

Chapter 5

Increasing Confidence in Correctness

In this chapter we investigate techniques that are used to increase confidence in the likelihood that a program will execute correctly. There are two primary approaches to this task. One is the mechanism of *testing*, where a function or procedure is executed with actual test values. The second is the process of *proving correctness*, where a more-or-less formal proof is developed to demonstrate that an algorithm will perform the desired task. Both techniques are useful, and neither should be considered a substitute for the other.

Techniques used in developing arguments for the correctness of programs include *pre* and *post conditions*, *assertions* and *invariants*. After introducing the tools used in program proving, this chapter concludes with a discussion of program testing techniques and objectives. Major topics include:

- Program proofs
- Assertions
- Invariants
- Program testing

5.1 Program Proofs

Proofs of programs are used by programmers to increase their own confidence in the correctness of their algorithms, and to discover and highlight those places where such confidence is misplaced (that is, find errors). To properly understand the role of program proofs the reader should remember that, in the large, programming is a social activity pursued by a

team of individuals working cooperatively on a given task. A program proof is not for the benefit of the computer; indeed, the greater portion of the proof is documented by comments in the program text, which are not processed by the computer at all. Similarly, a proof is only partially intended for the benefit of the original programmer. Most often, programmers examine each others code in a structured setting, called a *code walk-through*. In such a walk-through, a programmer traces the execution of an algorithm, and presents arguments to justify the contention that the particular section of code under scrutiny will perform as expected. Proofs of program correctness are an essential part of this process.

While the role of the programmer is to argue for the correctness of the program, the role of the other members participating in a code walk-through is to try to anticipate ways in which failure can occur, and highlight potential weakness. Although it is not our place here to discuss in detail principles of software engineering, we note in passing that great care must be taken during code walk-through to ensure that the atmosphere remains one of cooperation, and that programmers are not permitted to have the feeling that they are defending their code against “attacks” by other programmers.

5.1.1 Invariants

The major tool employed in formulating an argument that can be used to justify belief in the correctness of an algorithm is the *invariant*. Like a *pre condition*, an invariant is nothing more than a comment, but it is a comment that describes the state of the computation at the point the comment would be encountered during the course of execution. The art of using invariants lies in finding statements that are most meaningful to a human audience. To do so, an invariant should describe the processing being performed in the high-level language of the problem domain, and not necessarily in the low-level language of the actual calculation being performed.

Here is a program that contains an error. Let us see how the use of invariants could be used to uncover the bug. The program purports to characterize three sides of a triangle, returning 1 if the triangle is equilateral (all sides equal), 2 if isosceles (two sides equal), and 3 if scalene (no sides equal).

```
int triangle (int a, int b, int c) {
    // pre: input a, b, and c are sides of a triangle
    // post: returns a characterization of the triangle
    // 1 if equilateral, 2 if isosceles, and 3 is scalene
    if (a == b) {
        if (b == c) {
            // inv: a equals b, b equals c,
            // so a equals c and all are equal
            return 1; // equilateral
        }
        else {
            // inv: a equals b, but b does not equal c
```

```

        // so only two sides are equal
        return 2; // isosceles
    }
}
else {
    if (b == c) {
        // inv: a not equal to b, but b equals c,
        // so two sides are equal
        return 2; // isosceles
    }
    else {
        // inv: so no sides are equal
        return 3; // scalene
    }
}
}

```

To form an argument to justify belief in the correctness of an algorithm, we first assume the pre conditions (if any) are valid. We then trace execution from beginning to end, following all possible paths, and demonstrate that the invariants must be true when execution reaches the point where they are placed. Finally, we argue that the invariant establishes the correct result.

If we imagine a test case that fails the first if statement and also the second, we can see the error in the program. At the point of the invariant immediately prior to returning the value 3, the information we have is that *a* is not equal to *b*, and that *b* is not equal to *c* (since both these conditions have failed). But we do not therefore know that all sides are unequal, since it is possible for *a* to be equal to *c*. Thus, the invariant that we have written (which, if it were true, would justify the subsequent statement) cannot be supported by an argument. (This illustrates that invariants should not be assumed to be true until proven by argument.)

Invariants become more useful, but also more complicated, when programs contain loops. In a program with loops we cannot simply execute all possible paths. However, the concept of an invariant remains the same. That is, an invariant simply describes the state of execution at the point the invariant appears. In Chapter 4 we described an algorithm that could be used to find the minimum value from a vector of *n* floating-point quantities. This function could be augmented with invariants in the following fashion:

```

public double minimum (double [ ] values) {
    // pre: values has at least one element
    // post: returns smallest value in collection
    int n = values.length;
    // return the minimum value found
    // in the vector of double precision values
    double minValue = values [0];
}

```

```

    // inv 1: minValue is the minimum value found in
    // the range 0 .. 0
    for (int i = 1; i < n; i++) {
        // inv 2: minValue is the minimum value found in
        // the range 0 .. i-1
        if (values[i] < minValue)
            minValue = values[i];
        // inv 3: minValue is the minimum value found in
        // the range 0 .. i
    }
    // inv 4: minValue is the minimum value found in
    // the range 0 .. n-1
    return minValue;
}

```

In the next section we will describe how invariants are used in formulating an argument to increase confidence in the correct functioning of an algorithm.

5.1.2 Analyzing Loops

To create an argument that can be used to increase confidence in an algorithm, the programmer traces all possible execution paths through a program that lead from one invariant to the next. In each case, the programmer creates an argument which asserts that *if* the first invariant is true, and *if* the intervening statements are executed, *then* the second invariant must be true. We will illustrate this process with the function given in the previous section.

The first step is to verify the initial invariant. We assume the precondition has been satisfied, and therefore the assignment statement is used to initialize the variable `minValue` to the first element will succeed without raising an indexing exception. The variable, therefore, holds the minimum value of the vector in the range 0 to 0 (indeed, it holds the only value in this range).

```

...
double minValue = values [0];
    // inv 1: minValue is the minimum value found in
    // the range 0 .. 0
    for (int i = 1; i < n; i++) {
        // inv 2: minValue is the minimum value found in
        // the range 0 .. i-1
    }
...

```

We next argue from invariant 1 to invariant 2. Just as we did with induction, we are now making the *assumption* that invariant 1 is true, and arguing that if this is the case, then invariant 2 must be true when execution reaches the point at which it has been placed. Between these two points, the only value that is changed is the variable `i`, which is assigned the value 1, and has been tested to ensure it is less than `n`. But the quantity `i - 1` is therefore

Loop Invariants and Mathematical Induction

The validity of the use of loop invariants can be demonstrated by turning a proof based on invariants into a proof using mathematical induction. To do so, we simply let the *number of times execution has passed through the loop* be the induction value.

Consider how invariants are used in the analysis of the minimum value function considered in Section 5.1.2. Cases in which the loop is executed zero times and in which the loop is executed one time are handled as base cases. For the induction case, we assume the loop has been executed n times, and will loop at least $n + 1$ times (that is, at least one more time). The argument presented in the text then demonstrates that if invariant 3 is true during the n th iteration, then invariant 2 must be true on the $n + 1$ iteration.

This shows that no matter how many times the loop iterates, invariant 3 will always be true when it is encountered. A separate argument then traces the flow from invariant 3 to invariant 4, and we are finished.

0, and if invariant 1 were previously true, then invariant 2 must now be true, because it is asserting the same fact.

```

...
    // inv 2: minValue is the minimum value found in
    // the range 0 .. i-1
    if (values[i] < minValue)
        minValue = values[i];
    // inv 3: minValue is the minimum value found in
    // the range 0 .. i
...

```

Next, we consider the flow of execution between invariant 2 and invariant 3. This time, instead of assuming we are on the first iteration (as we did between invariant 1 and invariant 2) we now assume we are on some indefinite iteration. We therefore have no specific information concerning the value i , other than it must have a value between 1 and n (the limits of the loop). Invariant 2 tells us that the variable `minValue` holds the minimum value in the vector for indices in the range 0 to $i - 1$. If we examine the code between invariants 2 and 3, we see that we can divide our argument into two cases. Either the value held by the vector at index position i is smaller than this minimum, or it is not. In the former case the variable `minValue` is updated to hold this new value. Therefore, in either case, by the time we reach invariant 3 we can argue that the value held by `minValue` is now the smallest element in the range of data values indexed by 0 through the position i .

Note that invariant 2 and invariant 3 are very similar. We often say that invariant 2 has been *strengthened* in order to reach invariant 3. In this case, strengthening means that we

have extended the range of values under consideration from $(0 .. i - 1)$ to $(0 .. i)$.

From invariant 3 there are two possible directions that execution could flow. Both must be investigated. In both cases, the variable i will be incremented. The new value of i then either is or is not greater than or equal to the limit value n . If it is not, then the loop will iterate once more, and invariant 2 will once again be reached. In this circumstance we are assuming that invariant 3 is true, and arguing that invariant 2 must now be true. But invariant 3 is asserting the same condition as invariant 2, under the condition that variable i has been incremented.

The second possibility is that the loop terminates, and execution flows from invariant 3 to the final invariant 4. But this will only occur if the updated value of i (which is monotonically increasing) is now n . Therefore invariant 3 is transformed into invariant 4 with the substitution of $n - 1$ for i . (We must subtract one to “undo” the fact that i was incremented prior to the test.)

```

...
    // inv 1: minValue is the minimum value found in
    // the range 0 .. 0

for (int i = 1; i < n; i++) {
    ...
    // inv 3: minValue is the minimum value found in
    // the range 0 .. i
}

// inv 4: minValue is the minimum value found in
// the range 0 .. n-1

```

The last flow of control is the easiest to forget. It is possible to move directly from invariant 1 to invariant 4 without encountering any other invariants. This will occur only when the size of the vector is exactly one, and therefore the condition controlling the loop is false when it is first encountered. But in this circumstance the value n must be 1, and therefore invariant 4 is asserting the same condition as invariant 1.

5.1.3 Asserting the Outcome is Correct

The final invariant in any procedure should always assert the conditions corresponding to the desired outcome, that is the result specified by the post conditions. These may only be indirectly addressed by the actions of the program. For example, the invariants in the `isPrime` procedure deal with finding factors of the input value. Only if no factors are found is the number known to be prime. Note that, as in the minimum number procedure, the invariant at the end of the loop is a simple conjunction of the invariant at the start of the loop and the negation of the looping condition.

```

public boolean isPrime (int n) {
    // pre: n is greater than or equal to 2

```

```

    // post: true if n is prime, false otherwise
for (int i = 2; i * i <= n; i++) {
    // inv 1: n has no factors between 2 and i-1

    if (0 == n % i) {
        // inv 2: i divides n
        // therefore n is not prime
        return false;
    }

    // inv 3: n has no factors between 2 and i
}

// inv 4: n has no factors between 2 and
// ceiling(sqrt(n)), therefore number must be prime
return true;
}

```

5.1.4 Progress Toward an Objective

Invariants inside a loop describe an intermediate stage within a calculation. In such situations, only partial progress has been made toward an ultimate objective. To describe such invariants, it is necessary to understand not only the ultimate objective, but also how the intermediate stage relates to the intended outcome. (When viewed in this light, a loop is therefore one more example of the abstraction device we back in Chapter 1 termed *repetition*.)

For example, in the binary search procedure we initially know only that the target value, if it appears at all, appears in positions indexed between 0 and $n - 1$ (that is, the entire vector). By repeatedly examining the middle value in a range, we divide in half the region in which the value can potentially be found. This continues until the region is reduced to a single element.

```

int binarySearch (double [ ] data, double testValue) {
    // pre: elements in argument data are ordered smallest to largest
    // post: returns the index in data where testValue is found,
    // or index of next larger element if not in collection
    int low = 0;
    int high = data.length;

    while (low < high) {
        // inv: data[0 .. low-1] less than testValue
        // data[high .. max] greater than or equal to testValue
        mid = (low + high) / 2;
    }
}

```

```

    if (data[mid] < testValue)
        low = mid + 1;
    else
        high = mid;
}
// inv: data[0..low-1] less than testValue
// and testValue less than or equal to data[low+1]
return low;
}

```

When loops are nested, each loop may be pursuing its own objective, and each may therefore have a different set of invariants. In the bubble sort procedure, for example, the outer loop is placing elements into position, starting from the top. The inner loop is designed to ensure that the largest remaining value is moved into the topmost position. This is accomplished by moving the loop variable j upward through all possibilities, and maintaining the condition that the largest value seen so far is kept at position $j+1$. We could augment this procedure with invariants as follows:

```

void bubbleSort (double [ ] v) {
    // exchange the values in the vector v
    // so they appear in ascending order
    int n = v.length;

    for (int i = n - 1; i > 0; i--) {
        // inv: elements indexed i+1 to n-1 are correctly ordered

        for (int j = 0; j < i; j++) {
            // inv: v[j] holds largest value from range (0..j)

            if (v[j+1] < v[j]) { // if out of order, then swap
                double temp = v[j];
                v[j] = v[j + 1];
                v[j + 1] = temp;
            }

            // inv: v[j+1] holds largest value from range (0..j+1)
        }

        // inv: v[i] holds largest value from range (0..i)
        // inv: therefore, elements i to n-1 are correctly ordered
    }

    // inv: elements indexed 0 to n-1 are correctly ordered
}

```

5.1.5 Manipulating Unnamed Quantities

When using invariants it is often necessary to discuss quantities that are not explicitly named in the program. An example occurs in the greatest common divisor program from Chapter 3. To argue that this procedure is correct we require the observation that if d is a divisor for both n and m , and n is larger than m , then d must also be a divisor for $n - m$. (To see this, note that $\frac{n}{d}$ must be integer, as must $\frac{m}{d}$, and therefore $\frac{n}{d} - \frac{m}{d}$ must be integer, but this is the same as $\frac{n-m}{d}$). The invariants for the loop must therefore assert that while we have changed n and m , the GCD of the original values, which is not a quantity that has been given a name by the program, has not been altered.¹

```
public int gcd (int n, int m) {
    // pre: assume arguments are greater than or equal to zero
    // post: compute the greatest common divisor of arguments

    while (m != n) {
        if (n > m)
            n = n - m;
            // inv: gcd of n and m
            // has not been altered
        else
            m = m - n;
            // inv: gcd of n and m
            // has not been altered
        }

        // n equal to m,
        // so n is divisor of both
    return n;
}
```

5.1.6 Function Calls

When a procedure invokes another procedure, the argument concerning the correctness of the first procedure is always expressed conditionally, based on the assumption that the invoked procedure is performing in the correct manner. Although it is not necessary to repeat the argument for the correctness of the called procedure, it *is* necessary to address the question of whether the arguments that are being passed to the underlying procedure are valid; that is, within the range of elements the procedure is prepared to handle.

¹To be precise, all we have demonstrated by this argument is that the result will be a divisor of the original n and m values. To assert that the result is the largest possible divisor requires a more subtle mathematical argument, which is not relevant to our discussion here.

For example, the `printPrimes` procedures can be shown to be correct, based on the assumption that the `isPrime` procedure operates correctly. The pre condition to `isPrime` requires the index value to be larger than 2. It is easy to verify that this condition is satisfied, and so we assume that `isPrime` is producing the correct outcome. But with this assumption it is then easy to argue that `printPrimes` must be correct.

```
void printPrimes (int n) {
    // print numbers between 2 and n
    // indicating which are prime
    for (int i = 2; i <= n; i++) {
        if (isPrime(i))
            System.out.println("value " + i + " is prime");
        else
            System.out.println("value " + i + " is not prime");
    }
}
```

5.1.7 Recursive Algorithms

The analysis of recursive algorithms is simply an extension of the technique used for ordinary procedure calls. As we noted in Chapter 4, the execution of recursive procedures is always divided into two cases. One or more *base cases* can be handled without recursive invocation. The *recursive* (or *inductive*) cases are handled by the procedure calling itself.

The base cases are analyzed as any other procedure, using invariants if necessary. To handle the recursive case, we simply *assume* that the recursive invocation will perform correctly, and argue the remainder of the procedure accordingly. For example, consider the procedure from the previous chapter to print a positive integer value:

```
public void printInt (OutputStream out, int val) {
    // print the character representation of the argument value
    if (val < 10)
        out.write(digitChar(val));
    else {
        printInt(val/10); // print all but final character
        out.write(digitChar(val%10)); // print last character
    }
}
```

If the parameter value is less than ten, it is easy to see that the procedure performs as expected (assuming that the two procedures `digitChar` and `write` operate properly). The recursive case is hardly any more complex; we assume the recursive call will work, and the remainder is simply another set of calls on `write` and `digitChar`.

Once again, to place this technique on firm formal footing requires relating the analysis to mathematical induction. One argues that in situations where no recursive calls are invoked the correct results are produced. One then assumes that the correct result is produced in

all situations where n recursive calls are necessary. Based on this, one then formulates an argument that if $n + 1$ recursive calls are necessary the correct result will be produced, because this reduces to a case where n calls are necessary.

In both proving mathematical formulas using induction and in the analysis of recursive algorithms, the following steps must be performed:

1. Identify the base cases, and establish that the formula or algorithm works for these cases.
2. Argue that, in all situations, the reduction being performed must eventually reach one of the base cases.
3. Provide a conditional argument which asserts that *if* the recursive call produces the correct outcome, *then* the remainder of the program must produce the correct result.

The recursive version of the integer exponential algorithm illustrates that invariants are related more to the problem domain than to the mechanics of execution. In this case, the only invariant necessary is used to assert that although the recursive calculation being performed is different, the resulting value is the same as the original request.

```
public double power (double base, int n) {
    // pre: n is larger than or equal to zero
    // post: return the value yielded by raising the
    // double precision base to the integer exponent
    // assumes floating point overflow does not occur
    if (n == 0) // anything raised to zero power is 1
        return 1.0;
    else if (even(n))
        // base ^ n is same as
        // (base ^ 2) ^ (n / 2) for even n
        return power(base * base, n / 2);
    else
        // base ^ n is same as
        // base * (base ^ 2) ^ (n / 2) for odd n
        return power(base * base, n / 2) * base;
}
```

The Towers of Hanoi puzzle gives us an even more dramatic example of how the analysis of recursive algorithms is performed by conditionally assuming the correct functioning of the recursive case. This algorithm could be augmented with invariants as follows:

```
void solveHanoi (int n, char a, char b, char c) {
    // move n disks from tower a to tower b, using tower c

    if (n == 1) {
        System.out.println("move disk from tower " + a + " to " + b);
    }
}
```

```

        // inv: have moved stack of size 1
        // from tower a to tower b
    }
else {
    solveHanoi(n-1, a, c, b);
    // inv: have moved stack of size n-1 from
    // stack a to stack c

    System.out.println("move disk from tower " + a + " to " + b);
    // inv: move moved largest disk from stack a
    // to stack b

    solveHanoi(n-1, c, b, a);
    // inv: have moved stack of size n-1 from
    // stack c back to stack b, on top of
    // disk previously moved,
    // therefore now have stack of size n
    // on stack b
}
}

```

We *assume* the recursive call will perform the task we describe. Based on that assumption, we then prove that the current task is correctly executed. The base cases are handled by a separate argument. If all of these arguments are correct, then the entire algorithm must be correct, regardless of the input values.

5.2 Program Testing

Testing is the process of executing computer code on actual values, and verifying the resulting output for correctness. As a side effect, programming errors are often uncovered when the expected output is not produced. Testing should always be an intrinsic part of any software development effort.

Testing can, and should, be performed at all levels:

- Individual functions or methods can be tested as they are written.
- A class can be tested in isolation from the remainder of a program.
- Finally, a complete program can be tested as an application.

In the first two cases we are testing small portions of a program, independent of the remainder of the application. To do this it is often necessary to write short-term, temporary

Testing Alone Can Never Show Correctness

It has been said that testing can show the presence of errors, but never their absence. What this means is that a test case that produces an incorrect result clearly shows the the program is wrong, but a single test case (or even many test cases) that is correctly handled does not show the program is right. Testing should never be used as a substitute for a proof of correctness. (A programming project described at the end of this chapter has you explore this idea, by investigating exactly how many random test cases you would need to generate before you might uncover the error in the triangle program).

Neither testing nor a proof of correctness should be taken to be definitive evidence that a program is valid. Program proofs can be misleading if preconditions are not satisfied, or if the post conditions do not capture all the necessary properties required by the rest of a program. These are features that are often caught more easily by testing. So both techniques are useful mechanisms for increasing confidence in the validity of a program.

code to “harness” the software being tested. This harness code can be divided into two categories:

- *Driver* code acts as the calling procedure. This code sets up the argument values, global variables, or whatever input is necessary for the code under test; then it invokes the procedure, and finally validates the result (or prints the result, leaving it to the programmer to perform the validation).
- *Stub* code simulates the actions of any procedures that may be called by the algorithm under test. While stub code simulates the actions that will be found in the resulting application, they need not perform exactly the same process. For example, a stub might merely print the argument values, then prompt the programmer to supply the result that will be returned.

Functions should always be tested using a number of different input values. The following guidelines can be considered in creating good test values:

- Make sure every statement in the function is exercised by at least one test value.
- Make sure to use test data that exercises both the true and false alternatives for every if and while expression. Test each legal label of a switch statement.
- If there is a minimal legal input value, such as an empty array or a smallest integer value, use this as one of your test cases.

- If the function (or program) has both legal and illegal inputs, a set of test cases should include both clearly legal values and clearly illegal values, as well as values that are “barely” legal and “barely” not legal.
- If the program involves loops that can exercise a variable number of iterations, try to develop a test case in which the loop executes zero times.

Several more specific guidelines have also been suggested. The readings cited at the end of the chapter can be examined for further details.

5.3 Chapter Summary

An invariant is a statement that describes the state of computation when execution reaches a particular point in a program. By tracing the flow of execution from invariant to invariant, invariants can be used to structure arguments used to increase confidence in the validity of an algorithm or function.

Even when a proof of correctness has been presented, software testing should be employed as an alternative technique to increase the confidence in the correct performance of a function or application.

Key Concepts

- Program proofs
- Invariants
- Using invariants in tracing execution flow
- Relationship between recursion and mathematical induction
- Software testing

Further Reading

The argument that programming (and, indeed, mathematics) is a social activity was given forceful exposition in a classic paper by Richard DeMillo, Richard Lipton and the late Alan Perlis [DeMillo 79].

The notion of program invariants was developed by Robert Floyd. For this contribution (among others) he was presented with the 1979 Alan Turing award by the Association for Computing Machinery, a major computer science professional organization.

Many researchers have tried to raise the level of program proofs to a formal mathematical process. Such proofs tend to be many times more complex than the arguments we develop in this book. A good introduction to program proving can be found in the

book by David Gries [Gries 81]. Another prominent advocate of program proving is Edsger Dijkstra [Dijkstra 76].

A good description of the art of software testing is [Beizer 90].

Study Questions

1. What are the two primary mechanisms for increasing confidence in the correctness of programs?
2. For whom is a program proof created? That is, who will read a program proof?
3. What is a code walk-through?
4. What is an invariant?
5. Explain the steps used in proving a program correct using invariants.
6. Assume an algorithm consists of a single loop, such as the minimum algorithm presented in Section 5.1.1. Assume that within the loop there are invariants at both the beginning and end of the loop body, as well as invariants prior to and after the loop. How many different arguments must be given that move from invariant to invariant?
7. What is the objective toward which “progress” is being made in the binary search algorithm?
8. What is the unnamed quantity that is referred to by the invariants for the GCD algorithm?
9. What are the steps involved in proving the correctness of a recursive algorithm?
10. What is the unnamed quantity that is referred to by the invariants for the power procedure described in Section 5.1.7?
11. At what point in the development process should testing be performed?
12. What is a testing harness? A driver? A stub?
13. What are some basic guidelines for developing good test cases?

Exercises

1. Complete the task of arguing the correctness of the `isPrime` procedure described in Section 5.1.3. That is, give arguments to move from each invariant to each possible succeeding invariant.

2. Complete the task of arguing the correctness of the `binarySearch` procedure described in Section 5.1.4. That is, give arguments to move from each invariant to each possible succeeding invariant.

3. Provide invariants for the linear time `power` procedure from Chapter 4.

```
double power (double base, int n) {
    double result = 1.0;
    // inv 1:

    for (int i = 1; i <= n; i++) {
        // inv 2:
        result *= base;
        // inv 3:
    }

    // inv 4:
    return result;
}
```

4. Using the invariants you developed in the previous question, give arguments that support the validity of the invariants by tracing possible execution flows.

5. Complete the loop invariants for the following procedure that sums the values of an array.

```
//
//   sum the elements of a double array
//

double sumArray (double [ ] data) {
    int size = data.length;
    double sum = 0.0;
    // inv:
    for (int i = 0; i < size; i++) {
        // inv:
        sum += data[i];
        // inv:
    }
    // inv:
    return sum;
}
```

6. Using the invariants you developed in the previous question, give arguments that support the validity of the invariants by tracing possible execution flows.

7. Complete the loop invariants for the following procedure that computes the factorial function for a positive integer argument.

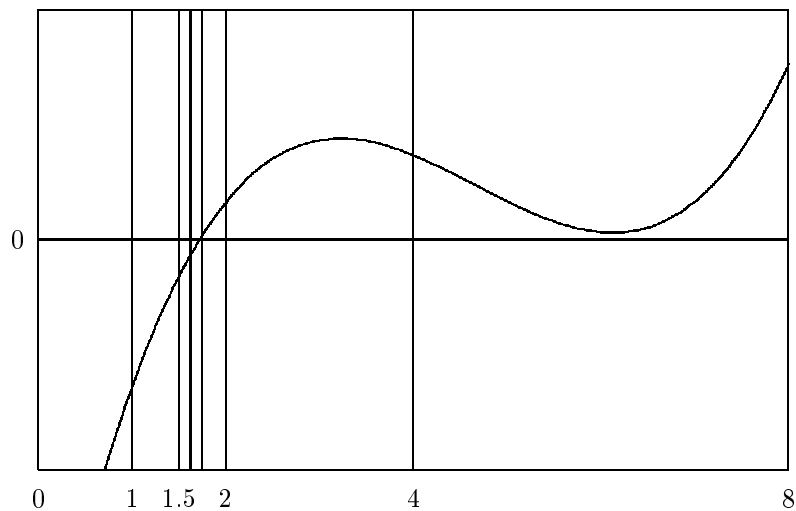
```
//  
//   compute the factorial of an integer value  
//  
  
double factorial (int val)  
{  
    double result = 1.0;  
    // inv:  
    for (int i = 1; i <= val; i++) {  
        // inv:  
        result *= i;  
        // inv:  
    }  
    // inv:  
    return result;  
}
```

8. Using the invariants you developed in the previous question, give arguments that support the validity of the invariants by tracing possible execution flows.
9. Provide a set of invariants for the insertion sort algorithm described in Section 4.2.4. Using these invariants as a base, give arguments that support the validity of the invariants by tracing possible execution flows.
10. What are some test cases one might use to exercise the function `minimum` described in Section 5.1.1?
11. What are some test cases one might use to exercise the function `isPrime` described in Section 5.1.3?
12. What are some test cases one might use to exercise the function `binarySearch` described in Section 5.1.4?
13. Augment the main program in the application developed in Chapter 9 with invariants, and with these provide an informal proof of correctness.

Programming Projects

1. Write a program that will generate three random integer values, and compare the output produced by the incorrect version of the `triangle` to that of the corrected program. How many random elements must you examine before you find a set of inputs that exposes the error in the algorithm?

- Using the `TaskTimer` utility described in Chapter 4, compare the execution time of the looping version and the recursive version of the integer power algorithm. Do this by raising the value 1.02 to ever larger integer exponents, and plotting the execution times.
- The key idea of binary search, repeatedly dividing a search area in half, is applicable to more than simply searching an ordered list of values. For example, we can use binary search to quickly discover the approximate value of a root for a continuous function. (A root is a point where the function has value zero, a continuous function is one that has no “gaps”, that can be drawn without lifting a pen from the paper). The idea is to start with two values which we know surround the root, because the function is positive at one point and negative at the other. Repeatedly divide the area in half, and move either the lower or the upper bound. Repeat until the difference between the lower and upper bounds is very small, at which point you have the approximate location of the root. The following shows this process applied to the function $x^2 - 14x^2 + 59x - 65$, using the initial values 0 and 8. The numbers represent the values of the midpoints. Implement a root finding program based on these ideas.



- Another application of the binary search technique is an algorithm for finding the square root of a positive value. For a given value n , we bound the square root by two values `low` and `high`. We can initially set `low` to zero and `high` to n . We repeatedly compute the midpoint between `low` and `high`, and change either the `low` or the `high` value depending upon whether the midpoint squared is greater than or less than the

value n (that is, whether the midpoint is greater than or less than the real square root). The process halts when the difference between the low and the high values becomes sufficiently small (for example, less than 0.0001).

Develop an algorithm for computing the square root using this process, and provide a proof of correctness. What type of inputs would you use to test your algorithm?