Introduction

From determining the cheapest way to make a hot dog to monitoring the workings of a factory, there are many complex *computational problems* to be solved. Before executable *code* can be produced, computer scientists need to be able to design the *algorithms* that lie behind the code, be able to understand and describe such algorithms abstractly, and be confident that they work correctly and efficiently. These are the goals of computer scientists.

A Computational Problem: A specification of a computational problem uses *preconditions* and *postconditions* to describe for each legal *input instance* that the computation might receive, what the required output or actions are. This may be a function mapping each input instance to the required output. It may be an optimization problem that requires a solution to be outputted that is "optimal" from among a huge set of possible solutions for the given input instance. It may also be an ongoing system or data structure that responds appropriately to a constant stream of input.

Example: The *sorting* problem is defined as follows:

Preconditions: The input is a list of n values, including possible repetitions.

Postconditions: The output is a list consisting of the same n values in nondecreasing order.

An Algorithm: An *algorithm* is a step-by-step procedure that, starting with an input instance, produces a suitable output. It is described at the level of detail and abstraction best suited to the human audience who must understand it. In contrast, *code* is an implementation of an algorithm that can be executed by a computer. *Pseudocode* lies between these two.

An Abstract Data Type: Computers use zeros and ones, ANDS and ORS, IFS and GOTOS. This does not mean that *we* have to. The description of an algorithm may talk of *abstract objects* such as integers, reals, strings, sets, stacks, graphs, and trees; *abstract operations* such as "sort the list," "pop the stack," or "trace a path"; and *abstract relationships* such as greater than, prefix, subset, connected, and child. To be useful, the nature of these objects and the effect of these operations need to be understood. However, in order to hide details that are tedious or irrelevant, the precise implementations of these data structure and algorithms do not need to be specified. For more on this see Chapter 3.

Correctness: An algorithm for the problem is *correct* if for every legal input instance, the required output is produced. Though a certain amount of logical thinking is requireds, the goal of this text is to teach how to think about, develop, and describe algorithms in such way that their correctness is transparent. See Chapter 40 for the formal steps required to prove correctness, and Chapter 34 for a discussion of *forall* and *exist* statements that are essential for making formal statements.

Running Time: It is not enough for a computation to eventually get the correct answer. It must also do so using a reasonable amount of time and memory space. The *running time* of an algorithm is a function from the *size n* of the input instance given to a bound on the number of *operations* the computation must do. (See Chapter 35.) The algorithm is said to be *feasible* if this function is a *polynomial* like $Time(n) = \Theta(n^2)$, and is said to be *infeasible* if this function is an *exponential* like $Time(n) = \Theta(2^n)$. (See Chapters 36 and 37 for more on the asymptotics of functions.) To be able to compute the running time, one needs to be able to add up the times taken in each iteration of a loop and to solve the recurrence relation defining the time of a recursive program. (See Chapter 38 for an understanding of $\sum_{i=1}^{n} i = \Theta(n^2)$, and Chapter 39 for an understanding of $T(n) = 2T(\frac{n}{2}) + n = \Theta(n \log n)$.)

Meta-algorithms: Most algorithms are best described as being either *iterative* or *recursive*. An iterative algorithm (Part I) takes one step at a time, ensuring that each step makes *progress* while maintaining the *loop invariant*. A recursive algorithm (Part II) breaks its instance into smaller instances, which it gets a *friend* to solve, and then combines their solutions into one of its own.

Optimization problems (Part III) form an important class of computational problems. The key algorithms for them are the following. *Greedy* algorithms (Chapter 22) keep grabbing the next object that looks best. *Recursive backtracking* algorithms (Chapter 23) try things and, if they don't work, backtrack and try something else. *Dynamic programming* (Chapter 24) solves a sequence of larger and larger instances, reusing the previously saved solutions for the smaller instances, until a solution is obtained for the given instance. *Reductions* (Chapter 28) use an algorithm for one problem to solve another. *Randomized* algorithms (Chapter 29) flip coins to help them decide what actions to take. Finally, *lower bounds* (Chapter 7) prove that there are no faster algorithms.

Part I

Iterative Algorithms and Loop Invariants

1 Iterative Algorithms: Measures of Progress and Loop Invariants

Using an *iterative algorithm* to solve a computational problem is a bit like following a road, possibly long and difficult, from your start location to your destination. With each iteration, you have a method that takes you a single step closer. To ensure that you move forward, you need to have a *measure of progress* telling you how far you are either from your starting location or to your destination. You cannot expect to know exactly where the algorithm will go and you need to expect some weaving and winding. On the other hand, you do not want to have to know how to handle every ditch and dead end in the world. A compromise between these two is to have a *loop invariant*, which defines a road (or region) that you may not leave. As you travel, worry about one step at a time. You must know how to get onto the road from any start location. From every place along the road, you must know what actions you will take in order to step forward while not leaving the road. Finally, when sufficient progress has been made along the road, you must know how to exit and reach your destination. Hopefully, this happens in a reasonable amount of time.

1.1 A Paradigm Shift: A Sequence of Actions vs. a Sequence of Assertions

Understanding iterative algorithms requires understanding the difference between a *loop invariant*, which is an *assertion* or picture of the computation at a particular point in time, and the actions that are required to maintain such a loop invariant.

One of the first important paradigm shifts that programmers struggle to make is from viewing an algorithm as a sequence of actions to viewing it as a sequence of snapshots of the state of the computer. Programmers tend to fixate on the first view, because code is a sequence of instructions for action and a computation is a sequence of actions. Though this is an important view, there is another. Imagine stopping time at key points during the computation and taking still pictures of the state of the computer. Then a computation can equally be viewed as a sequence of such snapshots. Having two ways of viewing the same thing gives one both more tools to handle it and a deeper understanding of it. An example of viewing a computation as an alteration between assertions about the current state of the computation and blocks of actions that bring the state of the computation to the next state is shown here.

```
Max(a, b, c)
    PreCond: Input has 3 numbers.
    $m = a$
    assert: $m$ is max in $\{a\}$.
    if($b > m$)
        $m = b$
    end if
    assert: $m$ is max in $\{a, b\}$.
    if($c > m$)
        $m = c$
    end if
    assert: $m$ is max in $\{a, b, c\}$.
    return($m$)
    PostCond: return max in $\{a, b, c\}$.
end algorithm
```

The Challenge of the Sequence-of-Actions View Suppose one is designing a new algorithm or explaining an algorithm to a friend. If one is thinking of it as sequence of actions, then one will likely start at the beginning: Do this. Do that. Do this. Shortly one can get lost and not know where one is. To follow this, one needs to keep track of how the state of the computer changes with each new action. In order to know what action to take next, one needs to have a global plan of where the computation is to go. To make it worse, the computation has many IFS and LOOPS so one has to consider all the various paths that the computation may take.

The Advantages of the Sequence of Snapshots View: This new paradigm is a useful one from which one can think about, explain, or develop an algorithm.

Pre- and Postconditions: Before one can consider an algorithm, one needs to carefully define the computational problem being solved by it. This is done with pre- and postconditions by providing the initial picture, or *assertion*, about the input instance and a corresponding picture or assertion about required output.

Proof of Correctness of the Algorithm: It is important that you *know* that your code works. The proof might be a formal mathematical one or it might be informal hand waving. Either way, the formal statement of what needs to be proved is as follows.

$$PreCond \ \& \ code \Rightarrow PostCond$$

This states that if the input instance happens to meet the precondition and the code is executed, then in the end the output will meet the postcondition. On the other hand, if the precondition was not met, then except for being polite, there are no requirements on what your code does.

Check Each Computation Path: The computation from pre- to postcondition will follow one of possibly many paths. For each such path, you need to be confident that the postcondition will be met in the end. Suppose the above Max(a, b, c) code had not two *if then else* statements but a sequence of r of them. There would then be 2^r paths though which the computation might follow. Checking each of these would be too much work. Instead, we will break the computation into subcomputations.

One Step at a Time: A main theme of this book is to implore you not to think about the entire computation all at once. As recommended in Alcoholics Anonymous, worry about one step at a time.

Start in the Middle: Instead of starting with the first line code, an alternative way to design an algorithm is to jump into the middle of the computation and to draw a static picture, or assertion, about the state we would like the computation to be in at this time. This picture does not need to state the exact value of each variable. Instead, it gives general properties and relationships between the various data structures that are key to understanding the algorithm. If this *assertion* is sufficiently general, it will capture not just this one point during the computation, but many similar points. Then it might become a part of a loop.

Sequence of Snapshots: Once one builds up a sequence of assertions in this way, one can see the entire path of the computation laid out before one.

Fill in the Actions: These assertions are just static snapshots of the computation with time stopped. No actions have been considered yet. The final step is to fill in actions (code) between consecutive assertions.

One Step at a Time: Each such block of actions can be executed completely independently of the others. It is much easier to consider them one at a time than to worry about the entire computation at once. In fact, one can complete these blocks in any order one wants and modify one block without worrying about the effect on the others.

Fly In from Mars: This is how you should fill in the code between the ith and the $i+1^{\text{st}}$ assertions. Suppose you have just flown in from Mars, and absolutely the only thing you know about the current state of your computation is that the ith *assertion* holds. The computation might actually be in a state that is completely impossible to arrive at, given the algorithm that has been designed so far. It is allowing this that provides independence between these blocks of actions.

Take One Step: Being in a state in which the ith assertion holds, your task is simply to write some simple code to do a few simple actions, that change the state of the computation so that the $i+1^{\text{st}}$ assertion holds.

Differentiating between Values: Define x_i to be the value of x when the computation is at the ith assertion and x_{i+1} to be that after the computation has gone around the loop and is at the $i+1^{\text{st}}$. You may prefer the notation x' and x'.

Code vs. Math Assertions: Denote as $code_i$ the assertion of the result of the code between these assertions. Code $x = x+1$, though a fine action, is incorrect as a statement. If you want to prove things you need to translate this into the mathematical assertion $code_i : x_{i+1} = x_i + 1$. Similarly translate the actions $x = x+1$; $y = y+x$; $x = x \times y$ into the assertion $x_{t+1} = (x_t+1)(y_t+x_t+1)$; $y_{t+1} = y_t + x_t + 1$.

Proof of Correctness of Each Step: The proof that your algorithm works can be done one block at a time. The formal statement of what needs to be proved is as follows:

$$\left\langle i^{th} - assertion \right\rangle \;\&\; code_i \;\Rightarrow\; \left\langle i+1^{\text{st}} - assertion \right\rangle.$$

For example, a statement $\langle i^{th} - assertion\rangle$*: x_i is odd* assures us that the state of the computation is such that i^{th} assertion holds for the values x_i. The statement $code_i$: $x_{i+1} = x_i + 1$ assures us that if you restart the code just long enough to execute the next block of code, the stated relationship holds between the x_i values and the x_{i+1} values. Our goal is to combine these two facts to prove that the statement $\langle i+1^{\text{st}} - assertion\rangle$*: x_{i+1} is even* holds for the x_{i+1} values.

Proof by Case: If you can prove: 1) There are only two cases, 2) If the first case is true, then our assertion is true, 3) If the second case is true, then our assertion is true, then we can conclude that our assertion is true.

In the above code, let's assume that computation has paused at the line *assert: m_i is max in* $\{a,b\}$ and that this assertion is true. From here, there are two paths that the computation might follow. If the first path is followed, we are assured that $c > m_i$ and that $m_{i+1} = c$. From these three facts, we can conclude *assert: m_{i+1} is max in* $\{a,b,c\}$. On the other hand, if the second path is followed, we are assured that $c \le m_i$ and that $m_{i+1} = m_i$. From these three facts, we can also conclude *assert: m_{i+1} is max in* $\{a,b,c\}$.

Proof by Transitivity: If we prove $PreCond \Rightarrow assert_1$, $assert_1 \Rightarrow assert_2$, $assert_2 \Rightarrow assert_3$, and $assert_3 \Rightarrow PostCond$, then by transitively we can conclude that $PreCond \Rightarrow PostCond$.

Loops: Once you see the pattern in the above Max(a,b,c) code, we can generalize each of its steps into a generic step and put that into a loop giving the following code.

```
loop
    assert: m is max in $\{L[1], L[2], \ldots, L[j]\}$.
    $j = j+1$
    if( $L[j] > m$ )
        $m = L[j]$
    end if
end loop
```

We prove this is correct just as we did before. The only difference is that we will be hitting the same assertion twice in a row. The natural thing is to assume that the computation has paused at the top of the loop after having iterated some t number of times. We would differentiate between values by defining j_t and m_t to be the current value of these variables. The problem with this is that it might make you tempted to assume something based on what has come before. Instead, suppose you have just flown in from Mars and absolutely the only thing you know is that you are at the top of the loop, j' and m' are the current value of these variables, and the assertion that m' is max in $\{L[1], L[2], \ldots, L[j']\}$ is true. As before, the *if* gives two cases. When the *if* condition is true, then the assertion from the code is $j'' = j' + 1$; $L[j''] > m'$; and $m'' = L[j'']$. From this you must prove that when the computation gets back to the top of the loop, the assertion will again be true, namely, that m'' is max in $\{L[1], L[2], \ldots, L[j'']\}$. You could prove this formally or informally. This proves that once the assertion at the top of the loop has been established, the code will maintain it for as many iterations as needed. We will refer to such assertions as *loop invariants.*

Correctness: Together these techniques formally prove that your code works. Equally important, understanding these steps will help you write correct code in the first place.

Game of Life: Don't worry so much. Taking one step at a time is a lower stress way to live your life.

```
begin routine Life(me)
    ⟨pre − cond⟩: I am born
    Hopefully my parents help
    loop
        ⟨loop − invariant⟩: I am well and reasonably on track
        exit when (I die)
        Maintain the loop invariant
        Make a little progress
        Make the world a little better
    end loop
    ⟨post − cond⟩: It was a well spent and good life
end routine
```

1.2 The Steps to Develop an Iterative Algorithm

Iterative Algorithms: A good way to structure many computer programs is to store the key information you currently know in some data structure and then have each iteration of the main loop take a step toward your destination by making a simple change to this data.

Loop Invariant: A *loop invariant* expresses important relationships among the variables that must be true at the start of every iteration and when the loop terminates. If it is true, then the computation is still on the road. If it is false, then the algorithm has failed.

The Code Structure: The basic structure of the code is as follows.

```
begin routine
    ⟨pre-cond⟩
    $code_{pre\text{-}loop}$ % Establish loop invariant
    loop
        ⟨loop-invariant⟩
```

```
            exit when *exit-cond* ⟩
            $code_{loop}$ % Make progress while maintaining the loop invariant
        end loop
        $code_{post\text{-}loop}$ % Clean up loose ends
        ⟨*post-cond*⟩
end routine
```

Proof of Correctness: Naturally, you want to be sure your algorithm will work on all specified inputs and give the correct answer.

Running Time: You also want to be sure that your algorithm completes in a reasonable amount of time.

The Most Important Steps: If you need to design an algorithm, do not start by typing in code without really knowing how or why the algorithm works. Instead, I recommend first accomplishing the following tasks. See Figure 1.1. These tasks need to fit together in very subtle ways. You may have to cycle through them a number of times, adjusting what you have done, until they all fit together as required.

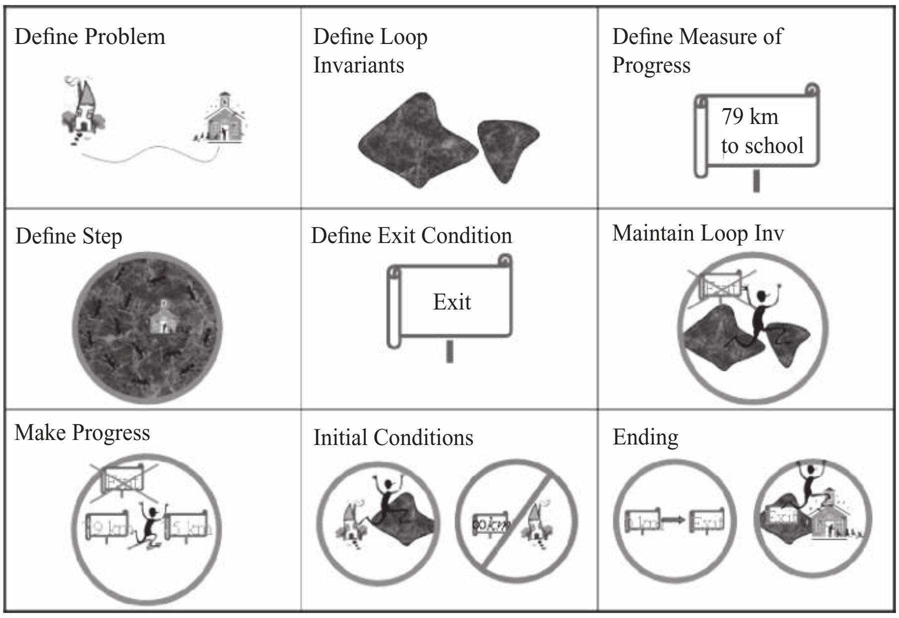

Fig. 1.1 The requirements of an iterative algorithm.

1) Specifications: What problem are you solving? What are its pre- and postconditions—i.e., where are you starting and where is your destination?

2) Basic Steps: What basic steps will head you more or less in the correct direction?

3) Measure of Progress: You must define a measure of progress: where are the distance markers along the road?

4) The Loop Invariant: You must define a loop invariant that will give a picture of the state of your computation when it is at the top of the main loop, in other words, define the road that you will stay on.

5) Main Steps: For every location on the road, you must write the pseudocode $code_{loop}$ to take a single step. You do not need to start with the first location. I recommend first considering a typical step to be taken during the middle of the computation.

6) Make Progress: Each iteration of your main step must make progress according to your measure of progress.

7) Maintain Loop Invariant: Each iteration of your main step must ensure that the loop invariant is true again when the computation gets back to the top of the loop. (Induction will then prove that it remains true always.)

8) Establishing the Loop Invariant: Now that you have an idea of where you are going, you have a better idea about how to begin. You must write the pseudocode $code_{pre\text{-}loop}$ to initially establish the loop invariant. How do you get from your house onto the correct road?

9) Exit Condition: You must write the condition $\langle exit\text{-}cond \rangle$ that causes the computation to break out of the loop.

10) Ending: How does the exit condition together with the invariant ensure that the problem is solved? When at the end of the road but still on it, how do you produce the required output? You must write the pseudocode $code_{pre\text{-}loop}$ to clean up loose ends and to return the required output.

11) Termination and Running Time: How much progress do you need to make before you know you will reach this exit? This is an estimate of the running time of your algorithm.

12) Special Cases: When first attempting to design an algorithm, you should only consider one general type of input instances. Later, you must cycle through the steps again considering other types of instances and special cases. Similarly, test your algorithm by hand on a number of different examples.

13) Coding and Implementation Details: Now you are ready to put all the pieces together and produce pseudocode for the algorithm. It may be necessary at this point to provide extra implementation details.

14) Formal Proof: If the above pieces fit together as required, then your algorithm works.

Example 1.2.1 (**The Find-Max Two-Finger Algorithm to Illustrate These Ideas**)

1) Specifications: An input instance consists of a list $L(1..n)$ of elements. The output consists of an index i such that $L(i)$ has maximum value. If there are multiple entries with this same value, then any one of them is returned.