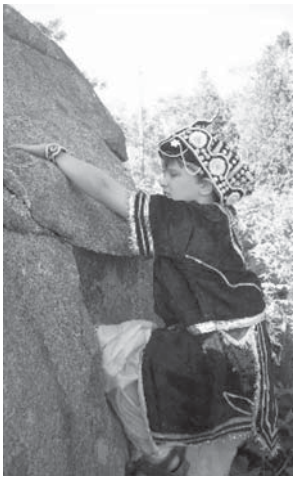
1 Iterative Algorithms: Measures of Progress and Loop Invariants



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 from your destination. You cannot expect to know exactly where the algorithm will go, so 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 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. Hence, we will start with trying to understand this difference.

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 handle this, one simultaneously 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 *IF*s 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 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.

Start in the Middle: Instead of starting with the first line of 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 $i+1$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 $i+1$st assertion holds.

Proof of Correctness of Each Step: The proof that your algorithm works can also be done one block at a time. You need to prove that if time is stopped and the state of the computation is such that the ith assertion holds and you start time again just long enough to execute the next block of code, then when you stop time again the state of the computation will be such that the $i+1$st assertion holds. This proof might be a formal mathematical proof, or it might be informal handwaving. Either way, the formal statement of what needs to be proved is as follows:

$$\langle ith\text{-}assertion\rangle \& \; code_i \;\Rightarrow\; \langle i+1st\text{-}assertion\rangle$$

Proof of Correctness of the Algorithm: All of these individual steps can be put together into a whole working algorithm. We assume that the input instance given meets the precondition. At some point, we proved that if the precondition holds and the first block of code is executed, then the state of the computation will be such

that first assertion holds. At some other point, we proved that if the first assertion holds and the second block of code is executed then the state of the computation will be such that second assertion holds. This was done for each block. All of these independently proved statements can be put together to prove that if initially the input instance meets the precondition and the entire code is executed, then in the end the state of the computation will be such that the postcondition has been met. This is what is required to prove that algorithm works.

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 towards 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-loop   % Establish loop invariant
    loop
        ⟨loop-invariant⟩
        exit when ⟨exit-cond⟩
        code_loop   % Make progress while maintaining the loop invariant
    end loop
    code_post-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

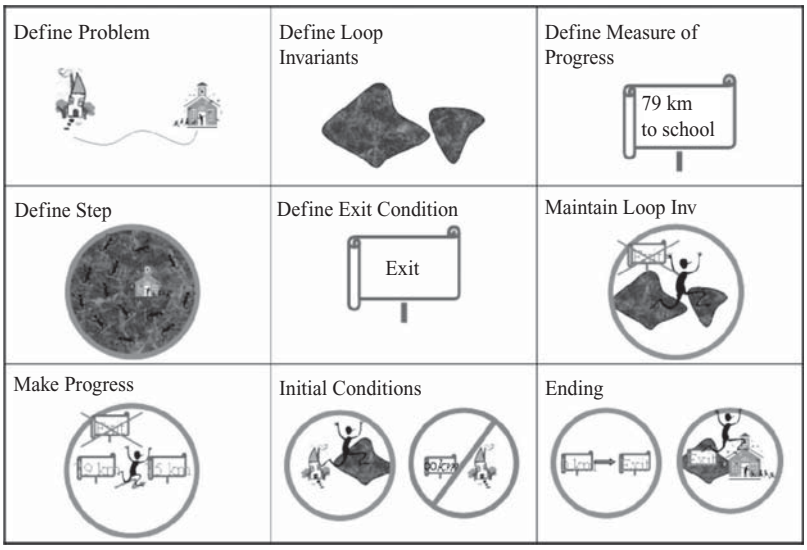


Figure 1.1: *The requirements of an iterative algorithm.*

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.

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 mile 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 ⟨*exit-cond*⟩ 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 $code_{post\text{-}loop}$ 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.

2) Basic Steps: You decide on the two-finger method. Your right finger runs down the list.

3) Measure of Progress: The measure of progress is how far along the list your right finger is.

4) The Loop Invariant: The loop invariant states that your left finger points to one of the largest entries encountered so far by your right finger.

5) Main Steps: Each iteration, you move your right finger down one entry in the list. If your right finger is now pointing at an entry that is larger then the left finger's entry, then move your left finger to be with your right finger.

6) Make Progress: You make progress because your right finger moves one entry.

7) Maintain Loop Invariant: You know that the loop invariant has been maintained as follows. For each step, the new left finger element is Max(old left finger element, new element). By the loop invariant, this is Max(Max(shorter list), new element). Mathematically, this is Max(longer list).

8) Establishing the Loop Invariant: You initially establish the loop invariant by pointing both fingers to the first element.

9) Exit Condition: You are done when your right finger has finished traversing the list.

10) Ending: In the end, we know the problem is solved as follows. By the exit condition, your right finger has encountered all of the entries. By the loop invariant, your left finger points at the maximum of these. Return this entry.

11) Termination and Running Time: The time required is some constant times the length of the list.

12) Special Cases: Check what happens when there are multiple entries with the same value or when $n = 0$ or $n = 1$.

13) Coding and Implementation Details:

algorithm $FindMax(L)$

⟨ ***pre-cond***⟩**:** L is an array of n values.
⟨ ***post-cond***⟩**:** Returns an index with maximum value.

```
begin
    i = 1; j = 1
    loop
        ⟨loop-invariant⟩: L[i] is max in L[1..j].
        exit when (j ≥ n)
        % Make progress while maintaining the loop invariant
        j = j + 1
        if( L[i] < L[j] ) then i = j
    end loop
    return(i)
end algorithm
```

14) Formal Proof: The correctness of the algorithm follows from the above steps.

A New Way of Thinking: You may be tempted to believe that measures of progress and loop invariants are theoretical irrelevancies. But industry, after many expensive mistakes, has a deeper appreciation for the need for correctness. Our philosophy is to learn how to think about, develop, and describe algorithms in such a way that their correctness is transparent. For this, measures of progress and loop invariants are

essential. The description of the preceding algorithms and their proofs of correctness are wrapped up into one.

Keeping Grounded: Loop invariants constitute a life philosophy. They lead to feeling grounded. Most of the code I mark as a teacher makes me feel ungrounded. It cycles, but I don't know what the variables mean, how they fit together, where the algorithm is going, or how to start thinking about it. Loop invariants mean starting my day at home, where I know what is true and what things mean. From there, I have enough confidence to venture out into the unknown. However, loop invariants also mean returning full circle to my safe home at the end of my day.

EXERCISE 1.2.1 *What are the formal mathematical things involving loop invariants that must be proved, to prove that if your program exits then it obtains the postcondition?*

1.3 More about the Steps

In this section I give more details about the steps for developing an iterative algorithm.

1) Specifications: Before we can design an iterative algorithm, we need to know precisely what it is supposed to do.

***Preconditions*:** What are the legal input instances? Any assertions that are promised to be true about the input instance are referred to as *preconditions*.

Postconditions: What is the required output for each legal instance? Any assertions that must be true about the output are referred to as *postconditions*.

Correctness: An algorithm for the problem is *correct* if for every legal input instance, the required output is produced. If the input instance does not meet the preconditions, then all bets are off. Formally, we express this as

$$\langle \textit{pre-cond} \rangle \ \& \ \textit{code}_{alg} \ \Rightarrow \ \langle \textit{post-cond} \rangle$$

This correctness is only with respect to the specifications.

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

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

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

The Contract: Pre- and postconditions are, in a sense, the contract between the implementer and the user (or invoker) of the coded algorithm.

Implementer: When you are writing a subroutine, you can assume the input comes to your program in the correct form, satisfying all the preconditions. You must write the subroutine so that it ensures that the postconditions hold after execution.

User: When you are using the subroutine, you must ensure that the input you provide meets the preconditions of the subroutine. Then you can trust that the output meets its postconditions.

2) Basic Steps: As a preliminary to designing the algorithm it can be helpful to consider what basic steps or operations might be performed in order to make progress towards solving this problem. Take a few of these steps on a simple input instance in order to get some intuition as to where the computation might go. How might the information gained narrow down the computation problem?

3) Measure of Progress: You need to define a function that, when given the current state of the computation, returns an integer value measuring either how much progress the computation has already made or how much progress still needs to be made. This is referred to either as a *measure of progress* or as a *potential function*. It must be such that the total progress required to solve the problem is not infinite and that at each iteration, the computation makes progress. Beyond this, you have complete freedom to define this measure as you like. For example, your measure might state the amount of the output produced, the amount of the input considered, the extent to which the search space has been narrowed, some more creative function of the work done so far, or how many cases have been tried. Section 1.4 outlines how these different measures lead to different types of iterative algorithms.

4) The Loop Invariant: Often, coming up with the loop invariant is the hardest part of designing an algorithm. It requires practice, perseverance, creativity, and insight. However, from it the rest of the algorithm often follows easily. Here are a few helpful pointers.

Definition: A *loop invariant* is an assertion that is placed at the top of a loop and that must hold true every time the computation returns to the top of the loop.

Assertions: More generally, an *assertion* is a statement made at some particular point during the execution of an algorithm about the current state of the computation's data structures that is either true or false. If it is false, then something has gone wrong in the logic of the algorithm. Pre- and postconditions are special cases of assertions that provide clean boundaries between systems, subsystems, routines, and subroutines. Within such a part, assertions can also provide checkpoints along the path of the computation to allow everyone to know what should have been accomplished so far. *Invariants* are the same, except they apply either

to a loop that is executed many times or to an object-oriented data structure that has an ongoing life.

Designing, Understanding, and Proving Correct: Generally, assertions are not tasks for the algorithm to perform, but are only comments that are added to assist the designer, the implementer, and the reader in understanding the algorithm and its correctness.

Debugging: Some languages allow you to insert assertions as lines of code. If during the execution such an assertion is false, then the program automatically stops with a useful error message. This is helpful both when debugging and after the code is complete. It is what is occurring when an error box pops up during the execution of a program telling you to contact the vendor if the error persists. Not all interesting assertions, however, can be tested feasibly within the computation itself.

Picture from the Middle: A loop invariant should describe what you would like the data structure to look like when the computation is at the beginning of an iteration. Your description should leave your reader with a visual image. Draw a picture if you like.

Don't Be Frightened: A loop invariant need not consist of formal mathematical mumbo jumbo if an informal description gets the idea across better. On the other hand, English is sometimes misleading, and hence a more mathematical language sometimes helps. Say things twice if necessary. I recommend pretending that you are describing the algorithm to a first-year student.

On the Road: A loop invariant must ensure that the computation is still on the road towards the destination and has not fallen into a ditch or landed in a tree.

A Wide Road: Given a fixed algorithm on a fixed input, the computation will follow one fixed line. When the algorithm designer knows exactly where this line will go, he can use a very tight loop invariant to define a very narrow road. On the other hand, because your algorithm must work for an infinite number of input instances and because you may pass many obstacles along the way, it can be difficult to predict where the computation might be in the middle of its execution. In such cases, using a very loose loop invariant to define a very wide road is completely acceptable. The line actually followed by the computation might weave and wind, but as long as it stays within the boundaries of the road and continues to make progress, all is well. An advantage of a wide road is that it gives more flexibility in how the main loop is implemented. A disadvantage is that there are then more places where the computation might be, and for each the algorithm must define how to take a step.

Example: As an example of a loose loop invariant, in the find-max two-finger algorithm, the loop invariant does not completely dictate which entry your

left finger should point at when there are a number of entries with the same maximum value.

Meaningful and Achievable: You want a loop invariant that is *meaningful*, meaning it is strong enough that, with an appropriate exit condition, it will guarantee the postcondition. You also want the loop invariant to be *achievable*, meaning you can establish and maintain it.

Know What a Loop Invariant Is: Be clear about what a loop invariant is. It is not code, a precondition, a postcondition, or some other inappropriate piece of information. For example, stating something that is always true, such as "$1 + 1 = 2$" or "The root is the max of any heap," may be useful information for the answer to the problem, but should not be a part of the loop invariant.

Flow Smoothly: The loop invariant should flow smoothly from the beginning to the end of the algorithm.

- At the beginning, it should follow easily from the preconditions.
- It should progress in small natural steps.
- Once the exit condition has been met, the postconditions should easily follow.

Ask for 100%: A good philosophy in life is to ask for 100% of what you want, but not to assume that you will get it.

Dream: Do not be shy. What would you like to be true in the middle of your computation? This may be a reasonable loop invariant, and it may not be.

Pretend: Pretend that a genie has granted your wish. You are now in the middle of your computation, and your dream loop invariant is true.

Maintain the Loop Invariant: From here, are you able to take some computational steps that will make progress while maintaining the loop invariant? If so, great. If not, there are two common reasons.

Too Weak: If your loop invariant is too weak, then the genie has not provided you with everything you need to move on.

Too Strong: If your loop invariant is too strong, then you will not be able to establish it initially or maintain it.

No Unstated Assumptions: You don't want loop invariants that lack detail or are too weak to proceed to the next step. Don't make assumptions that you don't

state. As a check, pretend that you are a Martian who has jumped into the top of the loop knowing *nothing* that is not stated in the loop invariant.

Example: In the find-max two-finger algorithm, the loop invariant does make some unstated assumptions. It assumes that the numbers above your right finger have been encountered by your right finger and those below it have not. Perhaps more importantly for, ± 1 errors, is whether or not the number currently being pointed has been encountered already. The loop invariant also assumes that the numbers in the list have not changed from their original values.

A Starry Night: How did van Gogh come up with his famous painting, *A Starry Night*? There's no easy answer. In the same way, coming up with loop invariants and algorithms is an art form.

Use This Process: Don't come up with the loop invariant after the fact. Use it to design your algorithm.

5) Main Steps: The pseudocode $code_{loop}$ must be defined so that it can be taken not just from where you think the computation might be, but from any state of the data structure for which the loop invariant is true and the exit condition has not yet been met.

Worry about *one step at a time*. Don't get pulled into the strong desire to understand the entire computation at once. Generally, this only brings fear and unhappiness. I repeat the wisdom taught by both the Buddhists and the twelve-step programs: Today you may feel like like you were dropped off in a strange city without knowing how you got there. Do not worry about the past or the future. Be reassured that you are somewhere along the correct road. Your goal is only to take one step so that you make progress and stay on the road. Another analogy is to imagine you are part of a relay race. A teammate hands you the baton. Your job is only to carry it once around the track and hand it to the next teammate.

6) Make Progress: You must prove that progress of at least one unit of your measure is made every time the algorithm goes around the loop. Sometimes there are odd situations in which the algorithm can iterate without making any measurable progress. This is not acceptable. The danger is that the algorithm will loop forever. You must either define another measure that better shows how you are making progress during such iterations or change the step taken in the main loop so that progress is made. The formal proof of this is similar to that for maintaining the loop invariant.

7) Maintain the Loop Invariant: You must prove that the loop invariant is maintained in each iteration.

The Formal Statement: Whether or not you want to prove it formally, the formal statement that must be true is

$$\langle \textit{loop-invariant}' \rangle \ \& \ \textit{not} \langle \textit{exit-cond} \rangle \ \& \ \textit{code}_{\textit{loop}} \ \Rightarrow \ \langle \textit{loop-invariant}'' \rangle$$

Proof Technique:

- Assume that the computation is at the top of the loop.
- Assume that the loop invariant is satisfied; otherwise the program would have already failed. Refer back to the picture that you drew to see what this tells you about the current state of the data structure.
- You can also assume that the exit condition is not satisfied, because otherwise the loop would exit.
- Execute the pseudocode $\textit{code}_{\textit{loop}}$, in one iteration of the loop. How does this change the data structure?
- Prove that when you get back to the top of the loop again, the requirements set by the loop invariant are met once more.

Different Situations: Many subtleties can arise from the huge number of different input instances and the huge number of different places the computation might find itself in.

- I recommend first designing the pseudocode $\textit{code}_{\textit{loop}}$ to work for a general middle iteration when given a large and general input instance. Is the loop invariant maintained in this case?
- Then try the first and last couple of iterations.
- Also try special case input instances. Before writing separate code for these, check whether the code you already have happens to handle these cases. If you are forced to change the code, be sure to check that the previously handled cases still are handled.
- To prove that the loop invariant is true in all situations, pretend that you are at the top of the loop, but you do not know how you got there. You may have dropped in from Mars. Besides knowing that the loop invariant is true and the exit condition is not, you know nothing about the state of the data structure. Make no other assumptions. Then go around the loop and prove that the loop invariant is maintained.

Differentiating between Iterations: The assignment $x = x + 2$ is meaningful as a line of code, but not as a mathematical statement. Define x' to be the value of x at the beginning of the iteration and x'' that after going around the loop one more time. The effect of the code $x = x + 2$ is that $x'' = x' + 2$.

8) Establishing the Loop Invariant: You must prove that the initial code establishes the loop invariant.

The Formal Statement: The formal statement that must be true is

$$\langle \textit{pre-cond} \rangle \ \& \ \textit{code}_{\textit{pre-loop}} \ \Rightarrow \ \langle \textit{loop-invariant} \rangle$$

Proof Technique:
- Assume that you are just beginning the computation.
- You can assume that the input instance satisfies the precondition; otherwise you are not expected to solve the problem.
- Execute the code $code_{pre\text{-}loop}$ before the loop.
- Prove that when you first get to the top of the loop, the requirements set by the loop invariant are met.

Easiest Way: Establish the loop invariant in the easiest way possible. For example, if you need to construct a set such that all the dragons within it are purple, the easiest way to do it is to construct the empty set. Note that all the dragons in this set are purple, because it contains no dragons that are not purple.

Careful: Sometimes it is difficult to know how to set the variables to make the loop invariant initially true. In such cases, try setting them to ensure that it is true after the first iteration. For example, what is the maximum value within an empty list of values? One might think 0 or ∞. However, a better answer is $-\infty$. When adding a new value, one uses the code $newMax = \max(oldMax, newValue)$. Starting with $oldMax = -\infty$, gives the correct answer when the first value is added.

9) Exit Condition: Generally you exit the loop when you have completed the task.

Stuck: Sometimes, however, though your intuition is that your algorithm designed so far is making progress each iteration, you have no clue whether, heading in this direction, the algorithm will ever solve the problem or how you would know it if it happens. Because the algorithm cannot make progress forever, there must be situations in which your algorithm gets stuck. For such situations, you must either think of other ways for your algorithm to make progress or have it exit. A good first step is to exit. In step 10, you will have to prove that when your algorithm exits, you actually are able to solve the problem. If you are unable to do this, then you will have to go back and redesign your algorithm.

Loop While vs Exit When: The following are equivalent:

```
while( A and B )        loop
    ...                     ⟨loop-invariant⟩
end while                   exit when (not A or not B)
                            ...
                        end loop
```

The second is more useful here because it focuses on the conditions needed to exit the loop, while the first focuses on the conditions needed to continue. Another advantage of the second is that it also allows you to slip in the loop invariant between the top of the loop and the exit condition.

10) Ending: In this step, you must ensure that once the loop has exited you will be able to solve the problem.

The Formal Statement: The formal statement that must be true is

$$\langle \textit{loop-invariant} \rangle \ \& \ \langle \textit{exit-cond} \rangle \ \& \ \textit{code}_{\textit{post-loop}} \ \Rightarrow \ \langle \textit{post-cond} \rangle$$

Proof Technique:

- Assume that you have just broken out of the loop.
- You can assume that the loop invariant is true, because you have maintained that it is always true.
- You can also assume that the exit condition is true by the fact that the loop has exited.
- Execute the code $\textit{code}_{\textit{post-loop}}$ after the loop to give a few last touches towards solving the problem and to return the result.
- From these facts alone, you must be able to deduce that the problem has been solved correctly, namely, that the postcondition has been established.

11) Termination and Running Time: You must prove that the algorithm does not loop forever. This is done by proving that if the measure of progress meets some stated amount, then the exit condition has definitely been met. (If it exits earlier than this, all the better.) The number of iterations needed is then bounded by this stated amount of progress divided by the amount of progress made each iteration. The running time is estimated by adding up the time required for each of these iterations. For some applications, space bounds (i.e., the amount of memory used) may also be important. We discuss important concepts related to running time in Chapters 23–26: time and space complexity, the useful ideas of logarithms and exponentials, BigOh (O) and Theta (Θ) notation and several handy approximations.

12) Special Cases: When designing an algorithm, you do not want to worry about every possible type of input instance at the same time. Instead, first get the algorithm to work for one general type, then another and another. Though the next type of input instances may require separate code, start by tracing out what the algorithm that you have already designed would do given such an input. Often this algorithm will just happen to handle a lot of these cases automatically without requiring separate code. When adding code to handle a special case, be sure to check that the previously handled cases still are handled.

13) Coding and Implementation Details: Even after the basic algorithm is outlined, there can be many little details to consider. Many of these implementation details can be hidden in abstract data types (see Chapter 3). If a detail does not really make a difference to an algorithm, it is best to keep all possibilities open, giving extra flexibility to the implementer. For many details, it does not matter which choice you make, but bugs can be introduced if you are not consistent and clear as to what you

have chosen. This text does not focus on coding details. This does not mean that they are not important.

14) Formal Proof: Steps 1–11 are enough to ensure that your iterative algorithm works, that is, that it gives the correct answer on all specified inputs. Consider some instance which meets the preconditions. By step 8 we establish the loop invariant the first time the computation is at the top of the loop, and by step 7 we maintain it each iteration. Hence by way of induction, we know that the loop invariant is true every time the computation is at the top of the loop. (See the following discussion.) Hence, by step 5, the step taken in the main loop is always defined and executes without crashing until the loop exits. Moreover, by step 6 each such iteration makes progress of at least one. Hence, by step 11, the exit condition is eventually met. Step 10 then gives that the postcondition is achieved, so that the algorithm works in this instance.

Mathematical Induction: Induction is an extremely important mathematical technique for proving universal statements and is the cornerstone of iterative algorithms. Hence, we will consider it in more detail.

Induction Hypothesis: For each $n \geq 0$, let $S(n)$ be the statement "If the loop has not yet exited, then the loop invariant is true when you are at the top of the loop after going around n times."

Goal: The goal is to prove that $\forall n \geq 0,\ S(n)$, namely, "As long as the loop has not yet exited, the loop invariant is always true when you are at the top of the loop."

Proof Outline: Proof by induction on n.

Base Case: Proving $S(0)$ involves proving that the loop invariant is true when the algorithm first gets to the top of the loop. This is achieved by proving the statement $\langle \textit{pre-cond} \rangle$ & $\textit{code}_{\textit{pre-loop}} \Rightarrow \langle \textit{loop-invariant} \rangle$.

Induction Step: Proving $S(n-1) \Rightarrow S(n)$ involves proving that the loop invariant is maintained. This is achieved by proving the statement $\langle \textit{loop-invariant}' \rangle$ & not $\langle \textit{exit-cond} \rangle$ & $\textit{code}_{\textit{loop}} \Rightarrow \langle \textit{loop-invariant}'' \rangle$.

Conclusion: By way of induction, we can conclude that $\forall n \geq 0,\ S(n)$, i.e., that the loop invariant is always true when at the top of the loop.

The Process of Induction:

$S(0)$ is true (by base case)
$S(0) \Rightarrow S(1)$ (by induction step, $n = 1$)
hence, $S(1)$ is true
$S(1) \Rightarrow S(2)$ (by induction step, $n = 2$)
hence, $S(2)$ is true
$S(2) \Rightarrow S(3)$ (by induction step, $n = 3$)
hence, $S(3)$ is true ...

Other Proof Techniques: Other formal steps for proving correctness are described in Chapter 28.

Faith in the Method: Convince yourself that these steps are sufficient to define an algorithm so that you do not have reconvince yourself every time you need to design an algorithm.

1.4 Different Types of Iterative Algorithms

To help you design a measure of progress and a loop invariant for your algorithm, here are a few classic types, followed by examples of each type.

More of the Output: If the solution is a structure composed of many pieces (e.g., an array of integers, a set, or a path), a natural thing to try is to construct the solution one piece at a time.

Measure of Progress: The amount of the output constructed.

Loop Invariant: The output constructed so far is correct.

More of the Input: Suppose the input consists of n objects (e.g., an array of n integers or a graph with n nodes). It would be reasonable for the algorithm to read them in one at a time.

Measure of Progress: The amount of the input considered.

Loop Invariant: Pretending that this prefix of the input is the entire input, I have a complete solution.

Examples: After i iterations of the preceding find-max two-finger algorithm, the left finger points at the highest score within the prefix of the list seen so far. After i iterations of one version of insertion sort, the first i elements of the input are sorted. See Figure 1.2.

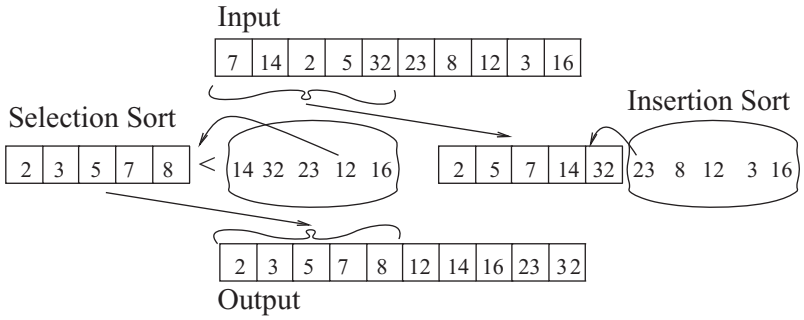


Figure 1.2: *The loop invariants for insertion sort and selection sort are demonstrated.*

Bad Loop Invariant: A common mistake is to give the loop invariant "I have handled and have a solution for each of the first i objects in the input." This is wrong because each object in the input does not need a separate solution; the input as a whole does. For example, in the find-max two-finger algorithm, one cannot know whether one element is the maximum by considering it in isolation from the other elements. An element is only the maximum in comparison with the other elements in the sublist.

Narrowing the Search Space: If you are searching for something, try narrowing the search space, maybe decreasing it by one or, even better, cutting it in half.

Measure of Progress: The size of the space in which you have narrowed the search.

Loop Invariant: If the thing being searched for is anywhere, then then it is in this narrowed sublist.

Example: Binary search.

Work Done: The measure of progress might also be some other more creative function of the work done so far.

Example: Bubble sort measures its progress by how many pairs of elements are out of order.

Case Analysis: Try the obvious thing. For which input instances does it work, and for which does it not work? Now you only need to find an algorithm that works for those later cases. An measure of progress might include which cases you have tried.

We will now give a simple examples of each of these. Though you likely know these algorithms already, use them to understand these different types of iterative algorithms and to review the required steps.

EXAMPLE 1.4.1 More of the Output—Selection Sort

1) Specifications: The goal is to rearrange a list of n values in nondecreasing order.

2) Basic Steps: We will repeatedly select the smallest unselected element.

3) Measure of Progress: The measure of progress is the number k of elements selected.

4) The Loop Invariant: The loop invariant states that the selected elements are the k smallest of the elements and that these have been sorted. The larger elements are in a set on the side.

5) Main Steps: The main step is to find the smallest element from among those in the remaining set of larger elements and to add this newly selected element to the end of the sorted list of elements.

6) Make Progress: Progress is made because k increases.

7) Maintain Loop Invariant: We must prove that $\langle$*loop-invariant*$'\rangle$ & *not* $\langle$*exit–cond*$\rangle$ & $code_{loop} \Rightarrow \langle$*loop-invariant*$''\rangle$. By the previous loop invariant, the newly selected element is at least the size of the previously selected elements. By the step, it is no bigger than the elements on the side. It follows that it must be the $k+1$st element in the list. Hence, moving this element from the set on the side to the end of the sorted list ensures that the selected elements in the new list are the $k+1$ smallest and are sorted.

8) Establishing the Loop Invariant: We must prove that $\langle$*pre-cond*$\rangle$ & $code_{pre\text{-}loop} \Rightarrow \langle$*loop-invariant*$\rangle$. Initially, $k = 0$ are sorted and all the elements are set aside.

9) Exit Condition: Stop when $k = n$.

10) Ending: We must prove $\langle$*loop-invariant*$\rangle$ & $\langle$*exit-cond*$\rangle$ & $code_{post\text{-}loop} \Rightarrow \langle$*post-cond*$\rangle$. By the exit condition, all the elements have been selected, and by the loop invariant these selected elements have been sorted.

11) Termination and Running Time: We have not considered how long it takes to find the next smallest element or to handle the data structures.

EXAMPLE 1.4.2 More of the Input—Insertion Sort

1) Specifications: Again the goal is to rearrange a list of n values in nondecreasing order.

2) Basic Steps: This time we will repeatedly insert some element where it belongs.

3) Measure of Progress: The measure of progress is the number k of elements inserted.

4) The Loop Invariant: The loop invariant states that the k inserted elements are sorted within a list and that, as before, the remaining elements are off to the side somewhere.

5) Main Steps: The main step is to take any of the elements that are off to the side and *insert* it into the sorted list where it belongs.

6) Make Progress: Progress is made because k increases.

7) Maintain Loop Invariant: $\langle$*loop-invariant*$'\rangle$ & *not* $\langle$*exit-cond*$\rangle$ & $code_{loop} \Rightarrow \langle$*loop-invariant*$''\rangle$. You know that the loop invariant has been maintained because the new element is inserted in the correct place in the previously sorted list.

8) Establishing the Loop Invariant: Initially, with $k = 1$, think of the first element in the array as a sorted list of length one.

9) Exit Condition: Stop when $k = n$.

10) Ending: $\langle$*loop-invariant*$\rangle$ & $\langle$*exit-cond*$\rangle$ & $code_{post\text{-}loop} \Rightarrow \langle$*post-cond*$\rangle$. By the exit condition, all the elements have been inserted, and by the loop invariant, these inserted elements have been sorted.

11) Termination and Running Time: We have not considered how long it takes to insert the element or to handle the data structures.

Example 1.4.3 Narrowing the Search Space—Binary Search

1) Specifications: An input instance consists of a sorted list $A[1..n]$ of elements and a key to be searched for. Elements may be repeated. If the key is in the list, then the output consists of an index i such that $A[i] = key$. If the key is not in the list, then the output reports this.

2) Basic Steps: Continue to cut the search space in which the key might be in half.

4) The Loop Invariant: The algorithm maintains a sublist $A[i..j]$ such that if the key is contained in the original list $A[1..n]$, then it is contained in this narrowed sublist. (If the element is repeated, then it might also be outside this sublist.)

3) Measure of Progress: The measure of progress is the number of elements in our sublist, namely $j - i + 1$.

5) Main Steps: Each iteration compares the key with the element at the center of the sublist. This determines which half of the sublist the key is not in and hence which half to keep. More formally, let *mid* index the element in the middle of our current sublist $A[i..j]$. If $key \leq A[mid]$, then the sublist is narrowed to $A[i..mid]$. Otherwise, it is narrowed to $A[mid + 1..j]$.

6) Make Progress: The size of the sublist decreases by a factor of two.

7) Maintain Loop Invariant: $\langle$*loop-invariant*$'\rangle$ & *not* $\langle$*exit-cond*$\rangle$ & $code_{loop} \Rightarrow \langle$*loop-invariant*$''\rangle$. The previous loop invariant gives that the search has been narrowed down to the sublist $A[i..j]$. If $key > A[mid]$, then because the list is sorted, we know that key is not in $A[1..mid]$ and hence these elements can be thrown away, narrowing the search to $A[mid + 1..j]$. Similarly if $key < A[mid]$. If $key = A[mid]$, then we could report that the key has been found. However, the loop invariant is also maintained by narrowing the search down to $A[i..mid]$.

8) Establishing the Loop Invariant: $\langle$*pre-cond*$\rangle$ & $code_{pre\text{-}loop} \Rightarrow \langle loop{-}invariant\rangle$. Initially, you obtain the loop invariant by considering the entire list as the sublist. It trivially follows that if the key is in the entire list, then it is also in this sublist.

9) Exit Condition: We exit when the sublist contains one (or zero) elements.

10) Ending: $\langle$*loop-invariant*$\rangle$ & $\langle$*exit-cond*$\rangle$ & $code_{post\text{-}loop} \Rightarrow \langle$*post-cond*$\rangle$. By the exit condition, our sublist contains at most one element, and by the loop invariant, if the

key is contained in the original list, then the key is contained in this sublist, i.e., must be this one element. Hence, the final code tests to see if this one element is the key. If it is, then its index is returned. If it is not, then the algorithm reports that the key is not in the list.

11) Termination and Running Time: The sizes of the sublists are approximately $n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \frac{n}{16}, \ldots, 8, 4, 2, 1$. Hence, only $\Theta(\log n)$ splits are needed. Each split takes $O(1)$ time. Hence, the total time is $\Theta(\log n)$.

12) Special Cases: A special case to consider is when the key is not contained in the original list $A[1..n]$. Note that the loop invariant carefully takes this case into account. The algorithm will narrow the sublist down to one (or zero) elements. The counter positive of the loop invariant then gives that if the key is not contained in this narrowed sublist, then the key is not contained in the original list $A[1..n]$.

13) Coding and Implementation Details: In addition to testing whether $key \leq A[mid]$, each iteration could test to see if $A[mid]$ is the key. Though finding the key in this way would allow you to stop early, extensive testing shows that this extra comparison slows down the computation.

EXAMPLE 1.4.4 Work Done—Bubble Sort

1) Specifications: The goal is to rearrange a list of n values in nondecreasing order.

2) Basic Steps: Swap elements that are out of order.

3) Measure of Progress: An *involution* is a pair of elements that are out of order, i.e., a pair i, j where $1 \leq i < j \leq n$, $A[i] > A[j]$. Our measure of progress will be the number of involutions in our current ordering of the elements. For example, in [1, 2, 5, 4, 3, 6], there are three involutions.

4) The Loop Invariant: The loop invariant is relatively weak, stating only that we have a permutation of the original input elements.

5) Main Steps: The main step is to find two adjacent elements that are out of order and to swap them.

6) Make Progress: Such a step decreases the number of involutions by one.

7) Maintain Loop Invariant: $\langle loop\text{-}invariant' \rangle$ & *not* $\langle exit\text{-}cond \rangle$ & $code_{loop} \Rightarrow \langle loop\text{-}invariant'' \rangle$. By the previous loop invariant we had a permutation of the elements. Swapping a pair of elements does not change this.

8) Establishing the Loop Invariant: $\langle pre\text{-}cond \rangle$ & $code_{pre\text{-}loop} \Rightarrow \langle loop-invariant \rangle$. Initially, we have a permutation of the elements.

9) Exit Condition: Stop when we have a sorted list of elements.

10) Ending: $\langle$*loop-invariant*$\rangle$ & $\langle$*exit-cond*$\rangle$ & $code_{post\text{-}loop} \Rightarrow \langle$*post-cond*$\rangle$. By the loop invariant, we have a permutation of the original elements, and by the exit condition these are sorted.

11) Termination and Running Time: Initially, the measure of progress cannot be higher than $n(n-1)/2$ because this is the number of pairs of elements there are. In each iteration, this measure decreases by one. Hence, after at most $n(n-1)/2$ iterations, the measure of progress has decreased to zero. At this point the list has been sorted and the exit condition has been met. We have not considered how long it takes to find two adjacent elements that are out of order.

EXERCISE 1.4.1 *(See solution in Part Five.) Give the implementation details and the running times for selection sort.*

EXERCISE 1.4.2 *(See solution in Part Five.) Give the implementation details and the running times for insertion sort. Does using binary search to find the smallest element or to find where to insert help? Does it make a difference whether the elements are stored in an array or in a linked list?*

EXERCISE 1.4.3 *(See solution in Part Five.) Give the implementation details and the running times for bubble sort: Use another loop invariant to prove that the total number of comparisons needed is* $O(n^2)$.

1.5 Typical Errors

In a study, a group of experienced programmers was asked to code binary search. Easy, yes? 80% got it wrong! My guess is that if they had used loop invariants, they all would have got it correct.

Be Clear: The code specifies the current subinterval $A[i..j]$ with two integers i and j. Clearly document whether the sublist includes the end points i and j or not. It does not matter which, but you must be consistent. Confusion in details like this is the cause of many bugs.

Math Details: Small math operations like computing the index of the middle element of the subinterval $A(i..j)$ are prone to bugs. Check for yourself that the answer is $mid = \lfloor \frac{i+j}{2} \rfloor$.

6) Make Progress: Be sure that each iteration progress is made in every special case. For example, in binary search, when the current sublist has even length, it is reasonable (as done above) to let *mid* be the element just to the left of center. It is also reasonable to include the middle element in the right half of the sublist. However,

together these cause a bug. Given the sublist $A[i..j] = A[3, 4]$, the middle will be the element indexed with 3, and the right sublist will be still be $A[mid..j] = A[3, 4]$. If this sublist is kept, no progress will be made, and the algorithm will loop forever.

7) Maintain Loop Invariant: Be sure that the loop invariant is maintained in every special case. For example, in binary search, it is reasonable to test whether $key < A[mid]$ or $key \geq A[mid]$. It is also reasonable for it to cut the sublist $A[i..j]$ into $A[i..mid]$ and $A[mid + 1..j]$. However, together these cause a bug. When *key* and $A[mid]$ are equal, the test $key < A[mid]$ will fail, causing the algorithm to think the key is bigger and to keep the right half $A[mid + 1..j]$. However, this skips over the key.

Simple Loop: Code like "$i = 1$; while($i \leq n$) $A[i] = 0$; $i = i + 1$; end while" is surprisingly prone to the error of being off by one. The loop invariant "When at the top of the loop, i indexes the next element to handle" helps a lot.

EXERCISE 1.5.1 *(See solution in Part Five.) You are now the professor. Which of the steps to develop an iterative algorithm did the student fail to do correctly in the following code? How? How would you fix it?*

algorithm $Eg(I)$

⟨ ***pre-cond***⟩***:*** *I is an integer.*
⟨ ***post-cond***⟩***:*** *Outputs* $\sum_{j=1}^{I} j$.
begin
    $s = 0$
    $i = 1$
    while($i \leq I$ *)*
        ⟨***loop-invariant***⟩***:*** *Each iteration adds the next term giving that* $s = \sum_{j=1}^{i} j$.
        $s = s + i$
        $i = i + 1$
    end loop
return(s)
end algorithm

1.6 Exercises

EXERCISE 1.6.1 *You are in the middle of a lake of radius 1. You can swim at a speed of 1 and can run infinitely fast. There is a smart monster on the shore who can't go in the water but can run at a speed of 4. Your goal is to swim to shore, arriving at a spot where the monster is not, and then run away. If you swim directly to shore, it will take you 1*

time unit. In this time, the monster will run the distance $\Pi < 4$ *around to where you land and eat you. Your better strategy is to maintain the most obvious loop invariant while increasing the most obvious measure of progress for as long as possible and then swim for it. Describe how this works.*

EXERCISE 1.6.2 *Given an undirected graph* G *such that each node has at most* $d + 1$ *neighbors, color each node with one of* $d + 1$ *colors so that for each edge the two nodes have different colors. Hint: Don't think too hard. Just color the nodes. What loop invariant do you need?*