SWAP=A(M)
        A(M)=A(MN)
        A(MN)=SWAP
        GOTO 50
C    ASSERTION R3
```

```
   40 IF (A(MN).LE.A(N)) GOTO 50
      SWAP=A(MN)
      A(MN)=A(N)
      A(N)=SWAP
C  ASSERTION R4
   50 IF (M+2.GE.N) GOTO 130
      AMN=A(MN)
      A(MN)=A(M+1)
      A(M+1)=AMN
      I=M+2
      J=N-1
C  ASSERTION R5
   60 IF (A(I).GE.AMN) GOTO 70
      I=I+1
      GOTO 60
   70 IF (A(J).LE.AMN) GOTO 80
      J=J-1
      GOTO 60
   80 IF (I.GE.J) GOTO 90
      SWAP=A(I)
      A(I)=A(J)
      A(J)=SWAP
      I=I+1
      J=J-1
      GOTO 60
   90 A(M+1)=A(J)
      A(J)=AMN
C  ASSERTION Q2
      IF (J-M.GT.1) GOTO 100
      IF (N-J.LE.1) GOTO 130
      M=J+1
      GOTO 20
```

```
C  ASSERTION Q4
   100 IF (N-J.LT.J-M) GOTO 110
       PTR=PRT+1
       LEFT(PTR)=J+1
       RIGHT(PTR)=N
       N=J-1
       GOTO 20
C  ASSERTION Q5
   110 IF (N-J.LE.1) GOTO 120
       PTR=PTR+1
       LEFT(PTR)=M
       RIGHT(PTR)=J-1
       M=J+1
       GOTO 20
   120 N=J-1
       GOTO 20
C  ASSERTION Q6
   130 IF (PTR.LT.1) RETURN
       M=LEFT(PTR)
       N=RIGHT(PTR)
       PTR=PTR-1
       GOTO 20
       END
```

NOTE: An assertion mentioned in a comment is true just before executing the line following the comment.

7. Acknowledgment

The research reported here was supported by research grants from the University of Waterloo and from the National Research Council.

8. References to the literature

1. M.H. van Emden: Increasing the efficiency of quicksort. Comm. ACM 13 (1970) 563-567

2. M.H. van Emden: Unstructured Systematic Programming. Research Report CS-76-09, Dept. of Computer Science University of Waterloo

3. C.A.R. Hoare: Algorithm 64: quicksort. Comm. ACM 4 (1961) 321

4. C.A.R. Hoare: Proof of correctness of data representations. Acta Informatica 1 (1972) 271-281

5. D.E. Knuth: Structured programming with goto statements. Computing Surveys 6 (1974) 261-302

6. R.S. Scowen: Algorithm 271: Quickersort. Comm. ACM 8 (1965) 669-670

7. R.C. Singleton: Algorithm 347: SORT Comm. ACM 12 (1969) 185-186