2  How to Describe and Evaluate Computer Algorithms

In the previous chapter, you got a taste of how we couch the running time of a computer algorithm: by focusing on the running time as a function of the input size, and specifically on the order of growth of the running time. In this chapter, we’ll back up a bit and see how we describe computer algorithms. Then we’ll see the notations that we use to characterize the running times of algorithms. We’ll wrap up this chapter by examining some techniques that we use to design and understand algorithms.

How to describe computer algorithms

We always have the option of describing a computer algorithm as a runnable program in a commonly used programming language, such as Java, C, C++, Python, or Fortran. Indeed, several algorithms textbooks do just that. The problem with using real programming languages to specify algorithms is that you can get bogged down in the details of the language, obscuring the ideas behind the algorithms. Another approach, which we took in Introduction to Algorithms, uses “pseudocode,” which looks like a mashup of various programming languages with English mixed in. If you’ve ever used a real programming language, you can figure out pseudocode easily. But if you have not ever programmed, then pseudocode might seem a bit mysterious.

The approach I’m taking in this book is that I’m not trying to describe algorithms to software or hardware, but to “wetware”: the gray matter between your ears. I am also going to assume that you have never written a computer program, and so I won’t express algorithms using any real programming language or even pseudocode. Instead, I’ll describe them in English, using analogies to real-world scenarios whenever I can. In order to indicate what happens when (what we call “flow of control” in programming), I’ll use lists and lists within lists. If you want to implement an algorithm in a real programming language, I’ll give you credit for being able to translate my description into runnable code.
Although I will try to keep descriptions as nontechnical as possible, this book is about algorithms for computers, and so I will have to use computing terminology. For example, computer programs contain procedures (also known as functions or methods in real programming languages), which specify how to do something. In order to actually get the procedure to do what it’s supposed to do, we call it. When we call a procedure, we supply it with inputs (usually at least one, but some procedures take no inputs). We specify the inputs as parameters within parentheses after the name of the procedure. For example, to compute the square root of a number, we might define a procedure SQUARE-ROOT(x); here, the input to the procedure is referred to by the parameter x. The call of a procedure may or may not produce output, depending on how we specified the procedure. If the procedure produces output, we usually consider the output to be something passed back to its caller. In computing parlance we say that the procedure returns a value.

Many programs and algorithms work with arrays of data. An array aggregates data of the same type into one entity. You can think of an array as being like a table, where given the index of an entry, we can talk about the array element at that index. For example, here is a table of the first five U.S. presidents:

<table>
<thead>
<tr>
<th>Index</th>
<th>President</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>George Washington</td>
</tr>
<tr>
<td>2</td>
<td>John Adams</td>
</tr>
<tr>
<td>3</td>
<td>Thomas Jefferson</td>
</tr>
<tr>
<td>4</td>
<td>James Madison</td>
</tr>
<tr>
<td>5</td>
<td>James Monroe</td>
</tr>
</tbody>
</table>

For example, the element at index 4 in this table is James Madison. We think of this table not as five separate entities, but as one table with five entries. An array is similar. The indices into an array are consecutive numbers that can start anywhere, but we will usually start them at 1. If you program in Java, C, or C++, you are used to arrays that start at 0. Starting arrays at 0 is nice for computers, but for wetware it’s often more intuitive to start at 1.
Arrays in computers have one other important characteristic: it takes equally long to access any element of an array. Once you give the computer an index \( i \) into an array, it can access the \( i \)th element as quickly as it can access the first element, regardless of the value of \( i \).

Let's see our first algorithm: searching an array for a particular value. That is, we are given an array, and we want to know which entry in the array, if any, holds a given value. To see how we can search an array, let's think of the array as a long bookshelf full of books, and suppose that you want to know where on the shelf you can find a book by Jonathan Swift. Now, the books on the shelf might be organized in some way, perhaps alphabetically by author, alphabetically by title, or, in a library, by call number. Or perhaps the bookshelf is like my bookshelf at home, where I have not organized my books in any particular way.

If you couldn't assume that the books were organized on the shelf, how would you find a book by Jonathan Swift? Here's the algorithm I would follow. I would start at the left end of the shelf and look at the leftmost book. If it's by Swift, I have located the book. Otherwise, I would look at the next book to the right, and if that book is by Swift, I have located the book. If not, I would keep going to the right, examining book after book, until either I find a book by Swift or I run off the right-hand end of the shelf, in which case I can conclude that the bookshelf does not contain any book by Jonathan Swift. (In Chapter 3, we'll see how to search for a book when the books are organized on the shelf.)

Here is how we can describe this searching problem in terms of computing. Let's think of the books on the bookshelf as an array of books. The leftmost book is in position 1, the next book to its right is in position 2, and so on. If we have \( n \) books on the shelf, then the rightmost book is in position \( n \). We want to find the position number on the shelf of any book by Jonathan Swift.

As a general computing problem, we are given an array \( A \) (the entire shelf full of books to search through) of \( n \) elements (the individual books), and we want to find whether a value \( x \) (a book by Jonathan Swift) is present in the array \( A \). If it is, then we want to determine an index \( i \) such that \( A[i] = x \) (the \( i \)th position on the shelf contains a book by Jonathan Swift). We also need some way to indicate that array \( A \) does not contain \( x \) (the bookshelf contains no books by Jonathan Swift). We do not assume that \( x \) appears at most once in the array (perhaps you have multiple copies of some book), and so if \( x \) is present in array \( A \), it may appear multiple times. All we want from a searching algorithm
is any index at which we'll find \( x \) in the array. We'll assume that the indices of this array start at 1, so that its elements are \( A[1] \) through \( A[n] \).

If we search for a book by Jonathan Swift by starting at the left end of the shelf, checking book by book as we move to the right, we call that technique linear search. In terms of an array in a computer, we start at the beginning of the array, examine each array element in turn (\( A[1] \), then \( A[2] \), then \( A[3] \), and so on, up through \( A[n] \)) and record where we find \( x \), if we find it at all.

The following procedure, \textit{LINEAR-SEARCH}, takes three parameters, which we separate by commas in the specification.

\begin{verbatim}
Procedure LINEAR-SEARCH(A, n, x)
Inputs:
• \( A \): an array.
• \( n \): the number of elements in \( A \) to search through.
• \( x \): the value being searched for.
Output: Either an index \( i \) for which \( A[i] = x \), or the special value NOT-FOUND, which could be any invalid index into the array, such as 0 or any negative integer.
1. Set answer to NOT-FOUND.
2. For each index \( i \), going from 1 to \( n \), in order:
   A. If \( A[i] = x \), then set answer to the value of \( i \).
3. Return the value of answer as the output.
\end{verbatim}

In addition to the parameters \( A, n, \) and \( x, \) the \textit{LINEAR-SEARCH} procedure uses a \textit{variable} named \textit{answer}. The procedure \textit{assigns} an initial value of \textit{NOT-FOUND} to \textit{answer} in step 1. Step 2 checks each array entry \( A[1] \) through \( A[n] \) to see if the entry contains the value \( x \). Whenever entry \( A[i] \) equals \( x \), step 2A assigns the current value of \( i \) to \textit{answer}. If \( x \) appears in the array, then the output value returned in step 3 is the last index in which \( x \) appeared. If \( x \) does not appear in the array, then the equality test in step 2A never evaluates to true, and the output value returned is \textit{NOT-FOUND}, as assigned to \textit{answer} back in step 1.

Before we continue discussing linear search, a word about how to specify repeated actions, such as in step 2. It is quite common in algorithms to perform some action for a variable taking values in some range. When we perform repeated actions, we call that a \textit{loop}, and we call each time through the loop an \textit{iteration} of the loop. For the loop of
step 2, I wrote "For each index $i$, going from 1 to $n$, in order." Instead, from now on, I'll write "For $i = 1$ to $n$," which is shorter, yet conveys the same structure. Notice that when I write a loop in this way, we have to give the loop variable (here, $i$) an initial value (here, 1), and in each iteration of the loop, we have to test the current value of the loop variable against a limit (here, $n$). If the current value of the loop variable is less than or equal to the limit, then we do everything in the loop's body (here, step 2A). After an iteration executes the loop body, we increment the loop variable—adding 1 to it—and go back and compare the loop variable, now with its new value, with the limit. We repeatedly test the loop variable against the limit, execute the loop body, and increment the loop variable, until the loop variable exceeds the limit. Execution then continues from the step immediately following the loop body (here, step 3). A loop of the form "For $i = 1$ to $n$" performs $n$ iterations and $n + 1$ tests against the limit (because the loop variable exceeds the limit in the $(n + 1)$st test).

I hope that you find it obvious that the LINEAR-SEARCH procedure always returns a correct answer. You might have noticed, however, that this procedure is inefficient: it continues to search the array even after it has found an index $i$ for which $A[i] = x$. Normally, you wouldn't continue searching for a book once you have found it on your bookshelf, would you? Instead, we can design our linear search procedure to stop searching once it finds the value $x$ in the array. We assume that when we say to return a value, the procedure immediately returns the value to its caller, which then takes control.

**Procedure** BETTER-LINEAR-SEARCH($A, n, x$)

**Inputs and Output:** Same as LINEAR-SEARCH.

1. For $i = 1$ to $n$:
   
   A. If $A[i] = x$, then return the value of $i$ as the output.

2. Return NOT-FOUND as the output.

Believe it or not, we can make linear search even more efficient. Observe that each time through the loop of step 1, the BETTER-LINEAR-SEARCH procedure makes two tests: a test in step 1 to determine whether $i \leq n$ (and if so, perform another iteration of the loop) and the equality test in step 1A. In terms of searching a bookshelf, these tests correspond to you having to check two things for each book: have you
gone past the end of the shelf and, if not, is the next book by Jonathan Swift? Of course, you don’t incur much of a penalty for going past the end of the shelf (unless you keep your face really close to the books as you examine them, there’s a wall at the end of the shelf, and you smack your face into the wall), but in a computer program it’s usually very bad to try to access array elements past the end of the array. Your program could crash, or it could corrupt data.

You can make it so that you have to perform only one check for every book you examine. What if you knew for sure that your bookshelf contained a book by Jonathan Swift? Then you’d be assured of finding it, and so you’d never have to check for running off the end of the shelf. You could just check each book in turn to see whether it’s by Swift.

But perhaps you lent out all your books by Jonathan Swift, or maybe you thought you had books by him but you never did, so you might not be sure that your bookshelf contains any books by him. Here’s what you can do. Take an empty box the size of a book and write on its narrow side (where the spine of a book would be) “Gulliver’s Travels by Jonathan Swift.” Replace the rightmost book with this box. Then, as you search from left to right along the bookshelf, you need to check only whether you’re looking at something that is by Swift; you can forget about having to check whether you’re going past the end of the bookshelf because you know that you’ll find something by Swift. The only question is whether you really found a book by Swift, or did you find the empty box that you had labeled as though it were by him? If you found the empty box, then you didn’t really have a book by Swift. That’s easy to check, however, and you need to do that only once, at the end of your search, rather than once for every book on the shelf.

There’s one more detail you have to be aware of: what if the only book by Jonathan Swift that you had on your bookshelf was the rightmost book? If you replace it by the empty box, your search will terminate at the empty box, and you might conclude that you didn’t have the book. So you have to perform one more check for that possibility, but it’s just one check, rather than one check for every book on the shelf.

In terms of a computer algorithm, we’ll put the value $x$ that we’re searching for into the last position, $A[n]$, after saving the contents of $A[n]$ into another variable. Once we find $x$, we test to see whether we really found it. We call the value that we put into the array a sentinel, but you can think of it as the empty box.
**Procedure SENTINEL-LINEAR-SEARCH**($A, n, x$)

*Inputs and Output:* Same as LINEAR-SEARCH.

2. Set $i$ to 1.
3. While $A[i] \neq x$, do the following:
   A. Increment $i$.
5. If $i < n$ or $A[n] = x$, then return the value of $i$ as the output.
6. Otherwise, return NOT-FOUND as the output.

Step 3 is a loop, but not one that counts through some loop variable. Instead, the loop iterates as long as a condition holds; here, the condition is that $A[i] \neq x$. The way to interpret such a loop is to perform the test (here, $A[i] \neq x$), and if the test is true, then do everything in the loop's body (here, step 3A, which increments $i$). Then go back and perform the test, and if the test is true, execute the body. Keep going, performing the test then executing the body, until the test comes up false. Then continue from the next step after the loop body (here, continue from step 4).

The SENTINEL-LINEAR-SEARCH procedure is a bit more complicated than the first two linear search procedures. Because it places $x$ into $A[n]$ in step 1, we are guaranteed that $A[i]$ will equal $x$ for some test in step 3. Once that happens, we drop out of the step-3 loop, and the index $i$ won't change thereafter. Before we do anything else, step 4 restores the original value in $A[n]$. (My mother taught me to put things back when I was done with them.) Then we have to determine whether we really found $x$ in the array. Because we put $x$ into the last element, $A[n]$, we know that if we found $x$ in $A[i]$ where $i < n$, then we really did find $x$ and we want to return the index $i$. What if we found $x$ in $A[n]$? That means we didn't find $x$ before $A[n]$, and so we need to determine whether $A[n]$ equals $x$. If it does, then we want to return the index $n$, which equals $i$ at this point, but if it does not, we want to return NOT-FOUND. Step 5 does these tests and returns the correct index if $x$ was originally in the array. If $x$ was found only because step 1 put it into the array, then step 6 returns NOT-FOUND. Although SENTINEL-LINEAR-SEARCH has to perform two tests after its loop terminates, it performs only one test in each loop iteration, thereby making it more efficient than either LINEAR-SEARCH or BETTER-LINEAR-SEARCH.
How to characterize running times

Let's return to the LINEAR-SEARCH procedure from page 13 and understand its running time. Recall that we want to characterize the running time as a function of the input size. Here, our input is an array $A$ of $n$ elements, along with the number $n$ and the value $x$ that we're searching for. The sizes of $n$ and $x$ are insignificant as the array gets large—after all, $n$ is just a single integer and $x$ is only as large as one of the $n$ array elements—and so we'll say that the input size is $n$, the number of elements in $A$.

We have to make some simple assumptions about how long things take. We will assume that each individual operation—whether it's an arithmetic operation (such as addition, subtraction, multiplication, or division), a comparison, assigning to a variable, indexing into an array, or calling or returning from a procedure—takes some fixed amount of time that is independent of the input size. The time might vary from operation to operation, so that division might take longer than addition, but when a step comprises just simple operations, each individual execution of that step takes some constant amount of time. Because the operations executed differ from step to step, and because of the extrinsic factors listed back on page 4, the time to execute a step might vary from step to step. Let's say that each execution of step $i$ takes $t_i$ time, where $t_i$ is some constant that does not depend on $n$.

Of course, we have to take into account that some steps execute multiple times. Steps 1 and 3 execute just once, but what about step 2? We have to test $i$ against $n$ a total of $n + 1$ times: $n$ times in which $i \leq n$, and once when $i$ equals $n + 1$ so that we drop out of the loop. Step 2A executes exactly $n$ times, once for each value of $i$ from 1 to $n$. We don't know in advance how many times we set $answer$ to the value of $i$; it could be anywhere from 0 times (if $x$ is not present in the array) to $n$ times (if every value in the array equals $x$). If we're going to be precise in our accounting—and we won't normally be this precise—we need to

---

2If you know a bit about actual computer architecture, you might know that the time to access a given variable or array element is not necessarily fixed, for it could depend on whether the variable or array element is in the cache, in main memory, or out on disk in a virtual-memory system. Some sophisticated models of computers take these issues into account, but it's often good enough to just assume that all variables and array entries are in main memory and that they all take the same amount of time to access.
recognize that step 2 does two different things that execute a different number of times: the test of \( i \) against \( n \) happens \( n + 1 \) times, but incrementing \( i \) happens only \( n \) times. Let's separate the time for line 2 into \( t'_2 \) for the test and \( t''_2 \) for incrementing. Similarly, we'll separate the time for step 2A into \( t'_{2A} \) for testing whether \( A[i] = x \) and \( t''_{2A} \) for setting answer to \( i \). Therefore, the running time of LINEAR-SEARCH is somewhere between
\[
t_1 + t'_2 \cdot (n + 1) + t''_2 \cdot n + t'_{2A} \cdot n + t''_{2A} \cdot 0 + t_3
\]
and
\[
t_1 + t'_2 \cdot (n + 1) + t''_2 \cdot n + t'_{2A} \cdot n + t''_{2A} \cdot n + t_3.
\]
Now we rewrite these bounds, collecting terms that multiply by \( n \) together, and collecting the rest of the terms, and we see that the running time is somewhere between the lower bound
\[
(t'_2 + t''_2 + t'_{2A}) \cdot n + (t_1 + t'_2 + t_3)
\]
and the upper bound
\[
(t'_2 + t''_2 + t'_{2A} + t''_{2A}) \cdot n + (t_1 + t'_2 + t_3).
\]
Notice that both of these bounds are of the form \( c \cdot n + d \), where \( c \) and \( d \) are constants that do not depend on \( n \). That is, they are both linear functions of \( n \). The running time of LINEAR-SEARCH is bounded from below by a linear function of \( n \), and it is bounded from above by a linear function of \( n \).

We use a special notation to indicate that a running time is bounded from above by some linear function of \( n \) and from below by some (possibly different) linear function of \( n \). We write that the running time is \( \Theta(n) \). That's the Greek letter theta, and we say "theta of \( n \)" or just "theta \( n \)." As promised in Chapter 1, this notation discards the low-order term \( t_1 + t'_2 + t_3 \) and the coefficients of \( n \) \( (t'_2 + t''_2 + t'_{2A} \) for the lower bound and \( t'_2 + t''_2 + t'_{2A} + t''_{2A} \) for the upper bound). Although we lose precision by characterizing the running time as \( \Theta(n) \), we gain the advantages of highlighting the order of growth of the running time and suppressing tedious detail.

This \( \Theta \)-notation applies to functions in general, not just those that describe running times of algorithms, and it applies to functions other than linear ones. The idea is that if we have two functions, \( f(n) \) and \( g(n) \), we say that \( f(n) \) is \( \Theta(g(n)) \) if \( f(n) \) is within a constant factor of \( g(n) \)
for sufficiently large \( n \). So we can say that the running time of \textsc{linear-search} is within a constant factor of \( n \) once \( n \) gets large enough.

There's an intimidating technical definition of \( \Theta \)-notation, but fortunately we rarely have to resort to it in order to use \( \Theta \)-notation. We simply focus on the dominant term, dropping low-order terms and constant factors. For example, the function \( n^2/4 + 100n + 50 \) is \( \Theta(n^2) \); here we drop the low-order terms \( 100n \) and \( 50 \), and we drop the constant factor \( 1/4 \). Although the low-order terms will dominate \( n^2/4 \) for small values of \( n \), once \( n \) goes above 400, the \( n^2/4 \) term exceeds \( 100n + 50 \). When \( n = 1000 \), the dominant term \( n^2/4 \) equals 250,000, while the low-order terms \( 100n + 50 \) amount to only 100,050; for \( n = 2000 \) the difference becomes 1,000,000 vs. 200,050. In the world of algorithms, we abuse notation a little bit and write \( f(n) = \Theta(g(n)) \), so that we can write \( n^2/4 + 100n + 50 = \Theta(n^2) \).

Now let's look at the running time of \textsc{better-linear-search} from page 14. This one is a little trickier than \textsc{linear-search} because we don't know in advance how many times the loop will iterate. If \( A[1] \) equals \( x \), then it will iterate just once. If \( x \) is not present in the array, then the loop will iterate all \( n \) times, which is the maximum possible. Each loop iteration takes some constant amount of time, and so we can say that in the worst case, \textsc{better-linear-search} takes \( \Theta(n) \) time to search an array of \( n \) elements. Why "worst case"? Because we want algorithms to have low running times, the worst case occurs when an algorithm takes the maximum time over any possible input.

In the best case, when \( A[1] \) equals \( x \), \textsc{better-linear-search} takes just a constant amount of time: it sets \( i \) to 1, checks that \( i \leq n \), the test \( A[i] = x \) comes up true, and the procedure returns the value of \( i \), which is 1. This amount of time does not depend on \( n \). We write that the best-case running time of \textsc{better-linear-search} is \( \Theta(1) \), because in the best case, its running time is within a constant factor of 1. In other words, the best-case running time is a constant that does not depend on \( n \).

So we see that we cannot use \( \Theta \)-notation for a blanket statement that covers all cases of the running time of \textsc{better-linear-search}. We cannot say that the running time is always \( \Theta(n) \), because in the best case it's \( \Theta(1) \). And we cannot say that the running time is always \( \Theta(1) \), because in the worst case it's \( \Theta(n) \). We can say that a linear function of \( n \) is an upper bound in all cases, however, and we have a notation for that: \( O(n) \). When we speak this notation, we say "big-oh of \( n \)" or just
"oh of \( n \)." A function \( f(n) \) is \( O(g(n)) \) if, once \( n \) becomes sufficiently large, \( f(n) \) is bounded from above by some constant times \( g(n) \). Again, we abuse notation a little and write \( f(n) = O(g(n)) \). For BETTER-LINEAR-SEARCH, we can make the blanket statement that its running time in all cases is \( O(n) \); although the running time might be better than a linear function of \( n \), it’s never worse.

We use \( O \)-notation to indicate that a running time is never worse than a constant times some function of \( n \), but how about indicating that a running time is never better than a constant times some function of \( n \)? That’s a lower bound, and we use \( \Omega \)-notation, which mirrors \( O \)-notation: a function \( f(n) \) is \( \Omega(g(n)) \) if, once \( n \) becomes sufficiently large, \( f(n) \) is bounded from below by some constant times \( g(n) \). We say that "\( f(n) \) is big-omega of \( g(n) \)" or just "\( f(n) \) is omega of \( g(n) \)," and we can write \( f(n) = \Omega(g(n)) \). Since \( O \)-notation gives an upper bound, \( \Omega \)-notation gives a lower bound, and \( \Theta \)-notation gives both upper and lower bounds, we can conclude that a function \( f(n) \) is \( \Theta(g(n)) \) if and only if \( f(n) \) is both \( O(g(n)) \) and \( \Omega(g(n)) \).

We can make a blanket statement about a lower bound for the running time of BETTER-LINEAR-SEARCH: in all cases it’s \( \Omega(1) \). Of course, that’s a pathetically weak statement, since we’d expect any algorithm on any input to take at least constant time. We won’t use \( \Omega \)-notation much, but it will occasionally come in handy.

The catch-all term for \( \Theta \)-notation, \( O \)-notation, and \( \Omega \)-notation is asymptotic notation. That’s because these notations capture the growth of a function as its argument asymptotically approaches infinity. All of these asymptotic notations give us the luxury of dropping low-order terms and constant factors so that we can ignore tedious details and focus on what’s important: how the function grows with \( n \).

Now let’s turn to SENTINEL-LINEAR-SEARCH from page 16. Just like BETTER-LINEAR-SEARCH, each iteration of its loop takes a constant amount of time, and there may be anywhere from 1 to \( n \) iterations. The key difference between SENTINEL-LINEAR-SEARCH and BETTER-LINEAR-SEARCH is that the time per iteration of SENTINEL-LINEAR-SEARCH is less than the time per iteration of BETTER-LINEAR-SEARCH. Both take a linear amount of time in the worst case, but the constant factor for SENTINEL-LINEAR-SEARCH is better. Although we’d expect SENTINEL-LINEAR-SEARCH to be faster in practice, it would be by only a constant factor. When we express the running times of BETTER-LINEAR-SEARCH and SENTINEL-LINEAR-SEARCH
using asymptotic notation, they are equivalent: $\Theta(n)$ in the worst case, $\Theta(1)$ in the best case, and $O(n)$ in all cases.

**Loop invariants**

For our three flavors of linear search, it was easy to see that each one gives a correct answer. Sometimes it’s a bit harder. There’s a wide range of techniques, more than I can cover here.

One common method of showing correctness uses a *loop invariant*: an assertion that we demonstrate to be true each time we start a loop iteration. For a loop invariant to help us argue correctness, we have to show three things about it:

**Initialization:** It is true before the first iteration of the loop.

**Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.

**Termination:** The loop terminates, and when it does, the loop invariant, along with the reason that the loop terminated, gives us a useful property.

As an example, here’s a loop invariant for BETTER-LINEAR-SEARCH:

At the start of each iteration of step 1, if $x$ is present in the array $A$, then it is present in the *subarray* (a contiguous portion of an array) from $A[i]$ through $A[n]$.

We don’t even need this loop invariant to show that if the procedure returns an index other than NOT-FOUND, then the index returned is correct: the only way that the procedure can return an index $i$ in step 1A is because $x$ equals $A[i]$. Instead, we will use this loop invariant to show that if the procedure returns NOT-FOUND in step 2, then $x$ is not anywhere in the array:

**Initialization:** Initially, $i = 1$ so that the subarray in the loop invariant is $A[1]$ through $A[n]$, which is the entire array.

**Maintenance:** Assume that at the start of an iteration for a value of $i$, if $x$ is present in the array $A$, then it is present in the subarray from $A[i]$ through $A[n]$. If we get through this iteration without returning, we know that $A[i] \neq x$, and therefore we can safely say that if $x$ is present in the array $A$, then it is present in the subarray from $A[i + 1]$ through $A[n]$. Because $i$ is incremented before the next iteration, the loop invariant will hold before the next iteration.
**Termination:** This loop must terminate, either because the procedure returns in step 1A or because \( i > n \). We have already handled the case where the loop terminates because the procedure returns in step 1A.

To handle the case where the loop terminates because \( i > n \), we rely on the contrapositive of the loop invariant. The contrapositive of the statement "if \( A \) then \( B \)" is "if not \( B \) then not \( A \)." The contrapositive of a statement is true if and only if the statement is true. The contrapositive of the loop invariant is "if \( x \) is not present in the subarray from \( A[i] \) through \( A[n] \), then it is not present in the array \( A \)."

Now, when \( i > n \), the subarray from \( A[i] \) through \( A[n] \) is empty, and so this subarray cannot hold \( x \). By the contrapositive of the loop invariant, therefore, \( x \) is not present anywhere in the array \( A \), and so it is appropriate to return NOT-FOUND in step 2.

Wow, that's a lot of reasoning for what's really just a simple loop! Do we have to go through all that every time we write a loop? I don't, but there are a few computer scientists who insist on such rigorous reasoning for every single loop. When I'm writing real code, I find that most of the time that I write a loop, I have a loop invariant somewhere in the back of my mind. It might be so far back in my mind that I don't even realize that I have it, but I could state it if I had to. Although most of us would agree that a loop invariant is overkill for understanding the simple loop in BETTER-LINEAR-SEARCH, loop invariants can be quite handy when we want to understand why more complex loops do the right thing.

**Recursion**

With the technique of recursion, we solve a problem by solving smaller instances of the same problem. Here's my favorite canonical example of recursion: computing \( n! \) ("\( n \)-factorial"), which is defined for nonnegative values of \( n \) as \( n! = 1 \) if \( n = 0 \), and

\[
n! = n \cdot (n - 1) \cdot (n - 2) \cdot (n - 3) \cdots 3 \cdot 2 \cdot 1
\]

if \( n \geq 1 \). For example, \( 5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120 \). Observe that

\[
(n - 1)! = (n - 1) \cdot (n - 2) \cdot (n - 3) \cdots 3 \cdot 2 \cdot 1,
\]

and so
Chapter 2: How to Describe and Evaluate Computer Algorithms

\[ n! = n \cdot (n-1)! \]

for \( n \geq 1 \). We have defined \( n! \) in terms of a "smaller" problem, namely \((n-1)!\). We could write a recursive procedure to compute \( n! \) as follows:

**Procedure FACTORIAL\( (n) \)**

*Input:* An integer \( n \geq 0 \).

*Output:* The value of \( n! \).

1. If \( n = 0 \), then return 1 as the output.
2. Otherwise, return \( n \) times the value returned by recursively calling FACTORIAL\( (n-1) \).

The way I wrote step 2 is pretty cumbersome. I could instead just write "Otherwise, return \( n \cdot \text{FACTORIAL}(n-1) \)," using the recursive call's return value within a larger arithmetic expression.

For recursion to work, two properties must hold. First, there must be one or more base cases, where we compute the solution directly without recursion. Second, each recursive call of the procedure must be on a smaller instance of the same problem that will eventually reach a base case. For the FACTORIAL procedure, the base case occurs when \( n \) equals 0, and each recursive call is on an instance in which the value of \( n \) is reduced by 1. As long as the original value of \( n \) is nonnegative, the recursive calls will eventually get down to the base case.

Arguing that a recursive algorithm works might feel overly simple at first. The key is to believe that each recursive call produces the correct result. As long as we are willing to believe that recursive calls do the right thing, arguing correctness is often easy. Here is how we could argue that the FACTORIAL procedure returns the correct answer. Clearly, when \( n = 0 \), the value returned, 1, equals \( n! \). We assume that when \( n \geq 1 \), the recursive call FACTORIAL\( (n-1) \) does the right thing: it returns the value of \((n-1)!\). The procedure then multiplies this value by \( n \), thereby computing the value of \( n! \), which it returns.

Here’s an example where the recursive calls are not on smaller instances of the same problem, even though the mathematics is correct. It is indeed true that if \( n \geq 0 \), then \( n! = (n+1)!/(n+1) \). But the following recursive procedure, which takes advantage of this formula, would fail to ever give an answer when \( n \geq 1 \):
**Procedure BAD-FACTORIAL(n)**

*Input and Output:* Same as FACTORIAL.

1. If $n = 0$, then return 1 as the output.
2. Otherwise, return $\text{BAD-FACTORIAL}(n + 1)/(n + 1)$.

If we were to call BAD-FACTORIAL(1), it would generate a recursive call of BAD-FACTORIAL(2), which would generate a recursive call of BAD-FACTORIAL(3), and so on, never getting down to the base case when $n$ equals 0. If you were to implement this procedure in a real programming language and run it on an actual computer, you would quickly see something like a “stack overflow error.”

We can often rewrite algorithms that use a loop in a recursive style. Here is linear search, without a sentinel, written recursively:

**Procedure RECURSIVE-LINEAR-SEARCH(A, n, i, x)**

*Inputs:* Same as LINEAR-SEARCH, but with an added parameter $i$.

*Output:* The index of an element equaling $x$ in the subarray from $A[i]$ through $A[n]$, or NOT-FOUND if $x$ does not appear in this subarray.

1. If $i > n$, then return NOT-FOUND.
2. Otherwise ($i \leq n$), if $A[i] = x$, then return $i$.
3. Otherwise ($i \leq n$ and $A[i] \neq x$), return
   
   \[ \text{RECURSIVE-LINEAR-SEARCH}(A, n, i + 1, x). \]

Here, the subproblem is to search for $x$ in the subarray going from $A[i]$ through $A[n]$. The base case occurs in step 1 when this subarray is empty, that is, when $i > n$. Because the value of $i$ increases in each of step 3’s recursive calls, if no recursive call ever returns a value of $i$ in step 2, then eventually $i$ becomes greater than $n$ and we reach the base case.

**Further reading**

Chapters 2 and 3 of CLRS [CLRS09] cover much of the material in this chapter. An early algorithms textbook by Aho, Hopcroft, and Ullman [AHU74] influenced the field to use asymptotic notation to analyze algorithms. There has been quite a bit of work in proving programs correct; if you would like to delve into this area, try the books by Gries [Gri81] and Mitchell [Mit96].