Chapter 3: Debugging, Testing and Proving Correctness

In this chapter we investigate tools that will help you to produce reliable and correct programs. During development of any program you will undoubtedly need to remove errors, and this will involve *debugging*. Once you believe your program (or portions of it) is correct you will want to increase your confidence in the program by systematic testing. Typically testing will uncover errors, which will lead to further debugging. Finally, the most powerful tool you can use to increase your confidence in a program or function is a proof of correctness. All of these tools are useful, and none should be considered to be a substitute for the others.

Hints on Debugging

There is no question that programming is a difficult task. Few nontrivial programs can be expected to run correctly the first time without error. Fortunately, there are many hints that can be used to help make debugging easier. Here are some of the more useful suggestions:

- Test small sections of a program in isolation. When you can identify a section of a program that is doing a specific task (this could be a loop or a function), write a small bit of code that tests this functionality. Gain confidence in the small pieces before considering the larger whole.

- When you see an error produced for a given input, try to find the simplest input that consistently reproduces the same error. Errors that cannot be reproduced are very difficult to eliminate, and simple inputs are much easier to reason about than more complex inputs.

- Once you have a simple test input that you know is handled incorrectly, play the role of the computer in your mind, and simulate execution of this test input. This will frequently lead you the location of your logical error.

- Think about what occurs before the point the error is noticed. An incorrect result is simply the symptom, and you must look earlier to find the cause.

- Use breakpoints or print statements to view the state of the computation in the middle. Starting with an input that produces the wrong result, try to reason backwards and determine what the values of variables would need to be to produce the output you see. Then check the state using break points or print statements. This can help isolate the portion of the program that contains the error.

- Don’t assume that just because one input is handled correctly that your program is correct.

- Use assertions and invariants to reason logically about your program.

- Most importantly, make sure you have the right mindset. Don't naturally assume that just because one section of a program works for most inputs, it must be correct. Question everything. Be open to any possibility. Look everywhere.

Assertions and Invariants

When analyzing algorithms two questions are important: Is the algorithm correct, and how fast does it run. In this chapter we are describing techniques that can be used to address the first. The second will be the subject of the next chapter.

One of the most powerful tools used to analyze algorithms is an *assertion*. An assertion is simply a comment that explains what you know to be true when execution reaches a specific point in the program. Assertions can include information you know from a specific statement, as well as information you know from tracing a flow of control.

```
int triangle(int a, int b, int c) {
  if (a == b)
    if (b == c)  return 1;
    else
      return 2;
  else if (b == c)
    return 2;
  else
    return 3;
}
```

For example, suppose you have written the function shown at the left. The program takes three integer values representing the sides of a triangle, and is intended to return a value that is 1 if the triangle is equilateral, 2 if it is isosceles (two sides equal in length), and 3 if it is scalene (no sides are equal). How would you increase your confidence that you have written the correct function?

```
int triangle(int a, int b, int c) {
  if (a == b)      /* we know a == b */
    if (b == c)  /* we know a==b & b==c */
      return 1;
    else /* we know a==b and b != c */
      return 2;
  else if (b == c) /* we know a!=b and b== c*/
    return 2;
  else /* we know a != b and b != c */
    return 3; /* ERROR! */
}
```

Assertions keep track of the information you know from following a path through the program. For example, after the first if statement we know that a is the same as b. After the second statement we know that b is the same as c, but we also remember the information from the first. Since a is equal to b and b is equal to c, it must be the case that all three are equal. Along the else path, however, we know that the original condition must be false, and so we can carry along that information. In this program keeping track of this information leads to the discovery of an error. After the last else statement we know that a is not equal to b and that b is not equal to c. But this does not imply, as the program is claiming, that all three values are different. It could still be the case that a is equal to c. The detection of this error has been simplified by explicitly keeping track of information using assertions.

Assertions become most useful when they are combined with loops. An assertion inside a loop is often termed an *invariant*, since it must describe a condition that does not vary during the course of executing the loop. To discover an invariant simply ask yourself why you think a program loop is doing the right thing, and then try to express "right thing" as a statement. For example, the function at right is computing the sum of an array of values. It does this by computing a partial sum, a sum up to a given point. So assertion number 3 is the easiest to discover. Once you discover assertion 3, then assertion 2 becomes clear – it is whatever is needed to ensure that assertion 3 will be true after the assignment. Assertion 4 is stating the expected result, while assertion 1 is asserting what is true before the loop begins.

```
double sum (double data[ ], int n) {
  double s = 0.0;
  /* 1. s is the sum of an empty array */
  for (int i = 0; i < n; i++) {
    /* 2. s is the sum of values from 0 to i-1 */
    s = s + data[i];
    /* 3. s is the sum of values from 0 to i */
  }
  /* 4. s is the sum from 0 to n-1 */
  return s;
}
```

Later in this chapter you will learn how to use invariants and assertions to prove that an algorithm or program is correct.

```
void bubbleSort (double data [ ], int n) {
  for (int i = n-1; i > 0; i--) {
    for (int j = 0; j < i; j++) {
      // data[j] is largest value in 0 .. j
      if (data[j] > data[j+1])
        swap(data, j, j+1)
      // data[j+1] is largest value in 0 .. j+1
    }
    data[i] is largest value in 0 .. i
  }
  // array is sorted
}
```

Notice how assertions require you to understand a high level description of what the algorithm is trying to do, and not simply a low level understanding of what the individual statements are doing. For example, consider the bubble sort algorithm shown at left. Bubble sort has two nested loops. The outer loop is the position of the array being filled. The inner loop is "bubbling" the largest element into this position. So once again at the end of the inner loop you want to make an assertion not only about the particular values at index locations j and j+1, but about everything the loop has seen before (namely, the elements with index values less than j). Once you identify this assertion, then the assertion at the beginning of the loop must be whatever is needed to prove the assertion at the end of the loop, and together these must be whatever is necessary to prove the assertion at the end of the outer loop.

Bubble Sort and Sorting

Bubble sort is the first of many sorting algorithms we will encounter in this book. Bubble sort is examined not because it is useful, it is not, but because it is very simple to analyze and understand. But there are many other sorting algorithms that are far more efficient. You should never use bubble sort for any real application, since there are many better alternatives.

In Worksheet 4 you will practice writing invariants and assertions that could be used to prove the correctness of a variety of programs.

Assertions and the assertion statement

Most programming languages include a statement, often termed the *assertion statement*, that performs a task that is similar but not exactly the same as the concept of the assertion described above. The assertions described here are written as comments, and are not executed by the program during execution. They need not be written in an executable form. The assertion statement, on the other hand, takes as argument an expression, and typically will halt execution and print an error message if the statement is not true. This can be very useful for verifying that input values satisfy whatever conditions are necessary for execution. Our square root example from the previous chapter, for example, could use an assertion statement to check that the input is a positive number:

```
double sqrt (double val) {
  assert (val >= 0); /* halt execution if value is not legal */
```

Because assertion statements are executed at run time, and halt execution if they are not satisfied, they should be used sparingly, but can be useful during debugging. The wikipedia entry for "assertions" contains a good discussion of assertions as used for program proofs compared with assertions used for routine error checking.

Introduction to the Binary Search Algorithm

An important algorithm that we will see many times in many different forms is the *binary search* algorithm. Binary search is similar to the way you guess a number if somebody says "I'm thinking of a value between 0 and 100. Can you find my number?" If you have played this game, you know the optimal strategy is to guess the value in the middle: "Is it larger or smaller than 50?" Suppose the other person answers "smaller". Then you again divide the range: "is it larger or smaller than 25?" By repeatedly apply this technique you very quickly find the hidden value. The binary search algorithm works in a similar fashion, but instead of numbers it looks for a specific value in an array of sorted numbers. Like the guessing game, it starts in the middle of the array, in one question eliminating one half of the possibilities, then in the next question breaking that subsection in half, and so on.

One version of the binary search algorithm is shown at right. Here n represents the number of values in the sorted data array, and the variable test is the value being searched for. You can verify that the algorithm is correct using the following invariant:

```
int binarySearch(double data[ ], int n, double test) {
  /* data is size n sorted array */
  int low = 0;
  int high = n;
  while (low < high) {
    mid = (low + high) / 2;
    if (data[mid] == test) return 1; /* true */
    if (data[mid] < testValue)
      low = mid + 1;
    else
      high = mid;
  }
  return 0; /* false */
}
```

Binary Search Invariant: All values with index positions smaller than low are less than test, and all values with index

positions larger than or equal to high are larger than or equal to test.

Prove to your satisfaction that the invariant is true at the beginning of the program (immediately after the assignments to low and high), and that it remains true at the end of the loop. Note that initially the variable high is not a legal index, and so the set of values with index positions larger than or equal to high is an empty set. In exercises at the end of the chapter we will use these to prove the algorithm is correct.

Testing and Boundary Cases

After you have translated an algorithm into a function or program, using assertions and invariants to increase your confidence in the correctness of your code, you should then always use *testing* to further assure yourself that your program works correctly. Testing is performed at many levels. You can (and should) test individual functions before you have a working application. This is termed *unit testing*. To perform a test, you need a main method. This is different from the main method you will eventually use for your program. A special purpose main method used in testing is termed a *test harness*. The test harness will feed one or more values into the function under test, and check the result. The box shows a test harness for the summation program given earlier in this chapter.

```
int main ( ) {
   double dataSetOne[ ] = {1.0, 2.0, 3.0, 4.0, 5.0};
   double sm = sum(dataSetOne, 5);
   println("result 1 should be 15, is: %g ", sm);
   return 0;
}
```

You should never content yourself with just one test case. A single test case cannot exercise all the possible ways a function can be used. As you develop test cases, think about the input data. If there are limits to the data, try to exercise values that are just at the edge of the limits. If the program uses conditional statements, use some data values that evaluate the condition true, and others that evaluate the condition false. This process is termed *boundary testing*. For example, think about testing the method min given earlier. Will the method find the correct value if the minimum is the first element in the array? If it is the last? If it is in the middle? If all values are the same? What about if there is only one element? What if there are no elements? Create a test case for each condition, and verify the result is correct. A collection of test cases is termed a *test suite*.

```
int main ( ) {
   double test1 [ ] = {1.0, 2.0, 3.0}; /* smallest first */
   double test2 [ ] = {3.0, 2.0, 1.0}; /* smallest last */
   double test3 [ ] = {2.0, 1.0, 3.0}; /* smallest middle */
   double test4 [ ] = {3.0, 1.0, 1.0, 2.0}; /* repeated smallest */
   double test5 [ ] = { }; /* no elements */
   double t1, t2, t3, t4, t5;
   t1 = min(test1, 3);
   t2 = min(test2, 3);
   t3 = min(test3, 3);
   t4 = min(test4, 4);
   printf("test cases 1, 2, 3, and 4: %g %g %g %g \n", t1, t2, t3, t4);
   t5 = min(test5, 0); /* should generate assertion error */
```

```
  printf("test case 5: %g \n", t5);
  return 0;
}
```

Notice that we expected one of these data sets to halt execution with an assertion error. After verifying that this is correct, you can comment out that particular test case while the others are processed.

In the test harness shown above we simply print the result, and count on the programmer running the test harness to verify the result. Sometimes it is possible to check the result directly. For example, if you were testing a method to compute a square root you could simply multiply the result by itself and verify that it produced the original number.

Question: Think about testing a sorting algorithm. Can you write a function that would test the result, rather than simply printing it out for the user to validate?

Once you are convinced that individual functions are working correctly, the next step is to combine calls on functions into more complex programs. Again you should perform testing to increase your confidence in the result. This is termed *integration testing*. Often you will uncover errors during integration testing. Once you fix these you should go back and re-execute the earlier test harness to ensure that the changes have not inadvertently introduced any new errors. This process is termed *regression testing*.

Some testing considers only the structure of the input and output values, and ignores the algorithm used to produce the result. This is termed *black box testing*. Other times you want to consider the structure of the function, for example to ensure that every if statement is exercised both with a value that makes it true and a value that makes it false. This is termed *white box testing*. Goals for white box testing should include that every statement is executed, and that every condition is evaluated both true and false. Other more complex test conditions can test the boundaries of a computation.

Testing alone should never be used to guarantee a program is working correctly. The famous computer scientist Edsger Dijkstra pointed out that testing can show the presence of errors but never their absence. Testing should be used in combination with logical thought, assertions, invariants, and proofs of correctness. All have a part to play in the development of a reliable program.

In worksheet 5 you will think about test cases for a variety of simple programs.

More on Program Proofs

We noted earlier in this chapter that the most powerful way to gain confidence in the correctness of a function or program is to develop a *proof* that the function is correct. In this section we will investigate this process in more detail, by examining another classic algorithm, a sorting algorithm named *selection sort*. Selection sort is easy to describe, which is why we study it. But like bubble sort it is also slow, so is not generally used in practice. In later lessons we will examine faster algorithms.

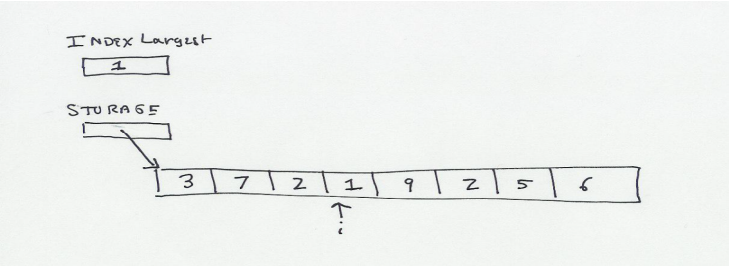
How to sort an array using selection sort: Find the index of the largest element in an array. Swap the largest value into the final location in the array. Then do the same with the next largest element. Then the next largest, and so on until you reach the smallest element. At that point the array will be sorted.

To develop this algorithm as executable code the first step is to isolate the smallest portion of the problem description that could be independently programmed, tested, and debugged. In this case you might select that first sentence: "find the index of the largest element in an array". How do you do that? The best way seems to be a loop.

```
double storage [ ]; /* size is n */
  ...
int indexLargest = 0;
for (int i = 1; i <=  n-1; i++) {
  if (storage[i] > storage[indexLargest])
    indexLargest = i;
}
```

How do we know this small bit of code is correct? As we discussed in the previous lessons, there are two general techniques that are used, and you should *always* use *both* of them. These two techniques are proofs and testing.

A *proof of correctness* is an informal argument that explains why you believe the code is correct. As you learned earlier, such proofs are built around *assertions*, which are statements that describe the relationships between variables when the computer reaches a point in execution. Using assertions, you simulate the execution of the algorithm in your mind, and argue both that the assertions are valid, and that they lead to the correct outcome.



In the code fragment above, we know that in the middle of execution the variable i represents some indefinite memory location. We have examined all values up to i, and indexLargest represents the largest value in that range. The values beyond index i have not yet been examined, and are therefore unknown. A drawing helps illustrate the relationships. What can you say about the relationship between i, indexLargest, and the data array? Invariants are written as comments, as in the following:

```
double storage [ ];
  ...
int position = n – 1;
int indexLargest = 0;
for (int i = 1; i <=  position; i++) {
  // inv: indexLargest is the index of the largest element in the range 0 .. (i-1)
  // (see picture)
  if (storage[i] > storage[indexLargest])
    indexLargest = i;
  // inv: indexLargest is the index of the largest element in the range 0 .. i
}
```

Notice how the invariant that comes after the if statement is a simple variation on the one that comes before. This is almost always the case. Once you find the pattern, then it becomes clear what the assertions must be that begin and end the loop. The first asserts what must be true before the loop starts, and the last must be what we want to be true after the loop finishes, which is normally the outcome we desire. These could be written as follows. We have numbered the invariants to help in the subsequent discussion:

```
double storage [ ];
   ...
int indexLargest = 0;
int position = n – 1;
// 1. indexLargest is the index of the largest element in the range 0 .. 0
for (int i = 1; i <=  position; i++) {
   // 2. indexLargest is the index of the largest element in the range 0 .. (i-1)
   if (storage[i] > storage[indexLargest])
      indexLargest = i;
   // 3. indexLargest is the index of the largest element in the range 0 .. i
}
// 4. indexLargest is index of largest element in  the range 0 .. (storage.length-1)
```

Identifying invariants is the first step. The next step is to form these into a proof that the program fragment produces the correct outcome. This is accomplished by a series of small arguments. Each argument moves from one invariant to the next, and use the knowledge you have of how the programming language changes the value of variables as execution progresses. Typically these arguments are very simple.

From invariant 2 to invariant 3. Here we assume invariant 2 is true and that we know nothing more about the variable i. But the if statement is checking the value of storage location i. If location i is the new largest element the value of indexLargest is changed. If it is not, then the previous largest value remains the correct index. Hence, if invariant 2 is true prior to the if statement, then invariant 3 must be true following the statement.

From invariant 3 back to invariant 2. This moves from the bottom of the loop back to the top. But during this process variable i is incremented by one. So if invariant 3 is true, and i is incremented, then invariant 2 is asserting the same thing.

All this may seem like a lot of work, but with practice loop invariants and proofs of correctness can become second nature, and seldom require as much analysis as we have presented here.

We return now to the development of the selection sort algorithm:

> **How to sort an array using selection sort:** Find the index of the largest element in an array. Swap this value into the final location in the array. Then do the same with the next largest element. Then the next largest, and so on until you reach the smallest element. At that point the array will be sorted.

Finding the largest value is a task that must be performed repeatedly. It is first performed to find the largest element in the array, and then the next largest, and then the one before that, and so on. So again a loop seems to be called for. Since we are looking for the largest value to fill a given position, let us name this loop variable position. The loop that is filling this variable looks as follows:

```
for (position = n – 1; position > 0; position--) {
  // find the largest element in 0 .. position
  int indexLargest = 0;
  ...
  // then swap into place
  swap(storage, indexLargest, position);
  // what is the invariant here?
}
```

Question: What can you say is true when execution reaches the comment at the end of the loop? What is true the first time the loop is executed? What can you say about the elements with index values larger than or equal to position after each succeeding iteration?

The selection sort algorithm combines the outer loop with the code developed earlier, wrapping this into a general method that we will name selectionSort.

In worksheets 6 and 7 you gain more experience working with invariants and proofs of correctness. These will introduce you to yet more sorting algorithms, termed gnome sort and insertion sort. The latter is very practical, and is often used to sort small arrays. (We will subsequently describe other algorithms that are even more efficient on large collections).

Proofs involving Multiple Functions

When a function calls another function, a proof assumes the called function will work as advertised. Assume that you have previously verified that the isPrime function works correctly. Use this fact to show the following prints the values of all the prime numbers from 2 to n:

```
void printPrimes (int n) {
  for (int i = 2; i <= n; i++) {
    if (isPrime(i))
      System.out.println("Number " + i + " is prime");
  }
}
```

A special case is the analysis of recursive functions. Here you first argue that the base case is correct, and then argue that the recursive case must be correct, assuming that the base case performs as defined. Provide assertions that demonstrate the following prints the binary representation of a number, assuming that the argument is positive.

```
void printBinary (int i) {
  assert (i >= 0);
  if ((i == 0) || (i == 1))
    print(i);
  else {
    printBinary(i/2);
    print(i%2);
  }
}
```

Recursion and Mathematical Induction

The analysis of recursive functions frequently mirrors the technique of *mathematical induction*. Both work by establishing a base case, then *reducing* other inputs until they reach the base case. In the printBinary function shown above, the base cases are the values zero and one. These are handed as a special case. All other values are handed by stripping off one binary digit, then invoking the function with a smaller number. Since on each iteration the number is smaller, it must eventually reach the base case.

To illustrate the similarities, recall the mathematical induction proof that the sum 1 + 2 + … + n is (n * (n + 1)) / 2. To prove this, we first verify that it is true for simple base cases, such as n=0, n=1, and n=2. Next, we will *assume* that it is true for all values smaller than n, and prove it is true for n. But if we have a sum from 1 to n, we can group all but the last term together.

$1 + 2 + \ldots + n = (1 + 2 + \ldots + (n-1)) + n$

Our induction hypothesis tells us that the first part is ((n - 1) * n) / 2. So the entire sum is ((n – 1) * n) / 2 + n. Simple arithmetic then shows that this is (n * (n + 1)) / 2. The argument shows that the result will hold for all positive integers

```
double exp (double a, int n) {
  if (n == 0) return 1.0;
  if (n == 1) return a;
  if (0 == n%2) return exp(a*a, n/2);
  else return a * exp(a, n-1);
}
```

Compare the structure of the preceding argument to the type of analysis you would do on a recursive function. Suppose, for example, we want to show that the function shown at right computes the value a raised to the n^{th} power. To prove this, we first verify that it works for simple base cases, such as n=0 and n=1. Next, we will assume that it works correctly for all values less than n, and prove that the function works correctly for n. To do this, we note two simple observations. If n is even, then a^n is the same as $(a*a)^{(n/2)}$. And if n is odd, then a^n is the same as $a * a^{(n-1)}$. Since both of these have exponents that are smaller than n, our assumption tells us that they will be correctly handled. Therefore the correct result is produced.

In later chapters we will encounter many recursive algorithms, and so it is important to become comfortable with the technique and understand the analysis of these functions.

Self Study Questions

1. What does it mean to say you are debugging a function?

2. What are some useful hints to help you debug a function or program?

3. What is an assertion?

4. What is an invariant?

5. Once you have identified assertions and invariants, how do you create a proof of correctness?

6. How is an assertion different from an assertion statement?

7. What is testing?

8. What is unit testing?

9. What is a test harness?

10. What is boundary testing?

11. What are some example boundary conditions?

12. What is a test suite?

13. What is integration testing?

14. What is regression testing and why is it performed?

15. What is black box testing?

16. What is white box testing?

17. Give an informal description in English (not code) explaining how the bubble sort algorithm operates.

18. Give a similar description of the selection sorting algorithm.

19. In what ways does the analysis of recursive algorithms mirror the idea of mathematical induction?

Exercises

1. Using only the specification (that is, black box testing), what are some test cases for the sqrt function described in the previous chapter?

2. What would be some good test cases for the min function described in the previous chapter?

3. Using assertions and invariants, prove that the min function described in the previous chapter produces the correct result.

4. The GCD function in the previous chapter required the input values to be positive, but did not check this condition. Add assertion statements that will verify this property.

5. Prove, by mathematical induction, that the sum of powers of 2 is one less than the next higher power. That is, for any non-negative integer n:

   sum I = 0 to n of 2^I equals $2^{n+1} - 1$

Analysis Exercises

1. Compare the selection sort algorithm with the bubble sort algorithm described in Lesson 4. How are they similar? How are they different?

2. What would be good test cases to exercise the bubble sort algorithm? Explain what property is being tested by our test cases.

3. What would be good test cases to exercise the selection sort algorithm?

4. From the specifications alone develop test cases for the triangle program. Then determine what value the program would produce for your test cases. Would your test cases have exposed the error?

5. What would be good test cases to exercise the binary search algorithm? Explain what property is being tested by each test case.

6. Using the invariants you discovered earlier, provide a proof of correctness for bubble sort. Do this by showing arguments that link each invariant to the next.

7. Using the invariant described in the section on binary search, provide a proof of correctness. Do this by showing the invariant is true at the start of execution, remains true after each execution of the while loop, and is still true when the while loop terminates.

8. In worksheet 4 you are asked to develop invariants for a number of functions. Having done so, number your invariants and provide short proofs for each path that leads from one invariant to the next.

9. Finish writing the invariants for selection sort, and then provide a proof of correctness by presenting arguments that link each invariant to the next.

10. Although Euclids GCD algorithm, described in the previous chapter, is one of the first algorithms, a proof of correctness is subtle. Traditionally it is divided into two steps. First, showing that the algorithm produces a value that is the divisor of the two input values. Second showing that it is the smallest such number. We will show how to do the first. The argument begins with the assumption that the divisor exists, even if we do not yet know its value. Let us call this divisor d. From basic arithmetic, we

know that if a and b are integers, and a > b, and d divides both a and b, then d must also divide (a-b). Using this hint, show that if d is a divisor of n and m when the function is first called, then d will be a divisor of n and m at each iteration of the while loop. The algorithm halts when n is equal to m, and so d is a divisor to both.

11. A sorting algorithm is said to be *stable* if two equal values retain their same relative ordering after sorting. Can you prove that bubble sort is stable? What about insertion sort?

12. The wikipedia entry for bubble sort includes a Boolean variable, named swapped, that is initially false inside the inner loop, and set to true if any two values are swapped. Explain why this can be used to improve the speed of the algorithm.

13. What is wrong with the following induction proof that for all positive numbers a and integer n, it must be true that a^{n-1} is 1. For the base case, have that for n = 1, a^{n-1} is a^0 which is 1. For the induction case assume it is true for 1, 2, 3, … n. To verify the condition for n+1, we have

$$a^{(n+1)-1} = a^n = (a^{n-1} * a^{n-1}) / a^{n-2} = 1 * 1 / 1 = 1$$

14. Explain why the following inductive argument cannot be used to demonstrate that all horses in a given corral are the same color. Suppose there is one horse in a correct, and that it is white. Thus, we have a base case, since for N equal to 1, all horses in the correct are white. Now add a second horse. The corral still contains the first horse, so we remove it. We have now reduced to our base case, and the horse that we removed was white, so both horses must be white. So for N equal to 2 we have our proof. We can continue in this fashion and show, no matter how many horses are in the corral, that they are all white.

Programming Assignments

1. Create a test harness to test the bubble sort algorithm. Feed the algorithm a variety of test cases and verify that it produces the correct result? Can you write a function that verifies the correct result instead of simply printing the values and asking the programmer if they are correct?

2. Do a similar task for the selection sort algorithm.

3. Compare empirically the running time of bubble sort and insertion sort. Do this by creating an array of size N containing random values, then time the sorting algorithm as it sorts the array. Print out the times for values of N ranging from 100 to 1000 in increments of 100.

4. In the next chapter we will show that the running time of both bubble sort and selection sort is proportional to the square of the number of elements. You can easily

see this empirically. Program a test harness that creates an array of random values of size n. Then time the execution of the bubble sort algorithm as it sorts this random array. Plot the running times for n = 100, 200, 300 up to 1000, and see what sort of graph this resembles.

On the Web

Wikipedia includes a very complete discussion of testing under the entry “Software Testing”. Related entries include “test case”, “unit testing”, “integration testing”, “white-box testing”, “black-box testing”, “debugging” and “software verification”. The entry for “bubble sort” includes an interactive demonstration, as well as detailed discussions of why it is not a good algorithm in practice. Selection sort is also the topic of a wikipedia entry. The wikipedia entry on “assertion(computing)” contains a link to an excellent article by computer science pioneer Tony Hoare on the development and use of assertions.