is true for every integer $n \geq a$. Can we reach the same conclusion using the principle of ordinary mathematical induction? Yes! To see this, let $Q(n)$ be the property

$P(j)$ is true for each integer j with $a \leq j \leq n$.

Then use ordinary mathematical induction to show that $Q(n)$ is true for every integer $n \geq b$. That is, prove:

1. $Q(b)$ is true.
2. For each integer $k \geq b$, if $Q(k)$ is true then $Q(k+1)$ is true.

H 34. It is a fact that every integer $n \geq 1$ can be written in the form

$$c_r \cdot 3^r + c_{r-1} \cdot 3^{r-1} + \cdots + c_2 \cdot 3^2 + c_1 \cdot 3 + c_0,$$

where $c_r = 1$ or 2 and $c_i = 0$, 1, or 2 for each integer $i = 0, 1, 2, \ldots, r-1$. Sketch a proof of this fact.

H* 35. Use mathematical induction to prove the existence part of the quotient-remainder theorem. In other words, use mathematical induction to prove that given any integer n and any positive integer d, there exist integers q and r such that $n = dq + r$ and $0 \leq r < d$.

H* 36. Prove that if a statement can be proved by ordinary mathematical induction, then it can be proved by the well-ordering principle.

H 37. Use the principle of ordinary mathematical induction to prove the well-ordering principle for the integers.

ANSWERS FOR TEST YOURSELF

1. than one **2.** a; k; $P(k+1)$ **3.** one integer; integer in S; S contains a least element

5.5 Application: Correctness of Algorithms

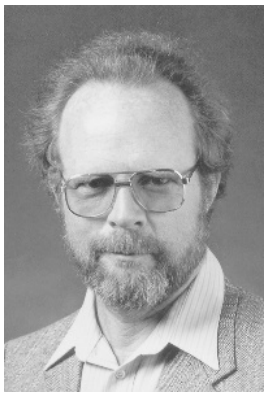
[P]rogramming reliably—must be an activity of an undeniably mathematical nature . . . . You see, mathematics is about thinking, and doing mathematics is always trying to think as well as possible. —Edsger W. Dijkstra (1981)

Edsger W. Dijkstra (1930–2002)

What does it mean for a computer program to be correct? Each program is designed to do a specific task—calculate the mean or median of a set of numbers, compute the size of the paychecks for a company payroll, rearrange names in alphabetical order, and so forth. We will say that a program is correct if it produces the output specified in its accompanying documentation for each set of input data of the type specified in the documentation.*

Most computer programmers write their programs using a combination of logical analysis and trial and error. In order to get a program to run at all, the programmer must first fix all syntax errors (such as writing **ik** instead of **if**, failing to declare a variable, or using a restricted keyword for a variable name). When the syntax errors have been removed, however, the program may still contain logical errors that prevent it from producing correct output. Frequently, programs are tested using sets of sample data for which the correct output is known in advance. And often the sample data are deliberately chosen to test the correctness of the program under extreme circumstances. But for most programs the number of possible sets of input data is either infinite or unmanageably large, and so no amount of program testing can give perfect confidence that the program will be correct for all possible sets of legal input data.

*Consumers of computer programs want an even more stringent definition of correctness. If a user puts in data of the wrong type, the user wants a decent error message, not a system crash.



Courtesy of Christiane Floyd

Robert W. Floyd (1936–2002)

Since 1967, with the publication of a paper by Robert W. Floyd,* considerable effort has gone into developing methods for proving programs correct at the time they are composed. One of the pioneers in this effort, Edsger W. Dijkstra, asserted that "we now take the position that it is not only the programmer's task to produce a correct program but also to demonstrate its correctness in a convincing manner."† Another leader in the field, David Gries, went so far as to say that "a program and its proof should be developed hand-in-hand, with the *proof* usually leading the way."** In this section we give an overview of the general format of correctness proofs and the details of one crucial technique, the *loop invariant procedure*, and we switch from using the term *program* to using the more general term *algorithm*.

Assertions

Consider an algorithm that is designed to produce a certain final state from a certain initial state. Both the initial and final states can be expressed as predicates involving the input and output variables. Often the predicate describing the initial state is called the **pre-condition for the algorithm**, and the predicate describing the final state is called the **post-condition for the algorithm**.

Example 5.5.1 **Algorithm Pre-Conditions and Post-Conditions**

Here are pre- and post-conditions for some typical algorithms.

a. Algorithm to compute a product of nonnegative integers

Pre-condition: The input variables m and n are nonnegative integers.
Post-condition: The output variable p equals mn.

b. Algorithm to find quotient and remainder of the division of one positive integer by another

Pre-condition: The input variables a and b are positive integers.
Post-condition: The output variables q and r are integers such that $a = bq + r$ and $0 \leq r < b$.

c. Algorithm to sort a one-dimensional array of real numbers

Pre-condition: The input variable $A[1], A[2], \ldots, A[n]$ is a one-dimensional array of real numbers.
Post-condition: The output variable $B[1], B[2], \ldots, B[n]$ is a one-dimensional array of real numbers with same elements as $A[1], A[2], \ldots, A[n]$ but with the property that $B[i] \leq B[j]$ whenever $i \leq j$. ■

A proof of algorithm correctness consists of showing that if the pre-condition for the algorithm is true for a collection of values for the input variables and if the statements of the algorithms are executed, then the post-condition is also true.

*R. W. Floyd, "Assigning meanings to programs," *Proc. Symp. Appl. Math.*, Amer. Math. Soc. **19** (1967), 19–32.

†Edsger Dijkstra in O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming* (London: Academic Press, 1972), p. 5.

**David Gries, *The Science of Programming* (New York: Springer-Verlag, 1981), p. 164.

The steps of an algorithm are divided into sections with assertions about the current state of algorithm variables inserted at strategically chosen points:

[*Assertion* 1*: pre-condition for the algorithm*]
{Algorithm statements}
[*Assertion* 2]
{Algorithm statements}
⋮
[*Assertion* $k-1$]
{Algorithm statements}
[*Assertion k: post-condition for the algorithm*]

Successive pairs of assertions are then treated as pre- and post-conditions for the algorithm statements between them. For each $i = 1, 2, \ldots, k-1$, one proves that if Assertion i is true and all the algorithm statements between Assertion i and Assertion $(i+1)$ are executed, then Assertion $(i+1)$ is true. Once all these individual proofs have been completed, one knows that Assertion k is true. And since Assertion 1 is the same as the pre-condition for the algorithm and Assertion k is the same as the post-condition for the algorithm, one concludes that the entire algorithm is correct with respect to its pre- and post-conditions.

Loop Invariants

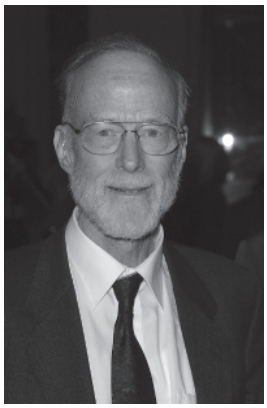
The method of loop invariants is used to prove correctness of a loop with respect to certain pre- and post-conditions. It is based on the principle of mathematical induction. Suppose that an algorithm contains a **while** loop and that entry to this loop is restricted by a condition G, called the **guard**. Suppose also that assertions describing the current states of algorithm variables have been placed immediately preceding and immediately following the loop. The assertion just preceding the loop is called the **pre-condition for the loop** and the one just following is called the **post-condition for the loop**. The annotated loop has the following appearance:

[*Pre-condition for the loop*]
while (G)
    [*Statements in the body of the loop. None contain branching statements that lead outside the loop.*]
end while
[*Post-condition for the loop*]

Definition

A loop is defined as **correct with respect to its pre- and post-conditions** if, and only if, whenever the algorithm variables satisfy the pre-condition for the loop and the loop terminates after a finite number of steps, the algorithm variables satisfy the post-condition for the loop.

Establishing the correctness of a loop uses the concept of loop invariant. A **loop invariant** is a predicate with domain a set of integers, which satisfies the following condition: For



R. Stonehouse/Camera Press/Redux

C. A. R. Hoare (born 1934)

each iteration of the loop, if the predicate is true before the iteration, then it is true after the iteration. Furthermore, if the predicate satisfies the following two additional conditions, the loop will be correct with respect to its pre- and post-conditions:

1. The predicate is true before the first iteration of the loop.
2. If the loop terminates after a finite number of iterations, the truth of the loop invariant ensures the truth of the post-condition for the loop.

The following theorem, called the *loop invariant theorem*, formalizes these ideas. It was first developed by C. A. R. Hoare in 1969.

Theorem 5.5.1 Loop Invariant Theorem

Let a **while** loop with guard G be given, together with pre- and post-conditions that are predicates in the algorithm variables. Also let a predicate $I(n)$, called the **loop invariant**, be given. If the following four properties are true, then the loop is correct with respect to its pre- and post-conditions.

I. Basis Property: The pre-condition for the loop implies that $I(0)$ is true before the first iteration of the loop.

II. Inductive Property: For every integer $k \geq 0$, if the guard G and the loop invariant $I(k)$ are both true before an iteration of the loop, then $I(k+1)$ is true after an iteration of the loop.

III. Eventual Falsity of Guard: After a finite number of iterations of the loop, the guard G becomes false.

IV. Correctness of the Post-Condition: If N is the least number of iterations after which G is false and $I(N)$ is true, then the values of the algorithm variables will be as specified in the post-condition of the loop.

Proof: The loop invariant theorem follows easily from the principle of mathematical induction. Assume that $I(n)$ is a predicate that satisfies properties I–IV of the loop invariant theorem. *[We will prove that the loop is correct with respect to its pre- and post-conditions.]* Properties I and II are the basis and inductive steps needed to prove the truth of the following statement:

For every integer $n \geq 0$, if the **while** loop iterates n times, then $I(n)$ is true. 5.5.1

Thus, by the principle of mathematical induction, since both I and II are true, statement (5.5.1) is also true.

Property III says that the guard G eventually becomes false. At that point the loop will have been iterated some number, say N, of times. Since $I(n)$ is true after the nth iteration for every $n \geq 0$, then $I(n)$ is true after the Nth iteration. That is, after the Nth iteration the guard is false and $I(N)$ is true. But this is the hypothesis of property IV, which is an if-then statement. Since statement IV is true (by assumption) and its hypothesis is true (by the argument just given), it follows (by modus ponens) that its conclusion is also true. That is, the values of all algorithm variables after execution of the loop are as specified in the post-condition for the loop.

Developing a good loop invariant is a tricky process. Although learning how to do it is beyond the scope of this book, it is worth pursuing in a more advanced course.

Another tricky aspect of handling correctness proofs arises from the fact that execution of an algorithm is a dynamic process—it takes place in time. As execution progresses, the values of variables keep changing, yet often their names stay the same. In the following discussion, when we need to make a distinction between the values of a variable just before execution of an algorithm statement and just after execution of the statement, we will attach the subscripts *old* and *new* to the variable name.

Example 5.5.2 **Correctness of a Loop to Compute a Product**

The following loop is designed to compute the product mx for a nonnegative integer m and a real number x, without using a built-in multiplication operation. Prior to the loop, variables i and *product* have been introduced and given initial values $i = 0$ and $product = 0$.

[Pre-condition: m is a nonnegative integer, x is a real number, $i = 0$, and $product = 0$.]

while $(i \neq m)$

1. $product := product + x$
2. $i := i + 1$

end while

[Post-condition: $product = mx$]

Let the loop invariant be

$$I(n)\text{: } i = n \quad \text{and} \quad product = nx$$

The guard condition G of the **while** loop is

$$G\text{: } i \neq m$$

Use the loop invariant theorem to prove that the **while** loop is correct with respect to the given pre- and post-conditions.

Solution

I. Basis Property: *[I(0) is true before the first iteration of the loop.]*

$I(0)$ is "$i = 0$ and $product = 0 \cdot x$," which is true before the first iteration of the loop because $0 \cdot x = 0$.

II. Inductive Property: *[If $G \wedge I(k)$ is true before a loop iteration (where $k \geq 0$), then $I(k + 1)$ is true after the loop iteration.]*

Suppose k is a nonnegative integer such that $G \wedge I(k)$ is true before an iteration of the loop. Then as execution reaches the top of the loop, $i \neq m$, $product = kx$, and $i = k$. Since $i \neq m$, the guard is passed and statement 1 is executed. Before execution of statement 1,

$$\text{product}_{\text{old}} = kx.$$

Thus execution of statement 1 has the following effect:

$$\text{product}_{\text{new}} = \text{product}_{\text{old}} + x = kx + x = (k + 1)x.$$

Similarly, before statement 2 is executed,

$$i_{\text{old}} = k,$$

so after execution of statement 2,

$$i_{\text{new}} = i_{\text{old}} + 1 = k + 1.$$

Hence after the loop iteration, the statement $I(k+1)$, namely, ($i = k+1$ and $product = (k+1)x$), is true. This is what we needed to show.

III. Eventual Falsity of Guard: *[After a finite number of iterations of the loop, G becomes false.]*

The guard G is the condition $i \neq m$, and m is a nonnegative integer. By I and II, it is known that

> for every integer $n \geq 0$, if the loop is iterated n times, then $i = n$ and $product = nx$.

So after m iterations of the loop, $i = m$. Thus G becomes false after m iterations of the loop.

IV. Correctness of the Post-Condition: *[If N is the least number of iterations after which G is false and I(N) is true, then the value of the algorithm variables will be as specified in the post-condition of the loop.]*

According to the post-condition, the value of *product* after execution of the loop should be mx. But if G becomes false after N iterations, $i = m$. And if $I(N)$ is true, $i = N$ and $product = Nx$. Since both conditions (G false and $I(N)$ true) are satisfied, $m = i = N$ and $product = mx$ as required. ■

In the remainder of this section, we present proofs of the correctness of the crucial loops in the division algorithm and the Euclidean algorithm. (These algorithms were given in Section 4.10.)

Correctness of the Division Algorithm

The division algorithm is supposed to take a nonnegative integer a and a positive integer d and compute nonnegative integers q and r such that $a = dq + r$ and $0 \leq r < d$. Initially, the variables r and q are introduced and given the values $r = a$ and $q = 0$. The crucial loop, annotated with pre- and post-conditions, is the following:

[Pre-condition: a is a nonnegative integer and d is a positive integer, r = a, and q = 0.]

while $(r \geq d)$

1. $r := r - d$
2. $q := q + 1$

end while

[Post-condition: q and r are nonnegative integers with the property that $a = qd + r$ and $0 \leq r < d$.]

Proof:

To prove the correctness of the loop, let the loop invariant be

$$I(n)\text{: } r = a - nd \geq 0 \quad \text{and} \quad n = q.$$

The guard of the **while** loop is

$$G\text{: } r \geq d.$$

I. Basis Property: *[$I(0)$ is true before the first iteration of the loop.]*

$I(0)$ is "$r = a - 0 \cdot d \geq 0$ and $q = 0$." But by the pre-condition, $r = a$, $a \geq 0$, and $q = 0$. So since $a = a - 0 \cdot d$, then $r = a - 0 \cdot d$ and $I(0)$ is true before the first iteration of the loop.

II. Inductive Property: *[If $G \wedge I(k)$ is true before an iteration of the loop (where $k \geq 0$), then $I(k+1)$ is true after iteration of the loop.]*

Suppose k is a nonnegative integer such that $G \wedge I(k)$ is true before an iteration of the loop. Since G is true, $r \geq d$ and the loop is entered. Also since $I(k)$ is true, $r = a - kd \geq 0$ and $k = q$. Hence, before execution of statements 1 and 2,

$$r_{\text{old}} \geq d \quad \text{and} \quad r_{\text{old}} = a - kd \quad \text{and} \quad q_{\text{old}} = k.$$

When statements 1 and 2 are executed, then

$$r_{\text{new}} = r_{\text{old}} - d = (a - kd) - d = a - (k+1)d \tag{5.5.2}$$

and

$$q_{\text{new}} = q_{\text{old}} + 1 = k + 1. \tag{5.5.3}$$

In addition, since $r_{\text{old}} \geq d$ before execution of statements 1 and 2, after execution of these statements,

$$r_{\text{new}} = r_{\text{old}} - d \geq d - d \geq 0. \tag{5.5.4}$$

Putting equations (5.5.2), (5.5.3), and (5.5.4) together shows that after iteration of the loop,

$$r_{\text{new}} \geq 0 \quad \text{and} \quad r_{\text{new}} = a - (k+1)d \quad \text{and} \quad q_{\text{new}} = k + 1.$$

Hence $I(k+1)$ is true.

III. Eventual Falsity of the Guard: *[After a finite number of iterations of the loop, G becomes false.]*

The guard G is the condition $r \geq d$. Each iteration of the loop reduces the value of r by d and yet leaves r nonnegative. Thus the values of r form a decreasing sequence of nonnegative integers, and so (by the well-ordering principle) there must be a smallest such r, say r_{min}. Then $r_{\text{min}} < d$. *[If r_{min} were greater than d, the loop would iterate another time, and a new value of r equal to $r_{\text{min}} - d$ would be obtained. But this new value would be smaller than r_{min}, which would contradict the fact that r_{min} is the smallest remainder obtained by repeated iteration of the loop.]* Hence as soon as the value $r = r_{\text{min}}$ is computed, the value of r becomes less than d, and so the guard G is false.

IV. Correctness of the Post-Condition: *[If N is the least number of iterations after which G is false and $I(N)$ is true, then the values of the algorithm variables will be as specified in the post-condition of the loop.]*

Suppose that for some nonnegative integer N, G is false and $I(N)$ is true. Then $r < d$, $r = a - Nd$, $r \geq 0$, and $q = N$. Since $q = N$, substitution gives

$$r = a - qd,$$