```
PROCEDURE AddUp(A: ARRAY OF REAL):REAL;
(*pre: none
 *post: result = Sum (i=0 to HIGH(A))A[i]
 *)
```

that is,

$$result = \sum_{i=0}^{\texttt{HIGH}(A)} A[i].$$

There is an obvious technique for doing this; we read through the elements of A with a variable subscript n and add them one by one into an accumulator S.

Now imagine freezing the computation at the point when we have read exactly n elements and added them all into S. Diagrammatically, the state of the computer can be seen in Figure 10.3

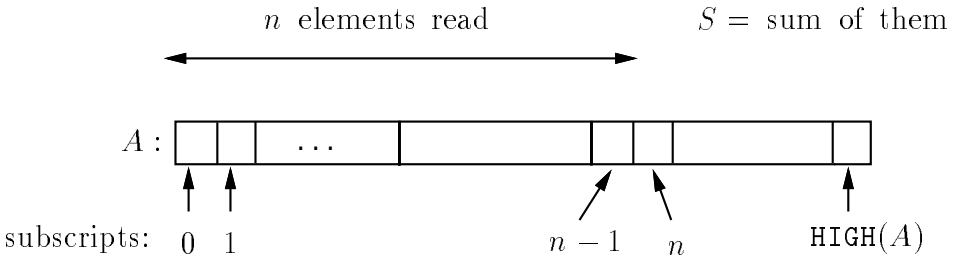


Figure 10.3

This diagram includes quite a lot. Importantly, it says exactly what values we intend to have in our variables n and S. An enormous number of programming errors are caused by imprecise ideas of what values variables are supposed to have. For instance, is $A[n]$ the last element read, or the next one to be read? Our diagram tells us. It also shows us that n varies from 0 (no elements read, at start) to $\texttt{HIGH}(A)+1$ (all the elements read, at finish). *Most important of all*, there is an easy link from the diagram to the post-condition. If we can ever get n to be $\texttt{HIGH}(A)+1$, then S must be the answer we want and all we need to do is `RETURN` S.

What the diagram is expressing is a *computational objective* — we intend to write the program so that after each iteration of the loop we have achieved a state as pictured by the diagram. At the same time, we want to push n up to $\texttt{HIGH}(A)+1$. We do not have to draw this diagram in a program comment; we can translate it into logic:

$$0 \leq n \leq \texttt{HIGH}(A)+1 \wedge S = \sum_{i=0}^{n-1} A[i]$$

This is the loop invariant. It also guides our programming:

- Initially (no elements read) we want $n = 0$ and $S = 0$ ($\sum_{i=0}^{-1} A[i]$, the empty sum).

- If $n = \texttt{HIGH}(A) + 1$, then S is the result we want and we can just return it.
- If $n \leq \texttt{HIGH}(A)$ then we want to read $A[n]$, add it to S, and increment n.

Thus the very act of formulating the invariant has subdivided our original problem into three smaller ones: initialization, finalization, and reestablishing the invariant. This is a very important aspect of the method.

And the variant? A natural number that decreases each time is the number of elements left to be read: this is $\texttt{HIGH}(A) + 1 - n$.

In effect we have now proved that the algorithm works, but we have not written the program yet! For the sake of our idiot computer, we must implement the algorithm in Modula-2:

```
PROCEDURE AddUp (A: ARRAY OF REAL):REAL;
(*pre: none
 *post: result = Sum (i=0 to HIGH(A))A[i]
 *)
VAR n: CARDINAL;
  S: REAL;
BEGIN
  S:=0.0;
  n:=0.0;
(* Loop invariant:
 *0<=n<= HIGH(A)+1 & S= Sum(i=0 to n-1) A[i]
 *Variant = HIGH(A)+1-n
 *)
  WHILE n<= HIGH(A) DO
    S:=S+A[n];
    n:=n+1
  END;
  RETURN S
END AddUp;
```

This is exactly the quantity of comments you should use in practice: the specification and the invariant and variant. Once you have actually written down the invariant, it is relatively easy — for you or for anyone else who needs to look at your code — to check the minor details. For instance,

- Is the invariant established initially? Yes, easy.
- Is the post-condition set up at the end? Yes. When the loop has terminated, we know both that $0 \leq n \leq \texttt{HIGH}(A) + 1$ (from the invariant) and that $n > \texttt{HIGH}(A)$ (because the loop test failed). Hence n must be

exactly $\texttt{HIGH}(A)+1$. Then the other part of the invariant tells us that S is the required result, and all we have to do is return it.

- When $A[n]$ is read, is n within range as an array subscript? Yes. We know at that point that the loop test succeeded, so $n \leq \texttt{HIGH}(A)$: it is in range.
- Does the loop body reestablish the invariant? Yes, this is fairly easy to see.
- Does the loop body decrease the variant? Yes, n is increased (by 1), so $\texttt{HIGH}(A)+1-n$ is decreased.
- Can the loop variant go negative? No. When the loop body is entered, we know $n \leq \texttt{HIGH}(A)$, so the variant is at least 1. After that iteration, it has decreased by exactly 1, so it is still at least 0.

These are all specific questions that can be asked about the correctness of the program, and for all of them the answer depends on the loop invariant. No other possible mid-condition in this program plays such a crucial role.

10.5 Loop invariants as a programming technique

The whole technique comes into operation as soon as you decide to use a loop structure. First, ask what the computer is supposed to look like at intermediate stages. Do not think about the dynamics of this (a common trap for beginners is to try to make a loop invariant by forcing the loop body into a logical notation); you must imagine freezing the computation at a crucial point and giving a static description of the internal state. There is already a vague picture at the back of your mind, and that is what you must bring out. Diagrams are absolutely invaluable here.

Also remember that you must understand *at that exact point in the computation* what the value of each computer variable signifies. If you do not know what values they are supposed to be storing, you will never be able to use those values correctly.

A critical test of the diagram is that under certain conditions (for example, $n = \texttt{HIGH}(A)+1$ in the `AddUp` example) you must be able to use the information carried by the diagram to arrive at the post-condition. The loop test should be the negation of these conditions (because you continue looping `WHILE` the conditions fail). At this point it is often easy to see a loop variant — the loop test is often equivalent to $\mathit{variant} > 0$.

Next, formalize the picture in logic to obtain a loop invariant. Perhaps your picture is incomplete; you will realize this later because you will find you do not quite understand how the program is supposed to be working. Then you fill in more details in the picture and refine the invariant. You now have an incomplete implementation:

```
PROCEDURE ...;
(*pre: ...
 *post: ...
 *)
VAR ... ;
BEGIN
  Initialize; (* Remains to be written *)
(*loop invariant: ...
 *variant =
 *)
  WHILE loop test DO
    Loop Body (* Remains to be written *)
  END;
  Finalize (* Remains to be written *)
END ... ;
```

There are three pieces of code that remain to be written: the initialization, the loop body and the finalization. (You probably saw fairly clearly how the finalization would work when you formulated the invariant.) Hence the original programming problem has been divided into three. Moreover, because you have formulated the invariant and variant, each of these three pieces has a precise job to do, a 'subcontract' of the contract (specification) for the overall procedure. These subcontracts can be specified with 'local' pre- and post-conditions.

Piece of code	*Local pre-condition*	*Local post-condition*
Initialize	Overall pre-condition	Invariant
Loop body	Invariant $\wedge$ Loop test	Invariant $\wedge$variant $<$ variant$_0$
Finalize	Invariant $\wedge\neg$Loop test	Overall post-condition

(We assume as usual that there are no side-effects when you evaluate the Loop test.) If you can implement Initialize, Loop body and Finalize to satisfy these local specifications, then you know they will automatically fit together in the `WHILE` loop to implement the overall specification correctly.

10.6 FOR loops

`FOR` loops are obviously very similar to `WHILE` loops, and you may well be used to seeing our `WHILE` loop examples coded as `FOR` loops (for instance this is quite easy for `AddUp`). In fact every `FOR` loop can be translated into a `WHILE` loop (see Exercise 6), and it follows that one way to reason with `FOR` loops is to give loop invariants and variants for the corresponding `WHILE` loops.

However, we are not going to recommend this here. One reason is that, for the purposes of reasoning, the control variable, for example, the i in `FOR`

`i := ...`, is often still needed after the last iteration, whereas its value in the computer has evaporated by then and is no longer accessible from the program. This has the effect that the `FOR` loops fit uncomfortably with the loop invariant reasoning, and in this book you will see `FOR` loops used less often than you might expect.

Nevertheless, there are some applications where `FOR` loops are particularly natural, namely when the different iterations of the body are more or less independent of each other and could even be done in parallel. You might think of the `WHILE` loop as being good for temporal iteration ('this then this then this, etc.') and the `FOR` loop as more spatial, less ordered ('do all these').

Here is a typical example:

```
CONST Size = ...;
TYPE Matrix = ARRAY [1..Size],[1..Size] OF INTEGER;

PROCEDURE ZeroMatrix (VAR A: Matrix);
(*pre: none
 *post: (A)i,j:CARDINAL. (1<=i<=Size & 1<=j<=Size -> A[i,j]=0)
 *)
VAR i,j: CARDINAL;
BEGIN
  FOR i := 1 TO Size DO
    FOR j := 1 TO Size DO
      A[i,j] := 0
    END
  END
END ZeroMatrix;
```

(NOTE: The logical variables i and j in the post-condition, bound by the $\forall$, are formally quite different from the computer variables `i` and `j`. However, the structure of the post-condition — $\texttt{Size}^2$ checks of zeroness — is so similar to that of the code — $\texttt{Size}^2$ assignments to 0 — that it seems fussy to insist on different symbols.)

It is possible to translate the `FOR` loops into `WHILE` loops and give an invariant for each. If you try this, you will see how clumsy it is. It is much simpler to argue as follows. To show that the post-condition holds at the end, let I and J be natural numbers between 1 and `Size`; we must show that at the end $A[I,J] = 0$. This is so, because

1. there was an iteration of the `FOR` loops (namely with $i = I$ and $j = J$) in which $A[I,J]$ became 0; and
2. once that was done, none of the other iterations would ever undo it.

The pattern is quite general. You reason that everything necessary was done, and then (because the iterations are independent) never undone. Note that no special argument is needed to show termination. `FOR` loops are bound to terminate unless you have a `BY` part of 0, for example,

```
FOR i := 1 TO 2 BY 0 DO ... END
```

As a general rule of thumb, if the iterations are fixed in number and independent of each other, then try to find a simple argument such as the one above and use a `FOR` loop. Otherwise, use a loop invariant and `WHILE` loop.

10.7 Summary

- The method of loop invariants is the method of mid-conditions applied to `WHILE` loops.
- The invariant is a mid-condition that should always be true immediately before the loop test is evaluated.
- Do not confuse the loop invariant with the loop test. They are both logical conditions, but
  1. the loop invariant is a mid-condition, used in reasoning, not evaluated by the computer, and intended to be true right through to the end;
  2. the loop test is a Boolean expression, evaluated by the computer, and is bound to be false after the last iteration.
- The invariant arises first (in your reasoned programm*ing*) as a computational objective, often after drawing a diagram; when the reasoned program is completed, the invariant is used to give a correctness proof.
- The invariant is used to divide the overall problem into three: initialization, loop body, and finalization.
- The loop variant, a number, is like a recursion variant and is used to prove termination.
- `FOR` loops are best reserved for simpler problems in which the iterations are independent of each other.

10.8 Exercises

1. The problem is to implement the following specification:

   ```
   PROCEDURE Negs(A: ARRAY OF INTEGER):CARDINAL;
   (*pre: none
    *post: no. of subscripts for which A[i]<0
    *)
   ```

   The idea is to inspect the elements starting at $A[0]$ and working up to `HIGH`(A):

   (a) Draw a diagram to illustrate the array when n elements have been inspected — make it clear what are the subscripts of the

last element to have been inspected and the next element to be inspected.

(b) What values will n take as the program proceeds?

(c) Write down the implementation (Modula-2 code), including the loop invariant and variant as comments in the usual way. The invariant should in effect translate the diagram of (a) into mathematical form.

(d) Use the invariant and the failure of the loop test to show that the post-condition is set up.

(e) Show that whenever an array element is accessed, the subscript is within bounds.

NOTE: (c) contains the ingredients that you should write down in your practical programming.

2. Develop reasoned Modula-2 programs along the lines of Exercise 1 to solve the following problems about arrays:

(a) Find the minimum element in an array of integers.

(b) Find whether an array of integers is in ascending order.

(c) Find the length of an array of CHARs, on the understanding that if it contains the character `NUL` (assumed predefined as a constant), then that and any characters after it are not to be counted. (In other words, `NUL` is understood as a terminator.)

(d) Find the median of an array of reals, that is, the array value closest to the middle in the sense that as many array elements are smaller than it as are greater than it. Is the problem any easier if the array is known to be sorted?

3. Develop the procedure `Search`:

```
PROCEDURE Search(A: ARRAY OF INTEGER; x:INTEGER):CARDINAL;
(*pre: Sorted(A)
 *post:  result <= HIGH(A)+1
 &(A)i:CARDINAL
       ((i< result ->A[i]<x)
 &(result <=i<= HIGH(A)->A[i]>=x))*)
```

Use a 'linear' search, inspecting the elements of A one by one starting at A[0].
Explain how the post-condition is deduced at the end (this is where sortedness is needed).

4. Implement the procedure `IsIn`, using a call of `Search` (Exercise 3):

```
PROCEDURE IsIn(x: INTEGER; A:ARRAY OF INTEGER):BOOLEAN;
(*pre: Sorted(A)
 *post:  result <->(E)i:Cardinal (i<= HIGH(A) & A[i]= x)
 *)
```

Using the pre- and post-conditions of `Search` (not the code), prove that your implementation of `IsIn` works correctly. What this means is that in every place where a result is returned, you must show that it is the correct result.

5. Give `FOR` loop implementations of the following:
   (a) `IsIn`
   (b) `Copy`

```
PROCEDURE Copy(A: ARRAY OF INTEGER;
       VAR B: ARRAY OF INTEGER);
(* Copies A to B
 *pre: HIGH(A) = HIGH(B)
 *post: B=A
 *)
```

6. Show how a `FOR` loop

```
FOR i := a TO b BY c DO
   S
END
```

can be translated into a `WHILE` loop. There are some tricky points:
   (a) If c is negative the translation is different.
   (b) The intention is that b and c should be evaluated *only once*, at the beginning. Hence you must be careful if they are expressions containing variables (actually, Modula-2 forbids this for c).

7. Consider the following problem:

```
PROCEDURE Copy (n,Astart,Bstart: CARDINAL; A:ARRAY OF INTEGER;
                VAR B: ARRAY OF INTEGER);
(*copies n elements from A, starting at A[Astart],to B,
 *starting at B[Bstart].
 *)
```

   (a) Give a `FOR` loop implementation of this, including pre- and post-conditions. Give your reasoning to show that it works.
   (b) If the array `A` is large, you might be tempted to call `A` as a `VAR` parameter, since then a local copy of it would not be made for use by the `Copy` procedure. If you did that, what might go wrong in the case where `A` and `B` are the same array? Can you give a sensible specification that allows for this possibility?

Chapter 11

Binary chop

How do you look up a word, 'binary', say, in a dictionary? What you *do not* do is to look through all the words in order, starting at page 1, until you find the word you want. If the dictionary had 1170 pages, you might have to check all of them before you found your word (if it was 'zymurgy'). Instead, you open the dictionary about half way through, at 'meridian', and you see that 'binary' must be in one of the pages in your left hand. You divide those about half way through, at 'drongo', and again you see that 'binary' must come before that. Each time, you halve the number of pages in which your word might be:

Stage:	0	1	2	3	4	5	6	7	8	9	10	11
Pages left:	1170	585	293	147	74	37	19	10	5	3	2	1

Hence, you have only to check eleven pages before you find your word. This method is called the 'binary chop algorithm', and it relies crucially on the fact that the entries in the dictionary are in alphabetical order. It is a very important algorithm in computing contexts, and, what is more, it is a good example of an algorithm that is very easy to get wrong if you try to write the code without any preliminary thought.

There is another important lesson in this algorithm, namely that the natural order of writing a procedure is not necessarily from top to bottom. (This is similar to the way you write a natural deduction proof.) You know already that the loop invariant should generally be worked out before the code; here the most important piece of code to be fixed is the finalization part.

11.1 A telephone directory

To explore different possible ways in which the algorithm might be used, imagine a telephone directory stored on a computer as an array of records,

each record comprising a name, an address and a telephone number. The records are stored in alphabetical order of names, but for different records under the same name it is perhaps not worth ordering them any more precisely.

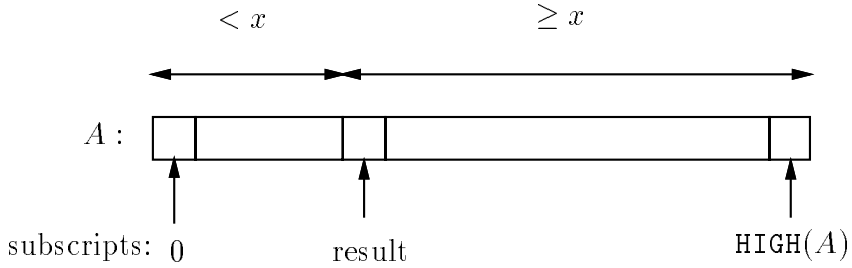
To look up a record, you supply a name and apply the binary chop algorithm. Although it is possible to use the algorithm simply to tell you *whether* the name is present in the directory, clearly in this case you need to know *where* it is so that you can then read the telephone number. Also, it is necessary to remember that there may be more than one record under the same name. It is most convenient if the algorithm tells you the subscript of the first one, so that you can then inspect the addresses one by one.

Now suppose that there is *no* record under the name you supplied. You might think that it is sufficient for the algorithm to tell you that, but consider the problem of updating the array. Any new record must be inserted in exactly the right place (after prising open a gap by shifting a lot of records up one place), and the binary chop algorithm can tell you where that right place is. (Note the payoffs here: lookup is very cheap, but update is expensive.)

Thus the algorithm apparently has many different situations to consider. It is an indication of the power of the algorithm that the cases are actually handled in a very uniform way.

11.2 Specification

Purely for the sake of example, let us take A to be an array of integers, its elements appearing in ascending order: if $i \leq j$, then $A[i] \leq A[j]$. (The method works not just for integers, but for any kind of data with an understood ordering — for instance, the telephone records described above, ordered alphabetically by name.) If x is an integer, the problem is to search for x in A. We can divide A into two blocks, one on the left where the elements are $< x$, and one on the right where they are $\geq x$. The answer is to be the subscript of the first element on the right:



We can translate this into logic. First, all elements with subscripts between 0 and *result*−1 inclusive are $< x$:

$$\forall i : nat \ \ (i < result \rightarrow A[i] < x) \tag{11.1}$$

Second, all elements with subscripts between result and $\texttt{HIGH}(A)$ inclusive are $\geq x$:

$$\forall i : nat \ \ (result \leq i \leq \texttt{HIGH}(A) \rightarrow A[i] \geq x) \tag{11.2}$$

Third, we should say what range the result will lie in. The extremes are when all the elements of the array are $\geq x$, when the result should be 0, and when all the elements are $< x$, when the result should be $\texttt{HIGH}(A) + 1$ (notice how in this case $A[result]$ is undefined):

$$0 \leq result \leq \texttt{HIGH}(A) + 1 \tag{11.3}$$

These three conditions will form the post-condition.

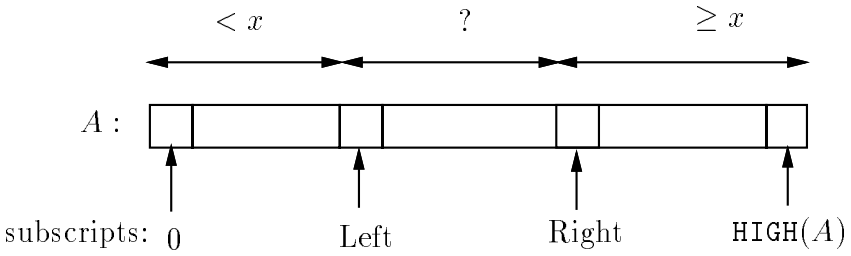
If x is present at all in the array, then we must have $A[result] = x$. (We shall prove that this holds a little later.) If x is absent, then either $A[result] > x$ or $result = \texttt{HIGH}(A) + 1$.

11.3 The algorithm

The algorithm uses two natural number variables *Left* and *Right*, which represent your two hands holding the dictionary: what you know at each stage is that the answer must be between *Left* and *Right*. At each iteration, you find the midpoint between *Left* and *Right* (call it *Middle*), and use that as a new *Left* or *Right*. Now this intuition is relatively simple, but it is tricky to say exactly what it means. Some points to be resolved are as follows:

- Should the answer be strictly between *Left* and *Right*, or not? Or strict at one end but not at the other? (Four possibilities here.) This is very important. If your ideas are not consistent throughout the program, then errors will arise.
- It is tempting to say something like $A[Left] < x$ and $A[Right] \geq x$, but might we ever want *Left* or *Right* to be $\texttt{HIGH}(A) + 1$, that is, not a valid subscript for A?

The key is to notice that *result* is used twice in the post-condition, once to show where the elements $< x$ are, and once to show where those $\geq x$ are. *Left* and *Right* can divide these two tasks between them: elements before *Left* are known to be $< x$, and elements at or after *Right* are known to be $\geq x$. In between, we do not know:



(Middle does not appear — it is used only for calculating within the loop body.)

We are trying to eliminate the '?' region, that is, to make *Left* and *Right* equal. Then we have essentially the same diagram as before, and *Left* (= *Right*) is the required result.

Initially, on the other hand, everything is '?' and so we want *Left*= 0 and *Right* = HIGH(A) + 1.

Actually, this idea, that when *Left*=*Right* we stop and return *Left* as result, is a fundamental design decision that strongly influences the rest: our initial decision is how to finalize! — though that should not come as a surprise by now. So the first program fragments we can write down are

```
(*loop invariant: ??? (formalizes picture)
 *variant = Right-Left  *)
WHILE Left < Right DO
  :
END;
RETURN Left
END Search;
```

There are different ideas, for instance 'when `Right = Left+1 return Right`'; which we could have chosen but we did not and, as it turns out, the method we have chosen is simpler.

Next, let us formulate the invariant. We have a picture already, but we also know that we are choosing $<$'s or $\leq$'s precisely to make the *Left* and *Right* parts of the invariant match parts 11.1 and 11.2 of the post-condition. Therefore, it has to be

$$\begin{array}{l} \mathit{Left} \leq \mathit{Right} \leq \mathtt{HIGH}(A) + 1 \\ \wedge \forall i : \mathit{nat}.\ ((i < \mathit{Left} \rightarrow A\mathtt{[}i\mathtt{]} < x) \wedge (\mathit{Right} \leq i \leq \mathtt{HIGH}(A) \rightarrow A\mathtt{[}i\mathtt{]} \geq x)) \end{array}$$

Let us also take the opportunity to say that the variant is $\mathit{Right} - \mathit{Left}$.

We have already dealt with the finalization; what next? The initialization is easy — we want $\mathit{Left} = 0$ and $\mathit{Right} = \mathtt{HIGH}(A) + 1$. All that remains is the loop body.

The idea is to find *Middle* between *Left* and *Right* and update either *Left* or *Right* depending on the value of $A\mathtt{[}\mathit{Middle}\mathtt{]}$. How should we do that? Let us be very careful to use the information precisely.

If $A\mathtt{[}\mathit{Middle}\mathtt{]} < x$, then *Middle* is in the '$< x$' area of the array; so *Left*, which is to be in the '?' area, can safely be set to $\mathit{Middle} + 1$. On the other hand, if $\mathtt{[}\mathit{Middle}\mathtt{]} \geq x$, then we must set *Right* to *Middle* (why not $\mathit{Middle} - 1$?). We have not said exactly what *Middle* is, but we have made a start on the loop body:

```
Middle := ?;
IF A[Middle] < x THEN
  Left := Middle+1
ELSE
  Right := Middle
END
```

It remains to assign a value to *Middle*, and it is important to see what precisely are the requirements here — all we know so far is that *Middle* should be (about) half way between *Left* and *Right*, or at least somewhere between them. Consider how the invariant $Left \leq Right$ is reestablished. The new *Left* may be $Middle + 1$, so we want $Middle + 1 \leq Right$, that is, $Middle < Right$. In the other case, the new *Right* is *Middle*, so we want $Left \leq Middle$. We can use a mid-condition to express these requirements as a computational objective:

```
Middle := ?;      (* Left <= Middle < Right *)
```

It is not difficult to see that *if* we can achieve this, then the rest of the loop body will reestablish the invariant and decrease the variant as well. We shall see soon that we can assign (*Left* + *Right*) `DIV 2`, the rounded-down average of *Left* and *Right*, to *Middle*. That is probably what you expected anyway, but more care is needed here than you might think. In Exercise 1 you will see a use of essentially the same algorithm in a different context where it is more natural to require $Left < Middle \leq Right$, and there it is necessary to use ((*Left* + *Right*) `DIV 2`) $+ 1$ — the problem comes when $Right = Left + 1$, so that $Middle = Left$.

11.4 The program

The program appears in Figure 11.1 Notice the order in which the parts appear:

1. The procedure heading, with specification and the fragments `BEGIN` and `END Search;`.
2. The framework for the loop: `WHILE Left < Right DO` and `END; RETURN Left;` also the slots for the invariant and variant (we have filled in the variant), and the `VAR` declarations.
3. The invariant, carefully formulated to match 2, and the post-condition.
4. The initialization.
5. Pieces of the loop body: an incompleted assignment statement `Middle := ?;`, and all of the `IF` statement.
6. The comment `(* Left <= Middle < Right *)`.
7. The assigned value `(Left+Right) DIV 2`.

At each stage, the choices to be made depended in a natural way on preceding choices, so the development had a certain logical inevitability.

```
PROCEDURE Search( A: ARRAY OF INTEGER; x: INTEGER) : CARDINAL;
(*pre: Sorted(A),
 *     i.e. (A)i,j:nat. ( i<=j<= HIGH(A) -> A[i]<=A[j] )
 *post: result <= HIGH(A)+1 & (A)i:nat.
 *      ( ( i<result -> A[i]<x ) & ( result<=i<=HIGH(A) -> A[i]>=x ) )
 *)
VAR Left,Right,Middle: CARDINAL;
BEGIN
  Left := 0;
  Right := HIGH(A)+1;
(*Loop invariant: Left <= Right <= HIGH(A)+1 & (A)i:nat.
 *           ( ( i< Left -> A[i]<x ) & ( Right<=i<= HIGH(A) -> A[i]>=x ) )
 *variant = Right-Left
 *)
  WHILE Left < Right DO
     Middle := (Left+Right) DIV 2;  (* Left <= Middle < Right *)
     IF A[Middle]<x
     THEN Left := Middle+1
     ELSE Right := Middle
     END
   END;
   RETURN Left
 END Search;
```

Figure 11.1

As an experiment, try to write the program code straight down from the top without thinking of invariants. You will probably find (everyone else does) that it is not easy to get it right.

11.5 Some detailed checks

We have already covered most of the important aspects of the invariant: that it is established correctly initially, that it is reestablished on each iteration, and that at the end it can be used to deduce the post-condition. The following are some small remaining questions.

At the end of each iteration, do we still have $Left \leq Right$*?* After the assignment to *Middle*, do we indeed have $Left \leq Middle < Right$? Because we are still looping, we know that $Left < Right$, that is, $Left \leq Right - 1$. Hence,

$$Middle = (Left + Right) \texttt{ DIV } 2 \geq (2 \times Left) \texttt{ DIV } 2 = Left$$
$$Middle = (Left + Right) \texttt{ DIV } 2 \leq (2 \times Right - 1) \texttt{ DIV } 2 = Right - 1 < Right$$

NOTE: the equality $(2 \times Right - 1)$ `DIV` $2 = Right - 1$ is correct according to the definition of Modula-2, given that $Right \geq 1$ (which we know because $Right > Left$): the fractional answer $(Right - 0.5)$ is *truncated* to $Right - 1$. But it is possible to imagine an integer division that might round the fractional answer $Right - 0.5$ up to $Right$. Therefore, if you translate this algorithm to languages other than Modula-2, you should check that their integer divisions behave as expected. Dijkstra and Feijen ('A Method of Programming') give a treatment that does not depend on the rounding method. However, their program only checks whether x is present in A and some elegance is lost when the method is extended to return the position — extra checking is needed to make up for the doubts about the integer division. In truth, the point of integer arithmetic is that it should be *exact*, and an inadequately specified integer division is a blunt instrument.

When A is subscripted, is the subscript within bounds? The only place is in '`IF A[`*Middle*`] ...`'. Can we guarantee that *Middle* $\leq$ `HIGH`(A)? Yes, because (as above) $Middle < Right$, and, by the invariant, $Right \leq$ `HIGH`$(A) + 1$.

Does the variant definitely decrease each time round? If *Right* is replaced by *Middle*, then it has definitely decreased; if *Left* is replaced by $Middle + 1$, then it has definitely increased. Either way, the variant has decreased.

11.6 Checking for the presence of an element

Suppose we only want to check whether x is present in A. If we calculate

```
r:= Search(A,x);
```

how can we use A, x and r to perform our check? Just to be sure, let us write down what we know about r solely from the post-condition for `Search`:

$$r \leq \texttt{HIGH}(A) + 1$$
$$\wedge \forall i : nat.\ ((i < r \rightarrow A[i] < x) \wedge (r \leq i \leq \texttt{HIGH}(A) \rightarrow A[i] \geq x)) \qquad (*)$$

If $A[r] = x$, then x must be present; while a quick look at one of the diagrams above makes it fairly clear that if $A[r] > x$ then x is absent. But wait! Is $A[r]$ defined? Not necessarily. r might be equal to `HIGH`$(A) + 1$; in this case, x is absent because all the elements are $< x$.

> Check that array subscripts are in bounds when you write the program, not when you run it.

The following is the program:

```
PROCEDURE IsIn(x: INTEGER; A: ARRAY OF INTEGER):BOOLEAN;
(*pre: (A)i,j:nat. (i<=j<= HIGH(A)->A[i] <=A[j])
 *post:  result <->(E)i:nat. (i<= HIGH(A) & A[i]= x)
 *)
VAR r: CARDINAL;
BEGIN
  r:= Search(A,x);
  RETURN r<= HIGH(A) AND A[r] = x
END IsIn;
```

The code above relies on Modula-2's short circuit evaluation. That is, `A[r] = x` will not be evaluated if `r > HIGH(A)`. In other languages, such as Pascal, Boolean expressions are evaluated completely even if the result is known after the first subexpression has been evaluated. The code after the `RETURN` would then need to be written as the following:

```
IF r <= HIGH(A)
THEN RETURN A[r] = x
ELSE RETURN FALSE
END
```

Let us show as rigourously as possible that the code for `IsIn` satisfies its specification: that if the returned Boolean value is `TRUE` then x is indeed present in A (that is, $\exists i : nat.\ (i \leq \texttt{HIGH}(A) \wedge A[i] = x)$), and that if `FALSE` is returned then x is absent (that is, $\neg\exists i : nat.(i \leq \texttt{HIGH}(A) \wedge A[i] = x)$).

[first case:]$r > \texttt{HIGH}(A)$, so `FALSE` is returned. We know that r is a natural number and that $r \leq \texttt{HIGH}(A) + 1$, so $r = \texttt{HIGH}(A) + 1$. Then from $(*), \forall i : nat.(i \leq \texttt{HIGH}(A) \rightarrow A[i] < x)$, in other words all the elements of A are $< x$ — so x must be absent. Note that the invalid array access $A[r]$ is not attempted here because of the way in which Modula-2 evaluates `AND`.

[second case:]$r \leq \texttt{HIGH}(A), A[r] = x$, so `TRUE` is returned. Certainly x is present, with subscript r.

[third case:]$r \leq \texttt{HIGH}(A), A[r] \neq x$, so `FALSE` is returned. Because $r \leq \texttt{HIGH}(A), (*)$ tells us that $A[r] \geq x$, so we must have $A[r] > x$. Now consider any subscript $i \leq \texttt{HIGH}(A)$. If $i < r$, then $(*)$ tells us that $A[i] < x$, while if $i \geq r$, then (using orderedness) $A[i] \geq A[r] > x$. Either way, $A[i] \neq x$, so $\neg\exists i : nat.\ (i \leq \texttt{HIGH}(A) \wedge A[i] = x)$.

11.7 Summary

- Binary chop is an important and efficient search algorithm if the elements are arranged in order. You should know it.
- The algorithm has many uses, but to use it effectively it is important to understand exactly what the result represents (that is, to have a clear

specification).

- There is a particular train of reasoning that leads to the algorithm easily; otherwise it is easy to get into a mess.

11.8 Exercises

1. What happens if you replace the assignment `Left := Middle+1` in `Search` by `Left := Middle`? (HINT: the invariant is still reestablished.) A common belief is that the problem can be corrected by stopping early, looping `WHILE Left+1 < Right`. Follow through this idea and see how it gives more complicated code.
2. The following is another version of `intsqrt` by the binary search algorithm:

```
intsqrt::num->num
||pre: x >= 0
||post: n = entier (sqrt x)
||      i.e. nat(n) & n^2 <= x & (n+1)^2 > x
||        where n = intsqrt x
intsqrt x = f x 0 (entier x)
            where f x l r = l,          if l = r
                          = ?,          if m*m <= x
                          = ?,          otherwise
                              where m = ?
 ||m satisfies some conditions
```

   Specify f precisely and in full, and complete the definition. (Beware! m is not $(l+r)div\ 2$, as you will see if you follow the method properly.)
3. Show that the specification of `Search` specifies the result uniquely. In other words, if there are two natural numbers r and r' that are both valid results, then $r = r'$.
   Use this to deduce the following. Suppose that in A there is exactly one index, i, for which $A[i] = x$. Then $i = Search(A, x)$.
4. There are other ways of giving the post-condition for `Search`. Here is one that translates the informal specification much more directly:

```
post1: (result <= HIGH(A)  & A[result]>=x
             & (A)i:nat. (i<=HIGH(A) & A[i]>=x->i>=result))
       \/(result=HIGH(A)+1
           & (A)i:nat. (i<=HIGH(A)->A[i]<x))
```

   Use natural deduction (together with standard properties of arithmetic) to prove that

$$pre \vdash (post \leftrightarrow post1)$$

where *pre* and *post* are the pre- and post-conditions for `Search` as originally specified, and *post*1 is as given above.
Can you think of any other equivalent post-conditions?

5. This question examines how you might use `Search` to update an ordered array.
First, a function `Search1` is intended to work in the same way as `Search`, but with a 'soft `HIGH`' called `High1` to allow for variable length lists of integers within a fixed length array. (We actually use a soft version of `HIGH`$(A)+1$. This allows us to specify an empty list by setting `High1` $= 0$.)

```
PROCEDURE Search1(A: ARRAY OF INTEGER;
  High1: CARDINAL; x: INTEGER):CARDINAL;
(*pre: High1<= HIGH(A)+1 & Sorted(A[0 to High1-1])
 *post: result <= High1  & (A)i:nat.
 *      ((i< result ->A[i]<x)
 *      & (result <=i< High1->A[i]>=x))
 *)
```

(It is obvious how to implement this: just initialize *Right* to `High1` instead of `HIGH`$(A)+1$ in the implementation for `Search`. Note that this works even in the case where `High1` $= 0$.)
Implement the following procedures, giving invariants and variants for all loops. The notation `A[i to j]` is introduced in Section 12.3.

```
PROCEDURE OpenUp(VAR A: ARRAY OF INTEGER; VAR High1: CARDINAL;
 NewGap: CARDINAL);
(*pre: NewGap <= High1<= HIGH(A)
 *post:  (E)x:Integer.
 *    A[0 to High1-1] =
 *      A_0[0 to NewGap-1]++[x]++A_0[NewGap to High1_0-1]
 *)
```

```
PROCEDURE Insert(VAR A: ARRAY OF INTEGER; VAR High1: CARDINAL;
 x:INTEGER);
(*pre: High1<= HIGH(A)  & Sorted(A[0 to High1-1])
 * post:  Sorted(A[0 to High1-1])
 *    & (E)s,t:[Integer]
 *      (A_0[0 to High1_0-1]=s++t
 * &A[0 to High1-1]=s++[x]++t)
 *)
```

(HINT: implement Insert using *Search*1 and *OpenUp*.)

6. Redo the proof that `IsIn` satisfies its specification using box proofs.

Chapter 12

Quick sort

12.1 Quick sort

Donald Knuth, in his book *Sorting and Searching*, gives an estimate of over 25 per cent for the proportion of computer running time that is spent on sorting. Whether this estimate is still accurate, we do not know, but his conclusion is still valid: whether (i) there are many important applications of sorting, or (ii) many people sort when they should not, or (iii) inefficient sorting algorithms are in common use, or something of all three, sorting is worthy of serious study as a practical matter.

As a general principle, if a program is used a lot then it is worth making it run quickly. In this chapter we present *quick sort,* an efficient sorting algorithm due to Tony Hoare. It is a good example of a combination of different kinds of argument. It is recursive, and the framework of the algorithm is very conveniently discussed as a Miranda function working on lists. However, when it is transferred to Modula-2 working on arrays, a significant improvement becomes possible using the 'Dutch national flag' algorithm, and this can be discussed using loop invariants — in fact it is a rather good example of a loop invariant that is a logical translation of a diagram.

12.2 Quick sort — functional version

The problem is, given a list, to sort it into order. We start off in Miranda. Since in Miranda datatypes have natural orderings, we do not need to say what our lists are lists of:

```
sort:: [*]->[*]
||pre:  none
||post: Sorted(sort xs) & Perm(sort xs,xs)
```

Idea: partition

It is so much easier to sort short lists than long ones that it helps to do a preliminary crude sort, a *partition* with respect to some key k (Figure 12.1).

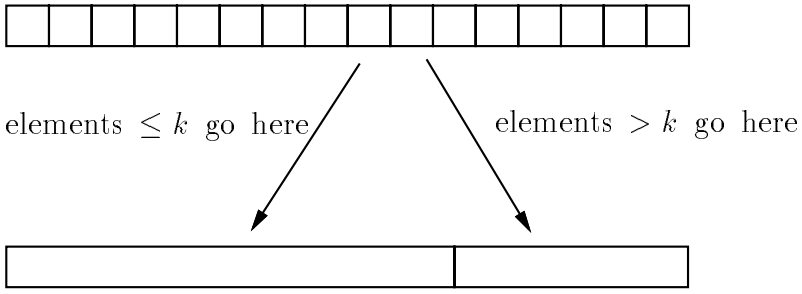


Figure 12.1

```
partition:: *->[*]->([*],[*])
||pre: none
||post: Perm(xs,ys++zs)
|| all elements of ys are <=k
|| all elements of zs are >k
||       where (ys,zs) = (partition k xs)
```

Note that the specification does not uniquely determine the function. If (ys, zs) is a possible result, so is (ys', zs') where ys', zs' are any permutations of ys and zs. It is simple to implement `partition` in Miranda, but we do not need to — it is the specification that is important, and in the end we will implement it by a totally imperative method. A pure functional quick sort is not terribly quick and uses lots of space.

Implementing quick sort

The idea is to do a partition first and then sort the two parts separately; they can be sorted using the same method, recursively. The head of the list can be the key:

```
qsort:: [*]->[*]
||pre: none
||post: Sorted(qsort (xs)) & Perm((qsort (xs)),xs)
||recursion variant = #xs
qsort []   = []
qsort (x:xs) = (qsort ys)++[x]++(qsort zs)
               where (ys,zs) = partition x xs
```

This is the essence of the recursion in the quick sort algorithm. To prove