Proof of a Program: FIND

C. A. R. Hoare
Queen's University, Belfast, Ireland*

A proof is given of the correctness of the algorithm "Find." First, an informal description is given of the purpose of the program and the method used. A systematic technique is described for constucting the program proof during the process of coding it, in such a way as to prevent the intrusion of logical errors. The proof of termination is treated as a separate exercise. Finally, some conclusions relating to general programming methodology are drawn.

KEY WORDS AND PHRASES: proofs of programs, programming methodology, program documentation, program correctness, theory of programming
CR CATEGORIES: 4.0, 4.22, 5.21, 5.23, 5.24

1. Introduction

In a number of papers [1, 2, 3] the desirability of proving the correctness of programs has been suggested and this has been illustrated by proofs of simple example programs. In this paper the construction of the proof of a useful, efficient, and nontrivial program, using a method based on invariants, is shown. It is suggested that if a proof is constructed as part of the coding process for an algorithm, it is hardly more laborious than the traditional practice of program testing.

2. The Program "Find"

The purpose of the program Find [4] is to find that element of an array $A[1:N]$ whose value is fth in order of magnitude; and to rearrange the array in such a way that this element is placed in $A[f]$; and furthermore, all elements with subscripts lower than f have lesser values, and all elements with subscripts greater than f have greater values. Thus on completion of the program, the following relationship will hold:

$$A[1], A[2], \cdots, A[f-1] \leq A[f] \leq A[f+1], \cdots, A[N]$$

This relation is abbreviated as Found.

One method of achieving the desired effect would be to sort the whole array. If the array is small, this would be a good method; but if the array is large, the time taken to sort it will also be large. The Find program is designed to take advantage of the weaker requirements to save much of the time which would be involved in a full sort.

The usefulness of the Find program arises from its application to the problem of finding the median or other quantiles of a set of observations stored in a computer array. For example, if N is odd and f is set to $(N + 1)/2$, the effect of the Find program will be to place an observation with value equal to the median in $A[f]$. Similarly the first quartile may be found by setting f to $(N + 1)/4$, and so on.

The method used is based on the principle that the desired effect of Find is to move lower valued elements of the array to one end—the "left-hand" end—and higher valued elements of the array to the other end—the "right-hand" end. (See Table I(a)). This suggests that the array be scanned, starting at the left-hand end and moving rightward. Any element encountered which is small will remain where it is, but any element which is large should be moved up to the right-hand end of the array, in exchange for a small one. In order to find such a small element, a separate scan is made, starting at the right-hand end and moving leftward. In this scan, any large element encountered remains where it is; the first small element encountered is moved down to the left-hand end in exchange for the large element already encountered in the rightward scan. Then both scans can be resumed until the next exchange is necessary. The process is repeated until the scans meet somewhere in the middle of the array. It is then known that all elements to the left of this meeting point will be small, and all elements to the right will be large. When this condition holds, we will say that the array is *split at* the given point into two parts (see Table I(b)).

The reasoning of the previous paragraph assumes that there is some means of distinguishing small elements from large ones. Since we are interested only in their comparative values, it is sufficient to select the value of some arbitrary element before either of the scans starts; any element with lower value than the selected element is counted as small, and any element with higher value is counted as large. The fact that the discriminating value is arbitrary means that the place where the two scans will meet is also arbitrary; but it does not affect the fact that the array will be split at the meeting point, wherever that may be.

Now consider the question on which side of the split the fth element in order of value is to be found. If the split is to the right of $A[f]$, then the desired element must of necessity be to the left of the split, and all elements to the right of the split will be greater than it. In this case, all elements to the right of the split can be ignored in any future processing, since they are already in their proper

* Department of Computer Science

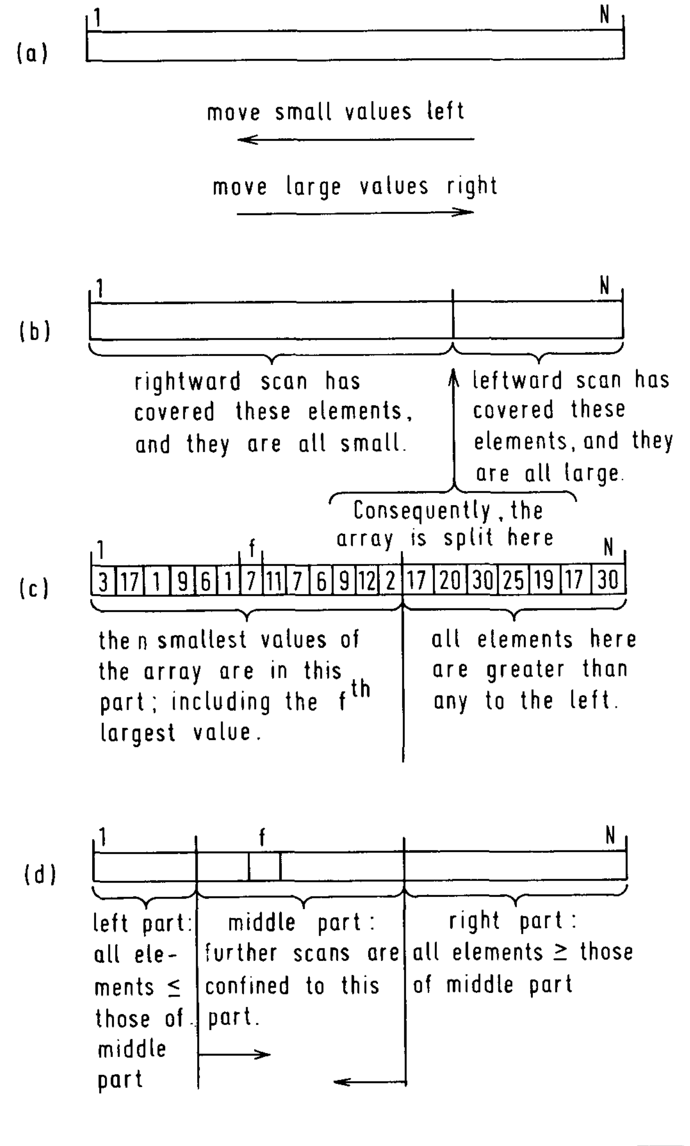
place, namely to the right of $A[f]$ (see Table I(c)). Similarly, if the split is to the left of $A[f]$, the element to be found must be to the right of the split, and all elements to the left of the split must be equal or less than it; furthermore, these elements can be ignored in future processing.

In either case, the program proceeds by repeating the rightward and leftward scans, but this time one of the scans will start at the split rather than at the beginning of the array. When the two scans meet again, it will be known that there is a second split in the array, this time perhaps on the other side of $A[f]$. Thus again, we may proceed with the rightward and leftward scans, but we start the rightward scan at the split on the left of $A[f]$ and the leftward scan at the split on the right, thus confining attention only to that part of the array that lies between the two splits; this will be known as the *middle part* of the array (see Table I(d)).

When the third scan is complete, the middle part of the array will be split again into two parts. We take the new middle part as that part which contains $A[f]$ and repeat the double scan on this new middle part. The process is repeated until the middle part consists of only one element, namely $A[f]$. This element will now be equal to or greater than all elements to the left and equal to or less than all elements to the right; and thus the desired result of Find will be accomplished

This has been an informal description of the method used by the program Find. Diagrams have been used to convey an understanding of how and why the method works, and they serve as an intuitive proof of its correctness. However, the method is described only in general terms, leaving many details undecided; and accordingly, the intuitive proof is far from watertight. In the next section, the details of the method will be filled in during the process of coding it in a formal programming language; and simultaneously, the details of the proof will be formalized in traditional logical notation. The end product of this activity will be a program suitable for computer execution, together with a proof of its correctness. The reader who checks the validity of the proof will thereby convince himself that the program requires no testing.

TABLE I


(a)
1
N
move small values left
move large values right
(b)
1
N
rightward scan has covered these elements, and they are all small.
leftward scan has covered these elements, and they are all large.
Consequently, the array is split here
(c)
1
f
N
3 17 1 9 6 1 7 11 7 6 9 12 2 17 20 30 25 19 17 30
the n smallest values of the array are in this part; including the f th largest value.
all elements here are greater than any to the left.
(d)
1
f
N
left part: all elements ≤ those of middle part
middle part: further scans are confined to this part.
right part: all elements ≥ those of middle part

3. Coding and Proof Construction

The coding and proof construction may be split into several stages, each stage dealing with greater detail than the previous one. Furthermore, each stage may be systematically analyzed as a series of steps.

3.1. Stage 1: Problem Definition

The first stage in coding and proof construction is to obtain a rigorous formulation of what is to be accomplished, and what may be assumed to begin with. In this case we may assume

(a) The subscript bounds of A are 1 and N.
(b) $1 \leq f \leq N$.

The required result is:

$$\forall p, q(1 \leq p \leq f \leq q \leq N \supset A[p] \leq A[f] \leq A[q]) \qquad \text{[Found]}$$

3.2. Stage 2: The General Method

(1) The first step in each stage is to decide what variables will be required to hold intermediate results of the program. In the case of Find, it will be necessary to know at all times the extent of the middle part, which is currently being scanned. This indicates the introduction of variables m and n to point to the first element $A[m]$ and the last element $A[n]$ of the middle part.

(2) The second step is to attempt to describe more for-

mally the purpose of each variable, which was informally described in the previous step. This purpose may be expressed as a formula of logic which is intended to remain true throughout the execution of the program, even when the value of the variable concerned is changed by assignment.[1] Such a formula is known as an *invariant*. As mentioned above, m is intended to point to the leftmost element of the middle part of the array; and the middle part at all times contains $A[f]$; consequently m is never greater than f. Furthermore, there is always a split just to the left of the middle part, that is between $m - 1$ and m. Thus the following formula should be true for m throughout execution of the program:

$$m \leq f \;\&\; \forall p, q(1 \leq p < m \leq q \leq N \supset A[p] \leq A[q]) \qquad [m\text{-invariant}]$$

Similarly, n is intended to point to the rightmost element of the middle part; it must never be less than f, and there will always be a split just to the right of it:

$$f \leq n \;\&\; \forall p, q(1 \leq p \leq n < q \leq N \supset A[p] \leq A[q]) \qquad [n\text{-invariant}]$$

(3) The next step is to determine the initial values for these variables. Since the middle part of the array is intended to be the part that still requires processing, and since to begin with the whole array requires processing, the obvious choice of initial values of m and n are 1 and N, respectively, indicating the first and last elements of the whole array. The code required is:

$m := 1;\;\; n := N$

(4) It is necessary next to check that these values satisfy the relevant invariants. This may be done by substituting the initial value for the corresponding variable in each invariant, and ensuring that the result follows from facts already known:

$$1 \leq f \leq N \supset 1 \leq f \;\& \\ \forall p, q(1 \leq p < 1 \leq q \leq N \supset A[p] \leq A[q]) \qquad [\text{Lemma 1}]$$

$$1 \leq f \leq N \supset f \leq N \;\& \\ \forall p, q(1 \leq p \leq N < q \leq N \supset A[p] \leq A[q]) \qquad [\text{Lemma 2}]$$

The quantified clause of each lemma is trivially true since the antecedents of the implications are always false.

(5) After setting the initial values, the method of the program is repeatedly to reduce the size of the middle part, until it contains only one element. This may be accomplished by an iteration of the form:

while $m < n$ **do** "reduce middle part"

(6) It remains to prove that this loop accomplishes the objectives of the program as a whole. If we write the body of the iteration properly (i.e. in such a way as to preserve the truth of all invariants) then all invariants will still be true on termination. Furthermore, termination will occur only when $m < n$ goes false. Thus it is necessary only to show that the combination of the truth of the invariants and the falsity of the while clause expression $m < n$ implies the truth of Found.

$$m \leq f \;\&\; \forall p, q(1 \leq p < m \leq q \leq N \supset A[p] \leq A[q]) \\ \& f \leq n \;\&\; \forall p, q(1 \leq p \leq n < q \leq N \supset A[p] \leq A[q]) \\ \& \neg\, m < n \\ \supset \forall p, q(1 \leq p \leq f \leq q \leq N \supset A[p] \leq A[f] \leq A[q]) \qquad [\text{Lemma 3}]$$

The antecedents imply that $m = n = f$. If $1 \leq p \leq f \leq q \leq N$, then either $p = f$, in which case $A[p] \leq A[f]$ is obvious, or $p < f$, in which case substituting f for both m and q in the first quantified antecedent gives $A[p] \leq A[f]$. A similar argument shows that $A[f] \leq A[q]$.

At this point, the general structure of the program is as follows:

```
m := 1; n := N;
while m < n do "reduce middle part"
```

Furthermore, this code has been proved to be correct, provided that the body of the contained iteration is correct.

3.3. Stage 3: Reduce the Middle Part

(1) The process for reducing the middle part involves a scan from the left and from the right. This requires two pointers, i and j, pointing to elements $A[i]$ and $A[j]$ respectively. In addition, a variable r is required to hold the arbitrary value which has been selected to act as a discriminator between "small" and "large" values.

(2) The i pointer is intended to pass over only those array elements with values smaller than r. Thus all array elements strictly to the left of the currently scanned element $A[i]$ will be known always to be equal to or less than r:

$$m \leq i \;\&\; \forall p(1 \leq p < i \supset A[p] \leq r) \qquad [i \text{ invariant}]$$

Similarly the j pointer passes over only large values, and all elements strictly to the right of the currently scanned element $A[j]$ are known always to be equal to or greater than r:

$$j \leq n \;\&\; \forall q(j < q \leq N \supset r \leq A[q]) \qquad [j\text{-invariant}]$$

Since the value of r does not change, there is no need for an r-invariant.

(3) The i pointer starts at the left of the middle part, i.e. at m; and the j pointer starts at the right of the middle part, i.e. at n. The initial value of r is taken from an arbitrary element of the middle part of the array. Since $A[f]$ is always in the middle part, its value is as good as any.

(4) The fact that the initial values satisfy the i- and j-invariants follows directly from the truth of the corre-

[1] Except possibly in certain "critical regions."

sponding m- and n-invariants; this is stated formally in the following lemmas:

$$f \leq N \mathbin{\&} m \leq f \mathbin{\&} \forall p, q(1 \leq p < m \leq q \leq N \supset A[p] \leq A[q]) \supset m \leq m \mathbin{\&} \forall p(1 \leq p < m \supset A[p] \leq A[f])$$ [Lemma 4]

$$1 \leq f \mathbin{\&} f \leq n \mathbin{\&} \forall p, q(1 \leq p \leq n < q \leq N \supset A[p] \leq A[q]) \supset n \leq n \mathbin{\&} \forall q(n < q \leq N \supset A[f] \leq A[q])$$ [Lemma 5]

The first of these is proved by setting q to f and the second by setting p to f.

(5) After setting the initial values, the method is to repeatedly add one to i and subtract one from j, until they cross over. This may be achieved by an iteration of the form:

```
while $i \leq j$ do "increase $i$ and decrease $j$"
```

On exit from this loop, $j < i$ and all invariants are intended to be preserved.

If j and i cross over above f, the proposed method assigns j as the new value of n; if they cross over below f, i is assigned as the new value of m.

```
   if $f \leq j$ then $n := j$
else if $i \leq f$ then $m := i$
          else go to $L$
```

The destination of the jump will be determined later.

(6) The validity of these assignments is proved by showing that the new value of n or m satisfies the corresponding invariant whenever the assignment takes place. In these proofs it can be assumed that the i- and j-invariants hold; and furthermore, since the assignment immediately follows the iteration of (5), it is known that $j < i$. Thus the appropriate lemma is:

$$j < i \mathbin{\&} \forall p(1 \leq p < i \supset A[p] \leq r) \mathbin{\&} \forall q(j < q \leq N \supset r \leq A[q]) \supset \textbf{if } f \leq j \textbf{ then } f \leq j \mathbin{\&} \forall p, q(1 \leq p \leq j < q \leq N) \supset A[p] \leq A[q]) \textbf{ else if } i \leq f \textbf{ then } i \leq f \mathbin{\&} \forall p, q(1 \leq p < i \leq q \leq N \supset A[p] \leq A[q])$$ [Lemma 6]

The proof of this is based on the fact that if $1 \leq p \leq j < q \leq N$, then $p < i$ (since $j < i$), and both $A[p] \leq r$ and $r \leq A[q]$. Hence $A[p] \leq A[q]$. Similarly, if $1 \leq p < i \leq q \leq N$, then $j < q$, and the same result follows.

It remains to determine the destination of the jump **go to** L. This jump is obeyed only if $j < f < i$, and it happens that in this case it can be proved that the condition Found has already been achieved. It is therefore legitimate to jump straight to the end of the program. The lemma which justifies this is:

$$1 \leq f \leq N \mathbin{\&} j < f < i \mathbin{\&} \forall p(1 \leq p < i \supset A[p] \leq r) \mathbin{\&} \forall q(j < q \leq N \supset r \leq A[q]) \supset \forall p, q(1 \leq p \leq f \leq q \leq N \supset A[p] \leq A[f] \leq A[q])$$ [Lemma 7]

This may be readily proved: if f is put for q in the antecedent, we obtain $r \leq A[f]$. Similarly, putting f for p in the antecedent we obtain $A[f] \leq r$. Hence $A[f] = r$. If $1 \leq p \leq f \leq q \leq N$, then $1 \leq p < i$ (since $f < i$) and $j < q \leq N$ (since $j < f$) and hence the i-invariant states that $A[p] \leq r$ and the j-invariant states that $r \leq A[q]$. But r has already been proved equal to $A[f]$.

This concludes the outline of the program required to reduce the middle part:

```
   $r := A[f]$; $i := m$;  $j := n$;
   while $i \leq j$ do "increase $i$ and decrease $j$";
   if $f \leq j$ then $n := j$
else if $i \leq f$ then $m := i$
          else go to $L$
```

This program has been proved to be correct, in that it preserves the truth of both the m- and n-invariants, provided that the body of the contained loop preserves these invariants as well as the i- and j-invariants.

3.4. Stage 4: Increase i and Decrease j

At this stage there is no need to introduce further variables and no further invariants are required. The construction of the code is not therefore split into the steps as before.

The first action of this part of the program is to use the i-pointer to scan rightward, passing over all elements with value less than r. This is accomplished by the loop:[2]

```
while $A[i] < r$ do $i := i + 1$
```

The fact that this loop preserves the truth of the invariant is expressed in the obvious lemma:

$$A[i] \leq r \mathbin{\&} m \leq i \mathbin{\&} \forall p(1 \leq p < i \supset A[p] \leq r) \supset m \leq i + 1 \mathbin{\&} \forall p(1 \leq p < i + 1 \supset A[p] \leq r)$$ [Lemma 8][3]

[2] The reason for the strict inequality is connected with termination. See Section 4.

[3] This lemma is not strictly true for some implementations of computer arithmetic. Suppose that N is the largest number representable in the integer range, that $m = i = N$, and that modulo arithmetic is used. Then $i + 1$ will be the smallest number representable, and will certainly be less than m. The easiest way to evade this problem is to impose on the user of the algorithm the insignificant restriction that $N <$ maxint, where maxint is the largest representable integer.

The next action is to use the j-pointer to scan leftward, passing over all elements greater than r. This is accomplished by the loop:

```
while $r < A[j]$ do $j := j - 1$
```

which is validated by the truth of:

$$r \leq A[j] \;\&\; j \leq n \;\&\; \forall q(j < q \leq N \supset r \leq A[q])$$
$$\supset j - 1 \leq n \;\&\; \forall q(j - 1 < q \leq N \supset r \leq A[q]) \qquad \text{[Lemma 9]}$$

On termination of the first loop, it is known that $r \leq A[i]$, and on termination of the second loop $A[j] \leq r$. If i and j have not crossed over, an exchange of the elements they point to takes place. After the exchange, it is obvious that

$$A[i] \leq r \leq A[j],$$

and hence Lemmas 8 and 9 justify a further increase in i and decrease in j:

```
if $i \leq j$ then
begin "exchange $A[i]$ and $A[j]$";
  $i := i + 1$;  $j := j - 1$
end
```

Thus the process of increasing i and decreasing j preserves the truth of all the invariants, provided that the exchange of $A[i]$ and $A[j]$ does so, and the program takes the form:

```
while $A[i] < r$ do $i := i + 1$;
while $r < A[j]$ do $j := j - 1$;
if $i \leq j$ then
        begin "exchange $A[i]$ and $A[j]$";
          $i := i + 1$;  $j := j - 1$
        end
```

3.5. Stage 5: Exchange $A[i]$ and $A[j]$

The code for performing the exchange is:

$$w := A[i]; \quad A[i] := A[j]; \quad A[j] := w$$

Although this code uses a new variable w, there is no need to establish an invariant for it, since its value plays a purely temporary role.

The proof that the exchange preserves the invariants is not trivial, and depends critically on the fact that $i \leq j$. Let A' stand for the value of the array as a whole after the exchange has taken place. Then obviously:

$$A'[i] = A[j] \qquad (1)$$

$$A'[j] = A[i] \qquad (2)$$

$$\forall s(s \neq i \;\&\; s \neq j \supset A'[s] = A[s]) \qquad (3)$$

The preservation of the i-invariant is stated in the lemma:

$$m \leq i \leq j \;\&\; \forall p(1 \leq p < i \supset A[p] \leq r)$$
$$\supset m \leq i \;\&\; \forall p(1 \leq p < i \supset A'[p] \leq r) \qquad \text{[Lemma 10]}$$

This is proved by observing that if $p < i \leq j$ then $p \neq i$ and $p \neq j$ and by (3), $A'[p] = A[p]$.

Similarly the preservation of the j-invariant is guaranteed by the lemma:

$$i \leq j \leq n \;\&\; \forall q(j < q \leq N \supset r \leq A[q])$$
$$\supset j \leq n \;\&\; \forall q(j \leq q \leq N \supset r \leq A'[q]) \qquad \text{[Lemma 11]}$$

The proof likewise proceeds by observing that $i \leq j < q$ implies that $q \neq i$ and $q \neq j$, and therefore by (3), $A'[q] = A[q]$.

The preservation of the m-invariant is guaranteed by the truth of the following lemma:

$$m \leq i \leq j \;\&\; \forall p, q(1 \leq p < m \leq q \leq N \supset A[p] \leq A[q])$$
$$\supset \forall p, q(1 \leq p < m \leq q \leq N \supset A'[p] \leq A'[q]) \qquad \text{[Lemma 12]}$$

Outline proof:

Assume $1 \leq p < m \leq q \leq N$; hence $p \neq i$ and $p \neq j$ (since $p < m \leq i \leq j$). Therefore by (3),

$$A'[p] = A[p]. \qquad (4)$$

Substituting i and then j for q in the antecedent, we obtain $A[p] \leq A[i]$ and $A[p] \leq A[j]$. Consequently $A'[p] \leq A'[j]$ and $A'[p] \leq A'[i]$ (from (4), (1), and (2)). Furthermore, for all $q \neq i$ and $q \neq j$, $A'[p] = A[p] \leq A[q] = A'[q]$ (by (4) and (3)). Hence $A'[p] \leq A'[q]$ for all $q(m \leq q \leq N)$.

The preservation of the n-invariant is guaranteed by a similar lemma:

$$i \leq j \leq n \;\&\; \forall p, q(1 \leq p \leq n < q \leq N \supset A[p] \leq A[q])$$
$$\supset \forall p, q(1 \leq p \leq n < q \leq N \supset A'[p] \leq A'[q]) \qquad \text{[Lemma 13]}$$

The proof is very similar to that of Lemma 12, and is left as an exercise.

3.6. The Whole Program

The gradual evolution of the program code and proof through several stages has been carried out in the previous sections. In presenting the code of the program as a whole, the essential invariants and other assertions have been preserved as comments. Thus a well-annotated version of the program appears in Table II.

4. Termination

The proof given so far has concentrated on proving the correctness of the program supposing that it terminates; and no attention has been given to the problem of proving termination. It is easier in this case to prove termination of the inner loops first.

The proof of the termination of:

```
while $A[i] < r$ do $i := i + 1$
```

depends on the recognition that at all times there will be an element in the middle part to the right of $A[i]$ whose

Table II

```
begin
  comment This program operates on an array A[1:N], and a
    value of f(1 ≤ f ≤ N). Its effect is to rearrange the elements
    of A in such a way that:
      ∀p,q(1≤p≤f≤q≤N⊃A[p]≤A[f]≤A[q]);
  integer m, n;  comment
      m ≤ f & ∀p,q(1≤p<m≤q≤N⊃A[p]≤A[q]),
      f ≤ n & ∀p,q(1≤p≤n<q≤N⊃A[p]≤A[q]);
    m := 1;  n := N;
  while m < n do
  begin integer r, i, j, w;
    comment
        m ≤ i & ∀p(1≤p<i⊃A[p]≤r),
        j ≤ n & ∀q(j<q≤N⊃r≤A[q]);
    r := A[f];  i := m;  j := n;
    while i ≤ j do
    begin while A[i] < r do i := i + 1;
      while r < A[j] do j := j − 1
      comment  A[j] ≤ r ≤ A[i];
      if i ≤ j then
      begin w := A[i];  A[i] := A[j];  A[j] := w;
        comment  A[i] ≤ r ≤ A[j];
        i := i + 1;  j := j − 1;
      end
    end increase i and decrease j;
    if f ≤ j then n := j
  else if i ≤ f then m := i
    else go to L
  end reduce middle part;
L:
end Find
```

value is equal to or greater than r. This element will act as a "stopper" to prevent the value of i from increasing beyond the value n. More formally, it is necessary to establish an additional invariant for i, which is true before and during the loop; i.e. throughout execution of "reduce middle part." This invariant is:

$$\exists p(i \le p \le n \;\&\; r \le A[p]) \qquad (5)$$

Obviously if this is true, the value of i is necessarily bounded by n; it cannot increase indefinitely, and the loop must therefore terminate.

The fact that (5) is an invariant for the duration of the particular loop is established by the following lemmas:

$$m \le f \le n \supset \exists p(m \le p \le n \;\&\; A[f] \le A[p]) \qquad \text{[Lemma 14]}$$

Proof: take f for p.

$$A[i] < r \;\&\; \exists p(i \le p \le n \;\&\; r \le A[p])$$
$$\supset \exists p(i + 1 \le p \le n \;\&\; r \le A[p]) \qquad \text{[Lemma 15]}$$

Proof: consider the p whose existence is asserted by the antecedent. Since $r \le A[p]$ & $A[i] < r$, $p \ne i$. Hence $i + 1 \le p$.

$$r \le A[i] \;\&\; i + 1 \le j - 1 \;\&\; j \le n$$
$$\supset \exists p(i + 1 \le p \le n \;\&\; r \le A'[p]) \qquad \text{[Lemma 16]}$$

Proof: Take j for p. Then $A'[p] = A'[j] = A[i] \ge r$.

Lemma 14 shows that the invariant is true after the initialization of "reduce middle part." Lemma 15 shows that the invariant is preserved by **while** $A[i] < r$ **do** $i := i + 1$, and Lemma 16 shows that the invariant is preserved by the final compound statement of "reduce middle part," providing that $i \le j$ after the execution of this statement. Since the body of the loop is not reentered unless this condition is satisfied, the invariant is unconditionally true at the beginning of the second and subsequent repetitions of "reduce middle part."

The termination of the loop

while $r < A[j]$ **do** $j := j - 1$

is established in a very similar manner. The additional invariant is

$$\exists q(m \le q \le j \;\&\; A[q] \le r) \qquad (6)$$

and the lemmas required are Lemma 14 and

$$r < A[j] \;\&\; \exists q(m \le q \le j \;\&\; A[q] \le r)$$
$$\supset \exists q(m \le q \le j - 1 \;\&\; A[q] \le r) \qquad \text{[Lemma 17]}$$

$$A[j] \le r \;\&\; i + 1 \le j - 1 \;\&\; m \le i$$
$$\supset \exists q(m \le q \le j - 1 \;\&\; A'[q] \le r) \qquad \text{[Lemma 18]}$$

The proofs of these lemmas are very similar to those for Lemmas 15 and 16.

This proof of termination is more than usually complex; if the program were rewritten to include an extra test ($i \le n$ or $m \le j$) in each loop, termination would have been obvious. However, the innermost loops would have been rather less efficient.

The proof of termination of the middle loop is rather simpler. The loop for increasing i and decreasing j must terminate; since if the conditional statement it contains is not obeyed then j is already less than i, and termination is immediate; whereas if $j \ge i$, then i is necessarily incremented and j decremented, and they must cross over after a finite number of such operations.

Proof of the termination of the outermost loop depends on the fact that on termination of the middle loop both $m < i$ and $j < n$. Therefore whichever one of the assignments $m := i$ or $n := j$ is executed, the distance between n and m is strictly decreased. If neither assignment is made, **go to** L is executed, and terminates the loop immediately.

The proof that at the end of the middle loop both $m < i$ and $j < n$ depends on the fact that on the first execution of the loop body the conditional **if** $i \le j$ **then** ... is actually executed. This is because at this stage $A[f]$ is still equal to r, and therefore the rightward scan of i cannot pass over $A[f]$. Similarly the leftward scan of j cannot pass over $A[f]$. Thus on termination of both innermost loops $i \le f \le j$. Thus the condition $i \le j$ is satisfied, and i is necessarily incremented, and j is necessarily decremented. Recall that this reasoning applies only to the first time round this loop—but once is enough to ensure $m < i$ and $j < n$, since i is a nondecreasing quantity and j is a nonincreasing quantity.

5. Reservation

In the proof of Find, one very important aspect of correctness has not been treated, namely, that the program merely rearranges the elements of the array A, without changing any of their values. If this requirement were not stated, there would be no reason why the program Find should not be written trivially:

```
for i := 1 step 1 until N do
  A [i] := i
```

since this fully satisfies all the other criteria for correctness.

The easiest way of stating this additional requirement is to forbid the programmer to change the array A in any other way than by exchanging two of its elements. This requirement is clearly met by the Find program and not by its trivial alternative.

If it is desired to formulate the requirement in terms of conditions and invariants, it is necessary to introduce the concept of a permutation; and to prove that for arbitrary A_0,

A is a permutation of A_0, [Perm]

is an invariant of the program. Informally this may be proved in three steps:

(a) "exchange $A[i]$ and $A[j]$," is the only part of the program which changes A,
(b) exchanging is a permutation,
(c) the composition of two permutations is also a permutation.

The main disadvantages of the formal approach are illustrated by this example. It is far from obvious that the invariance of Perm expresses exactly what we want to prove about the program; when the definition of Perm is fully and formally expressed, this is even less obvious; and finally, if the proof is formulated in the manner of the proofs of the other lemmas of this paper, it is very tedious.

Another problem which remains untreated is that of proving that all subscripts of A are within the bounds 1 to N.

6. Conclusion

This paper has illustrated a methodology for systematic construction of program proofs together with the programs they prove. It uses a "top-down" method of analysis to split the process into a number of stages, each stage embodying more detail than the previous one; the proof of the correctness of the program at each stage leads to and depends upon an accurate formulation of the characteristics of the program to be developed at the next stage.

Within each stage, there are a number of steps: the decision on the nature of the data required; the formulation of the invariants for the data; the construction of the code; the formulation and proof of the lemmas. In this paper, the stages and steps have been shown as a continuous progress, and it has not been necessary to go back and change decisions made earlier. In practice, reconsideration of earlier decisions is frequently necessary, and this imposes on the programmer the need to reestablish the consistency of invariants, program, lemmas, and proofs. The motivation for taking this extra trouble during the design and coding of a program is that it is hoped to reduce or eliminate trouble at phases which traditionally come later—program testing, documentation, and maintenance.

Similar systematic methods of program construction are described in [5] and [6]; this present paper, however, places greater emphasis on the formalization of the characteristics of the program as an aid to the avoidance of logical and coding errors. In future, it may be possible to enlist the aid of a computer in formulating the lemmas, and perhaps even in checking the proofs [7, 8].

Acknowledgments. The author is grateful to the referee and to the retiring editor for his meticulous comments and general encouragement in the preparation of this paper.

Received September, 1969; revised May, 1970

REFERENCES

1. Naur, P. Proof of algorithms by general snapshots. *BIT 6* (1966) 310–316.
2. Dijkstra, E. W. A constructive approach to the problem of program correctness. *BIT 8* (1968), 174–186.
3. Hoare, C. A. R. An axiomatic approach to computer programming. *Comm. ACM 12*, 10 (Oct. 1969), 576–580, 583.
4. Hoare, C. A. R. Algorithm 65, Find. *Comm. ACM 4*, 7 (July 1961), 321.
5. Naur, P. Programming by action clusters. *BIT 9* (1969), 250–258.
6. Dijkstra, E. W. Structured Programming [EWD249] T.H.E. (privately circulated).
7. Floyd, R. W. Assigning meanings to programs. Proc. Amer. Math. Soc. Symposium in Applied Mathematics, Vol. 19, pp. 19–31.
8. King, J. C. A program verifier. Ph.D. Th., Carnegie-Mellon U., Pittsburg, Pa., Sept. 1969.