1. We suppose that initially $x = 0$. The segment causes two things to happen. First y is assigned the value of 1. Next the value of $x + y$ is computed to be $0 + 1 = 1$, and so z is assigned the value 1. Therefore at the end z has the value 1, so the final assertion is satisfied.

3. We suppose that initially $y = 3$. The effect of the first two statements is to assign x the value 2 and z the value $2 + 3 = 5$. Next, when the **if**... **then** statement is encountered, since the value of y is 3, and $3 > 0$ is true, the statement $z := z + 1$ assigns the value $5 + 1 = 6$ to z (and the **else** clause is not executed). Therefore at the end, z has the value 6, so the final assertion $z = 6$ is true.

5. We generalize the rule of inference for the **if**... **then**... **else** statement, given before Example 3. Let p be the initial assertion, and let q be the final assertion. If *condition* 1 is true, then S_1 will get executed, so we need $(p \wedge \textit{condition } 1)\{S_1\}q$. Similarly, if *condition* 2 is true, but *condition* 1 is false, then S_2 will get executed, so we need $(p \wedge \neg(\textit{condition } 1) \wedge \textit{condition } 2)\{S_2\}q$. This pattern continues, until the last statement we need is $(p \wedge \neg(\textit{condition } 1) \wedge \neg(\textit{condition } 2) \wedge \cdots \wedge \neg(\textit{condition } n-1))\{S_n\}q$. Given all of these, we can conclude that pTq, where T is the entire statement. In symbols, we have

$$\begin{array}{l} (p \wedge \textit{condition } 1)\{S_1\}q \\ (p \wedge \neg(\textit{condition } 1) \wedge \textit{condition } 2)\{S_2\}q \\ \quad\vdots \\ (p \wedge \neg(\textit{condition } 1) \wedge \neg(\textit{condition } 2) \wedge \cdots \wedge \neg(\textit{condition } n-1))\{S_n\}q \\ \hline \therefore\ p\{\textbf{if } \textit{condition } 1 \textbf{ then } S_1 \textbf{ else if } \textit{condition } 2 \textbf{ then } S_2 \ldots \textbf{else } S_n\}q. \end{array}$$

7. The problem is similar to Example 4. We will use the loop invariant p: "$\textit{power} = x^{i-1}$ and $i \le n + 1$." Now p is true initially, since before the loop starts, $i = 1$ and $\textit{power} = 1 = x^0 = x^{1-1}$. (There is a technicality here: we define 0^0 to equal 1 in order for this to be correct if $x = 0$. There is no harm in this, since $n > 0$, so if $x = 0$, then the program certainly computes the correct answer $x^n = 0$.) We must now show that if p is true and $i \le n$ before some pass through the loop, then p remains true after that pass. The loop increments i by one. Hence since $i \le n$ before this pass through the loop, $i \le n+1$ after this pass. Also the loop assigns $\textit{power} \cdot x$ to $\textit{power}$. By the inductive hypothesis, $\textit{power}$ started with the value x^{i-1} (the old value of i). Therefore its new value is $x^{i-1} \cdot x = x^i = x^{(i+1)-1}$. But since $i + 1$ is the new value of i, the statement $\textit{power} = x^{i-1}$ is true at the completion of this pass through the loop. Hence p remains true, so p is a loop invariant. Furthermore, the loop terminates after n traversals, with $i = n + 1$, since i is assigned the value 1 prior to entering the loop, i is incremented by 1 on each pass, and the loop terminates when $i > n$. At termination we have $(i \le n + 1) \wedge \neg(i \le n)$, so $i = n + 1$. Hence $\textit{power} = x^{(n+1)-1} = x^n$, as desired.

9. We will break the problem up into the various statements that are left unproved in Example 5.

We must show that $p\{S_1\}q$, where p is the assertion that m and n are integers, and q is the proposition $p \wedge (a = |n|)$. This follows from Example 3 and the fact that the values of m and n have not been tampered with.

We must show that $q\{S_2\}r$, where r is the proposition $q \wedge (k = 0) \wedge (x = 0)$. This is clear, since in S_2, none of m, n, or a have been altered, but k and x have been assigned the value 0.

We must show that $(x = mk) \wedge (k \le a)$ is an invariant for the loop in S_3. Assume that this statement is true before a pass through the loop. In the loop, k is incremented by 1. If $k < a$ (the condition of the loop), then at the end of the pass, $k < a + 1$, so $k \le a$. Furthermore, since $x = mk$ at the beginning of the pass, and x is incremented by m inside the loop, we have $x = mk + m = m(k + 1)$ at the end of the pass, which is precisely the statement that $x = mk$ for the updated value of k. When the loop terminates (which it clearly does after a iterations), clearly $k = a$ (since $(k \le a) \wedge \neg(k < a)$), and so $x = ma$ at this point.

Finally we must show that $s\{S_4\}t$, where s is the proposition $(x = ma) \wedge (a = |n|)$, and t is the proposition $\mathit{product} = mn$. The program segment S_4 assigns the value x or $-x$ to $\mathit{product}$, depending on the sign of n. We need to consider the two cases. If $n < 0$, then since $a = |n|$ we know that $a = -n$. Therefore $\mathit{product} = -x = -(ma) = -m(-n) = mn$. If $n \not< 0$, then since $a = |n|$ we know that $a = n$. Therefore $\mathit{product} = x = ma = mn$.

11. To say that the program assertion $p\{S\}q_1$ is true is to say that if p is true before S is executed, then q_1 is true after S is executed. To prove this, let us assume that p is true and then S is executed. Since $p\{S\}q_0$, we know that after S is executed q_0 is true. By modus ponens, since q_0 is true and $q_0 \rightarrow q_1$ is true, we know that q_1 is true, as desired.

13. Our loop invariant p is the proposition "$\gcd(a,b) = \gcd(x,y)$ and $y \geq 0$." First note that p is true before the loop is entered, since at that point $x = a$, $y = b$, and y is a positive integer (we use the initial assertion here). Now assume that p is true and $y > 0$; then the loop will be executed again. Within the loop x and y are replaced by y and x **mod** y, respectively. According to Lemma 1 in Section 3.6, $\gcd(x,y) = \gcd(y, x \mathbin{\mathbf{mod}} y)$. Therefore after the execution of the loop, the value of $\gcd(x,y)$ remains what it was before. Furthermore, since y is a remainder, it is still greater than or equal to 0. Hence p remains true—it is a loop invariant. Furthermore, if the loop terminates, then it must be the case that $y = 0$. In this case we know that $\gcd(x,y) = x$, the desired final assertion. Therefore the program, which gives x as its output, has correctly computed $\gcd(a,b)$. Finally (although this is not necessary to establish *partial* correctness), we can prove that the loop must terminate, since each iteration causes the value of y to decrease by at least 1 (by the definition of **mod**). Thus the loop can be iterated at most b times.

split into two lists, L_1 and L_2. By the inductive hypothesis, *mergesort* correctly sorts each of these sublists, and so it remains only to show that *merge* correctly merges two sorted lists into one. This is clear, from the code given in Algorithm 10, since with each comparison, the smallest element in $L_1 \cup L_2$ not yet put into L is put there.

51. We need to compare every other element with a_1. Thus at least $n-1$ comparisons are needed (we will assume for Exercises 53 and 55 that the answer is exactly $n-1$). The actual number of comparisons depends on the actual encoding of the algorithm. With any reasonable encoding, it should be $O(n)$.

53. In our analysis we assume that a_1 is considered to be put between the two sublists, not as the last element of the first sublist (which would require an extra pass in some cases). In the worst case, the original list splits into lists of length 3 and 0 (with a_1 between them); by Exercise 51, this requires $4-1=3$ comparisons. No comparisons are needed to sort the second of these lists (since it is empty). To sort the first, we argue in the same way: the worst case is for a splitting into lists of length 2 and 0, requiring $3-1=2$ comparisons. Similarly, $2-1=1$ comparison is needed to split the list of length 2 into lists of length 1 and 0. In all, then, $3+2+1=6$ comparisons are needed in this worst case. (One can prove that this discussion really does deal with the worst case by looking at what happens in the various other cases.)

55. In our analysis we assume that a_1 is considered to be put between the two sublists, not as the last element of the first sublist (which would require an extra pass in some cases). We claim that the worst case complexity is $n(n-1)/2$ comparisons, and we prove this by induction on n. This is certainly true for $n=1$, since no comparisons are needed. Otherwise, suppose that the initial split is into lists of size k and $n-k-1$. By the inductive hypothesis, it will require $\big(k(k-1)/2\big)+\big((n-k-1)(n-k-2)/2\big)$ comparisons in the worst case to finish the algorithm. This quadratic function of k attains its maximum value if $k=0$ (or $k=n-1$), namely the value $(n-1)(n-2)/2$. Also, it took $n-1$ comparisons to perform the first splitting. If we add these two quantities ($(n-1)(n-2)/2$ and $n-1$) and do the algebra, then we obtain $n(n-1)/2$, as desired. Thus in the worst case the complexity is $O(n^2)$.

SECTION 4.5 Program Correctness

Entire books have been written on program verification; obviously here we barely scratch the surface. In some sense program verification is just a very careful stepping through a program to prove that it works correctly. There should be no problem verifying anything except loops; indeed it may seem that there is nothing to prove. Loops are harder to deal with. The trick is to find the right invariant. Once you have the invariant, it again is really a matter of stepping through one pass of the loop to make sure that the invariant is satisfied at the end of that pass, given that it was satisfied at the beginning. Analogous to our remark in the comments for Section 1.6, there is a deep theorem of logic (or theoretical computer science) that says essentially that there is no algorithm for proving correct programs correct, so the task is much more an art than a science.

Your proofs must be valid proofs, of course. You may use the rules of inference discussed in Section 1.5. What is special about proofs of program correctness is the addition of some special rules for this setting. These exercises (and the examples in the text) may seem overly simple, but unfortunately it is extremely hard to prove all but the simplest programs correct.