2 Getting Started

This chapter will familiarize you with the framework we'll use throughout the book to think about the design and analysis of algorithms. It is self-contained, but it does include several references to material that will be introduced in Chapters 3 and 4. (It also contains several summations, which Appendix A shows how to solve.)

We'll begin by examining the insertion sort algorithm to solve the sorting problem introduced in Chapter 1. We'll specify algorithms using a pseudocode that should be understandable to you if you have done computer programming. We'll see why insertion sort correctly sorts and analyze its running time. The analysis introduces a notation that describes how running time increases with the number of items to be sorted. Following a discussion of insertion sort, we'll use a method called divide-and-conquer to develop a sorting algorithm called merge sort. We'll end with an analysis of merge sort's running time.

2.1 Insertion sort

Our first algorithm, insertion sort, solves the ***sorting problem*** introduced in Chapter 1:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

The numbers to be sorted are also known as the ***keys***. Although the problem is conceptually about sorting a sequence, the input comes in the form of an array with n elements. When we want to sort numbers, it's often because they are the keys associated with other data, which we call ***satellite data***. Together, a key and satellite data form a ***record***. For example, consider a spreadsheet containing student records with many associated pieces of data such as age, grade-point average, and number of courses taken. Any one of these quantities could be a key, but when the

spreadsheet sorts, it moves the associated record (the satellite data) with the key. When describing a sorting algorithm, we focus on the keys, but it is important to remember that there usually is associated satellite data.

In this book, we'll typically describe algorithms as procedures written in a ***pseudocode*** that is similar in many respects to C, C++, Java, Python,[1] or JavaScript. (Apologies if we've omitted your favorite programming language. We can't list them all.) If you have been introduced to any of these languages, you should have little trouble understanding algorithms "coded" in pseudocode. What separates pseudocode from real code is that in pseudocode, we employ whatever expressive method is most clear and concise to specify a given algorithm. Sometimes the clearest method is English, so do not be surprised if you come across an English phrase or sentence embedded within a section that looks more like real code. Another difference between pseudocode and real code is that pseudocode often ignores aspects of software engineering—such as data abstraction, modularity, and error handling—in order to convey the essence of the algorithm more concisely.

We start with ***insertion sort***, which is an efficient algorithm for sorting a small number of elements. Insertion sort works the way you might sort a hand of playing cards. Start with an empty left hand and the cards in a pile on the table. Pick up the first card in the pile and hold it with your left hand. Then, with your right hand, remove one card at a time from the pile, and insert it into the correct position in your left hand. As Figure 2.1 illustrates, you find the correct position for a card by comparing it with each of the cards already in your left hand, starting at the right and moving left. As soon as you see a card in your left hand whose value is less than or equal to the card you're holding in your right hand, insert the card that you're holding in your right hand just to the right of this card in your left hand. If all the cards in your left hand have values greater than the card in your right hand, then place this card as the leftmost card in your left hand. At all times, the cards held in your left hand are sorted, and these cards were originally the top cards of the pile on the table.

The pseudocode for insertion sort is given as the procedure INSERTION-SORT on the facing page. It takes two parameters: an array A containing the values to be sorted and the number n of values of sort. The values occupy positions $A[1]$ through $A[n]$ of the array, which we denote by $A[1:n]$. When the INSERTION-SORT procedure is finished, array $A[1:n]$ contains the original values, but in sorted order.

---

[1] If you're familiar with only Python, you can think of arrays as similar to Python lists.

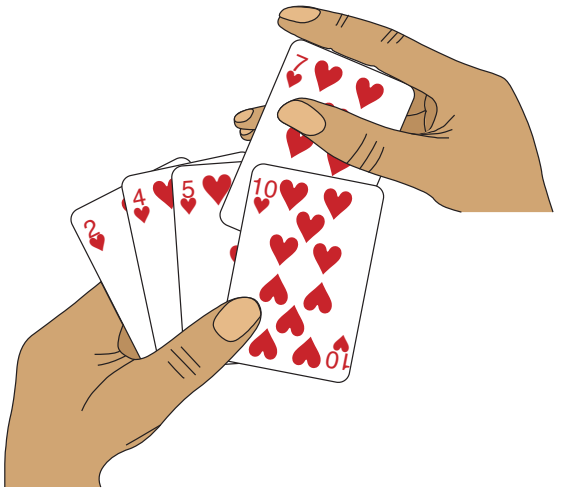

Figure 2.1 Sorting a hand of cards using insertion sort.

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

Loop invariants and the correctness of insertion sort

Figure 2.2 shows how this algorithm works for an array A that starts out with the sequence $\langle 5, 2, 4, 6, 1, 3 \rangle$. The index i indicates the "current card" being inserted into the hand. At the beginning of each iteration of the **for** loop, which is indexed by i, the ***subarray*** (a contiguous portion of the array) consisting of elements $A[1 : i - 1]$ (that is, $A[1]$ through $A[i - 1]$) constitutes the currently sorted hand, and the remaining subarray $A[i + 1 : n]$ (elements $A[i + 1]$ through $A[n]$) corresponds to the pile of cards still on the table. In fact, elements $A[1 : i - 1]$ are the elements *originally* in positions 1 through $i - 1$, but now in sorted order. We state these properties of $A[1 : i - 1]$ formally as a ***loop invariant***:

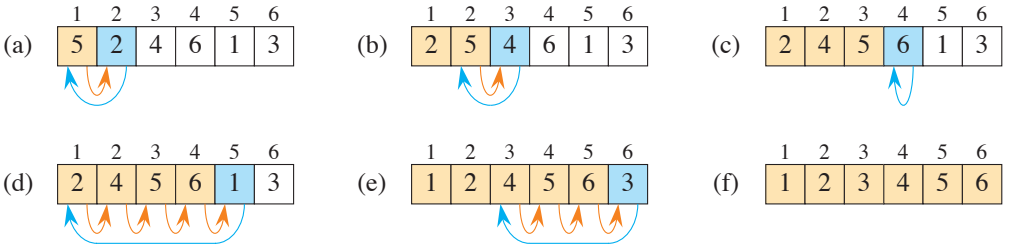

Figure 2.2 The operation of INSERTION-SORT(A, n), where A initially contains the sequence $\langle 5, 2, 4, 6, 1, 3 \rangle$ and $n = 6$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the blue rectangle holds the key taken from $A[i]$, which is compared with the values in tan rectangles to its left in the test of line 5. Orange arrows show array values moved one position to the right in line 6, and blue arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

> At the start of each iteration of the **for** loop of lines 1–8, the subarray $A[1 : i - 1]$ consists of the elements originally in $A[1 : i - 1]$, but in sorted order.

Loop invariants help us understand why an algorithm is correct. When you're using a loop invariant, you need to show three things:

Initialization: It is true prior to 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 terminates, the invariant—usually along with the reason that the loop terminated—gives us a useful property that helps show that the algorithm is correct.

When the first two properties hold, the loop invariant is true prior to every iteration of the loop. (Of course, you are free to use established facts other than the loop invariant itself to prove that the loop invariant remains true before each iteration.) A loop-invariant proof is a form of mathematical induction, where to prove that a property holds, you prove a base case and an inductive step. Here, showing that the invariant holds before the first iteration corresponds to the base case, and showing that the invariant holds from iteration to iteration corresponds to the inductive step.

The third property is perhaps the most important one, since you are using the loop invariant to show correctness. Typically, you use the loop invariant along with the condition that caused the loop to terminate. Mathematical induction typically applies the inductive step infinitely, but in a loop invariant the "induction" stops when the loop terminates.

Let's see how these properties hold for insertion sort.

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $i = 2$.[2] The subarray $A[1:i-1]$ consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (after all, how could a subarray with just one value not be sorted?), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the **for** loop works by moving the values in $A[i-1]$, $A[i-2]$, $A[i-3]$, and so on by one position to the right until it finds the proper position for $A[i]$ (lines 4–7), at which point it inserts the value of $A[i]$ (line 8). The subarray $A[1:i]$ then consists of the elements originally in $A[1:i]$, but in sorted order. ***Incrementing*** i (increasing its value by 1) for the next iteration of the **for** loop then preserves the loop invariant.

A more formal treatment of the second property would require us to state and show a loop invariant for the **while** loop of lines 5–7. Let's not get bogged down in such formalism just yet. Instead, we'll rely on our informal analysis to show that the second property holds for the outer loop.

Termination: Finally, we examine loop termination. The loop variable i starts at 2 and increases by 1 in each iteration. Once i's value exceeds n in line 1, the loop terminates. That is, the loop terminates once i equals $n+1$. Substituting $n+1$ for i in the wording of the loop invariant yields that the subarray $A[1:n]$ consists of the elements originally in $A[1:n]$, but in sorted order. Hence, the algorithm is correct.

This method of loop invariants is used to show correctness in various places throughout this book.

Pseudocode conventions

We use the following conventions in our pseudocode.

- Indentation indicates block structure. For example, the body of the **for** loop that begins on line 1 consists of lines 2–8, and the body of the **while** loop that

---

[2] When the loop is a **for** loop, the loop-invariant check just prior to the first iteration occurs immediately after the initial assignment to the loop-counter variable and just before the first test in the loop header. In the case of INSERTION-SORT, this time is after assigning 2 to the variable i but before the first test of whether $i \le n$.

begins on line 5 contains lines 6–7 but not line 8. Our indentation style applies to **if**-**else** statements[3] as well. Using indentation instead of textual indicators of block structure, such as **begin** and **end** statements or curly braces, reduces clutter while preserving, or even enhancing, clarity.[4]

- The looping constructs **while**, **for**, and **repeat**-**until** and the **if**-**else** conditional construct have interpretations similar to those in C, C++, Java, Python, and JavaScript.[5] In this book, the loop counter retains its value after the loop is exited, unlike some situations that arise in C++ and Java. Thus, immediately after a **for** loop, the loop counter's value is the value that first exceeded the **for** loop bound.[6] We used this property in our correctness argument for insertion sort. The **for** loop header in line 1 is **for** $i = 2$ **to** n, and so when this loop terminates, i equals $n+1$. We use the keyword **to** when a **for** loop increments its loop counter in each iteration, and we use the keyword **downto** when a **for** loop ***decrements*** its loop counter (reduces its value by 1 in each iteration). When the loop counter changes by an amount greater than 1, the amount of change follows the optional keyword **by**.
- The symbol "**//**" indicates that the remainder of the line is a comment.
- Variables (such as i, j, and *key*) are local to the given procedure. We won't use global variables without explicit indication.
- We access array elements by specifying the array name followed by the index in square brackets. For example, $A[i]$ indicates the ith element of the array A.

  Although many programming languages enforce 0-origin indexing for arrays (0 is the smallest valid index), we choose whichever indexing scheme is clearest for human readers to understand. Because people usually start counting at 1, not 0, most—but not all—of the arrays in this book use 1-origin indexing. To be

---

[3] In an **if**-**else** statement, we indent **else** at the same level as its matching **if**. The first executable line of an **else** clause appears on the same line as the keyword **else**. For multiway tests, we use **elseif** for tests after the first one. When it is the first line in an **else** clause, an **if** statement appears on the line following **else** so that you do not misconstrue it as **elseif**.

[4] Each pseudocode procedure in this book appears on one page so that you do not need to discern levels of indentation in pseudocode that is split across pages.

[5] Most block-structured languages have equivalent constructs, though the exact syntax may differ. Python lacks **repeat**-**until** loops, and its **for** loops operate differently from the **for** loops in this book. Think of the pseudocode line "**for** $i = 1$ **to** n" as equivalent to "for i in range(1, n+1)" in Python.

[6] In Python, the loop counter retains its value after the loop is exited, but the value it retains is the value it had during the final iteration of the **for** loop, rather than the value that exceeded the loop bound. That is because a Python **for** loop iterates through a list, which may contain nonnumeric values.

clear about whether a particular algorithm assumes 0-origin or 1-origin indexing, we'll specify the bounds of the arrays explicitly. If you are implementing an algorithm that we specify using 1-origin indexing, but you're writing in a programming language that enforces 0-origin indexing (such as C, C++, Java, Python, or JavaScript), then give yourself credit for being able to adjust. You can either always subtract 1 from each index or allocate each array with one extra position and just ignore position 0.

The notation ":" denotes a subarray. Thus, $A[i : j]$ indicates the subarray of A consisting of the elements $A[i], A[i + 1], \ldots, A[j]$.[7] We also use this notation to indicate the bounds of an array, as we did earlier when discussing the array $A[1 : n]$.

- We typically organize compound data into ***objects***, which are composed of ***attributes***. We access a particular attribute using the syntax found in many object-oriented programming languages: the object name, followed by a dot, followed by the attribute name. For example, if an object x has attribute f, we denote this attribute by $x.f$.

  We treat a variable representing an array or object as a pointer (known as a reference in some programming languages) to the data representing the array or object. For all attributes f of an object x, setting $y = x$ causes $y.f$ to equal $x.f$. Moreover, if we now set $x.f = 3$, then afterward not only does $x.f$ equal 3, but $y.f$ equals 3 as well. In other words, x and y point to the same object after the assignment $y = x$. This way of treating arrays and objects is consistent with most contemporary programming languages.

  Our attribute notation can "cascade." For example, suppose that the attribute f is itself a pointer to some type of object that has an attribute g. Then the notation $x.f.g$ is implicitly parenthesized as $(x.f).g$. In other words, if we had assigned $y = x.f$, then $x.f.g$ is the same as $y.g$.

  Sometimes a pointer refers to no object at all. In this case, we give it the special value NIL.

- We pass parameters to a procedure ***by value***: the called procedure receives its own copy of the parameters, and if it assigns a value to a parameter, the change is *not* seen by the calling procedure. When objects are passed, the pointer to the data representing the object is copied, but the object's attributes are not. For example, if x is a parameter of a called procedure, the assignment $x = y$ within

---

[7] If you're used to programming in Python, bear in mind that in this book, the subarray $A[i : j]$ includes the element $A[j]$. In Python, the last element of $A[i : j]$ is $A[j - 1]$. Python allows negative indices, which count from the back end of the list. This book does not use negative array indices.

the called procedure is not visible to the calling procedure. The assignment $x.f = 3$, however, is visible if the calling procedure has a pointer to the same object as x. Similarly, arrays are passed by pointer, so that a pointer to the array is passed, rather than the entire array, and changes to individual array elements are visible to the calling procedure. Again, most contemporary programming languages work this way.

- A **return** statement immediately transfers control back to the point of call in the calling procedure. Most **return** statements also take a value to pass back to the caller. Our pseudocode differs from many programming languages in that we allow multiple values to be returned in a single **return** statement without having to create objects to package them together.[8]
- The boolean operators "and" and "or" are ***short circuiting***. That is, evaluate the expression "x and y" by first evaluating x. If x evaluates to FALSE, then the entire expression cannot evaluate to TRUE, and therefore y is not evaluated. If, on the other hand, x evaluates to TRUE, y must be evaluated to determine the value of the entire expression. Similarly, in the expression "x or y" the expression y is evaluated only if x evaluates to FALSE. Short-circuiting operators allow us to write boolean expressions such as "$x \neq$ NIL and $x.f = y$" without worrying about what happens upon evaluating $x.f$ when x is NIL.
- The keyword **error** indicates that an error occurred because conditions were wrong for the procedure to have been called, and the procedure immediately terminates. The calling procedure is responsible for handling the error, and so we do not specify what action to take.

Exercises

2.1-1

Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on an array initially containing the sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2

Consider the procedure SUM-ARRAY on the facing page. It computes the sum of the n numbers in array $A[1:n]$. State a loop invariant for this procedure, and use its initialization, maintenance, and termination properties to show that the SUM-ARRAY procedure returns the sum of the numbers in $A[1:n]$.

---

[8] Python's tuple notation allows **return** statements to return multiple values without creating objects from a programmer-defined class.

```
SUM-ARRAY(A, n)
1  sum = 0
2  for i = 1 to n
3      sum = sum + A[i]
4  return sum
```

2.1-3
Rewrite the INSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing order.

2.1-4
Consider the ***searching problem***:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ stored in array $A[1:n]$ and a value x.

Output: An index i such that x equals $A[i]$ or the special value NIL if x does not appear in A.

Write pseudocode for ***linear search***, which scans through the array from beginning to end, looking for x. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

2.1-5
Consider the problem of adding two n-bit binary integers a and b, stored in two n-element arrays $A[0:n-1]$ and $B[0:n-1]$, where each element is either 0 or 1, $a = \sum_{i=0}^{n-1} A[i] \cdot 2^i$, and $b = \sum_{i=0}^{n-1} B[i] \cdot 2^i$. The sum $c = a + b$ of the two integers should be stored in binary form in an $(n+1)$-element array $C[0:n]$, where $c = \sum_{i=0}^{n} C[i] \cdot 2^i$. Write a procedure ADD-BINARY-INTEGERS that takes as input arrays A and B, along with the length n, and returns array C holding the sum.

2.2 Analyzing algorithms

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. You might consider resources such as memory, communication bandwidth, or energy consumption. Most often, however, you'll want to measure computational time. If you analyze several candidate algorithms for a problem,

you can identify the most efficient one. There might be more than just one viable candidate, but you can often rule out several inferior algorithms in the process.

Before you can analyze an algorithm, you need a model of the technology that it runs on, including the resources of that technology and a way to express their costs. Most of this book assumes a generic one-processor, ***random-access machine (RAM)*** model of computation as the implementation technology, with the understanding that algorithms are implemented as computer programs. In the RAM model, instructions execute one after another, with no concurrent operations. The RAM model assumes that each instruction takes the same amount of time as any other instruction and that each data access—using the value of a variable or storing into a variable—takes the same amount of time as any other data access. In other words, in the RAM model each instruction or data access takes a constant amount of time—even indexing into an array.[9]

Strictly speaking, we should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and yield little insight into algorithm design and analysis. Yet we must be careful not to abuse the RAM model. For example, what if a RAM had an instruction that sorts? Then you could sort in just one step. Such a RAM would be unrealistic, since such instructions do not appear in real computers. Our guide, therefore, is how real computers are designed. The RAM model contains instructions commonly found in real computers: arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return).

The data types in the RAM model are integer, floating point (for storing real-number approximations), and character. Real computers do not usually have a separate data type for the boolean values TRUE and FALSE. Instead, they often test whether an integer value is 0 (FALSE) or nonzero (TRUE), as in C. Although we typically do not concern ourselves with precision for floating-point values in this book (many numbers cannot be represented exactly in floating point), precision is crucial for most applications. We also assume that each word of data has a limit on the number of bits. For example, when working with inputs of size n, we typically

---

[9] We assume that each element of a given array occupies the same number of bytes and that the elements of a given array are stored in contiguous memory locations. For example, if array $A[1:n]$ starts at memory address 1000 and each element occupies four bytes, then element $A[i]$ is at address $1000 + 4(i - 1)$. In general, computing the address in memory of a particular array element requires at most one subtraction (no subtraction for a 0-origin array), one multiplication (often implemented as a shift operation if the element size is an exact power of 2), and one addition. Furthermore, for code that iterates through the elements of an array in order, an optimizing compiler can generate the address of each element using just one addition, by adding the element size to the address of the preceding element.

assume that integers are represented by $c \log_2 n$ bits for some constant $c \geq 1$. We require $c \geq 1$ so that each word can hold the value of n, enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily. (If the word size could grow arbitrarily, we could store huge amounts of data in one word and operate on it all in constant time—an unrealistic scenario.)

Real computers contain instructions not listed above, and such instructions represent a gray area in the RAM model. For example, is exponentiation a constant-time instruction? In the general case, no: to compute x^n when x and n are general integers typically takes time logarithmic in n (see equation (31.34) on page 934), and you must worry about whether the result fits into a computer word. If n is an exact power of 2, however, exponentiation can usually be viewed as a constant-time operation. Many computers have a "shift left" instruction, which in constant time shifts the bits of an integer by n positions to the left. In most computers, shifting the bits of an integer by 1 position to the left is equivalent to multiplying by 2, so that shifting the bits by n positions to the left is equivalent to multiplying by 2^n. Therefore, such computers can compute 2^n in 1 constant-time instruction by shifting the integer 1 by n positions to the left, as long as n is no more than the number of bits in a computer word. We'll try to avoid such gray areas in the RAM model and treat computing 2^n and multiplying by 2^n as constant-time operations when the result is small enough to fit in a computer word.

The RAM model does not account for the memory hierarchy that is common in contemporary computers. It models neither caches nor virtual memory. Several other computational models attempt to account for memory-hierarchy effects, which are sometimes significant in real programs on real machines. Section 11.5 and a handful of problems in this book examine memory-hierarchy effects, but for the most part, the analyses in this book do not consider them. Models that include the memory hierarchy are quite a bit more complex than the RAM model, and so they can be difficult to work with. Moreover, RAM-model analyses are usually excellent predictors of performance on actual machines.

Although it is often straightforward to analyze an algorithm in the RAM model, sometimes it can be quite a challenge. You might need to employ mathematical tools such as combinatorics, probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula. Because an algorithm might behave differently for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

Analysis of insertion sort

How long does the INSERTION-SORT procedure take? One way to tell would be for you to run it on your computer and time how long it takes to run. Of course, you'd

first have to implement it in a real programming language, since you cannot run our pseudocode directly. What would such a timing test tell you? You would find out how long insertion sort takes to run on your particular computer, on that particular input, under the particular implementation that you created, with the particular compiler or interpreter that you ran, with the particular libraries that you linked in, and with the particular background tasks that were running on your computer concurrently with your timing test (such as checking for incoming information over a network). If you run insertion sort again on your computer with the same input, you might even get a different timing result. From running just one implementation of insertion sort on just one computer and on just one input, what would you be able to determine about insertion sort's running time if you were to give it a different input, if you were to run it on a different computer, or if you were to implement it in a different programming language? Not much. We need a way to predict, given a new input, how long insertion sort will take.

Instead of timing a run, or even several runs, of insertion sort, we can determine how long it takes by analyzing the algorithm itself. We'll examine how many times it executes each line of pseudocode and how long each line of pseudocode takes to run. We'll first come up with a precise but complicated formula for the running time. Then, we'll distill the important part of the formula using a convenient notation that can help us compare the running times of different algorithms for the same problem.

How do we analyze insertion sort? First, let's acknowledge that the running time depends on the input. You shouldn't be terribly surprised that sorting a thousand numbers takes longer than sorting three numbers. Moreover, insertion sort can take different amounts of time to sort two input arrays of the same size, depending on how nearly sorted they already are. Even though the running time can depend on many features of the input, we'll focus on the one that has been shown to have the greatest effect, namely the size of the input, and describe the running time of a program as a function of the size of its input. To do so, we need to define the terms "running time" and "input size" more carefully. We also need to be clear about whether we are discussing the running time for an input that elicits the worst-case behavior, the best-case behavior, or some other case.

The best notion for ***input size*** depends on the problem being studied. For many problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the *number of items in the input*—for example, the number n of items being sorted. For many other problems, such as multiplying two integers, the best measure of input size is the *total number of bits* needed to represent the input in ordinary binary notation. Sometimes it is more appropriate to describe the size of the input with more than just one number. For example, if the input to an algorithm is a graph, we usually characterize the input size by both the number

of vertices and the number of edges in the graph. We'll indicate which input size measure is being used with each problem we study.

The ***running time*** of an algorithm on a particular input is the number of instructions and data accesses executed. How we account for these costs should be independent of any particular computer, but within the framework of the RAM model. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudocode. One line might take more or less time than another line, but we'll assume that each execution of the kth line takes c_k time, where c_k is a constant. This viewpoint is in keeping with the RAM model, and it also reflects how the pseudocode would be implemented on most actual computers.[10]

Let's analyze the INSERTION-SORT procedure. As promised, we'll start by devising a precise formula that uses the input size and all the statement costs c_k. This formula turns out to be messy, however. We'll then switch to a simpler notation that is more concise and easier to use. This simpler notation makes clear how to compare the running times of algorithms, especially as the size of the input increases.

To analyze the INSERTION-SORT procedure, let's view it on the following page with the time cost of each statement and the number of times each statement is executed. For each $i = 2, 3, \ldots, n$, let t_i denote the number of times the **while** loop test in line 5 is executed for that value of i. When a **for** or **while** loop exits in the usual way—because the test in the loop header comes up FALSE—the test is executed one time more than the loop body. Because comments are not executable statements, assume that they take no time.

The running time of the algorithm is the sum of running times for each statement executed. A statement that takes c_k steps to execute and executes m times contributes $c_k m$ to the total running time.[11] We usually denote the running time of an algorithm on an input of size n by $T(n)$. To compute $T(n)$, the running time of INSERTION-SORT on an input of n values, we sum the products of the *cost* and *times* columns, obtaining

---

[10] There are some subtleties here. Computational steps that we specify in English are often variants of a procedure that requires more than just a constant amount of time. For example, in the RADIX-SORT procedure on page 213, one line reads "use a stable sort to sort array A on digit i," which, as we shall see, takes more than a constant amount of time. Also, although a statement that calls a subroutine takes only constant time, the subroutine itself, once invoked, may take more. That is, we separate the process of ***calling*** the subroutine—passing parameters to it, etc.—from the process of ***executing*** the subroutine.

[11] This characteristic does not necessarily hold for a resource such as memory. A statement that references m words of memory and is executed n times does not necessarily reference mn distinct words of memory.

```
INSERTION-SORT(A, n)                                   cost   times
1  for i = 2 to n                                      c1     n
2      key = A[i]                                      c2     n − 1
3      // Insert A[i] into the sorted subarray A[1 : i − 1].   0      n − 1
4      j = i − 1                                       c4     n − 1
5      while j > 0 and A[j] > key                      c5     ∑_{i=2}^{n} t_i
6          A[j + 1] = A[j]                             c6     ∑_{i=2}^{n} (t_i − 1)
7          j = j − 1                                   c7     ∑_{i=2}^{n} (t_i − 1)
8      A[j + 1] = key                                  c8     n − 1
```

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) \; .$$

Even for inputs of a given size, an algorithm's running time may depend on *which* input of that size is given. For example, in INSERTION-SORT, the best case occurs when the array is already sorted. In this case, each time that line 5 executes, the value of *key*—the value originally in $A[i]$—is already greater than or equal to all values in $A[1 : i-1]$, so that the **while** loop of lines 5–7 always exits upon the first test in line 5. Therefore, we have that $t_i = 1$ for $i = 2, 3, \ldots, n$, and the best-case running time is given by

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \; . \end{aligned} \tag{2.1}$$

We can express this running time as $an + b$ for *constants* a and b that depend on the statement costs c_k (where $a = c_1 + c_2 + c_4 + c_5 + c_8$ and $b = c_2 + c_4 + c_5 + c_8$). The running time is thus a ***linear function*** of n.

The worst case arises when the array is in reverse sorted order—that is, it starts out in decreasing order. The procedure must compare each element $A[i]$ with each element in the entire sorted subarray $A[1 : i-1]$, and so $t_i = i$ for $i = 2, 3, \ldots, n$. (The procedure finds that $A[j] > key$ every time in line 5, and the **while** loop exits only when j reaches 0.) Noting that

$$\begin{aligned} \sum_{i=2}^{n} i &= \left(\sum_{i=1}^{n} i \right) - 1 \\ &= \frac{n(n+1)}{2} - 1 \quad \text{(by equation (A.2) on page 1141)} \end{aligned}$$

and

$$\sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i$$
$$= \frac{n(n-1)}{2} \quad \text{(again, by equation (A.2))} ,$$

we find that in the worst case, the running time of INSERTION-SORT is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned} \tag{2.2}$$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_k (now, $a = c_5/2 + c_6/2 + c_7/2$, $b = c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8$, and $c = -(c_2 + c_4 + c_5 + c_8)$). The running time is thus a ***quadratic function*** of n.

Typically, as in insertion sort, the running time of an algorithm is fixed for a given input, although we'll also see some interesting "randomized" algorithms whose behavior can vary even for a fixed input.

Worst-case and average-case analysis

Our analysis of insertion sort looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted. For the remainder of this book, though, we'll usually (but not always) concentrate on finding only the ***worst-case running time***, that is, the longest running time for *any* input of size n. Why? Here are three reasons:

- The worst-case running time of an algorithm gives an upper bound on the running time for *any* input. If you know it, then you have a guarantee that the algorithm never takes any longer. You need not make some educated guess about the running time and hope that it never gets much worse. This feature is especially important for real-time computing, in which operations must complete by a deadline.
- For some algorithms, the worst case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithm's worst case often occurs when the information is not present in the database. In some applications, searches for absent information may be frequent.

- The "average case" is often roughly as bad as the worst case. Suppose that you run insertion sort on an array of n randomly chosen numbers. How long does it take to determine where in subarray $A[1:i-1]$ to insert element $A[i]$? On average, half the elements in $A[1:i-1]$ are less than $A[i]$, and half the elements are greater. On average, therefore, $A[i]$ is compared with just half of the subarray $A[1:i-1]$, and so t_i is about $i/2$. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

In some particular cases, we'll be interested in the ***average-case*** running time of an algorithm. We'll see the technique of ***probabilistic analysis*** applied to various algorithms throughout this book. The scope of average-case analysis is limited, because it may not be apparent what constitutes an "average" input for a particular problem. Often, we'll assume that all inputs of a given size are equally likely. In practice, this assumption may be violated, but we can sometimes use a ***randomized algorithm***, which makes random choices, to allow a probabilistic analysis and yield an ***expected*** running time. We explore randomized algorithms more in Chapter 5 and in several other subsequent chapters.

Order of growth

In order to ease our analysis of the INSERTION-SORT procedure, we used some simplifying abstractions. First, we ignored the actual cost of each statement, using the constants c_k to represent these costs. Still, the best-case and worst-case running times in equations (2.1) and (2.2) are rather unwieldy. The constants in these expressions give us more detail than we really need. That's why we also expressed the best-case running time as $an + b$ for constants a and b that depend on the statement costs c_k and why we expressed the worst-case running time as $an^2 + bn + c$ for constants a, b, and c that depend on the statement costs. We thus ignored not only the actual statement costs, but also the abstract costs c_k.

Let's now make one more simplifying abstraction: it is the ***rate of growth***, or ***order of growth***, of the running time that really interests us. We therefore consider only the leading term of a formula (e.g., an^2), since the lower-order terms are relatively insignificant for large values of n. We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. For insertion sort's worst-case running time, when we ignore the lower-order terms and the leading term's constant coefficient, only the factor of n^2 from the leading term remains. That factor, n^2, is by far the most important part of the running time. For example, suppose that an algorithm implemented on a particular machine takes $n^2/100 + 100n + 17$ microseconds on an input of size n. Although the coefficients of $1/100$ for the n^2 term and 100 for the n term differ by four orders of magnitude, the $n^2/100$ term domi-

nates the $100n$ term once n exceeds 10,000. Although 10,000 might seem large, it is smaller than the population of an average town. Many real-world problems have much larger input sizes.

To highlight the order of growth of the running time, we have a special notation that uses the Greek letter Θ (theta). We write that insertion sort has a worst-case running time of $\Theta(n^2)$ (pronounced "theta of n-squared" or just "theta n-squared"). We also write that insertion sort has a best-case running time of $\Theta(n)$ ("theta of n" or "theta n"). For now, think of Θ-notation as saying "roughly proportional when n is large," so that $\Theta(n^2)$ means "roughly proportional to n^2 when n is large" and $\Theta(n)$ means "roughly proportional to n when n is large" We'll use Θ-notation informally in this chapter and define it precisely in Chapter 3.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth. Due to constant factors and lower-order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth. But on large enough inputs, an algorithm whose worst-case running time is $\Theta(n^2)$, for example, takes less time in the worst case than an algorithm whose worst-case running time is $\Theta(n^3)$. Regardless of the constants hidden by the Θ-notation, there is always some number, say n_0, such that for all input sizes $n \geq n_0$, the $\Theta(n^2)$ algorithm beats the $\Theta(n^3)$ algorithm in the worst case.

Exercises

2.2-1

Express the function $n^3/1000 + 100n^2 - 100n + 3$ in terms of Θ-notation.

2.2-2

Consider sorting n numbers stored in array $A[1:n]$ by first finding the smallest element of $A[1:n]$ and exchanging it with the element in $A[1]$. Then find the smallest element of $A[2:n]$, and exchange it with $A[2]$. Then find the smallest element of $A[3:n]$, and exchange it with $A[3]$. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as ***selection sort***. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the worst-case running time of selection sort in Θ-notation. Is the best-case running time any better?

2.2-3

Consider linear search again (see Exercise 2.1-4). How many elements of the input array need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case?

Using Θ-notation, give the average-case and worst-case running times of linear search. Justify your answers.

2.2-4
How can you modify any sorting algorithm to have a good best-case running time?

2.3 Designing algorithms

You can choose from a wide range of algorithm design techniques. Insertion sort uses the ***incremental*** method: for each element $A[i]$, insert it into its proper place in the subarray $A[1:i]$, having already sorted the subarray $A[1:i-1]$.

This section examines another design method, known as "divide-and-conquer," which we explore in more detail in Chapter 4. We'll use divide-and-conquer to design a sorting algorithm whose worst-case running time is much less than that of insertion sort. One advantage of using an algorithm that follows the divide-and-conquer method is that analyzing its running time is often straightforward, using techniques that we'll explore in Chapter 4.

2.3.1 The divide-and-conquer method

Many useful algorithms are ***recursive*** in structure: to solve a given problem, they ***recurse*** (call themselves) one or more times to handle closely related subproblems. These algorithms typically follow the ***divide-and-conquer*** method: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

In the divide-and-conquer method, if the problem is small enough—the ***base case***—you just solve it directly without recursing. Otherwise—the ***recursive case***—you perform three characteristic steps:

Divide the problem into one or more subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively.

Combine the subproblem solutions to form a solution to the original problem.

The ***merge sort*** algorithm closely follows the divide-and-conquer method. In each step, it sorts a subarray $A[p:r]$, starting with the entire array $A[1:n]$ and recursing down to smaller and smaller subarrays. Here is how merge sort operates:

3 Characterizing Running Times

The order of growth of the running time of an algorithm, defined in Chapter 2, gives a simple way to characterize the algorithm's efficiency and also allows us to compare it with alternative algorithms. Once the input size n becomes large enough, merge sort, with its $\Theta(n \lg n)$ worst-case running time, beats insertion sort, whose worst-case running time is $\Theta(n^2)$. Although we can sometimes determine the exact running time of an algorithm, as we did for insertion sort in Chapter 2, the extra precision is rarely worth the effort of computing it. For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.

When we look at input sizes large enough to make relevant only the order of growth of the running time, we are studying the ***asymptotic*** efficiency of algorithms. That is, we are concerned with how the running time of an algorithm increases with the size of the input *in the limit*, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient is the best choice for all but very small inputs.

This chapter gives several standard methods for simplifying the asymptotic analysis of algorithms. The next section presents informally the three most commonly used types of "asymptotic notation," of which we have already seen an example in Θ-notation. It also shows one way to use these asymptotic notations to reason about the worst-case running time of insertion sort. Then we look at asymptotic notations more formally and present several notational conventions used throughout this book. The last section reviews the behavior of functions that commonly arise when analyzing algorithms.

3.1 O-notation, Ω-notation, and Θ-notation

When we analyzed the worst-case running time of insertion sort in Chapter 2, we started with the complicated expression

$$\left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ - (c_2 + c_4 + c_5 + c_8) \; .$$

We then discarded the lower-order terms $(c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8)n$ and $c_2 + c_4 + c_5 + c_8$, and we also ignored the coefficient $c_5/2 + c_6/2 + c_7/2$ of n^2. That left just the factor n^2, which we put into Θ-notation as $\Theta(n^2)$. We use this style to characterize running times of algorithms: discard the lower-order terms and the coefficient of the leading term, and use a notation that focuses on the rate of growth of the running time.

Θ-notation is not the only such "asymptotic notation." In this section, we'll see other forms of asymptotic notation as well. We start with intuitive looks at these notations, revisiting insertion sort to see how we can apply them. In the next section, we'll see the formal definitions of our asymptotic notations, along with conventions for using them.

Before we get into specifics, bear in mind that the asymptotic notations we'll see are designed so that they characterize functions in general. It so happens that the functions we are most interested in denote the running times of algorithms. But asymptotic notation can apply to functions that characterize some other aspect of algorithms (the amount of space they use, for example), or even to functions that have nothing whatsoever to do with algorithms.

O-notation

O-notation characterizes an *upper bound* on the asymptotic behavior of a function. In other words, it says that a function grows *no faster* than a certain rate, based on the highest-order term. Consider, for example, the function $7n^3 + 100n^2 - 20n + 6$. Its highest-order term is $7n^3$, and so we say that this function's rate of growth is n^3. Because this function grows no faster than n^3, we can write that it is $O(n^3)$. You might be surprised that we can also write that the function $7n^3 + 100n^2 - 20n + 6$ is $O(n^4)$. Why? Because the function grows more slowly than n^4, we are correct in saying that it grows no faster. As you might have guessed, this function is also $O(n^5)$, $O(n^6)$, and so on. More generally, it is $O(n^c)$ for any constant $c \geq 3$.

Ω-notation

Ω-notation characterizes a *lower bound* on the asymptotic behavior of a function. In other words, it says that a function grows *at least as fast* as a certain rate, based —as in O-notation—on the highest-order term. Because the highest-order term in the function $7n^3 + 100n^2 - 20n + 6$ grows at least as fast as n^3, this function is $\Omega(n^3)$. This function is also $\Omega(n^2)$ and $\Omega(n)$. More generally, it is $\Omega(n^c)$ for any constant $c \leq 3$.

Θ-notation

Θ-notation characterizes a *tight bound* on the asymptotic behavior of a function. It says that a function grows *precisely* at a certain rate, based—once again—on the highest-order term. Put another way, Θ-notation characterizes the rate of growth of the function to within a constant factor from above and to within a constant factor from below. These two constant factors need not be equal.

If you can show that a function is both $O(f(n))$ and $\Omega(f(n))$ for some function $f(n)$, then you have shown that the function is $\Theta(f(n))$. (The next section states this fact as a theorem.) For example, since the function $7n^3+100n^2-20n+6$ is both $O(n^3)$ and $\Omega(n^3)$, it is also $\Theta(n^3)$.

Example: Insertion sort

Let's revisit insertion sort and see how to work with asymptotic notation to characterize its $\Theta(n^2)$ worst-case running time without evaluating summations as we did in Chapter 2. Here is the INSERTION-SORT procedure once again:

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

What can we observe about how the pseudocode operates? The procedure has nested loops. The outer loop is a **for** loop that runs $n - 1$ times, regardless of the values being sorted. The inner loop is a **while** loop, but the number of iterations it makes depends on the values being sorted. The loop variable j starts at $i - 1$

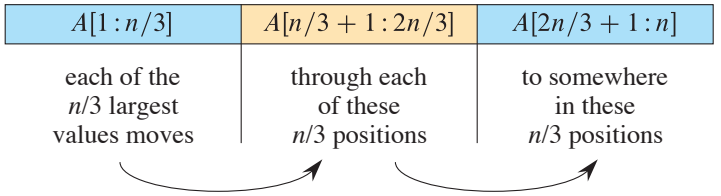

Figure 3.1 The $\Omega(n^2)$ lower bound for insertion sort. If the first $n/3$ positions contain the $n/3$ largest values, each of these values must move through each of the middle $n/3$ positions, one position at a time, to end up somewhere in the last $n/3$ positions. Since each of $n/3$ values moves through at least each of $n/3$ positions, the time taken in this case is at least proportional to $(n/3)(n/3) = n^2/9$, or $\Omega(n^2)$.

and decreases by 1 in each iteration until either it reaches 0 or $A[j] \leq key$. For a given value of i, the **while** loop might iterate 0 times, $i - 1$ times, or anywhere in between. The body of the **while** loop (lines 6–7) takes constant time per iteration of the **while** loop.

These observations suffice to deduce an $O(n^2)$ running time for any case of INSERTION-SORT, giving us a blanket statement that covers all inputs. The running time is dominated by the inner loop. Because each of the $n - 1$ iterations of the outer loop causes the inner loop to iterate at most $i - 1$ times, and because i is at most n, the total number of iterations of the inner loop is at most $(n - 1)(n - 1)$, which is less than n^2. Since each iteration of the inner loop takes constant time, the total time spent in the inner loop is at most a constant times n^2, or $O(n^2)$.

With a little creativity, we can also see that the worst-case running time of INSERTION-SORT is $\Omega(n^2)$. By saying that the worst-case running time of an algorithm is $\Omega(n^2)$, we mean that for every input size n above a certain threshold, there is at least one input of size n for which the algorithm takes at least cn^2 time, for some positive constant c. It does not necessarily mean that the algorithm takes at least cn^2 time for all inputs.

Let's now see why the worst-case running time of INSERTION-SORT is $\Omega(n^2)$. For a value to end up to the right of where it started, it must have been moved in line 6. In fact, for a value to end up k positions to the right of where it started, line 6 must have executed k times. As Figure 3.1 shows, let's assume that n is a multiple of 3 so that we can divide the array A into groups of $n/3$ positions. Suppose that in the input to INSERTION-SORT, the $n/3$ largest values occupy the first $n/3$ array positions $A[1:n/3]$. (It does not matter what relative order they have within the first $n/3$ positions.) Once the array has been sorted, each of these $n/3$ values ends up somewhere in the last $n/3$ positions $A[2n/3 + 1:n]$. For that to happen, each of these $n/3$ values must pass through each of the middle $n/3$ positions $A[n/3 + 1:2n/3]$. Each of these $n/3$ values passes through these middle

$n/3$ positions one position at a time, by at least $n/3$ executions of line 6. Because at least $n/3$ values have to pass through at least $n/3$ positions, the time taken by INSERTION-SORT in the worst case is at least proportional to $(n/3)(n/3) = n^2/9$, which is $\Omega(n^2)$.

Because we have shown that INSERTION-SORT runs in $O(n^2)$ time in all cases and that there is an input that makes it take $\Omega(n^2)$ time, we can conclude that the worst-case running time of INSERTION-SORT is $\Theta(n^2)$. It does not matter that the constant factors for upper and lower bounds might differ. What matters is that we have characterized the worst-case running time to within constant factors (discounting lower-order terms). This argument does not show that INSERTION-SORT runs in $\Theta(n^2)$ time in *all* cases. Indeed, we saw in Chapter 2 that the best-case running time is $\Theta(n)$.

Exercises

3.1-1
Modify the lower-bound argument for insertion sort to handle input sizes that are not necessarily a multiple of 3.

3.1-2
Using reasoning similar to what we used for insertion sort, analyze the running time of the selection sort algorithm from Exercise 2.2-2.

3.1-3
Suppose that α is a fraction in the range $0 < \alpha < 1$. Show how to generalize the lower-bound argument for insertion sort to consider an input in which the αn largest values start in the first αn positions. What additional restriction do you need to put on α? What value of α maximizes the number of times that the αn largest values must pass through each of the middle $(1 - 2\alpha)n$ array positions?

3.2 Asymptotic notation: formal definitions

Having seen asymptotic notation informally, let's get more formal. The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are typically the set $\mathbb{N}$ of natural numbers or the set $\mathbb{R}$ of real numbers. Such notations are convenient for describing a running-time function $T(n)$. This section defines the basic asymptotic notations and also introduces some common "proper" notational abuses.

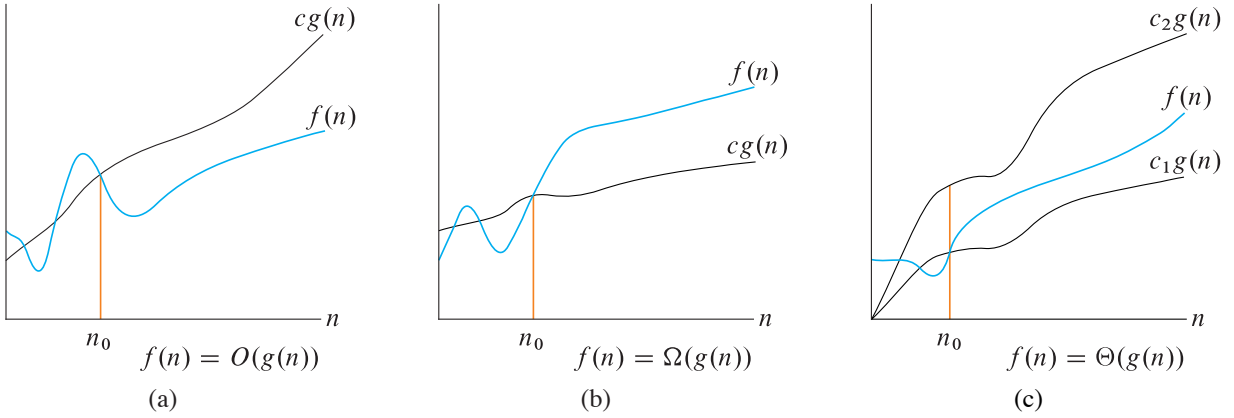

Figure 3.2 Graphic examples of the O, Ω, and Θ notations. In each part, the value of n_0 shown is the minimum possible value, but any greater value also works. **(a)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(b)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$. **(c)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that at and to the right of n_0, the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive.

O-notation

As we saw in Section 3.1, O-notation describes an ***asymptotic upper bound***. We use O-notation to give an upper bound on a function, to within a constant factor.

Here is the formal definition of O-notation. For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the *set of functions*

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.^1$$

A function $f(n)$ belongs to the set $O(g(n))$ if there exists a positive constant c such that $f(n) \le cg(n)$ for sufficiently large n. Figure 3.2(a) shows the intuition behind O-notation. For all values n at and to the right of n_0, the value of the function $f(n)$ is on or below $cg(n)$.

The definition of $O(g(n))$ requires that every function $f(n)$ in the set $O(g(n))$ be ***asymptotically nonnegative***: $f(n)$ must be nonnegative whenever n is sufficiently large. (An ***asymptotically positive*** function is one that is positive for all

[1] Within set notation, a colon means "such that."

sufficiently large n.) Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $O(g(n))$ is empty. We therefore assume that every function used within O-notation is asymptotically nonnegative. This assumption holds for the other asymptotic notations defined in this chapter as well.

You might be surprised that we define O-notation in terms of sets. Indeed, you might expect that we would write "$f(n) \in O(g(n))$" to indicate that $f(n)$ belongs to the set $O(g(n))$. Instead, we usually write "$f(n) = O(g(n))$" and say "$f(n)$ is big-oh of $g(n)$" to express the same notion. Although it may seem confusing at first to abuse equality in this way, we'll see later in this section that doing so has its advantages.

Let's explore an example of how to use the formal definition of O-notation to justify our practice of discarding lower-order terms and ignoring the constant coefficient of the highest-order term. We'll show that $4n^2 + 100n + 500 = O(n^2)$, even though the lower-order terms have much larger coefficients than the leading term. We need to find positive constants c and n_0 such that $4n^2 + 100n + 500 \le cn^2$ for all $n \ge n_0$. Dividing both sides by n^2 gives $4 + 100/n + 500/n^2 \le c$. This inequality is satisfied for many choices of c and n_0. For example, if we choose $n_0 = 1$, then this inequality holds for $c = 604$. If we choose $n_0 = 10$, then $c = 19$ works, and choosing $n_0 = 100$ allows us to use $c = 5.05$.

We can also use the formal definition of O-notation to show that the function $n^3 - 100n^2$ does not belong to the set $O(n^2)$, even though the coefficient of n^2 is a large negative number. If we had $n^3 - 100n^2 = O(n^2)$, then there would be positive constants c and n_0 such that $n^3 - 100n^2 \le cn^2$ for all $n \ge n_0$. Again, we divide both sides by n^2, giving $n - 100 \le c$. Regardless of what value we choose for the constant c, this inequality does not hold for any value of $n > c + 100$.

Ω-notation

Just as O-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an ***asymptotic lower bound***. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$

Figure 3.2(b) shows the intuition behind Ω-notation. For all values n at or to the right of n_0, the value of $f(n)$ is on or above $cg(n)$.

We've already shown that $4n^2 + 100n + 500 = O(n^2)$. Now let's show that $4n^2 + 100n + 500 = \Omega(n^2)$. We need to find positive constants c and n_0 such that $4n^2 + 100n + 500 \ge cn^2$ for all $n \ge n_0$. As before, we divide both sides by n^2,

giving $4 + 100/n + 500/n^2 \geq c$. This inequality holds when n_0 is any positive integer and $c = 4$.

What if we had subtracted the lower-order terms from the $4n^2$ term instead of adding them? What if we had a small coefficient for the n^2 term? The function would still be $\Omega(n^2)$. For example, let's show that $n^2/100 - 100n - 500 = \Omega(n^2)$. Dividing by n^2 gives $1/100 - 100/n - 500/n^2 \geq c$. We can choose any value for n_0 that is at least 10,005 and find a positive value for c. For example, when $n_0 = 10{,}005$, we can choose $c = 2.49 \times 10^{-9}$. Yes, that's a tiny value for c, but it is positive. If we select a larger value for n_0, we can also increase c. For example, if $n_0 = 100{,}000$, then we can choose $c = 0.0089$. The higher the value of n_0, the closer to the coefficient $1/100$ we can choose c.

Θ-notation

We use Θ-notation for ***asymptotically tight bounds***. For a given function $g(n)$, we denote by $\Theta(g(n))$ ("theta of g of n") the set of functions

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\} .$$

Figure 3.2(c) shows the intuition behind Θ-notation. For all values of n at and to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within constant factors.

The definitions of O-, Ω-, and Θ-notations lead to the following theorem, whose proof we leave as Exercise 3.2-4.

Theorem 3.1
For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ■

We typically apply Theorem 3.1 to prove asymptotically tight bounds from asymptotic upper and lower bounds.

Asymptotic notation and running times

When you use asymptotic notation to characterize an algorithm's running time, make sure that the asymptotic notation you use is as precise as possible without overstating which running time it applies to. Here are some examples of using asymptotic notation properly and improperly to characterize running times.

Let's start with insertion sort. We can correctly say that insertion sort's worst-case running time is $O(n^2)$, $\Omega(n^2)$, and—due to Theorem 3.1—$\Theta(n^2)$. Although

all three ways to characterize the worst-case running times are correct, the $\Theta(n^2)$ bound is the most precise and hence the most preferred. We can also correctly say that insertion sort's best-case running time is $O(n)$, $\Omega(n)$, and $\Theta(n)$, again with $\Theta(n)$ the most precise and therefore the most preferred.

Here is what we *cannot* correctly say: insertion sort's running time is $\Theta(n^2)$. That is an overstatement because by omitting "worst-case" from the statement, we're left with a blanket statement covering all cases. The error here is that insertion sort does not run in $\Theta(n^2)$ time in all cases since, as we've seen, it runs in $\Theta(n)$ time in the best case. We can correctly say that insertion sort's running time is $O(n^2)$, however, because in all cases, its running time grows no faster than n^2. When we say $O(n^2)$ instead of $\Theta(n^2)$, there is no problem in having cases whose running time grows more slowly than n^2. Likewise, we cannot correctly say that insertion sort's running time is $\Theta(n)$, but we can say that its running time is $\Omega(n)$.

How about merge sort? Since merge sort runs in $\Theta(n \lg n)$ time in all cases, we can just say that its running time is $\Theta(n \lg n)$ without specifying worst-case, best-case, or any other case.

People occasionally conflate O-notation with Θ-notation by mistakenly using O-notation to indicate an asymptotically tight bound. They say things like "an $O(n \lg n)$-time algorithm runs faster than an $O(n^2)$-time algorithm." Maybe it does, maybe it doesn't. Since O-notation denotes only an asymptotic upper bound, that so-called $O(n^2)$-time algorithm might actually run in $\Theta(n)$ time. You should be careful to choose the appropriate asymptotic notation. If you want to indicate an asymptotically tight bound, use Θ-notation.

We typically use asymptotic notation to provide the simplest and most precise bounds possible. For example, if an algorithm has a running time of $3n^2 + 20n$ in all cases, we use asymptotic notation to write that its running time is $\Theta(n^2)$. Strictly speaking, we are also correct in writing that the running time is $O(n^3)$ or $\Theta(3n^2 + 20n)$. Neither of these expressions is as useful as writing $\Theta(n^2)$ in this case, however: $O(n^3)$ is less precise than $\Theta(n^2)$ if the running time is $3n^2 + 20n$, and $\Theta(3n^2 + 20n)$ introduces complexity that obscures the order of growth. By writing the simplest and most precise bound, such as $\Theta(n^2)$, we can categorize and compare different algorithms. Throughout the book, you will see asymptotic running times that are almost always based on polynomials and logarithms: functions such as n, $n \lg^2 n$, $n^2 \lg n$, or $n^{1/2}$. You will also see some other functions, such as exponentials, $\lg \lg n$, and $\lg^* n$ (see Section 3.3). It is usually fairly easy to compare the rates of growth of these functions. Problem 3-3 gives you good practice.

Asymptotic notation in equations and inequalities

Although we formally define asymptotic notation in terms of sets, we use the equal sign ($=$) instead of the set membership sign ($\in$) within formulas. For example, we wrote that $4n^2 + 100n + 500 = O(n^2)$. We might also write $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. How do we interpret such formulas?

When the asymptotic notation stands alone (that is, not within a larger formula) on the right-hand side of an equation (or inequality), as in $4n^2 + 100n + 500 = O(n^2)$, the equal sign means set membership: $4n^2 + 100n + 500 \in O(n^2)$. In general, however, when asymptotic notation appears in a formula, we interpret it as standing for some anonymous function that we do not care to name. For example, the formula $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that $2n^2 + 3n + 1 = 2n^2 + f(n)$, where $f(n) \in \Theta(n)$. In this case, we let $f(n) = 3n + 1$, which indeed belongs to $\Theta(n)$.

Using asymptotic notation in this manner can help eliminate inessential detail and clutter in an equation. For example, in Chapter 2 we expressed the worst-case running time of merge sort as the recurrence

$$T(n) = 2T(n/2) + \Theta(n) .$$

If we are interested only in the asymptotic behavior of $T(n)$, there is no point in specifying all the lower-order terms exactly, because they are all understood to be included in the anonymous function denoted by the term $\Theta(n)$.

The number of anonymous functions in an expression is understood to be equal to the number of times the asymptotic notation appears. For example, in the expression

$$\sum_{i=1}^{n} O(i) ,$$

there is only a single anonymous function (a function of i). This expression is thus *not* the same as $O(1) + O(2) + \cdots + O(n)$, which doesn't really have a clean interpretation.

In some cases, asymptotic notation appears on the left-hand side of an equation, as in

$$2n^2 + \Theta(n) = \Theta(n^2) .$$

Interpret such equations using the following rule: *No matter how the anonymous functions are chosen on the left of the equal sign, there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid*. Thus, our example means that for *any* function $f(n) \in \Theta(n)$, there is *some* function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$ for all n. In other words, the right-hand side of an equation provides a coarser level of detail than the left-hand side.

We can chain together a number of such relationships, as in

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) \, . \end{aligned}$$

By the rules above, interpret each equation separately. The first equation says that there is *some* function $f(n) \in \Theta(n)$ such that $2n^2 + 3n + 1 = 2n^2 + f(n)$ for all n. The second equation says that for *any* function $g(n) \in \Theta(n)$ (such as the $f(n)$ just mentioned), there is *some* function $h(n) \in \Theta(n^2)$ such that $2n^2 + g(n) = h(n)$ for all n. This interpretation implies that $2n^2 + 3n + 1 = \Theta(n^2)$, which is what the chaining of equations intuitively says.

Proper abuses of asymptotic notation

Besides the abuse of equality to mean set membership, which we now see has a precise mathematical interpretation, another abuse of asymptotic notation occurs when the variable tending toward ∞ must be inferred from context. For example, when we say $O(g(n))$, we can assume that we're interested in the growth of $g(n)$ as n grows, and if we say $O(g(m))$ we're talking about the growth of $g(m)$ as m grows. The free variable in the expression indicates what variable is going to ∞.

The most common situation requiring contextual knowledge of which variable tends to ∞ occurs when the function inside the asymptotic notation is a constant, as in the expression $O(1)$. We cannot infer from the expression which variable is going to ∞, because no variable appears there. The context must disambiguate. For example, if the equation using asymptotic notation is $f(n) = O(1)$, it's apparent that the variable we're interested in is n. Knowing from context that the variable of interest is n, however, allows us to make perfect sense of the expression by using the formal definition of O-notation: the expression $f(n) = O(1)$ means that the function $f(n)$ is bounded from above by a constant as n goes to ∞. Technically, it might be less ambiguous if we explicitly indicated the variable tending to ∞ in the asymptotic notation itself, but that would clutter the notation. Instead, we simply ensure that the context makes it clear which variable (or variables) tend to ∞.

When the function inside the asymptotic notation is bounded by a positive constant, as in $T(n) = O(1)$, we often abuse asymptotic notation in yet another way, especially when stating recurrences. We may write something like $T(n) = O(1)$ for $n < 3$. According to the formal definition of O-notation, this statement is meaningless, because the definition only says that $T(n)$ is bounded above by a positive constant c for $n \geq n_0$ for some $n_0 > 0$. The value of $T(n)$ for $n < n_0$ need not be so bounded. Thus, in the example $T(n) = O(1)$ for $n < 3$, we cannot infer any constraint on $T(n)$ when $n < 3$, because it might be that $n_0 > 3$.

What is conventionally meant when we say $T(n) = O(1)$ for $n < 3$ is that there exists a positive constant c such that $T(n) \leq c$ for $n < 3$. This convention saves

us the trouble of naming the bounding constant, allowing it to remain anonymous while we focus on more important variables in an analysis. Similar abuses occur with the other asymptotic notations. For example, $T(n) = \Theta(1)$ for $n < 3$ means that $T(n)$ is bounded above and below by positive constants when $n < 3$.

Occasionally, the function describing an algorithm's running time may not be defined for certain input sizes, for example, when an algorithm assumes that the input size is an exact power of 2. We still use asymptotic notation to describe the growth of the running time, understanding that any constraints apply only when the function is defined. For example, suppose that $f(n)$ is defined only on a subset of the natural or nonnegative real numbers. Then $f(n) = O(g(n))$ means that the bound $0 \le T(n) \le cg(n)$ in the definition of O-notation holds for all $n \ge n_0$ over the domain of $f(n)$, that is, where $f(n)$ is defined. This abuse is rarely pointed out, since what is meant is generally clear from context.

In mathematics, it's okay—and often desirable—to abuse a notation, as long as we don't misuse it. If we understand precisely what is meant by the abuse and don't draw incorrect conclusions, it can simplify our mathematical language, contribute to our higher-level understanding, and help us focus on what really matters.

o-notation

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is not asymptotically tight. We formally define $o(g(n))$ ("little-oh of g of n") as the set

$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0\} .$$

For example, $2n = o(n^2)$, but $2n^2 \ne o(n^2)$.

The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \le f(n) \le cg(n)$ holds for *some* constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \le f(n) < cg(n)$ holds for *all* constants $c > 0$. Intuitively, in o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n gets large:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 .$$

Some authors use this limit as a definition of the o-notation, but the definition in this book also restricts the anonymous functions to be asymptotically nonnegative.

ω-notation

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically tight. One way to define it is by

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.

Formally, however, we define $\omega(g(n))$ ("little-omega of g of n") as the set

$$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\} .$$

Where the definition of o-notation says that $f(n) < cg(n)$, the definition of ω-notation says the opposite: that $cg(n) < f(n)$. For examples of ω-notation, we have $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$. The relation $f(n) = \omega(g(n))$ implies that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty ,$$

if the limit exists. That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n gets large.

Comparing functions

Many of the relational properties of real numbers apply to asymptotic comparisons as well. For the following, assume that $f(n)$ and $g(n)$ are asymptotically positive.

Transitivity:

$$\begin{aligned}
&f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \quad \text{imply} \quad f(n) = \Theta(h(n)) ,\\
&f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \quad \text{imply} \quad f(n) = O(h(n)) ,\\
&f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \quad \text{imply} \quad f(n) = \Omega(h(n)) ,\\
&f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) \quad \text{imply} \quad f(n) = o(h(n)) ,\\
&f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \quad \text{imply} \quad f(n) = \omega(h(n)) .
\end{aligned}$$

Reflexivity:

$$\begin{aligned}
&f(n) = \Theta(f(n)) ,\\
&f(n) = O(f(n)) ,\\
&f(n) = \Omega(f(n)) .
\end{aligned}$$

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

Transpose symmetry:

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)),$$
$$f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n)).$$

Because these properties hold for asymptotic notations, we can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b:

$$f(n) = O(g(n)) \text{ is like } a \leq b,$$
$$f(n) = \Omega(g(n)) \text{ is like } a \geq b,$$
$$f(n) = \Theta(g(n)) \text{ is like } a = b,$$
$$f(n) = o(g(n)) \text{ is like } a < b,$$
$$f(n) = \omega(g(n)) \text{ is like } a > b.$$

We say that $f(n)$ is ***asymptotically smaller*** than $g(n)$ if $f(n) = o(g(n))$, and $f(n)$ is ***asymptotically larger*** than $g(n)$ if $f(n) = \omega(g(n))$.

One property of real numbers, however, does not carry over to asymptotic notation:

Trichotomy: For any two real numbers a and b, exactly one of the following must hold: $a < b$, $a = b$, or $a > b$.

Although any two real numbers can be compared, not all functions are asymptotically comparable. That is, for two functions $f(n)$ and $g(n)$, it may be the case that neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds. For example, we cannot compare the functions n and $n^{1+\sin n}$ using asymptotic notation, since the value of the exponent in $n^{1+\sin n}$ oscillates between 0 and 2, taking on all values in between.

Exercises

3.2-1

Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ-notation, prove that $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$.

3.2-2

Explain why the statement, "The running time of algorithm A is at least $O(n^2)$," is meaningless.

3.2-3

Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

3.2-4

Prove Theorem 3.1.

3.2-5
Prove that the running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.

3.2-6
Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

3.2-7
We can extend our notation to the case of two parameters n and m that can go to ∞ independently at different rates. For a given function $g(n,m)$, we denote by $O(g(n,m))$ the set of functions

$$O(g(n,m)) = \{f(n,m) : \text{there exist positive constants } c, n_0, \text{ and } m_0 \text{ such that } 0 \le f(n,m) \le cg(n,m) \text{ for all } n \ge n_0 \text{ or } m \ge m_0\} .$$

Give corresponding definitions for $\Omega(g(n,m))$ and $\Theta(g(n,m))$.

3.3 Standard notations and common functions

This section reviews some standard mathematical functions and notations and explores the relationships among them. It also illustrates the use of the asymptotic notations.

Monotonicity

A function $f(n)$ is ***monotonically increasing*** if $m \le n$ implies $f(m) \le f(n)$. Similarly, it is ***monotonically decreasing*** if $m \le n$ implies $f(m) \ge f(n)$. A function $f(n)$ is ***strictly increasing*** if $m < n$ implies $f(m) < f(n)$ and ***strictly decreasing*** if $m < n$ implies $f(m) > f(n)$.

Floors and ceilings

For any real number x, we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (read "the floor of x") and the least integer greater than or equal to x by $\lceil x \rceil$ (read "the ceiling of x"). The floor function is monotonically increasing, as is the ceiling function.

Floors and ceilings obey the following properties. For any integer n, we have

$$\lfloor n \rfloor = n = \lceil n \rceil . \tag{3.1}$$

For all real x, we have