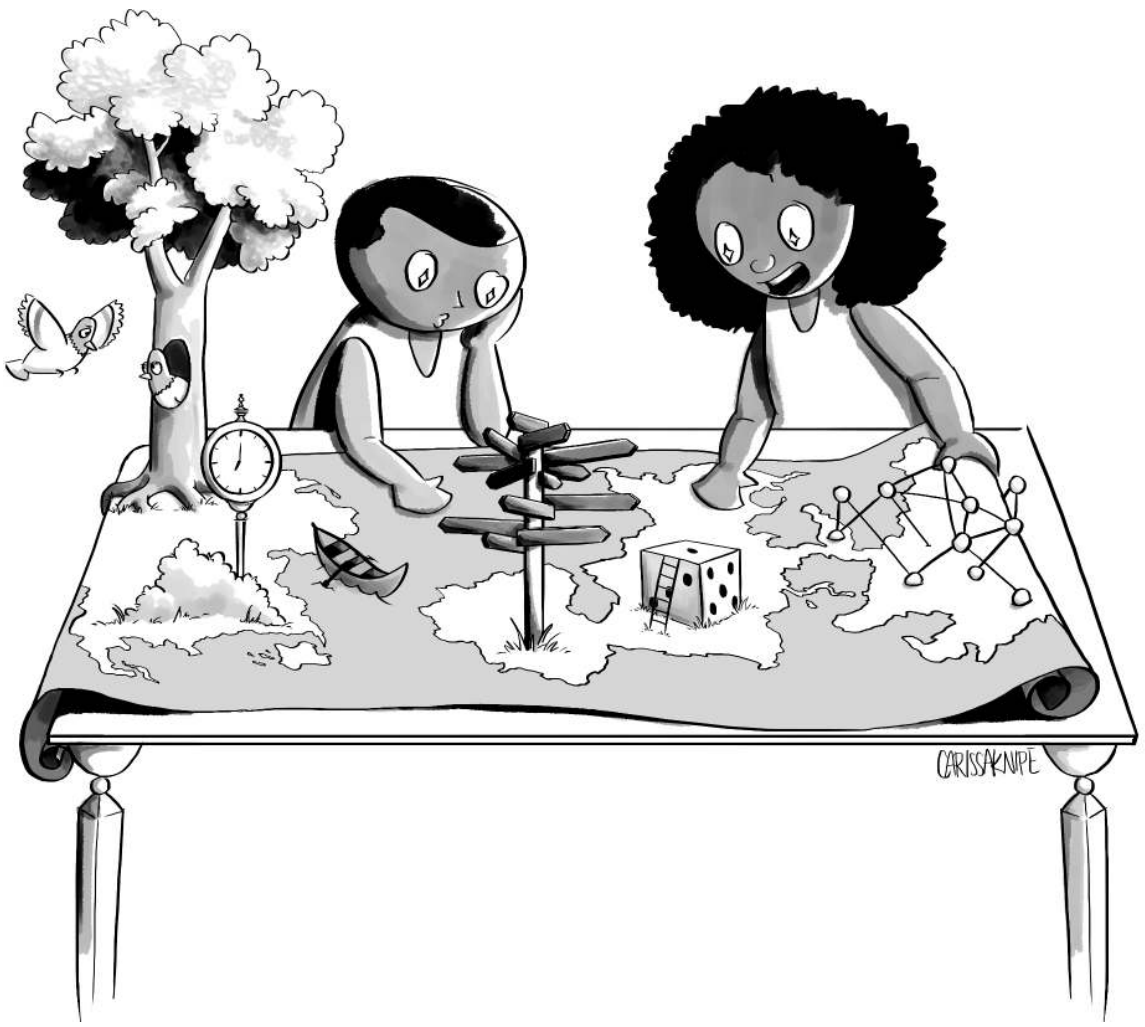
# 1 On the Point of this Book




*In which our heroes decide, possibly encouraged by a requirement for graduation, to set out to explore the world.*

## Why You Might Care

> Read much, but not many Books.
>
> Benjamin Franklin (1706–1790)
> *Poor Richard's Almanack* (1738)

This book is designed for an undergraduate student who has taken a computer science class or three. Most likely, you are a sophomore or junior prospective or current computer science major taking your first non-programming-based CS class. If you are a student in this position, you may be wondering why you're taking this class (or why you *have* to take this class!). Computer science students taking a class like this one sometimes don't see why this material has anything to do with computer science—particularly if you enjoy CS because you enjoy programming.

I want to be clear: programming is awesome! I get lost in code all the time—it would be better not count the number of hours that I spent writing the LaTeX code to draw the Fibonacci word fractal in Figure 5.27, or the divisibility relation in Figure 8.26, for example. (LaTeX, the tool used to typeset this book, is the standard typesetting package for computer scientists, and it's actually also a full-fledged, if somewhat bizarre, programming language.)

But there's more to CS than programming. In fact, many seemingly unrelated problems rely on the same sorts of abstract thinking. It's not at all obvious that an optimizing compiler (a program that translates source code in a programming language like C into something directly executable by a computer) would have anything important in common with a program to play chess perfectly. But, in fact, they're both tasks that are best understood using *logic* (Chapter 3) as a central component of any solution. Similarly, filtering spam out of your inbox ("given a message $m$, should $m$ be categorized as spam?") and doing speech recognition ("given an audio stream $s$ of a person speaking in English, what is the best 'transcript' reflecting the words spoken in $s$?") are both best understood using *probability* (Chapter 10).

And these, of course, are just examples; there are many, many ways in which we can gain insight and efficiency by thinking more abstractly about the commonalities of interesting and important CS problems. That is the goal of this book: to introduce the kind of mathematical, formal thinking that will allow you to understand ideas that are shared among disparate applications of computer science—and to make it easier for you to make your own connections, and to extend CS in even more new directions.

## How To Use This Book

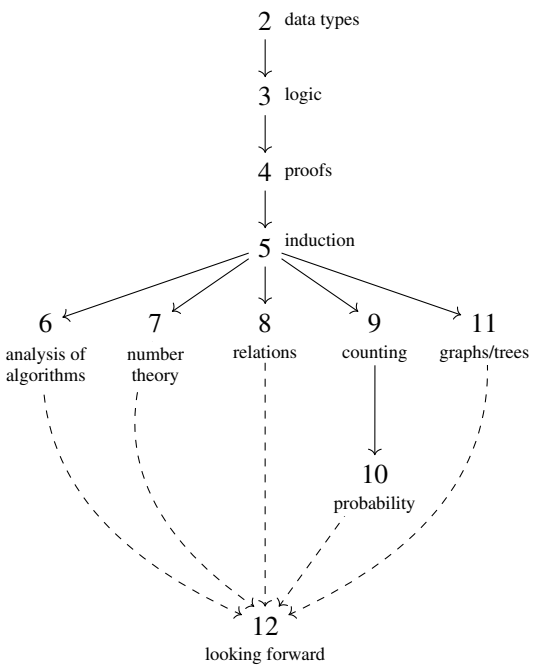
> "Tom's getting very profound," said Daisy, with an expression of unthoughtful sadness. "He reads deep books with long words in them."
>
> F. Scott Fitzgerald (1896–1940)
> *The Great Gatsby* (1925)

The brief version of the advice for how to use this book is: *it's your book; use it however you'd like.* (Will Shortz, the crossword editor of *The New York Times,* gives the analogous advice about crossword puzzles when he's asked whether Googling for an answer is cheating.) But my experience is that students do best when they read actively, with scrap paper close by; most people end up with a deeper understanding of a problem by trying to solve it themselves *first,* before they look at the solution.

I've assumed throughout that you're comfortable with programming in at least one language, including familiarity with recursion. It doesn't much matter which particular programming language you know; we'll

use features that are shared by almost all modern languages—things like conditionals, loops, functions, and recursion. You may or may not have already taken more than one programming-based CS course; many, but not all, institutions require data structures as a prerequisite for this material. There are times in the book when a data structures background may give you a deeper understanding (but the same is true in reverse if you study data structures after this material). There are similarly a handful of topics for which a rudimentary calculus background is valuable. But knowing/remembering calculus will be specifically useful only a handful of times in this book; the mathematical prerequisite for this material is really algebra and "mathematical maturity," which basically means having some degree of comfort with the idea of a mathematical definition and with the manipulation of a mathematical expression. (The few places where calculus is helpful are explicitly marked, and there's no point at which following a calculus-based discussion is necessary to understand the subsequent material.)

There are 10 technical chapters after this one in the book, followed by a brief concluding chapter (Chapter 12). Their dependencies are as shown at right. Aside from these dependencies, there are occasional references to other chapters, but these references are light, and intended to be enriching rather than essential. If you've skipped Chapter 6—many instructors will choose not cover this material, as it is frequently included in a course on algorithms instead of this one—then it will still be useful to have an informal sense of $O$, $\Omega$, and $\Theta$ notation in the context of the worst-case running time of an algorithm. (You might skim Section 6.1 and Section 6.6 before reading Chapters 7–11.)




When I teach the corresponding course at my institution, I typically include material from all chapters except Chapter 8 and Chapter 11, omitting various other sections and subsections as necessary to fit into the time constraints. (Our academic calendar is a bit compressed: we have three 10-week terms per year, with students taking three classes per term, and the courses end up being rather fast paced.) I also typically slightly reorder the later chapters, moving number theory (Chapter 7) to be the last topic of the course.

I've tried to include some helpful tips for problem solving throughout the book, along with a few warnings about common confusions and some notes on terminology and notation that may be helpful in keeping the words and symbols straight. There are also two kinds of extensions to the main material. The "Taking it Further" blocks give more technical details about the material under discussion—an alternate way of thinking about a definition, or a more technical derivation or extension that's relevant but not central, or a way that a concept is used in CS or a related field. You should read the "Taking it Further" blocks if—but only if!—you find them engaging. Each section also ends with one or more boxed-off "Computer Science Connections" that show how the core material in that section can be used in a wide variety of (interesting, I hope!) CS applications. No matter how interesting the core technical material may be, I think that it is what we can *do* with it that makes it worth studying.

## What This Book Is About

> I have no wish to talk nonsense.
>
> Charlotte Brontë (1816–1855)
> *Jane Eyre* (1847)

This book focuses on *discrete* mathematics, in which the entities of interest are distinct and separate. Discrete mathematics contrasts with *continuous* mathematics, as in calculus, which addresses infinitesimally small objects, which cannot be separated. We'll use summations rather than integrals, and we'll generally be thinking about things more like the integers ("$1, 2, 3, \ldots$") than like the real numbers ("all numbers between $\pi$ and 42").

> Be careful; there are two different words that are pronounced identically: *discrete,* adj., individually separate and distinct; and *discreet,* adj., careful and judicious in speech, especially to maintain privacy or avoid embarrassment. You wouldn't read a book about discreet mathematics; instead, someone who trusts you might quietly share it while making sure no one was eavesdropping.

Because this book is mostly focused on non-programming-based parts of computer science, in general the "output" that you produce when solving a problem will be something different from a program. Most typically, you will be asked to answer some question (quantitatively or qualitatively) and to justify that answer—that is, to *prove* your answer. (A *proof* is an ironclad, airtight argument that convinces its reader of your claim.) Remember that your task in solving a problem is to persuade your reader that your purported solution genuinely solves the problem. Above all, that means that your main task in writing is communication and persuasion.

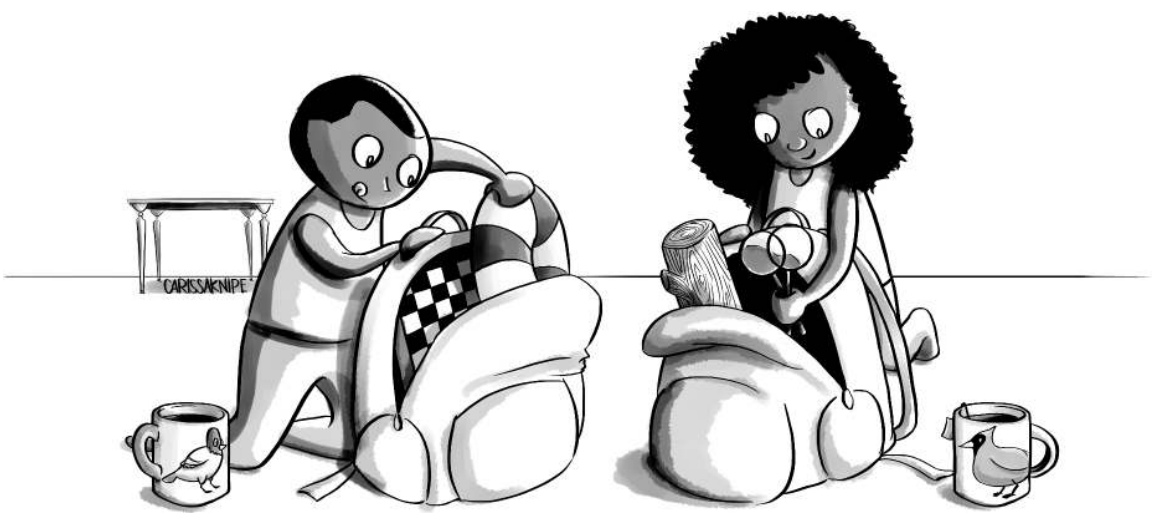
There are three very reasonable ways of thinking about this book.

- View #1 is that this book is about the mathematical foundations of computation. This book is designed to give you a firm foundation in mathematical concepts that are crucial to computer science: sets and sequences and functions, logic, proofs, probability, number theory, graphs, and so forth.
- View #2 is that this book is about practice. Essentially no particular example that we consider matters; what's crucial is for you to get exposure to and experience with formal reasoning. Learning specific facts about specific topics is less important than developing your ability to reason rigorously about formally defined structures.
- View #3 is that this book is about applications of computer science: error-correcting codes (how to represent data redundantly so that the original information is recoverable even in the face of data corruption), cryptography (how to communicate securely so that your information is understood by its intended recipient but not by anyone else), natural language processing (how to interpret the "meaning" of an English sentence spoken by a human using an automated dialogue system), and so forth. But, because solutions to these problems rely fundamentally on sets and counting and number theory and logic, we have to understand basic abstract structures in order to understand the solutions to these applied problems.

In the end, of course, all three views are right: I hope that this book will help to introduce some of the foundational technical concepts and techniques of theoretical computer science, and I hope that it will also help demonstrate that these theoretical approaches have relevance and value in work throughout computer science—in topics both theoretical and applied. And I hope that it will be at least a little bit of fun.

*Bon voyage!*

# 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.*

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


**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.)

## 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.

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.)

| | |
|---|---|
| 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.

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 $\pi$ and $\phi$ 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.)

**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. 21 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

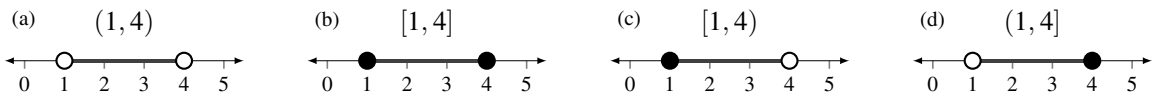



**Figure 2.3** Number lines representing real numbers between 1 and 4, with 1 included in the range in (b) and (c), and 4 included in the range in (b) and (d).

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.

### 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 $x$.

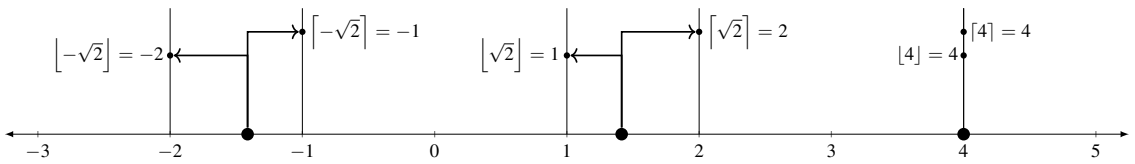



**Figure 2.4** The floor and ceiling of $-\sqrt{2}$, $\sqrt{2}$, and 4.

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.

### 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:

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

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: