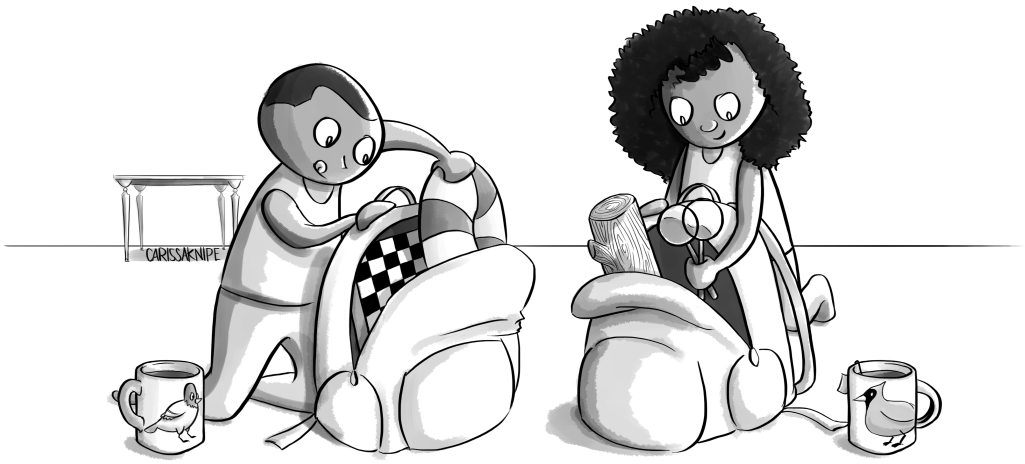
2 Basic Data Types



In which our heroes equip themselves for the journey ahead, by taking on the basic provisions that they will need along the road.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.1 Why You Might Care

> It is a capital mistake to theorize before one has data.
>
> Sir Arthur Conan Doyle (1859–1930)
> *A Scandal in Bohemia* (1892)

Imagine converting a color photograph to grayscale (as in Figure 2.1). Implementing this conversion requires interacting with a slew of foundational data types (the basic "kinds of things") that show up throughout CS. A pixel is a *sequence* of three color values, red, green, and blue. (And an image is a two-dimensional sequence of pixels.) Those color values are *integers* between 0 and 255 (because each is represented as a sequence of 8 bits). The translation process is a *function* taking inputs (any of the *set* of possible color values) and producing outputs (any of the set of grayscale values) using a particular formula.

Virtually every interesting computer science application uses these basic data types extensively. Cryptography, which is devoted to the secure storage and transmission of information so that a malicious third party cannot decipher that information, is typically based directly on integers, particularly large prime numbers. A ubiquitous task in machine learning is to "cluster" a set of entities into a collection of nonoverlapping subsets so that two entities in the same subset are similar and two entities in different subsets are dissimilar. In information retrieval, where we might seek to find the document from a large collection that is most relevant to a given query, it is common to represent each document by a vector (a sequence of numbers) based on the words used in the document, and to find the most relevant documents by identifying which ones "point in the same direction" as the query's vector. And functions are everywhere in CS, from data structures like hash tables to the routing that's done for every packet of information on the internet.

This chapter introduces concepts, terminology, and notation related to the most common data types that recur throughout this book, and throughout computer science. These basic entities—the Booleans (True and False), numbers (integers, rationals, and reals), sets, sequences, functions—are also the basic data types we use in modern programming languages. Some specific closely related topics will appear later in

This grayscale image was produced from the original color using a rough-cut formula $0.2126 \cdot \mathit{red} + 0.7152 \cdot \mathit{green} + 0.0722 \cdot \mathit{blue}$, rounded down to the nearest integer (in other words, using the *floor* function).

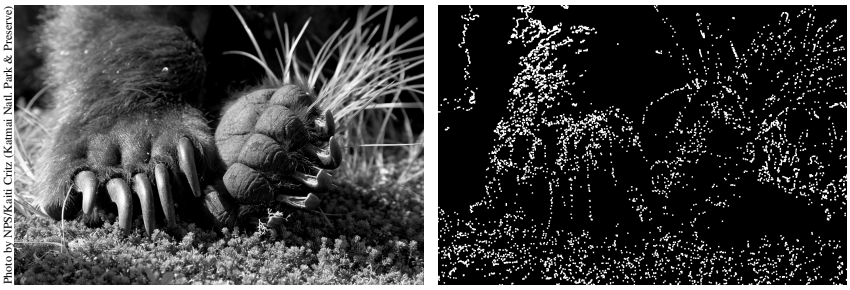


Photo by NPS/Kaiti Critz (Kaimai Natl. Park & Preserve)

Figure 2.1 Converting an image to grayscale. The second image's highlighted pixels—variously corresponding to brown fur, gray claws, and green plants—are all the same shade of gray. (That is, using the terminology of Section 2.5, the function is not one-to-one, a fact that has implications for designing interfaces when some users are colorblind.)

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

the book, as well. But, really, every chapter of this book is related to this chapter: our whole enterprise will involve building complex objects out of these simple ones (and, to be ready to understand the more complex objects, we have to understand the simple pieces first). And before we launch into the sea of applications, we need to establish some basic shared language. Much of the basic material in this chapter may be familiar, but regardless of whether you have seen it before, it is important and standard content with which it is important to be comfortable.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.2 Booleans, Numbers, and Arithmetic

> "And you do Addition?" the White Queen asked. "What's one and one and one and one and one and one and one and one and one and one?"
>
> "I don't know," said Alice. "I lost count."
>
> "She can't do Addition," the Red Queen interrupted.
>
> Lewis Carroll (1832–1898)
> *Through the Looking-Glass* (1871)

We start by introducing the most basic types of data: *Boolean* values (True and False), *integers* ($\ldots, -2, -1, 0, 1, 2, \ldots$), *rational numbers* (fractions with integers as numerators and denominators), and *real numbers* (including the integers and all the numbers in between them). The rest of this section will then introduce some basic numerical operations: absolute values and rounding, exponentiation and logarithms, summations and products. Figure 2.2 summarizes this section's notation and definitions.

2.2.1 Booleans: True and False

The most basic unit of data is the *bit*: a single piece of information, which either takes on the value 0 or the value 1. Every piece of stored data in a digital computer is stored as a sequence of bits. (See Section 2.4 for a formal definition of sequences.)

We'll view bits from several different perspectives: 1 and 0, on and off, yes and no, *True* and *False*. Bits viewed under the last of these perspectives have a special name, the *Booleans*:

Definition 2.1: Booleans.
A *Boolean value* is either True or False.

(Booleans are named after George Boole (1815–1864), a British mathematician, who was the first person to think about True as 1 and False as 0.) The Booleans are the central object of study of Chapter 3, on logic. In fact, they are in a sense the central object of study of this entire book: simply, we are interested in making true statements, with a proof to justify why the statement is true.

2.2.2 Numbers: Integers, Reals, and Rationals

We'll often encounter a few common types of numbers—*integers*, *reals*, and *rationals*:

Definition 2.2: Integers, reals, and rationals.
The *integers*, denoted by $\mathbb{Z}$, are those numbers with no fractional part: 0, the positive integers $(1, 2, \ldots)$, and the negative integers $(-1, -2, -3, \ldots)$.

The *real numbers*, denoted by $\mathbb{R}$, are those numbers that can be (approximately) represented by decimal numbers; informally, the reals include all integers and all numbers "between" any two integers.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Booleans	True and False
$\mathbb{Z}$	integers $(\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots)$
$\mathbb{Q}$	rational numbers
$\mathbb{R}$	real numbers
$[a, b]$	those real numbers x where $a \leq x \leq b$
(a, b)	those real numbers x where $a < x < b$
$[a, b)$	those real numbers x where $a \leq x < b$
$(a, b]$	those real numbers x where $a < x \leq b$
$\lvert x \rvert$	absolute value of x: $\lvert x \rvert = -x$ if $x < 0$; $\lvert x \rvert = x$ if $x \geq 0$
$\lfloor x \rfloor$	floor of x: x rounded down to the nearest integer
$\lceil x \rceil$	ceiling of x: x rounded up to the nearest integer
b^n	b multiplied by itself n times
$b^{1/n}$, or $\sqrt[n]{b}$	a number y such that $y^n = b$ (where $y \geq 0$ if possible), if one exists
$b^{m/n}$	$(b^{1/n})^m$
$\log_b x$	logarithm: $\log_b x$ is the value y such that $b^y = x$, if one exists
$n \bmod k$	modulo: $n \bmod k =$ the remainder when dividing n by k
$k \mid n$	k (evenly) divides n
$\sum$	summation: $\sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n$
$\prod$	product: $\prod_{i=1}^{n} x_i = x_1 \cdot x_2 \cdot \cdots \cdot x_n$

Figure 2.2 Summary of the basic mathematical notation introduced in Section 2.2.

The *rational numbers*, denoted by $\mathbb{Q}$, are those real numbers that can be represented as a ratio $\frac{n}{m}$ of two integers n and m, where n is called the *numerator* and $m \neq 0$ is called the *denominator*. A real number that is not rational is called an *irrational* number.

The superficially unintuitive notation for the integers, the symbol $\mathbb{Z}$, is a stylized "Z" that was chosen because of the German word *Zahlen*, which means "numbers." The name *rationals* comes from the word *ratio*; the symbol $\mathbb{Q}$ comes from its synonym *quotient.* (Besides, the symbol $\mathbb{R}$ was already taken by the reals, so the rationals got stuck with their second choice.)

Here are a few examples of each of these types of numbers:

Example 2.1: A few integers, reals, and rationals.
The following are all examples of integers: 1, 42, 0, and -17.

All of the following are real numbers: 1, 99.44, $\frac{1}{3} = 0.33333\cdots$, the ratio of the circumference of a circle to its diameter $\pi \approx 3.14159\cdots$, and the so-called *golden ratio* $\phi = (1 + \sqrt{5})/2 \approx 1.61803\cdots$.

Examples of rational numbers include $\frac{3}{2}$, $\frac{9}{5}$, $\frac{16}{4}$, and $\frac{4}{1}$. (In Chapter 8, we'll talk about the familiar notion of the equivalence of two rational numbers like $\frac{1}{2}$ and $\frac{2}{4}$, or like $\frac{16}{4}$ and $\frac{4}{1}$, based on common divisors. See Example 8.36.) Of the example real numbers above, all of 1 and 99.44 and $0.3333\cdots$ are rational numbers; we can write them as $\frac{1}{1}$ and $\frac{4972}{50}$ and $\frac{1}{3}$, for example. Both π and ϕ are irrational.

Note that all integers are rational numbers (with denominator equal to 1), and all rational numbers are real numbers. But not all rational numbers are integers and not all real numbers are rational: for example, $\frac{3}{2}$ is not an integer, and $\sqrt{2}$ is not rational. (We'll prove that $\sqrt{2}$ is not rational in Example 4.20.)

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science.* This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Taking it further: Definition 2.2 specifies $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ somewhat informally. To be completely rigorous, one can define the nonnegative integers as the smallest collection of numbers such that: (i) 0 is an integer; and (ii) if x is an integer, then $x+1$ is also an integer. See Section 5.4.1. (Of course, for even this definition to make sense, we'd need to define the number zero and also define the operation of adding one.) With a proper definition of the integers, it's fairly easy to define the rationals as ratios of integers. But formally defining the real numbers is surprisingly challenging; it was a major enterprise of mathematics in the late 1800s, and is often the focus of a first course in analysis in an undergraduate mathematics curriculum.

Nearly all programming languages support both integers (usually known as `ints`) and real numbers (usually known as `floats`); see p. 2-20 for a bit about how these basic numerical types are implemented in real computers. (Rational numbers are much less frequently implemented as basic data types in programming languages, though there are some exceptions, like Scheme.)

In addition to the basic symbols that we've introduced to represent the integers, the rationals, and the reals ($\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$), we will also introduce special notation for some specific subsets of these numbers. We will write $\mathbb{Z}^{\geq 0}$ and $\mathbb{Z}^{\leq 0}$ to denote the nonnegative integers $(0, 1, 2, \ldots)$ and nonpositive integers $(0, -1, -2, \ldots)$, respectively. Generally, when we write $\mathbb{Z}$ with a superscripted condition, we mean all those integers for which the stated condition is true. For example, $\mathbb{Z}^{\neq 1}$ denotes all integers aside from 1. Similarly, we write $\mathbb{R}^{>0}$ to denote the positive real numbers (every real number $x > 0$). Other conditions in the superscript of $\mathbb{R}$ are analogous.

We'll also use standard notation for *intervals* of real numbers, denoting all real numbers between two specified values. There are two variants of this notation, which allow "between two specified values" to either *include* or *exclude* those specified values. We use round parentheses to mean "exclude the endpoint" and square brackets to mean "include the endpoint" when we denote a range. For example, $(a, b]$ denotes those real numbers x for which $a < x \leq b$, and $[a, b)$ denotes those real numbers x for which $a \leq x < b$. Sometimes (a, b) and $[a, b]$ are, respectively, called the *open interval* and *closed interval* between a and b. These four types of intervals are also sometimes denoted via a *number line*, with open and closed circles denoting open and closed intervals; see Figure 2.3 for an example.

For two real numbers x and y, we will use the standard notation "$x \approx y$" to denote that x is *approximately equal to* y. This notation is defined informally, because what counts as "close enough" to be approximately equal will depend heavily on context.

(a) The interval $(1, 4)$ (b) The interval $[1, 4]$ (c) The interval $[1, 4)$ (d) The interval $(1, 4]$

Figure 2.3 Number lines representing real numbers between 1 and 4, with 1 included in the range in (b) and (c), and 4 included in the range in (b) and (d).

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.2.3 Absolute Value, Floor, and Ceiling

In the remaining parts of Section 2.2, we will give definitions of some standard arithmetic operations that involve the numbers we just defined. We'll start in here with three operations on a real number: absolute value, floor, and ceiling.

The *absolute value* of a real number x, written $|x|$, denotes how far x is from 0, disregarding the *sign* of x (that is, disregarding whether x is positive or negative):

Definition 2.3: Absolute value.

The *absolute value* of a real number x is $|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise.} \end{cases}$

For example, $|42.42| = 42.42$ and $|-128| = 128$. (Definition 2.3 uses notation for defining "by cases": the value of $|x|$ is x when $x \geq 0$, and the value of $|x|$ is $-x$ otherwise—that is, when $x < 0$.)

For a real number x, we can consider x "rounded down" or "rounded up," which are called the *floor* and *ceiling* of x, respectively:

Definition 2.4: Floor and ceiling.

The *floor* of a real number x, written $\lfloor x \rfloor$, is the largest integer that is less than or equal to x. The *ceiling* of x, written $\lceil x \rceil$, is the smallest integer that is greater than or equal to x.

Note that Definition 2.4 defines the floor and ceiling of negative numbers, too; the definition doesn't care whether x is greater than or less than 0. Here are a few examples:

Example 2.2: A few floors and ceilings.

We have $\lfloor \sqrt{2} \rfloor = \lfloor 1.4142\cdots \rfloor = 1$ and $\lfloor 2\pi \rfloor = \lfloor 6.28318\cdots \rfloor = 6$ and $\lfloor 4 \rfloor = 4$. For ceilings, we have $\lceil \sqrt{2} \rceil = 2$ and $\lceil 2\pi \rceil = 7$ and $\lceil 4 \rceil = 4$.

For negative numbers, $\lfloor -\sqrt{2} \rfloor = \lfloor -1.4142\cdots \rfloor = -2$ and $\lceil -\sqrt{2} \rceil = -1$.

The number line may give an intuitive way to think about floor and ceiling: $\lfloor x \rfloor$ denotes the first integer that we encounter moving left in the number line starting at x; $\lceil x \rceil$ denotes the first integer that we encounter moving right from x. (And x itself counts for both definitions.) See Figure 2.4.

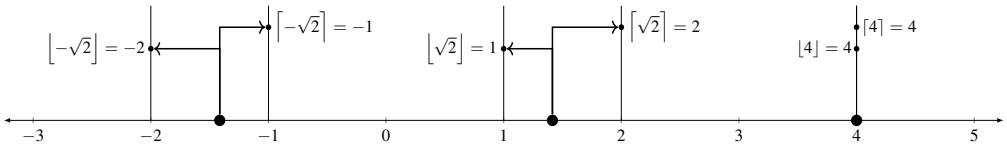


Figure 2.4 The floor and ceiling of $-\sqrt{2}$, $\sqrt{2}$, and 4.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.2.4 Exponentiation

We next consider raising a number to an *exponent* or *power*.

Definition 2.5: Raising a number to an integer power.
For a real number b and a nonnegative integer n, the number b^n denotes the result of multiplying b by itself n times:

$$b^0 = 1 \qquad \text{and, for } n \geq 1, \quad b^n = \underbrace{b \cdot b \cdot \ \cdots \ \cdot b}_{\longleftarrow\ n \text{ times} \longrightarrow}.$$

The number b is called the *base* and the integer n is called the *exponent*.

For example, $2^0 = 1$ and $2^2 = 2 \cdot 2 = 4$ and $2^5 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 32$ and $5^2 = 5 \cdot 5 = 25$.

Taking it further: Note that Definition 2.5 says that $b^0 = 1$ for *any* base b, including $b = 0$. The case of 0^0 is a little tricky: one is tempted to say *both* "0 to the anything is 0" *and* "anything to the 0 is 1." But, of course, these two statements are inconsistent. Lots of people define 0^0 as 1 (as Definition 2.5 does); lots of other people (particularly those who are giving definitions based on calculus) treat 0^0 as undefined. We'll live in the first camp because it won't get us in any trouble, and some results that we'll see in later chapters become more convoluted and more annoying to state if 0^0 were undefined. If you're interested in reading more about the (over 200 years worth of) history and the reasoning behind this decision, see [58].

Raising a base to nonintegral exponents

Consider the expression b^x for an exponent $x > 0$ that is not an integer. (It's all too easy to have done this calculation by typing numbers into a calculator without actually thinking about what the expression actually means!) Here's the definition of $b^{m/n}$ when the exponent $\frac{m}{n}$ is a rational number:

Definition 2.6: Raising a number to a positive rational power.
For a real number b and for a positive integer $n \neq 0$, the expression $b^{1/n}$ denotes the number y such that $y^n = b$. (If there are two values of y with $y^n = b$, then $b^{1/n}$ means the one that's positive. If there are no values of y with $y^n = b$, then $b^{1/n}$ is undefined.)

For a positive integer m, the expression $b^{m/n}$ denotes the *mth* power of $b^{1/n}$: that is, $b^{m/n} = (b^{1/n})^m$.

The value $b^{1/n}$ is called the *nth root* of b, and it can also be denoted by $\sqrt[n]{b}$.

For example, $16^{1/2} = 4$ because $16^{1/2}$ is the value y such that $y^2 = 16$, and $4^2 = 16$. Similarly, $16^{1/4} = 2$ because $2^4 = 16$. So $16^{1/2} = \sqrt[2]{16}$ is 4, and the 4th root of 16 is 2. (As the definition suggests, there may be more than one y such that $y^n = b$. For example, both $y = 2$ and $y = -2$ satisfy the condition that $y^4 = 16$. By the definition, if there are positive and negative values of y satisfying the requirement, we choose the positive one. So $16^{1/4} = 2$, not -2.) Here are a few other examples:

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Example 2.3: Some fractional exponents.
The value of $5^{1/2}$ is roughly 2.2360679774, because $2.2360679774^2 \approx 5$. (But note that this value of $5^{1/2}$ is only an approximation, because actually $2.2360679774^2 = 4.99999999955372691076 \neq 5$.)

When the base is negative, the nth root does not always exist. To find the value of $(-8)^{1/3}$, we need to find a y such that $y^3 = -8$. No $y \geq 0$ satisfies this condition, but $y = -2$ does. Thus $(-8)^{1/3} = -2$. On the other hand, let's try to compute the value of $(-8)^{1/2}$. We would need a value y such that $y^2 = -8$. But no $y \geq 0$ satisfies this condition, and no $y \leq 0$ does either. Thus $(-8)^{1/2}$ is undefined.

When we write $\sqrt{b}$ without explicitly indicating which root is intended, then we are talking about the *square root* of b. In other words, $\sqrt{b} = \sqrt[2]{b}$ denotes the (nonnegative) value y such that $y^2 = b$. An integer n is called a *perfect square* if $\sqrt{n}$ is an integer.

Taking it further: Definition 2.6 presents difficulties if we try to compute, say, $\sqrt{-1}$: the definition tells us that we need to find a number y such that $y^2 = -1$. But $y^2 \geq 0$ if $y \leq 0$ *and* also if $y \geq 0$, so no real number y satisfies the requirement $y^2 = -1$. (We had the same problem in Example 2.3, in trying to compute $(-8)^{1/2}$.) To handle this situation, several centuries ago some creative mathematicians defined the *imaginary numbers* as those resulting from taking the square root of negative numbers, specifically defining $\mathbf{i} = \sqrt{-1}$. (The name "real" to describe real numbers was chosen to contrast with the imaginary numbers.) We will not be concerned with imaginary numbers in this book, although—perhaps surprisingly—there are some very natural computational problems in which imaginary numbers are fundamental parts of the best algorithms solving them, such as the *Fast Fourier transform* in signal processing and speech processing (transcribing English words from a raw audio stream) or even quickly multiplying large numbers together [32].

Definition 2.7: Raising a number to a negative power.
When the exponent x is negative, then b^x is defined as $\frac{1}{b^{-x}}$.

For example, $2^{-4} = \frac{1}{2^4} = \frac{1}{16}$ and $25^{-3/2} = \frac{1}{25^{3/2}} = \frac{1}{(25^{1/2})^3} = \frac{1}{5^3} = \frac{1}{125}$.

For an irrational exponent x, the value of b^x is approximated arbitrarily closely by choosing a rational number $\frac{m}{n}$ sufficiently close to x and computing the value of $b^{m/n}$.

Taking it further: A rigorous treatment of irrational powers requires calculus; we will omit the details as they are tangential to our purposes in this book. But the basic idea is to approximate the exponent with a close-enough rational number, which then provably approximates the result of raising the base to that power. For example, we can approximate $2^\pi \approx 8.82497782708\cdots$ more and more closely as $2^3 = 8$, or $2^{31/10} = 8.5741\cdots$, or $2^{314/100} = 8.8815\cdots$, or $2^{3141/1000} = 8.8213\cdots$, etc.

While essentially every modern programming language supports exponentiation—including positive, fractional, and negative powers—in some form, often in a separate math library, the actual behind-the-scenes computation is rather complicated. See p. 2-22 for some discussion of the underlying steps that are done to compute a quantity like $\sqrt{x}$.

Here are a few useful facts about exponentiation:

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Theorem 2.8: Properties of exponentials.
For any real numbers a and b, and for any rational numbers x and y:

$$b^0 = 1 \tag{2.8.1}$$

$$b^1 = b \tag{2.8.2}$$

$$b^{x+y} = b^x \cdot b^y \tag{2.8.3}$$

$$(b^x)^y = b^{xy} \tag{2.8.4}$$

$$(ab)^x = a^x \cdot b^x \tag{2.8.5}$$

These properties follow fairly straightforwardly from the definition of exponentiation. (The properties of Theorem 2.8 carry over to irrational exponents, though the proofs are less straightforward. See the note after Definition 2.5 regarding (2.8.1).)

2.2.5 Logarithms

Problem-solving tip: I have found many CS students scared, and scarred, by logs. The fear appears to me to result from students attempting to *memorize* facts about logs without trying to think about what they *mean*. Mentally translating between logs and exponentials can help make these properties more intuitive and can help make them make sense. Often the intuition of a property of exponentials is reasonably straightforward to grasp.

The *logarithm* (or *log*) is the inverse operation to exponentiation: the value of an exponential b^y is the result of multiplying a number b by itself y times, while the value of a logarithm $\log_b x$ is the number of times we must multiply b by itself to get x.

Definition 2.9: Logarithm.
For a positive real number $b \neq 1$ and a real number $x > 0$, the *logarithm base b of x*, written $\log_b x$, is the real number y such that $b^y = x$.

Example 2.4: Some logs.
The quantity $\log_3 81$ is the power to which we must raise 3 to get 81—and thus $\log_3 81 = 4$, because $3^4 = 3 \cdot 3 \cdot 3 \cdot 3 = 81$. Similarly:

$$\underset{\text{because } 4^2 = 16}{\log_4 16 = 2} \qquad \underset{\text{because } 2 = \sqrt{4} = 4^{1/2}}{\log_4 2 = 0.5} \qquad \underset{\text{because } 128^0 = 1}{\log_{128} 1 = 0} \quad \text{and} \quad \underset{\text{because } 2^{1.5849625} = 2.999999998 \approx 3}{\log_2 3 \approx 1.5849625.}$$

For any base b, note that $\log_b x$ does get larger as the value of x increases, but it gets larger very slowly. Figure 2.5 illustrates the slow rate of growth of $\log_2 x$ and $\log_{10} x$ as x grows.

For a real number $x \leq 0$ and any base b, the expression $\log_b x$ is undefined. For example, the value of $\log_2(-4)$ would be the number y such that $2^y = -4$—but 2^y can never be negative. Similarly, logarithms base 1 are undefined: $\log_1 2$ would be the number y such that $1^y = 2$—but $1^y = 1$ for every value of y.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

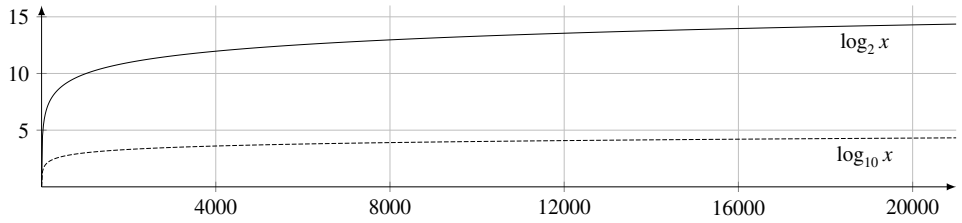


Figure 2.5 A graph of $\log_2 x$ and $\log_{10} x$.

Logarithms show up frequently in the analysis of data structures and algorithms, including a number that we will discuss in this book. Several facts about logarithms will be useful in these analyses, and are also useful in other settings. Here are a few (these properties generally follow directly from the analogous properties of exponentials in Theorem 2.8):

Theorem 2.10: Properties of logarithms.
For any real numbers $b > 1$, $c > 1$, $x > 0$, and $y > 0$, the following properties hold:

$$\log_b 1 = 0 \tag{2.10.1}$$

$$\log_b b = 1 \tag{2.10.2}$$

$$\log_b xy = \log_b x + \log_b y \quad \text{log of a product} \tag{2.10.3}$$

$$\log_b \tfrac{x}{y} = \log_b x - \log_b y \quad \text{log of a quotient} \tag{2.10.4}$$

$$\log_b x^y = y \log_b x \tag{2.10.5}$$

$$\log_b x = \frac{\log_c x}{\log_c b} \quad \text{“change of base” formula} \tag{2.10.6}$$

We will make use of one standard piece of notational shorthand: often the expression $\log x$ is written without an explicit base. When computer scientists write the expression $\log x$, we mean $\log_2 x$. One other base is commonly used in logarithms: the *natural logarithm* $\ln x$ denotes $\log_e x$, where $e \approx 2.718281828\cdots$ is defined from calculus as $e = \lim_{n\to\infty}(1 + \frac{1}{n})^n$.

Throughout this book (and throughout computer science), the assumed base of $\log x$ is 2. (Some computer scientists write $\lg x$ to denote $\log_2 x$; we'll simply write $\log x$.) But be aware that mathematicians or engineers may treat the default base to be e or 10.

2.2.6 Moduli and Division

So far, we've discussed multiplying numbers (repeatedly, to compute exponentials); here, we turn to the *division* of one number by another. When we consider dividing two integers—64 by 5, for example—there are several useful values to consider: regular-old division ($\frac{64}{5} = 12.8$), what's sometimes called *integer division* giving “the whole part” of the fraction ($\lfloor \frac{64}{5} \rfloor = 12$), and the *remainder* giving “the leftover part” of the fraction (the difference between 64 and $12 \cdot 5$, namely $64 - 60 = 4$).

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

We will return to these notions of division in great detail in Chapter 7, but we'll begin here with the formal definitions for the notions related to remainders:

Definition 2.11: Modulus (remainder).
For any integers $k > 0$ and n, the integer n mod k is the remainder when we divide n by k. Using the "floor" notation from Section 2.2.3, the value of n mod k is $n - k \cdot \lfloor \frac{n}{k} \rfloor$.

Example 2.5: Three values mod three.
The value of 8 mod 3 is 2, because $8 = 2 + 6$ (and 6 is a multiple of 3). (Or, using the definition directly: 8 mod 3 $= 2$ because $\lfloor \frac{8}{3} \rfloor = \lfloor 2.6666\cdots \rfloor = 2$, and $8 - 2 \cdot 3 = 8 - 6 = 2$.)
Similarly, 28 mod 3 $= 1$, because $\lfloor \frac{28}{3} \rfloor = 9$, and $28 - 9 \cdot 3 = 28 - 27 = 1$.
And 48 mod 3 $= 0$, because $\lfloor \frac{48}{3} \rfloor = 16$, and $48 - 16 \cdot 3 = 0$.

Taking it further: In many programming languages, the `/` operator performs integer division when its arguments are both integers, and performs "real" division when either argument is a floating point number. So the expression `64 / 5` will yield `12`, but `64.0 / 5` and `64 / 5.0` and `64.0 / 5.0` will all yield `12.8`. In this book, though, we will always mean "real" division when we write x/y or $\frac{x}{y}$. The n mod k operation, too, is a standard one in programming languages—it's written as `n % k` in many languages, including Java, Python, and C/C++, for example.

In Definition 2.11, we allowed n to be a negative integer, which may stretch your intuition about remainders a bit. Here's an example of this case of the definition:

Example 2.6: A negative integer mod 5.
We'll compute -3 mod 5 simply by following the definition of mod from Definition 2.11:

$$-3 \bmod 5 = (-3) - 5 \cdot \lfloor \tfrac{-3}{5} \rfloor = (-3) - 5 \cdot (-1) = (-3) + 5 = 2.$$

Viewed from an appropriate perspective, this calculation should actually be reasonably intuitive: the value $r = n$ mod k gives the amount r by which n exceeds its closest multiple of k. (And -3 is 2 more than a multiple of 5, namely -5, so -3 mod $5 = 2$.)

Notice that the value of n mod k is always at least 0 and at most $k - 1$, for any n and any $k > 0$; the remainder when dividing by k can never be k or more. When $\frac{n}{k}$ has zero remainder, then we say that k *(evenly) divides* n:

Definition 2.12: Integer k (evenly) divides integer n.
For any integers $k > 0$ and n, we say that k *divides* n, written $k \mid n$, if $\frac{n}{k}$ is an integer. Notice that $k \mid n$ is equivalent to n mod $k = 0$.

By rearranging the floor-based definition from Definition 2.11 when n mod $k = 0$, we can see that the condition $k \mid n$ is also equivalent to the condition $k \cdot \lfloor \frac{n}{k} \rfloor = n$.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Example 2.7: What 5 divides.
Because $5 \cdot \lfloor \frac{10}{5} \rfloor = 5 \cdot 2 = 10$, we know $5 \mid 10$. But $5 \cdot \lfloor \frac{9}{5} \rfloor = 5 \cdot 1 = 5 \neq 9$, so $5 \nmid 9$.

Some special numbers: evens, odds, primes, composites

A few special types of integers are defined in terms of their divisibility—specifically based on whether they are divisible by 2 (*evens* and *odds*), or whether they are divisible by any other integer except for 1 (*primes* and *composites*).

Definition 2.13: Even, odd, and parity.
A nonnegative integer n is *even* if $n \bmod 2 = 0$, and n is *odd* if $n \bmod 2 = 1$. The *parity* of n is its "oddness" or "evenness."

For example, we have $17 \bmod 2 = 1$ and $42 \bmod 2 = 0$, so 17 is odd and 42 is even.

Taking it further: If we view 0 as False and 1 as True (see Section 2.2.1), then the value $n \bmod 2$ can be interpreted as a Boolean value. In fact, there's a deeper connection between arithmetic and the Booleans than might be readily apparent. The "exclusive or" of two Boolean values p and q (which we will encounter in Section 3.2.3) is denoted $p \oplus q$, and the expression $p \oplus q$ is true when one but not both of p and q is true. The exclusive or is sometimes referred to as the *parity function*, because $p + q$ is odd (viewing p and q as numerical values, 0 or 1) exactly when $p \oplus q$ is true (viewing p and q as Boolean values, False or True).

Definition 2.14: Prime and composite numbers.
A positive integer $n > 1$ is *prime* if the only positive integers that evenly divide n are 1 and n itself. A positive integer $n > 1$ is *composite* if it is not prime.

Notice that the definition of prime numbers does not include 0 and 1, and neither does the definition of composite numbers: in other words, 0 and 1 are neither composite nor prime.

Example 2.8: Prime numbers.
Is 77 prime? What about 7?

Solution. 77 is not prime, because it is evenly divisible by 7. In other words, because $77 \bmod 7 = 0$ (and the integer 7 that evenly divides 77 is neither 1 nor 77 itself), 77 is composite. (11 also divides 77.)

On the other hand, 7 is prime. Convincing yourself that a number *is* prime is harder than convincing yourself that one is *not* prime, but we can see it by trying all the possible divisors (every positive integer except 1 and 7): $7 \bmod 2 = 1$ and $7 \bmod 3 = 1$ and $7 \bmod 4 = 3$ and $7 \bmod 5 = 2$ and $7 \bmod 6 = 1$, and furthermore $7 \bmod d = 7$ for any $d \geq 8$. None of these remainders is zero, so 7 is prime.

Example 2.9: Small primes and composites.
The first ten prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23, and 29.
The first ten composite numbers are 4, 6, 8, 9, 10, 12, 14, 15, 16, and 18.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Chapter 7 is devoted to the properties of modular arithmetic, prime numbers, and the like. These quantities have deep and important connections to cryptography, error-correcting codes, and other applications that we'll explore later.

2.2.7 Summations and Products

There is one final piece of notation related to numbers that we need to introduce: a simple way of expressing the *sum* or *product* of a collection of numbers. We'll start with the compact *summation notation* that allows us to express the result of adding many numbers:

Definition 2.15: Summation notation.
Let $x_1, x_2, \ldots, x_n$ be a sequence of n numbers. We write $\sum_{i=1}^{n} x_i$ (usually read as "the sum for i equals 1 to n of x_i") to denote the sum of the x_is:

$$\sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n.$$

The variable i is called the *index of summation* or the *index variable*.

Note that $\sum_{i=1}^{0} x_i = 0$: when you add nothing together, you end up with zero.

A note about notation: $\sum_{i=1}^{n} x_i$ and $\sum_{i=1}^{n} x_i$ mean exactly the same thing.

(The placement of the "limits" (1 and n) is an aesthetic matter of layout, not a matter of meaning.)

It may be helpful to interpret this summation notation as if it expressed a **for** loop, as shown in Figure 2.6.

Example 2.10: Some small summations.
Let $a_1 = 2$, $a_2 = 4$, $a_3 = 8$, and $a_4 = 16$, and let $b_1 = 1$, $b_2 = 2$, $b_3 = 3$, $b_4 = 4$, and $b_5 = 8$. Then

$$\sum_{i=1}^{4} a_i = a_1 + a_2 + a_3 + a_4 \qquad \text{and} \qquad \sum_{i=1}^{5} b_i = b_1 + b_2 + b_3 + b_4 + b_5$$
$$= 2 + 4 + 8 + 16 = 30 \qquad = 1 + 2 + 3 + 4 + 8 = 18.$$

In general, instead of just adding x_i in the ith term of the sum, we can add any expression involving the index of summation. (We can also start the index of summation at a value other than 1: to denote the sum $x_j + x_{j+1} + \cdots + x_n$, we write $\sum_{i=j}^{n} x_i$.) Here are a few examples:

```
1 result := 0
2 for i := 1, 2, ..., n:
3     result := result + x_i
4 return result
```

This **for** loop interpretation of $\sum_{i=1}^{n} x_i$ might help make the "empty sum" (when $n = 0$) more intuitive: the value of $\sum_{i=1}^{0} x_i = 0$ is simply 0 because *result* is set to 0 in line 1, and it never changes, because $n = 0$ (and therefore line 3 is never executed).

Figure 2.6 A **for** loop that returns the value of $\sum_{i=1}^{n} x_i$.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Example 2.11: Some sums.
Let $a_1 = 2$, $a_2 = 4$, $a_3 = 8$, and $a_4 = 16$. Then

$$\sum_{i=2}^{4} a_i \quad \text{and} \quad \sum_{i=1}^{3}(a_i + 1) \quad \text{and} \quad \sum_{i=1}^{4} i$$
$$= 4 + 8 + 16 \qquad = (2+1) + (4+1) + (8+1) \qquad = 1 + 2 + 3 + 4$$
$$= 28 \qquad = 17 \qquad = 10.$$

Example 2.12: Some more sums.
As in Example 2.11, let $a_1 = 2$, $a_2 = 4$, $a_3 = 8$, and $a_4 = 16$. Evaluate the following four expressions:

$$\sum_{i=1}^{4} i^2 \quad \text{and} \quad \sum_{i=2}^{4} i^2 \quad \text{and} \quad \sum_{i=1}^{4}(a_i + i^2) \quad \text{and} \quad \sum_{i=1}^{4} 5.$$

Solution. Here are the values of these sums:

$$\sum_{i=1}^{4} i^2 = 30 \quad \text{and} \quad \sum_{i=2}^{4} i^2 = 29 \quad \text{and} \quad \sum_{i=1}^{4}(a_i + i^2) = 60 \quad \text{and} \quad \sum_{i=1}^{4} 5 = 20.$$
$$1^2 + 2^2 + 3^2 + 4^2 \qquad 2^2 + 3^2 + 4^2 \qquad (2+1^2) + (4+2^2) + (8+3^2) + (16+4^2) \qquad 5+5+5+5$$

Two special types of summations arise frequently enough to have special names. A *geometric series* is $\sum_{i=1}^{n} r^i$ for some real number r; an *arithmetic series* is $\sum_{i=1}^{n} i \cdot r$ for a real number r. (See Section 5.2.2.)

We will very occasionally consider an *infinite* sequence of numbers $x_1, x_2, \ldots, x_i, \ldots$; we may write $\sum_{i=1}^{\infty} x_i$ to denote the infinite sum of these numbers. Here's one example of an infinite summation:

Example 2.13: An infinite sum.
Define $x_i = 1/2^i$, so that $x_1 = \frac{1}{2}$, $x_2 = \frac{1}{4}$, $x_3 = \frac{1}{8}$, and so forth. We can write $\sum_{i=1}^{\infty} x_i$ to denote $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots$. The value of this summation is 1: each term takes the sum halfway closer to 1.

While the **for** loop in Figure 2.6 would run forever if we tried to apply it to an infinite summation, the idea remains precisely the same: we successively add the value of each term to the *result* variable. (We will discuss this type of infinite sum in detail in Section 5.2.2, too.)

Reindexing summations

Just as in a **for** loop, the "name" of the index variable in a summation doesn't matter, as long as it's used consistently. For example, both $\sum_{i=1}^{5} a_i$ and $\sum_{j=1}^{5} a_j$ denote the value of $a_1 + a_2 + a_3 + a_4 + a_5$.

We can also rewrite a summation by *reindexing* it (also known as using a *change of index* or a *change of variable*), by adjusting both the limits of the sum (lower and upper) and what's being summed while ensuring that, overall, exactly the same things are being added together. Here are two examples:

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Example 2.14: Shifting by two.

The sums $\sum_{i=3}^{n} i$ and $\sum_{j=1}^{n-2}(j+2)$ are equal, because both express $3+4+5+\cdots+n$. (We have applied the substitution $j := i-2$ to get from the first summation to the second.)

Example 2.15: Counting backward.

Consider the summation $\sum_{i=0}^{n}(n-i)$. If we define $j := n-i$, then the summed value becomes j instead of $n-i$, and the range $i = 0, 1, \ldots, n$ corresponds to $j = n-0, n-1, \ldots, n-n$ (or, more simply, to $j = n, n-1, \ldots, 0$). Thus $\sum_{i=0}^{n}(n-i)$ and $\sum_{j=0}^{n} j$ are equal. (See Figure 2.7.)

Reindexing can be surprisingly helpful when we're confronted by ungainly summations; doing so can often turn the given summation into something more familiar.

Nested sums

We can sum any expression that depends on the index variable—including summations. These summations are called *double summations* or, more generally, *nested summations*. Just as with nested loops in programs, the key is to read "from the inside out" in simplifying a summation. Here are two examples:

Example 2.16: A double sum.

Let's compute the value of the nested sum $\sum_{i=1}^{6}\sum_{j=1}^{i} 5$.

For any i, the value of $\sum_{j=1}^{i} 5$ is just $5i$, because we are summing i different copies of the number 5. (See Figure 2.8.) Therefore the given sum's value is $\sum_{i=1}^{6} 5i = 5+10+15+20+25+30 = 105$.

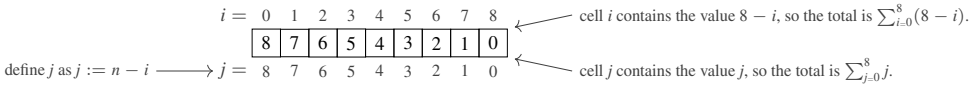


Figure 2.7 Computing $0+1+\cdots+8$ in two ways: $\sum_{j=0}^{8} j = \sum_{i=0}^{8}(8-i)$. (See Example 2.15.)

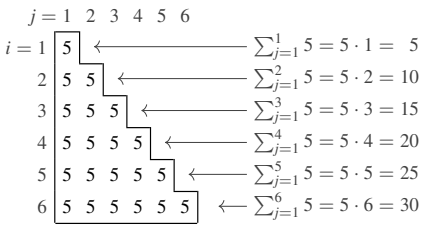


Figure 2.8 The terms of $\sum_{i=1}^{6}\sum_{j=1}^{i} 5$.

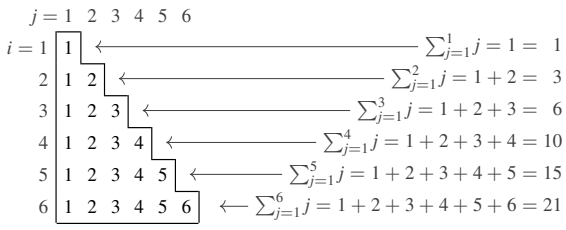


Figure 2.9 The terms of $\sum_{i=1}^{6}\sum_{j=1}^{i} j$.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Example 2.17: A slightly more complicated double sum.

What is the value of the sum $\sum_{i=1}^{6} \sum_{j=1}^{i} j$?

Solution. The inner sum ($\sum_{j=1}^{i} j$) has the values shown in Figure 2.9: 1 for $i = 1$, 3 for $i = 2$, 6 for $i = 3$, etc. Thus $\sum_{i=1}^{6} \sum_{j=1}^{i} j = 1 + 3 + 6 + 10 + 15 + 21 = 56$.

When you're programming and need to write two nested loops, it sometimes ends up being easier to write the loops with one variable in the outer loop rather than the other variable. Similarly, it may turn out to be easier to think about a nested sum by *reversing the summation*—that is, swapping which variable is the "outer" summation and which is the "inner." If we have any sequence $a_{i,j}$ of numbers indexed by two variables i and j, then $\sum_{i=1}^{n} \sum_{j=1}^{n} a_{i,j}$ and $\sum_{j=1}^{n} \sum_{i=1}^{n} a_{i,j}$ have precisely the same value.

Here are two examples of reversing the order of a double summation, for the tables shown in Figure 2.10:

Example 2.18: Adding up the numbers in a small table.

Consider the table in Figure 2.10a. Write $a_{i,j}$ to denote the element in the ith row and jth column of the table. Then we can compute the sum of elements in the table by adding up the row sums:

$$\sum_{i=1}^{3} \sum_{j=1}^{4} a_{i,j} = \underbrace{23}_{\substack{\text{sum of elements} \\ \text{in row 1}}} + \underbrace{18}_{\substack{\text{sum of elements} \\ \text{in row 2}}} + \underbrace{19}_{\substack{\text{sum of elements} \\ \text{in row 3}}} = 60.$$

Or, by adding up the column sums, the sum of elements in the table is also

$$\sum_{j=1}^{4} \sum_{i=1}^{3} a_{i,j} = \underbrace{15}_{\substack{\text{sum of elements} \\ \text{in column 1}}} + \underbrace{15}_{\substack{\text{sum of elements} \\ \text{in column 2}}} + \underbrace{15}_{\substack{\text{sum of elements} \\ \text{in column 3}}} + \underbrace{15}_{\substack{\text{sum of elements} \\ \text{in column 4}}} = 60.$$

	1	2	3	4
1	7	5	6	5
2	5	5	1	7
3	3	5	8	3

(a) A small table.

$j=$	1	2	3	4	5	6	7	8
$i=1$	−1	−1	−2	−2	−3	−3	−4	−4
2	1	1	2	2	3	3	4	4
3	−1	−1	−2	−2	−3	−3	−4	−4
4	1	1	2	2	3	3	4	4
5	−1	−1	−2	−2	−3	−3	−4	−4
6	1	1	2	2	3	3	4	4
7	−1	−1	−2	−2	−3	−3	−4	−4
8	1	1	2	2	3	3	4	4

(b) $\sum_{i=1}^{n} \sum_{j=1}^{n} \left((-1)^i \cdot \lceil \frac{j}{2} \rceil\right)$, for $n = 8$.

$j=$	1	2	3	4	5	6
$i=1$	1					
2	1	2				
3	1	2	3			
4	1	2	3	4		
5	1	2	3	4	5	
6	1	2	3	4	5	6

(c) The terms of $\sum_{i=1}^{6} \sum_{j=1}^{i} j$.

Figure 2.10 A few tables whose elements we'll sum "row-wise" and "column-wise."

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Example 2.19: A double sum, reversed.

Let $n = 8$. What is the value of the sum $\sum_{i=1}^{n}\sum_{j=1}^{n}\left[(-1)^i \cdot \lceil \frac{j}{2} \rceil\right]$?

Solution. We are computing the sum of all the values contained in the table in Figure 2.10b. The *hard* way to add up all of these values is by computing the row sums, and then adding them all up. (The given equation expresses this hard way.) The *easier* way is reverse the summation, and to instead compute

$$\sum_{j=1}^{n}\sum_{i=1}^{n}\left[(-1)^i \cdot \lceil \tfrac{j}{2} \rceil\right].$$

For each j, observe that $\sum_{i=1}^{n}(-1)^i \cdot \lceil \frac{j}{2} \rceil$ is actually zero! (This value is just $(\lceil \frac{j}{2} \rceil)\frac{n}{2} + (-\lceil \frac{j}{2} \rceil)\frac{n}{2}$.) In other words, every column sum in the table is zero. Thus the entire summation is $\sum_{j=1}^{n} 0$, which is just 0.

Problem-solving tip: When you're looking at a complicated double summation, try reversing it; it may be much easier to analyze the other way around.

Note that computing the sum from Example 2.19 when $n = 100$ or $n = 100{,}000$ remains just as easy if we use the column-based approach: as long as n is an even number, every column sum is 0, and thus the entire summation is 0. (The row-based approach is ever-more painful to use as n gets large.)

Here's one more example—another view of the double sum $\sum_{i=1}^{6}\sum_{j=1}^{i} j$ from Example 2.17—where reversing the summation makes the calculation simpler:

Example 2.20: A double sum, redone.

The value of $\sum_{i=1}^{6}\sum_{j=1}^{i} j$ is the sum of all the numbers in the table in Figure 2.10c. We solved Example 2.17 by first computing $\sum_{j=1}^{i} j$, which is the sum of the numbers in the ith row. We then summed these values over the six different values of i to get 56.

Alternatively, we can compute the desired sum by looking at *columns* instead of *rows*. The sum of the table's elements is also $\sum_{j=1}^{6}\left[\sum_{i=j}^{6} j\right]$, where $\sum_{i=j}^{6} j$ is the sum of the numbers in the jth *column*. Because there are a total of $(7-j)$ terms in $\sum_{i=j}^{6} j$, the sum of the numbers in the jth column is precisely $j \cdot (7-j)$. (For example, the 4th column's sum is $4 \cdot (7-4) = 4 \cdot 3 = 12$.) Thus the overall summation is

$$\begin{aligned}\sum_{i=1}^{6}\sum_{j=1}^{i} j &= \sum_{j=1}^{6}[j \cdot (7-j)] \\ &= \underset{=6}{(1\cdot 6)} + \underset{=10}{(2\cdot 5)} + \underset{=12}{(3\cdot 4)} + \underset{=12}{(4\cdot 3)} + \underset{=10}{(5\cdot 2)} + \underset{=6}{(6\cdot 1)} = 56.\end{aligned}$$

Products

The $\sum$ notation allows us to express repeated *addition* of a sequence of numbers; there is analogous notation to represent repeated *multiplication* of numbers, too:

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Definition 2.16: Product notation.
Let $x_1, x_2, \ldots, x_n$ be a sequence of n numbers. We write $\prod_{i=1}^{n} x_i$ (usually read as "the product for i equals 1 to n of x_i") to denote the product of the x_is:

$$\prod_{i=1}^{n} x_i = x_1 \cdot x_2 \cdot \ \cdots \ \cdot x_n.$$

(The summation and product notation have a secret mnemonic to help you remember what each means: "Σ" is the Greek letter Sigma, which starts with the same letter as the word *sum.* And "Π" is the Greek letter Pi, which starts with the same letter as the word *product.*)

Example 2.21: Some products.
Here are a few small examples of products:

$$\prod_{i=1}^{4} i = 24 \quad (1 \cdot 2 \cdot 3 \cdot 4 = 24) \qquad \text{and} \qquad \prod_{i=0}^{4} i = 0 \quad (0 \cdot 1 \cdot 2 \cdot 3 \cdot 4 = 0) \qquad \text{and} \qquad \prod_{i=1}^{4} i^2 = 576 \quad (1^2 \cdot 2^2 \cdot 3^2 \cdot 4^2 = 576) \qquad \text{and} \qquad \prod_{i=1}^{4} 5 = 625. \quad (5 \cdot 5 \cdot 5 \cdot 5 = 625)$$

There are direct analogues between the notions regarding $\sum$ and corresponding notions for $\prod$: the **for** loop interpretation (Figure 2.11), infinite products, reindexing, and nested products. One slight difference worthy of note: the value of $\prod_{i=1}^{0} x_i$ is 1; when we multiply by nothing, we're multiplying by one.

```
1 result := 1
2 for i := 1, 2, ..., n
3     result := result · x_i
4 return result
```

Figure 2.11 A **for** loop that returns the value of $\prod_{i=1}^{n} x_i$.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science.* This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Computer Science Connections

Integers and `int`s, Reals and `float`s

Every modern programming language has types corresponding to integers and real numbers, often called something like `int` (short for "integer") and `float` (short for *floating-point number*; more about this name and the floating point representation is below). In most programming languages, though, these types differ from $\mathbb{Z}$ and $\mathbb{R}$ in important ways.

Every piece of data stored on a computer is stored as a sequence of bits, and typically the bit sequence storing a number has some fixed length. For example, an `int` stored using 7 bits can range from 0000000 (the number 0 represented in binary) to 1111111 (the number $2^7 - 1 = 127$ represented in binary). Typically, the first bit in an `int`'s representation is reserved as the *sign bit* (set to True for a negative number and False for a positive number), and the remaining bits store the value of the number. (See Figure 2.12.) Thus there's a bound on the largest `int`, depending on the number of bits used to represent `int`s in a particular programming language: 32,767 in Pascal ($= 2^{15} - 1$, using 16 bits per `int`: 1 sign bit and 15 data bits), and 2,147,483,647 in Java ($= 2^{31} - 1$; 32 bits, of which 1 is a sign bit).

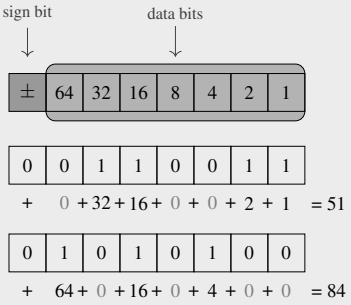


Figure 2.12 The values of each bit position, and the integers 51 and 84, represented as 8-bit signed integers.

A crucial point about $\mathbb{Z}$ and $\mathbb{R}$ is that they are *infinite*: there is no smallest integer, there's no biggest real number, and there isn't even a biggest real number that is smaller than 1. In almost every programming language, however, there is a smallest `int`, a biggest `float`, and a biggest `float` that's smaller than 1: after all, there are only finitely many possible `float`s (perhaps 2^{64} different values), and one of these 2^{64} values is the smallest `float`.

The finite nature of these programming language data types can cause some subtle bugs in programs. There are issues related to *integer overflow* if we try to store "too large" an integer: for example, when we compute 32767 + 1 in Pascal, the result is −32768. And there are bugs related to *underflow* if we try to store "too small" a floating-point number: for example, I tried to compute $(0.0000000001)^{33}$ in Python, and got 0.0 as the result. (But $(0.0000000001)^{32}$ was, correctly, 10^{-320}.) Similarly, there are rounding errors implicit in floating point representations of numbers: because there are only finitely many different `float`s, the infinitely many real numbers cannot all be stored exactly. For example, when I type `0.0006 - 0.0004 == 0.0002` into a Python interpreter, I get `False` as output. (According to Python, `0.0006 - 0.0004` is `0.00019999999999999993`, not `0.0002`.)

The name *float* originates with a clever idea that's used to mitigate these issues of imprecision: roughly, we allow the decimal point to "float" in the representation of different numbers. Perhaps the first idea for (approximately) representing real numbers in binary is a "fixed-point" representation: we reserve some fixed number of bits for the part of the number before the decimal point, and reserve some fixed number of bits for the part after the decimal point. But that can waste bits. Instead, in floating point representation, the idea is

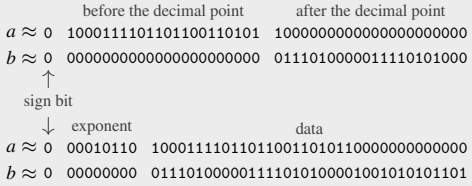


Figure 2.13 Fixed- and floating-point representations of $a = 2349877.5$ (the per-island average population of New Zealand, according to the most recent census) and $b = 0.45359237$ (the number of kilograms in a pound, according to the UK's Weights and Measures Act 1985). Floating point representation, at bottom, stores a as $2^{22} \cdot 0.56025445\cdots$ and b as $2^0 \cdot 0.45359237$, leaving room for more bits to represent the value itself more accurately.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

basically similar to scientific notation: we represent a number as the product of a power of two and a fraction between 0 and 1. (See Figure 2.13.)

(This description only roughly approximates the way that floating-point representation works to store numbers; see a good computer architecture textbook, such as [97], for more. Another interesting detail is the *2's complement* storage of integers, which allows a single representation of positive and negative integers so that addition "just works," even with a sign bit.)

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Computer Science Connections

Computing Square Roots, and Not Computing Square Roots

Many programmers happily use numerical operations without thinking about how they're implemented—but a little knowledge of what's happening behind the scenes can speed up programs. Computer hardware directly and efficiently executes basic arithmetic operations like $+$, $\cdot$, and $/$, but more complex calculations may require many of these basic operations. Consider computing $\sqrt{x}$ given an input value x, for example. A promising idea is to use some kind of *iterative improvement* algorithm: we start with a guess y_0 of the value of $\sqrt{x}$, and then update our guess to a new guess y_1 (by observing in some way whether y_0 was too big or too small). We continue to improve our guess until we've reached a value y such that y^2 is "close enough" to x. (We can specify the *tolerance* of the algorithm—that is, how close counts as "close enough.")

A simple implementation of this idea is called *Heron's method*, named after the 1st-century Greek mathematician Heron of Alexandria and shown in Figure 2.14. It relies on the nonobvious fact that the average of y and $\frac{x}{y}$ is closer to $\sqrt{x}$ than y was. (Unless y is exactly equal to $\sqrt{x}$.) Almost two millennia later, Isaac Newton developed a general technique for computing values of numerical expressions involving exponentials, among other things. This technique, known as *Newton's method*, uses calculus—specifically, using derivatives to figure out how far to move from a current guess y_i in making the next guess y_{i+1}. Like Heron's method, Newton's method is an example in *scientific computing*, the subfield of computer science devoted to efficient computation of numerical values, often for the purposes of simulating a complex system. (Many interesting questions and techniques are used in scientific computing; see [103] for one outstanding, and classic, reference for some of this material.)

Input: A positive real number x.
Output: A real number y with $y^2 \approx x$.
1 Let y_0 be arbitrary, and let $i := 0$.
2 **while** $(y_i)^2$ is too far from x:
3 let $y_{i+1} := \frac{y_i + \frac{x}{y_i}}{2}$ and $i := i + 1$
4 **return** y_i

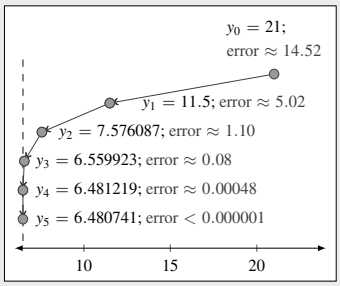


Figure 2.14 Heron's method for square roots, and an example of computing $\sqrt{x}$ for $x = 42$, using $\frac{x}{2}$ as the initial guess.

Work in scientific computing has improved the efficiency of numerical computation. But even better is to be aware of the fact that operations like square roots require significant computation "under the hood," and to avoid them when possible. To take one particular example, consider applying a *blur filter* to an image: replace each pixel p by the average of all pixels within a radius-r circle centered at p. (See Figure 2.15.) To compute

Figure 2.15 A blur filter, and Ada Lovelace (1815–1852), "the first computer programmer," who developed algorithms for the Analytical Engine, invented by Charles Babbage (1791–1871), the first (as-yet-theoretical) "computer." The second image of her is blurred with a radius of 9. For more on Lovelace and Babbage and their work, I cannot overstate the amazingness of Sydney Padua's *The Thrilling Adventures of Lovelace and Babbage* [95].

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

the blurred value at pixel p, we can just scan all pixels q in the square within $\pm r$ rows/columns of p, comparing the distance between p and q to r. The "obvious" approach is to follow that description verbatim: calculate $d := \sqrt{(p_x + q_x)^2 + (p_y + q_y)^2}$, and test whether $d \leq r$. But the alternative is to simply square both sides of the comparison, and instead test whether $(p_x + q_x)^2 + (p_y + q_y)^2 \leq r^2$. While there is no important mathematical difference between these two formulas, there *is* a computational difference. Because square roots are expensive to compute, it turns out that in my Python implementation of a blur filter, using the alternative approach was about 12% faster than using the "obvious" way—a big win for a small tweak.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

EXERCISES

2.1 What are the smallest and largest integers that are in the interval $(111, 202)$?

2.2 What are the smallest and largest integers that are in the interval $[111, 202)$?

2.3 What are the smallest and largest integers that are in the interval $(17, 42)$ but not in the interval $(39, 99]$?

2.4 What are the smallest and largest integers that are in the interval $[17, 42]$ but not in the interval $[39, 99)$?

2.5 If x and y are integers, is $x + y$ necessarily an integer? Explain.

2.6 If x and y are rational numbers, is $x + y$ necessarily rational? Explain.

2.7 If x and y are irrational numbers, is $x + y$ necessarily irrational? Explain.

2.8 What is the value of $\lfloor 2.5 \rfloor + \lceil 3.75 \rceil$?

2.9 What is the value of $\lfloor 3.14159 \rfloor \cdot \lceil 0.87853 \rceil$?

2.10 What is the value of $(\lfloor 3.14159 \rfloor)^{\lceil 3.14159 \rceil}$?

2.11 Most programming languages provide two different functions called *floor* and *truncate* to trim real numbers to integers. In these languages, $\texttt{floor}(x)$ is defined exactly as we defined $\lfloor x \rfloor$, and $\texttt{trunc}(x)$ is defined to simply delete any digits that appear after the decimal point in writing x. So $\texttt{trunc}(3.14159) = 3.14159 = 3$. Explain why programming languages might have both `floor` and `trunc`—that is, explain the circumstances under which $\texttt{floor}(x)$ and $\texttt{trunc}(x)$ give different values.

2.12 Using floor, ceiling, and standard arithmetic notation, give an expression for a real number x rounded to the nearest integer. ("Round up" for a number that's exactly between two integers—for example, 7.5 rounds to 8.)

2.13 Again using floor, ceiling, and standard arithmetic notation, give an expression for a real number x rounded to the nearest 0.1.

2.14 Generalize Exercise 2.13: give an expression rounding x to the nearest 10^{-k}, for any number k of digits after the decimal point.

2.15 Again using floor, ceiling, and standard arithmetic notation, give an expression for a real number x *truncated* to k digits after the decimal point—that is, leaving off the $(k+1)$st digit and beyond. (For example, 3.1415926 truncated with 3 digits is 3.141, and truncated with 4 digits is 3.1415.) [Many programming languages provide a facility for displaying formatted output in this style. For example, `printf("%.3f", x)` in C (or Java, or Python, or others) prints the value of `x` with only 3 digits after the decimal point. (The "f" of "printf" stands for formatted; the "f" of `"%.3f"` stands for float.)]

2.16 For what value(s) of x in the interval $[2, 3]$ is $x - \frac{\lfloor x \rfloor + \lceil x \rceil}{2}$ the largest?

2.17 For what value(s) of x in the interval $[2, 3]$ is $x - \frac{\lfloor x \rfloor + \lceil x \rceil}{2}$ the smallest?

2.18 Let x be a real number. Rewrite $\lfloor \lfloor x \rfloor \rfloor$ as simply as possible.

2.19 Rewrite $\lceil \lceil x \rceil \rceil$ as simply as possible.

2.20 Rewrite $\lfloor \lceil x \rceil \rfloor$ as simply as possible.

2.21 Rewrite $\lceil \lfloor x \rfloor \rceil$ as simply as possible.

2.22 Are $|\lfloor x \rfloor|$ and $\lfloor |x| \rfloor$ always equal? Explain.

2.23 Are $1 + \lfloor x \rfloor$ and $\lfloor 1 + x \rfloor$ always equal? Explain.

2.24 Are $\lfloor x \rfloor + \lfloor y \rfloor$ and $\lfloor x + y \rfloor$ always equal? Explain.

2.25 Let x be a real number. Describe (in English) what $1 + \lfloor x \rfloor - \lceil x \rceil$ represents. Explain.

2.26 In performing a binary search for x in a sorted n-element array $A[1 \ldots n]$ (see Figure 6.15b), the first thing we do is to compare the value of x and the value of $A\left[\left\lfloor \frac{1+n}{2} \right\rfloor\right]$. Assume that all elements of A are distinct. How many elements of A are *less than* $A\left[\left\lfloor \frac{1+n}{2} \right\rfloor\right]$? How many are *greater*? Write your answers as simply as possible.

Complete Exercises 2.27–2.40 by hand, without using a calculator or computer.

2.27 Which is bigger, 3^{10} or 10^3?

2.28 What is the value of 4^8?

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.29 What is the value of $(1/4)^8$?

2.30 What is the value of $(-4)^8$?

2.31 What is the value of $(-4)^9$?

2.32 What is the value of $256^{1/4}$?

2.33 What is the value of $8^{1/4}$?

2.34 What is the value of $8^{3/4}$?

2.35 What is the value of $(-9)^{1/4}$?

2.36 What is the value of $\log_2 8$?

2.37 What is the value of $\log_2(1/8)$?

2.38 What is the value of $\log_8 2$?

2.39 What is the value of $\log_{1/8} 2$?

2.40 Which is bigger, $\log_{10} 17$ or $\log_{17} 10$?

2.41 Using the definition of logarithms and the properties of exponentials from Theorem 2.8, justify Theorem 2.10.1: $\log_b 1 = 0$ for any $b > 1$.

2.42 Again using the definition of logarithms and Theorem 2.8, justify Theorem 2.10.2: $\log_b b = 1$ for any $b > 1$.

2.43 Do the same for Theorem 2.10.3: $\log_b xy = \log_b x + \log_b y$ for any $b > 1$ and $x > 0$ and $y > 0$.

2.44 Do the same for Theorem 2.10.5: $\log_b x^y = y \log_b x$ for any $b > 1$ and $x > 0$ and $y > 0$.

2.45 Do the same for Theorem 2.10.6: $\log_b x = \frac{\log_c x}{\log_c b}$ for any $b > 1$ and $c > 1$ and $x > 0$.

Using the results from Exercises 2.41–2.45 and the fact that $\log_b x = \log_b y$ *exactly when* $x = y$ *(for any base* $b > 1$*), justify the following additional properties of logarithms:*

2.46 For any real numbers $b > 1$ and $x > 0$, we have that $b^{[\log_b x]} = x$.

2.47 For any real numbers $b > 1$ and $a, n > 0$, we have that $n^{[\log_b a]} = a^{[\log_b n]}$.

2.48 Theorem 2.10.4: for any $b > 1$ and $x > 0$ and $y > 0$, we have that $\log_b \frac{x}{y} = \log_b x - \log_b y$.

2.49 Using notation defined in this chapter, define the "hyperceiling" $\overline{\lceil n \rceil}$ of a positive integer n, where $\overline{\lceil n \rceil}$ is the smallest exact power of two that is greater than or equal to n. (That is, $\overline{\lceil n \rceil}$ denotes the smallest value of 2^k where $2^k \geq n$ and k is a nonnegative integer.)

2.50 Similar to Exercise 2.49: when writing down an integer n on paper using standard decimal notation, we need enough columns for all the digits of n (and perhaps one additional column for a "$-$" if $n < 0$). Write down an expression indicating how many columns we need to represent n. *(Hint: use the case notation introduced in Definition 2.3, and be sure that your expression is well defined—that is, it doesn't "generate any errors"—for all integers n.)*

Complete Exercises 2.51–2.59 by hand, without using a calculator or computer.

2.51 What is the value of 202 mod 2?

2.52 What is the value of 202 mod 3?

2.53 What is the value of 202 mod 10?

2.54 What is the value of -202 mod 10?

2.55 What is the value of 17 mod 42?

2.56 What is the value of 42 mod 17?

2.57 What is the value of 17 mod 17?

2.58 What is the value of -42 mod 17?

2.59 What is the value of -42 mod 42?

2.60 Observe the Python behavior of the `%` operator (the Python notation for mod) that's shown in Figure 2.16. The first two lines ($3 \bmod 5 = 3$ and $-3 \bmod 5 = 2$) are completely consistent with the definition that we gave for mod (Definition 2.11), including

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

```
3 % 5       # value:  3
-3 % 5      # value:  2
3 % -5      # value:  -2
-3 % -5     # value:  -3
```

Figure 2.16 Python's implementation of % ("mod").

its use for n mod k when n is negative (as in Example 2.6). But we haven't defined what n mod k means for $k < 0$. Propose a formal definition of % in Python that's consistent with Figure 2.16.

2.61 What is the smallest positive integer n where $n \bmod 2 = 0$ and $n \bmod 3 = 0$, and $n \bmod 5 = 0$?

2.62 What is the smallest positive integer n where $n \bmod 2 = 1$ and $n \bmod 3 = 1$, and $n \bmod 5 = 1$?

2.63 What is the smallest positive integer n where $n \bmod 2 = 0$ and $n \bmod 3 = 1$, and $n \bmod 5 = 0$?

2.64 What is the smallest positive integer n where $n \bmod 3 = 2$ and $n \bmod 5 = 3$, and $n \bmod 7 = 5$?

2.65 What is the smallest positive integer n where $n \bmod 2 = 1$ and $n \bmod 3 = 2$ and $n \bmod 5 = 3$, and $n \bmod 7 = 4$?

2.66 *(programming required.)* Write a program to determine whether a given positive integer n is prime by testing all possible divisors between 2 and $n - 1$. Use your program to find all prime numbers less than 202.

2.67 *(programming required.)* A *perfect number* is a positive integer n that has the following property: n is equal to the sum of all positive integers $k < n$ that evenly divide n. For example, 6 is a perfect number, because 1, 2, and 3 are the positive integers less than 6 that evenly divide 6—and $6 = 1 + 2 + 3$. Write a program that finds the four smallest perfect numbers.

2.68 *(programming required.)* Write a program to find all integers between 1 and 1000 that are evenly divisible by *exactly three* different integers.

Compute the values of the following summations and products.

2.69 $\sum_{i=1}^{6} 6$

2.70 $\sum_{i=1}^{6} i^2$

2.71 $\sum_{i=1}^{6} 2^{2i}$

2.72 $\sum_{i=1}^{6} i \cdot 2^i$

2.73 $\sum_{i=1}^{6} (i + 2^i)$

2.74 $\prod_{i=1}^{6} 6$

2.75 $\prod_{i=1}^{6} i^2$

2.76 $\prod_{i=1}^{6} 2^{2i}$

2.77 $\prod_{i=1}^{6} i \cdot 2^i$

2.78 $\prod_{i=1}^{6} (i + 2^i)$

Compute the values of the following nested summations.

2.79 $\sum_{i=1}^{6} \sum_{j=1}^{6} (i \cdot j)$

2.80 $\sum_{i=1}^{6} \sum_{j=i}^{6} (i \cdot j)$

2.81 $\sum_{i=1}^{6} \sum_{j=1}^{i} (i \cdot j)$

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.82 $\sum_{i=1}^{8} \sum_{j=i}^{8} i$

2.83 $\sum_{i=1}^{8} \sum_{j=i}^{8} j$

2.84 $\sum_{i=1}^{8} \sum_{j=i}^{8} (i+j)$

2.85 $\sum_{i=1}^{4} \sum_{j=i}^{4} (j^i)$

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.3 Sets: Unordered Collections

> Our torments also may in length of time
> Become our Elements.
>
> John Milton (1608–1674)
> *Paradise Lost* (1667)

Section 2.2 introduced the primitive types of objects that we'll use throughout the book. We turn now to *collections* of objects, analogous to lists and arrays in programming languages. We start in this section with *sets*, in which objects are collected without respect to order or repetition. (Section 2.4 will address *sequences*, which are collections of objects in which order and repetition *do* matter.) The definitions and notation related to sets are summarized in Figure 2.17.

Definition 2.17: Sets.
A *set* is an unordered collection of objects.

(Sets are typically denoted by uppercase letters, often by a mnemonic letter: S for a set of students, D for a set of documents, etc. As we saw, the common sets from mathematics defined in Section 2.2.2 are often written using a "blackboard bold" font: $\mathbb{Z}$, $\mathbb{R}$, and $\mathbb{Q}$.) Here are a few small examples:

Example 2.22: Some sets.
Here are three sets: the set of bits $\{0, 1\}$, the set of prime numbers $\{2, 3, 5, 7, 11, \ldots\}$, and the set of basic arithmetic operators $\{+, -, \cdot, /\}$. (We've written these sets using standard notation by listing the objects in the set between curly braces { and }.)

Set membership—that is, the question *is the object x one of the objects in the collection S?,* for a particular object x and a particular set S—is the central notion for sets:

Definition 2.18: Set membership.
For a set S and an object x, the expression $x \in S$ is true when x is one of the objects contained in the set S. When $x \in S$, we say that x is an *element* or *member* of S or, more simply, that x is *in* S.

The expression $x \notin S$ is the negation of the expression $x \in S$: that is, $x \notin S$ is true whenever x is not an element of S (and thus whenever $x \in S$ is false).

Example 2.23: Some set memberships.
The integer 0 is an element of the set of bits, and $+$ is in the set of basic arithmetic operators. But 1 is not an element of the set of prime numbers, and 8 is not in the set of bits.

A second key concept about a set is its *cardinality*, or *size*:

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

set membership	$x \in S$	x is one of the elements of S
cardinality	$\|S\|$	the number of distinct elements in the set S
set enumeration	$\{x_1, x_2, \ldots, x_k\}$	the set containing elements $x_1, x_2, \ldots, x_k$
set abstraction	$\{x \in U : P(x)\}$	the set containing all $x \in U$ for which $P(x)$ is true; U is the "universe" of candidate elements
empty set	$\{\}$ or $\varnothing$	the set containing no elements
complement	$\sim S = \{x \in U : x \notin S\}$	the set of all elements in the universe U that aren't in S; U may be left implicit if it's obvious from context
union	$S \cup T = \{x : x \in S \text{ or } x \in T\}$	the set of all elements in either S or T (or both)
intersection	$S \cap T = \{x : x \in S \text{ and } x \in T\}$	the set of all elements in both S and T
set difference	$S - T = \{x : x \in S \text{ and } x \notin T\}$	the set of all elements in S but not in T
set equality	$S = T$	every $x \in S$ is also in T, and every $x \in T$ is also in S
subset	$S \subseteq T$	every $x \in S$ is also in T
proper subset	$S \subset T$	$S \subseteq T$ but $S \neq T$
superset	$S \supseteq T$	every $x \in T$ is also in S
proper superset	$S \supset T$	$S \supseteq T$ but $S \neq T$
power set	$\mathscr{P}(S)$	the set of all subsets of S

Figure 2.17 A summary of set notation.

Definition 2.19: Set cardinality.
The *cardinality* of a set S, denoted by $|S|$, is the number of distinct elements in S.

Example 2.24: Some set sizes.
The cardinality of the set of bits is 2, because there are two distinct elements of that set (namely 0 and 1).

The cardinality of the set S of prime numbers between 10 and 20 is $|S| = 4$: the four elements of S are 11, 13, 17, and 19.

Chapter 9 is devoted entirely to the apparently trivial problem of *counting*—given a (possibly convoluted) description of a set S, find $|S|$—which turns out to have some interesting and useful applications, and isn't as easy as it seems.

Taking it further: In this book, we will be concerned almost exclusively with the cardinality of *finite* sets, but one can also ask questions about the cardinality of sets like $\mathbb{Z}$ or $\mathbb{R}$ that contain an infinite number of distinct elements. For example, it's possible to prove that $|\mathbb{Z}| = |\mathbb{Z}^{\geq 0}|$, which is a pretty amazing result: *there are as many nonnegative integers as there are integers!* (And that's true despite the fact that every nonnegative integer *is* an integer!) But it's also possible to prove that $|\mathbb{Z}| \neq |\mathbb{R}|$: . . . *but there are more real numbers than integers!* More amazingly, one can use similar ideas to prove that there are fewer computer programs than there are problems to solve, and that therefore there are some problems that are not solved by any computer program. This idea is the central focus of the study of *computability* and *uncomputability*. See Section 4.4.4 and p. 9-46.

2.3.1 Building Sets from Scratch

There are two standard ways to specify a set "from scratch": by simply listing each of the elements of the set, or by defining the set as the collection of objects for which a particular logical condition is true.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Set definition via exhaustive enumeration

A set can be specified using an exhaustive listing of its elements—that is, by writing a complete list of its elements inside the curly braces { and }. Here are a few examples:

Example 2.25: Some exhaustively enumerated sets.

- The set of even prime numbers is $\{2\}$.
- The set of prime numbers between 10 and 20 is $\{11, 13, 17, 19\}$.
- The set of 2-digit perfect squares is $\{81, 64, 25, 16, 36, 49\}$.
- The set of bits is $\{0, 1\}$.
- The set of Turing Award winners in 2004, 2008, 2012, and 2016 is {Tim Berners-Lee, Vint Cerf, Shafi Goldwasser, Bob Kahn, Barbara Liskov, Silvio Micali}.

Taking it further: The Turing Award is the most prestigious award given in computer science—the "Nobel Prize of CS," it's sometimes called. The award, named after the pioneering computer scientist Alan Turing (1912–1954)—also a WWII-era code-breaking hero; see p. 9-75—has been awarded annually since 1966 to an individual or small group of collaborators for "major contributions of lasting importance" to the field. The detailed citations and biographies of the winners are available at `https://amturing.acm.org`.

Vint Cerf and Bob Kahn (2004) were honored for their collaboration, starting in the early 1970s, in developing the Transmission Control Protocol and Internet Protocol (TCP/IP), which form the very foundation of the internet. Barbara Liskov (2008) earned her Turing Award for her foundational work on design of computer systems and programming languages, especially on data abstraction and modular programming, and on fault tolerance and distributed computing. Shafi Goldwasser and Silvio Micali (2012) were jointly honored for their work in cryptography, computational complexity, and probabilistic algorithms, which began while they were earning graduate degrees at Berkeley (his in 1982 and hers in 1984) and continued when they both became faculty members at MIT. Sir Tim Berners-Lee (2016) won his award for inventing the World Wide Web—and some highly influential web-centered protocols—around 1990 when he worked at CERN, a physics lab in Geneva.

Recall that a set is an *unordered* collection, and thus the order in which the elements are listed doesn't matter when specifying a set via exhaustive enumeration. Any repetition in the listed elements is also unimportant. For example:

Example 2.26: The same set, three ways.
The set $\{2+2,\ 2\cdot 2,\ 2/2,\ 2-2\}$ is precisely identical to the set $\{0, 1, 4\}$, both of which are precisely identical to $\{4, 0, 1\}$. Also note that $|\{2+2,\ 2\cdot 2,\ 2/2,\ 2-2\}| = 3$; despite there being four entries in the list of elements, there are only three *distinct* objects in the set.

It's important to remember that the integer 2 and the set $\{2\}$ are two entirely different kinds of things. For example, note that $2 \in \{2\}$, but that $\{2\} \notin \{2\}$; the lone element in $\{2\}$ is *the number two*, not *the set containing the number two*.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Set definition via set abstraction

Instead of explicitly listing all of a set's elements, we can also define a set in terms of a condition that is true for the elements of the set and that's false for every object that is not an element of the set. Defining a set this way uses *set abstraction* notation:

Definition 2.20: Set abstraction.
Let U be a set of possible elements, called the *universe*. Let $P(x)$ be a condition (also called a *predicate*) that, for every $x \in U$, is either true or false. Then

$$\{x \in U : P(x)\}$$

denotes the set of all objects $x \in U$ for which $P(x)$ is true.

That is, for a candidate $y \in U$, the element y is a member of the set $\{x \in U : P(x)\}$ when $P(y) =$ True, and $y \notin \{x \in U : P(x)\}$ when $P(y) =$ False. (A fully proper version of Definition 2.20 requires *functions*, described in Section 2.5.)

(The colon in the notation for set abstraction is read as "such that," so the set in Definition 2.20 would be read "the set of all x in U such that P of x.")

Example 2.27: Most of Example 2.25, redone.

- The set of even prime numbers is $\{x \in \mathbb{Z}^{>1} : x \text{ is prime and } x \text{ is even}\}$.
- The set of 2-digit perfect squares is $\{n \in \mathbb{Z} : \sqrt{n} \in \mathbb{Z} \text{ and } 10 \leq n \leq 99\}$.
- The set of bits is $\{b \in \mathbb{Z} : b^2 = b\}$.

For this set abstraction notation to meaningfully define a set S, we must specify the universe U of candidates from which the elements of S are drawn. We will permit ourselves to be sloppy in our notation, and when the universe U is clear from context we will allow ourselves to write $\{x : P(x)\}$ instead of $\{x \in U : P(x)\}$.

Taking it further: The notational sloppiness of omitting the universe in set abstraction will be a convenience for us, and it will not cause us any trouble—but it turns out that one must be careful! In certain strange scenarios when defining sets, there are subtle but troubling paradoxes that arise if we allow the universe to be anything at all. The key problem can be seen in *Russell's paradox*, named after the British philosopher/mathematician Bertrand Russell (1872–1970); Russell's discovery of this paradox revealed an inconsistency in the commonly accepted foundations of mathematics in the early 20th century.

Here is a sketch of Russell's Paradox. Let X denote the set of all sets that do not contain themselves: that is, let $X = \{S : S \notin S\}$. For example, $\{2\} \in X$ because $\{2\} \notin \{2\}$, and $\mathbb{R} \in X$ because $\mathbb{R}$ is not a real number, so $\mathbb{R} \notin \mathbb{R}$. On the other hand, if we let T^* denote the set of all sets, then $T^* \notin X$: because T^* is a set, and T^* contains all sets, then $T^* \in T^*$ and therefore $T^* \notin X$. Here's the problem: is $X \in X$? Suppose that $X \in X$: then $X \in \{S : S \notin S\}$ by the definition of X, and thus $X \notin X$. But suppose that $X \notin X$; then, by the definition of X, we have $X \in X$. So if $X \in X$ then $X \notin X$, and if $X \notin X$ then $X \in X$—but that's absurd!

One standard way to escape this paradox is to say that the set X cannot be defined—because, to be able to define a set using set abstraction, we need to start from a defined universe of candidate elements. (And the set T^* cannot be defined either.) The *Liar's Paradox*, dating back about 3000 years, is a similar paradox: is "this sentence is false" true (nope!) or false (nope!)? In both

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Russell's Paradox and the Liar's Paradox, the fundamental issue relates to *self-reference*; many other mind-twisting paradoxes are generated through self-reference, too. For more on these and other paradoxes, see [111].

Definition 2.20 lets us write $\{x \in U : P(x)\}$ to denote the set containing exactly those elements x of U for which $P(x)$ is True. We will extend this notation to allow ourselves to write more complicated expressions to the left of the colon, as in the following example:

Example 2.28: 2-digit perfect squares, again (see Examples 2.25 and 2.27).
We can also write the set of 2-digit perfect squares as either $\{x^2 : x \in \mathbb{Z} \text{ and } 10 \leq x^2 \leq 99\}$ or as $\{x^2 : x \in \{4,5,6,7,8,9\}\} = \{4^2, 5^2, 6^2, 7^2, 8^2, 9^2\}$.

To properly define this extended form of the set-abstraction notation, we again need the idea of *functions*, which are defined in Section 2.5. See Definition 2.49 for a proper definition of this extended notation.

Taking it further: Modern programming languages support the use of *lists* to store a collection of objects. While these lists store ordered collections, there are some very close parallels between these lists and sets. In fact, the ways we've described building sets have very close connections to ideas in certain programming languages like Scheme and Python; see p. 2-40.

The empty set

One particularly useful set—despite its simplicity—is the *empty set*, also sometimes called the *null set*:

Definition 2.21: The empty set.
The *empty set*, denoted $\{\}$ or $\varnothing$, is the set that contains no elements.

The definition of the empty set as $\{\}$ *is* an exhaustive listing of all of the elements of the set—though, because there aren't any elements, there are no elements in the list.

Alternatively, we could have used the set abstraction notation to define the empty set as $\varnothing = \{x : \text{False}\}$. This definition may seem initially confusing, but it's in fact a direct application of Definition 2.20: the condition P for this set is $P(x) = \text{False}$ (that is: for every object x, the value of $P(x)$ is False), and we've defined $\varnothing$ to contain every object y such that $P(y) = \text{True}$. But there *isn't* any object y such that $P(y) = \text{True}$—because $P(y)$ is always false—and thus there's no $y \in \{x : P(x)\}$.

Notice that, because there are zero elements in $\varnothing$, its cardinality is zero: in other words, $|\varnothing| = 0$. One other special type of set is defined based on its cardinality; a *singleton set* is a set S that contains exactly one element—that is, a set S such that $|S| = 1$.

2.3.2 Building Sets from Other Sets

There are a number of ways to create new sets from two sets A and B. We will define these operations formally, but it is sometimes more intuitive to look at a visual representation of sets called a *Venn diagram*, which represent sets as circular "blobs" that contain points (elements), enclosed in a rectangle that denotes the universe. (Venn diagrams are named after the British logician/philosopher John Venn (1834–1923).)

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Example 2.29: Venn diagram of odds and primes.
Let $U = \{1, 2, \ldots, 10\}$. Let $P = \{2, 3, 5, 7\}$ denote the set of primes in U, and let $O = \{1, 3, 5, 7, 9\}$ denote the set of odd numbers in U. A Venn diagram illustrating these sets is shown in Figure 2.18.

We will now define four standard ways of building a new set in terms of one or two existing sets: *complement, union, intersection,* and *set difference.*

Definition 2.22: Set complement.
The *complement* of a set A with respect to the universe U, written $\sim A$ (or sometimes $\overline{A}$), is the set of all elements *not* contained within A. Formally, $\sim A = \{x \in U : x \notin A\}$. (When the universe is obvious from context, we may leave it implicit.)

For example, if the universe is $\{1, 2, \ldots, 10\}$, then $\sim\{1, 2, 3\} = \{4, 5, 6, 7, 8, 9, 10\}$ and $\sim\{3, 4, 5, 6\} = \{1, 2, 7, 8, 9, 10\}$. (See Figure 2.19a.)

Definition 2.23: Set union.
The *union* of two sets A and B, denoted $A \cup B$, is the set of all elements in *either A or B* (or both). Formally, $A \cup B = \{x : x \in A \text{ or } x \in B\}$.

For example, $\{1, 2, 3\} \cup \{3, 4, 5, 6\} = \{1, 2, 3, 4, 5, 6\}$. (See Figure 2.19b.)

Definition 2.24: Set intersection.
The *intersection* of two sets A and B, denoted $A \cap B$, is the set of all elements in *both A and B*. Formally, $A \cap B = \{x : x \in A \text{ and } x \in B\}$.

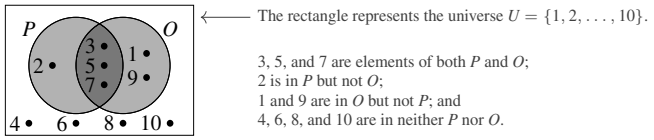
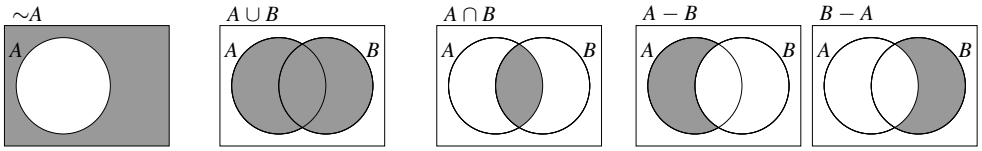


Figure 2.18 A Venn diagram for the set O of odd numbers and the set P of prime numbers between 1 and 10.



(a) The complement $\sim A$ of a set A.

(b) The union $A \cup B$ of two sets A and B.

(c) The intersection $A \cap B$ of A and B.

(d) The difference of A and B. On the left is the set $A - B$; on the right is $B - A$.

Figure 2.19 Venn diagrams of the basic set operations: complement, union, intersection, and set difference. (The complement is taken with respect to the universe U, which is represented by the rectangle.)

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

For example, $\{1,2,3\} \cap \{3,4,5,6\} = \{3\}$. (See Figure 2.19c.)

Analogously to summation and product notation ($\sum$ and $\prod$), we will sometimes write $\bigcup_{i=1}^{n} S_i$ to denote $S_1 \cup S_2 \cup \cdots \cup S_n$, and we will sometimes write $\bigcap_{i=1}^{n} S_i$ to denote $S_1 \cap S_2 \cap \cdots \cap S_n$.

Definition 2.25: Set difference.
The *difference* of two sets A and B, denoted $A - B$, is the set of all elements contained *in the set A but not in the set B*. Formally, $A - B = \{x : x \in A \text{ and } x \notin B\}$. (Note that $A - B$ and $B - A$ are different sets.)

(Some people write $A \setminus B$ instead of $A - B$ to denote set difference.)

For example, $\{1,2,3\} - \{3,4,5,6\} = \{1,2\}$ and $\{3,4,5,6\} - \{1,2,3\} = \{4,5,6\}$. (See Figure 2.19d.)

In expressions that use more than one of these set operators, the $\sim$ operator "binds tightest"—that is, in an expression like $\sim S \cup T$, we mean $(\sim S) \cup T$ and not $\sim(S \cup T)$. We use parentheses to specify the order of operations among $\cap$, $\cup$, and $-$. Here's a slightly more complicated example that combines set operations:

Example 2.30: Combining odds and primes.
As in Example 2.29, define $U = \{1,2,\ldots,10\}$, with $P = \{2,3,5,7\}$ and $O = \{1,3,5,7,9\}$ as the primes and odd numbers in U. List the elements of the sets $P \cap \sim O$ and $\sim(P \cup O)$ and $\sim P - \sim O$.

Solution. For each part, we plug in the definitions:

$$P \cap \sim O = \{2,3,5,7\} \cap \underbrace{\sim\{1,3,5,7,9\}}_{=\{2,4,6,8,10\}} = \{2\}$$

$$\sim(P \cup O) = \sim(\underbrace{\{2,3,5,7\} \cup \{1,3,5,7,9\}}_{=\{1,2,3,5,7,9\}}) = \{4,6,8,10\}$$

$$\sim P - \sim O = \underbrace{\sim\{2,3,5,7\}}_{=\{1,4,6,8,9,10\}} - \underbrace{\sim\{1,3,5,7,9\}}_{=\{2,4,6,8,10\}} = \{1,9\}.$$

Thus $P \cap \sim O = \{2\}$ is the set of prime nonodd numbers; $\sim(P \cup O) = \{4,6,8,10\}$ is the set of numbers that *aren't* either prime or odd; and $\sim P - \sim O = \{1,9\}$ is the set of nonprime numbers that aren't nonodd, or, more simply stated, all nonprime odd numbers.

Of course, we can also combine more than two sets in expressions using these set operators—for example, $A \cup B \cup C$ denotes the set $\{x : x \in A \text{ or } x \in B \text{ or } x \in C\}$. We can use Venn diagrams to visualize set operations that involve more than two sets; see Figure 2.20 for a few examples.

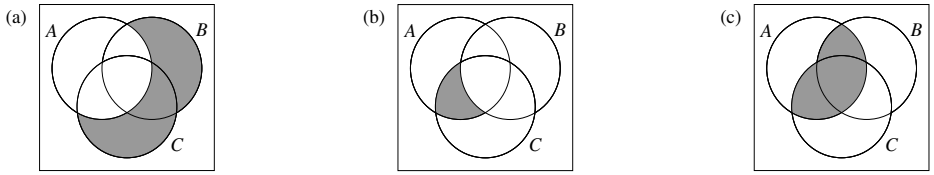


Figure 2.20 Some three-set Venn diagrams: (a) $(B \cup C) - A$; (b) $(A - B) \cap C$; and (c) $A \cap (B \cup C)$.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Arithmetic operations on sets

We'll wrap up with a bit of notation that will allow us to perform mathematical operations on the elements of a set. In Section 2.2.7, we introduced summation and product notation, so that we could write $\sum_{i=1}^{n} x_i$ to represent $x_1 + x_2 + \cdots + x_n$, and $\prod_{i=1}^{n} x_i$ to represent $x_1 \cdot x_2 \cdot \cdots \cdot x_n$. We will sometimes wish to represent the sum or product of the elements of a particular set (instead of a sequence of values like $x_1, x_2, \ldots, x_n$). It will also be handy to refer to the smallest or largest element in a set.

Definition 2.26: Sum, product, minimum, and maximum of a set.
Let S be a set. Then the expressions

$$\sum_{x \in S} x \quad \text{and} \quad \prod_{x \in S} x \quad \text{and} \quad \min_{x \in S} x \quad \text{and} \quad \max_{x \in S} x$$

respectively denote the sum of the elements of S, the product of the elements of S, the smallest element in S, and the largest element in S.

For example, for the set $S = \{1, 2, 4, 8\}$, the sum of the elements of S is $\sum_{x \in S} x = 15$; the product of the elements of S is $\prod_{x \in S} x = 64$; the minimum of S is $\min_{x \in S} x = 1$; and the maximum of S is $\max_{x \in S} x = 8$.

2.3.3 Comparing Sets

In the same way that two numbers x and y can be compared (we can ask questions like: does $x = y$? is $x \leq y$? is $x \geq y$?), we can also compare two sets A and B. Here, we will define the analogous notions of comparison for sets. We'll begin by defining what it means for two sets to be equal:

Definition 2.27: Set equality.
Two sets A and B are *equal*, denoted $A = B$, if A and B have exactly the same elements. (In other words, sets A and B are not equal if there's an element $x \in A$ but $x \notin B$, or if there's an element $y \in B$ but $y \notin A$.)

This definition formalizes the idea that order and repetition don't matter in sets: for example, the sets $\{4, 4\}$ and $\{4\}$ are equal because there is no element $x \in \{4, 4\}$ where $x \notin \{4\}$ and there is no element $y \in \{4\}$ where $y \notin \{4, 4\}$. This definition also implies that the empty set is unique: any set containing no elements is identical to $\varnothing$.

> **Taking it further:** Definition 2.27 is sometimes called the *axiom of extensionality*. (All of mathematics, including a completely rigorous definition of the integers and all of arithmetic, can be built up from a small number of axioms about sets, including this one.) The point is that the only way to compare two sets is by their "externally observable" properties. For example, the following two sets are *exactly* the same set: $\{x : x > 10 \text{ is an even prime number}\}$, and $\{y : y \text{ is a country with a 128-letter name}\}$. (Namely, both of these sets are $\varnothing$.)

The other common type of comparison between two sets A and B is the *subset* relationship, which expresses that every element of A is also an element of B:

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Definition 2.28: Subset.
A set A is a *subset* of a set B, written $A \subseteq B$, if every $x \in A$ is also an element of B. (In other words, $A \subseteq B$ is equivalent to $A - B = \{\}$.)

For example, $\{1,3,5\} \subseteq \{1,2,3,4,5\}$, because $1 \in \{1,2,3,4,5\}$ and $3 \in \{1,2,3,4,5\}$ and $5 \in \{1,2,3,4,5\}$. Similarly, $\{1,3,5\} \subseteq \{1,3,5\}$.

Notice that $\{\} \subseteq S$ for *any* set S: it's impossible for there to be an $x \in \{\}$ that satisfies $x \notin S$, because there is no element $x \in \{\}$ in the first place—and if there's no $x \in \{\}$ at all, then there's certainly no $x \in \{\}$ such that $x \notin S$.

Definition 2.29: Proper subset.
A set A is a *proper subset* of a set B, written $A \subset B$, if $A \subseteq B$ and $A \neq B$. In other words, $A \subset B$ whenever $A \subseteq B$ but $B \not\subseteq A$.

For example, let $A = \{1,2,3\}$. Then $A \subseteq \{1,2,3,4\}$ and $A \subseteq \{1,2,3\}$ and $A \subset \{1,2,3,4\}$, but A is not a proper subset of $\{1,2,3\}$.

When $A \subset B$ or $A \subseteq B$, we refer to A as the (possibly proper) subset of B; we can also call B the (possibly proper) *superset* of A:

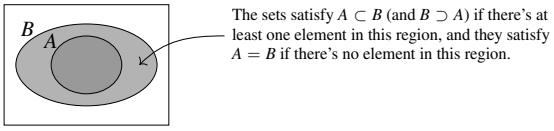
Definition 2.30: Superset and proper superset.
Let A be a set. A set B is a *superset* of A, written $B \supseteq A$, if $A \subseteq B$. The set B is a *proper superset* of A, written $B \supset A$, if $A \subset B$.

Figure 2.21a illustrates subsets, proper subsets, supersets, and proper supersets.

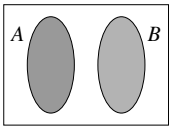
Example 2.31: Subsets and supersets.
Let $A = \{3,4,5\}$ and $B = \{4,5,6\}$. Identify a set C satisfying the following conditions, or state that the requirement is impossible to achieve and explain why.

1 $A \subseteq C$ and $C \supseteq B$
2 $A \supseteq C$ and $C \subseteq B$
3 $A \supseteq C$ and $C \supseteq B$

Solution. The first two conditions are achievable, but the third isn't.



(a) Two sets satisfying $A \subseteq B$ and, equivalently, $B \supseteq A$.



(b) Two disjoint sets A and B.

Figure 2.21 One set that's a subset of another, and two sets that are disjoint.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

1. Let $C = \{3, 4, 5, 6\}$; both A and B are (proper) subsets of this set.
2. We can choose $C = \{4, 5\}$, because $\{4, 5\} \subseteq A$ and $\{4, 5\} \subseteq B$.
3. It's impossible to satisfy $\{3, 4, 5\} \supseteq C$ and $C \supseteq \{4, 5, 6\}$ simultaneously. If $6 \in C$ then we don't have $\{3, 4, 5\} \supseteq C$, but if $6 \notin C$ we don't have $C \supseteq \{4, 5, 6\}$. We can't have $6 \in C$ and we can't have $6 \notin C$, so we're stuck with an impossibility.

Here's one last piece of set-related terminology that we'll use: two sets A and B are called *disjoint* if they have no elements in common.

Definition 2.31: Disjoint sets.
Two sets A and B are *disjoint* if there is no $x \in A$ where $x \in B$—in other words, if $A \cap B = \{\}$.

For example, the sets $\{1, 2, 3\}$ and $\{4, 5, 6\}$ are disjoint because $\{1, 2, 3\} \cap \{4, 5, 6\} = \{\}$, but the sets $\{2, 3, 5, 7\}$ and $\{2, 4, 6, 8\}$ are not disjoint because 2 is an element of both. See Figure 2.21b.

2.3.4 Sets of Sets

Just as we can have a list of lists in a programming language like Scheme or Java, we can also consider a set that has sets as its elements. (After all, sets are just collections of objects, and one kind of object that can be collected is a set itself.)

Example 2.32: Set of sets of numbers.
The set $A = \{\mathbb{Z}, \mathbb{R}, \mathbb{Q}\}$ of the sets defined in Section 2.2.2 is itself a set. This set has cardinality $|A| = 3$, because A has three distinct elements—namely $\mathbb{Z}$ and $\mathbb{R}$ and $\mathbb{Q}$. (Of course, all three of these elements of A are themselves sets, and each of these three elements of A has infinite cardinality.)

Example 2.33: A set of smaller sets.
Consider the set $B = \{\{\}, \{1, 2, 3\}\}$. Note that $|B| = 2$: B has two elements, namely $\{\}$ and $\{1, 2, 3\}$. Therefore $\{\} \in B$ because $\{\}$ is one of the two elements of B. However $1 \notin B$, because 1 is not one of the two elements of B—that is, $1 \neq \{\}$ and $1 \neq \{1, 2, 3\}$—although 1 *is* an element of one of the two elements of B.

There are two important types of sets of sets that we will define in the remainder of this section, both derived from a base set S.

Partitions

The first interesting use of a set of sets is to form a *partition* of S into a set of disjoint subsets whose union is precisely S.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Definition 2.32: Partition.
A *partition* of a set S is a set $\{A_1, A_2, \ldots, A_k\}$ of nonempty sets $A_1, A_2, \ldots, A_k$, for some $k \geq 1$, such that (i) $A_1 \cup A_2 \cup \cdots \cup A_k = S$; and (ii) for any i and $j \neq i$, the sets A_i and A_j are disjoint.

A useful way of thinking about a partition of a set S is that we've divided S up into several (nonoverlapping) subcategories. See Figure 2.22 for an illustration of a partition of a set S. Here's an example of one set partitioned many different ways:

Example 2.34: Several partitions of the same set.
Consider the set $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Here are some different ways to partition S:

$$\{\{1,3,5,7,9\},\{2,4,6,8,10\}\} \qquad \textit{evens and odds}$$

$$\{\{1,2,3,4,5,6,7,8,9\},\{10\}\} \qquad \textit{one- and two-digit numbers}$$

$$\{\{1,4,7,10\},\{2,5,8\},\{3,6,9\}\} \qquad x \bmod 3 = 0 \textit{ and } x \bmod 3 = 1 \textit{ and } x \bmod 3 = 2$$

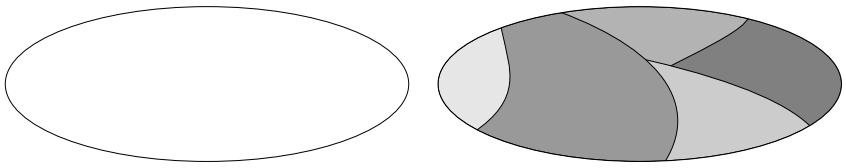
$$\{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\},\{8\},\{9\},\{10\}\} \qquad \textit{all separate}$$

$$\{\{1,2,3,4,5,6,7,8,9,10\}\} \qquad \textit{all together}$$

In each case, each of the 10 numbers from S is in one, and only one, of the listed sets (and no elements not in S appear in any of the listed sets).

It's worth noting that the last two ways of partitioning S in Example 2.34 genuinely *are* partitions. For the partition $\{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\},\{8\},\{9\},\{10\}\}$, we have $k = 10$ different disjoint sets whose union is precisely S. For the partition $\{\{1,2,3,4,5,6,7,8,9,10\}\}$, we have $k = 1$: there's only one "subcategory" in the partitioning, and every $x \in S$ is indeed contained in one (the only one!) of these "subcategories." (And no two distinct subcategories overlap, because there aren't even two distinct subcategories at all!)

Taking it further: One way to helpfully organize a massive set S of data—for example, students or restaurants or web pages—is to partition S into small *clusters*. The idea is that two elements in the same cluster will be "similar," and two entities in different clusters will be "dissimilar." (So students might be clustered by their majors or dorms; restaurants might be clustered by their cuisine or geography; and web pages might be clustered based on the set of words that appear in them.) For more, see p. 2-42.



(a) The set S. (b) S partitioned into 5 subsets.

Figure 2.22 A visualization of partitioning a set S into disjoint nonempty subsets whose union equals S itself.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Power sets

Our second important type of a set of sets is the *power set* of a set S, which is the set of all subsets of S:

Definition 2.33: Power set.
The *power set* of a set S, written $\mathscr{P}(S)$, denotes the set of all subsets of S: that is, a set A is an element of $\mathscr{P}(S)$ precisely if $A \subseteq S$. In other words, $\mathscr{P}(S) = \{A : A \subseteq S\}$.

The power set of S is also occasionally denoted by 2^S, in part because—as we'll see in Chapter 9—$|\mathscr{P}(S)|$ is $2^{|S|}$. The name "power set" also comes from this fact: the cardinality of $\mathscr{P}(S)$ is 2 to the power of $|S|$.

Here are some small examples, and one example that's a bit more complicated:

Example 2.35: Some small power sets.
Here are the power sets of $\{0\}$, $\{0,1\}$, and $\{0,1,2\}$:

$$\begin{aligned}\mathscr{P}(\{0\}) &= \{\{\}, \{0\}\}\\ \mathscr{P}(\{0,1\}) &= \{\{\}, \{0\}, \{1\}, \{0,1\}\}\\ \mathscr{P}(\{0,1,2\}) &= \{\{\}, \{0\}, \{1\}, \{2\}, \{0,1\}, \{0,2\}, \{1,2\}, \{0,1,2\}\}\end{aligned}$$

For the second of these examples: there are four elements in $\mathscr{P}(\{0,1\})$—the empty set, two singleton sets $\{0\}$ and $\{1\}$, and the two-element set $\{0,1\}$ itself, because $\{0,1\} \subseteq \{0,1\}$ is a subset of itself.

Example 2.36: $\mathscr{P}(\mathscr{P}(\{0,1\}))$.
The power set of the power set of $\{0,1\}$ is

$$\mathscr{P}(\mathscr{P}(\{0,1\})) = \mathscr{P}(\{\{\}, \{0\}, \{1\}, \{0,1\}\});$$

the 16 elements of this set are shown in Figure 2.23.

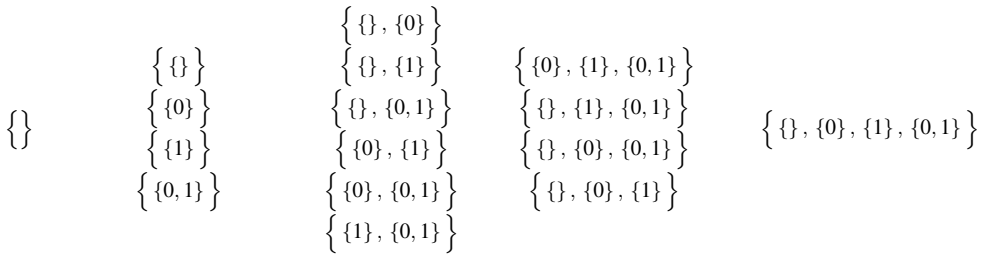


Figure 2.23 The power set of the power set of $\{0,1\}$. There are a total of $1+4+6+4+1 = 16$ sets in $\mathscr{P}(\mathscr{P}(\{0,1\}))$: one with 0 elements, four with 1 element, six with 2 elements, four with 3 elements, and one with 4 elements.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Computer Science Connections

Set Building in Languages (and MapReduce)

Programming languages like Python, Scheme, or ML make heavy use of lists and also allow higher-order functions (functions that take other functions as parameters). The experience of programming in these languages sometimes feels very similar to using the set-construction notions from Section 2.3.1. These mechanisms for building sets in mathematical notation closely parallel built-in functionality for building *lists* in programs in these languages:

- build a list from scratch by writing out its elements.
- build a list from an existing list using the function `filter`, which takes two parameters (a list `U`, corresponding to the universe, and a function `P`) and returns a new list containing all $x \in$ `U` for which `P`(x) is true.
- build a list from an existing list using the function `map`, which takes two parameters (a list `U` and a function `f`) and returns a new list containing `f`(x) for every element x of `U`.

(Unlike sets, the `map` function can cause repetitions in the stored list: `map(square,L)` where `L` contains both 2 and -2 will lead to 4 being present twice. But some languages, including Python, also have syntax for *sets* instead of *lists,* creating an unordered, duplicate-free collection of elements.)

```
1  # Some helper functions to use as predicates.
2  def even(x):    return x % 2 == 0
3  def square(x):  return x**2
4  def false(x):   return False
5
6  # The definitions.
7  L = [1, 2, 4, 8, 16]                          L = {1, 2, 4, 8, 16}
8  M = [x for x in L if x < 10]                  M = {x ∈ L : x < 10} = {1, 2, 4, 8}
9  N = filter(even, L)                           N = {x ∈ L : x is even} = {2, 4, 8, 16}
10 O = map(square, L)                            O = {x² : x ∈ L} = {1, 4, 16, 64, 256}
11 P = [square(x) for x in L if even(x)]         P = {x² : x ∈ L and x is even}
12                                                   = {4, 16, 64, 256}
13 Q = [x for x in L if false(x)]                Q = {x ∈ L : False} = {}
```

Figure 2.24 List-building code in Python analogous to our set-building notation. (Lines 8, 11, and 13 use Python's list comprehension syntax.)

Python has `filter` and `map` built in; Scheme has `filter` and `map` either built in or in a standard library. (In Python, there's even an explicit *list comprehension* syntax to create a list without using `filter` or `map`, which even more closely parallels the set-abstraction notation from Definitions 2.20 and 2.49.) See Figure 2.24 for Python, and Figure 2.25 for Scheme.

While the technical details are a bit different, the basic idea underlying `map` forms half of a programming model called *MapReduce* that's become increasingly popular for processing very large datasets. (The `reduce` function in Python or Scheme lets a programmer specify how to take a list of values and "reduce" it to a single value—for example, by summing.) MapReduce is a distributed-computing framework

```
1  ;;; Some helper functions to use as predicates.
2  (define even? (lambda (x) (= (modulo x 2) 0)))
3  (define square (lambda (x) (* x x)))
4  (define false? (lambda (x) #f))
5
6  ;;; The definitions.
7  (define L (list 1 2 4 8 16))                 L = {1, 2, 4, 8, 16}
8
9  (define N (filter even? L))                   N = {x ∈ L : x is even} = {2, 4, 8, 16}
10 (define O (map square L))                     O = {x² : x ∈ L} = {1, 4, 16, 64, 256}
11 (define P (map square (filter even? L)))      P = {x² : x ∈ L and x is even}
12                                                   = {4, 16, 64, 256}
13 (define Q (filter false? L))                  Q = {x ∈ L : False} = {}
```

Figure 2.25 The analogous list-building code in Scheme. (There is Scheme code analogous to the definition of `M`, but it requires an additional helper function or some more complicated syntax.)

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science.* This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

that processes data using two user-
specified functions: a "map" function that's applied to every element of the dataset, and a "reduce" function that collects together the outputs of the map function. Implementations of MapReduce allow these computations to occur in parallel, on a cluster of machines, vastly speeding processing time. (For more on MapReduce—a computing paradigm that has had a major impact on modern parallel and distributed computing—see [37].)

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Computer Science Connections

Clustering Data Sets (and Speech Processing)

Partitioning a set is a task that arises frequently in various applications, usually with a goal like *clustering* a large collection of data points. The goal is that elements placed into the same cluster should be "very similar," and elements in different clusters should be "not very similar." There is a huge array of applications of clustering (more on that below), but here's a simplified version of one example application, in the area of *speech processing*.

Software systems that interact with users as they speak in natural language—that is, as they talk in English, or Mandarin, or other real-world languages—have developed with rapidly increasing quality over the last decade. (You probably have a system that can perform this task in your pocket right now.) *Speech recognition*—taking an audio input, and identifying what English word is being spoken from the acoustic properties of the audio signal—turns out to be a very challenging problem. Figure 2.26 illustrates some of the reasons for the difficulty, showing a *spectrogram* generated by the Praat software tool. There are essentially no acoustic correlates to the divisions between words—the things that look like pauses in Figure 2.26 actually correspond to the moments that my mouth fully cut off the airflow during the sentence, like on "t" of *clustering*—which is one of the many reasons that speech recognition is so difficult.

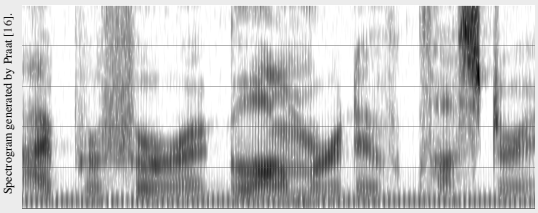


Figure 2.26 A *spectrogram* of me pronouncing the sentence "I prefer agglomerative clustering." In a spectrogram, the x-axis is time, and the y-axis is frequency; a darkly shaded frequency f at time t shows that the speech at time t had an intense component at frequency f.

But we can do something with more data. First, we can partition a *training set* of many speakers saying a collection of common words into subsets based on which word was spoken, and then use the average acoustic properties of the utterances to guess which word was spoken in novel audio. Figure 2.27 shows the frequencies of the two lowest *formants*—frequencies of very high intensity—in the utterances of a half-dozen college students saying the words *bat* and *beat*. The points are partitioned by the pronounced word. The *centroid* of each cluster (the center of mass of the points) can serve as a prototypical version of each word's acoustics. (These particular speakers followed the general trend for English vowels, but their formant frequencies don't quite match the statistics you'd expect from a larger sample of speakers.)

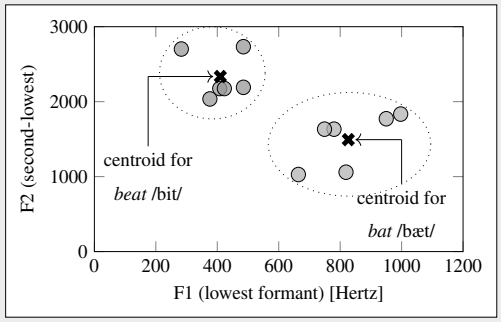


Figure 2.27 The frequencies of the first two formants in utterances by six speakers saying the words *beat* and *bat*.

Speech recognition is an important application, but there are many other reasons to perform clustering on a data set. We might try to cluster a set N of news articles into "topics" $C_1, C_2, \ldots, C_k$, where any two articles x, y that

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

are both in the same cluster C_i are similar (say, with respect to the words contained within them), but if $x \in C_i$ and $y \in C_{j \neq i}$ then x and y are not very similar. Or we might try to cluster the people in a social network into *communities*, so that a person in community c has a large fraction of her friends who are also in community c. Understanding these clusters—and understanding what properties of a data point "cause" it to be in one cluster rather than another—can help reveal the structure of a large data set, and can also be useful in building a system to react to new data. Or we might want to use clusters for *anomaly detection*: given a large data set—for example, of user behavior on a computer system, or the trajectory of a car on a highway—we might be able to identify those data points that do not seem to be part of a normal pattern. These data points may be the result of suspicious behavior that's worth further investigation (or that might trigger a warning to the driver of the car that they have strayed from a lane). For more about clustering, clustering algorithms, and applications of clustering, see a good data-mining book, like [76].

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

EXERCISES

2.86 Let $H = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{f}\}$ denote the set of hexadecimal digits. Is $6 \in H$?

2.87 For the same set H: is $\texttt{h} \in H$?

2.88 For the same set H: is $\texttt{a70e} \in H$?

2.89 For the same set H: What is $|H|$?

2.90 Let $S = \{0+0,\ 0+1,\ 1+0,\ 1+1,\ 0\cdot 0,\ 0\cdot 1,\ 1\cdot 0,\ 1\cdot 1\}$ be the set of results of adding any two bits together or multiplying any two bits together. Which of 0, 1, 2, and 3 are elements of S?

2.91 For the same set S: what is $|S|$?

2.92 Let $T = \{n \in \mathbb{Z} : 0 \leq n \leq 20 \text{ and } n \bmod 2 = n \bmod 3\}$. Let $H = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{f}\}$. Identify at least one element of H that is not an element of T.

2.93 For the same sets H and T: identify at least one element of T that is not an element of H.

2.94 For the same set T, and for S as defined in Exercise 2.90: identify at least one element of T that is not an element of S.

2.95 For the same set T, and for S as defined in Exercise 2.90: identify at least one element of S that is not an element of T.

2.96 For T as defined in Exercise 2.92: what is $|T|$?

2.97 Rewrite the following set by exhaustively listing its elements: $\{n \in \mathbb{Z} : 0 \leq n \leq 20 \text{ and } n \bmod 5 = n \bmod 7\}$.

2.98 Do the same for $\{n \in \mathbb{Z} : 10 \leq n \leq 30 \text{ and } n \bmod 5 = n \bmod 7\}$.

Let $A = \{1, 3, 4, 5, 7, 8, 9\}$ and let $B = \{0, 4, 5, 9\}$. Define $C = \{0, 3, 6, 9\}$. Where relevant, assume that the universe is the set $U = \{0, 1, 2, \ldots, 9\}$. What are the following sets?

2.99 $A \cap B$

2.100 $A \cup B$

2.101 $A - B$

2.102 $B - A$

2.103 $\sim B$

2.104 $A \cup \sim C$

2.105 $\sim C - \sim B$

2.106 $C - \sim C$

2.107 $\sim(C - \sim A)$

2.108 In general, $A - B$ and $B - A$ do *not* denote the same set. (See Figure 2.19d.) But your friends Evan and Yasmin wander by and tell you the following. Let E denote the set of CS homework questions that Evan has not yet solved. Let Y denote the set of CS homework questions that Yasmin has not yet solved. Evan and Yasmin claim that $E - Y = Y - E$. Is this possible? If so, under what circumstances? If not, why not? Justify your answer.

Let D and E be arbitrary sets. For each set given below, indicate which of the following statements is true:

- *the given set* must *be a subset of D (for every choice of D and E);*
- *the given set* may *be a subset of D (for certain choices of D and E); or*
- *the given set* cannot *be a subset of D (for any choice of D and E).*

If you answer "must" or "cannot," justify your answer (1–2 sentences). If you answer "may," identify an example D_1, E_1 for which the given set is a subset of D_1, and an example D_2, E_2 for which the given set is not a subset of D_2.

2.109 $D \cup E$

2.110 $D \cap E$

2.111 $D - E$

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.112 $E - D$

2.113 $\sim D$

Let $F = \{1, 2, 4, 8\}$, $G = \{1, 3, 9\}$, and $H = \{0, 5, 6, 7\}$. Where relevant, let $U = \{0, 1, 2, \ldots, 9\}$ be the universe.

2.114 Are F and G disjoint?

2.115 Are G and $\sim F$ disjoint?

2.116 Are $F \cap G$ and H disjoint?

2.117 Are H and $\sim H$ disjoint?

2.118 Let S and T be two sets, with $n = |S|$ and $m = |T|$. What is the smallest cardinality that $S \cup T$ can have. Give an example of the minimum-sized sets. (You should give a *family* of examples—that is, describe a smallest-possible set for *any* values of n and m.)

2.119 Repeat for $S \cap T$. (That is, what's the smallest possible value of $|S \cap T|$ in terms of n and m? Give a family of examples.)

2.120 Repeat for $S - T$.

2.121 What's the *largest* possible value of $S \cup T$ in terms of n and m? Give a family of examples.

2.122 Repeat for the largest possible value of $S \cap T$.

2.123 Repeat for the largest possible value of $S - T$.

In a variety of CS applications, it's useful to be able to compute the similarity *of two sets A and B. (More about one of these applications, collaborative filtering, below.) There are a number of different ideas of how to measure set similarity, all based on the intuition that the larger $|A \cap B|$ is, the more similar the sets A and B are. Here are two basic measures of set similarity that are sometimes used:*

- the cardinality measure: *the similarity of A and B is $|A \cap B|$.*
- the Jaccard coefficient: *the similarity of A and B is $\frac{|A \cap B|}{|A \cup B|}$. (The Jaccard coefficient is named after the Swiss botanist Paul Jaccard (1868–1944), who was interested in how similar or different the distributions of various plants were in different regions [63].)*

2.124 Let $A = \{\text{chocolate, hazelnut, cheese}\}$; $B = \{\text{chocolate, cheese, cardamom, cherries}\}$; and $C = \{\text{chocolate}\}$. Compute the similarities of each pair of these sets using the cardinality measure.

2.125 Repeat the previous exercise for the Jaccard coefficient.

2.126 Suppose we have a collection of sets $A_1, A_2, \ldots, A_n$. Consider the following claim:

Claim: Suppose that the set A_v is the most similar set to the set A_u in this collection (aside from A_u itself). Then A_u is necessarily the set that is most similar to A_v (aside from A_v itself).

Decide whether you think this claim is true for the cardinality measure of set similarity, and justify your answer. (That is, argue why it must be true, or give an example showing that it's false.)

2.127 Repeat the previous exercise for the Jaccard coefficient.

Taking it further: A *collaborative filtering system*, or *recommender system*, seeks to suggest new products to a user u on the basis of the similarity of u's past behavior to the past behavior of other users in the system. Collaborative filtering systems are mainstays of many popular commercial online sites (like Amazon or Netflix, for example). One common approach to collaborative filtering is the following. Let U denote the set of users of the system, and for each user $u \in U$, define the set S_u of products that u has purchased. To make a product recommendation to a user $u \in U$: first, identify the user $v \in U - \{u\}$ such that S_v is the set "most similar" to S_u; then, second, recommend the products in $S_v - S_u$ to user u (if any exist). This approach is called *nearest-neighbor collaborative filtering,* because the v found in step (i) is the other person closest to u. The measure of set similarity used in step (i) is all that's left to decide, and either cardinality or the Jaccard coefficient are reasonable choices. The idea behind the Jaccard coefficient is that the *fraction* of agreement matters more than the *total amount* of agreement: a {Cat's Cradle, Catch 22} purchaser is more similar to a {Slaughterhouse Five, Cat's Cradle} purchaser than someone who bought *every* book Amazon sells.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

For each of the following claims, decide whether you think the statement is true for all sets of integers A, B, C. If it's true for every A, B, C, then explain why. (A Venn diagram may be helpful.) If it's not true for every A, B, C, then provide an example for which it does not hold.

2.128 $A \cap B = \sim(\sim A \cup \sim B)$

2.129 $A \cup B = \sim(\sim A \cap \sim B)$

2.130 $(A - B) \cup (B - C) = (A \cup B) - C$

2.131 $(B - A) \cap (C - A) = (B \cap C) - A$

2.132 List all of the different ways to partition the set $\{1, 2, 3\}$.

Consider the table of distances shown in Figure 2.28 for a set $P = \{Alice, \ldots, Frank\}$ of people. Suppose we partition P into subsets $S_1, \ldots, S_k$. Define the intracluster distance *as the largest distance between two people who are in the same cluster, and define the* intercluster distance *as the smallest distance between two people who are in different clusters:*

$$\textit{intracluster distance} = \max_i \left[\max_{x,y \in S_i} \textit{distance}(x, y) \right] \quad \textit{and} \quad \textit{intercluster distance} = \min_{i,j \neq i} \left[\min_{x \in S_i, y \in S_j} \textit{distance}(x, y) \right].$$

2.133 Partition P into 3 or fewer subsets so that the intracluster distance is ≤ 2.0.

2.134 Partition P into subsets $S_1, \ldots, S_k$ so the intracluster distance is as small as possible. (You choose k.)

2.135 Partition P into subsets $S_1, \ldots, S_k$ so the intercluster distance is as large as possible. (Again, you choose k.)

2.136 Define $S = \{1, 2, \ldots, 100\}$. Let $W = \{x \in S : x \bmod 2 = 0\}$, $H = \{x \in S : x \bmod 3 = 0\}$, and $O = S - H - W$. Is $\{W, H, O\}$ a partition of S?

2.137 What is the power set of $\{1, a\}$?

2.138 What is the power set of $\{1\}$?

2.139 What is the power set of $\{\}$?

2.140 What is the power set of $\mathscr{P}(1)$?

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

	Alice	*Bob*	*Charlie*	*Desdemona*	*Eve*	*Frank*
Alice	0.0	1.7	1.2	0.8	7.2	2.9
Bob	1.7	0.0	4.3	1.1	4.3	3.4
Charlie	1.2	4.3	0.0	7.8	5.2	1.3
Desdemona	0.8	1.1	7.8	0.0	2.1	1.9
Eve	7.2	4.3	5.2	2.1	0.0	1.9
Frank	2.9	3.4	1.3	1.9	1.9	0.0

Figure 2.28 Some distances between people.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.4 Sequences, Vectors, and Matrices: Ordered Collections

> For nothing matters except life; and, of course, order.
>
> Virginia Woolf (1882–1941)
> "Montaigne," *The Common Reader* (1925)

In Section 2.3, we introduced sets—collections of objects in which the order of those objects doesn't matter. In many circumstances, though, order *does* matter: if a Java method takes two parameters, then swapping the order of those parameters will usually change what the method does; if there's an interesting site at longitude -1.3310 and latitude 52.5990, then showing up at longitude 52.5990 and latitude -1.3310 won't do. In this section, we turn to *ordered* collections of objects, called *sequences*. A summary of the notation related to sequences is given in Figure 2.29.

Definition 2.34: Sequence, list, and tuple.
A *sequence*—also known as a *list* or *tuple*—is an ordered collection of objects, typically called *components* or *entries*. When the number of objects in the collection is 2, 3, 4, or n, the sequence is called an *(ordered) pair*, *triple*, *quadruple*, or, *n-tuple*, respectively.

We'll write a sequence inside of angle brackets $\langle$ and $\rangle$, as in $\langle$Northfield, Minnesota, USA$\rangle$ or $\langle 0, 1\rangle$. (Some people use parentheses instead of angle brackets, as in $(128, 128, 0)$ instead of $\langle 128, 128, 0\rangle$.)

For two sets A and B, we frequently will refer to the set of ordered pairs whose two elements, in order, come from A and B:

Definition 2.35: Cartesian product.
The *Cartesian product* of two sets A and B is the set $A \times B = \{\langle a, b\rangle : a \in A \text{ and } b \in B\}$ containing all ordered pairs where the first component comes from A and the second component comes from B.

For example, $\{0, 1\} \times \{2, 3\}$ is the set $\{\langle 0, 2\rangle, \langle 0, 3\rangle, \langle 1, 2\rangle, \langle 1, 3\rangle\}$.

> The Cartesian product is named after René Descartes, the 17th-century French philosopher/mathematician. (The English adjectival form uses only the *cartes* part of his last name Des*cartes*.) Descartes is also the namesake of the Cartesian plane, which we'll see soon, and he was a major contributor in mathematics, particularly geometry. But Descartes is probably most famous as a philosopher, for the *cogito ergo sum* ("I think therefore I am") argument, in which Descartes—after adopting a highly skeptical view about all claims, even apparently obviously true ones (which is not a bad attitude to adopt when thinking about the kind of material in this book, for what it's worth!)—attempts to argue that he himself must exist.

We can also view any particular cell in a 2-dimensional grid—like a cell in a spreadsheet, or a square on a chess board—as a sequence:

Example 2.37: Chess positions.
A chess board is an 8-by-8 grid. Chess players use what's called "Algebraic notation" to refer to the columns (which they call *files*) using the letters a through h, and they refer to the rows (which they call

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

sequence/ordered tuple	$\langle a_1, a_2, \ldots, a_n \rangle$
Cartesian product	$A \times B = \{\langle a, b \rangle : a \in A \text{ and } b \in B\}$
the set of all n-element sequences of S	$S^n = S \times S \times \cdots \times S$ (n times)
vector	$x \in \mathbb{R}^n$
vector length, for $x \in \mathbb{R}^n$	$\|x\| = \sqrt{\sum_{i=1}^n x_i^2}$
vector addition, for vectors $x, y \in \mathbb{R}^n$	$x + y = \langle x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n \rangle$
scalar product, for $a \in \mathbb{R}$ and $x \in \mathbb{R}^n$	$ax = \langle a \cdot x_1, a \cdot x_2, \ldots, a \cdot x_n \rangle$
dot product, for vectors $x, y \in \mathbb{R}^n$	$x \bullet y = \sum_{i=1}^n x_i \cdot y_i$
matrix	$M \in \mathbb{R}^{n \times m}$
identity matrix	a matrix $I \in \mathbb{R}^{n \times n}$ where $I = \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{bmatrix}$
scalar multiplication, for $a \in \mathbb{R}$ and $M \in \mathbb{R}^{n \times m}$	a matrix $N \in \mathbb{R}^{n \times m}$ where $N_{i,j} = a \cdot M_{i,j}$
matrix addition, for $M, M' \in \mathbb{R}^{n \times m}$	a matrix $N \in \mathbb{R}^{n \times m}$ where $N_{i,j} = M_{i,j} + M'_{i,j}$
matrix multiplication, for $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times p}$	a matrix $M \in \mathbb{R}^{n \times p}$ where $M_{i,j} = \sum_{k=1}^m A_{i,k} B_{k,j}$
matrix inverse, for $M \in \mathbb{R}^{n \times n}$	a matrix $M^{-1} \in \mathbb{R}^{n \times n}$ where $MM^{-1} = I$ (if any such M^{-1} exists)

Figure 2.29 A summary of notation for sequences, vectors, and matrices.

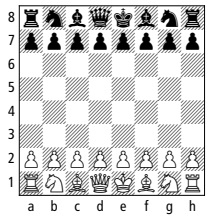


Figure 2.30 The squares of a chess board.

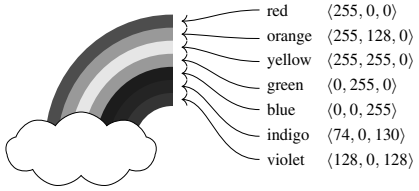


Figure 2.31 The seven traditional colors of the rainbow.

ranks) using the numbers 1 through 8. (See Figure 2.30.) Thus the square containing the white queen ♕ is $\langle \mathsf{d}, 1 \rangle$; the full set of squares of the chess board is $\{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathsf{f}, \mathsf{g}, \mathsf{h}\} \times \{1, 2, 3, 4, 5, 6, 7, 8\}$; and the squares containing knights—the ♘ pieces (both white and black)—are $\{\langle \mathsf{b}, 1 \rangle, \langle \mathsf{g}, 1 \rangle, \langle \mathsf{b}, 8 \rangle, \langle \mathsf{g}, 8 \rangle\}$. The set of squares with knights could also be written as $\{\mathsf{b}, \mathsf{g}\} \times \{1, 8\}$.

Here's another example, about color representation on computers:

Example 2.38: RGB color values.
The *RGB color space* represents colors as ordered triples, where each component is an element of $\{0, 1, \ldots, 255\}$. RGB stands for *red–green–blue*; the three components of a color, respectively, represent how red, how green, and how blue the color is. Formally, a color is an element of $\{0, 1, \ldots, 255\} \times \{0, 1, \ldots, 255\} \times \{0, 1, \ldots, 255\}$.

The order of these components matters; for example, the color $\langle 0, 0, 255 \rangle$ is pure blue, while the color $\langle 255, 0, 0 \rangle$ is pure red. An example of the seven traditional colors of the rainbow is shown in Figure 2.31.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Taking it further: An annoying pedantic point: we are being sloppy with notation in Example 2.38; we only defined the Cartesian product for two sets, so when we write $S \times S \times S$ we "must" mean either $S \times (S \times S)$ or $(S \times S) \times S$. We're going to ignore this issue, and simply write statements like $\langle 0, 1, 1 \rangle \in \{0,1\} \times \{0,1\} \times \{0,1\}$—even though we *ought* to instead be writing statements like $\langle 0, \langle 1, 1 \rangle \rangle \in \{0,1\} \times (\{0,1\} \times \{0,1\})$. (A similar shorthand shows up in programming languages like Scheme, where pairing—"cons"ing—a single element `3` with a list `(2 1)` yields the three-element list `(3 2 1)`, rather than the two-element pair `(3 . (2 1))`, where the second element is a two-element list.)

Beyond the "obvious" sequences like Examples 2.37 and 2.38, we've also already seen some definitions that don't seem to involve sequences, but implicitly *are* about ordered tuples of values. One example is the rational numbers (see Section 2.2.2):

Example 2.39: Rational numbers as sequences.
We can define the *rational numbers* (also known as *fractions*) as the set $\mathbb{Q} = \mathbb{Z} \times \mathbb{Z}^{>0}$. Under this view, a rational number is represented as a pair $\langle n, d \rangle \in \mathbb{Z} \times \mathbb{Z}^{>0}$, with a numerator n and a denominator d.

For example, the fractions $\frac{1}{2}$ and $\frac{202}{808}$ would be represented as $\langle 1, 2 \rangle$ and $\langle 202, 808 \rangle$, respectively. (To flesh out the details of this representation, we also have to consider reducing fractions to lowest terms, to establish the equivalence of fractions like $\langle 2, 4 \rangle$ and $\langle 1, 2 \rangle$. See Example 8.36.)

We will often consider sequences of elements that are all drawn from the same set, and there is special notation for such a sequence:

Definition 2.36: Sequences of elements from the same set.
For a set S and a positive integer n, we write S^n to denote $S^n = \underbrace{S \times S \times \ldots \times S}_{n \text{ times}}$.

Thus S^n denotes the set of all sequences of length n where each component of the sequence is an element the set S. For example, the RGB values from Example 2.38 are elements of $\{0, 1, \ldots, 255\}^3$, and $\{0,1\}^3$ denotes the set $\{\langle 0,0,0 \rangle, \langle 0,0,1 \rangle, \langle 0,1,0 \rangle, \langle 0,1,1 \rangle, \langle 1,0,0 \rangle, \langle 1,0,1 \rangle, \langle 1,1,0 \rangle, \langle 1,1,1 \rangle\}$. This notation also lets us write $\mathbb{R} \times \mathbb{R}$, called the *Cartesian plane,* as $\mathbb{R}^2$—the way you might have written it in a high school algebra class. (See Figure 2.32.)

In certain contexts, sequences of elements from the same set (as in Definition 2.36) are called *strings.* For a set Σ, called an *alphabet*, a *string over* Σ is an element of Σ^n for some nonnegative integer n. (In other

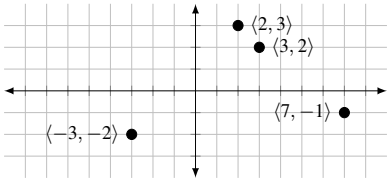


Figure 2.32 A few points in $\mathbb{R}^2$. The first component represents the x-axis (horizontal) position; the second component represents the y-axis (vertical) position.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science.* This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

words, a string is any element of $\bigcup_{n\in\mathbb{Z}^{\geq 0}} \Sigma^n$.) The *length* of a string $x \in \Sigma^n$ is n. For example, the set of 5-letter words in English is a subset of $\{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}\}^5$. We allow strings to have length zero: for any alphabet Σ, there is only one sequence of elements from Σ of length 0, called the *empty string*; it's denoted by ϵ, and for any alphabet Σ, we have $\Sigma^0 = \{\epsilon\}$. When writing strings, it is customary to omit the punctuation (angle brackets and commas), so we write $\mathtt{ABRACADABRA} \in \{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}\}^{11}$ and $11010011 \in \{0, 1\}^8$.

2.4.1 Vectors

As we've already seen, we can create sequences of many types of things: we can view sequences of letters as strings (like $\mathtt{ABRACADABRA} \in \{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}\}^{11}$), or sequences of three integers between 0 and 255 as colors (like $\langle 119, 136, 153\rangle \in \{0, 1, \ldots, 255\}^3$, officially called "light slate gray"). Perhaps the most pervasive type of sequence, though, is a sequence of real numbers, called a *vector*.

Definition 2.37: Vector.
A *vector* (or *n-vector*) x is a sequence $x \in \mathbb{R}^n$, for some positive integer n. For a vector $x \in \mathbb{R}^n$ and for any index $i \in \{1, 2, \ldots, n\}$, we write x_i to denote the ith component of x.

For example, $\langle 0, 1\rangle$, $\langle 1, 0\rangle$, and $\langle \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\rangle$ are all vectors in $\mathbb{R}^2$. For the vector $x = \langle 1/2, \sqrt{3}/2\rangle$, we have $x_1 = 1/2$ and $x_2 = \sqrt{3}/2$.

> **Taking it further:** Vectors are used in a tremendous variety of computational contexts: computer graphics (representing the line-of-sight from the viewer's eye to an object in a scene), machine learning (a *feature vector* describing which characteristics a particular object has, which can be used in trying to classify that object as satisfying a condition or failing to satisfy a condition), among many others. The discussion on p. 2-61 describes the *vector-space model* for representing a document d as a vector whose components correspond to the number of times each word appears in d. Vectors and matrices (the topics of the remainder of this section) are the main focus of a math course in linear algebra. In this section, we're only mentioning a few highlights of vectors and matrices; you can find much more in any good textbook on linear algebra.
>
> A warning for C or Java or Python (or …) programmers: notice that our vectors' components are indexed starting at one, not zero. For a vector $x \in \mathbb{R}^n$, the expression x_i is meaningless unless $i \in \{1, 2, \ldots, n\}$. The expression x_0 doesn't mean anything.

Vectors are sometimes contrasted with *scalars*, which are just numbers: that is, a scalar is an element of $\mathbb{R}$. Vectors are also sometimes written in square brackets, so we may see an n-vector x written as $x = [x_1, x_2, \ldots, x_n]$. We may encounter vectors in which the components are a restricted kind of number—for example, integers or bits. Elements of $\{0, 1\}^n$ are often called *bit vectors* or *bitstrings*.

Here's an example of using vectors to compute distances between points:

Example 2.40: Train stations in Manhattan.
Let's (very roughly!) represent a location in Manhattan as a vector—specifically, as a point $\langle x, y\rangle \in \mathbb{R}^2$ representing the intersection of xth Avenue and yth Street. Define the *walking distance* between points p and q in Manhattan as $|p_1 - q_1| + |p_2 - q_2|$: the number of east–west blocks between p and q *plus*

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

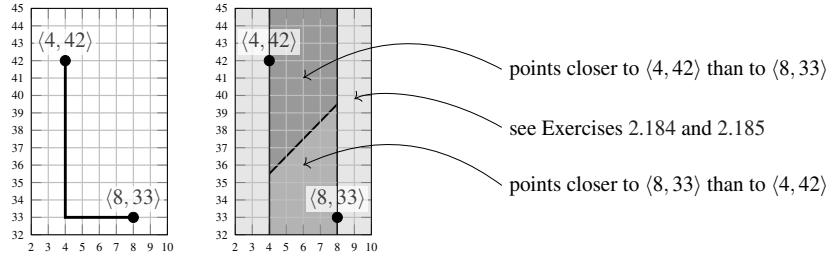


Figure 2.33 Illustrations of Manhattan train stations.

the number of north–south blocks between p and q. (Note that walking distance is different from the straight-line distance between the points.)

The two major train stations in Manhattan are Penn Station, located at $s = \langle 8, 33\rangle$, and Grand Central Station, at $g = \langle 4, 42\rangle$. First, determine the walking distance between Penn Station and Grand Central. Then describe the points that are closer (in walking distance) to Penn Station than to Grand Central.

Solution. The distance between $s = \langle 8, 33\rangle$ and $g = \langle 4, 42\rangle$ is

$$|s_1 - g_1| + |s_2 - g_2| = |8 - 4| + |33 - 42| = 4 + 9 = 13.$$

For the second question, let's compute some points that are equidistant to the two stations. (Those points are on the boundary of the region of points closer to g and the region of points closer to s.) For example, a point $\langle 4, y\rangle$ has distances $|42 - y|$ and $4 + |y - 33|$ to the stations; these distances are both equal to 6.5 when $y = 35.5$.

More generally, let's think about a point whose x-coordinate falls between 4 and 8. For any offset $0 \le \delta \le 4$, the distance between the point $\langle 4+\delta, y\rangle$ and the two stations are $\delta + |42 - y|$ and $4 - \delta + |y - 33|$. These two values are both equal to 6.5 when $y = 35.5 + \delta$. (For example, when $\delta = 4$, then $y = 39.5$.) Thus the points $\langle 4 + 0, 35.5 + 0\rangle = \langle 4, 35.5\rangle$ and $\langle 4 + 4, 35.5 + 4\rangle = \langle 8, 39.5\rangle$ are both equidistant to s and g, as are all points on the line segment between them. (See Figure 2.33.) The remaining cases of the analysis—figuring out which points with x-coordinate less than 4 or greater than 8 are closer to s or g—are left to Exercises 2.184 and 2.185.

Taking it further: The measure of walking distance between points described in Example 2.40 is used surprisingly commonly in computer science applications—and, appropriately enough, it's actually named after Manhattan. The *Manhattan distance* between two points $p, q \in \mathbb{R}^n$ is defined as $\sum_{i=1}^{n} |p_i - q_i|$. (We're summing the number of "blocks" of difference in each of the n dimensions; we take the absolute value of each difference because we care about the difference in each dimension rather than which point has the higher value in that component.)

Here's one more useful definition about vectors:

Definition 2.38: Vector length.
The *length* of a vector $x \in \mathbb{R}^n$ is defined as $\|x\| = \sqrt{\sum_{i=1}^{n} (x_i)^2}$.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

For example, $\|\langle 2, 8\rangle\| = \sqrt{2^2 + 8^2} = \sqrt{4 + 64} = \sqrt{68} \approx 8.246$. If we draw a vector $x \in \mathbb{R}^2$ in the Cartesian plane, then $\|x\|$ denotes the length of the line from $\langle 0, 0\rangle$ to x. (See Figure 2.34.) A vector $x \in \mathbb{R}^n$ is called a *unit vector* if $\|x\| = 1$.

Vector arithmetic

We will now define basic arithmetic for vectors: *vector addition*, which is performed component-wise (adding the corresponding elements of the two vectors), and two forms of multiplication—one for multiplying a vector by a scalar (also component-wise) and one for multiplying two vectors together.

Definition 2.39: Vector addition.
The *sum* of two vectors $x, y \in \mathbb{R}^n$, written $x+y$, is a vector $z \in \mathbb{R}^n$, where for every index $i \in \{1, 2, \ldots, n\}$ we have $z_i = x_i + y_i$. (Note that the sum of two vectors with different sizes is meaningless.)

For example, $\langle 1.1, 2.2, 3.3\rangle + \langle 2, 0, 2\rangle = \langle 3.1, 2.2, 5.3\rangle$.

The first type of multiplication for vectors is *scalar multiplication*, when we multiply a vector by a real number. As with vector addition, scalar multiplication acts on each component independently, by rescaling each component by the same factor:

Definition 2.40: Scalar product.
Given a vector $x \in \mathbb{R}^n$ and a real number $a \in \mathbb{R}$, the *scalar product* ax is a vector $z \in \mathbb{R}^n$, where $z_i = ax_i$ for every index $i \in \{1, 2, \ldots, n\}$.

For example, $3 \cdot \langle 1, 2, 3\rangle = \langle 3, 6, 9\rangle$ and $-1.5 \cdot \langle 1, -1\rangle = \langle -1.5, 1.5\rangle$ and $0 \cdot \langle 1, 2, 3, 5, 8\rangle = \langle 0, 0, 0, 0, 0\rangle$.

The second type of vector multiplication, the *dot product,* takes two vectors as input and multiplies them together to produce a single scalar as output:

Definition 2.41: Dot product.
Given two vectors $x, y \in \mathbb{R}^n$, the *dot product* of x and y, denoted $x \bullet y$, is given by summing the products of the corresponding components:

$$x \bullet y = \sum_{i=1}^{n} x_i \cdot y_i.$$

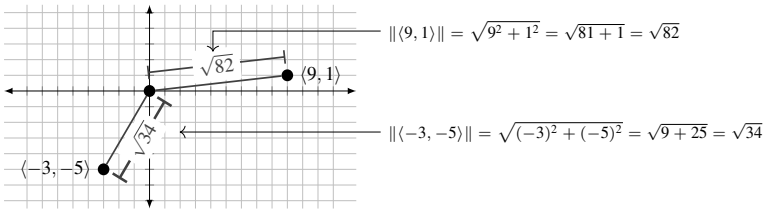


Figure 2.34 Two vector lengths: $\|\langle 9, 1\rangle\|$ and $\|\langle -3, -5\rangle\|$.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

For example, $\langle 1,2,3\rangle \bullet \langle 4,5,6\rangle = 1\cdot 4 + 2\cdot 5 + 3\cdot 6 = 4 + 10 + 18 = 32$.

Intuitively, the dot product of two vectors measures the extent to which they point in the "same direction." (Again, the dimensions of the vectors in a dot product have to match up: if $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$ are vectors where $n \neq m$, then $x \bullet y$ is meaningless.)

Figure 2.35 shows a few unit vectors, and each of their dot products with one of the vectors. Here are two other examples, which use dot products for some useful applications:

Example 2.41: Common classes.
Let $C = \langle \text{CS1}, \text{CS2}, \ldots, \text{CS8}\rangle$ denote the list of all courses offered by a (somewhat narrowly focused) university. For a particular student, let the bit vector u represent the courses taken by that student, so that $u_i = 1$ if the student has taken course c_i (and $u_i = 0$ otherwise). For example, a student who's taken only CS1 and CS8 would be represented by $x = \langle 1,0,0,0,0,0,0,1\rangle$, and a student who's taken everything except CS3 would be represented by $y = \langle 1,1,0,1,1,1,1,1\rangle$.

The dot product of two student vectors represents the number of common courses that they've taken. For example, the number of common classes taken by x and y is

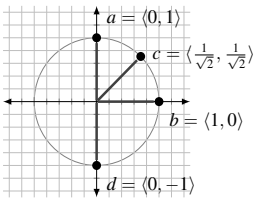
$$\begin{aligned} x \bullet y = \sum_{i=1}^{8} x_i y_i &= 1\cdot 1 + 0\cdot 1 + 0\cdot 0 + 0\cdot 1 + 0\cdot 1 + 0\cdot 1 + 0\cdot 1 + 1\cdot 1 \\ &= 1 + 0 + 0 + 0 + 0 + 0 + 0 + 1, \end{aligned}$$

which is 2. Specifically, the two common courses taken by x and y are CS1 and CS8.

Example 2.42: GPAs.
Let $g \in \mathbb{R}^n$ be an n-vector where g_i denotes the grade (measured on the grade point scale) that you got in the ith class that you've taken in your college career. Let $c \in \mathbb{R}^n$ be an n-vector where c_i denotes the number of credits for the ith class you took. Then your grade point average (GPA) is given by $\frac{g \bullet c}{\sum_{i=1}^{n} c_i}$.

For example, suppose your school gives grade points on the scale 4.0 = A, 3.7 = A-, 3.3 = B+, 3.0 = B, etc. Suppose you took CS 111 (6 credits), CS 201 (6 credits), and Mbira Lessons (4 credits), and got grades of B+, A-, and B, respectively. Then $g = \langle 3.3, 3.7, 3.0\rangle$ and $c = \langle 6,6,4\rangle$, and

$$\text{your GPA} = \frac{g \bullet c}{\sum_{i=1}^{3} c_i} = \frac{3.3\cdot 6 + 3.7\cdot 6 + 3.0\cdot 4}{6+6+4} = \frac{19.8 + 22.2 + 12.0}{16} = \frac{54}{16} = 3.375.$$



$$\begin{aligned} c \bullet a &= c_1 \cdot a_1 + c_2 \cdot a_2 = \tfrac{1}{\sqrt{2}}\cdot 0 + \tfrac{1}{\sqrt{2}}\cdot 1 &&= \tfrac{1}{\sqrt{2}} \\ c \bullet b &= c_1 \cdot b_1 + c_2 \cdot b_2 = \tfrac{1}{\sqrt{2}}\cdot 1 + \tfrac{1}{\sqrt{2}}\cdot 0 &&= \tfrac{1}{\sqrt{2}} \\ c \bullet c &= c_1 \cdot c_1 + c_2 \cdot c_2 = \tfrac{1}{\sqrt{2}}\cdot\tfrac{1}{\sqrt{2}} + \tfrac{1}{\sqrt{2}}\cdot\tfrac{1}{\sqrt{2}} &&= \tfrac{1}{2} + \tfrac{1}{2} = 1 \\ c \bullet d &= c_1 \cdot d_1 + c_2 \cdot d_2 = \tfrac{1}{\sqrt{2}}\cdot 0 + \tfrac{1}{\sqrt{2}}\cdot -1 &&= -\tfrac{1}{\sqrt{2}} \end{aligned}$$

Figure 2.35 Four unit vectors, and each of their dot products with c.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.4.2 Matrices

If a vector is analogous to an array of numbers, then a *matrix* is analogous to a two-dimensional array of numbers:

Definition 2.42: Matrix.
An *n-by-m matrix M* is a two-dimensional table of real numbers containing *n rows* and *m columns*. The $\langle i,j\rangle$th entry of the matrix appears in the ith row and jth column, and we denote that entry by $M_{i,j}$. Such a matrix M is an element of $\mathbb{R}^{n\times m}$, and we refer to M as having *size* or *dimension* n-by-m.

(The plural of matrix is *matrices,* which rhymes with the word "cheese.") Thus a matrix looks like

$$M = \begin{bmatrix} M_{1,1} & M_{1,2} & \dots & M_{1,m} \\ M_{2,1} & M_{2,2} & \dots & M_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ M_{n,1} & M_{n,2} & \dots & M_{n,m} \end{bmatrix}.$$

It's possible to think of a two-dimensional array in a programming language as a one-dimensional array *of one-dimensional arrays*; similarly, if you'd like, you can think of an n-by-m matrix as a sequence of n vectors, all of which are elements of $\mathbb{R}^m$. This view of an n-by-m matrix is as an element of $(\mathbb{R}^m)^n$.

Three small matrices are shown in Figure 2.36a. (The $\langle 2,1\rangle$st entry is circled in each.) In Figure 2.36a, A is a 2-by-3 matrix, B is a 3-by-2 matrix, and I is a 3-by-3 matrix.

$$A = \begin{bmatrix} 3 & 1 & 4 \\ \textcircled{9} & 7 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 5 & 3 \\ \textcircled{4} & 8 \\ 6 & 9 \end{bmatrix} \quad I = \begin{bmatrix} 1 & 0 & 0 \\ \textcircled{0} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(a) Three matrices, with the $\langle 2,1\rangle$st entry circled in each.

$$M = \begin{bmatrix} \textcircled{1} & 2 & 3 \\ 4 & \textcircled{5} & 6 \\ 7 & 8 & \textcircled{9} \end{bmatrix}$$

(b) A matrix with the entries of the main diagonal circled.

Figure 2.36 A few matrices.

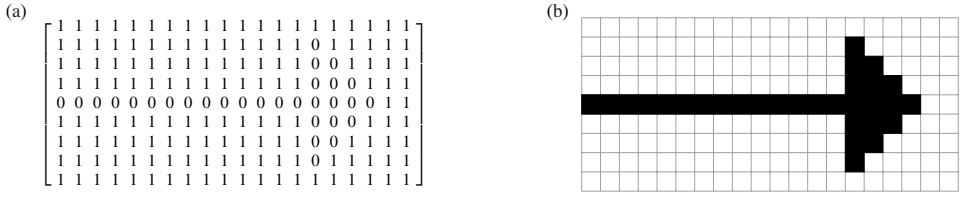


Figure 2.37 (a) A matrix representing a black-and-white bitmapped image, and (b) the image.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science.* This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Example 2.43: Bitmaps.
One handy application of matrices is as an easy way to represent images. A black-and-white image can be represented as a matrix with all entries in $\{0, 1\}$: each 1 entry represents white in the corresponding pixel; each 0 represents black. See Figure 2.37a for an example.

Taking it further: The picture in Figure 2.37 is a simple black-and-white image, but we can use the same basic structure for grayscale or color images. Instead of an integer in $\{0, 1\}$ as each entry in the matrix, a grayscale pixel could be represented using a real number in $[0, 1]$—or, more practically, a number in $\{\frac{0}{255}, \frac{1}{255}, \ldots, \frac{255}{255}\}$. For color images, each entry would be an RGB triple (see Example 2.38). These matrix-based representations of an image are often called *bitmaps*. Bitmaps are highly inefficient ways of storing images; most computer graphics file formats use much cleverer (and more space-efficient) representations.

Here are few other examples of the pervasive applications of matrices in computer science. A *term–document matrix* can be used to represent a collection of documents: the entry $M_{d,k}$ of the matrix M stores the number of times that keyword k appears in document d. An *adjacency matrix* (see Chapter 11) can represent the page-to-page hyperlinks of the web in a matrix M, where $M_{i,j} = 1$ if web page i has a hyperlink to web page j (and $M_{i,j} = 0$ otherwise). A *rotation matrix* (see p. 2-63) can be used in computer graphics to re-render a scene from a different perspective.

A matrix $M \in \mathbb{R}^{m \times n}$ is called *square* if $m = n$. For a square matrix $M \in \mathbb{R}^{n \times n}$, we may say that the size of M is n (rather than saying that its size is n-by-n). A square matrix M is called *symmetric* if, for all indices $i, j \in \{1, 2, \ldots, n\}$, we have $M_{i,j} = M_{j,i}$. The *main diagonal* of a square matrix $M \in \mathbb{R}^{n \times n}$ is the sequence consisting of the entries $M_{i,i}$ for $i = 1, 2, \ldots, n$. (An example is shown in Figure 2.36b.) One special square matrix that will arise frequently is the *identity matrix*, which has ones on the main diagonal and zeros everywhere else (see Figure 2.38):

Definition 2.43: Identity matrix.

The *n-by-n identity matrix* is the matrix $I \in \mathbb{R}^{n \times n}$ whose entries satisfy $I_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j. \end{cases}$

Note that there is a different n-by-n identity matrix for every $n \geq 1$; for example, the 1-by-1, 2-by-2, …, 5-by-5 identity matrices are shown in Figure 2.38.

As with vectors, we will need to define the basic arithmetic operations of addition and multiplication for matrices. Just as with vectors, adding two n-by-m matrices or multiplying a matrix by a scalar is done component by component:

$$\begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 2.38 The identity matrix I generically, and for size up to 5.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Definition 2.44: Matrix addition and scalar multiplication.
Given a matrix $M \in \mathbb{R}^{n\times m}$ and a real number $a \in \mathbb{R}$, the product aM is a matrix $N \in \mathbb{R}^{n\times m}$ where $N_{i,j} = aM_{i,j}$ for all indices $i \in \{1, 2, \ldots, n\}$ and $j \in \{1, 2, \ldots, m\}$.

Given two matrices $M, M' \in \mathbb{R}^{n\times m}$, the sum $M + M'$ is a matrix $N \in \mathbb{R}^{n\times m}$ where $N_{i,j} = M_{i,j} + M'_{i,j}$ for all indices $i \in \{1, 2, \ldots, n\}$ and $j \in \{1, 2, \ldots, m\}$.

Again, just as with vectors, adding two matrices that are not the same size is meaningless.

Example 2.44: Matrix arithmetic for some small matrices.
Consider the following matrices:

$$A = \begin{bmatrix} 0 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 6 \\ 0 & 0 & 4 \end{bmatrix} \qquad I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Then we have:

$$A + B = \begin{bmatrix} 1 & 4 & 5 \\ 2 & 0 & 8 \\ 2 & 2 & 4 \end{bmatrix} \qquad 4B = \begin{bmatrix} 4 & 8 & 12 \\ 0 & 0 & 24 \\ 0 & 0 & 16 \end{bmatrix} \qquad A + 3I = \begin{bmatrix} 3 & 2 & 2 \\ 2 & 3 & 2 \\ 2 & 2 & 3 \end{bmatrix} \qquad A - 3I = \begin{bmatrix} -3 & 2 & 2 \\ 2 & -3 & 2 \\ 2 & 2 & -3 \end{bmatrix}.$$

Matrix multiplication

Multiplying matrices is a bit more complicated than the other vector/matrix operations that we've seen so far. The product of two matrices is a *matrix*, rather than a single number: the entry in the ith row and jth column of AB is derived from the ith row of A and the j column of B. More precisely:

Definition 2.45: Matrix multiplication.
The product AB of two matrices $A \in \mathbb{R}^{n\times m}$ and $B \in \mathbb{R}^{m\times p}$ is an n-by-p matrix $M \in \mathbb{R}^{n\times p}$ whose entries are, for any $i \in \{1, 2, \ldots n\}$ and $j \in \{1, 2, \ldots, p\}$,

$$M_{i,j} = \sum_{k=1}^{m} A_{i,k} B_{k,j}.$$

As usual, if the dimensions of the matrices A and B don't match—if the number of columns in A is different from the number of rows in B—then AB is undefined.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

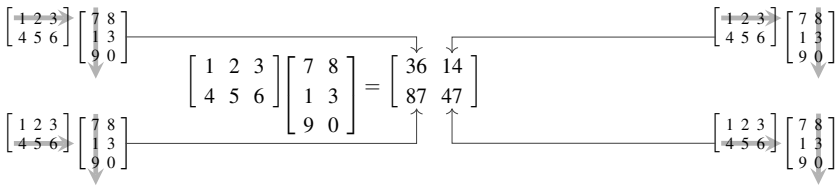


Figure 2.39 Multiplying two matrices.

Example 2.45: Multiplying some small matrices.
Let's compute the product of a sample 2-by-3 matrix and a 3-by-2 matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \cdot \begin{bmatrix} 7 & 8 \\ 1 & 3 \\ 9 & 0 \end{bmatrix}$$

Note that, by definition, the result will be a 2-by-2 matrix. Its entries are:

$$\begin{bmatrix} 1\cdot 7 + 2\cdot 1 + 3\cdot 9 & 1\cdot 8 + 2\cdot 3 + 3\cdot 0 \\ 4\cdot 7 + 5\cdot 1 + 6\cdot 9 & 4\cdot 8 + 5\cdot 3 + 6\cdot 0 \end{bmatrix} = \begin{bmatrix} 7+2+27 & 8+6+0 \\ 28+5+54 & 32+15+0 \end{bmatrix} = \begin{bmatrix} 36 & 14 \\ 87 & 47 \end{bmatrix}.$$

For example, the 14 in ⟨row #1, column #2⟩ of the result was calculated by successively multiplying the first matrix's first row $\langle 1, 2, 3\rangle$ by the second matrix's second column $\langle 8, 3, 0\rangle$. A visual representation of this multiplication is shown in Figure 2.39.

More compactly, we could write matrix multiplication using the dot product from Definition 2.41: for two matrices $A \in \mathbb{R}^{n\times m}$ and $B \in \mathbb{R}^{m\times p}$, the $\langle i, j\rangle$th entry of AB is the value of $A_{i,(1\ldots m)} \bullet B_{(1\ldots m),j}$.

> *Problem-solving tip:* To help keep matrix multiplication straight, it may be helpful to compute the $\langle i, j\rangle$th entry of AB by simultaneously tracing the ith row of A with the index finger of your left hand, and the jth column of B with the index finger of your right hand. Multiply the two numbers that you're pointing at, and add the result to a running tally; when you've traced the whole row/column, the running tally is $(AB)_{i,j}$.

Be careful: matrix multiplication is not commutative—that is, for matrices A and B, the values AB and BA are generally different! (This asymmetry is unlike numerical multiplication: for $x, y \in \mathbb{R}$, it is always the case that $xy = yx$.) In fact, because the number of columns of A must match the number of rows of B for AB to even be meaningful, it's possible for BA to be meaningless or a different size from AB.

Example 2.46: Multiplying the other way around.
If we multiply the matrices from Example 2.45 in the other order, we get

$$\begin{bmatrix} 7 & 8 \\ 1 & 3 \\ 9 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 39 & 54 & 69 \\ 13 & 17 & 21 \\ 9 & 18 & 27 \end{bmatrix}$$

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

This matrix differs from the result in Example 2.45—it's not even the same size!

You'll show in the exercises that, for any n-by-m matrix A, the result of multiplying A by the identity matrix I yields A itself: that is, $AI = A$. You'll also explore the *inverse* of a matrix A: that is, the matrix A^{-1} such that $AA^{-1} = I$ (if any such A^{-1} exists).

Here's another way to use matrices to combine different types of information:

Example 2.47: Programming language knowledge.
Let A be an n-by-m matrix where $A_{i,j} = 1$ if student i has taken class j (and $A_{i,j} = 0$ otherwise). Let B be an m-by-p matrix where $B_{j,k} = 1$ if class j uses programming language k (and $B_{j,k} = 0$ otherwise). What does the matrix AB represent?

Solution. First, note that the resulting matrix AB has n rows and p columns; that is, its size is (number of students)-by-(number of languages). For a student i and a programming language k, we have that

$$\begin{aligned}(AB)_{i,k} &= \sum_{j=1}^{m} A_{i,j}B_{j,k} \\ &= \sum_{j=1}^{m} \left[\begin{cases} 1 & \text{if student } i \text{ took class } j \textit{ and } j \text{ uses language } k \\ 0 & \text{otherwise} \end{cases} \right]\end{aligned}$$

because $0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 0$, so the only terms of the sum that are 1 occur when both $A_{i,j}$ ("student i took class j?") and $B_{j,k}$ ("class j uses language k?") are true (that is, 1). Thus $(AB)_{i,k}$ denotes the number of classes that use language k that student i took.

Figure 2.40 shows a concrete example. (For example, the cell corresponding to Alice and C is computed by $\langle 0,1,1,1,1\rangle \bullet \langle 0,0,1,1,0\rangle$—the dot product of Alice's row of A with C's column of B—which has the value $0 \cdot 0 + 1 \cdot 0 + 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 0 = 2$. This entry reflects the fact that Alice has taken two classes that use C: (i) introduction to computer systems, and (ii) programming languages.)

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

$$A = \begin{array}{c} \begin{matrix} \text{intro} & \text{data structures} & \text{intro systems} & \text{prog langs} & \text{theory of comp} \end{matrix} \\ \left[\begin{array}{ccccc} 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{array}\right] \end{array} \begin{matrix} \\ \text{Alice} \\ \text{Bob} \\ \text{Charlie} \end{matrix}$$

$$B = \begin{array}{c} \begin{matrix} \text{Perl} & \text{Python} & \text{C} & \text{Java} & \text{Assembly} & \text{C++} & \text{Scheme} \end{matrix} \\ \left[\begin{array}{ccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array} \begin{matrix} \\ \text{intro} \\ \text{data structures} \\ \text{intro systems} \\ \text{prog langs} \\ \text{theory of comp} \end{matrix}$$

$$AB = \begin{array}{c} \begin{matrix} \text{Perl} & \text{Python} & \text{C} & \text{Java} & \text{Assembly} & \text{C++} & \text{Scheme} \end{matrix} \\ \left[\begin{array}{ccccccc} 0 & 2 & 2 & 2 & 2 & 1 & 1 \\ 0 & 3 & 1 & 2 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array} \begin{matrix} \\ \text{Alice} \\ \text{Bob} \\ \text{Charlie} \end{matrix}$$

Figure 2.40 Three students, five courses, and seven programming languages.

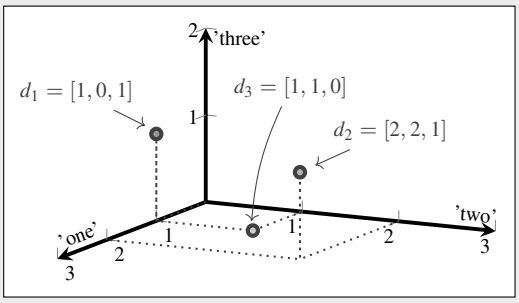
This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Computer Science Connections

The Vector Space Model

Here's a classic application of vectors, taken from *information retrieval*, the subfield of computer science devoted to searching for information relevant to a given query in large datasets. (For more on information retrieval, see [84].) We start with a large *corpus* of documents—for example, transcripts of all email messages that you've sent in your entire life. (The word *corpus* comes from the Latin for "body"; it simply means a body of texts.) Tasks might include *clustering* the corpus into subcollections ("which of my email messages are spam?"), or finding stored documents similar to a given query ("find the emails most relevant to 'good restaurants in Chicago' in my archives").

The *vector space model* is a standard approach to representing text documents for the purposes of information retrieval. We choose a list of n *terms* that might appear in a document. We then represent a document d as an n-vector x of integers, where x_i is the number of times that the ith term appears in the document d. See Figure 2.41 for an example. Because documents that are about similar topics tend to contain similar vocabulary, we can judge the similarity of documents d and d' based on "how similar" their corresponding vectors x and x' are. A first stab at measuring similarity between x and x' is to compute the dot product $x \bullet x'$; this approach counts the number of times any word in d appears in d'. (And if a word appears twice in d, then each appearance in d' counts twice for the dot product.) But this first approach has an issue in that it favors longer documents: a document that lists all the words in the dictionary would correspond to a vector $[1, 1, 1, 1, 1, \ldots]$—which would therefore have a large dot product with all documents in the corpus.



d_1	*Three is one of the loneliest numbers.*	$\longrightarrow [1, 0, 1]$
d_2	*A one and a two and a one, two, three.*	$\longrightarrow [2, 2, 1]$
d_3	*One, two, buckle my shoe.*	$\longrightarrow [1, 1, 0]$

Figure 2.41 An example from the vector-space model: three documents translated into vectors using the keywords *'one'*, *'two'*, and *'three'*, and a plot of them in $\mathbb{R}^3$.

To compensate for the fact that longer documents have more words, it's a good idea to *normalize* these vectors so that they have the same length, by using $x/\|x\|$ and $x'/\|x'\|$ to represent the documents, instead of x and x'. It turns out that the dot product of the normalized vectors computes the cosine of the *angle* between these representations of the documents. We can then measure the similarity of x and x' based on how small the angle between them is.

This second approach is better, but it still suffers from counting common occurrences of the word *the* and the word *normalize* as equally indicative of the similarity of documents. That's clearly wrong: some words carry more information than others. Information retrieval systems apply different weights to different terms in measuring similarity; one common approach is called *term frequency–inverse document frequency (TFIDF)*, which downweights terms that appear in many documents in the corpus. TFIDF is due to the British computer scientist Karen Spärck Jones (1935–2007); she developed it in the early 1970s [64], predating by decades the major shift in natural language processing from purely symbolic techniques to statistical ones.

Of course, real information retrieval systems are usually quite a lot more complicated than we've discussed so far. For example, a document that talks about *sofas* would be judged to be completely unrelated to a document that

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

talks about *couches*, which seems a little foolish. Handling synonyms requires a more complicated approach, often based around analyzing the *term–document matrix* that simultaneously represents the entire corpus. (For example, if documents that discuss *sofas* use very similar *other* words to documents that discuss *couches*—like *change* and *cushion* and *nap*—then we might be able to infer something about *sofas* and *couches*.) A recent flurry of research has attempted to find new ways to build a vector representation for *words,* often using machine-learning techniques to train the representation on large amounts of human-written texts. A prominent recent example is the *word2vec* system [87]. This algorithm embedding can do some remarkable things—including doing a good job of solving analogy problems using vector arithmetic—but there are also some reasons to be very cautious about using these tools. Researchers have shown that the embeddings themselves, and the analogies they produce, can be racist and sexist—in ways that, distressingly, reproduce the racism and sexism in the training data [17, 25].

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science.* This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

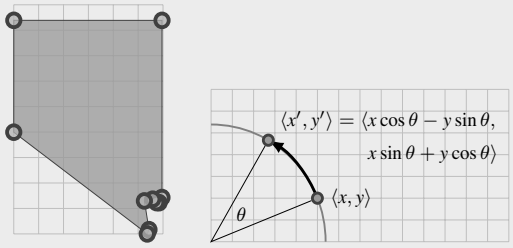
Computer Science Connections

Rotation Matrices

When an image is *rendered* (drawn) using computer graphics, we typically proceed by transforming a 3-dimensional representation of a *scene*, a model of the world, into a 2-dimensional *image* fit for a screen. The scene is typically represented by a collection of points in $\mathbb{R}^3$, each defining a vertex of a polygon. The *camera* (the eye from which the scene is viewed) is another point in $\mathbb{R}^3$, with an orientation describing the direction of view. We then *project* the polygons' points into $\mathbb{R}^2$.

Nearly every step of the rendering computation is done using matrix multiplications, taking into account the position and direction of view of the camera, and the position of the given point. While a full account of this rendering algorithm isn't *too* difficult, we'll consider a simpler problem that still includes the interesting matrix computations: the *rotation* of a set of points in $\mathbb{R}^2$ by an angle θ. (You can learn more about the way that the full-scale computer graphics algorithms work in [61]. The full-scale problem requires thinking about the angle of view with two parameters, akin to "azimuth" and "elevation" in orienteering: the direction θ in the horizontal plane and the angle ϕ away from a straight horizontal view. Other image transformations that are important in graphics need a slick, if counter-intuitive, trick to be represented by matrix operations: it's common to represent a 3-dimensional point by a 4-vector, rather than a 3-vector, which allows all of the most common transformations to be implemented using matrices.)

Suppose that we have a scene that consists of a collection of points in $\mathbb{R}^2$. As an example, Figure 2.42a shows a collection of hand-collected points in $\mathbb{R}^2$ that represent the borders of the state of Nevada. Now imagine that we wish to rotate Nevada by some angle θ—that is, to rotate each boundary point $\langle x, y\rangle$ by an angle θ around the point $\langle 0, 0\rangle$. If you dig deep into your dim recollections of trigonometry, you'd be able to convince yourself that we can rotate a point $\langle x, 0\rangle$ around the point $\langle 0, 0\rangle$ by moving it to $\langle x\cos\theta, x\sin\theta\rangle$. More generally, the point $\langle x, y\rangle$ becomes the point $\langle x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta\rangle$ when it's rotated by an angle θ. (See Figure 2.42b.)



(a) Ten points representing Nevada.

(b) A point $\langle x, y\rangle$ rotated by an angle θ around the origin.

Figure 2.42 Some points in $\mathbb{R}^2$, and how to rotate them.

Now, to rotate a sequence of points $\langle x_1, y_1\rangle, \ldots, \langle x_n, y_n\rangle$ by an angle θ, we'll use matrices. Write a matrix with the ith column corresponding to the ith point, and perform matrix multiplication as follows:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \end{bmatrix} = \begin{bmatrix} x_1\cos\theta - y_1\sin\theta & x_2\cos\theta - y_2\sin\theta & \cdots & x_n\cos\theta - y_n\sin\theta \\ x_1\sin\theta + y_1\cos\theta & x_2\sin\theta + y_2\cos\theta & \cdots & x_n\sin\theta + y_n\cos\theta \end{bmatrix}.$$

(The matrix $R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$ is called a *rotation matrix*.)

The result is that we have rotated an entire collection of points—arranged in the 2-by-n matrix M—by multiplying M by this rotation matrix: in other words, RM is a 2-by-n matrix of the rotated points. (See Figure 2.43.)

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

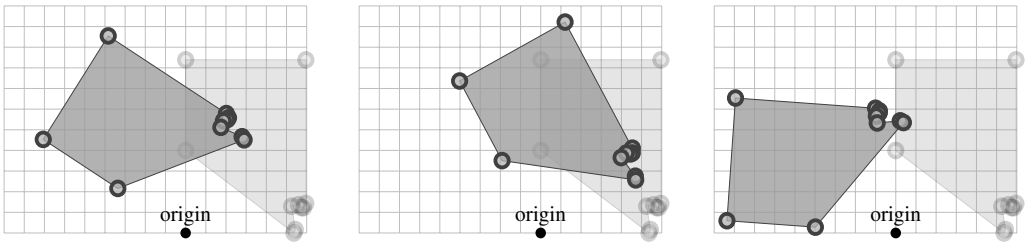


Figure 2.43 Nevada, rotated by three different angles.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

EXERCISES

2.141 Write out all of the elements of $\{1,2,3\} \times \{1,4,16\}$.

2.142 Do the same for $\{1,4,16\} \times \{1,2,3\}$.

2.143 Do the same for $\{1\} \times \{1\} \times \{1\}$.

2.144 Do the same for $\{1,2\} \times \{2,3\} \times \{1,4,16\}$.

2.145 Suppose $A \times B = \{\langle 1,1\rangle, \langle 2,1\rangle\}$. What are A and B?

2.146 Let $S = \{1,2,3,4,5,6,7,8\}$, and let T be an unknown set. Suppose that $|S \times T| = 16$ and $\langle 1,2\rangle, \langle 3,4\rangle \in S \times T$. What can you conclude about T? Be as precise as possible: if you can list the elements of T exhaustively, do so; if you can't, identify any elements that you can conclude must be (or must not be) in T.

2.147 Again with $S = \{1,2,3,4,5,6,7,8\}$, suppose that $S \times T = \varnothing$. What can you conclude about T?

2.148 Again with $S = \{1,2,3,4,5,6,7,8\}$, suppose that $(S \times T) \cap (T \times S) = \{\langle 3,3\rangle\}$. What can you conclude about T?

2.149 Again with $S = \{1,2,3,4,5,6,7,8\}$, suppose that $S \times T = T \times S$. What can you conclude about T?

Recall that Algebraic notation denotes the squares of the chess board as $\{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e},\mathsf{f},\mathsf{g},\mathsf{h}\} \times \{1,2,3,4,5,6,7,8\}$*, as in Figure 2.44a. For each of the following questions, identify sets S and T such that the set of cells containing the designated pieces can be described as* $S \times T$*.*

2.150 the white rooks (♖)

2.151 the bishops (♗, white or black)

2.152 the pawns (♙, white or black)

2.153 no pieces at all

2.154 Enumerate the elements of $\{0,1,2\}^3$.

2.155 Enumerate the elements of $\{\mathtt{A},\mathtt{B}\} \times \{\mathtt{C},\mathtt{D}\}^2 \times \{\mathtt{E}\}$.

2.156 Enumerate the elements of $\bigcup_{i=1}^{3} \{0,1\}^i$.

Let $\Sigma = \{\mathtt{A},\mathtt{B},\ldots,\mathtt{Z}\}$ *denote the English alphabet. Using notation from this chapter, give an expression that denotes each of the following sets. It may be useful to recall that* Σ^k *denotes the set of strings consisting of a sequence of k elements from* Σ*, so* Σ^0 *contains the unique string of length* 0 *(called the* empty string*, and typically denoted by* ϵ*—or by* `""` *in most programming languages).*

2.157 The set of 8-letter strings.

2.158 The set of 5-letter strings that do not contain any vowels $\{\mathtt{A},\mathtt{E},\mathtt{I},\mathtt{O},\mathtt{U}\}$.

2.159 The set of 6-letter strings that do not contain more than one vowel. (So `GRITTY`, `QWERTY`, and `BRRRRR` are fine; but `EEEEEE`, `THREAT`, `STRENGTHS`, and `A` are not.)

2.160 The set of 6-letter strings that contain at most one type of vowel—multiple uses of the same vowel are fine, but no two different vowels can appear. (So `BANANA`, `RHYTHM`, and `BOOBOO` are fine; `ESCAPE` and `STRAIN` are not.)

Recall that the length of a vector $x \in \mathbb{R}^n$ *is given by* $\|x\| = \sqrt{\sum_{i=1}^{n} x_i^2}$*. Considering the vectors* $a = \langle 1,3\rangle$*,* $b = \langle 2,-2\rangle$*,* $c = \langle 4,0\rangle$*, and* $d = \langle -3,-1\rangle$*, state the values of each of the following:*

2.161 $\|a\|$

2.162 $\|b\|$

2.163 $\|c\|$

2.164 $a + b$

2.165 $3d$

2.166 $2a + c - 3b$

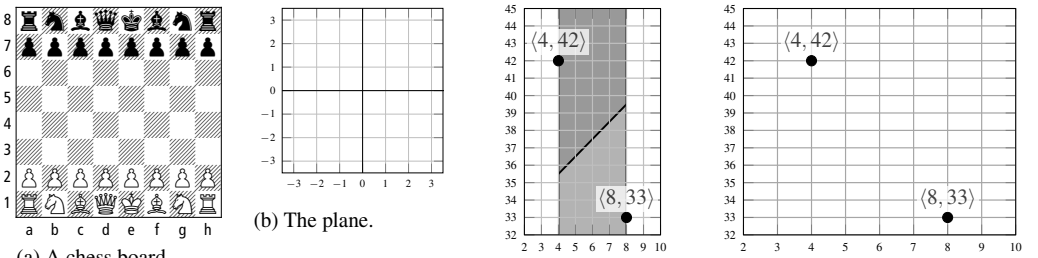
2.167 $\|a\| + \|c\|$ and $\|a + c\|$

2.168 $\|a\| + \|b\|$ and $\|a + b\|$

2.169 $3\|d\|$ and $\|3d\|$

2.170 Explain why, for an arbitrary vector $x \in \mathbb{R}^n$ and an arbitrary scalar $a \in \mathbb{R}$, $\|ax\| = a\|x\|$.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.



(a) A chess board. (b) The plane. (c) Manhattan train stations (unscaled). (d) Manhattan train stations (scaled).

Figure 2.44 A few two-dimensional grids: a chess board, the Cartesian plane, and Manhattan (twice).

2.171 For any two vectors $x, y \in \mathbb{R}^n$, we have $\|x\|+\|y\| \geq \|x+y\|$. Under precisely what circumstances do we have $\|x\|+\|y\| = \|x+y\|$ for $x, y \in \mathbb{R}^n$? Explain briefly.

2.172 For the vectors $a = \langle 1, 3\rangle$ and $b = \langle 2, -2\rangle$, what is $a \bullet b$?

2.173 For the vectors $a = \langle 1, 3\rangle$ and $d = \langle -3, -1\rangle$, what is $a \bullet d$?

2.174 For the vector $c = \langle 4, 0\rangle$, what is $c \bullet c$?

2.175 Recall the definitions of *Manhattan distance* and *Euclidean distance* between two vectors $x, y \in \mathbb{R}^n$:

$$\text{Manhattan distance} = \sum_{i=1}^{n} |x_i - y_i| \qquad \text{Euclidean distance} = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}.$$

What are the Manhattan and Euclidean distances between the vectors $a = \langle 1, 3\rangle$ and $b = \langle 2, -2\rangle$?

2.176 What are the Manhattan and Euclidean distances between $a = \langle 1, 3\rangle$ and $d = \langle -3, -1\rangle$?

2.177 What are the Manhattan and Euclidean distances between $b = \langle 2, -2\rangle$ and $c = \langle 4, 0\rangle$?

2.178 What's the *largest* possible Euclidean distance between x and y if their Manhattan distance is 1? Justify.

2.179 What's the *smallest* possible Euclidean distance between x and y if their Manhattan distance is 1? Justify.

2.180 What's the smallest possible Euclidean distance between x and y if $x, y \in \mathbb{R}^n$ (not just $n = 2$) if their Manhattan distance is 1? Explain.

2.181 Consider Figure 2.44b, and sketch the set $\{x \in \mathbb{R}^2 :$ the Euclidean distance between x and $\langle 0, 0\rangle$ is at most $2\}$.

2.182 Consider Figure 2.44b, and sketch the set $\{x \in \mathbb{R}^2 :$ the Manhattan distance between x and $\langle 0, 0\rangle$ is at most $2\}$.

In Example 2.40, we considered two train stations located at points $s = \langle 8, 33\rangle$ and $g = \langle 4, 42\rangle$. (See Figure 2.44c.) In that example, we showed that, for an offset $\delta \in [0, 4]$, the Manhattan distance between the point $\langle 4 + \delta, y\rangle$ and s is smaller than the Manhattan distance between the point $\langle 4 + \delta, y\rangle$ and g when $y < 35.5 + \delta$.

2.183 Show that the point $\langle 16, 40\rangle$ is closer to one station under Manhattan distance, and to the other under Euclidean distance.

2.184 Let $\delta \geq 0$. Under Manhattan distance, describe the values of y for which the point $\langle 8 + \delta, y\rangle$ is closer to s than to g.

2.185 Let $\delta \geq 0$. Under Manhattan distance, describe the values of y for which the point $\langle 4 - \delta, y\rangle$ is closer to s than to g.

2.186 In the real-world island of Manhattan, the east–west blocks are roughly twice the length of the north–south blocks. As such, the more accurate picture of distances in the city is shown in Figure 2.44d. Assuming it takes 1.5 minutes to walk a north–south (up–down) block and 3 minutes to walk an east–west (left–right) block, give a formula for the walking distance between $\langle x, y\rangle$ and Penn Station, at $s = \langle 8, 33\rangle$.

2.187 A *Voronoi diagram*—named after the Ukrainian mathematician Georgy Voronoy (1868–1908)—is a decomposition of the plane $\mathbb{R}^2$ into regions based on a given set S of points. The region "belonging" to a point $x \in S$ is $\{y \in \mathbb{R}^2 : d(x, y) \leq \min_{z \in S} d(z, y)\}$,

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

where $d(\cdot,\cdot)$ denotes Euclidean distance. (In other words, the region "belonging" to point x is that portion of the plane that's closer to x than any other point in S.) Compute the Voronoi diagram of the set of points $\{\langle 0,0\rangle, \langle 4,5\rangle, \langle 3,1\rangle\}$. That is, compute:

- the set of points $y \in \mathbb{R}^2$ that are closer to $\langle 0,0\rangle$ than $\langle 4,5\rangle$ or $\langle 3,1\rangle$ under Euclidean distance;
- the set of points $y \in \mathbb{R}^2$ that are closer to $\langle 4,5\rangle$ than $\langle 0,0\rangle$ or $\langle 3,1\rangle$ under Euclidean distance; and
- the set of points $y \in \mathbb{R}^2$ that are closer to $\langle 3,1\rangle$ than $\langle 0,0\rangle$ or $\langle 4,5\rangle$ under Euclidean distance.

2.188 Compute the Voronoi diagram of the set of points $\{\langle 2,2\rangle, \langle 8,1\rangle, \langle 5,8\rangle\}$.

2.189 Compute the Voronoi diagram of the set of points $\{\langle 0,7\rangle, \langle 3,3\rangle, \langle 8,1\rangle\}$.

2.190 *(programming required.)* Write a program that takes three points as input and produces a representation of the Voronoi diagram of those three points as output.

> **Taking it further:** Voronoi diagrams are used frequently in computational geometry, among other areas of computer science. (For example, a coffee-shop chain might like to build a mobile app that is able to quickly answer the question *What store is closest to me right now?* for any customer at any time. Voronoi diagrams can allow precomputation of these answers.) Given any set S of n points, it's reasonably straightforward to compute (an inefficient representation of) the Voronoi diagram of those points by computing the line that's equidistant between each pair of points, as you saw in the last few exercises. But there are cleverer ways of computing Voronoi diagrams more efficiently; for more, see a good textbook on computational geometry, like [36].

2.191 What size is the matrix M in Figure 2.45?

2.192 For the matrix M in Figure 2.45, what is $M_{3,1}$?

2.193 For the matrix M in Figure 2.45, list every $\langle i,j\rangle$ such that $M_{i,j} = 7$.

2.194 For the matrix M in Figure 2.45, what is $3M$?

Considering the matrices in Figure 2.45, what are the values of the given expressions (if they're defined)? (If the given quantity is undefined, say so—and say why.)

2.195 $A + C$

2.196 $B + F$

2.197 $D + E$

2.198 $A + A$

2.199 $-2D$

2.200 $0.5F$

2.201 AB

2.202 AC

2.203 AF

2.204 BC

$$M = \begin{bmatrix} 3 & 9 & 2 \\ 0 & 9 & 8 \\ 6 & 2 & 0 \\ 7 & 5 & 5 \\ 7 & 2 & 4 \\ 1 & 6 & 7 \end{bmatrix} \quad A = \begin{bmatrix} 0 & 8 & 0 \\ 9 & 6 & 0 \\ 2 & 3 & 3 \end{bmatrix} \quad B = \begin{bmatrix} 5 & 8 \\ 7 & 5 \\ 3 & 2 \end{bmatrix} \quad C = \begin{bmatrix} 7 & 2 & 7 \\ 3 & 5 & 6 \\ 1 & 2 & 5 \end{bmatrix} \quad D = \begin{bmatrix} 3 & 1 \\ 0 & 8 \end{bmatrix}$$

$$E = \begin{bmatrix} 8 & 4 \\ 3 & 2 \end{bmatrix} \quad F = \begin{bmatrix} 1 & 2 & 9 \\ 5 & 4 & 0 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \quad H = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 2.45 An assortment of matrices.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.205 *DE*

2.206 *ED*

2.207 For G and H from Figure 2.45, what is $0.25G + 0.75H$?

2.208 What is $0.5G + 0.5H$?

2.209 Identify two *other* matrices G' and H' with the same average—that is, $\{G, H\} \neq \{G', H'\}$ but $0.5G + 0.5H = 0.5G' + 0.5H'$.

2.210 *(programming required.)* A common computer graphics effect in the spirit of the last few exercises is *morphing* one image into another—that is, slowly changing the first image into the second. There are sophisticated techniques for this task, but a simple form can be achieved just by averaging. Given two n-by-m images represented by matrices A and B—say grayscale images, with each entry in $[0, 1]$—we can produce a "weighted average" of the images as $\lambda A + (1 - \lambda)B$, for a parameter $\lambda \in [0, 1]$. See Figure 2.46. Write a program, in a programming language of your choice, that takes three inputs—an image A, an image B, and a weight $\lambda \in [0, 1]$—and produces a new image $\lambda A + (1 - \lambda)B$. (You'll need to research an image-processing library to use in your program.)

2.211 Let A be an m-by-n matrix. Let I be the n-by-n identity matrix. Explain why the matrix AI is identical to the matrix A.

If M is an n-by-n matrix, then the product of M with itself is also an n-by-n matrix. We write matrix powers in the normal way that we defined powers of integers (or of the Cartesian product of sets): $M^k = M \cdot M \cdots M$, multiplied k times. (M^0 is the n-by-n identity matrix I.)

2.212 What is J^3? (See Figure 2.47.)

2.213 What is K^2? (See Figure 2.47.)

2.214 What is K^4?

2.215 What is K^9? *(Hint: $M^{2k} = (M^k)^2$.)*

Taking it further: The *Fibonacci numbers* are defined recursively as the sequence $f_1 = 1, f_2 = 1$, and $f_n = f_{n-1} + f_{n-2}$ for $n \geq 3$. The first several Fibonacci numbers are $1, 1, 2, 3, 5, 8, 13, \ldots$. As we'll see in Exercises 5.62 and 6.99, there's a very fast algorithm to find the nth Fibonacci number based on computing K^n.

2.216 Let A by an n-by-n matrix. The *inverse* of A, denoted A^{-1}, is also an n-by-n matrix, with the property that $AA^{-1} = I$. There's a general algorithm that one can develop to invert matrices, but here you'll calculate inverses by hand. Note that

$$\underbrace{\begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix}}_{A} \cdot \underbrace{\begin{bmatrix} x & y \\ z & w \end{bmatrix}}_{B} = \begin{bmatrix} x+z & y+w \\ 2x+z & 2y+w \end{bmatrix}.$$

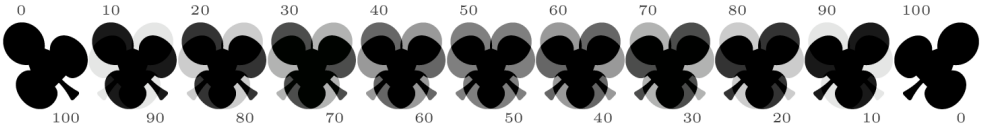


Figure 2.46 Morphing one rotated ♣ symbol into another.

$$J = \begin{bmatrix} 2 & 3 \\ 1 & 1 \end{bmatrix} \quad K = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad N = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad P = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad Q = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Figure 2.47 Another assortment of matrices.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Thus A^{-1} is B if the following four conditions hold: $x+z=1$ and $y+w=0$ and $2x+z=0$ and $2y+w=1$. Find the values of x, y, w, and z that satisfy these four conditions.

2.217 Using the same approach as in the previous exercise, find the inverse of the matrix L from Figure 2.47.

2.218 Using the same approach, find the inverse of N from Figure 2.47.

2.219 Using the same approach, find the inverse of P from Figure 2.47.

2.220 Not all matrices have inverses—for example, Q from Figure 2.47 doesn't have an inverse. Explain why not.

An error-correcting code *(see Section 4.2) is a method for redundantly encoding information so that the information can still be retrieved even in the face of some errors in transmission/storage. The* Hamming code *is a particular error-correcting code for* 4*-bit chunks of information. The Hamming code can be described using matrix multiplication: given a* message $m \in \{0,1\}^4$*, we* encode *m as mG* mod 2*, where*

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

*(Here you should interpret the "*mod 2*" as describing an operation to* each *element of the output vector.)*

For example, $[1,1,1,1] \cdot G = [1,1,1,1,3,3,3]$*, so we'd encode* $[1,1,1,1]$ *as* $[1,1,1,1,3,3,3] \bmod 2 = [1,1,1,1,1,1,1]$.

2.221 What is the Hamming code encoding of $[0,0,0,0]$?

2.222 What is the Hamming code encoding of $[0,1,1,0]$?

2.223 What is the Hamming code encoding of $[1,0,0,1]$?

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.5 Functions

> I am told there are people who do not care for maps, and find it hard to believe.
>
> Robert Louis Stevenson (1850–1894)
> in *My First Book* (1894), writing about *Treasure Island*

A *function* transforms an input value into an output value; that is, a function f takes an *argument* or *parameter* x, and *returns* a value $f(x)$. Functions are familiar from both algebra and from programming. In algebra, we frequently encounter mathematical functions like $f(x) = x+6$, which means that, for example, we have $f(3) = 9$ and $f(4) = 10$. In programming, we often write or invoke functions that use an algorithm to transform an input into an output, like a function **sort**—so that **sort**$(\langle 3,1,4,1,5,9\rangle) = \langle 1,1,3,4,5,9\rangle$, for example.

In this section, we'll give definitions of functions and of some terminology related to functions, and also discuss a few special types of functions. (Functions themselves are a special case of *relations,* and we will revisit the definition of functions when we discuss them in Chapter 8.)

2.5.1 Basic Definitions

We start with the definition of a function itself:

Definition 2.46: Function.
Let A and B be sets. A *function f from A to B*, written $f : A \to B$, assigns to each input value $a \in A$ a unique output value $b \in B$; the unique value b assigned to a is denoted by $f(a)$. We sometimes say that f *maps a to $f(a)$*.

Note that A and B are allowed to be the same set; for example, a function might have inputs and outputs that are both elements of $\mathbb{Z}$.

Here are two small examples: first, a function *not* for Boolean inputs that maps True to False, and False to True; and, second, a function *square* that returns its input multiplied by itself.

Example 2.48: Not function.
The function $not : \{\text{True}, \text{False}\} \to \{\text{True}, \text{False}\}$ can be defined with the table in Figure 2.48. Given an input x, we find the output value $not(x)$ by locating x in the first column of the table and reading the value in that row's second column. Thus $not(\text{True}) = \text{False}$ and $not(\text{False}) = \text{True}$.

Example 2.49: Square function.
The function $square : \mathbb{R} \to \mathbb{R}$ can be defined as $square(x) = x^2$: for any input $x \in \mathbb{R}$, the output is the real number x^2. Thus, for example, $square(8) = 64$, because the function *square* assigns the output $8^2 = 64$ to the input 8.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

x	$not(x)$
True	False
False	True

x	$double(x)$
0	0
1	2
2	4
3	6
4	8
5	10
6	12
7	14

$$quantize(n) = \begin{cases} 26 & \text{if } 0 \le n \le 51 \\ 78 & \text{if } 52 \le n \le 103 \\ 130 & \text{if } 104 \le n \le 155 \\ 182 & \text{if } 156 \le n \le 207 \\ 234 & \text{if } 208 \le n \le 255 \end{cases}$$

Figure 2.48 The functions *not* and *double* specified using a table, and *quantize* specified by cases.

Note, too, that a function $f : A \to B$ might have a set A of inputs that are *pairs*; for example, the function that takes two numbers and returns their average is the function $average : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, where $average(\langle x, y \rangle) = (x + y)/2$. (We interpret $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$ as $(\mathbb{R} \times \mathbb{R}) \to \mathbb{R}$.) When there is no danger of confusion, we drop the angle brackets and simply write, for example, $average(3, 2)$ instead of $average(\langle 3, 2 \rangle)$.

As we've already seen in Examples 2.48 and 2.49, the rule by which a function assigns an output to a given input can be specified either symbolically—typically via an algebraic expression—or exhaustively, by giving a table describing the input/output relationship. The table-based definition only makes sense when the set of possible inputs is *finite*; otherwise the table would have to be infinitely large. (And it's only *practical* to define a function with a table if the set of possible inputs is pretty small!)

Here's an example of specifying the same function in two different ways, once symbolically and once using a table:

Example 2.50: Doubling function.
Let's define the function *double* that doubles its input value, for any input in $\{0, 1, \ldots, 7\}$. (That is, we are defining a function $double : \{0, 1, \ldots, 7\} \to \mathbb{Z}$.) We can write *double* symbolically by defining $double(x) = 2 \cdot x$. To define *double* using a table, we specify the output corresponding to every one of the 8 possible inputs, as shown in Figure 2.48.

The functions that we've discussed so far are all fairly simple, but even simple functions can have some valuable applications. Here's one that compresses images so that they take up less space:

Example 2.51: Reducing the colorspace of an image.
The pixels in a grayscale image are all elements of $\{0, 1, \ldots, 255\}$. To reduce the space requirements for a large image, we can consider a form of *lossy compression* (that is, compression that loses some amount of data) by replacing each pixel with one chosen from a smaller list of candidate colors. That is, instead of having 256 different shades of gray, we might have 128 or 64 or even fewer shades. Define $quantize : \{0, 1, \ldots, 255\} \to \{0, 1, \ldots, 255\}$ as shown in Figure 2.48. We can apply *quantize* to every pixel in a grayscale image, and then use a much smaller number of bits per pixel in storing the resulting image. See Figure 2.49 for an example.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

(a) *quantize*

(b) Rear Admiral Grace Murray Hopper (1906–1992), in 1976, when she was a captain.

(c) The same image, compressed to use only 5 shades of gray using the *quantize* function.

Figure 2.49 Colorspace reduction. In PNG format, image (c) uses only $\approx 15\%$ of the space consumed by image (b).

Taking it further: In 1976 (the date of the image in Figure 2.49), Grace Hopper was the Head of the Navy Programming Language Section of the Office of the Chief of Naval Operations, in the U.S. Navy. She worked on the team that developed some of the very first computers—the Harvard Mark I and the UNIVAC. (Some UNIVAC manuals are visible behind her.) She coined the name "compiler" (see p. 3-34) and wrote the first of them, and was a major player in the design of influential programming languages like COBOL that, for the first time, used English words as part of their syntax.

As for images themselves: a *byte* is a sequence of 8 bits. Using 8 bits, we can represent the numbers from 00000000 to 11111111—that is, from 0 to 255. Thus a pixel with $\{0, 1, \ldots, 255\}$ as possible grayscale values in an image requires one byte of storage for each pixel. If we don't do something cleverer, a moderately sized 2048-by-1536 image (the size of many iPads) requires over 3 megabytes even if it's grayscale. (A color image requires three times that amount of space.) Techniques similar to the compression function from Example 2.51 are used in a variety of CS applications—including, for example, in automatic speech recognition, where each sample from a sound stream is stored using one of only, say, 256 different possible values instead of a floating-point number, which requires much more space.

Domain and codomain

The *domain* and *codomain* of a function are its sets of possible inputs and outputs:

Definition 2.47: Domain/codomain.
For a function $f : A \to B$, the set A is called the *domain* of the function $f : A \to B$, and the set B is called the *codomain* of the function $f : A \to B$.

Let's identify the domain and codomain from the previous examples of this section:

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Example 2.52: Some domains and codomains.
For the functions from Examples 2.48–2.51:

The function *not* (Example 2.48) has domain $\{\text{True}, \text{False}\}$ and codomain $\{\text{True}, \text{False}\}$.

The function *square* (Example 2.49) has domain $\mathbb{R}$ and codomain $\mathbb{R}$.

The function *double* (Example 2.50) has domain $\{0, 1, \ldots, 7\}$ and codomain $\mathbb{Z}$.

Both the domain and codomain of *quantize* (Example 2.51) are $\{0, 1, \ldots, 255\}$.

Note that for three of these functions, the domain and codomain are actually the same set; for the function $double : \{0, 1, \ldots, 7\} \to \mathbb{Z}$, they're different.

When the domain and codomain are clear from context (or they are unimportant for the purposes of a discussion), then they may be left unwritten.

Taking it further: This possibility of implicitly representing the domain and codomain of a function is also present in code. Some programming languages (like Java) require the programmer to explicitly write out the types of the inputs and outputs of a function; in some (like Python), the input and output types are left implicit. In Java, for example, one would write an `isPrime` function with the explicit declaration that the input is an integer (`int`) and the output is a Boolean (`boolean`). In Python, one would write the function without any explicit type information.

```
boolean isPrime(int n) {
   /* code to check primality of n */
}
```

```
def isPrime(n):
    # code to check primality of n
```

But regardless of whether they're written out or left implicit, these functions *do* have a domain (the set of valid inputs) and a codomain (the set of possible outputs).

Range/Image

For a function $f : A \to B$, the set A (the domain) is the set of all possible inputs, and the set B (the codomain) is the set of all possible outputs. But not all of the possible outputs are necessarily actually *achieved*: in other words, there may be an element $b \in B$ for which there's no $a \in A$ with $f(a) = b$. For example, we defined $square : \mathbb{R} \to \mathbb{R}$ in Example 2.49, but there is no real number x such that $square(x) = -1$. The *range* or *image* defines the set of actually achieved outputs:

Definition 2.48: Range/image.
The *range* or *image* of a function $f : A \to B$ is the set of all $b \in B$ such that $f(a) = b$ for some $a \in A$. Using the notation of Section 2.3, the range of f is the set

$$\{y \in B : \text{there exists at least one } x \in A \text{ such that } f(x) = y\}.$$

We'll start with the functions from earlier in the section, and then look at a slightly more complex example:

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Example 2.53: Some ranges.
For *not* (Example 2.48), *double* (Example 2.50), and *quantize* (Example 2.51), the range is easy to determine: it's the set of values that appear in the "output" column of the table defining the function. The ranges are $\{\text{True}, \text{False}\}$ for *not*; $\{0, 2, 4, 6, 8, 10, 12, 14\}$ for *double*; and $\{26, 78, 130, 182, 234\}$ for *quantize*.

For *square* (Example 2.49), it's clear that the range includes no negative numbers, because there's no $y \in \mathbb{R}$ such that $y^2 < 0$. In fact, the range of *square* is precisely $\mathbb{R}^{\geq 0}$: for any $x \in \mathbb{R}^{\geq 0}$, there's an input to *square* that produces x as output—specifically $\sqrt{x}$.

Example 2.54: The smallest divisor function.
Define a function $sd : \mathbb{Z}^{\geq 2} \to \mathbb{Z}^{\geq 2}$ as follows. Given an input $n \in \mathbb{Z}^{\geq 2}$, the value of $sd(n)$ is the *smallest integer* $k \geq 2$ *that evenly divides* n. For example:

$sd(2) = 2$ because $2 \mid 2$

$sd(3) = 3$ because $3 \mid 3$ but $2 \nmid 3$

$sd(4) = 2$ because $2 \mid 4$

$sd(121) = 11.$ because $11 \mid 121$ but $2 \nmid 121, 3 \nmid 121, \ldots, 10 \nmid 121$

What are the domain, codomain, and range of *sd*?

Solution. The domain and codomain of *sd* are easy to determine: they are both $\mathbb{Z}^{\geq 2}$. Any integer $n \geq 2$ is a valid input to *sd*, and we defined the function *sd* as producing an integer $k \geq 2$ as its output. (The domain and codomain are simply written in the function's definition, before and after the arrow in $sd : \mathbb{Z}^{\geq 2} \to \mathbb{Z}^{\geq 2}$.) The range is a bit harder to see, but it turns out to be the set P of all prime numbers. Let's argue that P is the range of *sd* by showing that (i) every prime number $p \in P$ *is* in the range of *sd*, and (ii) every number p in the range of P is a prime number.

Claim (i). Let $p \in \mathbb{Z}^{\geq 2}$ be any prime number. Then $sd(p) = p$: by the definition of primality, the only integers than evenly divide p are 1 and p itself (and $1 \geq 2$ isn't true!). Therefore every prime number p is in the range of *sd*, because there's an input to *sd* such that the output is p.

Claim (ii). Let p be any number in the range of *sd*—that is, suppose $sd(n) = p$ for some n. We will argue that p must be prime. Imagine that p were instead composite—that is, there is an integer k satisfying $2 \leq k < p$ that evenly divides p. But then $sd(n) = p$ is impossible: if p evenly divides n, then k *also* evenly divides n, and $k < p$, so k would be a smaller divisor of n. (For example, if n were evenly divisible by the composite number 15, then n would *also* be evenly divisible by 3 and 5—two factors of 15—so $sd(n) \neq 15$.) Therefore every number in the range of *sd* is prime.

Together (i) and (ii) let us conclude that the range of *sd* is precisely the set of all prime numbers.

Problem-solving tip: Example 2.54 illustrates a useful general technique if we wish to show that two sets A and B are equal. One nice way to establish that $A = B$ is to show that $A \subseteq B$ and $B \subseteq A$. That's what we did to establish the range of *sd* in Example 2.54. Let's call R the range of *sd*. We showed in (i) that every element of P is in R (that is, $P \subseteq R$); and in (ii) that every element of R is in P (that is, $R \subseteq P$). Together these facts establish that $R = P$.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

We will also introduce a minor extension to the set-abstraction notation from Section 2.3 that's related to the range of a function. (We used this notation informally in Example 2.28.) Consider a function $f : A \to B$ and a set $U \subseteq A$. We denote by $\{f(x) : x \in U\}$ the set of all output values of the function f when it's applied to the elements $x \in U$:

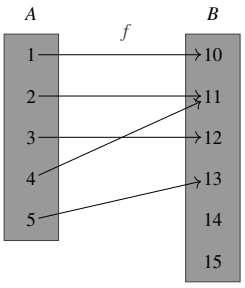
Definition 2.49: Set abstraction using functions.
For a function $f : A \to B$ and a set $U \subseteq A$, we write $\{f(x) : x \in U\}$ as shorthand for the set $\{b \in B :$ there exists some $u \in U$ for which $f(u) = b\}$.

Remember that order and repetition of elements in a set don't matter, which means that the set $\{f(x) : x \in A\}$ is precisely the range of the function $f : A \to B$.

A visual representation of functions

The table-based and symbolic representations of functions that we've discussed fully represent a function, but sometimes a more visual representation of a function is clearer. Consider a function $f : A \to B$. We can give a picture representing f by putting the elements of A into one column, the elements of B into a second column, and drawing an arrow from each $a \in A$ to the value of $f(a) \in B$. Notice that the definition of a function guarantees that *every element in the first column has one and only one arrow going from it to the second column*: if $f : A \to B$ is a function, then every $a \in A$ is assigned a unique output $f(a) \in B$.

Figure 2.50 shows a small example. We can read the domain, codomain, and range directly from this picture: the domain is the set of elements in the first column; the codomain is the set of elements in the second column; and the range is the set of elements in the second column *for which there is at least one incoming arrow.* In Figure 2.50, the range of f is $\{10, 11, 12, 13\}$. (There are no arrows pointing to 14 or 15, so these two numbers are in the codomain but not the range of f.)



The function f as a table:

x	$f(x)$
1	10
2	11
3	12
4	11
5	13

Figure 2.50 A picture of a function $f : A \to B$, where $A = \{1, \ldots, 5\}$ and $B = \{10, \ldots, 15\}$.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science.* This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Function composition

Suppose we have two functions $f : A \to B$ and $g : B \to C$. Given an input $a \in A$, we can find $f(a) \in B$, and then apply g to map $f(a)$ to an element of C, namely $g(f(a)) \in C$. This successive application of f and g defines a new function, called the *composition* of f and g, whose domain is A and whose codomain is C.

Definition 2.50: Function composition.
For two functions $f : A \to B$ and $g : B \to C$, the function $g \circ f : A \to C$ maps an element $a \in A$ to $g(f(a)) \in C$. The function $g \circ f$ is called the *composition of f and g*.

Notice a slight oddity of the notation: $g \circ f$ applies the function f *first* and the function g *second*, even though g is written first.

Example 2.55: Function composition, four ways.
Let $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$ be defined by $f(x) = 2x + 1$ and $g(x) = x^2$. Then:

- The function $g \circ f$, given an input x, produces output
$$g(f(x)) = g(2x + 1) = (2x + 1)^2 = 4x^2 + 4x + 1.$$
- The function $f \circ g$ maps x to $f(g(x)) = f(x^2) = 2x^2 + 1$.
- The function $g \circ g$ maps x to $g(g(x)) = g(x^2) = (x^2)^2 = x^4$.
- The function $f \circ f$ maps x to $f(f(x)) = f(2x + 1) = 2(2x + 1) + 1 = 4x + 3$.

As with many function-related concepts, the visual representation of functions gives a nice way of thinking about function composition: the function $g \circ f$ corresponds to the "short-circuiting" of the pictures of the functions f and g. A small example of this visualization is shown in Figure 2.51. The composition $g \circ f$ is given by following *two* arrows in the diagram: one arrow defined by f, and then one arrow defined by g.

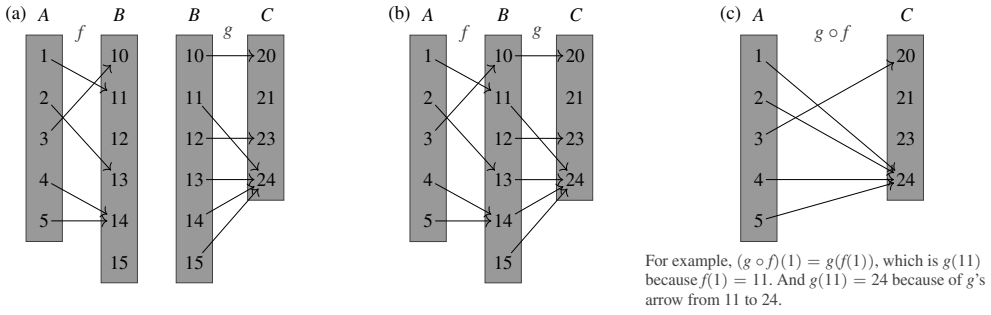


Figure 2.51 Two functions $f : A \to B$ and $g : B \to C$, (a) separately and (b) pasted together. Their composition $g \circ f$ is shown in (c), based on successively following two arrows from (b).

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.5.2 Onto and One-to-One Functions

We now turn to two special categories of functions—*onto* and *one-to-one* functions—distinguished by how many different input values (always at least one? never more than one?) are mapped to each output value.

Onto functions

A function $f : A \to B$ is *onto* if every *possible* output in B is, in fact, an *actual* output:

Definition 2.51: Onto functions.
A function $f : A \to B$ is called *onto* if, for every $b \in B$, there exists at least one $a \in A$ for which $f(a) = b$.
An onto function is also sometimes called a *surjective* function.

Alternatively, using the terminology of Section 2.5.1, a function f is onto if f's codomain equals f's range. It may be easier to think about onto functions using the visual representation of functions: a function f is onto if *there's at least one arrow pointing at every element in the second column*. (See Figure 2.52.)

As an example, here are two of our previous functions, one of which is onto and one of which isn't:

Example 2.56: An onto function and a non-onto function.
The function $not : \{\text{True}, \text{False}\} \to \{\text{True}, \text{False}\}$ is onto: there's an input value that produces True (namely False), and there's an input value that produces False (namely True). Every element of the codomain is "hit" by *not*, so the function is onto.

On the other hand, the function $quantize : \{0, 1, \ldots, 255\} \to \{0, 1, \ldots, 255\}$ from Example 2.51 is not onto. Recall that the only output values achieved were $\{26, 78, 130, 182, 234\}$. For example, then, there is no value of x for which $quantize(x) = 42$. Thus 42 is not in the range of *quantize*, and therefore this function is not onto.

Here is a collection of a few more examples, where we'll try to construct onto and non-onto functions meeting a certain description:

Example 2.57: Sample onto/non-onto functions.
Let $A = \{0, 1, 2\}$ and $B = \{3, 4\}$. Give an example of a function that satisfies the following descriptions; if there's no such function, explain why it's impossible.

1. an onto function $f : A \to B$.
2. a function $g : A \to B$ that is *not* onto.
3. an onto function $h : B \to A$.

Solution. The first two are possible, but the third is not:

1. Define $f(0) = 3$, $f(1) = 4$, and $f(2) = 4$.
2. Define $g(0) = 3$, $g(1) = 3$, and $g(2) = 3$.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

3 Impossible! A function h whose domain is $\{3,4\}$ only has two output values, namely $h(3)$ and $h(4)$. For a function whose codomain is $\{0,1,2\}$ to be onto, we need three different output values to be achieved. These two conditions cannot be simultaneously satisfied, so there is no onto function from B to A.

Figure 2.52 illustrates the two functions from Example 2.57; the fact that f is onto and g is not onto is immediately visible.

One-to-one functions

An onto function $f : A \to B$ guarantees that every element $b \in B$ is "hit at least once" by f—that is, that $b = f(a)$ for at least one $a \in A$. A *one-to-one function* $f : A \to B$ guarantees that every element $b \in B$ is "hit *at most once*" by f:

Definition 2.52: One-to-one functions.
A function $f : A \to B$ is called *one-to-one* if, for any $b \in B$, there is at most one $a \in A$ such that $f(a) = b$. (Alternatively, we could say that $f : A \to B$ is one-to-one if, for any $a_1 \in A$ and $a_2 \in A$ with $a_1 \neq a_2$, we have that $f(a_1) \neq f(a_2)$.) A one-to-one function is also sometimes called an *injective* function.

(Terminologically, a one-to-one function sits in contrast to a *many-to-one* function, in which many different input values map to the same output value. Thinking about what a many-to-one function would mean may help to make the name "one-to-one" more intuitive.)

Taking it further: One of the many places that functions are used in computer science is in designing the data structure known as a *hash table*, discussed on p. 2-85. The idea is that we will store a piece of data called x in a location $h(x)$, for some function h called a *hash function*. We want to choose h to ensure that this function is "not-too-many-to-one" so that no location has to store too much information.

Let's look at two of our previous functions, *double* and *quantize*, as an example:

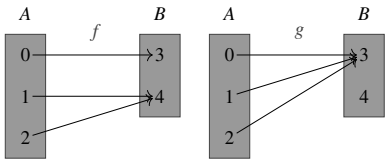


Figure 2.52 An onto function $f : \{0,1,2\} \to \{3,4\}$ and a function $g : \{0,1,2\} \to \{3,4\}$ that is not onto. (For an onto function from A to B, every element of B has at least one incoming arrow.)

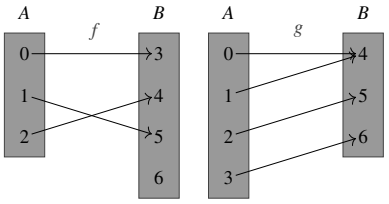


Figure 2.53 A one-to-one function f and a non–one-to-one function g. (For a one-to-one function from A to B, no element of B has more than one incoming arrow.)

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Example 2.58: A one-to-one function and a non–one-to-one function.
The function *double* : $\{0, 1, \ldots, 7\} \to \mathbb{Z}$, defined in Example 2.50, is one-to-one. The table of outputs for the function (see Figure 2.48) shows that no number appears more than once in the second column. Because every element of the codomain is "hit" by *double* at most once, the function is one-to-one. (Note *double* is not onto, because there are elements of the codomain that are "hit" zero times—but it is one-to-one, because no element of the codomain is hit twice.)

On the other hand, the function *quantize* : $\{0, 1, \ldots, 255\} \to \{0, 1, \ldots, 255\}$ from Example 2.51 is not one-to-one. Recall that $quantize(42) = 26$ and $quantize(17) = 26$. Thus 26 is the output for two or more distinct inputs, and therefore this function is not one-to-one.

As with the definition of onto, it may be easier to think about one-to-one functions using our visual two-column representation: a function f is one-to-one if *there's at most one arrow pointing at every element in the second column*. Figure 2.53 shows two small examples using this visual perspective: the function f in is one-to-one because no element of B has multiple incoming arrows, but the function g is not one-to-one, because $4 \in B$ has two incoming arrows.

One-to-one and onto functions

Let $f : A \to B$ be a function. We can restate the definitions of f being onto or one-to-one as follows:

- f is *onto* if, for every $b \in B$, we have $|\{a \in A : f(a) = b\}| \geq 1$.
- f is *one-to-one* if, for every $b \in B$, we have $|\{a \in A : f(a) = b\}| \leq 1$.

Therefore a function $f : A \to B$ that is *both* one-to-one *and* onto guarantees that $\{a \in A : f(a) = b\}$ has cardinality *equal to* 1—that is, for any $b \in B$, there is *exactly* one element $a \in A$ so that $f(a) = b$. (There is at most one such a because f is one-to-one, and at least one such a because f is onto.) A function with both of these properties is called a *bijection*:

Definition 2.53: Bijection.
A function $f : A \to B$ is called a *bijection* if f is one-to-one and onto. In other words, f is a bijection if $|\{a \in A : f(a) = b\}| = 1$ for every $b \in B$.

Here are two examples of bijections:

Example 2.59: Two bijections.
The function *not* : $\{\text{True}, \text{False}\} \to \{\text{True}, \text{False}\}$ from Example 2.48 is a bijection. There's exactly one input value whose output is True, namely False; and there's exactly one input value whose output is False, namely True.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Similarly, the function $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x - 1$ is also a bijection. For every $b \in \mathbb{R}$, there is exactly one a such that $f(a) = b$: specifically, the value $a = b + 1$.

If $f : A \to B$ is a bijection, then every input in A is assigned by f to a unique value in B. This fact means that we can define a new function, denoted f^{-1}, that reverses this assignment—given $b \in B$, the function $f^{-1}(b)$ identifies the $a \in A$ to which b was assigned by f. This function f^{-1} called the *inverse* of f:

Definition 2.54: Function inverses.
Let f be a bijection. Then $f^{-1} : B \to A$ is a function called the *inverse* of f, where $f^{-1}(b) = a$ whenever $f(a) = b$.

Here is an example of finding inverses of a few functions:

Example 2.60: Three inverses.
What is the inverse of each of the following functions?

1. $f : \mathbb{R} \to \mathbb{R}$, where $f(x) = \frac{x}{2}$.
2. $square : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$, where $square(x) = x^2$.
3. $not : \{\text{True}, \text{False}\} \to \{\text{True}, \text{False}\}$.

Solution. We can find the function f^{-1}, the inverse of f, by solving the equation $y = \frac{x}{2}$ for x. We see that $2y = x$. Thus the function $f^{-1} : \mathbb{R} \to \mathbb{R}$ is given by $f^{-1}(y) = 2y$. For any real number $x \in \mathbb{R}$, we have that $f(x) = \frac{x}{2}$ and $f^{-1}(\frac{x}{2}) = x$. (For example, $f(3) = 1.5$ and $f^{-1}(1.5) = 3$.)

The inverse of *square* is the function $square^{-1}(y) = \sqrt{y}$. (Note that $square : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ *is* a bijection—otherwise this problem wouldn't be solvable!—because the domain and the codomain are both the equal to the set of nonnegative real numbers. For example, $3^2 = 9$ and $(-3)^2 = 9$; if we had allowed both negative and positive inputs, then *square* would not have been one-to-one. And there's no $x \in \mathbb{R}$ such that $x^2 = -9$; if we had allowed negative outputs, then *square* would not have been onto.)

Finally, for *not*, note that $not(not(\text{True})) = not(\text{False}) = \text{True}$ and $not(not(\text{False})) = not(\text{True}) = \text{False}$. Thus the inverse of the function *not* is the function *not* itself!

If $f : A \to B$ is a bijection, then, for any $a \in A$, observe that applying f^{-1} to $f(a)$ gives a back as output: that is, $f^{-1}(f(a)) = a$. In other words, $f^{-1} \circ f$ is the *identity function*, defined by $id : A \to A$ where $id(a) = a$.

In our visualization, a bijection $f : A \to B$ has exactly one arrow coming into every element in B, and by definition it also has exactly one arrow leaving every element in A. The inverse of f is precisely the function that results from reversing the direction of each arrow. (The fact that every right-hand column element has exactly one incoming arrow under f is precisely what guarantees that reversing the direction of each arrow still results in the arrow diagram of a *function*.)

Figure 2.54 shows an example of a bijection and its inverse illustrated in this manner. This picture-based approach should help to illustrate why a function that is not onto or that is not one-to-one fails to have an

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Figure 2.54 A bijection $f : \{0, 1, 2, 3\} \to \{4, 5, 6, 7\}$ and its inverse $f^{-1} : \{4, 5, 6, 7\} \to \{0, 1, 2, 3\}$.

inverse. If $f : A \to B$ is not onto, then there exists some element $b^* \in B$ that's never the value of f, so $f^{-1}(b^*)$ would be undefined. On the other hand, if f is not one-to-one, then there exists $b^\dagger$ such that $f(a) = b^\dagger$ and $f(a') = b^\dagger$ for $a \neq a'$; thus $f^{-1}(b^\dagger)$ would have to be *both* a and a', which is forbidden by the definition of a function.

2.5.3 Polynomials

We'll turn now to *polynomials*, a special type of function whose input and output are both real numbers, and where $f(x)$ is the sum of powers of x:

Definition 2.55: Polynomial.
A *polynomial* is a function $f : \mathbb{R} \to \mathbb{R}$ of the form $f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k$ where each $a_i \in \mathbb{R}$ and $a_k \neq 0$, for some $k \in \mathbb{Z}^{\geq 0}$. (More compactly, we can write this function as $f(x) = \sum_{i=0}^{k} a_i x^i$.)

The real numbers $a_0, a_1, \ldots, a_k$ are called the *coefficients* of the polynomial, and the values $a_0, a_1 x, a_2 x^2, \ldots, a_k x^k$ being added together are called the *terms* of the polynomial.

A few examples of polynomials: $f(x) = 7x$, $g(x) = x^{202} - 201x^{111}$, and $h(x) = x^2 - 2$. The function h is graphed in Figure 2.55—in other words, for every $x \in \mathbb{R}$, the point $\langle x, h(x) \rangle$ is drawn.

> Annoyingly, the word "graph" among computer scientists is completely ambiguous. It can mean a graph in the sense of a plot of a mathematical function (as in Figure 2.55). But it can also mean a network (like a social network, or a disease propagation network, or an ethernet network) of interacting entities. We'll spend a huge amount of time on the latter kind of graph—they are the subject of all of Chapter 11.

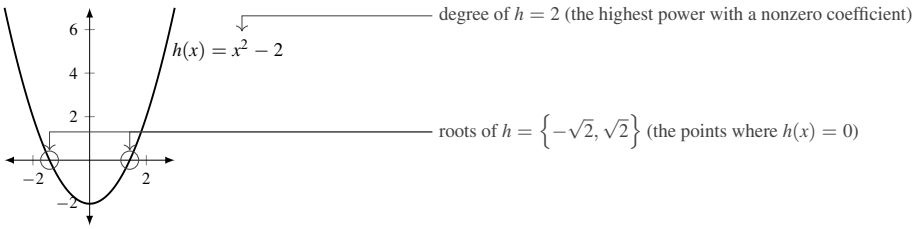


Figure 2.55 A graph of the polynomial $h(x) = x^2 - 2$.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

There are two additional definitions related to polynomials that will be useful. The first is the *degree* of the polynomial $p(x)$, which is the highest power of x in p's terms:

Definition 2.56: Degree.
The *degree* of a polynomial $f(x) = \sum_{i=0}^{k} a_i x^i$ is the largest index i such that $a_i \neq 0$—that is, the highest power of x with a nonzero coefficient.

Here are a few examples:

Example 2.61: Some degrees.
For the polynomials $f(x) = x + x^3$ and $g(x) = x^9$, the degree of f is 3 and the degree of g is 9. For the polynomial $p(x)$ with $a_0 = 1$, $a_1 = 3$, and $a_2 = 0$, the degree of p is 1, because $p(x) = 1 + 3x + 0x^2 = 1 + 3x$.

Some more examples of polynomials with small degrees are shown in Figure 2.56.

The second useful notion about a polynomial $p(x)$ is a *root*, which is a value of x where the graph of p crosses the x axis:

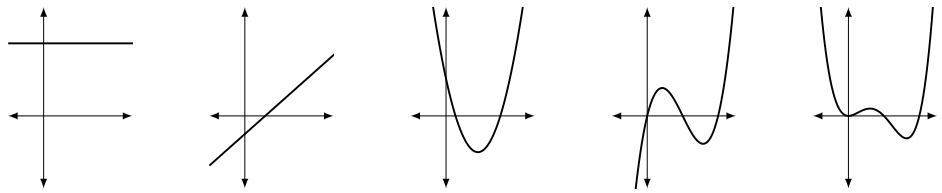
Definition 2.57: Roots.
The *roots* of a polynomial $p(x)$ are the values in the set $\{x \in \mathbb{R} : p(x) = 0\}$.

Here are a couple of examples:

Example 2.62: Some roots.
The roots of the polynomial $f(x) = x + x^2$ are 0 and -1. For $g(x) = x^9$, the only root is 0.

A useful general theorem relates the number of different roots for a polynomial to its degree: a polynomial p with degree k has at most k different values of x for which $p(x) = 0$ (unless p is *always* equal to 0):

Theorem 2.58: (Nonzero) polynomials of degree k have at most k roots.
Let $p(x)$ be a polynomial of degree at most k. Then p has at most k roots *unless* $p(x)$ *is zero for every value* $x \in \mathbb{R}$.



(a) Degree 0. (b) Degree 1. (c) Degree 2. (d) Degree 3. (e) Degree 4.

Figure 2.56 Graphs of some polynomials of degree 0, 1, 2, 3, and 4.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

When $p(x)$ is zero for every value $x \in \mathbb{R}$, we sometimes write $p(x) \equiv 0$ and say that p is *identically zero*.

We won't give a formal proof of Theorem 2.58, but here's one way to convince yourself of the basic idea. Think about how many times a polynomial of degree k can "change direction" from increasing to decreasing or from decreasing to increasing. Observe that a polynomial p must change directions between any two roots. (Draw a picture!) A polynomial of degree 0 never changes direction, so it's either always zero or never zero. A polynomial $p(x)$ of degree $d \geq 1$ can change directions only at a point where its slope is precisely equal to zero—that is, a point x where the derivative p' of p satisfies $p'(x) = 0$. Using calculus, we can show that the derivative of a polynomial of degree $d \geq 1$ is a polynomial of degree $d - 1$. The idea of a *proof by mathematical induction* is to combine the above intuition to prove the theorem.

> **Taking it further:** Here's some more detailed intuition of how to prove Theorem 2.58 using a proof by mathematical induction; see Chapter 5 for much more detail on this form of proof. (This argument relies on calculus.)
>
> Think first about a degree-zero polynomial—that is, a constant function $p(x) = a$. The theorem is clear for this case: either $a = 0$ (in which case $p(x) \equiv 0$); or $a \neq 0$, in which case $p(x) \neq 0$ for any x. (See Figure 2.56a.)
>
> Now think about a degree-1 polynomial—that is, $p(x) = ax + b$ for $a \neq 0$. The derivative of p is a constant function—namely $p'(x) = a \neq 0$. Imagine what it would mean for p to have two roots: as we move from smaller x to larger x, at some point r we cross the x-axis, say from $p(r-\epsilon) < 0$ to $p(r+\epsilon) > 0$. (See Figure 2.56b.) In order to find another root larger than r, the function p would have to change from increasing to decreasing—in other words, there would have to be a point at which $p'(x) = 0$. But we just argued that a degree-zero polynomial like $p'(x)$ that is not identically zero is never zero. So we can't find another root.
>
> Now think about a degree-2 polynomial—that is, $p(x) = ax^2 + bx + c$ for $a \neq 0$. After a root, p will have to change direction to head back toward the x-axis. That is, between any two roots of p, there must be a point where the derivative of p is zero: that is, there is a root of the degree-one polynomial $p'(x) = 2ax + b$ between any two roots of p. But p' has at most one root, as we just argued, so p has at most two roots.
>
> And so forth! We can apply the same argument for degree 3, then degree 4, and so on, up to any degree k.

2.5.4 Algorithms

While functions are a valuable mathematical abstraction, computer scientists are fundamentally interested in *computing* things. So, in addition to the type of functions that we've discussed so far in this section, we will also often talk about mapping an input x to a corresponding output $f(x)$ in the way that a computer program would, by computing the value of $f(x)$ using an *algorithm*:

Definition 2.59: Algorithm.
An *algorithm* is step-by-step procedure to transform an input into an output.

In other words, an algorithm is function—but specified as a sequence of simple operations, of the type that could be written as a program in your favorite programming language; in fact, these step-by-step procedures are even *called* functions in many programming languages. (It's probably worth noting that it's unusual for a book like this one to introduce algorithms in the context of functions. But, because the point of an algorithm really *is* to transform inputs into outputs, it can be helpful to think of an algorithm as a description

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

a function f that specifies *how* to calculate the output $f(x)$ from a given input x, instead of simply describing *what* the value $f(x)$ is.)

We will write algorithms in *pseudocode*, rather than in any particular programming language. In other words, we will specify the steps of the algorithm in a style that is neither Python nor Java nor English, but something in between; it's written in a style that "looks" like a program, but is designed to communicate the steps to a human reader, rather than to a computer executing the code. We will aim to write pseudocode that can be interpreted straightforwardly by a reader who has used any modern programming language; we will always try to avoid getting bogged down in detailed syntax, and instead emphasize trying to communicate algorithms clearly. Translating the pseudocode for an algorithm into any programming language should be straightforward (though of course would take energy and concentration, and a willingness to correct the inevitable bugs in the code, to actually carry out successfully).

We will make use of the standard elements of any programming language in our pseudocode: conditionals ("if"), loops ("for" and "while"), function definitions and function calls (including recursive function calls), and functions returning values. We will use the symbol ":=" to denote assignment and the symbol "=" to denote equality testing, so that $x := 3$ sets the value of x to be 3, and $x = 3$ is True (if x is 3) or False (if x is not 3). We assume a basic familiarity with these programming constructs throughout the book.

> Our notation of := for assignment and = for equality testing is borrowed from the programming language Pascal. In a lot of other programming languages, like C and Java and Python, assignment is expressed using `=` and equality testing is expressed using `==`.

We will spend significant energy later in the book on proving algorithms correct (Chapters 4 and 5)—that is, showing that an algorithm computes the correct output for any given input—and on analyzing the efficiency of algorithms (Chapter 6). But Figure 2.57 shows one small example to get us started. This algorithm finds the index of the maximum element of a list. (More properly, this algorithm finds the index of the *first* maximum element.)

findMaxIndex(L):
Input: A list L with $n \geq 1$ elements $L[1], \ldots, L[n]$.
Output: An index i such that $L[i]$ is the maximum value in L.

```
1 maxIndex := 1
2 for i := 2 to n:
3     if L[i] > L[maxIndex] then
4         maxIndex := i
5 return maxIndex
```

Figure 2.57 An algorithm to find the index of the maximum element of a list.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Computer Science Connections

Hash Tables and Hash Functions

Consider the following scenario: we have a set S of elements that we must store, each of which is chosen from a *universe* U of all possible elements. We need to be able to answer the question "is x in S?" quickly. (We might also have data associated with each $x \in S$, and seek to find the associated data rather than just determining membership.) Furthermore, the set S might change over time, either by insertion of a new element or deletion of an existing element. How might we efficiently organize the data to support these operations?

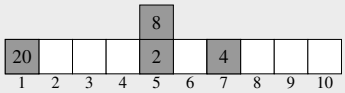
A *hash table*, one of the most frequently used data structures in computer science, is designed to store a set like S, as follows:

- we define a table $T[1 \ldots n]$.
- we choose a *hash function* $h : U \to \{1, \ldots, n\}$.
- each element $x \in S$ is stored in the cell $T[h(x)]$.

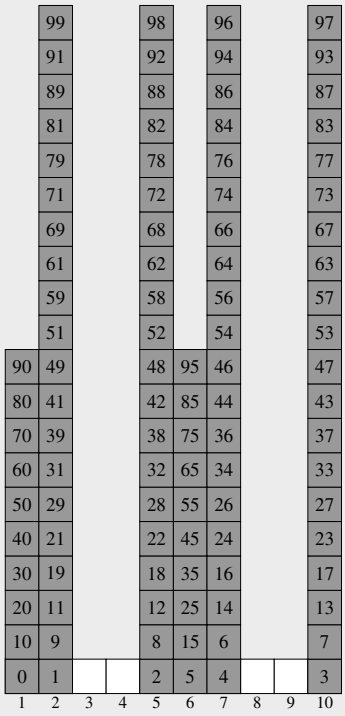
To make this idea practical, we'll make sure that n is a lot smaller than $|U|$. That means that h cannot be a one-to-one function, and thus we will need to deal with the possibility of *collisions*, when we try to store two different elements in the same cell. There are several different choices about how to handle collisions (and we'll explore several of the other strategies in Chapter 10), but, for simplicity, let's assume that we store them all in that cell, in a list. (This strategy for handling collisions is called *chaining*.) For example, see the hash function and hash table in Figure 2.58.

To insert a value x into the table, we merely need to compute $h(x)$ and place the value into the list in the cell $T[h(x)]$. Answering the question "is x stored in the table?" is similar; we compute $h(x)$ and look through whatever entries are stored in that list. As a result, the performance of this data structure is almost entirely dependent on how many collisions are generated—that is, how long the lists are in the cells of the table.

A "good" hash function $h : U \to \{1, \ldots, n\}$ is one that distributes the possible values of U as evenly as possible across the n different cells. The more evenly the function spreads out U across the table, the smaller the typical length of the list in a cell, and therefore the more efficiently the program would run. (Figure 2.58b says that its hash function is not a very good one—the number of elements per cell is extremely variable,



(a) A hash table filled with 4, 2, 8, and 20, using the hash function $h(x) = (x^2 \bmod 10) + 1$. These elements go into cells $h(4) = (16 \bmod 10) + 1 = 7$ and $h(2) = 5$ and $h(20) = 1$ and $h(8) = 5$.



(b) The table filled with $\{0, 1, \ldots, 99\}$. (Note that four cells contain zero elements, two cells contain 10 elements each, and four cells contain 20 elements each.)

Figure 2.58 A hash table, nearly empty and filled.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

which means that this hash function does a poor job of spreading out its inputs across the table.)

Programming languages like Python and Java have built-in implementations of hash tables, and they use some mildly complex iterative arithmetic operations in their hash functions. But designing a good hash function for whatever kind of data you end up storing can be the difference between a slow implementation and a blazingly fast one. Incidentally, there are at least two other concerns with efficiency: first, the hash function must be able to be computed quickly; otherwise the overhead of figuring out which cell to check is too much. There's also some cleverness in choosing the size of the table and in deciding when to *rehash* everything in the table into a bigger table if the lists get too long (on average).

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

EXERCISES

2.224 Consider the function $f : \{0, 1, \ldots 7\} \to \{0, 1, \ldots 7\}$ defined by $f(x) = (x^2 + 3) \bmod 8$. What is $f(3)$?

2.225 For the same function, what is $f(7)$?

2.226 For the same function, identify the values of x for which $f(x) = 3$.

2.227 Redefine f using a table.

2.228 Example 2.51 introduced a function compressing a grayscale image to use only five different shades of gray. (See Figure 2.59 for a reminder.) Using basic arithmetic notation (including $\lfloor\ \rfloor$ and/or $\lceil\ \rceil$ if appropriate), redefine *quantize* without using cases.

2.229 Let's generalize the quantization idea to a *two-argument* function, so that $quantize(n, k)$ takes both an input color $n \in \{0, 1, \ldots, 255\}$ and a number k of "quanta." (We insist that $1 \leq k \leq 256$.) In other words, k is the number of different (approximately) equally spaced output values, and the input color n is translated to the closest of these k values. What are the domain and range of $quantize(n, k)$?

2.230 Repeat Exercise 2.228 for $quantize(n, k)$. You should ensure that $quantize(n, 5)$ yields the function from Exercise 2.228. *(Hint: first determine how big a range of colors should be mapped to a particular quantum, rounding the size up. Then figure out which quantum the given input n corresponds to.)*

2.231 A function $f : A \to B$ is said to be c-to-1 if, for every output value $b \in B$, there are exactly c different values $a \in A$ such that $f(a) = b$. (These functions are useful in counting; see the Division Rule in Theorem 9.11.) For what values of k is it *possible* to define a c-to-1 (for some integer c) quantizing function that transforms $\{0, 1, \ldots, 255\}$ into a set of k quanta?

2.232 *(programming required.)* Implement quantization in a programming language of your choice. Specifically, implement $quantize(n, k)$, and apply it to every pixel of an input image. (You'll need to research an image-processing library to use in your program.)

Many of the pieces of basic numerical notation that we've introduced can be thought of as functions. For each of the following, state the domain *and* range *of the given function.*

2.233 $f(x) = |x|$

2.234 $f(x) = \lfloor x \rfloor$

2.235 $f(x) = 2^x$

2.236 $f(x) = \log_2 x$

2.237 $f(x) = x \bmod 2$

2.238 $f(x) = 2 \bmod x$

2.239 $f(x, y) = x \bmod y$

2.240 $f(x) = 2 \mid x$

2.241 $f(x) = \|x\|$

2.242 $f(\theta) = \langle \cos\theta, \sin\theta \rangle$

2.243 Let $T = \{1, \ldots, 12\} \times \{0, 1, \ldots, 59\}$ denote the set of numbers that can be displayed on a digital clock in twelve-hour mode. Define a function $add : T \times \mathbb{Z}^{\geq 0} \to T$ so that $add(t, x)$ denotes the time that's x minutes later than t. Do so using only standard symbols from arithmetic.

Define the functions $f(x) = x \bmod 10$, $g(x) = x + 3$, and $h(x) = 2x$. Rewrite the following functions using a single algebraic expression. (For example, the function $g \circ g$ is given by the definition $(g \circ g)(x) = g(g(x)) = x + 6$.)

$$quantize(n) = \begin{cases} 26 & \text{if } 0 \leq n \leq 51 \\ 78 & \text{if } 52 \leq n \leq 103 \\ 130 & \text{if } 104 \leq n \leq 155 \\ 182 & \text{if } 156 \leq n \leq 207 \\ 234 & \text{if } 208 \leq n \leq 255 \end{cases}$$

Note: the ranges of colors associated with each of these five "quanta" are only *approximately* equal because of issues of integrality. Here, the first four quanta correspond to 52 different colors; the last quantum corresponds to only $256 - 52 \cdot 4 = 48$ different colors.)

Figure 2.59 The function from Example 2.51.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.244 $f \circ f$

2.245 $h \circ h$

2.246 $f \circ g$

2.247 $g \circ h$

2.248 $h \circ g$

2.249 $f \circ h$

2.250 $f \circ g \circ h$

2.251 Let $f(x) = 3x + 1$ and let $g(x) = 2x$. Identify a function h such that $g \circ h$ and f are identical.

2.252 For the same functions f and g, identify a function h such that $h \circ g$ and f are identical.

2.253 Define a function $f : \{0, 1, 2, 3\} \to \{0, 1, 2, 3\}$ as $f(x) = x$. Is f onto?

2.254 Now define $f : \{0, 1, 2, 3\} \to \{0, 1, 2, 3\}$ as $f(x) = x^2 \bmod 4$. Is f onto?

2.255 What about for $f : \{0, 1, 2, 3\} \to \{0, 1, 2, 3\}$ defined as $f(x) = x^2 - x \bmod 4$?

2.256 Define $f : \{0, 1, 2, 3\} \to \{0, 1, 2, 3\}$ by $f(0) = 3, f(1) = 2, f(2) = 1$, and $f(3) = 0$. Is f onto?

2.257 What about for $f : \{0, 1, 2, 3\} \to \{0, 1, 2, 3\}$ defined as $f(0) = 1, f(1) = 2, f(2) = 1$, and $f(3) = 2$—is f onto?

2.258 Thinking of $f(x) = x^2 \bmod 8$ as a function $f : \{0, 1, 2, 3\} \to \{0, 1, \ldots, 7\}$, is f one-to-one?

2.259 Thinking of $f(x) = x^3 \bmod 8$ as a function $f : \{0, 1, 2, 3\} \to \{0, 1, \ldots, 7\}$, is f one-to-one?

2.260 Thinking of $f(x) = (x^3 - x) \bmod 8$ as a function $f : \{0, 1, 2, 3\} \to \{0, 1, \ldots, 7\}$, is f one-to-one?

2.261 Thinking of $f(x) = (x^3 + 2x) \bmod 8$ as a function $f : \{0, 1, 2, 3\} \to \{0, 1, \ldots, 7\}$, is f one-to-one?

2.262 Define $f(0) = 3, f(1) = 1, f(2) = 4$, and $f(3) = 1$. For this function $f : \{0, 1, 2, 3\} \to \{0, 1, \ldots, 7\}$, is f one-to-one?

A heap *is a data structure that is used to represent a collection of items, each of which has an associated priority. (See p. 5-38.) A heap can be represented as a complete binary tree—a binary tree with no "holes" as you read in left-to-right, top-to-bottom order—but a heap can also be stored more efficiently as an array, in which the elements are stored in that same left-to-right and top-to-bottom order. See Figure 2.60.*

2.263 For a heap stored as an array $A[1 \ldots n]$, state the domain and range of the function **parent**. Is **parent** one-to-one?

2.264 State the domain and range of **left** and **right** for the heap as stored in $A[1 \ldots n]$. Are **left** and **right** one-to-one?

2.265 Give both a mathematical description *and* an English-language description of the meaning of the function **parent** $\circ$ **left**. Assume that A is infinite (that is, don't worry about encountering an i such that **left**(i) or **right**(i) is undefined).

2.266 Repeat for **parent** $\circ$ **right**. (Continue to think of A as infinite.)

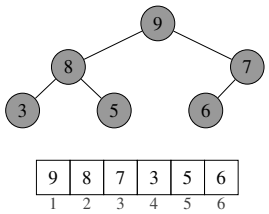
2.267 Repeat for **left** $\circ$ **parent**. (Continue to think of A as infinite.)

2.268 Repeat for **right** $\circ$ **parent**. (Continue to think of A as infinite.)

2.269 Consider $f : \mathbb{R} \to \mathbb{R}$, where $f(x) = 3x + 1$. What is the inverse of f?

2.270 Consider $g : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$, where $g(x) = x^3$. What is the inverse of g?

2.271 Consider $h : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 1}$, where $h(x) = 3^x$. What is the inverse of h?



To store a heap in an array, we define three functions that allow us to compute the index of the *parent* of a node; the index of the *left* child of a node; and the index of the *right* child of a node. Here are the functions: for an index i into the array,

parent$(i) := \lfloor \frac{i}{2} \rfloor$ **left**$(i) := 2i$ **right**$(i) := 2i + 1$.

For example, the parent of the node labeled 8 is labeled 9, the left child of the node labeled 8 is labeled 3, and the right child is labeled 5. And the node labeled 8 has index 2 in the array, and **parent**$(2) = 1$ (the index of the node labeled 9); **left**$(2) = 4$ (the index of the node labeled 3); and **right**$(2) = 5$ (the index of the node labeled 5).

Figure 2.60 A maximum heap, as a tree and as an array.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.272 Why doesn't the function $f : \{0, \ldots, 23\} \to \{0, \ldots, 11\}$ where $f(n) = n \bmod 12$ have an inverse?

2.273 Define $p(x) = 3x^3 + 2x^2 + x + 0$. What is the degree of p?

2.274 Define $p(x) = 9x^3$. What is the degree of p?

2.275 Define $p(x) = 4x^4 + x^2 - (2x)^2$. What is the degree of p?

2.276 If p and q are both polynomials with degree 7, what is the smallest and largest possible degree of $f(x) = p(x) + q(x)$?

2.277 If p and q are both polynomials with degree 7, what is the smallest and largest possible degree of $f(x) = p(x) \cdot q(x)$?

2.278 If p and q are both polynomials with degree 7, what is the smallest and largest possible degree of $f(x) = p(q(x))$?

2.279 Give an example of a polynomial p of degree 2 such that p has exactly 0 roots.

2.280 Give an example of a polynomial p of degree 2 such that p has exactly 1 root.

2.281 Give an example of a polynomial p of degree 2 such that p has exactly 2 roots.

2.282 The *median* of a list L of n numbers is the number in the "middle" of L in sorted order. Describe an algorithm to find the median of a list L. (Don't worry about efficiency.) You may find it useful to make use of the algorithm in Figure 2.57.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

2.6 Chapter at a Glance

Booleans, Numbers, and Arithmetic

A *Boolean value* is True or False. The *integers* $\mathbb{Z}$ are $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$. The *real* numbers $\mathbb{R}$ are the integers and all numbers in between. The *closed interval* $[a, b]$ consists of all real numbers x where $a \leq x \leq b$; the *open interval* (a, b) excludes a and b. The *rational* numbers $\mathbb{Q}$ are those numbers that can be represented as a/b for integers a and $b \neq 0$. Here is some useful notation involving numbers:

- *exponentiation*: b^k is $b \cdot b \cdot \ \cdots \ \cdot b$, where b is multiplied k times;
- *logarithms*: $\log_b x$ is the number y such that $b^y = x$;
- *absolute value*: $|x|$ is x for $x \geq 0$, and $|x| = -x$ for $x < 0$;
- *floor* and *ceiling*: $\lfloor x \rfloor$ is the largest integer $n \leq x$; $\lceil x \rceil$ is the smallest integer $n \geq x$;
- *modulus*: n mod k is the remainder when n is divided by k.

If $n \bmod d = 0$, then d is a *factor* of n or *evenly divides* n, written $d \mid n$. If $2 \mid n$ for a positive integer n, then n is *even* ("has even *parity*"); otherwise n is *odd*. An integer $n \geq 2$ is *prime* if it has no positive integer factors other than 1 and n; otherwise n is *composite*. (Note that 0 and 1 are neither prime nor composite.)

For a collection of numbers $x_1, x_2, \ldots, x_n$, their sum $x_1 + x_2 + \cdots + x_n$ is written formally as $\sum_{i=1}^{n} x_i$, and their product $x_1 \cdot x_2 \cdot \ \cdots \ \cdot x_n$ is written $\prod_{i=1}^{n} x_i$.

Sets: Unordered Collections

A *set* is an unordered collection of objects called *elements*. A set can be specified by listing its elements inside braces, as $\{x_1, x_2, \ldots, x_n\}$. A set can also be denoted by $\{x : P(x)\}$, which contains all objects x such that $P(x)$ is true. The set of possible values x that are considered is the *universe* U, which is sometimes left implicit.

Standard sets include the *empty set* $\{\}$ (also written $\varnothing$), which contains no elements; the *integers* $\mathbb{Z}$; the *real numbers* $\mathbb{R}$; and the *booleans* $\{\text{True}, \text{False}\}$. We write $\mathbb{Z}^{\geq 0} = \{0, 1, 2, \ldots\}$ and $\mathbb{Z}^{<0} = \{-1, -2, \ldots\}$, etc. For a set A and an object x, the expression $x \in A$ ("x is in A") is true whenever x is in the set A. (So $y \in \{x : P(x)\}$ whenever $P(y) = \text{True}$, and $y \in \{x_1, x_2 \ldots, x_n\}$ whenever $x_i = y$ for some i.) The *cardinality* of a set A, written $|A|$, is the number of distinct elements in A.

Given two sets A and B, the *union* of A and B is $A \cup B = \{x : x \in A \text{ or } x \in B\}$. The *intersection* of A and B is $A \cap B = \{x : x \in A \text{ and } x \in B\}$. The *set difference* of A and B is $A - B = \{x : x \in A \text{ and } x \notin B\}$. The *complement* of a set A is $\sim A = U - A = \{x : x \in U \text{ and } x \notin A\}$, where U is the universe.

A *subset* of a set B is a set A such that every element of A is also an element of B; this relationship is denoted by $A \subseteq B$. If A is a subset of B, then B is a *superset* of A, written $B \supseteq A$. A *proper subset* of B is a set A that is a subset of B but $A \neq B$, written $A \subset B$. Such a set B is a *proper superset* of A, written $B \supset A$.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Two sets A and B are *disjoint* if $A \cap B = \varnothing$. A *partition* of a set S is a collection of sets $A_1, A_2, \ldots, A_k$, where $A_1 \cup A_2 \cup \cdots \cup A_k = S$ and, for any distinct i and j, the sets A_i and A_j are disjoint.

The *power set* of a set A, written $\mathscr{P}(A)$, is the set of all subsets of A.

Sequences, Vectors, and Matrices: Ordered Collections

A *sequence* (or *tuple*, *(ordered) pair*, *triple*, *quadruple*, …, *n-tuple*, …) is an ordered collection of objects called *components* or *entries*, written inside angle brackets. The set $A \times B = \{\langle a, b\rangle : a \in A \text{ and } b \in B\}$ is the *Cartesian product* of sets A and B; the set $A \times B$ contains all pairs where the first component comes from A and the second from B. For a set S and a number $n \geq 0$, the set S^n denotes the n-fold Cartesian product of S with itself: $S^n = S \times S \times \ldots \times S$, where S occurs n times in this product.

A *vector* (or *n-vector*) is an element of $\mathbb{R}^n$, for some positive integer $n \geq 2$. (An element of $\mathbb{R}^1 = \mathbb{R}$ is called a *scalar*.) A *bit vector* is an element of $\{0,1\}^n$. Vectors are sometimes written in square brackets: $x = [x_1, x_2, \ldots, x_n]$. For a vector x, write x_i to denote the ith component of x. (But x_i is meaningless unless $i \in \{1, 2, \ldots, n\}$.) The *size* or *dimensionality* of $x \in \mathbb{R}^n$ is n.

For a vector $x \in \mathbb{R}^n$ and a real number $a \in \mathbb{R}$, the *scalar product* ax is a vector where $(ax)_i = ax_i$. For two vectors $x, y \in \mathbb{R}^n$, the sum of x and y is a vector $x + y$, where $(x + y)_i = x_i + y_i$. The *dot product* of two vectors $x, y \in \mathbb{R}^n$ is $x \bullet y = \sum_{i=1}^{n} x_i y_i$. Both $x + y$ and $x \bullet y$ are meaningless unless x and y have the same dimensionality.

An *n-by-m matrix* M is an element of $(\mathbb{R}^n)^m$, which is also sometimes written $\mathbb{R}^{n \times m}$. Such a matrix M has n *rows* and m *columns*, and its entries are referenced with subscripts, first by row and then by column:

$$M = \begin{bmatrix} M_{1,1} & M_{1,2} & \ldots & M_{1,m} \\ M_{2,1} & M_{2,2} & \ldots & M_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ M_{n,1} & M_{n,2} & \ldots & M_{n,m} \end{bmatrix}$$

A matrix $M \in \mathbb{R}^{n \times m}$ is *square* if $n = m$. For a size n, the *identity matrix* is $I \in \mathbb{R}^{n \times n}$ has ones on the main diagonal (the entries $I_{i,i} = 1$) and zeros everywhere else.

Given a matrix $M \in \mathbb{R}^{n \times m}$ and a real number $a \in \mathbb{R}$, the matrix aM is specified by $(aM)_{i,j} = aM_{i,j}$. Given two matrices $M, M' \in \mathbb{R}^{n \times m}$, the matrix $M + M'$ is specified by $(M + M')_{i,j} = M_{i,j} + M'_{i,j}$. (The sum $M + M'$ is meaningless if M and M' have different dimensions.) The product of two matrices $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times p}$ is a matrix $AB \in \mathbb{R}^{n \times p}$ whose components are given by $(AB)_{i,j} = \sum_{k=1}^{m} A_{i,k} B_{k,j}$. (More compactly, $(AB)_{i,j} = A_{i,(1\ldots m)} \bullet B_{(1\ldots m),j}$.) If the number of rows in A is different from the number of columns in B then AB is meaningless. The *inverse* of M is a matrix M^{-1} such that $MM^{-1} = I$ (if any such matrix M^{-1} exists).

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Functions

A *function* $f : A \to B$ maps every element $a \in A$ to some element $f(a) \in B$. The *domain* of f is A and the *codomain* is B. The *image* or *range* of f is $\{f(x) : x \in A\}$, the set of elements of the codomain "hit" by some element of A according to f.

The *composition* of a function $f : A \to B$ and a function $g : B \to C$ is written $g \circ f : A \to C$, where $(g \circ f)(x) = g(f(x))$. A function $f : A \to B$ is *one-to-one* or *injective* if $f(x) = f(y)$ implies that $x = y$. The function f is *onto* or *surjective* if the image is equal to the codomain. If $f : A \to B$ is one-to-one and onto, it is *bijective*. For a bijection $f : A \to B$, the function $f^{-1} : B \to A$ is the *inverse* of f, where $f^{-1}(b) = a$ when $f(a) = b$.

A *polynomial* $p : \mathbb{R} \to \mathbb{R}$ is a function $p(x) = a_0 + a_1 x + \cdots + a_k x^k$, where each $a_i \in \mathbb{R}$ is a *coefficient*. The *degree* of p is k. The *roots* of p are $\{x : p(x) = 0\}$. A polynomial of degree k that is not always zero has at most k different roots.

An *algorithm* is a step-by-step procedure that transforms an input into an output.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

Key Terms and Results

Key Terms

Booleans, Numbers, Arithmetic

- booleans, integers, reals, rationals
- open intervals, closed intervals
- absolute value, floor $\lfloor\ \rfloor$, ceiling $\lceil\ \rceil$
- exponentiation, logarithms
- modulus, remainder, divides
- even, odd, prime, parity
- summation $\sum$, product $\prod$
- nested summations, nested products

Sets

- set, element, membership, cardinality
- exhaustive enumeration
- set abstraction, universe
- the empty set $\varnothing = \{\}$
- Venn diagram
- complement $\sim$, union $\cup$, intersection $\cap$, set difference $-$
- (proper) subset, (proper) superset
- disjoint sets
- partitions
- power set

Sequences, Vectors, Matrices

- sequence, list, ordered pair, n-tuple
- Cartesian product
- vector, dot product
- matrix, identity matrix
- matrix multiplication
- matrix inverse

Functions

- domain, codomain, image/range
- function composition
- one-to-one, onto functions
- bijection, inverse

Key Results

Booleans, Numbers, and Arithmetic

1 The value of b^n is $b \cdot b \cdots b$, multiplied together n times. If $n < 0$, then $b^n = 1/(b^{-n})$. For rational exponents, $b^{1/m}$ is the number x such that $x^m = b$, and $b^{n/m} = (b^{1/m})^n$.

2 For a positive real number $b \neq 1$ and a real number $x > 0$, the quantity $\log_b x$ (the log base b of x) is the real number y such that $b^y = x$.

3 Consider integers $k > 0$ and n. Then $k \mid n$ ("k divides n") if $\frac{n}{k}$ is an integer—or, equivalently, if $n \bmod k = 0$.

4 As long as the terms being added remain unchanged, we can reindex a summation (for example, shifting the variable over which the sum is taken, or reversing the order of nested sums) without affecting the total value of the sum. The same is true for products.

Sets: Unordered Collections

1 A set can be specified using exhaustive enumeration (a list of its elements), or by abstraction (a condition describing when an object is an element of the set).

2 Two sets S and T are equal if every element of S is an element of T and every element of T is an element of S.

Sequences, Vectors, and Matrices

1 For vectors $x, y \in \mathbb{R}^n$, the *dot product* of x and y is $x \bullet y = \sum_{i=1}^{n} x_i y_i$.

2 The product AB of two matrices $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times p}$ is an n-by-p matrix $M \in \mathbb{R}^{n \times p}$ whose components are given by $M_{i,j} = \sum_{k=1}^{m} A_{i,k} B_{k,j}$.

Functions

1 A one-to-one and onto function $f : A \to B$ has an inverse function $f^{-1} : B \to A$, where $f(a) = b$ precisely when $f^{-1}(b) = a$.

2 A polynomial of degree k that is not always zero has at most k different roots.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.

This material will be published by Cambridge University Press as *Connecting Discrete Mathematics and Computer Science* by David Liben-Nowell, and an older edition of the material was published by John Wiley & Sons, Inc as *Discrete Mathematics for Computer Science*. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © David Liben-Nowell 2020–2021. This version was posted on October 5, 2021.