2

Mathematical Preliminaries

This chapter presents mathematical notation, background, and techniques used throughout the book. This material is provided primarily for review and reference. You might wish to return to the relevant sections when you encounter unfamiliar notation or mathematical techniques in later chapters.

Section 2.7 on estimation might be unfamiliar to many readers. Estimation is not a mathematical technique, but rather a general engineering skill. It is enormously useful to computer scientists doing design work, because any proposed solution whose estimated resource requirements fall well outside the problem's resource constraints can be discarded immediately, allowing time for greater analysis of more promising solutions.

2.1 Sets and Relations

The concept of a set in the mathematical sense has wide application in computer science. The notations and techniques of set theory are commonly used when describing and implementing algorithms because the abstractions associated with sets often help to clarify and simplify algorithm design.

A **set** is a collection of distinguishable **members** or **elements**. The members are typically drawn from some larger population known as the **base type**. Each member of a set is either a **primitive element** of the base type or is a set itself. There is no concept of duplication in a set. Each value from the base type is either in the set or not in the set. For example, a set named **P** might consist of the three integers 7, 11, and 42. In this case, **P**'s members are 7, 11, and 42, and the base type is integer.

Figure 2.1 shows the symbols commonly used to express sets and their relationships. Here are some examples of this notation in use. First define two sets, **P** and **Q**.

$$\mathbf{P} = \{2, 3, 5\}, \qquad \mathbf{Q} = \{5, 10\}.$$

$\{1, 4\}$	A set composed of the members 1 and 4
$\{x \mid x$ is a positive integer$\}$	A set definition using a **set former** Example: the set of all positive integers
$x \in \mathbf{P}$	x is a member of set **P**
$x \notin \mathbf{P}$	x is not a member of set **P**
$\emptyset$	The null or empty set
$\lvert\mathbf{P}\rvert$	Cardinality: size of set **P** or number of members for set **P**
$\mathbf{P} \subseteq \mathbf{Q}$, $\mathbf{Q} \supseteq \mathbf{P}$	Set **P** is included in set **Q**, set **P** is a subset of set **Q**, set **Q** is a superset of set **P**
$\mathbf{P} \cup \mathbf{Q}$	Set Union: all elements appearing in **P** OR **Q**
$\mathbf{P} \cap \mathbf{Q}$	Set Intersection: all elements appearing in **P** AND **Q**
$\mathbf{P} - \mathbf{Q}$	Set difference: all elements of set **P** NOT in set **Q**

Figure 2.1 Set notation.

$|\mathbf{P}| = 3$ (because **P** has three members) and $|\mathbf{Q}| = 2$ (because **Q** has two members). The union of **P** and **Q**, written $\mathbf{P} \cup \mathbf{Q}$, is the set of elements in either **P** or **Q**, which is $\{2, 3, 5, 10\}$. The intersection of **P** and **Q**, written $\mathbf{P} \cap \mathbf{Q}$, is the set of elements that appear in both **P** and **Q**, which is $\{5\}$. The set difference of **P** and **Q**, written $\mathbf{P} - \mathbf{Q}$, is the set of elements that occur in **P** but not in **Q**, which is $\{2, 3\}$. Note that $\mathbf{P} \cup \mathbf{Q} = \mathbf{Q} \cup \mathbf{P}$ and that $\mathbf{P} \cap \mathbf{Q} = \mathbf{Q} \cap \mathbf{P}$, but in general $\mathbf{P} - \mathbf{Q} \neq \mathbf{Q} - \mathbf{P}$. In this example, $\mathbf{Q} - \mathbf{P} = \{10\}$. Note that the set $\{4, 3, 5\}$ is indistinguishable from set **P**, because sets have no concept of order. Likewise, set $\{4, 3, 4, 5\}$ is also indistinguishable from **P**, because sets have no concept of duplicate elements.

The **powerset** of a set **S** is the set of all possible subsets for **S**. Consider the set $\mathbf{S} = \{a, b, c\}$. The powerset of **S** is

$$\{\emptyset,\ \{a\},\ \{b\},\ \{c\},\ \{a, b\},\ \{a, c\},\ \{b, c\},\ \{a, b, c\}\}.$$

A collection of elements with no order (like a set), but with duplicate-valued elements is called a **bag**.[1] To distinguish bags from sets, I use square brackets [] around a bag's elements. For example, bag [3, 4, 5, 4] is distinct from bag [3, 4, 5], while set $\{3, 4, 5, 4\}$ is indistinguishable from set $\{3, 4, 5\}$. However, bag [3, 4, 5, 4] is indistinguishable from bag [3, 4, 4, 5].

[1]The object referred to here as a bag is sometimes called a **multilist**. But, I reserve the term multilist for a list that may contain sublists (see Section 12.1).

A **sequence** is a collection of elements with an order, and which may contain duplicate-valued elements. A sequence is also sometimes called a **tuple** or a **vector**. In a sequence, there is a 0th element, a 1st element, 2nd element, and so on. I indicate a sequence by using angle brackets $\langle\rangle$ to enclose its elements. For example, $\langle 3, 4, 5, 4\rangle$ is a sequence. Note that sequence $\langle 3, 5, 4, 4\rangle$ is distinct from sequence $\langle 3, 4, 5, 4\rangle$, and both are distinct from sequence $\langle 3, 4, 5\rangle$.

A **relation** R over set **S** is a set of ordered pairs from **S**. As an example of a relation, if **S** is $\{a, b, c\}$, then

$$\{\langle a, c\rangle, \langle b, c\rangle, \langle c, b\rangle\}$$

is a relation, and

$$\{\langle a, a\rangle, \langle a, c\rangle, \langle b, b\rangle, \langle b, c\rangle, \langle c, c\rangle\}$$

is a different relation. If tuple $\langle x, y\rangle$ is in relation R, we may use the infix notation xRy. We often use relations such as the less than operator ($<$) on the natural numbers, which includes ordered pairs such as $\langle 1, 3\rangle$ and $\langle 2, 23\rangle$, but not $\langle 3, 2\rangle$ or $\langle 2, 2\rangle$. Rather than writing the relationship in terms of ordered pairs, we typically use an infix notation for such relations, writing $1 < 3$.

Define the properties of relations as follows, with R a binary relation over set **S**.

- R is **reflexive** if aRa for all $a \in$ **S**.
- R is **symmetric** if whenever aRb, then bRa, for all $a, b \in$ **S**.
- R is **antisymmetric** if whenever aRb and bRa, then $a = b$, for all $a, b \in$ **S**.
- R is **transitive** if whenever aRb and bRc, then aRc, for all $a, b, c \in$ **S**.

As examples, for the natural numbers, $<$ is antisymmetric (because there is no case where aRb and bRa) and transitive; $\leq$ is reflexive, antisymmetric, and transitive, and $=$ is reflexive, symmetric (and antisymmetric!), and transitive. For people, the relation "is a sibling of" is symmetric and transitive. If we define a person to be a sibling of himself, then it is reflexive; if we define a person not to be a sibling of himself, then it is not reflexive.

R is an **equivalence relation** on set **S** if it is reflexive, symmetric, and transitive. An equivalence relation can be used to partition a set into **equivalence classes**. If two elements a and b are equivalent to each other, we write $a \equiv b$. A **partition** of a set **S** is a collection of subsets that are disjoint from each other and whose union is **S**. An equivalence relation on set **S** partitions the set into subsets whose elements are equivalent. See Section 6.2 for a discussion on how to represent equivalence classes on a set. One application for disjoint sets appears in Section 11.5.2.

Example 2.1 For the integers, $=$ is an equivalence relation that partitions each element into a distinct subset. In other words, for any integer a, three things are true.

1. $a = a$,

2. if $a = b$ then $b = a$, and
3. if $a = b$ and $b = c$, then $a = c$.

Of course, for distinct integers a, b, and c there are never cases where $a = b$, $b = a$, or $b = c$. So the claims that $=$ is symmetric and transitive are vacuously true (there are never examples in the relation where these events occur). But because the requirements for symmetry and transitivity are not violated, the relation is symmetric and transitive.

Example 2.2 If we clarify the definition of sibling to mean that a person is a sibling of him- or herself, then the sibling relation is an equivalence relation that partitions the set of people.

Example 2.3 We can use the modulus function (defined in the next section) to define an equivalence relation. For the set of integers, use the modulus function to define a binary relation such that two numbers x and y are in the relation if and only if $x \bmod m = y \bmod m$. Thus, for $m = 4$, $\langle 1, 5\rangle$ is in the relation because $1 \bmod 4 = 5 \bmod 4$. We see that modulus used in this way defines an equivalence relation on the integers, and this relation can be used to partition the integers into m equivalence classes. This relation is an equivalence relation because

1. $x \bmod m = x \bmod m$ for all x;
2. if $x \bmod m = y \bmod m$, then $y \bmod m = x \bmod m$; and
3. if $x \bmod m = y \bmod m$ and $y \bmod m = z \bmod m$, then $x \bmod m = z \bmod m$.

A binary relation is called a **partial order** if it is antisymmetric and transitive.[2] The set on which the partial order is defined is called a **partially ordered set** or a **poset**. Elements x and y of a set are **comparable** under a given relation if either xRy or yRx. If every pair of distinct elements in a partial order are comparable, then the order is called a **total order** or **linear order**.

Example 2.4 For the integers, relations $<$ and $\leq$ define partial orders. Operation $<$ is a total order because, for every pair of integers x and y such that $x \neq y$, either $x < y$ or $y < x$. Likewise, $\leq$ is a total order because, for every pair of integers x and y such that $x \neq y$, either $x \leq y$ or $y \leq x$.

---

[2] Not all authors use this definition for partial order. I have seen at least three significantly different definitions in the literature. I have selected the one that lets $<$ and $\leq$ both define partial orders on the integers, because this seems the most natural to me.

Example 2.5 For the powerset of the integers, the subset operator defines a partial order (because it is antisymmetric and transitive). For example, $\{1, 2\} \subseteq \{1, 2, 3\}$. However, sets $\{1, 2\}$ and $\{1, 3\}$ are not comparable by the subset operator, because neither is a subset of the other. Therefore, the subset operator does not define a total order on the powerset of the integers.

2.2 Miscellaneous Notation

Units of measure: I use the following notation for units of measure. "B" will be used as an abbreviation for bytes, "b" for bits, "KB" for kilobytes ($2^{10} = 1024$ bytes), "MB" for megabytes (2^{20} bytes), "GB" for gigabytes (2^{30} bytes), and "ms" for milliseconds (a millisecond is $\frac{1}{1000}$ of a second). Spaces are not placed between the number and the unit abbreviation when a power of two is intended. Thus a disk drive of size 25 gigabytes (where a gigabyte is intended as 2^{30} bytes) will be written as "25GB." Spaces are used when a decimal value is intended. An amount of 2000 bits would therefore be written "2 Kb" while "2Kb" represents 2048 bits. 2000 milliseconds is written as 2000 ms. Note that in this book large amounts of storage are nearly always measured in powers of two and times in powers of ten.

Factorial function: The **factorial** function, written $n!$ for n an integer greater than 0, is the product of the integers between 1 and n, inclusive. Thus, $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. As a special case, $0! = 1$. The factorial function grows quickly as n becomes larger. Because computing the factorial function directly is a time-consuming process, it can be useful to have an equation that provides a good approximation. Stirling's approximation states that $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$, where $e \approx 2.71828$ (e is the base for the system of natural logarithms).[3] Thus we see that while $n!$ grows slower than n^n (because $\sqrt{2\pi n}/e^n < 1$), it grows faster than c^n for any positive integer constant c.

Permutations: A **permutation** of a sequence **S** is simply the members of **S** arranged in some order. For example, a permutation of the integers 1 through n would be those values arranged in some order. If the sequence contains n distinct members, then there are $n!$ different permutations for the sequence. This is because there are n choices for the first member in the permutation; for each choice of first member there are $n - 1$ choices for the second member, and so on. Sometimes one would like to obtain a **random permutation** for a sequence, that is, one of the $n!$ possible permutations is selected in such a way that each permutation has equal probability of being selected. A simple **C**++ function for generating a random permutation is as follows. Here, the n values of the sequence are stored in positions 0

[3] The symbol "≈" means "approximately equal."

through $n-1$ of array **A**, function **swap(A, i, j)** exchanges elements **i** and **j** in array **A**, and **Random(n)** returns an integer value in the range 0 to $n-1$ (see the Appendix for more information on **swap** and **Random**).

```
// Randomly permute the "n" values of array "A"
template<typename E>
void permute(E A[], int n) {
  for (int i=n; i>0; i--)
    swap(A, i-1, Random(i));
}
```

Boolean variables: A **Boolean variable** is a variable (of type **bool** in **C**++) that takes on one of the two values **true** and **false**. These two values are often associated with the values 1 and 0, respectively, although there is no reason why this needs to be the case. It is poor programming practice to rely on the correspondence between 0 and **false**, because these are logically distinct objects of different types.

Logic Notation: We will occasionally make use of the notation of symbolic or Boolean logic. $A \Rightarrow B$ means "A implies B" or "If A then B." $A \Leftrightarrow B$ means "A if and only if B" or "A is equivalent to B." $A \vee B$ means "A or B" (useful both in the context of symbolic logic or when performing a Boolean operation). $A \wedge B$ means "A and B." $\sim A$ and $\overline{A}$ both mean "not A" or the negation of A where A is a Boolean variable.

Floor and ceiling: The **floor** of x (written $\lfloor x \rfloor$) takes real value x and returns the greatest integer $\leq x$. For example, $\lfloor 3.4 \rfloor = 3$, as does $\lfloor 3.0 \rfloor$, while $\lfloor -3.4 \rfloor = -4$ and $\lfloor -3.0 \rfloor = -3$. The **ceiling** of x (written $\lceil x \rceil$) takes real value x and returns the least integer $\geq x$. For example, $\lceil 3.4 \rceil = 4$, as does $\lceil 4.0 \rceil$, while $\lceil -3.4 \rceil = \lceil -3.0 \rceil = -3$.

Modulus operator: The **modulus** (or **mod**) function returns the remainder of an integer division. Sometimes written $n \bmod m$ in mathematical expressions, the syntax for the **C**++ modulus operator is **n % m**. From the definition of remainder, $n \bmod m$ is the integer r such that $n = qm + r$ for q an integer, and $|r| < |m|$. Therefore, the result of $n \bmod m$ must be between 0 and $m-1$ when n and m are positive integers. For example, $5 \bmod 3 = 2$; $25 \bmod 3 = 1$, $5 \bmod 7 = 5$, and $5 \bmod 5 = 0$.

There is more than one way to assign values to q and r, depending on how integer division is interpreted. The most common mathematical definition computes the mod function as $n \bmod m = n - m\lfloor n/m \rfloor$. In this case, $-3 \bmod 5 = 2$. However, Java and **C**++ compilers typically use the underlying processor's machine instruction for computing integer arithmetic. On many computers this is done by truncating the resulting fraction, meaning $n \bmod m = n - m(\text{trunc}(n/m))$. Under this definition, $-3 \bmod 5 = -3$.

Unfortunately, for many applications this is not what the user wants or expects. For example, many hash systems will perform some computation on a record's key value and then take the result modulo the hash table size. The expectation here would be that the result is a legal index into the hash table, not a negative number. Implementers of hash functions must either insure that the result of the computation is always positive, or else add the hash table size to the result of the modulo function when that result is negative.

2.3 Logarithms

A **logarithm** of base b for value y is the power to which b is raised to get y. Normally, this is written as $\log_b y = x$. Thus, if $\log_b y = x$ then $b^x = y$, and $b^{log_b y} = y$.

Logarithms are used frequently by programmers. Here are two typical uses.

Example 2.6 Many programs require an encoding for a collection of objects. What is the minimum number of bits needed to represent n distinct code values? The answer is $\lceil \log_2 n \rceil$ bits. For example, if you have 1000 codes to store, you will require at least $\lceil \log_2 1000 \rceil = 10$ bits to have 1000 different codes (10 bits provide 1024 distinct code values).

Example 2.7 Consider the binary search algorithm for finding a given value within an array sorted by value from lowest to highest. Binary search first looks at the middle element and determines if the value being searched for is in the upper half or the lower half of the array. The algorithm then continues splitting the appropriate subarray in half until the desired value is found. (Binary search is described in more detail in Section 3.5.) How many times can an array of size n be split in half until only one element remains in the final subarray? The answer is $\lceil \log_2 n \rceil$ times.

In this book, nearly all logarithms used have a base of two. This is because data structures and algorithms most often divide things in half, or store codes with binary bits. Whenever you see the notation $\log n$ in this book, either $\log_2 n$ is meant or else the term is being used asymptotically and so the actual base does not matter. Logarithms using any base other than two will show the base explicitly.

Logarithms have the following properties, for any positive values of m, n, and r, and any positive integers a and b.

1. $\log(nm) = \log n + \log m$.
2. $\log(n/m) = \log n - \log m$.
3. $\log(n^r) = r \log n$.
4. $\log_a n = \log_b n / \log_b a$.

The first two properties state that the logarithm of two numbers multiplied (or divided) can be found by adding (or subtracting) the logarithms of the two numbers.[4] Property (3) is simply an extension of property (1). Property (4) tells us that, for variable n and any two integer constants a and b, $\log_a n$ and $\log_b n$ differ by the constant factor $\log_b a$, regardless of the value of n. Most runtime analyses in this book are of a type that ignores constant factors in costs. Property (4) says that such analyses need not be concerned with the base of the logarithm, because this can change the total cost only by a constant factor. Note that $2^{\log n} = n$.

When discussing logarithms, exponents often lead to confusion. Property (3) tells us that $\log n^2 = 2 \log n$. How do we indicate the square of the logarithm (as opposed to the logarithm of n^2)? This could be written as $(\log n)^2$, but it is traditional to use $\log^2 n$. On the other hand, we might want to take the logarithm of the logarithm of n. This is written $\log \log n$.

A special notation is used in the rare case when we need to know how many times we must take the log of a number before we reach a value ≤ 1. This quantity is written $\log^* n$. For example, $\log^* 1024 = 4$ because $\log 1024 = 10$, $\log 10 \approx 3.33$, $\log 3.33 \approx 1.74$, and $\log 1.74 < 1$, which is a total of 4 log operations.

2.4 Summations and Recurrences

Most programs contain loop constructs. When analyzing running time costs for programs with loops, we need to add up the costs for each time the loop is executed. This is an example of a **summation**. Summations are simply the sum of costs for some function applied to a range of parameter values. Summations are typically written with the following "Sigma" notation:

$$\sum_{i=1}^{n} f(i).$$

This notation indicates that we are summing the value of $f(i)$ over some range of (integer) values. The parameter to the expression and its initial value are indicated below the $\sum$ symbol. Here, the notation $i = 1$ indicates that the parameter is i and that it begins with the value 1. At the top of the $\sum$ symbol is the expression n. This indicates the maximum value for the parameter i. Thus, this notation means to sum the values of $f(i)$ as i ranges across the integers from 1 through n. This can also be

[4] These properties are the idea behind the slide rule. Adding two numbers can be viewed as joining two lengths together and measuring their combined length. Multiplication is not so easily done. However, if the numbers are first converted to the lengths of their logarithms, then those lengths can be added and the inverse logarithm of the resulting length gives the answer for the multiplication (this is simply logarithm property (1)). A slide rule measures the length of the logarithm for the numbers, lets you slide bars representing these lengths to add up the total length, and finally converts this total length to the correct numeric answer by taking the inverse of the logarithm for the result.

written $f(1) + f(2) + \cdots + f(n-1) + f(n)$. Within a sentence, Sigma notation is typeset as $\sum_{i=1}^{n} f(i)$.

Given a summation, you often wish to replace it with an algebraic equation with the same value as the summation. This is known as a **closed-form solution**, and the process of replacing the summation with its closed-form solution is known as **solving** the summation. For example, the summation $\sum_{i=1}^{n} 1$ is simply the expression "1" summed n times (remember that i ranges from 1 to n). Because the sum of n 1s is n, the closed-form solution is n. The following is a list of useful summations, along with their closed-form solutions.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \tag{2.1}$$

$$\sum_{i=1}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6} = \frac{n(2n+1)(n+1)}{6}. \tag{2.2}$$

$$\sum_{i=1}^{\log n} n = n \log n. \tag{2.3}$$

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \text{ for } 0 < a < 1. \tag{2.4}$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1} \text{ for } a \neq 1. \tag{2.5}$$

As special cases to Equation 2.5,

$$\sum_{i=1}^{n} \frac{1}{2^i} = 1 - \frac{1}{2^n}, \tag{2.6}$$

and

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1. \tag{2.7}$$

As a corollary to Equation 2.7,

$$\sum_{i=0}^{\log n} 2^i = 2^{\log n + 1} - 1 = 2n - 1. \tag{2.8}$$

Finally,

$$\sum_{i=1}^{n} \frac{i}{2^i} = 2 - \frac{n+2}{2^n}. \tag{2.9}$$

The sum of reciprocals from 1 to n, called the **Harmonic Series** and written $\mathcal{H}_n$, has a value between $\log_e n$ and $\log_e n + 1$. To be more precise, as n grows, the

summation grows closer to

$$\mathcal{H}_n \approx \log_e n + \gamma + \frac{1}{2n}, \tag{2.10}$$

where γ is Euler's constant and has the value 0.5772...

Most of these equalities can be proved easily by mathematical induction (see Section 2.6.3). Unfortunately, induction does not help us derive a closed-form solution. It only confirms when a proposed closed-form solution is correct. Techniques for deriving closed-form solutions are discussed in Section 14.1.

The running time for a recursive algorithm is most easily expressed by a recursive expression because the total time for the recursive algorithm includes the time to run the recursive call(s). A **recurrence relation** defines a function by means of an expression that includes one or more (smaller) instances of itself. A classic example is the recursive definition for the factorial function:

$$n! = (n-1)! \cdot n \text{ for } n > 1; \quad 1! = 0! = 1.$$

Another standard example of a recurrence is the Fibonacci sequence:

$$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2) \text{ for } n > 2; \quad \text{Fib}(1) = \text{Fib}(2) = 1.$$

From this definition, the first seven numbers of the Fibonacci sequence are

$$1, 1, 2, 3, 5, 8, \text{ and } 13.$$

Notice that this definition contains two parts: the general definition for $\text{Fib}(n)$ and the base cases for $\text{Fib}(1)$ and $\text{Fib}(2)$. Likewise, the definition for factorial contains a recursive part and base cases.

Recurrence relations are often used to model the cost of recursive functions. For example, the number of multiplications required by function **`fact`** of Section 2.5 for an input of size n will be zero when $n = 0$ or $n = 1$ (the base cases), and it will be one plus the cost of calling **`fact`** on a value of $n - 1$. This can be defined using the following recurrence:

$$\mathbf{T}(n) = \mathbf{T}(n-1) + 1 \text{ for } n > 1; \quad \mathbf{T}(0) = \mathbf{T}(1) = 0.$$

As with summations, we typically wish to replace the recurrence relation with a closed-form solution. One approach is to **expand** the recurrence by replacing any occurrences of **T** on the right-hand side with its definition.

Example 2.8 If we expand the recurrence $\mathbf{T}(n) = \mathbf{T}(n-1) + 1$, we get

$$\begin{aligned} \mathbf{T}(n) &= \mathbf{T}(n-1) + 1 \\ &= (\mathbf{T}(n-2) + 1) + 1. \end{aligned}$$

We can expand the recurrence as many steps as we like, but the goal is to detect some pattern that will permit us to rewrite the recurrence in terms of a summation. In this example, we might notice that

$$(\mathbf{T}(n-2)+1)+1 = \mathbf{T}(n-2)+2$$

and if we expand the recurrence again, we get

$$\mathbf{T}(n) = \mathbf{T}(n-2)+2 = \mathbf{T}(n-3)+1+2 = \mathbf{T}(n-3)+3$$

which generalizes to the pattern $\mathbf{T}(n) = \mathbf{T}(n-i)+i$. We might conclude that

$$\begin{aligned}\mathbf{T}(n) &= \mathbf{T}(n-(n-1))+(n-1)\\ &= \mathbf{T}(1)+n-1\\ &= n-1.\end{aligned}$$

Because we have merely guessed at a pattern and not actually proved that this is the correct closed form solution, we should use an induction proof to complete the process (see Example 2.13).

Example 2.9 A slightly more complicated recurrence is

$$\mathbf{T}(n) = \mathbf{T}(n-1)+n; \quad T(1) = 1.$$

Expanding this recurrence a few steps, we get

$$\begin{aligned}\mathbf{T}(n) &= \mathbf{T}(n-1)+n\\ &= \mathbf{T}(n-2)+(n-1)+n\\ &= \mathbf{T}(n-3)+(n-2)+(n-1)+n.\end{aligned}$$

We should then observe that this recurrence appears to have a pattern that leads to

$$\begin{aligned}\mathbf{T}(n) &= \mathbf{T}(n-(n-1))+(n-(n-2))+\cdots+(n-1)+n\\ &= 1+2+\cdots+(n-1)+n.\end{aligned}$$

This is equivalent to the summation $\sum_{i=1}^{n} i$, for which we already know the closed-form solution.

Techniques to find closed-form solutions for recurrence relations are discussed in Section 14.2. Prior to Chapter 14, recurrence relations are used infrequently in this book, and the corresponding closed-form solution and an explanation for how it was derived will be supplied at the time of use.

2.5 Recursion

An algorithm is **recursive** if it calls itself to do part of its work. For this approach to be successful, the "call to itself" must be on a smaller problem then the one originally attempted. In general, a recursive algorithm must have two parts: the **base case**, which handles a simple input that can be solved without resorting to a recursive call, and the recursive part which contains one or more recursive calls to the algorithm where the parameters are in some sense "closer" to the base case than those of the original call. Here is a recursive C++ function to compute the factorial of n. A trace of **fact**'s execution for a small value of n is presented in Section 4.2.4.

```
long fact(int n) {       // Compute n! recursively
  // To fit n! into a long variable, we require n <= 12
  Assert((n >= 0) && (n <= 12), "Input out of range");
  if (n <= 1)  return 1; // Base case: return base solution
  return n * fact(n-1);  // Recursive call for n > 1
}
```

The first two lines of the function constitute the base cases. If $n \leq 1$, then one of the base cases computes a solution for the problem. If $n > 1$, then **fact** calls a function that knows how to find the factorial of $n - 1$. Of course, the function that knows how to compute the factorial of $n - 1$ happens to be **fact** itself. But we should not think too hard about this while writing the algorithm. The design for recursive algorithms can always be approached in this way. First write the base cases. Then think about solving the problem by combining the results of one or more smaller — but similar — subproblems. If the algorithm you write is correct, then certainly you can rely on it (recursively) to solve the smaller subproblems. The secret to success is: Do not worry about *how* the recursive call solves the subproblem. Simply accept that it *will* solve it correctly, and use this result to in turn correctly solve the original problem. What could be simpler?

Recursion has no counterpart in everyday, physical-world problem solving. The concept can be difficult to grasp because it requires you to think about problems in a new way. To use recursion effectively, it is necessary to train yourself to stop analyzing the recursive process beyond the recursive call. The subproblems will take care of themselves. You just worry about the base cases and how to recombine the subproblems.

The recursive version of the factorial function might seem unnecessarily complicated to you because the same effect can be achieved by using a **while** loop. Here is another example of recursion, based on a famous puzzle called "Towers of Hanoi." The natural algorithm to solve this problem has multiple recursive calls. It cannot be rewritten easily using **while** loops.

The Towers of Hanoi puzzle begins with three poles and n rings, where all rings start on the leftmost pole (labeled Pole 1). The rings each have a different size, and

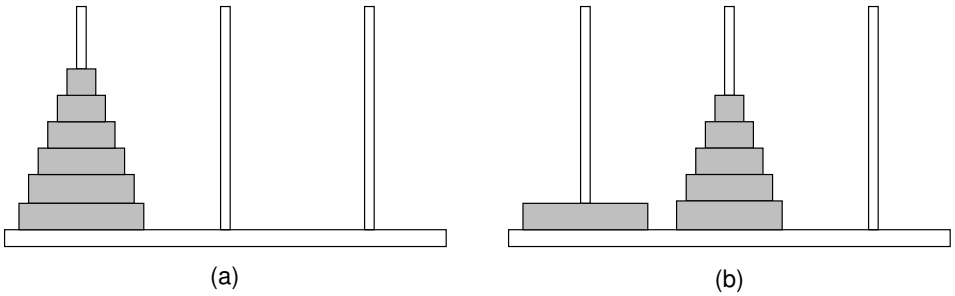

Figure 2.2 Towers of Hanoi example. (a) The initial conditions for a problem with six rings. (b) A necessary intermediate step on the road to a solution.

are stacked in order of decreasing size with the largest ring at the bottom, as shown in Figure 2.2(a). The problem is to move the rings from the leftmost pole to the rightmost pole (labeled Pole 3) in a series of steps. At each step the top ring on some pole is moved to another pole. There is one limitation on where rings may be moved: A ring can never be moved on top of a smaller ring.

How can you solve this problem? It is easy if you don't think too hard about the details. Instead, consider that all rings are to be moved from Pole 1 to Pole 3. It is not possible to do this without first moving the bottom (largest) ring to Pole 3. To do that, Pole 3 must be empty, and only the bottom ring can be on Pole 1. The remaining $n - 1$ rings must be stacked up in order on Pole 2, as shown in Figure 2.2(b). How can you do this? Assume that a function X is available to solve the problem of moving the top $n - 1$ rings from Pole 1 to Pole 2. Then move the bottom ring from Pole 1 to Pole 3. Finally, again use function X to move the remaining $n - 1$ rings from Pole 2 to Pole 3. In both cases, "function X" is simply the Towers of Hanoi function called on a smaller version of the problem.

The secret to success is relying on the Towers of Hanoi algorithm to do the work for you. You need not be concerned about the gory details of *how* the Towers of Hanoi subproblem will be solved. That will take care of itself provided that two things are done. First, there must be a base case (what to do if there is only one ring) so that the recursive process will not go on forever. Second, the recursive call to Towers of Hanoi can only be used to solve a smaller problem, and then only one of the proper form (one that meets the original definition for the Towers of Hanoi problem, assuming appropriate renaming of the poles).

Here is an implementation for the recursive Towers of Hanoi algorithm. Function **`move(start, goal)`** takes the top ring from Pole **`start`** and moves it to Pole **`goal`**. If **`move`** were to print the values of its parameters, then the result of calling **`TOH`** would be a list of ring-moving instructions that solves the problem.

```
void TOH(int n, Pole start, Pole goal, Pole temp) {
  if (n == 0) return;           // Base case
  TOH(n-1, start, temp, goal); // Recursive call: n-1 rings
  move(start, goal);            // Move bottom disk to goal
  TOH(n-1, temp, goal, start); // Recursive call: n-1 rings
}
```

Those who are unfamiliar with recursion might find it hard to accept that it is used primarily as a tool for simplifying the design and description of algorithms. A recursive algorithm usually does not yield the most efficient computer program for solving the problem because recursion involves function calls, which are typically more expensive than other alternatives such as a **`while`** loop. However, the recursive approach usually provides an algorithm that is reasonably efficient in the sense discussed in Chapter 3. (But not always! See Exercise 2.11.) If necessary, the clear, recursive solution can later be modified to yield a faster implementation, as described in Section 4.2.4.

Many data structures are naturally recursive, in that they can be defined as being made up of self-similar parts. Tree structures are an example of this. Thus, the algorithms to manipulate such data structures are often presented recursively. Many searching and sorting algorithms are based on a strategy of **divide and conquer**. That is, a solution is found by breaking the problem into smaller (similar) subproblems, solving the subproblems, then combining the subproblem solutions to form the solution to the original problem. This process is often implemented using recursion. Thus, recursion plays an important role throughout this book, and many more examples of recursive functions will be given.

2.6 Mathematical Proof Techniques

Solving any problem has two distinct parts: the investigation and the argument. Students are too used to seeing only the argument in their textbooks and lectures. But to be successful in school (and in life after school), one needs to be good at both, and to understand the differences between these two phases of the process. To solve the problem, you must investigate successfully. That means engaging the problem, and working through until you find a solution. Then, to give the answer to your client (whether that "client" be your instructor when writing answers on a homework assignment or exam, or a written report to your boss), you need to be able to make the argument in a way that gets the solution across clearly and succinctly. The argument phase involves good technical writing skills — the ability to make a clear, logical argument.

Being conversant with standard proof techniques can help you in this process. Knowing how to write a good proof helps in many ways. First, it clarifies your thought process, which in turn clarifies your explanations. Second, if you use one of the standard proof structures such as proof by contradiction or an induction proof,

then both you and your reader are working from a shared understanding of that structure. That makes for less complexity to your reader to understand your proof, because the reader need not decode the structure of your argument from scratch.

This section briefly introduces three commonly used proof techniques: (i) deduction, or direct proof; (ii) proof by contradiction, and (iii) proof by mathematical induction.

2.6.1 Direct Proof

In general, a **direct proof** is just a "logical explanation." A direct proof is sometimes referred to as an argument by deduction. This is simply an argument in terms of logic. Often written in English with words such as "if ... then," it could also be written with logic notation such as "$P \Rightarrow Q$." Even if we don't wish to use symbolic logic notation, we can still take advantage of fundamental theorems of logic to structure our arguments. For example, if we want to prove that P and Q are equivalent, we can first prove $P \Rightarrow Q$ and then prove $Q \Rightarrow P$.

In some domains, proofs are essentially a series of state changes from a start state to an end state. Formal predicate logic can be viewed in this way, with the various "rules of logic" being used to make the changes from one formula or combining a couple of formulas to make a new formula on the route to the destination. Symbolic manipulations to solve integration problems in introductory calculus classes are similar in spirit, as are high school geometry proofs.

2.6.2 Proof by Contradiction

The simplest way to *disprove* a theorem or statement is to find a counterexample to the theorem. Unfortunately, no number of examples supporting a theorem is sufficient to prove that the theorem is correct. However, there is an approach that is vaguely similar to disproving by counterexample, called Proof by Contradiction. To prove a theorem by contradiction, we first *assume* that the theorem is *false*. We then find a logical contradiction stemming from this assumption. If the logic used to find the contradiction is correct, then the only way to resolve the contradiction is to recognize that the assumption that the theorem is false must be incorrect. That is, we conclude that the theorem must be true.

Example 2.10 Here is a simple proof by contradiction.

Theorem 2.1 *There is no largest integer.*

Proof: Proof by contradiction.

Step 1. Contrary assumption: Assume that there *is* a largest integer. Call it B (for "biggest").

Step 2. Show this assumption leads to a contradiction: Consider $C = B + 1$. C is an integer because it is the sum of two integers. Also,

$C > B$, which means that B is not the largest integer after all. Thus, we have reached a contradiction. The only flaw in our reasoning is the initial assumption that the theorem is false. Thus, we conclude that the theorem is correct. □

A related proof technique is proving the contrapositive. We can prove that $P \Rightarrow Q$ by proving $(\text{not } Q) \Rightarrow (\text{not } P)$.

2.6.3 Proof by Mathematical Induction

Mathematical induction can be used to prove a wide variety of theorems. Induction also provides a useful way to think about algorithm design, because it encourages you to think about solving a problem by building up from simple subproblems. Induction can help to prove that a recursive function produces the correct result.. Understanding recursion is a big step toward understanding induction, and vice versa, since they work by essentially the same process.

Within the context of algorithm analysis, one of the most important uses for mathematical induction is as a method to test a hypothesis. As explained in Section 2.4, when seeking a closed-form solution for a summation or recurrence we might first guess or otherwise acquire evidence that a particular formula is the correct solution. If the formula is indeed correct, it is often an easy matter to prove that fact with an induction proof.

Let **Thrm** be a theorem to prove, and express **Thrm** in terms of a positive integer parameter n. Mathematical induction states that **Thrm** is true for any value of parameter n (for $n \geq c$, where c is some constant) if the following two conditions are true:

1. **Base Case: Thrm** holds for $n = c$, and
2. **Induction Step:** If **Thrm** holds for $n - 1$, then **Thrm** holds for n.

Proving the base case is usually easy, typically requiring that some small value such as 1 be substituted for n in the theorem and applying simple algebra or logic as necessary to verify the theorem. Proving the induction step is sometimes easy, and sometimes difficult. An alternative formulation of the induction step is known as **strong induction**. The induction step for strong induction is:

2a. Induction Step: If **Thrm** holds for all k, $c \leq k < n$, then **Thrm** holds for n.

Proving either variant of the induction step (in conjunction with verifying the base case) yields a satisfactory proof by mathematical induction.

The two conditions that make up the induction proof combine to demonstrate that **Thrm** holds for $n = 2$ as an extension of the fact that **Thrm** holds for $n = 1$. This fact, combined again with condition (2) or (2a), indicates that **Thrm** also holds

for $n = 3$, and so on. Thus, **Thrm** holds for all values of n (larger than the base cases) once the two conditions have been proved.

What makes mathematical induction so powerful (and so mystifying to most people at first) is that we can take advantage of the *assumption* that **Thrm** holds for all values less than n as a tool to help us prove that **Thrm** holds for n. This is known as the **induction hypothesis**. Having this assumption to work with makes the induction step easier to prove than tackling the original theorem itself. Being able to rely on the induction hypothesis provides extra information that we can bring to bear on the problem.

Recursion and induction have many similarities. Both are anchored on one or more base cases. A recursive function relies on the ability to call itself to get the answer for smaller instances of the problem. Likewise, induction proofs rely on the truth of the induction hypothesis to prove the theorem. The induction hypothesis does not come out of thin air. It is true if and only if the theorem itself is true, and therefore is reliable within the proof context. Using the induction hypothesis it do work is exactly the same as using a recursive call to do work.

Example 2.11 Here is a sample proof by mathematical induction. Call the sum of the first n positive integers $\mathbf{S}(n)$.

Theorem 2.2 $S(n) = n(n+1)/2$.

Proof: The proof is by mathematical induction.

1. **Check the base case.** For $n = 1$, verify that $\mathbf{S}(1) = 1(1+1)/2$. $\mathbf{S}(1)$ is simply the sum of the first positive number, which is 1. Because $1(1+1)/2 = 1$, the formula is correct for the base case.
2. **State the induction hypothesis.** The induction hypothesis is

$$\mathbf{S}(n-1) = \sum_{i=1}^{n-1} i = \frac{(n-1)((n-1)+1)}{2} = \frac{(n-1)(n)}{2}.$$

3. **Use the assumption from the induction hypothesis for $n-1$ to show that the result is true for n.** The induction hypothesis states that $\mathbf{S}(n-1) = (n-1)(n)/2$, and because $\mathbf{S}(n) = \mathbf{S}(n-1) + n$, we can substitute for $\mathbf{S}(n-1)$ to get

$$\begin{aligned} \sum_{i=1}^{n} i &= \left(\sum_{i=1}^{n-1} i\right) + n = \frac{(n-1)(n)}{2} + n \\ &= \frac{n^2 - n + 2n}{2} = \frac{n(n+1)}{2}. \end{aligned}$$

Thus, by mathematical induction,

$$\mathbf{S}(n) = \sum_{i=1}^{n} i = n(n+1)/2.$$

□

Note carefully what took place in this example. First we cast $\mathbf{S}(n)$ in terms of a smaller occurrence of the problem: $\mathbf{S}(n) = \mathbf{S}(n-1) + n$. This is important because once $\mathbf{S}(n-1)$ comes into the picture, we can use the induction hypothesis to replace $\mathbf{S}(n-1)$ with $(n-1)(n)/2$. From here, it is simple algebra to prove that $\mathbf{S}(n-1) + n$ equals the right-hand side of the original theorem.

Example 2.12 Here is another simple proof by induction that illustrates choosing the proper variable for induction. We wish to prove by induction that the sum of the first n positive odd numbers is n^2. First we need a way to describe the nth odd number, which is simply $2n-1$. This also allows us to cast the theorem as a summation.

Theorem 2.3 $\sum_{i=1}^{n}(2i-1) = n^2$.

Proof: The base case of $n = 1$ yields $1 = 1^2$, which is true. The induction hypothesis is

$$\sum_{i=1}^{n-1}(2i-1) = (n-1)^2.$$

We now use the induction hypothesis to show that the theorem holds true for n. The sum of the first n odd numbers is simply the sum of the first $n-1$ odd numbers plus the nth odd number. In the second line below, we will use the induction hypothesis to replace the partial summation (shown in brackets in the first line) with its closed-form solution. After that, algebra takes care of the rest.

$$\begin{aligned}\sum_{i=1}^{n}(2i-1) &= \left[\sum_{i=1}^{n-1}(2i-1)\right] + 2n - 1 \\ &= [(n-1)^2] + 2n - 1 \\ &= n^2 - 2n + 1 + 2n - 1 \\ &= n^2.\end{aligned}$$

Thus, by mathematical induction, $\sum_{i=1}^{n}(2i-1) = n^2$. □

Example 2.13 This example shows how we can use induction to prove that a proposed closed-form solution for a recurrence relation is correct.

Theorem 2.4 *The recurrence relation* $\mathbf{T}(n) = \mathbf{T}(n-1)+1; \quad \mathbf{T}(1) = 0$ *has closed-form solution* $\mathbf{T}(n) = n - 1$.

Proof: To prove the base case, we observe that $\mathbf{T}(1) = 1 - 1 = 0$. The induction hypothesis is that $\mathbf{T}(n-1) = n - 2$. Combining the definition of the recurrence with the induction hypothesis, we see immediately that

$$\mathbf{T}(n) = \mathbf{T}(n-1) + 1 = n - 2 + 1 = n - 1$$

for $n > 1$. Thus, we have proved the theorem correct by mathematical induction. □

Example 2.14 This example uses induction without involving summations or other equations. It also illustrates a more flexible use of base cases.

Theorem 2.5 *2¢ and 5¢ stamps can be used to form any value (for values* ≥ 4*).*

Proof: The theorem defines the problem for values ≥ 4 because it does not hold for the values 1 and 3. Using 4 as the base case, a value of 4¢ can be made from two 2¢ stamps. The induction hypothesis is that a value of $n - 1$ can be made from some combination of 2¢ and 5¢ stamps. We now use the induction hypothesis to show how to get the value n from 2¢ and 5¢ stamps. Either the makeup for value $n - 1$ includes a 5¢ stamp, or it does not. If so, then replace a 5¢ stamp with three 2¢ stamps. If not, then the makeup must have included at least two 2¢ stamps (because it is at least of size 4 and contains only 2¢ stamps). In this case, replace two of the 2¢ stamps with a single 5¢ stamp. In either case, we now have a value of n made up of 2¢ and 5¢ stamps. Thus, by mathematical induction, the theorem is correct. □

Example 2.15 Here is an example using strong induction.

Theorem 2.6 *For* $\mathrm{n} > 1$*,* n *is divisible by some prime number.*

Proof: For the base case, choose $n = 2$. 2 is divisible by the prime number 2. The induction hypothesis is that *any* value a, $2 \leq a < n$, is divisible by some prime number. There are now two cases to consider when proving the theorem for n. If n is a prime number, then n is divisible by itself. If n is not a prime number, then $n = a \times b$ for a and b, both integers less than n but greater than 1. The induction hypothesis tells us that a is divisible by some prime number. That same prime number must also divide n. Thus, by mathematical induction, the theorem is correct. □

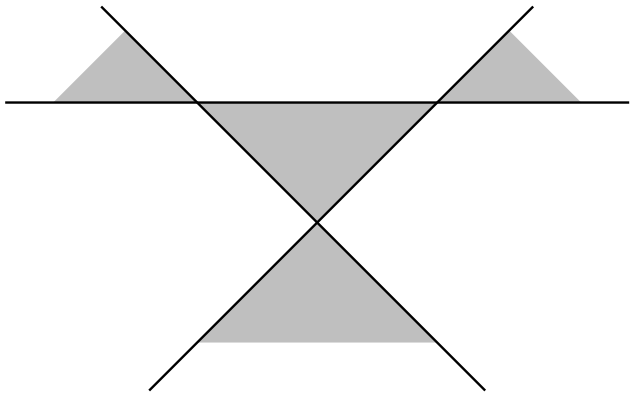

Figure 2.3 A two-coloring for the regions formed by three lines in the plane.

Our next example of mathematical induction proves a theorem from geometry. It also illustrates a standard technique of induction proof where we take n objects and remove some object to use the induction hypothesis.

Example 2.16 Define a **two-coloring** for a set of regions as a way of assigning one of two colors to each region such that no two regions sharing a side have the same color. For example, a chessboard is two-colored. Figure 2.3 shows a two-coloring for the plane with three lines. We will assume that the two colors to be used are black and white.

Theorem 2.7 *The set of regions formed by* n *infinite lines in the plane can be two-colored.*

Proof: Consider the base case of a single infinite line in the plane. This line splits the plane into two regions. One region can be colored black and the other white to get a valid two-coloring. The induction hypothesis is that the set of regions formed by $n-1$ infinite lines can be two-colored. To prove the theorem for n, consider the set of regions formed by the $n-1$ lines remaining when any one of the n lines is removed. By the induction hypothesis, this set of regions can be two-colored. Now, put the nth line back. This splits the plane into two half-planes, each of which (independently) has a valid two-coloring inherited from the two-coloring of the plane with $n-1$ lines. Unfortunately, the regions newly split by the nth line violate the rule for a two-coloring. Take all regions on one side of the nth line and reverse their coloring (after doing so, this half-plane is still two-colored). Those regions split by the nth line are now properly two-colored, because the part of the region to one side of the line is now black and the region to the other side is now white. Thus, by mathematical induction, the entire plane is two-colored. □

Compare the proof of Theorem 2.7 with that of Theorem 2.5. For Theorem 2.5, we took a collection of stamps of size $n-1$ (which, by the induction hypothesis, must have the desired property) and from that "built" a collection of size n that has the desired property. We therefore proved the existence of *some* collection of stamps of size n with the desired property.

For Theorem 2.7 we must prove that *any* collection of n lines has the desired property. Thus, our strategy is to take an *arbitrary* collection of n lines, and "reduce" it so that we have a set of lines that must have the desired property because it matches the induction hypothesis. From there, we merely need to show that reversing the original reduction process preserves the desired property.

In contrast, consider what is required if we attempt to "build" from a set of lines of size $n-1$ to one of size n. We would have great difficulty justifying that *all* possible collections of n lines are covered by our building process. By reducing from an arbitrary collection of n lines to something less, we avoid this problem.

This section's final example shows how induction can be used to prove that a recursive function produces the correct result.

Example 2.17 We would like to prove that function `fact` does indeed compute the factorial function. There are two distinct steps to such a proof. The first is to prove that the function always terminates. The second is to prove that the function returns the correct value.

Theorem 2.8 *Function* `fact` *will terminate for any value of* n.

Proof: For the base case, we observe that `fact` will terminate directly whenever $n \leq 0$. The induction hypothesis is that `fact` will terminate for $n-1$. For n, we have two possibilities. One possibility is that $n \geq 12$. In that case, `fact` will terminate directly because it will fail its assertion test. Otherwise, `fact` will make a recursive call to `fact(n-1)`. By the induction hypothesis, `fact(n-1)` must terminate. □

Theorem 2.9 *Function* `fact` *does compute the factorial function for any value in the range 0 to 12.*

Proof: To prove the base case, observe that when $n = 0$ or $n = 1$, `fact(n)` returns the correct value of 1. The induction hypothesis is that `fact(n-1)` returns the correct value of $(n-1)!$. For any value n within the legal range, `fact(n)` returns $n *$ `fact(n-1)`. By the induction hypothesis, `fact(n-1)` $= (n-1)!$, and because $n*(n-1)! = n!$, we have proved that `fact(n)` produces the correct result. □

We can use a similar process to prove many recursive programs correct. The general form is to show that the base cases perform correctly, and then to use the induction hypothesis to show that the recursive step also produces the correct result.

Prior to this, we must prove that the function always terminates, which might also be done using an induction proof.

2.7 Estimation

One of the most useful life skills that you can gain from your computer science training is the ability to perform quick estimates. This is sometimes known as "back of the napkin" or "back of the envelope" calculation. Both nicknames suggest that only a rough estimate is produced. Estimation techniques are a standard part of engineering curricula but are often neglected in computer science. Estimation is no substitute for rigorous, detailed analysis of a problem, but it can serve to indicate when a rigorous analysis is warranted: If the initial estimate indicates that the solution is unworkable, then further analysis is probably unnecessary.

Estimation can be formalized by the following three-step process:

1. Determine the major parameters that affect the problem.
2. Derive an equation that relates the parameters to the problem.
3. Select values for the parameters, and apply the equation to yield an estimated solution.

When doing estimations, a good way to reassure yourself that the estimate is reasonable is to do it in two different ways. In general, if you want to know what comes out of a system, you can either try to estimate that directly, or you can estimate what goes into the system (assuming that what goes in must later come out). If both approaches (independently) give similar answers, then this should build confidence in the estimate.

When calculating, be sure that your units match. For example, do not add feet and pounds. Verify that the result is in the correct units. Always keep in mind that the output of a calculation is only as good as its input. The more uncertain your valuation for the input parameters in Step 3, the more uncertain the output value. However, back of the envelope calculations are often meant only to get an answer within an order of magnitude, or perhaps within a factor of two. Before doing an estimate, you should decide on acceptable error bounds, such as within 25%, within a factor of two, and so forth. Once you are confident that an estimate falls within your error bounds, leave it alone! Do not try to get a more precise estimate than necessary for your purpose.

Example 2.18 How many library bookcases does it take to store books containing one million pages? I estimate that a 500-page book requires one inch on the library shelf (it will help to look at the size of any handy book), yielding about 200 feet of shelf space for one million pages. If a shelf is 4 feet wide, then 50 shelves are required. If a bookcase contains

5 shelves, this yields about 10 library bookcases. To reach this conclusion, I estimated the number of pages per inch, the width of a shelf, and the number of shelves in a bookcase. None of my estimates are likely to be precise, but I feel confident that my answer is correct to within a factor of two. (After writing this, I went to Virginia Tech's library and looked at some real bookcases. They were only about 3 feet wide, but typically had 7 shelves for a total of 21 shelf-feet. So I was correct to within 10% on bookcase capacity, far better than I expected or needed. One of my selected values was too high, and the other too low, which canceled out the errors.)

Example 2.19 Is it more economical to buy a car that gets 20 miles per gallon, or one that gets 30 miles per gallon but costs $3000 more? The typical car is driven about 12,000 miles per year. If gasoline costs $3/gallon, then the yearly gas bill is $1800 for the less efficient car and $1200 for the more efficient car. If we ignore issues such as the payback that would be received if we invested $3000 in a bank, it would take 5 years to make up the difference in price. At this point, the buyer must decide if price is the only criterion and if a 5-year payback time is acceptable. Naturally, a person who drives more will make up the difference more quickly, and changes in gasoline prices will also greatly affect the outcome.

Example 2.20 When at the supermarket doing the week's shopping, can you estimate about how much you will have to pay at the checkout? One simple way is to round the price of each item to the nearest dollar, and add this value to a mental running total as you put the item in your shopping cart. This will likely give an answer within a couple of dollars of the true total.

2.8 Further Reading

Most of the topics covered in this chapter are considered part of Discrete Mathematics. An introduction to this field is *Discrete Mathematics with Applications* by Susanna S. Epp [Epp10]. An advanced treatment of many mathematical topics useful to computer scientists is *Concrete Mathematics: A Foundation for Computer Science* by Graham, Knuth, and Patashnik [GKP94].

See "Technically Speaking" from the February 1995 issue of *IEEE Spectrum* [Sel95] for a discussion on the standard for indicating units of computer storage used in this book.

Introduction to Algorithms by Udi Manber [Man89] makes extensive use of mathematical induction as a technique for developing algorithms.

For more information on recursion, see *Thinking Recursively* by Eric S. Roberts [Rob86]. To learn recursion properly, it is worth your while to learn the programming languages LISP or Scheme, even if you never intend to write a program in either language. In particular, Friedman and Felleisen's "Little" books (including *The Little LISPer*[FF89] and *The Little Schemer*[FFBS95]) are designed to teach you how to think recursively as well as teach you the language. These books are entertaining reading as well.

A good book on writing mathematical proofs is Daniel Solow's *How to Read and Do Proofs* [Sol09]. To improve your general mathematical problem-solving abilities, see *The Art and Craft of Problem Solving* by Paul Zeitz [Zei07]. Zeitz also discusses the three proof techniques presented in Section 2.6, and the roles of investigation and argument in problem solving.

For more about estimation techniques, see two Programming Pearls by John Louis Bentley entitled *The Back of the Envelope* and *The Envelope is Back* [Ben84, Ben00, Ben86, Ben88]. *Genius: The Life and Science of Richard Feynman* by James Gleick [Gle92] gives insight into how important back of the envelope calculation was to the developers of the atomic bomb, and to modern theoretical physics in general.

2.9 Exercises

2.1 For each relation below, explain why the relation does or does not satisfy each of the properties reflexive, symmetric, antisymmetric, and transitive.

- **(a)** "isBrotherOf" on the set of people.
- **(b)** "isFatherOf" on the set of people.
- **(c)** The relation $R = \{\langle x, y\rangle \mid x^2 + y^2 = 1\}$ for real numbers x and y.
- **(d)** The relation $R = \{\langle x, y\rangle \mid x^2 = y^2\}$ for real numbers x and y.
- **(e)** The relation $R = \{\langle x, y\rangle \mid x \bmod y = 0\}$ for $x, y \in \{1, 2, 3, 4\}$.
- **(f)** The empty relation $\emptyset$ (i.e., the relation with no ordered pairs for which it is true) on the set of integers.
- **(g)** The empty relation $\emptyset$ (i.e., the relation with no ordered pairs for which it is true) on the empty set.

2.2 For each of the following relations, either prove that it is an equivalence relation or prove that it is not an equivalence relation.

- **(a)** For integers a and b, $a \equiv b$ if and only if $a + b$ is even.
- **(b)** For integers a and b, $a \equiv b$ if and only if $a + b$ is odd.
- **(c)** For nonzero rational numbers a and b, $a \equiv b$ if and only if $a \times b > 0$.
- **(d)** For nonzero rational numbers a and b, $a \equiv b$ if and only if a/b is an integer.