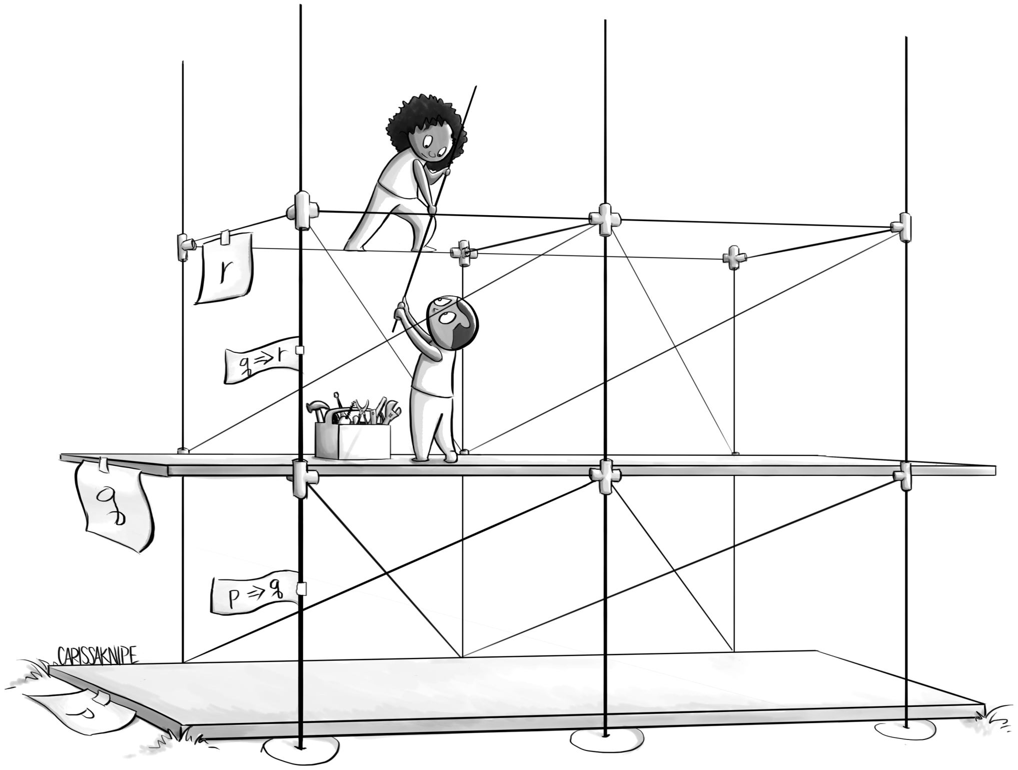
4 Proofs



In which our heroes build ironclad scaffolding to support their claims, thereby making them impervious to any perils they might encounter.

4.1 Why You Might Care

> *Madame, Madame, un bon mot ne prouve rien.*
> Madam, madam, a witty saying proves nothing.
>
> Voltaire (1694–1778)
> *Le Dîner Du Comte De Boulainvilliers* (1768)

A typical modern computational system is structured like a tower, with each layer's proper behavior contingent on the correctness of the one below. The web site that you use to send money to a friend relies on both a stack of networking protocols (HTTP relying on TCP, which is relying on IP, etc.), as well as a stack of applications on your computer or your phone (your browser relying on your operating system, which is relying on the hardware itself). A key theme in computer science is this idea of *abstraction:* that, so long as it's working properly, you can rely on the next layer in one of these towers (or a function in a large program, or . . .) without worrying about *how* exactly it works. You just have to trust *that* it works.

This chapter is about the mathematical version of a tower of trust. A *proof* is a convincing argument that establishes a particular claim as fact. That claim might be something explicitly computational: *Bubble Sort performs fewer comparisons than Merge Sort when the input array is already sorted,* for example. Or the claim might be noncomputational, at least superficially: a property of an operating system, the impossibility of designing a voting system with a certain set of properties. Generally speaking, our goal—in this chapter, in this book—is to establish new facts. And that's precisely the point of a proof: to derive a new fact from old facts, while persuading the reader that the new fact is, indeed, a fact. (For example, we can derive a new fact using Modus Ponens: if we know both p and $p \Rightarrow q$, then we can conclude that q is a fact, too.)

Why are proofs useful in computer science? First, proofs help prevent bugs. Whether or not they write down in full detail a proof that their code is correct, good software developers are always reasoning carefully about whether a function performs the task it's supposed to perform, or whether a particular optimization continues to meet the given specification. For a theoretical computer scientist, proofs are bread and butter: proofs of correctness for novel algorithms, or proofs of the hardness of solving a particular problem. For both theoretically and practically oriented computer scientists, a proof often yields great insight that can avoid a brute force solution, improve the efficiency of the code, or unearth some structural property of a problem that reveals that the problem doesn't even need to be solved in the first place.

We begin in Section 4.2 with an extended exploration of *error-correcting codes,* systems that allow for the reliable transmission and storage of information even in environments that corrupt data as it's stored/transmitted/received/retrieved. (For example, CDs and DVDs are susceptible to scratches, and deep-space satellites' transmissions are susceptible to radiation.) This section will merely scratch the surface of error-correcting codes, but it will serve as a nice introduction to error-correcting codes—and to proofs. Then, in Section 4.3, the technical meat of this chapter, we will develop a toolbox of techniques to use in proofs, and some strategies for choosing among them. We'll also illustrate these proof techniques with a hefty collection of examples about arithmetic. (In Section 4.5, we'll also catalogue some common types of mistakes in purported proofs, so that you can avoid them—and recognize bogus proofs when others attempt them.)

While the proof techniques themselves are the "point" of this chapter, in many cases the *fact* that we're proving is at least as interesting as the *proof of that fact*. Throughout our tour of proof techniques, we'll encounter a variety of examples of potentially interesting facts, in Section 4.4: about propositional logic, including the fact that we need only one logical connective ("nand") to express every proposition; about geometry (the Pythagorean Theorem); about prime numbers; and about *uncomputability* (there are problems that cannot be solved by any computer!).

4.2 An Extended Application with Proofs: Error-Correcting Codes

> Facts or opinions which are to pass through the hands of so many, to be misconceived by folly in one, and ignorance in another, can hardly have much truth left.
>
> Jane Austen (1775–1817)
> *Persuasion* (1818)

This section introduces *error-correcting codes,* a way of encoding data so that it can be transmitted correctly even in the face of (a limited number of) errors in transmission. These codes are used widely—for example, on DVDs/CDs and in file transfer protocols—and they're interesting to study on their own. But, despite appearances, they are not the point of this section! Rather, they're mostly an excuse to introduce a technical topic with some interesting (and nonobvious) results—and to persuade you of a few of those results. In other words, this section is really about proofs.

Error-detecting and error-correcting codes: the basic idea

Visa and Mastercard use 16-digit numbers for their credit and debit cards, but it turns out that there are only 10^{15} valid credit-card numbers: a number is valid only if a particular arithmetic calculation on the digits—more or less, adding up the digits and taking the result modulo 10—always turns out to be zero. (See Exercises 4.1–4.5 for details of the calculation.) Or, to describe this fact in another way: if you get a (mildly gullible) friend to read you any 15 digits of their credit-card number, you can figure out the 16th digit. Less creepily, this system means that there's an *error-detection* mechanism built into credit-card numbers: if any one digit in your number is mistranscribed, then a very simple algorithm can reject that incorrect card number as invalid (because the calculation above will yield an answer other than zero).

In this section, we'll explore encoding schemes with this sort of error-handling capability. Suppose that you have some binary data that you wish to transmit to a friend across an imperfect channel—that is, one that (due to cosmic rays, hardware failures, or whatever) occasionally mistransmits a 0 as a 1, or vice versa. (When we refer to an *error* in a bitstring x, what we mean is a "substitution error," where some single bit in x is flipped.) The fundamental idea will be to add redundancy to the transmitted data; if there is enough redundancy relative to the number of errors, then enough correct information will be transmitted to allow the receiver to reconstruct the original message. We'll explore both *error-detecting codes* that are able to recognize *whether* an error has occurred (at least, as long as there aren't too many errors) and *error-correcting codes* that can *fix* a small number of errors. To reiterate, though: while we're focusing on error-correcting and error-detecting codes in this section, the fundamental purpose of this section is to introduce proof techniques. Along the way, we'll see some interesting results about error-correcting codes, but the takeaway message is really about the methods that we'll use to prove those results.

Taking it further: Aside from credit-card numbers, other examples of error-detecting or error-correcting codes include *checksums* on a transferred file—we might break a large file we wish to transmit into 32-bit blocks, transmit those blocks individually, and transmit as a final 32-bit block the XOR of all previously transmitted blocks—as a way to check that the file was transmitted properly. Error-correcting codes are also used in storing data on media (hard disks and CDs/DVDs, for example) so that one can reconstruct stored data even in the face of hardware errors (or scratches on the disc). QR ("quick response") codes use a similar type of error correction. The idea of error detection appears in other contexts, too. Double-entry bookkeeping has been argued to be one of the most important inventions of the last 1000 years. UPC ("universal product code") bar codes on products in supermarkets use error checking similar to that in credit-card numbers. There are error-detection aspects in DNA. And "the buddy system" from elementary school field trips detects any one "deletion error" among the group (though two "deletions" may evade detection of the system).

4.2.1 A Formal Introduction

Imagine a *sender* who wishes to transmit a *message* $m \in \{0,1\}^k$ to a *receiver.* A *code* $\mathcal{C}$ is a subset of $\{0,1\}^n$, listing the set of legal *codewords*; each k-bit *message* m is encoded as an n-bit codeword $c \in \mathcal{C}$. The codeword is then transmitted to the receiver, but it may be corrupted during transmission. The recipient of the (possibly corrupted) n-bit string c' decodes c' into a new message $m' \in \{0,1\}^k$. The goal is that, so long as the corruption is limited, the decoded message is identical to the original message—in other words, that $m = m'$ as long as $c' \approx c$. (We'll make the meaning of "$\approx$" precise soon.) Figure 4.1 shows a schematic of the process. (For an error-*detecting* code, the receiver still receives the bitstring c', but determines *whether the originally transmitted codeword was corrupted* instead of *which codeword was originally transmitted,* as in an error-correcting code.)

Measuring the distance between bitstrings

Before we get to codes themselves, we need a way to quantify how similar or different two bitstrings are:

Definition 4.1: Hamming distance.
Let $x, y \in \{0,1\}^n$ be two n-bit strings. The *Hamming distance* between x and y, denoted by $\Delta(x,y)$, is the number of positions in which x and y differ. In other words,

$$\Delta(x,y) = \left| \left\{ i \in \{1,2,\ldots,n\} : x_i \neq y_i \right\} \right|.$$

(Hamming distance is undefined if x and y don't have the same length.)

(The Hamming distance is named after Richard Hamming (1915–1998), an American mathematician/computer scientist who was, among other things, the third winner of the Turing Award.)

For example, $\Delta(011,101) = 2$ because $\underline{01}1$ and $\underline{10}1$ differ in bit positions #1 and #2 (but match in bit position #3), and $\Delta(0011,0111) = 1$ because $0\underline{0}11$ and $0\underline{1}11$ differ in bit #2. Similarly, $\Delta(0000,1111) = 4$ because all four bits differ, and $\Delta(10101,10101) = 0$ because all five bits match.

The Hamming distance is a *metric,* which means that it satisfies the following properties, for all bitstrings $x, y, z \in \{0,1\}^n$:

- *"reflexivity":* $\Delta(x,y) = 0$ if and only if $x = y$;

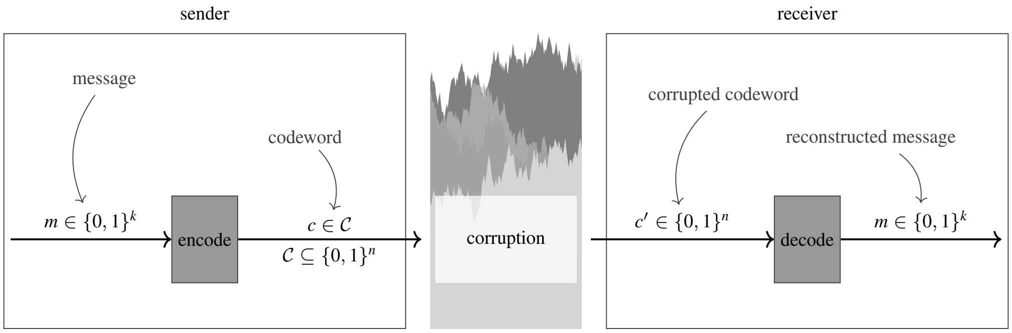


Figure 4.1 A schematic view of error-correcting codes. The goal is that, as long as there isn't *too* much corruption, the received message m' is identical to the sent message m.

- *"symmetry":* $\Delta(x, y) = \Delta(y, x)$; and
- *"the triangle inequality":* $\Delta(x, y) \leq \Delta(x, z) + \Delta(z, y)$. (See Figure 4.2.)

Informally, the fact that Δ is a metric means that it generally matches your intuitions about geometric (Euclidean) distance. (See Exercise 4.6.)

Error-detecting and error-correcting codes

Definition 4.2: Codes, messages, and codewords.
A *code* is a set $\mathcal{C} \subseteq \{0,1\}^n$, where $|\mathcal{C}| = 2^k$ for some integer $1 \leq k \leq n$. Any element of $\{0,1\}^k$ is called a *message,* and the elements of $\mathcal{C}$ are called *codewords.*

(It might seem a bit strange to require that the number of codewords in $\mathcal{C}$ be a precise power of two—but doing so is convenient, as it allows us to consider all k-bit strings as the set of possible messages, for $k = \log_2 |\mathcal{C}|$.) Here's an example of a code:

Example 4.1: A small code.
The set $\mathcal{C} = \{000000, 101010, 000111, 100001\}$ is a code. Because $|\mathcal{C}| = 4 = 2^2$, there are four messages, namely the four elements of $\{0,1\}^2 = \{00, 01, 10, 11\}$. And because $\mathcal{C} \subseteq \{0,1\}^6$, the codewords—the four elements of the set $\mathcal{C}$—are elements of $\{0,1\}^6$.

We can think of a code as being defined by a pair of operations:

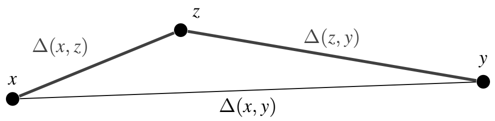


Figure 4.2 The triangle inequality. The distance from x to y isn't decreased by "stopping off" at z along the way.

encoding: given a message $m \in \{0,1\}^k$, which codeword in $\mathcal{C}$ should we transmit? (We'd break up a longer message into a sequence of k-bit message chunks.)

decoding: from a received (possibly corrupted) bitstring $c' \in \{0,1\}^n$, what message should we infer was sent? (Or, if we trying to *detect* errors rather than correct them: from a received bitstring $c' \in \{0,1\}^n$, do we say that an error occurred, or not?)

For the moment, we'll consider a generic (and slow) way of encoding and decoding. Given $\mathcal{C}$, we build a table mapping messages to codewords, by matching up the ith-largest message with the ith-largest codeword (with both the messages from $\{0,1\}^k$ and the codewords in $\mathcal{C}$ sorted in numerical order). We encode a message m by the codeword in m's row of the table. For error detection, we respond to a received bitstring c' by reporting "no error" if c' appears in the table, and reporting "error" if c' does not appear in the table. For error correction, we decode a received bitstring c' by identifying the codeword $c \in \mathcal{C}$ *that's closest to* c'*,* measured by Hamming distance; we decode c' as the message in row c of the table. (If there's a tie, we choose one of the tied-for-closest codewords arbitrarily.)

Example 4.2: Encoding and decoding with a small code.

For the code $\mathcal{C} = \{000000, 101010, 000111, 100001\}$ from Example 4.1, sorting the four codewords (and the messages from $\{0,1\}^2$) yields the table in Figure 4.3.

For example, we encode the message 10 as the codeword 100001.

If we receive the bitstring 111110, we report "error" because 111110 is not in $\mathcal{C}$.

For the received bitstring 111110, we have $\Delta(111110, \underline{00000}0) = 5$ and $\Delta(111110, \underline{000}11\underline{1}) = 4$ and $\Delta(111110, 1\underline{00001}) = 5$ and $\Delta(111110, 1\underline{0}1\underline{0}10) = 2$. The last of these distances is smallest, so we would decode 111110 as the message 11 (corresponding to codeword 101010).

The danger in error detection is that we're sent a codeword $c \in \mathcal{C}$ that's corrupted into a bitstring c', but we report "no error" because $c' \in \mathcal{C}$. (Note that we're never wrong when we report "error.") The danger in error correction is that we report another codeword $c'' \in \mathcal{C}$ because c' is closer to c'' than it is to c. (As we'll see soon, these dangers are really about Hamming distance *between codewords*: we might make a mistake if two codewords in $\mathcal{C}$ are too close together, relative to the number of errors.) Here are the precise definitions of error-detecting and error-correcting codes:

message	*codeword*
00	000000
01	000111
10	100001
11	101010

Figure 4.3 The message/codeword table for the code from Example 4.1.

Definition 4.3: Error-detecting and error-correcting codes.
Let $\mathcal{C} \subseteq \{0,1\}^n$ be a code, and let $\ell \geq 1$ be any integer.

We say that $\mathcal{C}$ can *detect* ℓ errors if, for any codeword $c \in \mathcal{C}$ and for any sequence of up to ℓ errors applied to c, we can correctly report "error" or "no error."

The code $\mathcal{C}$ can *correct* ℓ errors if, for any codeword $c \in \mathcal{C}$ and for any sequence of up to ℓ errors applied to c, we can correctly identify that c was the original codeword.

Here's an example, for our small example code:

Example 4.3: Error detection and correction in a small code.
Recall $\mathcal{C} = \{000000, 101010, 000111, 100001\}$ from Example 4.1. Figure 4.4 shows every bitstring $x \in \{0,1\}^6$, and the Hamming distance between x and each codeword in $\mathcal{C}$.

This code can detect one error, because the 24 "one error" bitstrings in Figure 4.5 are all different from the 4 "no errors" bitstrings; we can correctly report whether the bitstring in question is a codeword (no errors) or one of the 24 non-codewords (one error). (There are 24 single-bit errors that can happen to codewords in $\mathcal{C}$: there are 4 choices of codeword, and, for each, 6 different one-bit errors that can occur.) Or, to state this fact in a different way: the highlighted columns of Figure 4.4 corresponding to uncorrupted codewords are not within one error of any other codeword.

On the other hand, $\mathcal{C}$ *cannot* detect two errors. If we receive the bitstring 000000, we can't distinguish whether the original codeword was 000000 (and no errors occurred) or whether the original codeword was 100001 (and two errors occurred, in $\underline{0}0000\underline{0}$). (Receiving the bitstring 100001 creates the same problem.)

The code $\mathcal{C}$ also cannot *correct* even one error. Consider the bitstring 100000. We cannot distinguish (i) the original codeword was $\underline{0}00000$ (and one error occurred) from (ii) the original codeword was $10000\underline{1}$ (and one error occurred). Or, to state this fact differently: 100000 appears *twice* in the list of 24 "one error" bitstrings in Figure 4.5.

4.2.2 Distance and Rate

Our goal with error-correcting codes is to ensure that the decoded message m' is identical to the original message m, as long as there aren't too many errors in the transmission. At a high level, we will achieve this goal by ensuring that the codewords in our code are all "very different" from each other. If every pair of distinct codewords c_1 and c_2 are far apart (in Hamming distance), then the closest codeword c to the received transmission c' will correspond to the original message, even if "a few" errors occur. (We'll quantify "very" and "a few" soon.)

Intuitively, this desire suggests adding a lot of redundancy to our codewords, by making them more redundant. But we must balance this desire for robustness against another desire that pulls in the opposite direction: we'd like to transmit a small number of bits (so that the number of "wasted" non-data bits is

The bitstrings 000000, 000111, 100001, and 101010 corresponding to uncorrupted codewords are not within one error of any other codeword.

$c' =$	000000	000001	000010	000011	000100	000101	000110	000111	001000	001001	001010	001011	001100	001101	001110	001111	010000	010001	010010	010011	010100	010101	010110	010111	011000	011001	011010	011011	011100	011101	011110	011111
$\Delta(c', 000000)$	0	1	1	2	1	2	2	3	1	2	2	3	2	3	3	4	1	2	2	3	2	3	3	4	2	3	3	4	3	4	4	5
$\Delta(c', 000111)$	3	2	2	1	2	1	1	0	4	3	3	2	3	2	2	1	4	3	3	2	3	2	2	1	5	4	4	3	4	3	3	2
$\Delta(c', 100001)$	2	1	3	2	3	2	4	3	3	2	4	3	4	3	5	4	3	2	4	3	4	3	5	4	4	3	5	4	5	4	6	5
$\Delta(c', 101010)$	3	4	2	3	4	5	3	4	2	3	1	2	3	4	2	3	4	5	3	4	5	6	4	5	3	4	2	3	4	5	3	4

The bitstrings 000001 and 100000 are within a single error of two *different* codewords.

(continued)	100000	100001	100010	100011	100100	100101	100110	100111	101000	101001	101010	101011	101100	101101	101110	101111	110000	110001	110010	110011	110100	110101	110110	110111	111000	111001	111010	111011	111100	111101	111110	111111
$\Delta(c', 000000)$	1	2	2	3	2	3	3	4	2	3	3	4	3	4	4	5	2	3	3	4	3	4	4	5	3	4	4	5	4	5	5	6
$\Delta(c', 000111)$	4	3	3	2	3	2	2	1	5	4	4	3	4	3	3	2	5	4	4	3	4	3	3	2	6	5	5	4	5	4	4	3
$\Delta(c', 100001)$	1	0	2	1	2	1	3	2	2	1	3	2	3	2	4	3	2	1	3	2	3	2	4	3	3	2	4	3	4	3	5	4
$\Delta(c', 101010)$	2	3	1	2	3	4	2	3	1	2	0	1	2	3	1	2	3	4	2	3	4	5	3	4	2	3	1	2	3	4	2	3

Figure 4.4 The Hamming distance of every 6-bit string to all codewords from Example 4.1.

no errors	*one error*					
000000	100000	010000	001000	000100	000010	000001
101010	001010	111010	100010	101110	101000	101011
000111	100111	010111	001111	000011	000101	000110
100001	000001	110001	101001	100101	100011	100000

Figure 4.5 The 24 single-bit errors for codewords in $\mathcal{C} = \{000000, 101010, 000111, 100001\}$.

	000000	000111	100001	101010
000000	0	3	2	3
000111	3	0	3	4
100001	2	3	0	3
101010	3	4	3	0

Figure 4.6 The Hamming distance between codewords of $\mathcal{C}$ from Example 4.1.

small). There's a seeming trade-off between these two measures of the quality of a code: increasing error tolerance suggests making the codewords longer (so there's room for them to differ more); increasing efficiency suggests making the codewords shorter (so there are fewer wasted bits). Let's formally define both of these measures of code quality:

Definition 4.4: Minimum distance.
The *minimum distance* of a code $\mathcal{C}$ is the smallest Hamming distance between two distinct codewords of $\mathcal{C}$: that is, the minimum distance of $\mathcal{C}$ is $\min\{\Delta(x, y) : x, y \in \mathcal{C} \text{ and } x \neq y\}$.

Definition 4.5: Rate.
The *rate* of a code $\mathcal{C}$ is the ratio between message length and codeword length. That is, if $\mathcal{C}$ is a code where $|\mathcal{C}| = 2^k$ and $\mathcal{C} \subseteq \{0, 1\}^n$, then the rate of $\mathcal{C}$ is the ratio $\frac{k}{n}$.

(Quiz question: if we hadn't restricted the minimum in Definition 4.4 to be only over pairs such that $x \neq y$, what would the minimum distance have been?)

Let's compute the rate and minimum distance for our running example:

Example 4.4: Distance and rate in a small code.
Recall the code $\mathcal{C} = \{000000, 101010, 000111, 100001\}$ from Example 4.1.

The minimum distance of $\mathcal{C}$ is 2, because $\Delta(000000, 100001) = 2$. You can check Figure 4.4 (or see Figure 4.6) to see that no other pair of codewords is closer.

The rate of $\mathcal{C}$ is $\frac{2}{6}$, because $|\mathcal{C}| = 4 = |\{0,1\}^2|$, and the codewords have length 6.

Relating minimum distance and error detection/correction

We have now defined enough of the concepts that we can state a first nontrivial theorem, which characterizes the error-detecting and error-correcting capabilities of a code $\mathcal{C}$ in terms of the minimum distance of $\mathcal{C}$. Here is the statement:

Theorem 4.6: Relationship of minimum distance to detecting/correcting errors.
Let $t \geq 0$ be any integer. If the minimum distance of a code $\mathcal{C}$ is $2t+1$, then $\mathcal{C}$ can detect $2t$ errors and correct t errors.

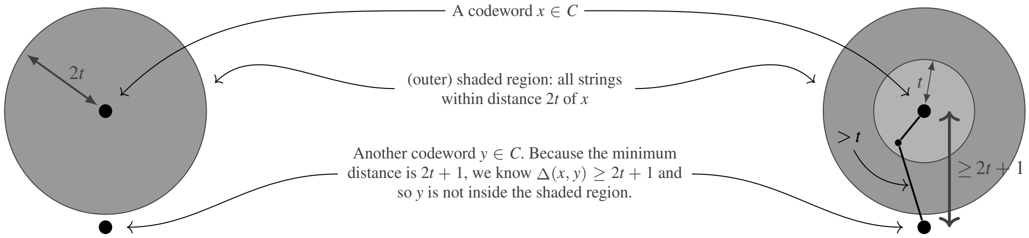
We're now going to try to prove Theorem 4.6—that is, we're going to try to generate a convincing argument that this statement is true. As with any statement that you try to prove, our first task is to *understand* what exactly the claim is saying. In this case, the theorem makes a statement about a generic nonnegative integer t and a generic code $\mathcal{C}$. Plugging in particular values for t can help make the claim clearer:

- If the minimum distance of a code $\mathcal{C}$ is 9—that is, the minimum distance is $2t+1$ for $t = 4$—then the claim says $\mathcal{C}$ can detect $2t = 2 \cdot 4 = 8$ errors and correct $t = 4$ errors.
- Suppose the minimum distance of $\mathcal{C}$ is 7. Writing $7 = 2t+1$ for $t = 3$, the claim states that $\mathcal{C}$ can detect 6 errors and correct 3 errors.
- If the minimum distance of $\mathcal{C}$ is 5, then $\mathcal{C}$ can detect 4 errors and correct 2 errors.
- If the minimum distance of $\mathcal{C}$ is 3, then $\mathcal{C}$ can detect 2 errors and correct 1 error.
- If the minimum distance of $\mathcal{C}$ is 1, then $\mathcal{C}$ can detect 0 errors and correct 0 errors.

Problem-solving tip: Step #1 in proving any claim is to understand what it's saying! (You can't persuade someone of something you don't understand.) One good way to start to do so is by plugging particular values into the statement.

Now that we have a better sense of what the theorem says, let's prove it:

Proof of Theorem 4.6. First we'll prove the error-detection condition, by arguing for the following claim: if a code $\mathcal{C}$ has minimum distance $2t+1$, then $\mathcal{C}$ can detect $2t$ errors. In other words, for an arbitrary codeword $x \in \mathcal{C}$ and an arbitrary received bitstring c' with $\Delta(x, c') \leq 2t$, our error-detection algorithm must be correct. (If $\Delta(x, c') > 2t$, then we're not obliged to correctly state that an error occurred, because


2t
A codeword $x \in C$
(outer) shaded region: all strings within distance $2t$ of x
Another codeword $y \in C$. Because the minimum distance is $2t + 1$, we know $\Delta(x, y) \geq 2t + 1$ and so y is not inside the shaded region.
t
$> t$
$\geq 2t + 1$

(a) No codewords are within distance $2t$ of each other, so no codeword except c itself can be inside the shaded region. This code can detect up to $2t$ errors.

(b) Any bitstring within distance t of the codeword x (that is, inside the smaller circle) must be more than t away from any other codeword.

Figure 4.7 Visualizations of the proof of Theorem 4.6. Suppose that $\mathcal{C}$ is a code whose minimum distance is $2t + 1$.

we're only arguing that we can detect $2t$ errors.) Recall that our error-detection algorithm reports "no error" if $c' \in \mathcal{C}$, and it reports "error" if $c' \notin \mathcal{C}$. (It may be helpful to think about this proof via Figure 4.7a.)

If $\Delta(x, c') = 0$, then no error occurred (because the received bitstring matches the transmitted one). In this case, our error-detection algorithm correctly reports "no error"—because $c' \in \mathcal{C}$ (because $c' = x$, and x was a codeword).

On the other hand, suppose $1 \leq \Delta(x, c') \leq 2t$—so an error occurred. The only way that we'd fail to detect the error is if the received bitstring c' is itself *another* codeword. But this situation can't happen, by the definition of minimum distance: for any codeword $x \in \mathcal{C}$, the set $\{c' : \Delta(x, c') \leq 2t\}$ *cannot* contain any elements of $\mathcal{C}$—otherwise the minimum distance of $\mathcal{C}$ would be $2t$ or smaller.

For the error-correction condition, suppose that $x \in \mathcal{C}$ is the transmitted codeword, and the received bitstring c' satisfies $\Delta(x, c') \leq t$. We have to persuade ourselves that x is the codeword closest to c' in Hamming distance. Let $y \in \mathcal{C} - \{x\}$ be any other codeword. We'll start from the triangle inequality, which tells us that $\Delta(x, y) \leq \Delta(x, c') + \Delta(c', y)$ and therefore that $\Delta(c', y) \geq \Delta(x, y) - \Delta(x, c')$. Now we'll prove that c' is closer to x than it is to y (see Figure 4.7b):

$$\begin{aligned}
\Delta(c', y) &\geq \Delta(x, y) - \Delta(x, c') && \textit{triangle inequality} \\
&\geq (2t + 1) - \Delta(x, c') && \Delta(x, y) \geq 2t + 1 \textit{ by definition of minimum distance} \\
&\geq (2t + 1) - t && \Delta(x, c') \leq t \textit{ by assumption} \\
&= t + 1 && \\
&> \Delta(x, c'). && \Delta(x, c') \leq t \textit{ by assumption, and } t < t + 1
\end{aligned}$$

This chain of inequalities shows c' is closer to x than it is to y. (Pedantically speaking, we're also relying on the *symmetry* of Hamming distance here: $\Delta(c', y) = \Delta(y, c')$. Again, see Exercise 4.6.) Because y was a generic codeword in $\mathcal{C} - \{x\}$, we can conclude that the original codeword x is the one closest to c'. □

It is customary to mark the end of one's proofs typographically; here, we're using a traditional box symbol: □. Other people may write "QED," short for the Latin phrase *quod erat demonstrandum* ("that which was to be demonstrated").

Before we move on from the theorem, let's reflect a little bit on the proof. (We'll concentrate on the error-correction half.) The most complicated part was unwinding the definitions in the theorem statement, in particular of "$\mathcal{C}$ has minimum distance $2t+1$" and "$\mathcal{C}$ can correct t errors." Eventually, we had to argue for the claim

For every $x \in \mathcal{C}$, $y \in \mathcal{C} - \{x\}$, and $c' \in \{0,1\}^n$: if $\Delta(x,c') \leq t$ then $\Delta(x,c') < \Delta(y,c')$.

(In other words, if c' is within t errors of x, then c' is closer to x than to any other codeword.) In the end, we were able to state the proof as a relatively simple sequence of inequalities.

After proving a theorem, it's also worth briefly reflecting on what the theorem does *not* say. Theorem 4.6, for example, only addresses codes with a minimum distance that's an odd number. You'll be asked to consider the error-correcting and error-detecting properties of a code $\mathcal{C}$ with an even minimum distance in Exercise 4.13. We also didn't show that we couldn't do better: Theorem 4.6 says that a code $\mathcal{C}$ with minimum distance $2t+1$ *can* correct t errors, but the theorem doesn't say that $\mathcal{C}$ *can't* correct $t+1$ (or more) errors. (But, in fact, it can't; see Exercise 4.12.)

Problem-solving tip: (Two separate tips related to the proof of Theorem 4.6.) First, when you're trying to prove a claim of the form $p \Rightarrow q$, try to massage p to look as much like q as possible. A good first step in doing so is to expand out the definitions of the premises, and then try to see what additional facts you can infer. Second, a useful general strategy: try to draw a picture to help you clarify/understand the problem that you're trying to solve.

Outline of the remainder of the section

Intuitively, rate and minimum distance are measures of the inherent tension in an error-correcting code. A code that has a higher distance means that we are more robust to errors: the farther apart codewords are, the more corruption can occur before we're unable to reconstruct the original message. A code that has a higher rate means that we are "wasting" fewer bits in providing this robustness: the larger the rate, the more our codeword contains "data" rather than "redundancy." In the rest of this section, we're going to prove several more theorems about error-correcting codes, exploring the trade-off between rate and distance. (But it's also worth noting that it's not a strict trade-off: sometimes we can improve in one measure without costing ourselves in the other!) And, as we go, we'll continue to try to reflect on the proof techniques that we use to establish these claims. Here are the three main theorems that we'll prove in the rest of this section:

Theorem 4.7: Good news.
There exists a code with 4-bit messages, minimum distance 3, and rate $\frac{1}{3}$.

Theorem 4.8: Better news.
There exists a code with 4-bit messages, minimum distance 3, and rate $\frac{4}{7}$.

Theorem 4.9: Bad news.
There does not exist a code with 4-bit messages, minimum distance 3, and any rate strictly better than $\frac{4}{7}$.

Notice that the first two of these results say that a code with particular properties exists, while the third result says that it's impossible to create a code with a different set of properties. Also notice that Theorem 4.8 is an improvement on Theorem 4.7: we've made the rate better (higher) without making the minimum distance worse. (When we can, we'll prove more general versions of these theorems, too, not limited to 4-bit messages with minimum distance 3.)

We'll prove Theorem 4.7 and Theorem 4.8 "by construction"—specifically, by building a code with the desired parameters. But, because Theorem 4.9 says that a code with certain properties fails to exist, we'll prove the result with a *proof by contradiction:* we temporarily assume that a code with 4-bit messages with distance 3 and rate strictly better than $\frac{4}{7}$ *does exist,* and reasoning logically from that assumption, we will derive a false statement (a contradiction). Because $p \Rightarrow \text{False} \equiv \neg p$, we can conclude that the assumption must have been false, and no such code can exist.

4.2.3 Repetition Codes

Intuitively, a good error-correcting code will amplify even a small difference between two different messages—a single differing bit—into a larger difference between the corresponding codewords. Perhaps the most obvious implementation of this idea is simply to encode a message m by repeating the bits of m several times. This idea gives rise to a simple error-correcting code, called the *repetition code.* (Actually, there are many different versions of the repetition code, depending on how many times we repeat m in the codeword.) Here's the basic definition:

Definition 4.10: Repetition code.
Let $\ell \in \mathbb{Z}^{\geq 2}$. The Repetition$_\ell$ *code* for k-bit messages consists of the codewords

$$\Big\{ \underbrace{m\, m\, \cdots\, m}_{\longleftarrow\ \ell \text{ times}\ \longrightarrow} : m \in \{0,1\}^k \Big\}.$$

That is, the codeword corresponding to a message $m \in \{0,1\}^k$ is the ℓ-fold repetition of the message m, so each codeword is an element of $\{0,1\}^{k\ell}$.

Here are some small examples of encoding/decoding using repetition codes:

Example 4.5: Some codewords for the repetition code.

The message 00111 corresponds to the codeword 00111 00111 00111 under the Repetition$_3$ code. If we encode the same message using the Repetition$_5$ code instead, then the resulting codeword is 00111 00111 00111 00111 00111.

For decoding, suppose that we receive the (possibly corrupted) bitstring $c' = 0010\ 0110\ 0010$ under the Repetition$_3$ code. We detect that an error occurred: c' is not a codeword, because the only codewords are 12-bit strings where all three 4-bit thirds are identical. For error correction: the closest codeword to c' is 0010 $0\underline{0}10$ 0010, so we decode c' as the message 0010.

The message/codeword table for the Repetition$_3$ code for 4-bit messages is shown in Figure 4.8. The distance and rate properties of the repetition code are relatively easy to see (from the definition or from this style of table):

Lemma 4.11: Distance and rate of the repetition code.
The Repetition$_\ell$ code has rate $\frac{1}{\ell}$ and minimum distance ℓ.

Proof. Recall that the rate of a code is the ratio $\frac{k}{n}$, where k is the length of the messages and n is the length of the codewords. A k-bit message is encoded as a $(k\ell)$-bit codeword (ℓ repetitions of k bits), and so the rate of this code is $\frac{k}{k\ell} = \frac{1}{\ell}$.

For the minimum distance, consider any two messages $m, m' \in \{0,1\}^k$ with $m' \neq m$. We know that m and m' must differ in at least one bit position, say bit position i. (Otherwise $m = m'$.) But if $m_i \neq m'_i$, then

$$\overset{\text{the codeword corresponding to } m}{\underset{\longleftarrow\ \ell \text{ times}\ \longrightarrow}{m\ m\ \cdots m}} \qquad \text{and} \qquad \overset{\text{the codeword corresponding to } m}{\underset{\longleftarrow\ \ell \text{ times}\ \longrightarrow}{m'\ m'\ \cdots m'}}$$

differ in at least one bit in each of the ℓ "blocks" (in the ith position of the block)—for a total of at least ℓ differences. Furthermore, the Repetition$_\ell$ encodings of the messages $000\cdots0$ and $100\cdots0$ differ in only ℓ places (the first bit of each "block"), so the minimum distance of the Repetition$_\ell$ code is exactly ℓ. □

Lemma 4.11 says that the Repetition$_3$ code on 4-bit messages (see Figure 4.8) has minimum distance 3 and rate $\frac{1}{3}$. Thus we've proven Theorem 4.7: we had to show that a code with these parameters exists, and we did so *by explicitly building such a code.* This proof is an example of a "proof by construction": to show that an object with a particular property exists, we've explicitly built an object with that property.

		(continued)		*(continued)*		*(continued)*	
message	*codeword*	*message*	*codeword*	*message*	*codeword*	*message*	*codeword*
0000	0000 0000 0000	0100	0100 0100 0100	1000	1000 1000 1000	1100	1100 1100 1100
0001	0001 0001 0001	0101	0101 0101 0101	1001	1001 1001 1001	1101	1101 1101 1101
0010	0010 0010 0010	0110	0110 0110 0110	1010	1010 1010 1010	1110	1110 1110 1110
0011	0011 0011 0011	0111	0111 0111 0111	1011	1011 1011 1011	1111	1111 1111 1111

Figure 4.8 The Repetition$_3$ code for 4-bit messages.

Problem-solving tip: When you're trying to prove a claim of the form $\exists x : P(x)$, try using a proof by construction first. (There are other ways to prove an existential claim, but this approach is great when it's possible.)

It's also worth noticing that we started out by describing a *generic* way to do encoding and decoding for error-correcting codes: after we build the table (like the one in Figure 4.8), we encode a message by finding the corresponding codeword in the table, and we decode a bitstring c' by looking at every codeword and identifying the one closest to c'. For particular codes, we may be able to give a *much* more efficient algorithm—and, indeed, we can do so for repetition codes. See Exercise 4.21.

4.2.4 Hamming Codes

When we're encoding 4-bit messages, the Repetition_3 code achieves minimum distance 3 with 12-bit codewords. (So its rate is $\frac{1}{3}$.) But it turns out that we can do better by defining another, cleverer code: the *Hamming code* maintains the same minimum distance, while improving the rate from $\frac{1}{3}$ to $\frac{4}{7}$. (The Hamming code, like the Hamming distance, is named after Richard Hamming, who invented this code in 1950 [53]. He was frustrated that programs he started running before going home on Friday nights often failed over the weekend because of a single bit error in memory.)

The basic idea of the Hamming code is to use an extra bit that, like the 16th digit of a credit card number, redundantly reports a value computed from the previous components of the message. Concretely, we could tack a single bit b onto the message m, where b reports the *parity* of m—that is, whether there are an even or odd number of bits set to 1 in m. If a single error occurs in the message, then b would be inconsistent with the message m, and we'd detect that error. (See Exercise 4.19.) In fact, for the Hamming code, we'll use several *different* parity bits, corresponding to different subsets of the bits of m.

Definition 4.12: Parity function.
The *parity* of a sequence $\langle a_1, a_2, \ldots, a_k \rangle$ of bits is denoted either $\mathit{parity}(a_1, a_2, \ldots, a_k)$ or $a_1 \oplus a_2 \oplus \cdots \oplus a_k$, and its value is

$$a_1 \oplus a_2 \oplus \cdots \oplus a_k = \begin{cases} 1 & \text{if there are an odd number of } i \text{ such that } a_i = 1 \\ 0 & \text{if there are an even number of } i \text{ such that } a_i = 1. \end{cases}$$

The parity of a and b can be denoted as $a \oplus b$ because if you think of $a, b \in \{0, 1\}$, where True $= 1$ and False $= 0$, then $\mathit{parity}(a, b)$ is the XOR of a and b. (And XOR was also denoted by $\oplus$.) (We could also have defined this function as $\mathit{parity}(a_1, \ldots, a_k) = [\sum_{i=1}^{k} a_i] \bmod 2$.)

Hamming's insight was that it's possible to achieve good error-correction properties by using three different parity bits, corresponding to different subsets of the message bits. It's easiest to think of this code in terms of its encoding algorithm:

		(continued)		*(continued)*		*(continued)*	
message	*codeword*	*message*	*codeword*	*message*	*codeword*	*message*	*codeword*
0000	0000000	0100	0100101	1000	1000011	1100	1100110
0001	0001111	0101	0101010	1001	1001100	1101	1101001
0010	0010110	0110	0110011	1010	1010101	1110	1110000
0011	0011001	0111	0111100	1011	1011010	1111	1111111

Figure 4.9 The Hamming code for 4-bit messages.

Definition 4.13: Hamming code.
The *Hamming code* is defined via the following encoding function. We will encode a 4-bit message $\langle a, b, c, d \rangle$ as the following 7-bit codeword:

$$\langle \underbrace{a,\quad b,\quad c,\quad d,}_{\longleftarrow \text{ message bits } \longrightarrow} \quad \underbrace{b \oplus c \oplus d,\quad a \oplus c \oplus d,\quad a \oplus b \oplus d}_{\longleftarrow \text{ parity bits } \longrightarrow} \rangle.$$

Applying this encoding to every 4-bit message yields the table of messages and their corresponding codewords shown in Figure 4.9. A few examples are shown in detail in Figure 4.10. (We could have described encoding for the Hamming code using matrix multiplication instead; see Exercises 2.221–2.223.)

Before we analyze the rate and minimum distance of the Hamming code, let's start to develop some intuition by looking at a few received (possibly corrupted) codewords. (We'll also begin to work out an efficient decoding algorithm as we go.)

Example 4.6: Some Hamming code decoding problems.
You receive the following (possibly corrupted) Hamming code codewords:

0000010 and 1000000 and 1011010 and 1110111.

Assuming that at most one error occurred in each transmission, what were the original messages?

Solution. We'll consider these (possibly corrupted) codewords in order:

0000010. We've received message bits 0000 and parity bits 010. Everything in the received codeword is consistent with the message being $m = 0000$, except for the second parity bit. So we infer that the second parity bit was corrupted, the transmitted codeword was 0000000, and the message was 0000.

Could there have been a one-bit error in message bits instead? No: these parity bits are consistent only with a message $\langle a, b, c, d \rangle$ with $a \neq b$ (because the first two received parity bits differ), and therefore

message	$a, b, c, d \longrightarrow a, b, c, d, (b \oplus c \oplus d), (a \oplus c \oplus d), (a \oplus b \oplus d)$	*codeword*
0000	$0, 0, 0, 0 \longrightarrow 0, 0, 0, 0, (0 \oplus 0 \oplus 0), (0 \oplus 0 \oplus 0), (0 \oplus 0 \oplus 0)$	0000000
1000	$1, 0, 0, 0 \longrightarrow 1, 0, 0, 0, (0 \oplus 0 \oplus 0), (1 \oplus 0 \oplus 0), (1 \oplus 0 \oplus 0)$	1000011
1110	$1, 1, 1, 0 \longrightarrow 1, 1, 1, 0, (1 \oplus 1 \oplus 0), (1 \oplus 1 \oplus 0), (1 \oplus 1 \oplus 0)$	1110000

Figure 4.10 A few sample Hamming code encodings.

with $d = 1$ (because $a \neq b$ implies that $a \oplus b \oplus d = 1 \oplus d = \neg d$, and the third parity bit $a \oplus b \oplus d$ is 0). But 10?1 and 01?1 are both at least two errors away from the received message 0000.

1000000. We've received message bits 1000 and parity bits 000. If the message bits were uncorrupted, then the correct parity bits would have been 011. But then we would have to have suffered *two* transmission errors in the parity bits, and we're assuming that at most one error occurred. Thus the error is in the message bits; the original message is 0000, and the first bit of the message was corrupted.

1011010. The parity bits for the message 1011 are indeed 010, so 1011010 is itself a legal codeword for the message 1011, and no errors occurred at all.

1110111. These received bits are consistent with the message 1111 with parity bits 111, where the fourth bit of the message was flipped.

From this example, the basic approach to decoding the Hamming code should start to coalesce. Briefly, we compute what the parity bits *should have been,* supposing that the received message bits (the first four bits of the received codeword) are correct; comparing the computed parity bits to the received parity bits allows us to deduce which, if any, of the transmitted bits were erroneous. (More on efficient decoding later.)

Why does this approach to decoding work? (And, relatedly, why were the parity bits of the Hamming code chosen the way that they were?) There are two critical properties in the Hamming code's parity bits.

First, *every message bit appears in at least two parity bits.* Thus any error in a received parity bit is distinguishable from an error in a received message bit: an erroneous message bit will cause at least two parity bits to look wrong; an erroneous parity bit will cause only that one parity bit to look wrong.

Second, *no two message bits appear in precisely the same set of parity bits.* Thus any error in a received message bit has a different "signature" of wrong-looking parity bits: an error in bit a affects parity bits #2 and #3; b affects parity bits #1 and #3; c affects #1 and #2; and d affects all three parity bits. Because all four of these signatures are different, we can distinguish *which* message bit was corrupted based on which set of two or more parity bits look wrong.

Rate and minimum distance of the Hamming code

Let's use the intuition that we've developed so far to establish the rate and minimum distance for the Hamming code:

Lemma 4.14: Distance and rate of the Hamming code.
The Hamming code has rate $\frac{4}{7}$ and minimum distance 3.

Proof. The rate is straightforward to compute: we have 4-bit messages and 7-bit codewords, so the rate is $\frac{4}{7}$ by definition.

There are several ways to convince yourself that the minimum distance is 3—perhaps the simplest way (though certainly the most tedious) is to compute the Hamming distance between each pair of codewords in Figure 4.9. (There are only 16 codewords, so we just have to check that all $\frac{16 \cdot 15}{2} = 120$ pairs of distinct codewords have Hamming distance at least three.) You'll write a program to verify this claim in Exercise 4.24. But here's a different argument.

Consider any two distinct messages $m \in \{0,1\}^4$ and $m' \in \{0,1\}^4$. We must establish that the codewords c and c' associated with m and m' satisfy $\Delta(c,c') \geq 3$. We'll argue for this fact by looking at three separate cases, depending on $\Delta(m,m')$:

Case I: $\Delta(m,m') \geq 3$. Then we're done immediately: the message bits of c and c' differ in at least three positions (even without looking at the parity bits).

Case II: $\Delta(m,m') = 2$. Then at least one of the three parity bits contains exactly one of the two bit positions where $m_i \neq m'_i$. (This fact follows from the second crucial property above, that no two message bits appear in precisely the same set of parity bits.) Therefore this parity bit differs in c and c'. Thus there are two message bits and at least one parity bit that differ, so $\Delta(c,c') \geq 3$.

Case III: $\Delta(m,m') = 1$. Then at least two of the three parity bits contain the bit position where $m_i \neq m'_i$. (This fact follows from the first crucial property above, that every message bit appears in at least two parity bits.) Thus there are at least two parity bits and one message bit that differ, and $\Delta(c,c') \geq 3$.

Note that $\Delta(m,m')$ must be 1, 2, or ≥ 3—it can't be zero because $m \neq m'$—so, no matter what $\Delta(m,m')$ is, we've established that $\Delta(c,c') \geq 3$.

Because, for the codewords corresponding to messages 0000 and 1110, we have $\Delta(0000000, 1110000) = 3$, the minimum distance is in fact exactly equal to three. □

Lemma 4.14 says that the Hamming code encodes 4-bit messages with minimum distance 3 and rate $\frac{4}{7}$; thus we've proven Theorem 4.8. Let's again reflect a little on the proof. Our proof of the minimum distance in Lemma 4.14 was a *proof by cases:* we divided pairs of codewords into three different categories (differing in 1, 2, or ≥ 3 bits), and then used three different arguments to show that the corresponding codewords differed in ≥ 3 places. So we showed that the desired distance property was true in all three cases—and, crucially, that one of the cases applies for every pair of codewords.

> *Problem-solving tip:* If you discover that a proposition seems true "for different reasons" in different circumstances (and those circumstances seem to cover all possible scenarios!), then a proof by cases may be a good strategy to employ.

Although we're mostly omitting any discussion of the efficiency of encoding and decoding, it's worth a brief mention here. (The speed of these algorithms is a big deal for error-correcting codes used in practice.) The algorithm for decoding under the Hamming code is suggested by Figure 4.11: we calculate what the parity bits *would have been* if the received message bits were uncorrupted, and identify which received parity bits don't match those calculated parity bits. Figure 4.11 tells us what inference to draw from each constellation of mismatched parity bits.

parity bit #1: $b \oplus c \oplus d$	✓	✗	✓	✓	✗	✗	✓	✗
parity bit #2: $a \oplus c \oplus d$	✓	✓	✗	✓	✗	✓	✗	✗
parity bit #3: $a \oplus b \oplus d$	✓	✓	✓	✗	✓	✗	✗	✗
location of error (if any)	none!	parity bit #1	parity bit #2	parity bit #3	bit c	bit b	bit a	bit d

Figure 4.11 Decoding the Hamming code. We conclude that the stated error occurred if the received parity bits and those calculated from the received message bits mismatch in the listed places.

Why does this decoding algorithm allow us to correct any single error? First, a low-level answer: the Hamming code has a minimum distance of $3 = 2 \cdot 1 + 1$, so Lemma 4.6 tells us that we can correct up to one error. So we know that a decoding scheme is possible. At a higher level, the reason that this decoding procedure works properly is that there are eight possible "≤ 1 error" corruptions of a codeword x—namely one 0-error string (x itself) and seven 1-error strings (one corresponding to an error in each of the seven bit positions of x)—and there are also eight different subsets of the three parity bits that can be "wrong." The Hamming code works by carefully selecting the parity bits in a way that each of these eight bitstrings corresponds to a different one of the eight parity-bit subsets. In Exercises 4.25–4.28, you'll explore longer versions of the Hamming code (with longer messages and more parity bits) with the same relationship.

Taking it further: As we've said, our attention here is mostly on the proofs and the proof techniques that we've used to establish the claims in this section, rather than on error-correcting codes themselves. But see p. 4-23 for an introduction to *Reed–Solomon codes,* the basis of the error-correcting codes used in CDs/DVDs (among other applications).

4.2.5 Upper Bounds on Rates

In Sections 4.2.3 and 4.2.4, we've constructed two different codes, both for 4-bit messages with minimum distance 3: the repetition code (rate $\frac{4}{12}$) and the Hamming code (rate $\frac{4}{7}$). Because the message lengths and minimum distances match, and because higher rates are better, the Hamming code is better. Here we'll consider whether we can improve the rate further, while still encoding 4-bit messages with minimum distance 3. (In other words, can we make the codewords shorter than 7 bits?) The answer turns out to be "no"—and we'll prove that it's impossible.

"Balls" around codewords

We'll start by thinking about "balls" around codewords in a general code. (The *ball of radius r around* $x \in \{0,1\}^n$ is the set $\{x' : \Delta(x, x') \leq r\}$—that is, the set of all points that are within Hamming distance r of x.) Here's a first observation:

Lemma 4.15: The size of a ball of radius 1 in $\{0,1\}^n$.
Let $x \in \{0,1\}^n$, and define $X = \big\{x' \in \{0,1\}^n : \Delta(x, x') \leq 1\big\}$. Then $|X| = n + 1$.

Proof. The bitstring x itself is an element of X, as are all bitstrings x' that differ from x in exactly one position. There are n such strings x': one that is x with the first bit flipped, one that is x with the second bit flipped; . . .; and one that is x with the nth bit flipped. Thus there are $1 + n$ total bitstrings in X. □

Here's a second useful fact about these balls: in a code $\mathcal{C}$, the balls around codewords (of radius related to the minimum distance of $\mathcal{C}$) cannot overlap.

Lemma 4.16: Balls around codewords are disjoint.
Let $\mathcal{C} \subseteq \{0,1\}^n$ be a code with minimum distance $2t + 1$. For distinct codewords $x, y \in \mathcal{C}$, the sets $\left\{x' \in \{0,1\}^n : \Delta(x,x') \leq t\right\}$ and $\left\{y' \in \{0,1\}^n : \Delta(y,y') \leq t\right\}$ are disjoint.

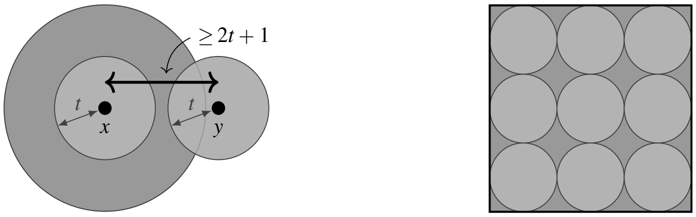
Proof. Suppose not: in other words, suppose that the set $X = \left\{x' \in \{0,1\}^n : \Delta(x,x') \leq t\right\}$ and the set $Y = \left\{y' \in \{0,1\}^n : \Delta(y,y') \leq t\right\}$ are *not* disjoint. We will derive a contradiction from this assumption—that is, a statement that can't possibly be true—so we'll have proven that $X \cap Y \neq \varnothing \Rightarrow$ False. But this implication allows us to conclude that $X \cap Y = \varnothing$, because $\neg p \Rightarrow \text{False} \equiv p$. That is, we're using a *proof by contradiction.*

To start again from the beginning: suppose that X and Y are not disjoint. That is, suppose that there is some bitstring $z \in \{0,1\}^n$ such that $z \in X$ and $z \in Y$. In other words, by definition of X and Y, there is a bitstring $z \in \{0,1\}^n$ such that $\Delta(x,z) \leq t$ *and* $\Delta(y,z) \leq t$. But if $\Delta(x,z) \leq t$ and $\Delta(y,z) \leq t$, then, by the triangle inequality, we know

$$\Delta(x,y) \leq \Delta(x,z) + \Delta(z,y) \leq t + t = 2t.$$

Therefore $\Delta(x,y) \leq 2t$—but then we have two distinct codewords $x, y \in \mathcal{C}$ with $\Delta(x,y) \leq 2t$. This condition contradicts the assumption that the minimum distance of $\mathcal{C}$ is $2t + 1$. (See Figure 4.12a.) □

We could have used Lemma 4.16 to establish the error-correction part of Theorem 4.6—a bitstring corrupted by $\leq t$ errors from a codeword c is closer to c than to any other codeword—but here we'll use it, plus Lemma 4.15, to establish a upper bound on the rate of codes. But, first, let's pause to look at a similar argument in a different (but presumably more familiar) domain: normal Euclidean geometry.



(a) If the minimum distance is $2t + 1$, the "balls" of radius t around each codeword are disjoint.

(b) How many unit circles (circles with radius 1) can be packed into a 6-by-6 square?

Figure 4.12 Circle packing.

Problem-solving tip: When you're facing a problem in a less familiar domain, try to find an analogous problem in a different, more familiar setting to help gain intuition.

In a *circle-packing* problem, we are given an enclosing shape, and we're asked to place ("pack") as many nonoverlapping unit circles (of radius 1) into that shape as possible. (*Sphere packing*—what grocers have to do with oranges—is the 3-dimensional analogue.) How many unit circles can we fit into a 6-by-6 square, for example? (See Figure 4.12b.) Here's an argument that it's at most 11: a unit circle has area $\pi \cdot 1^2 = \pi$, and the 6-by-6 square has area 36; thus we certainly can't fit more than $\frac{36}{\pi} \approx 11.459$ nonoverlapping circles into the square. There isn't *room* for 12. (In fact, we can't even fit 10, because the circles won't nestle together without wasting space "in between." Thus, in this case we'd say that the area-based bound is *loose*.)

Using packing arguments to derive bounds on error-correcting codes

Now, let's return to error-correcting codes, and use the circle-packing intuition (and the last two lemmas) to prove a bound on the number of n-bit codewords that can "fit" into $\{0,1\}^n$ with minimum distance 3:

Lemma 4.17: The "sphere-packing bound": distance-3 version.
Let $\mathcal{C} \subseteq \{0,1\}^n$ be a code with minimum distance three. Then $|\mathcal{C}| \leq \frac{2^n}{n+1}$.

Proof. For each $x \in \mathcal{C}$, let $S_x = \left\{x' \in \{0,1\}^n : \Delta(x',x) \leq 1\right\}$ be the ball of radius 1 around x. Lemma 4.15 says that $|S_x| = n+1$ for each x. Further, Lemma 4.16 says that every element of $\{0,1\}^n$ is in at most one S_x because the balls are disjoint. Therefore,

$$|\left\{x' \in \{0,1\}^n : x' \text{ is in one of the } S_x \text{ balls}\right\}| = \sum_{x \in \mathcal{C}} |S_x| = \sum_{x \in \mathcal{C}} (n+1) = |\mathcal{C}| \cdot (n+1).$$

Also observe that every element of any S_x is an n-bit string. There are only 2^n different n-bit strings, so therefore

$$|\left\{x' \in \{0,1\}^n : x' \text{ is in one of the } S_x \text{ balls}\right\}| \leq 2^n.$$

Putting together these two facts, we see that $|\mathcal{C}| \cdot (n+1) \leq 2^n$. Solving for $|\mathcal{C}|$ yields the desired relationship: $|\mathcal{C}| \leq \frac{2^n}{n+1}$. □

Corollary 4.18. Any code with messages of length 4 and minimum distance 3 has codewords of length at least 7. (Thus the Hamming code has the best possible rate among all such codes.)

Proof. By Lemma 4.17, we know that $|\mathcal{C}| \leq \frac{2^n}{n+1}$. With 4-bit messages we have $|\mathcal{C}| = 16$, so we know that $16 \leq \frac{2^n}{n+1}$, or, equivalently, that $2^n \geq 16 \cdot (n+1)$. And $2^7 = 16 \cdot (7+1)$, while for any $n < 7$ this inequality does not hold. □

Corollary 4.18 implies Theorem 4.9, so we've now proven the three claims that we set out to establish. Before we close, though, we'll mention a few extensions. Lemma 4.16 was general, for any code with an odd minimum distance. But Lemma 4.15 was specifically about codes with minimum distance 3. To generalize the latter lemma, we'd need techniques from *counting* (see Chapter 9, specifically Section 9.4.)

Another interesting question: when is the bound from Lemma 4.17 exactly achievable? If we have k-bit messages, n-bit codewords, and minimum distance 3, then Lemma 4.17 says that $2^k \leq \frac{2^n}{n+1}$, or, taking logs, that $k \leq n - \log_2(n+1)$. Because k has to be an integer, this bound is exactly achievable only when $n+1$ is an exact power of two. (For example, if $n = 9$, this bound requires us to have $2^k \leq \frac{2^9}{10} = \frac{512}{10} = 51.2$. In other words, we need $k \leq \log_2 51.2 \approx 5.678$. But, because $k \in \mathbb{Z}$, in fact we need $k \leq 5$. That means that this bound is *not* exactly achievable for $n = 9$.) However, it's possible to give a version of the Hamming code for $n = 15$ and $k = 7$ with minimum distance 3, as you'll show in Exercise 4.26. (In fact, there's a version of the Hamming code for any $n = 2^\ell - 1$; see Exercise 4.28.)

Computer Science Connections

Reed–Solomon Codes

The error-correcting codes that are used in CDs, DVDs, and QR codes (see Figure 4.13) are a bit more complicated than Repetition or Hamming codes, but they perform better. These codes are called *Reed–Solomon codes*—named after Irving Reed (1923–2012) and Gustave Solomon (1930–1996), who invented them in 1960—and they're based on polynomials and modular arithmetic. We'll leave out a lot of the details, but here is a brief sketch of how they work.

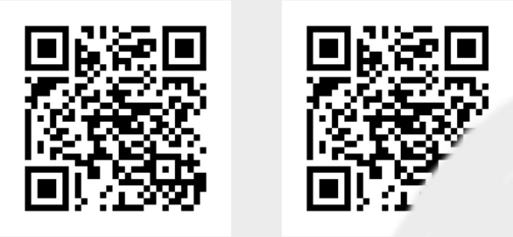


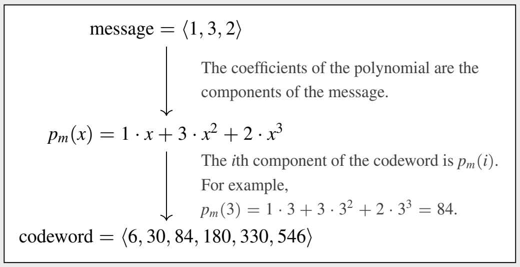
Figure 4.13 A QR ("quick response") code, which is still readable even if some portions are misread or damaged. (Either version of this QR code can be correctly decoded.)

The first difference from the codes that we've seen so far: we're going to go beyond bits, to a larger "alphabet" of characters in our messages and codewords. Instead of encoding messages from $\{0,1\}^k$, we're going to encode messages from $\{0,1,\ldots,q\}^k$, for some integer q.

Here's the core idea of the encoding. Given a message $m = \langle m_1, m_2, \ldots, m_k \rangle$, we will define a polynomial $p_m(x)$ with the *coefficients of the polynomial corresponding to the characters of the message*:

$$p_m(x) = \sum_{i=1}^{k} m_i \cdot x^i.$$

Now, to encode the message m, we evaluate the polynomial for several values of x: specifically, the codeword corresponding to m will be $\langle p_m(1), p_m(2), \ldots, p_m(n) \rangle$. See Figure 4.14 for an example.



message $= \langle 1, 3, 2 \rangle$

The coefficients of the polynomial are the components of the message.

$p_m(x) = 1 \cdot x + 3 \cdot x^2 + 2 \cdot x^3$

The ith component of the codeword is $p_m(i)$. For example, $p_m(3) = 1 \cdot 3 + 3 \cdot 3^2 + 2 \cdot 3^3 = 84$.

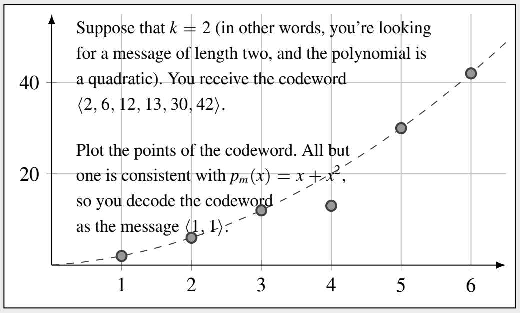
codeword $= \langle 6, 30, 84, 180, 330, 546 \rangle$

Figure 4.14 Encoding the message $\langle 1, 3, 2 \rangle$ using $n = 6$.

Suppose that we use a k-character message and an n-character output. (We're going to have to pick a value of n—bigger for more error correction, smaller for more efficiency.) It's easy enough to compute that the rate is $\frac{k}{n}$. But what about the minimum distance of the code? Consider two distinct messages m and m'. Note that p_m and $p_{m'}$ are two different polynomials of degree at most k, which means that $f(x) = p_m(x) - p_{m'}(x)$ is a polynomial of degree at most k, too. Therefore f has at most k roots, by Lemma 2.58. And $\{x : f(x) = 0\} = \{x : p_m(x) = p_{m'}(x)\}$, which means that there are at most k values of x for which $p_m(x) = p_{m'}(x)$. We encoded m and m' by evaluating p_m and $p_{m'}$ on n different inputs, so there are at least $n - k$ inputs on which these two polynomials *disagree*. Thus the minimum distance is at least $n - k$. For example, if we pick $n = 2k$, then we achieve rate $\frac{1}{2}$ and minimum distance k.

How might we decode Reed–Solomon codes? Efficient decoding algorithms rely on some results from linear algebra, but the basic idea is to find the degree-k polynomial that goes through as many of the given points as possible. See Figure 4.15.

We've left out a few key details of Reed–Solomon codes. One is that we computed the rate misleadingly: we only counted the number of slots, rather than the



Suppose that $k = 2$ (in other words, you're looking for a message of length two, and the polynomial is a quadratic). You receive the codeword $\langle 2, 6, 12, 13, 30, 42 \rangle$.

Plot the points of the codeword. All but one is consistent with $p_m(x) = x + x^2$, so you decode the codeword as the message $\langle 1, 1 \rangle$.

Figure 4.15 Decoding a (corrupted) codeword.

"size" of those slots. (Figure 4.14 shows that the numbers can get pretty big!) In real Reed–Solomon codes, every value is stored *modulo a prime*. See p. 7-38 for how (and why) this fix works. There's also an additional clever trick used in the physical layout of the encoded information on a CD, DVD, or QR code: the bits for a particular codeword are spread out over the physical encoding, so that a single physical scratch or tear doesn't cause errors all to occur in the same codeword.

EXERCISES

4.1 *(programming required.)* The algorithm for testing whether a given credit-card number is valid is shown in Figure 4.16a, and an example is shown in Figure 4.16b. (Try executing **cc-check** from Figure 4.16a on a few credit-card numbers, to make sure that you've understood the algorithm correctly.) This code can detect any one substitution error, because

$$\begin{array}{cccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 0 & 2 & 4 & 6 & 8 & 10 \to 1+0=1 & 12 \to 1+2=3 & 14 \to 1+4=5 & 16 \to 1+6=7 & 18 \to 1+8=9 \end{array}$$

are all distinct (so changing any single digit, including in odd-indexed digits, changes the overall value of *sum*). Implement **cc-check** in a programming language of your choice. Extend your implementation so that, if it's given any 16-digit credit/debit-card number with a single digit replaced by a `"?"`, it computes and outputs the correct missing digit.

4.2 Suppose that we modified **cc-check** so that, instead of *adding the ones digit and (if it exists) the tens digit* of d_i to *sum* in Line 7 of the algorithm, we instead simply added d_i. (That is, replace Line 7 by $sum := sum + d_i$.) Does this modified code still allow us to detect any single substitution error?

4.3 Consider modifying **cc-check** so that, rather than *doubling* odd-indexed digits in Line 4 of the algorithm, we *tripled* the odd-indexed digits. (That is, replace Line 4 by $d_i := 3 \cdot n_i$.) Does this modified code still allow us to detect any single substitution error?

4.4 Similarly, imagine modifying **cc-check** to *quintuple* instead of double (that is, we replace Line 4 by $d_i := 5 \cdot n_i$). Does this modified code still allow us to detect any single substitution error?

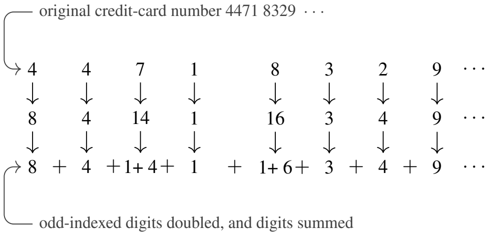
4.5 There are simpler schemes that can detect a single substitution error than the one in **cc-check**: for example, we could simply ensure that the sum of all the digits themselves (undoubled) is divisible by 10. (Just skip the doubling step.) The credit-card encoding system includes the more complicated doubling step to help it detect a different type of error, called a *transposition error,* where two adjacent digits are recorded in reverse order. (If two digits are swapped, then the "wrong" digit is multiplied by two, and so this kind of error might be detectable.) Does **cc-check** detect every possible transposition error?

A metric space *consists of a set X and a function $d : X \times X \to \mathbb{R}^{\geq 0}$, called a* distance function*, where d obeys the following three properties:*

- reflexivity*: for any x and y in X, we have $d(x, x) = 0$, and $d(x, y) \neq 0$ whenever $x \neq y$.*
- symmetry*: for any $x, y \in X$, we have $d(x, y) = d(y, x)$.*
- triangle inequality*: for any $x, y, z \in X$, we have $d(x, y) \leq d(x, z) + d(z, y)$.*

```
cc-check(n):
Input: a 16-digit credit-card number $n \in \{0, 1, \ldots, 9\}^{16}$
1 $sum := 0$
2 for $i = 1, 2, \ldots, 16$:
3     if $i$ is odd then
4         $d_i := 2 \cdot n_i$
5     else
6         $d_i := n_i$
7     Increase sum by the ones' and tens' digits of $d_i$.
8     // In other words, $sum := sum + (d_i \bmod 10) + \lfloor \frac{d_i}{10} \rfloor$.
9 return True if $sum \bmod 10 = 0$, and False otherwise.
```

(a) The algorithm itself.


original credit-card number 4471 8329 ···
4 4 7 1 8 3 2 9 ···
8 4 14 1 16 3 4 9 ···
8 + 4 +1+ 4+ 1 + 1+ 6+ 3 + 4 + 9 ···
odd-indexed digits doubled, and digits summed

(b) An example of the calculation.

Figure 4.16 An algorithm for testing the validity of credit-card numbers.

When it satisfies all three conditions, we call the function d a metric. *The next few exercises consider several different distance functions $d : \{0,1\}^n \times \{0,1\}^n \to \mathbb{Z}^{\geq 0}$. For each, you're asked to prove whether the given function d is a metric. (In other words, you should either prove that d satisfies reflexivity, symmetry, and the triangle inequality; or prove that d fails to satisfy one or more of these properties.)*

4.6 In this section, we've measured the distance between bitstrings using the Hamming distance, which counts the number of positions in which x and y differ. Prove that the Hamming distance function $\Delta : \{0,1\}^n \times \{0,1\}^n \to \mathbb{Z}^{\geq 0}$ is a metric. *(Hint: think about one bit at a time.)*

4.7 For $x, y \in \{0,1\}^n$, define $d(x,y)$ as the smallest $i \in \{0,1,\ldots,n\}$ such that $x_{i+1,\ldots,n} = y_{i+1,\ldots,n}$. For example, $d(01000, 10101) = 5$ and $d(010\underline{00}, 101\underline{00}) = 3$ and $d(01\underline{000}, 10\underline{000}) = 2$ and $d(1\underline{1010}, 0\underline{1010}) = 1$. (This function measures how far into x and y we must go before the remaining parts match; we could also define $d(x,y)$ as the largest $i \in \{0,1,\ldots,n\}$ such that $x_i \neq y_i$, where we treat $x_0 \neq y_0$.) Is d a metric?

4.8 For $x, y \in \{0,1\}^n$, define $d(x,y)$ as the length of the longest consecutive run of differing bits in corresponding positions of x and y—that is, $d(x,y) = \max\{j - i : \text{for all } k = i, i+1, \ldots, j \text{ we have } x_k \neq y_k\}$. For example, $d(\underline{010}00, \underline{101}01) = 3$ and $d(0\underline{010}0, 0\underline{101}0) = 3$ and $d(\underline{01}000, \underline{10}000) = 2$ and $d(\underline{1}10\underline{1}0, \underline{0}10\underline{0}0) = 1$. Is d a metric?

4.9 For $x, y \in \{0,1\}^n$, define $d(x,y)$ as the difference in the number of ones that appears in the two bitstrings—that is,

$$d(x,y) = \Big|\, |\{i : x_i = 1\}| - |\{i : y_i = 1\}| \,\Big|.$$

(The vertical bars here are a little confusing: the bars around $|\{i : x_i = 1\}|$ and $|\{i : y_i = 1\}|$ denote set cardinality, while the outer vertical bars denote absolute value.) For example, $d(01000, 10101) = |1-3| = 2$ and $d(01000, 10100) = |1-2| = 1$ and $d(01000, 10000) = |1-1| = 0$ and $d(11010, 01010) = |2-2| = 0$. Is d a metric?

4.10 The distance version of the *Sørensen index* (a.k.a. the *Dice coefficient*) defines the distance based on the fraction of ones in x or y that are in the same positions. (The Sørensen/Dice measure is named after independent work by two ecologists from the 1940s, the Danish botanist Thorvald Sørensen (1902–1973) and the American mammalogist Lee Raymond Dice (1887–1977).) Specifically,

$$d(x,y) = 1 - \frac{2\sum_i x_i \cdot y_i}{\sum_i x_i + y_i}.$$

For example, $d(01000, 10101) = 1 - \frac{2 \cdot 0}{1+3} = 1 - \frac{0}{4} = 1$ and $d(00\underline{1}00, 01\underline{1}10) = 1 - \frac{2 \cdot 1}{1+3} = 1 - \frac{2}{4} = 1/2$ and $d(0\underline{1}000, 1\underline{1}000) = 1 - \frac{2 \cdot 1}{1+2} = 1 - \frac{2}{3} = 1/3$ and $d(1\underline{1}0\underline{1}0, 0\underline{1}0\underline{1}0) = 1 - \frac{2 \cdot 2}{3+2} = 1 - \frac{2}{5} = 3/5$. Is d a metric?

4.11 For $x, y \in \{0,1\}^n$, define $d(x,y)$ as the difference in the numbers that are represented by the two strings in binary. Writing this function formally is probably less helpful (particularly because the higher powers of 2 have lower indices), but here it is: $d(x,y) = \left|\sum_{i=1}^{n} x_i \cdot 2^{n-i} - \sum_{i=1}^{n} y_i 2^{n-i}\right|$. For example, $d(01000, 10101) = |8 - 21| = 13$ and $d(01000, 10100) = |8 - 20| = 12$ and $d(01000, 10000) = |8 - 16| = 8$ and $d(11010, 01010) = |26 - 10| = 16$. Is d a metric?

4.12 Show that we can't improve on the parameters in Theorem 4.6: for any integer $t \geq 0$, prove that a code with minimum distance $2t+1$ cannot correct $t+1$ or detect $2t+1$ errors.

4.13 Theorem 4.6 describes the error-detecting and error-correcting properties for a code whose minimum distance is any odd integer. This exercise asks you to give the analogous analysis for a code whose minimum distance is any even integer. Let $t \geq 1$ be any integer, and let $\mathcal{C}$ be a code with minimum distance $2t$. How many errors can $\mathcal{C}$ detect and correct? Prove your answers.

Let $c \in \{0,1\}^n$ be a codeword. Until now, we've mostly talked about substitution errors, *in which a single bit of c is flipped from* 0 *to* 1, *or from* 1 *to* 0. *The next few exercises explore two other types of errors.*

An erasure error *occurs when a bit of c isn't successfully transmitted, but the recipient is informed that the transmission of the corresponding bit wasn't successful. We can view an erasure error as replacing a bit c_i from c with a '?' (as in Exercise 4.1,* *for credit-card numbers). Thus, unlike a substitution error, the recipient* knows *which bit was erased. (So a codeword* 1100110 *might become* 1?0011? *after two erasure errors.) When codeword $c \in \{0,1\}^n$ is sent, the receiver gets a corrupted codeword $c' \in \{0,1,?\}^n$ and where all unerased bits were transmitted correctly (that is, if $c'_i \in \{0,1\}$, then $c'_i = c_i$).*

A deletion error *is like a "silent erasure" error: a bit fails to be transmitted, but there's no indication to the recipient as to where the deletion occurred. (So a codeword* 1100110 *might become* 10011 *after two deletion errors.)*

4.14 Let $\mathcal{C}$ be a code that can *detect* t substitution errors. Prove that $\mathcal{C}$ can *correct* t erasure errors.

4.15 Let $\mathcal{C}$ be a code that can correct t deletion errors. Prove that $\mathcal{C}$ can correct t erasure errors.

4.16 Give an example of a code that *can* correct one erasure error, but *can't* correct one deletion error.

4.17 Consider the "code" where all n-bit strings are codewords. (That is, $\mathcal{C} = \{0,1\}^n$.) What is the rate and minimum distance, and how many errors can it detect or correct?

4.18 Define the *trivial code* as $\mathcal{C} = \{0^n, 1^n\}$. What is the rate and minimum distance, and how many errors can it detect or correct?

4.19 Consider the *parity-check code*, defined as follows: the codewords are all n-bit strings with an even number of bits set to 1. What is the rate and minimum distance, and how many errors can it detect or correct?

4.20 Let's extend the idea of the parity-check code as an add-on to any existing code with odd minimum distance. Let $\mathcal{C} \subseteq \{0,1\}^n$ be a code with minimum distance $2t+1$, for some integer $t \geq 0$. Consider a new code $\mathcal{C}'$, in which we augment every codeword of C by adding a *parity bit*, which is zero if the number of ones in the original codeword is even and one if the number is odd, as follows:

$$\mathcal{C}' = \left\{ \langle x_1, x_2, \ldots, x_n, (\textstyle\sum_{i=1}^n x_i) \bmod 2 \rangle : x \in \mathcal{C} \right\}.$$

Then the minimum distance of $\mathcal{C}'$ is $2t+2$. *(Hint: consider two distinct codewords $x, y \in \mathcal{C}$. You have to argue that the corresponding codewords $x', y' \in \mathcal{C}$ have Hamming distance $2t+2$ or more. Use two different cases, depending on the value of $\Delta(x,y)$.)*

4.21 Show that we can correctly decode the Repetition$_\ell$ code as follows: given a bitstring c', for each bit position i, we take the majority vote of the ℓ blocks' ith bit in c', breaking ties arbitrarily. (In other words, prove that this algorithm actually gives the codeword that's closest to c'.)

4.22 In some error-correcting codes, for certain errors, we may be able to correct more errors than Theorem 4.6 suggests: that is, the minimum distance is $2t+1$, but we can correct certain sequences of $> t$ errors. We've already seen that we can't successfully correct *every* such sequence of errors, but we may be able to successfully handle *some* sequences of errors using the standard algorithm for error correction (returning the closest codeword). The Repetition$_3$ code with 4-bit messages is only guaranteed to correct 1 error. What's the largest number of errors that can possibly be corrected successfully by this code? Explain your answer.

4.23 In contrast to the Repetition$_3$ code (as in Exercise 4.22), under the Hamming code we *never* correct more than 1 error successfully. Prove why not.

4.24 *(programming required.)* Write a program, in a programming language of your choice, to verify that any two codewords in the Hamming code differ in at least three bit positions.

Let's find the "next" Hamming code, with 7*-bit messages and* 11*-bit codewords and a minimum distance of* 3*. We'll use the same style of codeword as in Definition 4.13: the first* 7 *bits of the codeword will simply be the message, and the next* 4 *bits will be parity bits (each for some subset of the message bits).*

4.25 To achieve minimum distance 3, it will suffice to have parity bits with the following properties:

(a) each bit of the original message appears in at least two parity bits; and

(b) no two bits of the original message appear in exactly the same set of parity bits.

Prove that these conditions are sufficient. That is, prove that any set of parity bits that satisfy conditions (a) and (b) ensure that the resulting code has minimum distance 3.

4.26 Define 4 parity bits for 11-bit messages that satisfy conditions (a) and (b) from Exercise 4.25.

4.27 Define 5 parity bits for 26-bit messages that satisfy conditions (a) and (b) from Exercise 4.25.

4.28 Let $\ell \in \mathbb{Z}^{>0}$, and let $n = 2^\ell - 1$. Prove that a code with n-bit codewords, minimum distance 3, and messages of length $n - \ell$ is achievable. *(Hint: look at all ℓ-bit bitstrings; use the bits to identify which message bits are part of which parity bits.)*

4.29 You have come into possession of 8 bottles of "poison," except, you've learned, 7 are fake poison and only 1 is really poisonous. Your master plan to take over the world requires you to identify the poison *by tomorrow.* Luckily, as an evil genius, you have a small collection of very expensive rats, which you can use for testing. You can give samples from bottles to multiple rats simultaneously (a rat can receive a mixture of samples from more than one bottle), and then wait for a day to see which ones die. Obviously you can identify the real poison with 8 rats (one bottle each), or even with 7 (one bottle each, one unused bottle; if all rats survive then the leftover bottle is the poison). But how many rats do you *need* to identify the poison? (Make the number as small as possible.)

> **Taking it further:** The idea in Exercise 4.29 is very similar to the idea of "pooled testing," which held major promise in the early stages of the COVID-19 pandemic as a way to mitigate the paucity of available tests for the disease. Usually, in pooled testing, the idea is a two-stage approach: simultaneously test the samples from k individuals for the disease, and then, if (and only if) the pool test is positive, complete k individual tests afterwards to determine which individual(s) was/were infected. (Exercise 4.29 takes a slightly different, but related, approach; it asks you to determine how to perform more pooled tests simultaneously, to avoid having to do individual tests.) One key difference: this exercise tells you to *assume* that there is exactly one positive sample; in the COVID-testing scenario, the number of positive samples is unknown. Another key difference: real-life tests are not 100% accurate, and accounting for false positive or false negatives is critical. The appropriate tradeoffs in the size of the test pool depend on the accuracy of the test and the prevalence of the disease. For an example analysis of the pooled-testing approach for COVID-19, see [28].

Let $c \in \{0,1\}^{23}$. A handy fact (which you'll show in Exercise 9.135, after we've developed the necessary tools for counting to figure out this quantity): the number of 23*-bit strings c' with $\Delta(c, c') \leq 3$ is exactly* $2048 = 2^{11} = 2^{23-12}$. *This fact means that (according to a generalization of Lemma 4.17) it might be possible to achieve a code with* 12*-bit messages;* 23*-bit codewords; and minimum distance* 7. *In fact, these parameters* are *achievable—and a code that achieves these parameters is surprisingly simple to construct. The* Golay code *is an error-correcting code that can be constructed by the so-called "greedy" algorithm in Figure 4.17. (The all-zero vector will be added to S in the first iteration of the* **while** *loop; a hundred and twenty-seven iterations later,* 00000000000000001111111 *will be the second element added to S, and so forth.)*

(The Golay code is named after Marcel Golay (1902–1989), a Swiss researcher who discovered them in 1949, just before Hamming discovered what would later be called the Hamming code. Golay also made major contributions in chemistry and optics. A slight variant of the Golay code was used by NASA around 1980 to communicate with the Voyager spacecraft as they traveled to Saturn and Jupiter.)

4.30 *(programming required.)* Write a program, in a language of your choice (but see the warning below), that implements the algorithm in Figure 4.17, and outputs the list of the $2^{12} = 4096$ different 23-bit codewords of the Golay code in a file, one per line.

Implementation hint: suppose you represent the set S as an array, appending each element that passes the test in Line 3 to the end of the array. When you add a bitstring x to S, the very next thing you do is to consider adding $x + 1$ to S. Implementing Line 3 by

```
1 $S := \varnothing$
2 for $x \in \{0,1\}^{23}$ (in numerical order):    // $x$ = 00000000000000000000000, 00000000000000000000001, ..., 11111111111111111111111
3     if $\Delta(x, y) \geq 7$ for all $y \in S$ then
4         add $x$ to $S$
5 return $S$
```

Figure 4.17 The "greedy algorithm" for generating the Golay code.

starting at the x-end of the array will make your code *much* faster than if you start at the 00000000000000000000000-end of the array. Think about why!

Implementation warning: this algorithm is not very efficient! We're doing 2^{23} iterations, each of which might involve checking the Hamming distance of as many as 2^{12} pairs of strings. On an aging laptop, my Python solution took about ten minutes to complete; if you ignore the implementation hint from the previous paragraph, it took 80 minutes. (I also implemented a solution in C; it took about 10 seconds following the hint, and 100 seconds not following the hint.)

4.31 You and six other friends are imprisoned by an evil genius, in a room filled with eight bubbling bottles marked as "poison." (Though, really, seven of them look perfectly safe to you.) The evil genius, though, admires skill with bitstrings and computation, and offers you all a deal. You and your friends will each have a red or blue hat placed on your heads randomly. (Each hat has a 50% chance of being red and 50% chance of being blue, independent of all other hats' colors.) Each person can each see all hats except their own. After a brief moment to look at each others' hats, all of you must simultaneously say one of three things: RED, BLUE, or PASS. The evil genius will release all of you from your imprisonment if:

- everyone who says RED or BLUE correctly identifies their hat color; and
- at least one person says a color (that is, not everybody says PASS).

You may collaborate on a strategy before the hats are placed on your heads, but once the hat is in place, no communication is allowed. An example strategy: all 7 of you pick a random color and say it. (You succeed with probability $(1/2)^7 = 1/128 \approx 0.0078$.) Another example: you number yourselves $1, 2, \ldots, 7$, and person #7 picks a random color and says it; everyone else passes. (You succeed with probability $1/2$.)

Can you succeed with probability better than $1/2$? If so, how?

4.32 In Section 4.2.5, we proved an upper bound for the rate of a code with a particular minimum distance, based on the volume of "spheres" around each codeword. There are other bounds that we can prove, with different justifications. Here you'll prove what's called the *Singleton bound.* (Confusingly, the Singleton bound is named after Richard Singleton (1928–2007), an American computer scientist; it has nothing to do with singleton sets [sets containing only one element].)

Suppose that we have a code $\mathcal{C} \subseteq \{0,1\}^n$ with $|\mathcal{C}| = 2^k$ and minimum distance d. Prove that $k \leq n - d + 1$.

(Hint: what happens if we delete the first $d-1$ bits from each codeword?)

4.3 Proofs and Proof Techniques

> The proof of the pudding is in the eating.
>
> Miguel de Cervantes (1547–1616)
> *Don Quixote* (1605)

In Section 4.2, we saw a number of claims about error-correcting codes—and, more importantly, proofs that those claims were true. These proofs used several different styles of argument: proofs that involved straight-ahead reasoning by starting from the relevant definitions; proofs that used "case-based" reasoning; and proofs "by contradiction" that argued that x must be true because something impossible would happen if x were false. Indeed, whenever you face a claim that you need to prove, a variety of different strategies (including these strategies from Section 4.2) are possible approaches for you to employ. This section is devoted to outlining these and some other common proof strategies. We'll first catalogue these techniques in Section 4.3.1, and then, in Section 4.3.2, we'll reflect briefly on the strategies and how to choose among them—and also reflect on the *writing* part of writing proofs.

What is a proof?

This chapter is devoted to techniques for proving claims—but before we explore proof techniques, let's spend a few words discussing what a proof actually *is:*

Definition 4.19: Proof.
A *proof* of a proposition is a convincing argument that the proposition is true.

Definition 4.19 says that a proof is a "convincing argument," but it doesn't say *to whom* the argument should be convincing. The answer is: *to your reader.* This definition may be frustrating, but the point is that a proof is a piece of writing, and—just like with fiction or a persuasive essay—you must write *for your audience.*

Taking it further: Different audiences will have very different expectations for what counts as "convincing." A formal logician might not find an argument convincing unless she saw every last step, no matter how allegedly obvious or apparently trivial. An instructor of early-to-mid-level computer science class might be convinced by a proof written in paragraph form that omits some comparatively simple steps, like those that invoke the commutativity of addition, for example. A professional CS researcher reading a publication in conference proceedings would expect "elementary" calculus to be omitted. Some of the debates over what counts as convincing to an audience—in other words, what counts as a "proof"—were surprisingly controversial, particularly as computer scientists began to consider claims that had previously been the exclusive province of mathematicians. See p. 4-48 for a discussion of the *Four-Color Theorem,* which triggered many of these discussions in earnest.

To give an example of writing for different audiences, we'll give several proofs of the same result. Here's a claim regarding divisibility and factorials. (Recall that $n!$, pronounced "n factorial," is the product of the positive integers up to n; that is, $n! = n \cdot (n-1) \cdot (n-2) \cdots 1$.)

> Let n be a positive integer and let k be any integer satisfying $2 \leq k \leq n$.
>
> Then $n! + 1$ is not evenly divisible by k. (†)

Before reading further, spend a minute trying to convince yourself why (†) is true. We'll prove (†) three times, using three different levels of detail:

Example 4.7: Factorials: Proof I.

Proof (heavy detail). By the definition of factorial, we have that $n! = \prod_{i=1}^{n} i$, which can be rewritten as $n! = \left[\prod_{i=1}^{k-1} i\right] \cdot k \cdot \left[\prod_{i=k+1}^{n} i\right]$. Let $m = \left[\prod_{i=1}^{k-1} i\right] \cdot \left[\prod_{i=k+1}^{n} i\right]$. Thus we have that $n! = k \cdot m$ and $m \in \mathbb{Z}$, because the product of any finite set of integers is also an integer.

Observe that $n! + 1 = mk + 1$. We claim that there is no integer ℓ such that $k\ell = n! + 1$. First, there is no $\ell \leq m$ such that $k\ell = n! + 1$, because $k\ell \leq km = n! < n! + 1$. Second, there is no $\ell \geq m + 1$ such that $k\ell = n! + 1$, because $k \geq 2$ implies that $k\ell \geq k(m+1) = n! + k > n! + 1$. Because there is no such integer $\ell \leq m$ and no such integer $\ell > m$, the claim follows. □

Example 4.8: Factorials: Proof II.

Proof (medium detail). Define $m = \frac{n!}{k}$, so that $n! = mk$ and $n! + 1 = mk + 1$. Because k is an integer between 2 and n, the definition of factorial implies that m is an integer. But because $k \geq 2$, we know $mk < mk + 1 < (m+1)k$. Thus $mk + 1$ is not evenly divisible by k, because this quantity is strictly between two consecutive integral multiples of k, namely $m \cdot k$ and $(m+1) \cdot k$. □

Example 4.9: Factorials: Proof III.

Proof (light detail). Note that k evenly divides $n!$. The next integer evenly divisible by k is $n! + k$. But $k \geq 2$, so $n! < n! + 1 < n! + k$. The claim follows immediately. □

Writing tip: As you study the material in this book, you will frequently be given a claim and asked to prove it. To complete this task well, you must think about the question of *for whom* you are writing your proof. A reasonable guideline is that your audience for your proofs is a classmate or a fellow reader of this book who has read and understood everything up to the point of the claim that you're proving, but hasn't thought about this particular claim at all.

Which of the three proofs from Examples 4.7, 4.8, and 4.9 is best? *It depends!* The right level of detail depends on your intended reader. A typical reader of this book would probably be happiest with the medium-detail proof from Example 4.8, but it is up to you to tailor your proof to your desired reader.

Taking it further: It turns out that one can encode literally all of mathematics using a handful of set-theoretic axioms, and a lot of patience. It's possible to write down everything in this book in ultraformal set-theoretic notation, which serves the purpose of making arguments 100% airtight. But the high-level computer science content can be hard to see in that style of proof. If you've ever programmed in assembly language before, there's a close analogy: you can express every program that you've ever written in extremely low-level machine code, or you can write it in a high-level language like C or Java or Python or Scheme (and, one hopes, make the algorithm much more understandable for the reader). We'll prove a lot of facts in this book, but at the Python-like level of proof. Someone could "compile" our proofs down into the low-level set-theoretic language—but we won't bother. (Lest you underestimate the difficulty of this task: a proof that $2+2=4$ would require hundreds of steps in this low-level proof!) There are subfields of computer science ("formal methods" or "formal verification," or "automated theorem proving") that take this ultrarigorous approach: start from a list of axioms, and a list of inference rules, and a desired theorem, and derive the theorem by applying the inference rules. When it is absolutely life-or-death critical that the proof be 100% verified, then these approaches tend to be used: in verifying protocols in distributed computing, or in verifying certain crucial components of a processor, or the software components of an airplane, for example.

4.3.1 Proof Techniques

We will describe three general strategies for proofs:

direct proof: we prove a statement φ by repeatedly inferring new facts from known facts to eventually conclude φ. (Sometimes we'll divide our work into separate cases and give different proofs in each case. And if φ is of the form $p \Rightarrow q$, we'll generally assume p and then try to infer q under that assumption.)

proof by contrapositive: when the statement that we're trying to prove is an implication $p \Rightarrow q$, we can instead prove $\neg q \Rightarrow \neg p$—the *contrapositive* of the original claim. The contrapositive is logically equivalent to the original implication, so once we've proven $\neg q \Rightarrow \neg p$, we can also conclude $p \Rightarrow q$.

proof by contradiction: we prove a statement φ by repeatedly *assuming* $\neg\varphi$, and proving something impossible—that is, proving $\neg\varphi \Rightarrow$ False. Because $\neg\varphi$ therefore cannot be true, we can conclude that φ must be true.

> "When you have eliminated the impossible, whatever remains, however improbable, must be the truth."
> — Sir Arthur Conan Doyle (1859–1930), *The Sign of the Four* (1890)

We'll give some additional examples of each proof technique as we go, proving some purely arithmetic claims to illustrate the strategy.

Almost every claim that we'll prove here—or that you'll ever need to prove—will be a universally quantified statement, of the form $\forall x \in S : P(x)$. (Often the quantification will not be explicit: we view any unquantified variable in a statement as being implicitly universally quantified.) To prove a claim of the form $\forall x \in S : P(x)$, we usually proceed by considering a generic element $x \in S$, and then proving that $P(x)$ holds. (Considering a "generic" element means that we make no further assumptions about x, other than assuming that $x \in S$.) Because this proof establishes that an arbitrary $x \in S$ makes $P(x)$ true, we can conclude that $\forall x \in S : P(x)$.

Direct proofs

The simplest type of proof for a statement φ is a derivation of φ from known facts. This type of argument is called a *direct proof*:

Definition 4.20: Direct proof.
A *direct proof* of a proposition φ starts from known facts and implications, and repeatedly applies logical deduction to derive new facts, eventually leading to the conclusion φ.

Most of the proofs in Section 4.2 were direct proofs. Here's another, less complicated example:

Example 4.10: Divisibility by 4.
Let's prove the correctness of a simple test of whether a given integer is divisible by 4:

Claim: Any positive integer n is divisible by 4 if and only if its last two digits are themselves divisible by 4. (That is, n is divisible by 4 if and only if n's last two digits are in $\{00, 04, 08, \ldots, 92, 96\}$.)

Proof. Let $d_k, d_{k-1}, \ldots, d_1, d_0$ denote the digits of n, reading from left to right, so that

$$n = d_0 + 10d_1 + 100d_2 + 1000d_3 + \cdots + 10^k d_k,$$

or, dividing both sides by 4,

$$\tfrac{n}{4} = \tfrac{d_0}{4} + \tfrac{10d_1}{4} + 25d_2 + 250d_3 + \cdots + 25 \cdot 10^{k-2} d_k. \qquad (*)$$

The integer n is a divisible by 4 if and only if $\frac{n}{4}$ is an integer, which because of $(*)$ occurs if and only if the right-hand side of $(*)$ is an integer. And that's true if and only if $\frac{d_0+10d_1}{4}$ is an integer, because all other terms in the right-hand side of $(*)$ are integers. Therefore $4 \mid n$ if and only if $4 \mid (d_0 + 10d_1)$. The last two digits of n are precisely $d_0 + 10d_1$, so the claim follows. □

Note that this argument considers a generic positive integer n, and establishes the result for that generic n. The proof relies on two previously known facts: (1) an integer n is divisible by 4 if and only if $\frac{n}{4}$ is an integer; and (2) for an integer a, we have that $x + a$ is an integer if and only if x is an integer. The argument itself uses these two basic facts to derive the desired claim.

Let's give another example, this time for an implication. The proof strategy of *assuming the antecedent*, discussed in Definition 3.24 in Section 3.4.3, is a form of direct proof. To prove an implication of the form $\varphi \Rightarrow \psi$, we *assume* the antecedent φ and then prove ψ under this assumption. This proof establishes $\varphi \Rightarrow \psi$ because the only way for the implication to be false is when φ is true but ψ is false, but the proof shows that ψ is true whenever φ is true. Here's an example of this type of direct proof, for a basic fact about rational numbers. (Recall that a number x is *rational* if and only if there exist integers n and $d \neq 0$ such that $x = \frac{n}{d}$.)

Example 4.11: The product of rational numbers is rational.

Claim: If x and y are rational numbers, then so is xy.

Proof. Assume the antecedent—that is, assume that x and y are rational. By the definition of rationality, then, there exist integers n_x, n_y, $d_x \neq 0$, and $d_y \neq 0$ such that $x = \frac{n_x}{d_x}$ and $y = \frac{n_y}{d_y}$. Therefore

$$xy = \frac{n_x}{d_x} \cdot \frac{n_y}{d_y} = \frac{n_x n_y}{d_x d_y}.$$

Both $n_x n_y$ and $d_x d_y$ are integers, because the product of any two integers is also an integer. And $d_x d_y \neq 0$ because both $d_x \neq 0$ and $d_y \neq 0$. Thus xy is a rational number, by the definition of rationality. □

Proof by cases

Sometimes we'll be asked to prove a statement of the form $\forall x \in S : P(x)$ that indeed seems true for every $x \in S$—but the "reason" that $P(x)$ is true seems to be different for different "kinds" of elements x. For example, Lemma 4.14 argued that the Hamming distance between two Hamming-code codewords was at least three, based on three different arguments based on whether the corresponding messages differed in 1, in 2, or in 3 or more positions. This proof was an example of a *proof by cases:*

Definition 4.21: Proof by cases.
To give a *proof by cases* of a proposition φ, we identify a set of *cases* and then prove two different types of facts: (1) "in every case, φ holds"; and (2) one of the cases has to hold.

(Proofs by cases need not be direct proofs, but plenty of them are.) Here are two small examples of proofs by cases for properties in arithmetic:

Example 4.12: Certain squares.

Claim: Let n be any integer. Then $n \cdot (n+1)^2$ is even.

Proof. We'll give a proof by cases, based on the parity of n:

- If n is even, then any multiple of n is also even, so we're done.
- If n is odd, then $n+1$ must be even. Thus any multiple of $n+1$ is also even, so we're done again.

Because the integer n must be either even or odd, and the quantity $n \cdot (n+1)^2$ is an even number in either case, the claim follows. □

Example 4.13: An easy fact about absolute values.

Claim: Let $x \in \mathbb{R}$. Then $-|x| \leq x \leq |x|$.

Proof. Observe that $x \geq 0$ or $x \leq 0$. In both cases, we'll show the desired inequality:

- For the case that $x \geq 0$, we know $-x \leq 0 \leq x$. By the definition of absolute value, we have $|x| = x$ and $-|x| = -x$. Thus $-|x| = -x \leq 0 \leq x = |x|$.

- For the case that $x \leq 0$, we know $x \leq 0 \leq -x$. By the definition of absolute value, we have $|x| = -x$ and $-|x| = x$. Thus $-|x| = x \leq 0 \leq -x = |x|$. □

Note that a proof by cases is only valid if the cases are *exhaustive*—that is, if every situation falls into one of the cases. (If, for example, you try to prove $\forall x \in \mathbb{R} : P(x)$ with the cases $x > 0$ and $x < 0$, you've left out $x = 0$—and your proof isn't valid!) But the cases do not need to be mutually exclusive (that is, they're allowed to overlap), as long as the cases really do cover all the possibilities; in Example 4.13, we handled the $x = 0$ case in *both* cases $x \geq 0$ and $x \leq 0$. If all possible values of x are covered by *at least* one case, and the claim is true in every case, then the proof is valid.

Here's another slightly more complex example, where we'll prove the triangle inequality for the absolute value function. (See Figure 4.2.)

Example 4.14: Triangle inequality for absolute values.

Claim: Let $x, y, z \in \mathbb{R}$. Then $|x - y| \leq |x - z| + |y - z|$.

Proof. Without loss of generality, assume that $x \leq y$. (If $y \leq x$, then we simply swap the names of x and y, and nothing changes in the claim.)

Because we're assuming $x \leq y$, we must show that $|x - z| + |y - z| \geq |x - y| = y - x$. We'll consider three cases: $z \leq x$, or $x \leq z \leq y$, or $y \leq z$. See Figure 4.18.

Case I: $z \leq x$. When $z \leq x$—that is, z is smaller than both x and y—we have

$$\begin{aligned}
|x - z| + |y - z| &\geq |y - z| && \textit{$|x - z| \geq 0$ by the definition of absolute value.} \\
&= y - z && \textit{$x \leq y$ by assumption and $z \leq x$ in Case I, so $z \leq y$ too.} \\
&\geq y - x. && \textit{$z \leq x$ in Case I, so $-z \geq -x$.}
\end{aligned}$$

Case II: $x \leq z \leq y$. When $x \leq z \leq y$—that is, z is between x and y—we have

$$\begin{aligned}
|x - z| + |y - z| &= (z - x) + |y - z| && \textit{definition of absolute value and $x \leq z$ in Case II.} \\
&= (z - x) + (y - z) && \textit{definition of absolute value and $z \leq y$ in Case II.} \\
&= y - x. && \textit{algebra/rearranging terms.}
\end{aligned}$$

Case III: $y \leq z$. When $y \leq z$—that is, z is larger than both x and y—we have

$$\begin{aligned}
|x - z| + |y - z| &\geq |x - z| && \textit{$|y - z| \geq 0$ by the definition of absolute value.} \\
&= z - x && \textit{$x \leq y$ by assumption and $y \leq z$ in Case III, so $x \leq z$ too.} \\
&\geq y - x. && \textit{$z \geq y$ in Case III.}
\end{aligned}$$

In all three cases, we've shown that $|x - z| + |y - z| \geq y - x$, so the claim follows. □

Writing tip: The phrase "without loss of generality" (which we used at the start of the proof in Example 4.14) indicates that we won't explicitly write out all the cases in the proof, because the omitted ones are virtually identical to the ones that we *are* writing out. It allows you to avoid cut-and-paste-and-search-and-replace arguments for two very similar cases.

Notice the creative demand if you choose to develop a proof by cases: you have to choose which cases to use! The proposition itself does not necessarily make obvious an appropriate choice of which different cases to use.

Proof by contrapositive

When we seek to prove a claim φ, it suffices to instead prove any proposition that is logically equivalent to φ. (For example, a proof by cases with two cases q and $\neg q$ corresponds to the logical equivalence $p \equiv (q \Rightarrow p) \wedge (\neg q \Rightarrow p)$.) A valid proof of any logically equivalent proposition can be used to prove that φ is true, but a few logical equivalences turn out to be particularly useful. A *proof by contrapositive* is a very common proof technique that relies on this principle:

Definition 4.22: Proof by contrapositive.
To give a *proof by contrapositive* of an implication $\varphi \Rightarrow \psi$, we instead give a proof of the implication $\neg\psi \Rightarrow \neg\varphi$.

Recall from Section 3.4.3 that an implication $p \Rightarrow q$ is logically equivalent to its *contrapositive* $\neg q \Rightarrow \neg p$. (An implication is true unless its antecedent is true and its conclusion is false, so $\neg q \Rightarrow \neg p$ is true unless $\neg q$ is true and $\neg p$ is false, which is precisely when $p \Rightarrow q$ is false.) Here are two small examples of proofs using the contrapositive, one about absolute values and one about rational numbers:

Example 4.15: The sum of the absolute values vs. the absolute value of the sum.

Claim: If $|x| + |y| \neq |x + y|$, then $xy < 0$.

Proof. We'll prove the contrapositive:

$$\text{If } xy \geq 0, \text{ then } |x| + |y| = |x + y|. \qquad (*)$$

To prove $(*)$, assume the antecedent; that is, assume that $xy \geq 0$. We must prove $|x| + |y| = |x + y|$. Because $xy \geq 0$, there are two cases: either both $x \geq 0$ and $y \geq 0$, or both $x \leq 0$ and $y \leq 0$.

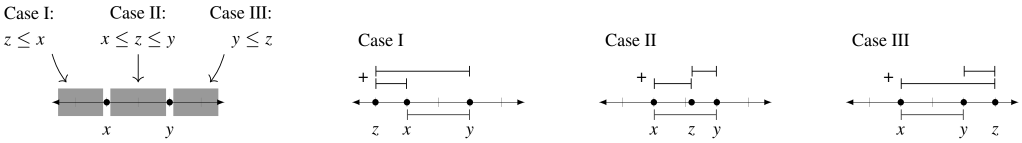


Figure 4.18 The three cases for Example 4.14: z can fall to the left of x, between x and y, or to the right of y. In each case, we argue that the sum of the lengths of the marked lines is at least $y - x$.

Case I: $x \geq 0$ and $y \geq 0$. Then $|x| + |y| = x + y$, by the definition of absolute value. And $|x+y| = x+y$ too, because $x \geq 0$ and $y \geq 0$ implies that $x + y \geq 0$ as well.

Case II: $x \leq 0$ and $y \leq 0$. In this case $|x| + |y| = -x + -y$, again by the definition of absolute value. And $|x+y| = -(x+y) = -x + -y$ too, because $x \leq 0$ and $y \leq 0$ implies that $x + y \leq 0$ as well. □

Writing tip: Help your reader figure out what's going on! If you're going to use a proof by contrapositive, *say you're using a proof by contrapositive!* Don't leave 'em guessing. This tip applies for all proof techniques: your job is to convince your reader, so be kind and informative to your reader.

Example 4.16: Irrational quotients have an irrational numerator or denominator.

Claim: Let $y \neq 0$. If $\frac{x}{y}$ is irrational, then either x is irrational or y is irrational (or both).

Proof. We will prove the contrapositive:

$$\text{If } x \text{ is rational and } y \text{ is rational, then } \tfrac{x}{y} \text{ is rational.} \qquad (\dagger)$$

(Note that, by De Morgan's Laws, $\neg$ (x is irrational or y is irrational) is equivalent to x being rational and y being rational.)

To prove (†), assume the antecedent—that is, assume that x is rational and y is rational. By definition, then, there exist four integers n_x, n_y, $d_x \neq 0$, and $d_y \neq 0$ such that $x = \frac{n_x}{d_x}$ and $y = \frac{n_y}{d_y}$. Thus $\frac{x}{y} = \frac{n_x d_y}{d_x n_y}$. (By the assumption that $y \neq 0$, we know that $n_y \neq 0$, and thus $d_x n_y \neq 0$.) Both the numerator and denominator are integers, so $\frac{x}{y}$ is rational. □

Of course, you can always reuse previous results in any proof—and Example 4.11 is particularly useful for the claim in Example 4.16. Here's a second, shorter proof:

Example 4.17: Irrational quotients, Version B.

Claim: Let $y \neq 0$. If $\frac{x}{y}$ is irrational, then either x is irrational or y is irrational (or both).

Proof. We prove the contrapositive. Assume that x and y are rational. By definition, then, $y = \frac{n}{d}$ for some integers n and $d \neq 0$. Therefore $\frac{1}{y} = \frac{d}{n}$ is rational too. (By the assumption that $y \neq 0$, we know that $n \neq 0$.) But $\frac{x}{y} = x \cdot \frac{1}{y}$, and both x and $\frac{1}{y}$ are rational. Therefore Example 4.11 implies that $\frac{x}{y}$ is rational too. □

Here's one more example of a proof that uses the contrapositive. When proving an "if and only if" statement $\varphi \Leftrightarrow \psi$, we can instead give proofs of both $\varphi \Rightarrow \psi$ and $\psi \Rightarrow \varphi$, because $\varphi \Leftrightarrow \psi$ and $(\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$ are logically equivalent. This type of proof is sometimes called a *proof by mutual implication.* (We can also prove $\varphi \Leftrightarrow \psi$ by giving a chain of logically equivalent statements that transform φ into ψ, but it is often easier to prove one direction at a time.) Here's an example of a proof by mutual implication, which also uses the contrapositive to prove one of the directions:

Example 4.18: Even integers (and only even integers) have even squares.

Claim: Let n be any integer. Then n is even if and only if n^2 is even.

Proof. We proceed by mutual implication.

First, we will show that if n is even, then n^2 is even too. Assume that n is even. Then, by definition, there exists an integer k such that $n = 2k$. Therefore $n^2 = (2k)^2 = 4k^2 = 2 \cdot (2k^2)$. Thus n^2 is even too, because there exists an integer ℓ such that $n^2 = 2\ell$. (Namely, $\ell = 2k^2$.)

Second, we will show the converse: if n^2 is even, then n is even. We will instead prove the contrapositive: if n is not even, then n^2 is not even. Assume that n is not even. Then n is odd, and there exists an integer k such that $n = 2k + 1$. Therefore $n^2 = (2k + 1)^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$. Thus n^2 is odd too, because there exists an integer ℓ such that $n^2 = 2\ell + 1$. (Namely, $\ell = 2k^2 + 2k$.) □

Proofs by contradiction

The proof techniques that we've described so far establish a claim φ by arguing that φ *must be true.* Here, we'll look at the other side of the coin, and prove φ has to be true by proving that φ *cannot be false.* This approach is called a *proof by contradiction:* we prove that something impossible must happen if φ is false (that is, we prove $\neg\varphi \Rightarrow$ False); thus the assumption $\neg\varphi$ led us to an absurd conclusion, and we must reject the assumption $\neg\varphi$ and instead conclude its negation φ. (Or, as my grandfather Jay Liben (1913–2006) always used to say (really!): "If the conclusion is obviously false, reexamine the premises.")

Definition 4.23: Proof by contradiction.
To prove φ using a proof by contradiction, we *assume* the negation of φ and derive a *contradiction*; that is, we assume $\neg\varphi$ and prove False.

(This proof technique is based on the logical equivalence of φ and the proposition $\neg\varphi \Rightarrow$ False. A proof by contradiction is also called *reductio ad absurdum,* which is Latin for "reduction to an absurdity.")

We used a proof by contradiction in Lemma 4.16: to show that two particular sets X and Y were disjoint, we assumed that an element $z \in X \cap Y$ *did* exist (that is, we assumed that X and Y were *not* disjoint), and we showed that this assumption led to a violation of the assumptions in the definitions of X and Y. Here's another small example:

Example 4.19: Trying to solve $15x + 111y = 55057$ for integers x and y.

Claim: Suppose $15x + 111y = 55057$, for two real numbers x and y. Then either x or y (or both) is not an integer.

Proof. Suppose not: that is, suppose that x and y are integers with $15x + 111y = 55057$. But $15x + 111y = 3 \cdot (5x + 37y)$, so $\frac{55057}{3} = 5x + 37y$. But then, by the assumption that x and y are integers,

$\frac{55057}{3}$ must therefore be an integer, because $5x + 37y$ is—but $\frac{55057}{3} = 18352.333\cdots \notin \mathbb{Z}$. Therefore the assumption that both $x \in \mathbb{Z}$ and $y \in \mathbb{Z}$ was false, and at least one of x and y must be nonintegral. □

Writing tip: It's always a good idea to help your reader with "signposts" in your writing. In a proof by contradiction, announce at the outset that you're assuming $\neg\varphi$ for the purposes of deriving a contradiction; when you reach a contradiction, *say* that you've reached a contradiction, and declare that therefore the assumption $\neg\varphi$ was false, and φ is true.

Here is another example of a proof by contradiction, for a classical result showing that there are numbers that aren't rational:

Example 4.20: The irrationality of $\sqrt{2}$.

Claim: $\sqrt{2}$ is not rational.

Proof. We proceed by contradiction.

Assume that $\sqrt{2}$ is rational. Therefore, by the definition of rationality, there exist integers n and $d \neq 0$ such that $\frac{n}{d} = \sqrt{2}$, where n and d are in lowest terms (that is, where n and d have no common divisors).

Squaring both sides yields that $\frac{n^2}{d^2} = 2$, and therefore that $n^2 = 2d^2$. Because $2d^2$ is even, we know that n^2 is even. Therefore, by Example 4.18 ("n is even if and only if n^2 is even") we have that n is itself even.

Because n is even, there exists an integer k such that $n = 2k$, which implies that $n^2 = 4k^2$. Thus $n^2 = 4k^2$ and $n^2 = 2d^2$, so $2d^2 = 4k^2$ and $d^2 = 2k^2$. Hence d^2 is even, and—again using Example 4.18—we have that d is even.

But now we have a contradiction: we assumed that $\frac{n}{d}$ was in lowest terms, but we have now shown that n and d are both even! Thus the original assumption that $\sqrt{2}$ was rational was false, and we can conclude that $\sqrt{2}$ is irrational. □

Note again the structure of this proof: *suppose that* $\sqrt{2}$ is rational; *therefore* we can write $\sqrt{2} = \frac{n}{k}$ where n and k have no common divisors, and (a few steps later) *therefore* n and k are both even. Because n and k cannot *both* have no common divisors *and* also both be even, we've derived an absurdity. The only way we could have gotten to this absurdity is via our assumption that $\sqrt{2}$ was rational—so we conclude that this assumption must have been false, and therefore $\sqrt{2}$ is irrational.

(Exercise 4.58 asks you to adapt the proof in Example 4.20 to show that the *cube* root of 2 is also irrational, and, similarly, Exercise 4.59 asks you to use an analogous argument to show that $\sqrt{3}$ is also irrational. Meanwhile, Exercise 4.104 asks you to figure out what's wrong with an attempted adaptation of the proof in Example 4.20 to show that $\sqrt{4}$ is irrational—which it isn't, as $\sqrt{4} = 2$ is a rational number.)

Note that, when you're trying to prove an implication $\varphi \Rightarrow \psi$, a proof by contrapositive has some similarity to a proof by contradiction:

- in a proof by contrapositive, we prove $\neg\psi \Rightarrow \neg\varphi$, by assuming $\neg\psi$ and proving $\neg\varphi$.

- in a proof by contradiction, we prove False under the assumption $\neg(\varphi \Rightarrow \psi)$—that is, under the assumption that $\varphi \wedge \neg\psi$. (Note that there's an extra creative demand here: you have to figure out which contradiction to derive—something that's not generally made immediately clear by the given claim.)

Proofs by contrapositive are generally preferred over proofs by contradiction when a proof by contrapositive is possible. A proof by contradiction can be hard to follow because we're asking the reader to temporarily accept an assumption that we'll later show to be false, and there can be a mental strain in keeping track of what's been assumed and what was previously known. (Notice that the claim in Example 4.20 wasn't an implication, so a proof by contrapositive wasn't an option. The proofs of Lemma 4.16 (from Section 4.2) and Example 4.19, though, could have been rephrased as proofs by contrapositive.)

Proofs by construction and disproofs by counterexample

So far we've concentrated on proofs of universally quantified statements, where you are asked to show that some property holds for all elements of a given set. (Every example proof in this section, except the two proofs by contradiction about the irrationality of $\sqrt{2}$ and the infinitude of primes, were proofs of a "for all" statement—and, actually, even those two claims could have been phrased as universal quantifications. For example, we could have phrased Example 4.20 as the following claim: for all integers n and $d \neq 0$, we have $n \neq d \cdot \sqrt{2}$.) Sometimes you'll confront a universally quantified statement that's false, though. The easiest way to prove that $\forall x \in S : P(x)$ is false is using a *disproof by counterexample:*

Definition 4.24: Disproof by counterexample.
A *counterexample* to a claim $\forall x \in S : P(x)$ is a particular element $y \in S$ such that $P(y)$ is false. A *disproof by counterexample* shows that $\forall x \in S : P(x)$ is false by identifying a counterexample $y \in S$ and providing a proof that $P(y)$ is false.

Finding a counterexample for a claim requires creativity: you have to think about why a claim might not be true, and then try to construct an example that embodies that reason.

Problem-solving tip: One way you might try to identify counterexample to a claim is by writing a program: write a loop that tries a bunch of examples; if you ever find one for which the claim is false, then you've found a counterexample. Just because you haven't found a counterexample with your program doesn't mean that there isn't one—unless you've tried *all* the elements of S—but if you *do* find a counterexample, it's still a counterexample no matter how you found it!

Here is a small example:

Example 4.21: Unique sums of squares.

Claim: Let n be a positive integer such that $n = a^2 + b^2$ for positive integers a and b. Then n cannot be expressed as the sum of the squares of two positive integers except a and b. (Alternatively, this claim could be written more tersely as: *No positive integer is expressible in two different ways as the sum of two perfect squares.)*

The claim is false, and we will prove that it is false by counterexample. We can start trying some examples. One easy class of potential counterexamples is $a^2 + 1$ for an integer a. Although $1^2 + 1^2 = 2$ can't be expressed a different way, what about 5? 10? 17? 26? 37? 50? 65? 82? By testing these examples, we find that 65 is a counterexample to the claim. Observe that $1^2+8^2 = 1+64 = 65$, and $4^2+7^2 = 16+49 = 65$. Another counterexample is 50, as $50 = 5^2 + 5^2 = 1^2 + 7^2$.

What about when you're asked to prove an existential claim $\exists x : P(x)$? One approach is to prove the claim by contradiction: you assume $\forall x : \neg P(x)$, and then derive some contradiction. This type of proof is called *nonconstructive*: you have proven that an object with a certain property must exist, but you haven't actually described a particular object with that property. In contrast, a *proof by construction* actually identifies a specific object that has the desired property:

Definition 4.25: Proof by construction.
A *constructive proof* or *proof by construction* for a claim $\exists x \in S : P(x)$ actually builds an object satisfying the property P: first, we identify a particular element $y \in S$; and, second, we prove $P(y)$.

For example, here's a claim that we'll prove twice, once nonconstructively and once constructively:

Example 4.22: The last two digits of some squares.

Claim: There exist distinct integers $x, y \in \{1901, 1902, \ldots, 2024\}$ such that the last two digits of x^2 and y^2 are the same. (In other words, $x^2 \bmod 100 = y^2 \bmod 100$.)

Nonconstructive proof. There are 124 different numbers in the set $\{1901, 1902, \ldots, 2014\}$. There are only 100 different possible values for the last two digits of numbers. Thus, because there are 114 elements assigned to only 100 categories, there must be some category that contains more than one element. □

Constructive proof. Let $x = 1986$ and $y = 1964$. Both numbers' squares have 96 as their last two digits: $1986^2 = 3{,}944{,}196$ and $1964^2 = 3{,}857{,}296$. □

The nonconstructive proof in Example 4.22 makes use of the so-called *pigeonhole principle,* which we'll see in Section 9.3.3: if there are more things (in this case, numbers between 1901 and 2024) than there are categories of things (the possible values for the last two digits) then there must be at least two things that are in the same category.

It's generally preferable to give a constructive proof when you can. A constructive proof is sometimes harder to develop than a nonconstructive proof, though: it may require more insight about the kind of object that can satisfy a given property, and more creativity in figuring out how to actually construct that object.

Taking it further: A constructive proof of a claim is generally more satisfying for the reader than a nonconstructive proof. A proof by contradiction may leave a reader unsettled—okay, the claim is true, but what can we do with that?—while a constructive proof may be useful in designing an algorithm, or it may suggest further possible claims to try to prove. (There's even a school of thought in logic called *constructivism* that doesn't count a proof by contradiction as a proof!)

4.3.2 Some Brief Thoughts about Proof Strategy

So far in this section, we've concentrated on developing a toolbox of proof techniques. But when you're confronted with a new claim and asked to prove it, you face a difficult task in figuring out which approach to take. (It's even harder if you're asked to formulate a claim and *then* prove it!) As we discussed in Chapter 3, there's no formulaic approach that's guaranteed to work—you must be creative, open-minded, persistent. You will have to accept that you will explore approaches that end up being dead ends. This section will give a few brief pointers about proof strategy—some things to try when you're just starting to attack a new problem. We'll start with some concrete advice in the form of a three-step plan for proofs, largely inspired by an outstanding book by George Pólya (1887–1985) [102].

Step 1: Understand what you're trying to do. Read the statement that you're trying to prove. Reread it. What are the assumptions? What is the desired conclusion? (That is, what are you trying to prove under the given assumptions?) Remind yourself of any unfamiliar notation or terminology. Pick a simple example and make sure the alleged theorem holds for your example. (If not, either you've misunderstood something or the claim is false.) Reread the statement again.

If you're not given a specific claim—for example, you're asked to prove or disprove a given statement, or if you're asked for the "best possible" solution to a problem—then it's harder but even more important to understand what you're trying to do. Play around with some examples to generate a sense of what might be plausibly true. Then try to form a conjecture based on these examples or the intuition that you've developed.

Step 2: Do it. Now that you have an understanding of the statement that you're trying to prove, it's time to actually prove it. You might start by trying to think about slightly different problems to help grant yourself insight about this one. Are there results that you already know that "look similar" to this one? Can you solve a more general problem? Make the premises look as much like the conclusion as possible. Expand out the definitions; write down what you know and what you have to derive, in primitive terms. Can you derive some facts from the given hypotheses? Are there easier-to-prove statements that would suffice to prove the desired conclusion?

Look for a special case: add assumptions until the problem is easy, and then see if you can remove the extra assumptions. Restate the problem. Restate it again. Make analogies to problems that you've already solved. Could those related problems be directly valuable? Or could you use a similar technique to what you used in that setting? Try to use a direct proof first; if you're finding it difficult to construct a direct proof of an implication, try working on the contrapositive instead. If both of these approaches fail, try a proof by contradiction.

Problem-solving tip: If you're totally stuck in attempting to prove a statement true, switch to trying to prove it false. If you succeed, you're done—or, by figuring out why you're struggling to construct a counterexample, you may figure out how to prove that the statement is true.

When you have a candidate plan of attack, try to execute it. If there's a picture that will help clarify the steps in your plan, draw it. Sketch out the "big" steps that you'd need to make the whole proof work. Make sure they fit together. Then crank through the details of each big step. Do the algebra. Check the algebra. If it all works out, great! If not, go back and try again. Where did things go off the rails, and can you fix them?

Think about how to present your proof; then actually write it. Note that what you did in *figuring out* how to prove the result might or might not be the best way to *present* the proof.

Step 3: Think about what you've done. Check to make sure your proof is reasonable. Did you actually use all the assumptions? (If you didn't, do you believe the stronger claim that has the smaller set of assumptions?) Look over all the steps of your proof. Turn your internal skepticism dial to its maximum, and reread what you just wrote. Ask yourself *Why?* as you think through each step. Don't let yourself get away with anything.

Problem-solving tip: Check your work! If your claim says something about a general n, test it for $n = 1$. Compare your answer to a plot, or the output of a quick program.

After you're satisfied that your proof is correct, work to improve it. Can you strengthen the result by making the conclusion stronger or the assumptions weaker? Can you make the proof constructive? Simplify the argument as much as you can. Are there unnecessary steps? Are there unnecessarily complex steps? Are there subclaims that would be better as separate lemmas?

Combining the steps. It's important to be willing to move back and forth among these steps. You'll try to prove a claim φ, and then you'll discover a counterexample to φ—so you go back and modify the claim to a new claim φ' and try to prove φ' instead. You'll formulate a draft of a proof of φ' but discover a bug when you check your work while reflecting on the proof. You'll go back to proving φ', fix the bug, and discover a new proof that's bugfree. You'll think about your proof and realize that it didn't use all the assumptions of φ', so you'll formulate a stronger claim φ'' and then go through the proof of φ'' and reflect again about the proof.

Taking it further: One of the most famous—and prolific!—mathematicians of modern times was Paul Erdős (1913–1996), a Hungarian mathematician who wrote literally thousands of papers over his career, on a huge range of topics. Erdős used to talk about a mythical "Book" of proofs, containing the perfect proof of every theorem (the clearest, the most elegant—the best!). See p. 4-46 for more about The Book, and about Paul Erdős himself.

4.3.3 Some Brief Thoughts about Writing Good Proofs

When you're writing a proof, it's important to remember that you are *writing*. Proofs, like novels or persuasive essays, form a particular genre of writing. Treat writing a proof with the same care and attention that you would give to writing an essay.

Writing tip: Draft. Write. Edit. *Rewrite.*

Writing tip: In writing a proof, keep your reader informed about the status of every sentence. And make sure that everything you write *is* a sentence. For example, every sentence contains a verb. (Note that a symbol like "=" is read as "is equal to" and *is* a verb.) Is the sentence an assumption? A goal? A conclusion? Annotate your sentences with signaling words and phrases to make it clear what each statement is doing. For example, introduce statements that follow logically from previous statements with words like *hence*, *thus*, *so*, *therefore*, and *then*.

Make your argument self-contained; include definitions of all variables and all nonstandard notation. State all assumptions, and explain your notation. Choose your notation and terminology carefully; name your variables well. Here's an example, of doing it well and doing it poorly. (Thanks to Josh Davis for suggesting Examples 4.23 and 4.24.)

Example 4.23: A theorem, stated poorly.

Theorem: $a^2 + b^2 = c^2$.

This formulation is a *terrible* way of phrasing the theorem: the reader has no idea what a, b, and c *are*, or even that the theorem has anything whatsoever to do with geometry. (Example 4.23 is an attempt to state the Pythagorean Theorem, from geometry, which states that the square of the length of the hypotenuse of a right triangle is equal to the sum of the squares of the lengths of its legs.) Here's a much better statement of the Pythagorean Theorem:

Example 4.24: Pythagorean Theorem, stated well.

Theorem: Let a and b denote the lengths of the legs of a right triangle, and let c denote the length of its hypotenuse. Then $a^2 + b^2 = c^2$.

If you are worried that your audience has forgotten the geometric terminology from this statement, then you might add the following clarification:

As reminder from geometry, a *right triangle* is a 3-sided polygon with one 90° angle, called a *right angle*. The two sides adjacent to the right angle are called *legs* and the third side is called the *hypotenuse*. Figure 4.19 shows an example of a right triangle. Here the legs are labeled a and b, and the hypotenuse is labeled c. As is customary, the right angle is marked with the special square-shaped symbol □.

Because the "standard" phrasing of the Pythagorean Theorem—which you might have heard in high school—calls the length of the legs a and b and the length of the hypotenuse c, we use the standard variable names. Calling the leg lengths θ and ϕ and the hypotenuse r would be hard on the reader; conventionally in

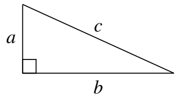


Figure 4.19 A right triangle.

geometry θ and ϕ are angles, while r is a radius. *Whenever you can, make life as easy as possible for your reader.* (By the way, we'll prove the Pythagorean Theorem in Example 4.30, and you'll prove it again in Exercise 4.75.)

Above all, remember that your primary goal in writing is communication. Just as when you are programming, it is possible to write two solutions to a problem that both "work," but which differ tremendously in readability. Document! Comment your code; explain *why* this statement follows from previous statements. Make your proofs—and your code!—a pleasure to read.

Computer Science Connections

Paul Erdős, "The Book," and Erdős Numbers

After you've completed a proof of a claim—and after you've celebrated completing it—you should think again about the problem. In programming, there are often many fundamentally different algorithms to solve a particular problem; in proofs, there are often many fundamentally different ways of proving a particular theorem. And, just as in programming, some approaches will be more elegant, more clear, or more efficient than others.

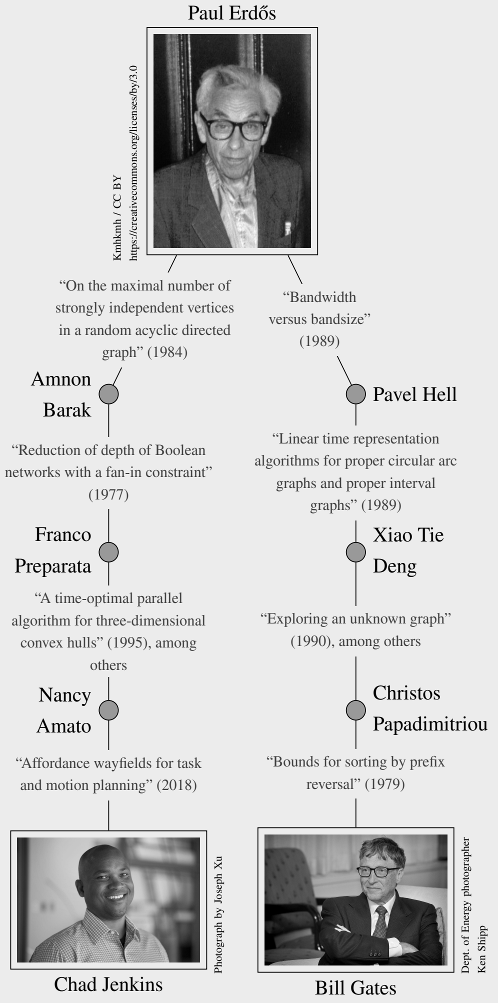

Paul Erdős
Kmhkmh / CC BY https://creativecommons.org/licenses/by/3.0
"On the maximal number of strongly independent vertices in a random acyclic directed graph" (1984)
"Bandwidth versus bandsize" (1989)
Amnon Barak
Pavel Hell
"Reduction of depth of Boolean networks with a fan-in constraint" (1977)
"Linear time representation algorithms for proper circular arc graphs and proper interval graphs" (1989)
Franco Preparata
Xiao Tie Deng
"A time-optimal parallel algorithm for three-dimensional convex hulls" (1995), among others
"Exploring an unknown graph" (1990), among others
Nancy Amato
Christos Papadimitriou
"Affordance wayfields for task and motion planning" (2018)
"Bounds for sorting by prefix reversal" (1979)
Photograph by Joseph Xu
Dept. of Energy photographer Ken Shipp
Chad Jenkins
Bill Gates

Figure 4.20 An example of Erdős numbers. The University of Michigan roboticist Chad Jenkins and Microsoft co-founder Bill Gates both have an Erdős number of at most 4, having written a paper with someone who wrote a paper with someone who wrote a paper with someone who wrote a paper with Paul Erdős.

Paul Erdős (1913–1996), a world-famous mathematician who published approximately 1500 papers (including papers on math, physics, and computer science), used to talk about "The Book" of proofs. "The Book" contains the ideal proof of each theorem—the most elegant, insightful, and beautiful proof. (If you believe in God, then The Book contains God's proofs.) Proving a theorem is great, but giving a "book proof" is even better. (There's even a non-metaphorical book called *Proofs from The Book* that collects some of the most elegant known proofs of some theorems [6].)

Erdős was one of the most respected mathematicians of his time—and one of the most eccentric, too. (He forswore most material possessions, and instead traveled the world, crashing in the guest rooms of his research collaborators for months at time.) Because of Erdős's prolific publication record and his great respect from the research community, a measure of a certain type of fame sprung up around him. A researcher's *Erdős number* is 1 if she has coauthored a published paper with Erdős; it's 2 if she has coauthored a paper with someone with an Erdős number of one; and so forth. (See Figure 4.20.) If you're more of a movie person than a peripatetic mathematician person, then you may be familiar with a similar notion from the entertainment world, the so-called *Bacon game,* where the goal is to connect a given actor to Kevin Bacon via the shortest chain of intermediaries, where two actors are linked if they have appeared together in a movie.

Although Erdős numbers themselves are really nothing more than a nerdy source of amusement, the ideas underlying them are fundamental in *graph theory*, the subject of Chapter 11. A closely related topic is the *small-world phenomenon*, also known as "six degrees of separation" (a phrase that originated from an important early paper by the social psychologist Stanley Milgram [88]), the principle that almost any two people are likely to be connected by a short chain of intermediate friends. This phenomenon has spawned a massive amount of recent research by computer scientists, who have begun working to analyze questions about human behavior that have only become visible in a social-media era in which it is now possible to study collective decision making on a massive scale.

Computer Science Connections

The Four-Color Theorem (and Massive Computer-Generated Proofs)

As we've said, what we mean by a "proof" is an argument that convinces the audience that the claim is true. But there are some kinds of arguments that may test your intuition about what counts (or should count) as convincing.

To take one example, consider the fact that *Checkers is a draw when both players play optimally.* The proof of this fact hinges on showing that the software system Chinook will never lose a checkers game (see p. 3-54). (So if Chinook plays against itself, the game *must* be a draw.) The relevant part of this story is that the fact that Chinook never loses was established via massive computation to perform a large-scale search of the checkers game tree [112]. Is that argument convincing? Can such an argument *ever* be convincing? It's clear that no human reader can accommodate the 5×10^{20} checkers board positions in their brain, so it's not convincing in the sense that a reader would be able to verify every step of the argument. But, on the other hand, a reader could potentially be convinced that Chinook's *code* is correct, even if the *output* is too big for a reader to find convincing.

The philosophical question about whether a large-scale computer-generated proof "counts" actually as a proof first arose in the late 1970s, in the context of the *Four-Color Theorem:*

The Four-Color Theorem: Any "map" of contiguous geometric regions can be colored using four colors so that no two adjacent regions share the same color.

(See Figure 4.21.) Two quick notes about the theorem statement: first, *adjacent* means sharing a positive-length border; two regions meeting at a point don't need different colors. Second, the requirement of regions being *contiguous* means the map can't require two disconnected regions (like the Lower 48 States and Alaska) to get the same color.

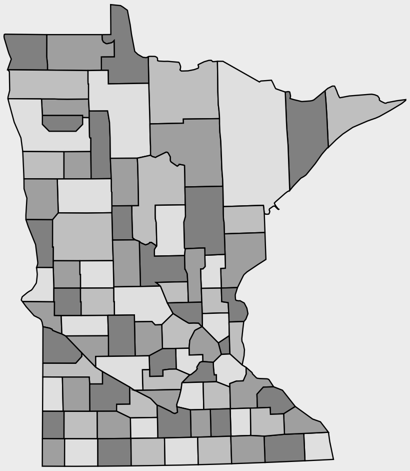


Figure 4.21 A four-colored map of the 87 counties in Minnesota. No two counties that share a border are shaded with the same color.

The first accepted proof of four-color theorem was given by Kenneth Appel (1932–2013) and Wolfgang Haken (b. 1928) in 1976 [8]. (Well, almost the first: a purported proof by Alfred Kempe (1849–1922) was published in 1879 [71] and was generally accepted—but a full decade later a fatal bug was found!) Appel and Haken's proof was fundamentally computational. Appel and Haken first identified a set of 1936 different map configurations and proved (in the traditional way, by giving a convincing argument) that, if the four-color theorem were false, it would fail on one of these 1936 configurations. They then wrote a computer program that showed how to color each one of these 1936 configurations using only four colors. The theorem follows ("if there were a counterexample at all, there'd be a counterexample in one of the 1936 cases—and there are no counterexamples in the 1936 cases").

A great deal of controversy followed the publication of Appel and Haken's work. Some mathematicians felt strongly that a proof that's too massive for a human to understand is not a proof at all. Others were happy to accept the proof, particularly because the four-colorability question had been posed, and remained unresolved, for centuries.

Computer scientists, by our nature, tend to be more accepting of computational proof than mathematicians—but there are still plenty of interesting questions to ponder. For example, as we discussed on p. 3-54, some errors in the *execution* of the code that generates Chinook's proof are known to have occurred, simply because hardware errors happen at a high enough rate that they will arise in a computation of this size. Thus bit-level corruption may have occurred, without 100% correction, in Chinook's proof that checkers is a draw under optimal play. So is Chinook's "proof" really a proof? (Of course, there are also plenty of human-generated purported proofs that contain errors!)

EXERCISES

Consider the following claims about rational numbers. Prove (or disprove, as indicated) each of them.

4.33 Prove that, if x and y are rational numbers, then $x - y$ is also rational.

4.34 Prove that, if x and y are rational numbers and $y \neq 0$, then $\frac{x}{y}$ is also rational.

4.35 One of the following statements is true and one is false:

- If xy and x are both rational, then y is too.
- If $x - y$ and x are both rational, then y is too.

Decide which statement is true and which is false, and give proof/disproof of both.

Prove the following claims about divisibility.

4.36 Prove that the binary representation of any odd integer ends with a 1.

4.37 Prove that a positive integer n is divisible by 5 if and only if its last digit is 0 or 5.

4.38 Let k be any positive integer. Prove that any positive integer n is divisible by 2^k if and only if its last k digits are divisible by 2^k. (This exercise is a generalization of Example 4.10.)

4.39 Let n be any integer. Prove by cases that $n^3 - n$ is evenly divisible by 3.

4.40 Let n be any integer. Prove by cases that $n^2 + 1$ is *not* evenly divisible by 3.

4.41 Prove that $|x| + |y| \geq |x + y|$ for any real numbers x and y.

4.42 Prove that $|x| - |y| \leq |x - y|$ for any real numbers x and y.

4.43 Prove that the product of the absolute values of x and y is equal to the absolute value of their product for any real numbers x and y. That is, prove that $|x| \cdot |y| = |x \cdot y|$.

4.44 Suppose that $x, y \in \mathbb{R}$ satisfy $|x| \leq |y|$. Prove that $\frac{|x+y|}{2} \leq |y|$.

4.45 Let A and B be sets. Prove that $A \times B = B \times A$ if and only if $A = \varnothing$ or $B = \varnothing$ or $A = B$. Prove the result by mutual implication, where the proof of the $\Rightarrow$ direction proceeds by contrapositive.

Let $x \geq 0$ and $y \geq 0$ be arbitrary real numbers. The arithmetic mean *of x and y is $(x + y)/2$, their average. The* geometric mean *of x and y is $\sqrt{xy}$.*

4.46 First, a warm-up exercise: prove that $x^2 \geq 0$ for any real number x. *(Hint: no, it's not supposed to be hard.)*

4.47 Prove the *Arithmetic Mean–Geometric Mean inequality*: for $x, y \in \mathbb{R}^{\geq 0}$, we have $\sqrt{xy} \leq (x + y)/2$. *(Hint: $(x - y)^2 \geq 0$ by Exercise 4.46. Use algebraic manipulation to make this inequality look like the desired one.)*

4.48 Prove that the arithmetic mean and geometric mean of x and y are equal if and only if $x = y$.

In Chapter 2, when we defined square roots, we introduced Heron's method, *a first-century algorithm to compute $\sqrt{x}$ given x. See p. 2-22, or Figure 4.22 for a reminder. Here you'll prove two properties that help establish why this algorithm correctly computes square roots. (The second property that you'll prove—Exercise 4.50—shows that Heron's method improves its estimate of $\sqrt{x}$ in*

```
Input: A positive real number $x$
Output: A real number $y$ where $y^2 \approx x$

Let $y_0$ be arbitrary, and let $i := 0$.
while $(y_i)^2$ is too far away from $x$
    let $y_{i+1} := \frac{y_i + \frac{x}{y_i}}{2}$, and let $i := i + 1$.
return $y_i$
```

Figure 4.22 A reminder of Heron's method for computing square roots.

every iteration. We won't *prove "how much" improvement Heron's method achieves in an iteration, or even that this algorithm is converging to the correct answer—let alone quickly!—but, in fact, it is.)*

4.49 Let's assume that we start with an overestimate: assume that $y_0 \geq \sqrt{x}$. Prove that, for every $i \geq 1$, we have $y_i \geq \sqrt{x}$. In other words, prove that if $y \geq \sqrt{x}$ then $(y + \frac{x}{y})/2 \geq \sqrt{x}$ too.

4.50 Continue to assume that we started with an overestimate: suppose that $y > \sqrt{x}$. Prove that $\frac{x}{y}$ is closer to $\sqrt{x}$ than y is—that is, prove that $|\frac{x}{y} - \sqrt{x}| < |y - \sqrt{x}|$. Now, using this result and Exercise 4.44, prove that y_{i+1} as computed in Heron's Method is closer to $\sqrt{x}$ than y_i, as long as $y_i > \sqrt{x}$. *(Hint: for the first part, show that* $|y - \sqrt{x}| - |\sqrt{x} - \frac{x}{y}| > 0$*.)*

Prove the following claims using a proof by contrapositive.

4.51 Let $n \in \mathbb{Z}^{\geq 0}$. If $n \bmod 4 \in \{2, 3\}$, then n is not a perfect square.

4.52 Let n and m be integers. If nm is not evenly divisible by 3, then neither n nor m is evenly divisible by 3. (In fact, the converse is true too, but you don't have to prove it.)

4.53 Let $n \in \mathbb{Z}^{\geq 0}$. If $2n^4 + n + 5$ is odd, then n is even.

Prove the following claims using a proof by mutual implication, using a proof by contrapositive for one direction.

4.54 Let n be any integer. Then n^3 is even if and only if n is even.

4.55 Let n be any integer. Then n is divisible by 3 if and only if n^2 is divisible by 3.

Prove the following claims using a proof by contradiction.

4.56 Let x, y be positive real numbers. If $x^2 - y^2 = 1$, then x or y (or both) is not an integer.

4.57 Suppose $12x + 3y = 254$, for real numbers x and y. Then either x or y (or both) is not an integer.

4.58 Adapt Example 4.20 to prove that $\sqrt[3]{2} = 2^{1/3}$ is irrational. (You may find Exercise 4.54 helpful.)

4.59 Adapt Example 4.20 to prove that $\sqrt{3}$ is irrational. (You may find Exercise 4.55 helpful.)

4.60 Consider an array $A[1 \ldots n]$. A value x is called a *strict majority element of A* if strictly more than half of the elements in A are equal to x—in other words, if

$$\Big|\{i \in \{1, 2, \ldots, n\} : A[i] = x\}\Big| > \frac{n}{2}.$$

Give a proof by contradiction that every array has at most one strict majority element.

In Example 4.11, Exercise 4.33, and Exercise 4.34, we proved that if x and y are both rational, then so are all three of xy, $x - y$, and $\frac{x}{y}$. The converse of each of these three statements is false. ***Disprove*** *the following claims by giving counterexamples:*

4.61 If xy is rational, then x and y are rational.

4.62 If $x - y$ is rational, then x and y are rational.

4.63 If $\frac{x}{y}$ is rational, then x and y are rational.

4.64 In Example 4.21, we disproved the following claim by giving a counterexample:

Claim 1: No positive integer is expressible in two different ways as the sum of two perfect squares.

Let's consider a related claim that is not disproved by our counterexamples from Example 4.21:

Claim 2: No positive integer is expressible in *three* different ways as the sum of two perfect squares.

Disprove Claim 2 by giving a counterexample.

4.65 Leonhard Euler, an 18th-century Swiss mathematician to whom the idea of an abstract formal model of networks (graphs; see Chapter 11) is due, made the observation that the polynomial

$$f(n) = n^2 + n + 41$$

yields a prime number when it's evaluated for many small integers n: for example, $f(0) = 41$ and $f(1) = 43$ and $f(2) = 47$ and $f(3) = 53$, and so forth. Prove or disprove the following claim: *the function $f(n)$ yields a prime for every nonnegative integer n.*

4.4 Some Examples of Proofs

> Few things are harder to put up with than the annoyance of a good example.
>
> Mark Twain (1835–1910)
> *Pudd'nhead Wilson* (1894)

We've now catalogued a variety of proof techniques, discussed some strategies for proving novel statements, and described some ideas about presenting proofs well. Section 4.3 illustrated some proof techniques with a few simple examples each, entirely about numbers and arithmetic. In this section, we'll give a few "bigger"—and perhaps more interesting!—examples of theorems and proofs.

4.4.1 A Proof about Propositional Logic: Conjunctive/Disjunctive Normal Form

We'll start with a result about propositional logic, namely showing that any proposition is logically equivalent to another proposition that has a very specific structure. Recall the definitions of *conjunctive* and *disjunctive normal form* from Definitions 3.16 and 3.17:

Definition 4.26: Reminder: Conjunctive/Disjunctive Normal Form.
In propositional logic, a *literal* is a Boolean variable or its negation (like p or $\neg p$).

A proposition φ is in *conjunctive normal form (CNF)* if φ is the conjunction of one or more *clauses*, where each *clause* is the disjunction of one or more literals.

A proposition φ is in *disjunctive normal form (DNF)* if φ is the disjunction of one or more *clauses*, where each *clause* is the conjunction of one or more literals.

Here are two small examples of CNF and DNF:

$$\underbrace{(\neg p \vee q \vee \neg r) \wedge (\neg q \vee r)}_{\text{conjunctive normal form}} \qquad \text{and} \qquad \underbrace{(\neg p \wedge \neg q \wedge r) \vee (\neg q \wedge \neg r \wedge s) \vee (r)}_{\text{disjunctive normal form}} .$$

Back in Chapter 3, we claimed that every proposition is logically equivalent to one in CNF and one in DNF, but we didn't prove it. Here we will. First, though, let's recall an example from Chapter 3 and brainstorm a bit about how to generalize that result into the desired theorem. In Example 3.27, we converted $p \Leftrightarrow q$ into DNF as the logically equivalent proposition $(p \wedge q) \vee (\neg p \wedge \neg q)$. Note that this expression has two clauses $p \wedge q$ and $\neg p \wedge \neg q$, *each of which is true in one and only one row of the truth table.* And our full proposition $(p \wedge q) \vee (\neg p \wedge \neg q)$ is true in precisely two rows of the truth table. (See Figure 4.23.)

p	q	$p \Leftrightarrow q$	$p \wedge q$	$\neg p \wedge \neg q$	$(p \wedge q) \vee (\neg p \wedge \neg q)$
T	T	T	T	F	T
T	F	F	F	F	F
F	T	F	F	F	F
F	F	T	F	T	T

Figure 4.23 Truth table for $p \Leftrightarrow q$ and the clauses for converting it to DNF.

Can we make this idea general? Yes! For an arbitrary proposition φ, and for any particular row of the truth table for φ, we can construct a clause that's true in that row and only in that row. We can then build a DNF proposition that's logically equivalent to φ by "or"ing together each of the clauses corresponding to the rows in which φ is true. And then we're done! (Well, we're *almost* done! There is one subtle bug in this proof sketch—can you find it? We'll fix the issue in the last paragraph of the proof below.)

Theorem 4.27: All propositions are expressible in DNF (Theorem 3.19).
For any proposition φ, there exists a proposition ψ_{dnf} in disjunctive normal form such that $\varphi \equiv \psi_{\text{dnf}}$.

Proof. Let φ be an arbitrary proposition, say over the Boolean variables $p_1, \ldots, p_k$.

For any particular truth assignment f for the variables $p_1, \ldots, p_k$, we'll construct a conjunction c_f that's true under f and false under all other truth assignments. Let $x_1, x_2, \ldots, x_\ell$ be the variables assigned true by f, and $y_1, y_2, \ldots, y_{k-\ell}$ be the variables assigned false by f. Then the clause

$$c_f = x_1 \wedge x_2 \wedge \cdots \wedge x_\ell \wedge \neg y_1 \wedge \neg y_2 \wedge \cdots \wedge \neg y_{k-\ell}$$

is true under f, and c_f is false under every other truth assignment.

We can now construct a DNF proposition ψ_{dnf} that is logically equivalent to φ by "or"ing together the clause c_f for each truth assignment f that makes φ true. Build the truth table for φ, and let S_φ denote the set of truth assignments for $p_1, \ldots, p_k$ under which φ is true. If the truth assignments in S_φ are $\{f_1, f_2, \ldots, f_m\}$, then define

$$\psi_{\text{dnf}} = c_{f_1} \vee c_{f_2} \vee \cdots \vee c_{f_m}. \qquad (*)$$

Then ψ_{dnf} is true under every truth assignment f under which φ was true (because the clause c_f is true under f). And, for a truth assignment f under which φ was false, every disjunct in ψ_{dnf} evaluates to false, so the entire disjunction is false under such a f, too. Thus $\varphi \equiv \psi_{\text{dnf}}$.

There's one thing we have to be careful about: what happens if $S_\varphi = \varnothing$—that is, if φ is unsatisfiable? (This issue is the minor bug we mentioned before the theorem statement.) The construction in $(*)$ doesn't work, but it's easy to handle this case separately: we simply choose an unsatisfiable DNF proposition like $p \wedge \neg p$ as ψ_{dnf}. □

Problem-solving tip: Although we didn't phrase it as such from the beginning, our proof of Theorem 4.27 was actually a proof by cases, with two cases corresponding to φ being unsatisfiable and φ being satisfiable.Be on the lookout for special cases (like an unsatisfiable φ in Theorem 4.27), and see whether you can handle them separately from the argument for the "typical" case.

As an illustration, let's use the construction from Theorem 4.27 to transform an example proposition into disjunctive normal form:

p	q	r	$q \wedge r$	$p \Rightarrow (q \wedge r)$	
T	T	T	T	T	$p \wedge q \wedge r$
T	T	F	F	F	$p \wedge q \wedge \neg r$
T	F	T	F	F	$p \wedge \neg q \wedge r$
T	F	F	F	F	$p \wedge \neg q \wedge \neg r$
F	T	T	T	T	$\neg p \wedge q \wedge r$
F	T	F	F	T	$\neg p \wedge q \wedge \neg r$
F	F	T	F	T	$\neg p \wedge \neg q \wedge r$
F	F	F	F	T	$\neg p \wedge \neg q \wedge \neg r$

Figure 4.24 The truth table for $p \Rightarrow (q \wedge r)$.

Example 4.25: Converting $p \Rightarrow (q \wedge r)$ to DNF.
Find a proposition in DNF logically equivalent to $p \Rightarrow (q \wedge r)$.

Solution. To convert $p \Rightarrow (q \wedge r)$ to DNF, we start from the truth table, and then "or" together the propositions corresponding to each row that's marked as True. The truth table is shown in Figure 4.24. Our DNF proposition will therefore have five clauses, one for each of the five truth assignments under which this implication is true:

$$\underset{\text{TTT}}{(p \wedge q \wedge r)} \;\vee\; \underset{\text{FTT}}{(\neg p \wedge q \wedge r)} \;\vee\; \underset{\text{FTF}}{(\neg p \wedge q \wedge \neg r)} \;\vee\; \underset{\text{FFT}}{(\neg p \wedge \neg q \wedge r)} \;\vee\; \underset{\text{FFF}}{(\neg p \wedge \neg q \wedge \neg r)} \,.$$

Conjunctive normal form

Now that we've proven that we can translate any proposition into disjunctive normal form (the "or of ands"), we'll turn our attention to conjunctive normal form (the "and of ors").

Theorem 4.28: All propositions are expressible in CNF.
For any proposition φ, there exists a proposition φ_{cnf} in conjunctive normal form such that $\varphi \equiv \varphi_{\text{cnf}}$.

Though it's not initially obvious, Theorem 4.28 actually turns out to be not so hard to prove if we make use of the DNF result. The crucial idea—and, once again, it's an idea that requires some genuine creativity to come up with!—is that it's fairly simple to turn the *negation* of a DNF proposition into a CNF proposition. So, to build a CNF proposition logically equivalent to φ, we'll construct a DNF proposition that is logically equivalent to $\neg\varphi$; we can then negate that DNF proposition and use De Morgan's Laws to convert the resulting proposition into CNF. Here are the details:

Problem-solving tip: Try being lazy first! Think about whether there's a way to use a previously established result to make the current problem easier.

Proof. If φ is a tautology, the task is easy; just define $\varphi_{\text{cnf}} = p \vee \neg p$.

Otherwise, φ is a nontautology, say over the variables $p_1, \ldots, p_k$. Using Theorem 4.27, we can construct a DNF proposition ψ that is logically equivalent to $\neg\varphi$. (Note that, using our construction from

Theorem 4.27, the proposition ψ will have k literals in every clause, because $\neg\varphi$ is satisfiable.) Thus the form of ψ will be

$$\psi = (c_1^1 \wedge \cdots \wedge c_k^1) \vee (c_1^2 \wedge \cdots \wedge c_k^2) \vee \cdots \vee (c_1^m \wedge \cdots \wedge c_k^m)$$

for some $m \geq 1$, where each c_i^j is a literal. Recall that $\psi \equiv \neg\varphi$, so we also know that $\neg\psi \equiv \varphi$. Let's negate ψ:

$$\begin{aligned}
\neg\psi &= \neg\left[(c_1^1 \wedge \cdots \wedge c_k^1) \vee (c_1^2 \wedge \cdots \wedge c_k^2) \vee \cdots \vee (c_1^m \wedge \cdots \wedge c_k^m)\right] \\
&\equiv \neg(c_1^1 \wedge \cdots \wedge c_k^1) \wedge \neg(c_1^2 \wedge \cdots \wedge c_k^2) \wedge \cdots \wedge \neg(c_1^m \wedge \cdots \wedge c_k^m) \quad \textit{De Morgan's Law: } \neg(p \vee q) \equiv \neg p \wedge \neg q \\
&\equiv (\neg c_1^1 \vee \cdots \vee \neg c_k^1) \wedge (\neg c_1^2 \vee \cdots \vee \neg c_k^2) \cdots \wedge (\neg c_1^m \vee \cdots \vee \neg c_k^m).
\end{aligned}$$

De Morgan's Law: $\neg(p \wedge q) \equiv \neg p \vee \neg q$, applied once per clause

But this expression is in CNF once we remove any doubly negated literals—that is, we replace any occurrences of $\neg\neg p$ by p instead. Thus we've constructed a proposition in conjunctive normal form that's logically equivalent to $\neg\psi \equiv \varphi$. □

As an illustration of this construction, let's convert $p \Rightarrow (q \wedge r)$—which we converted to DNF in Example 4.25—to conjunctive normal form too:

Example 4.26: Converting $p \Rightarrow (q \wedge r)$ to CNF.

We are considering the proposition $\varphi = p \Rightarrow (q \wedge r)$. We'll convert it into CNF, using Theorem 4.28.

We start from the truth table for $\neg\varphi$, shown in Figure 4.25. We construct a DNF proposition equivalent to $\neg\varphi$, following the technique of Theorem 4.27. Our proposition $\neg\varphi$ has three clauses, one for each of the truth assignments under which $\neg\varphi$ is true (and φ is false):

$$\neg\varphi \equiv \underset{\text{TTF}}{(p \wedge q \wedge \neg r)} \;\vee\; \underset{\text{TFT}}{(p \wedge \neg q \wedge r)} \;\vee\; \underset{\text{TFF}}{(p \wedge \neg q \wedge \neg r)}$$

We negate this proposition and use De Morgan's Laws to push around the negations:

$$\varphi \equiv \neg\left[(p \wedge q \wedge \neg r) \;\vee\; (p \wedge \neg q \wedge r) \;\vee\; (p \wedge \neg q \wedge \neg r)\right]$$

				φ	$\neg\varphi$	
p	q	r	$q \wedge r$	$p \Rightarrow (q \wedge r)$	$\neg(p \Rightarrow (q \wedge r))$	
T	T	T	T	T	F	$p \wedge q \wedge r$
T	T	F	F	F	T	$p \wedge q \wedge \neg r$
T	F	T	F	F	T	$p \wedge \neg q \wedge r$
T	F	F	F	F	T	$p \wedge \neg q \wedge \neg r$
F	T	T	T	T	F	$\neg p \wedge q \wedge r$
F	T	F	F	T	F	$\neg p \wedge q \wedge \neg r$
F	F	T	F	T	F	$\neg p \wedge \neg q \wedge r$
F	F	F	F	T	F	$\neg p \wedge \neg q \wedge \neg r$

Figure 4.25 The truth table for $\neg(p \Rightarrow (q \wedge r))$.

$$\begin{aligned}
&\equiv \neg(p \wedge q \wedge \neg r) \;\wedge\; \neg(p \wedge \neg q \wedge r) \;\wedge\; \neg(p \wedge \neg q \wedge \neg r) && \textit{De Morgan} \\
&\equiv (\neg p \vee \neg q \vee \neg\neg r) \;\wedge\; (\neg p \vee \neg\neg q \vee \neg r) \;\wedge\; (\neg p \vee \neg\neg q \vee \neg\neg r) && \textit{De Morgan} \\
&\equiv (\neg p \vee \neg q \vee r) \;\wedge\; (\neg p \vee q \vee \neg r) \;\wedge\; (\neg p \vee q \vee r). && \textit{Double Negation}
\end{aligned}$$

So $(\neg p \vee \neg q \vee r) \wedge (\neg p \vee q \vee \neg r) \wedge (\neg p \vee q \vee r)$ is a CNF proposition that's logically equivalent to the given proposition $p \Rightarrow (q \wedge r)$. (We can verify via truth table that we've gotten it right.)

One last comment about these proofs: it's worth emphasizing again that there's genuine creativity required in proving these theorems. Through the strategies from Section 4.3.2 and through practice, you can get better at having the kinds of creative ideas that lead to proofs—but that doesn't mean that these results should have been "obvious" to you in advance. It takes a real moment of insight to see how to use the truth table to develop the DNF proposition to prove Theorem 4.27, or how to use the DNF formula of the negation to prove Theorem 4.28.

Taking it further: Theorems 4.27 and 4.28 said that "a proposition ψ (of a particular form) exists for every φ"—but our proofs actually described an algorithm to *build* ψ from φ. (That's a more computational way to approach a question: a statement like "such-and-such exists!" is the kind of thing more typically proven by mathematicians, and "a such-and-such can be found with this algorithm!" is a claim more typical of computer scientists.) Our algorithms in Theorems 4.27 and 4.28 aren't very efficient, unfortunately; they require 2^k steps just to build the truth table for a k-variable proposition. We'll give a (sometimes, and somewhat) more efficient algorithm in Chapter 5 (see Section 5.4.3) that operates directly on the form of the proposition ("syntax") rather than on using the truth table ("semantics").

Some other results about propositional logic

In the exercises, you'll be asked to prove a large collection of other facts about propositional logic. We'll highlight one of them, which is similar in spirit to the theorems about DNF and CNF: you'll show that any proposition φ is logically equivalent to a simpler proposition that uses only one kind of logical connective, called "nand." For reasons of physics, building the physical circuitry for the logical connective *nand*—as in "not and," where p *nand* q means $\neg(p \wedge q)$—is much simpler than other logical connectives. (The physical reasons relate specifically to the way that *transistors*—the most basic building blocks for digital circuits—work.) The truth table for nand—also known as the *Sheffer stroke* |, after the logician Henry Sheffer (1882–1964)—appears in Figure 4.26. It turns out that every (*every!*) logical connective can be

p	q	$p \mid q$	$p \downarrow q$
T	T	F	F
T	F	T	F
F	T	T	F
F	F	T	T

Figure 4.26 The truth table for nand (also known as the Sheffer stroke |), and nor (also known as Peirce's arrow ↓).

expressed in terms of $|$. In other words, if you have enough nand gates, then you will be able to build *any logical circuit* that you want. Here is a theorem that formally states this result:

Theorem 4.29: All propositions are expressible using only $|$.
For any Boolean formula φ over $p_1, \ldots, p_k$, there exists a proposition $\psi_{\text{nand-only}}$ such that (i) $\psi_{\text{nand-only}}$ is logically equivalent to φ, and (ii) $\psi_{\text{nand-only}}$ contains only $p_1, \ldots, p_k$ and the logical connective $|$.

The theorem follows from Exercise 4.69, where you'll show that every logical connective can be expressed in terms of $|$. (To give a fully rigorous proof, we will need to use mathematical induction, the subject of Chapter 5. Mathematical induction will essentially allow us to apply the results of Exercise 4.69 recursively to translate an arbitrary proposition φ into $\psi_{\text{nand-only}}$.)

> **Taking it further:** Indeed, real circuits are typically built exclusively out of nand gates, using logical equivalences to construct and/or/not gates from a small number of nand gates. Although it may be initially implausible if this is the first time that you've heard it, the processor of a physical computer is essentially nothing more than a giant circuit built out of nand gates and wires. With some thought, you can build a circuit that takes two integers (represented in binary, as a 64-bit sequence) and computes their sum. Similarly, but more thought-provokingly, you can build a circuit that takes an *instruction* (add these numbers; compare those numbers; save this thing in memory; load the other thing from memory) and performs the requested action. That circuit is a computer!

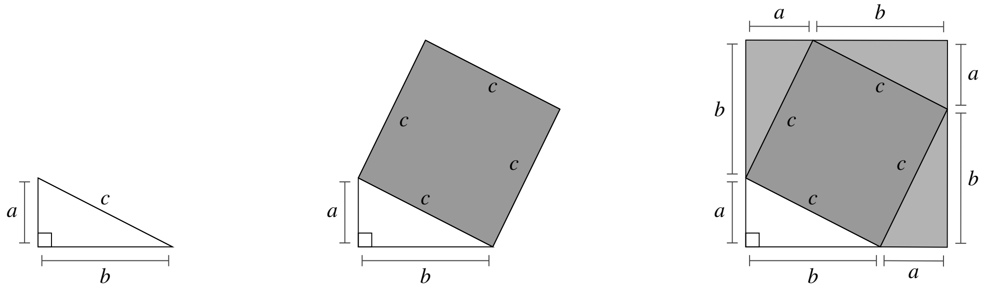
Incidentally, all of the logical connectives can also be defined in terms of the logical connective known as *Peirce's arrow* $\downarrow$ and also known as *nor*, as in "not or." (Peirce's arrow is named after the logician Charles Peirce (1839–1914).) Its truth table is also shown in Figure 4.26. You'll prove the analogous result to Theorem 4.29 for Peirce's arrow in Exercise 4.70.

4.4.2 The Pythagorean Theorem

Example 4.23 presented the Pythagorean Theorem, which you probably once saw in a long-ago geometry class: the square of the length of hypotenuse of a right triangle equals the sum of the squares of the lengths of the legs. (The original formulation of the Pythagorean Theorem is attributed to Pythagoras, a Greek mathematician/philosopher who lived around 500 BCE, though it may have been known to the Mesopotamians or others for many centuries prior.)

Theorem 4.30: The Pythagorean Theorem.
Let a and b denote the lengths of the legs of a right triangle, and let c denote the length of its hypotenuse. Then $a^2 + b^2 = c^2$.

Let's prove it. In brainstorming about this theorem, here's an idea that turns out to be helpful. Because the statement of Pythagorean theorem involves side lengths raised to the second power ("squared"), we might be able to think about the problem using geometric squares, appropriately configured. Here's a proof that proceeds using this geometric idea:



(a) The right triangle.

(b) . . . with an added square.

(c) . . . and three added triangles.

Figure 4.27 Illustrations for the proof of the Pythagorean Theorem, Theorem 4.30.

Proof. Starting with the given right triangle in Figure 4.27a, draw a square with side length c, where one side of the square coincides with the hypotenuse of the given triangle, as in Figure 4.27b. Now draw three new triangles, each identical to the first. Place these three new triangles symmetrically around the square that we just drew, so that each side of the square coincides with the hypotenuse of one of the four triangles, as in Figure 4.27c. Each of these four triangles has leg lengths a and b and hypotenuse c. Including both the c-by-c square and the four triangles, the resulting figure is a square with side length $a + b$.

To complete the proof, we will account for the area of Figure 4.27c in two different ways. First, because a square with side length x has area x^2, we have that

$$\text{area of the enclosing square} = (a+b)^2 = a^2 + 2ab + b^2.$$

Second, this enclosing square can be decomposed into a c-by-c square and four identical right triangles with leg lengths a and b. Because the area of a right triangle with leg lengths x and y is $\frac{xy}{2}$, we also have that

$$\begin{aligned}\text{area of the enclosing square} &= 4 \cdot (\text{area of one triangle}) + c^2\\ &= 4 \cdot \tfrac{1}{2}ab + c^2\\ &= 2ab + c^2.\end{aligned}$$

But the area of the enclosing square is the same regardless of whether we count it all together, or in its five disjoint pieces. Therefore $a^2 + 2ab + b^2 = 2ab + c^2$. The theorem follows by subtracting $2ab$ from both sides. □

There are *many* proofs of the Pythagorean Theorem—in fact, hundreds! There is a classic proof attributed to Euclid (see the next section), and many subsequent and different proof approaches followed over the millennia, including one published, of all people, by U.S. President James Garfield. (There's even a book that collects over 350 different proofs of the result [81].) There's an important lesson to draw from the many proofs of this theorem: *there's more than one way to do it.* ("There's more than one way to do it" is also the motto of the programming language Perl.) Just as there are usually many fundamentally different algorithms for the same problem (think about sorting, for example), there are usually many fundamentally different techniques that can prove the same theorem. Keep an open mind; there is absolutely no shame in proving a result using a different approach than the "standard" way!

4.4.3 Prime Numbers

We'll return to arithmetic for our next set of examples, a pair of proofs about the prime numbers. Recall that a positive integer $n \geq 2$ is *prime* if and only if the only positive integers that divide n evenly are 1 and n itself. Also recall that a positive integer $n \geq 2$ that is not prime is called *composite*. (That is, the integer n is composite if and only if there exists a positive integer $k \notin \{1, n\}$ such that k divides n evenly.)

We'll start with another example of a proof by contradiction, to show that there are infinitely many prime numbers. (A similar proof to the one for Theorem 4.31 dates back around 2300 years. It's due to Euclid, the ancient Greek mathematician after whom Euclidean geometry—and the Euclidean algorithm (see Section 7.2.4)—is named.)

Theorem 4.31: An infinitude of primes.
There are infinitely many prime numbers.

Proof. We proceed by contradiction. Suppose, for the purposes of deriving a contradiction, that there are only finitely many primes. This assumption means that there is a largest prime number, which we will call p. Consider the integer $p!$, the factorial of this largest prime p. Let's consider two separate cases: either $p! + 1$ is prime, or $p! + 1$ is not prime.

Case I: $p! + 1$ is prime. Then we have a contradiction of the assumption that p is the largest prime, because $p! + 1 > p$ is also prime.

Case II: $p! + 1$ is not prime. If $p! + 1$ is not prime, then by definition it is evenly divisible by some integer k satisfying $2 \leq k \leq p!$. But we proved in Example 4.7 that $p! + 1$ is not evenly divisible by any integer between 2 and p, inclusive. Thus the smallest integer k that evenly divides $p! + 1$ must exceed p. Further, this integer k must be prime: if k were not prime, then some $2 \leq k' < k$ divides k and therefore that same k' divides $p! + 1$—but k was defined as the *smallest* divisor of $p! + 1$. Thus $k > p$ is prime, and again we have a contradiction of the assumption that p is the largest prime.

In either case, we have a contradiction! Thus the original assumption—there are only finitely many prime numbers—is false, and so there are infinitely many primes. □

We'll now turn to another result about prime numbers, relating to the *primality testing* problem: you are given a positive integer n, and you have to determine whether n is prime. The definition of primality says that n is composite if there's an integer $k \in \mathbb{Z} - \{1, n\}$ such that $k \mid n$, but it shouldn't be too hard to see that n is composite if and only if there's an integer $k \in \{2, 3, \ldots, n-1\}$ such that $k \mid n$. (That is, the largest possible divisor of n is $n - 1$.) But we can do better, strengthening this result by shrinking the largest candidate value of k:

Theorem 4.32: A composite number n has a factor $\leq \sqrt{n}$.
A positive integer $n \geq 2$ is evenly divisible by some other integer $k \in \{2, 3, \ldots, \lceil \sqrt{n} \rceil\}$ if and only if n is composite.

Proof. We'll proceed by mutual implication.

The forward direction follows immediately from the definition of composite numbers: if there's some integer $k \in \{2, 3, \ldots, \lceil \sqrt{n} \rceil\}$ with $k \neq n$ such that k evenly divides n, then by definition n is composite. (That integer k satisfies $k \mid n$ and $k \notin \{1, n\}$.)

For the other direction, assume that the integer $n \geq 2$ is composite. By definition of composite, there exists a positive integer $k \notin \{1, n\}$ such that $n \bmod k = 0$—that is, there exist positive integers $k \notin \{1, n\}$ and d such that $dk = n$, so $d \mid n$ and $k \mid n$. We must have that $d \neq 1$ (otherwise $dk = 1 \cdot k = k = n$, but $k < n$) and $d \neq n$ (otherwise $dk = nk > n$, but $dk = n$). Thus there exist positive integers $d, k \notin \{1, n\}$ such that $dk = n$. But if both $d > \sqrt{n}$ and $k > \sqrt{n}$, then $dk > \sqrt{n} \cdot \sqrt{n} = n$, which contradicts the fact that $dk = n$. Thus either $d \leq \sqrt{n}$ or $k \leq \sqrt{n}$. □

Taking it further: Generating large prime numbers (and testing the primality of large numbers) is a crucial step in many modern cryptographic systems. See p. 4-68 for some discussion of algorithms for testing primality suggested by these proofs, and a bit about the role that they play in modern cryptographic systems.

4.4.4 Uncomputability

We'll close this section with one of the most important results in computer science, dating from the early 20th century: *there are problems that cannot be solved by computers.* At that time, great thinkers were pondering some of the most fundamental questions that can be asked in CS. What is a computer? What is computation? What is a program? What tasks can be solved by computers/programs? One of the deepest and most mind-numbing results of this time was a proof, developed independently by Alan Turing and by Alonzo Church in the mid-1930s, that there are *uncomputable* problems. That is, there is a problem P for which it's possible to give a completely formal description of the right answer—but it's not possible to write a program that solves P.

Here, we'll prove this theorem. Specifically, we'll describe the *halting problem*, and prove that it's uncomputable. (Informally, the halting problem is: *given a function p written in Python and an input x, does p get stuck in an infinite loop when it's run on x?*) The result is a great example of a proof by contradiction, where we will exploit the abyss of self-reference to produce the contradiction.

Problems

Before we address the computability of the halting problem, we have to define precisely what we mean by a "problem" and "computable." A *problem* is the kind of task that we wish to solve with a computer program. We will focus on yes–no problems, called *decision problems*:

Definition 4.33: Problem.
A *problem* is a description of a set of valid inputs, and a specification of the corresponding output for each them. A *decision problem* is one where the output is either "yes" or "no."

(In other words, a decision problem is specified by a description of a set of possible inputs, along with a description of those inputs for which the correct answer is "yes.")

We've already encountered several decision problems. Here are two:

Example 4.27: The PRIMALITY *decision problem.*
For the PRIMALITY decision problem, the set of possible inputs is the set of positive integers; the set of "yes" inputs is the set of prime numbers. (The "no" inputs are 1 and the composites.)

Example 4.28: The SATISFIABILITY *decision problem.*
For the SATISFIABILITY decision problem, any propositional-logic proposition φ is a valid input, and φ is a "yes" input if and only if φ is satisfiable.

An *instance* of a problem is a valid input for that problem. (An invalid input is one that isn't the right "kind of thing" for that problem.) We will refer to an instance x of a problem P as a *yes-instance* if the correct output is "yes," and as a *no-instance* if the correct output is "no." For example, 17 or 18 are both instances of PRIMALITY; 17 is a yes-instance, while 18 is a no-instance; $p \vee \neg p$ is an invalid input.

Computability

Problems are the things that we'll be interested in solving via computer programs. Informally, problems that can be solved by computer are called *computable* and those that cannot be solved by computer are called *uncomputable.* It'll be easiest to think of computability in terms of your favorite programming language, whatever it may be. For concreteness, we'll pretend it's Python, though any language would do.

Taking it further: The original definition of computability given by Alan Turing used an abstract device called a *Turing machine*; a programming language is called *Turing complete* if it can solve any problem that can be solved by a Turing machine. Every non-toy programming language is Turing complete: Java, C, C++, Python, Ruby, Perl, Haskell, BASIC, Fortran, Assembly Language, whatever.

Formally, we'll define computability in terms of the existence of an algorithm, which we will think of as a function written in Python:

Definition 4.34: Computability.
A decision problem P is *computable* if there exists a Python function $\mathcal{A}$ that solves P. That is, P is computable if there exists a Python function $\mathcal{A}$ such that, on any input x, two conditions hold: (i) $\mathcal{A}$ terminates when run on x; and (ii) $\mathcal{A}(x)$ returns true if and only if x is a yes-instance of P.

Notice that we insist that the Python function $\mathcal{A}$ must actually terminate on any input x: it's not allowed to run forever. Furthermore, running $\mathcal{A}(x)$ returns True if x is a yes-instance of P and running $\mathcal{A}(x)$ returns False if x is a no-instance of P.

The decision problems from Examples 4.27 and 4.28 are both computable:

Example 4.29: Computability of PRIMALITY.
PRIMALITY is computable: both **isPrime** and **isPrimeBetter** (p. 4-68) are algorithms that could be implemented as a Python function that (i) terminates when run on any positive integer, and (ii) returns True on input n if and only if n is prime.

Example 4.30: Computability of SATISFIABILITY.
SATISFIABILITY is computable, too: as we discussed in Section 3.3.1, we can exhaustively try all truth assignments for φ, checking whether any of them satisfies φ. This algorithm is slow—if φ has n variables, there are 2^n different truth assignments—but it is guaranteed to terminate for any input φ, and correctly decides whether φ is satisfiable.

Programs that take source code as input

The inputs to the problems or programs that we've talked about so far have been integers (for PRIMALITY) or Boolean formulas (for SATISFIABILITY). Of course, other input types like rational numbers or lists are possible, too. *Programs that take programs as input* are a particularly important category.

Taking it further: Although you might not have thought about them in these terms, you've frequently encountered programs that take programs as input. For example, in any introductory CS class, you've seen one frequently: the Python interpreter `python`, the Java compiler `javac`, and the C compiler `gcc` all take programs (written in Python or Java or C, respectively) as input.

It's easy to think up some decision problems where the input is a Python program. Here's one, about commenting code. (For example, it's probably not too hard to imagine an Intro CS instructor setting up an automated grading system for programs that gives an automatic zero to any submitted assignment that contains no comments.)

Example 4.31: The COMMENTED *decision problem.*
Define the decision problem COMMENTED as follows:

Input: the Python source code s for a function.
Output: "yes" if s contains at least one comment; "no" otherwise.

```
def commentedTester(sourceCode):
  for ch in sourceCode:
    if ch == "#" and is not inside quotes:
      return True
  return False
```

```
def absValue(n):
  if n > 0:
    return n
  else:
    return -1 * n
```

```
def isEven(n):
  # % is Python's mod operator
  if n % 2 == 0:
    return True  # n is even
  else:
    return False # n is odd
```

Figure 4.28 Python source code for three functions.

In Python, a comment starts with # and goes until the end of the line, so as long as a # appears somewhere in the source code *s*—and not inside quotation marks—then *s* is a yes-instance of COMMENTED; otherwise *s* is a no-instance.

The COMMENTED problem is computable: testing whether *s* is a yes-instance can be done by looking at the characters of *s* one by one, and testing to see whether any one of those characters starts a comment. A Python program `commentedTester` that solves COMMENTED is shown in Figure 4.28. (The details of testing whether `ch` is inside quotes are omitted from the source code, but otherwise the code for `commentedTester` is valid, runnable Python code.)

Consider running `commentedTester` on the other instances shown Figure 4.28. Observe that `absValue` is a no-instance of COMMENTED, because it doesn't contain the comment character # at all, and `isEven` is a yes-instance of COMMENTED, because it contains three comments. As desired, if we ran `commentedTester` on these two pieces of source code, the output would be `False` and `True`, respectively.

Example 4.31 showed that the decision problem COMMENTED is computable by giving a Python function `commentedTester` that solves COMMENTED. Because we can run `commentedTester` on any piece of Python source code we please, let's do something a little bizarre: let's run `commentedTester` on the source code for `commentedTester` itself (!). There weren't any comments in `commentedTester`—the only # in the code is inside quotes—so the source code of `commentedTester` is a no-instance of COMMENTED. Put a different way, if s_{ct} denotes the source code of `commentedTester`, then running s_{ct} on s_{ct} returns `False`. This idea of taking some source code *s* and running *s on s itself* will be essential in the rest of this section.

The Halting Problem

The key decision problem that we'll consider is the *halting problem*:

Definition 4.35: The Halting Problem.
Define the decision problem HALTINGPROBLEM as follows:

Input: a pair $\langle s, x \rangle$, where *s* is the source code of a syntactically valid Python function that takes one argument, and *x* is any value.
Output: "yes" if *s* terminates when run on input *x*; "no" otherwise.

That is, $\langle s, x \rangle$ is a yes-instance of haltingProblem if $s(x)$ terminates (doesn't get stuck in an infinite loop), and it's a no-instance if $s(x)$ does get stuck in an infinite loop.

We can now use the idea of running a function with itself as input to show that the Halting Problem is uncomputable, by contradiction:

Theorem 4.36: Uncomputability of the Halting Problem.
haltingProblem is uncomputable.

Proof. We give a proof by contradiction. Suppose for the sake of contradiction that the Halting Problem is computable—that is, assume

$$\text{There's a Python function } \mathcal{A}_{\text{halting}} \text{ solving the Halting Problem.} \tag{1}$$

(In other words, for the Python source code s of a one-argument function, and any value x, running $\mathcal{A}_{\text{halting}}(s, x)$ always terminates, and $\mathcal{A}_{\text{halting}}(s, x)$ returns `True` if and only if running s on x does not result in an infinite loop.)

Now consider the Python function `makeSelfSafe` in Figure 4.29. The function `makeSelfSafe` takes as input the Python source code s of a one-argument function, tests whether running s on s itself is "safe" (does not cause an infinite loop), and if it's safe then it runs s on s. We claim that `makeSelfSafe` never gets stuck in an infinite loop:

$$\text{For any Python source code } s, \texttt{makeSelfSafe}(s) \text{ terminates.} \tag{2}$$

To see that (2) is true, observe that Line 2 of the algorithm always terminates, by assumption (1). Line 3 of the algorithm ensures that s is called on input s if and only if $\mathcal{A}_{\text{halting}}(s, s)$ said that s terminates when run on s. And, by assumption, $\mathcal{A}_{\text{halting}}$ is always correct. Thus s is run on input s *only if* s terminates when run on input s. So Lines 3–4 of the algorithm always terminates. And Line 5 of the algorithm doesn't do anything except return, so it terminates immediately. Thus (2) follows.

Write s_{mss} to denote the Python source code of `makeSelfSafe`. Because s_{mss} is itself Python source code, Fact (2) implies that

$$\texttt{makeSelfSafe}(s_{\text{mss}}) \text{ terminates.} \tag{3}$$

```
1 def makeSelfSafe(s):   # the input s is the Python source code of a one-argument function.
2    safe = $\mathcal{A}_{\text{halting}}$(s,s)
3    if safe:
4       run s on input s
5    return True
```

Figure 4.29 The Python code for `makeSelfSafe`.

In other words, running s_{mss} on s_{mss} terminates. Thus, by the assumption (1) that $\mathcal{A}_{\text{halting}}$ is correct, we can conclude that

$$\mathcal{A}_{\text{halting}}(s_{\text{mss}}, s_{\text{mss}}) \text{ returns true.} \tag{4}$$

But now consider what happens when we run makeSelfSafe on its own source code—that is, when we compute makeSelfSafe(s_{mss}). Observe that safe is set to true in Line 2 of the algorithm, by Fact (4). Thus Line 4 calls makeSelfSafe(s_{mss}) recursively! But therefore makeSelfSafe(s_{mss}) calls makeSelfSafe(s_{mss}), which calls makeSelfSafe(s_{mss}), and so on, *ad infinitum.* In other words,

$$\texttt{makeSelfSafe}(s_{\text{mss}}) \text{ does not terminate.} \tag{5}$$

But (3) and (5) are contradictory! Thus the only assumption that we made, namely (1), was false. Therefore there does not exist a correct always-terminating algorithm for the Halting Problem. That is, the Halting Problem is uncomputable. □

To summarize Theorem 4.36: we showed that the assumption of the existence of an algorithm for the halting problem leads to a contradiction, and therefore we conclude that such an algorithm cannot exist. The contradiction is, at its heart, about self-reference—an algorithmic version of the Liar's Paradox: *This sentence is false.*

Taking it further: *Computability theory* is the study of what problems can and cannot be solved by computers. Computability was a primary focus of theoretical computer science from the 1930s through roughly the 1970s. (After that time, the focus of theoretical computer scientists began to shift to *complexity theory*, which addresses the question of what problems can and cannot be solved *efficiently* by computers.) You can read more about the halting problem in any textbook on computability theory, or a remarkable book by Douglas Hofstadter [56]. For extra amusement, you can find a full proof of Theorem 4.36 by Geoffrey K. Pullum in poem form, reproduced in Figure 4.30. And see p. 4-70 for a discussion of some practically relevant problems that are also uncomputable.

Scooping the Loop Snooper: A proof that the Halting Problem is undecidable

Geoffrey K. Pullum

No general procedure for bug checks will do.
Now, I won't just assert that, I'll prove it to you.
I will prove that although you might work till you drop,
you cannot tell if computation will stop.

For imagine we have a procedure called P
that for specified input permits you to see
whether specified source code, with all of its faults,
defines a routine that eventually halts.

You feed in your program, with suitable data,
and P gets to work, and a little while later
(in finite compute time) correctly infers
whether infinite looping behavior occurs.

If there will be no looping, then P prints out 'Good.'
That means work on this input will halt, as it should.
But if it detects an unstoppable loop,
then P reports 'Bad!'—which means you're in the soup.

Well, the truth is that P cannot possibly be,
because if you wrote it and gave it to me,
I could use it to set up a logical bind
that would shatter your reason and scramble your mind.

Here's the trick that I'll use—and it's simple to do.
I'll define a procedure, which I will call Q,
that will use P's predictions of halting success
to stir up a terrible logical mess.

For a specified program, say A, one supplies,
the first step of this program called Q I devise
is to find out from P what's the right thing to say
of the looping behavior of A run on A.

If P's answer is 'Bad!', Q will suddenly stop.
But otherwise, Q will go back to the top,
and start off again, looping endlessly back,
till the universe dies and turns frozen and black.

And this program called Q wouldn't stay on the shelf;
I would ask it to forecast its run on *itself.*
When it reads its own source code, just what will it do?
What's the looping behavior of Q run on Q?

If P warns of infinite loops, Q will quit;
yet P is supposed to speak truly of it!
And if Q's going to quit, then P should say 'Good.'
Which makes Q start to loop! (P denied that it would.)

No matter how P might perform, Q will scoop it:
Q uses P's output to make P look stupid.
Whatever P says, it cannot predict Q:
P is right when it's wrong, and is false when it's true!

I've created a paradox, neat as can be—
and simply by using your putative P.
When you posited P you stepped into a snare;
Your assumption has led you right into my lair.

So where can this argument possibly go?
I don't have to tell you; I'm sure you must know.
A *reductio:* There cannot possibly be
a procedure that acts like the mythical P.

You can never find general mechanical means
for predicting the acts of computing machines;
it's something that cannot be done. So we users
must find our own bugs. Our computers are losers!

Figure 4.30 A (slightly different) proof of Theorem 4.36, in poetic form. Copyright © 2012 by Geoffrey K. Pullum. Used here by permission of the author.

Computer Science Connections

Cryptography and the Generation of Prime Numbers

Prime numbers are used extensively in cryptographic systems. For example, the *RSA cryptosystem*—named after the first letters of its inventors' last names [106]; see Section 7.5 for *much* more— uses as a primary step the generation of two large prime numbers, each of which might be an integer containing ≈ 128 bits in its binary representation.

Prime numbers are useful in cryptography because of an asymmetry in the apparent difficulty of two directions of a particular problem in arithmetic. (See Figure 4.31.) If you are given two (big) prime numbers p and q, then computing their product pq is easy. But if you are given a number n that is guaranteed to be the product of two prime numbers, finding those two numbers appears to be much harder. For example, if you're told that $n = 504{,}761$, it will probably take a long time to figure out that $n = 251 \cdot 2011$. But if you're told that $p = 251$ and $q = 2011$, then you should be able to calculate $pq = 504{,}761$ in maybe a minute or so, even if you have to do all the calculation by hand.

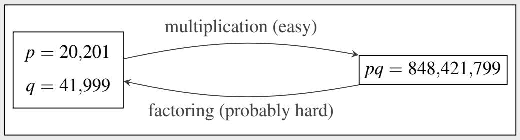


Figure 4.31 Two directions of a problem. (Maybe strangely, "easy" is a technical term for problems in CS, meaning that there exists an efficient algorithm to solve it. We aren't *sure* that factoring is "hard"—it's just that there's no efficient algorithm that is known.)

A crucial step in RSA is the generation of large prime numbers. One way to accomplish this step is by choosing a random integer of the appropriate size and then testing whether that number is prime. (We keep retrying until the random number turns out to be prime.) So how do we test for primality?

A little consideration of the definition implies that we can test whether an integer n is prime by testing all candidate divisors between 2 and $n - 1$. This algorithm requires us to do roughly n divisibility checks (actually, to be precise, $n - 2$ divisibility checks). Using Theorem 4.32, the algorithm can be improved to do only about $\sqrt{n}$ divisibility checks, as in Figure 4.32. Although there are other, generally better ways of evaluating them (see Chapter 6), we can test these two algorithms empirically. A Python implementation using $n - 1$ calls to **isPrime** to find all primes in the integers $\{2, \ldots, n\}$ took about three minutes for $n = 65{,}536$ on an ancient laptop of mine. For the same n on the same machine, **isPrimeBetter** took about a second. This difference is a nice example of the way in which theoretical, proof-based techniques can improve actual widely used algorithms.

```
isPrime($n$):
$k := 2$
while $k < n$:
    if $n$ is evenly divisible by $k$ then
        return False
    $k := k + 1$
return True
```

```
isPrimeBetter($n$):
$k := 2$
while $k \leq \lceil \sqrt{n} \rceil$:
    if $n$ is evenly divisible by $k$ and $n \neq k$ then
        return False
    $k := k + 1$
return True
```

Figure 4.32 Slow and less slow primality testing. (We could further save roughly another factor of two by checking only $k = 2$ and odd $k \geq 3$.)

In part because of its importance to cryptography, there has been significant work on primality testing over recent decades—improving far beyond the roughly $\sqrt{n}$ division tests of **isPrimeBetter**. In general, an efficient algorithm for a number n should require a number of steps proportional to $\log n$—the number of bits or digits in the representation of n—rather than proportional to n or even $\sqrt{n}$. (When you add two 10-digit numbers by hand, you want to do about 10 operations, rather than about 1,000,000,000 operations.) Thus **isPrimeBetter** still does not count as "efficient"—in fact, it's still quite slow on large inputs.

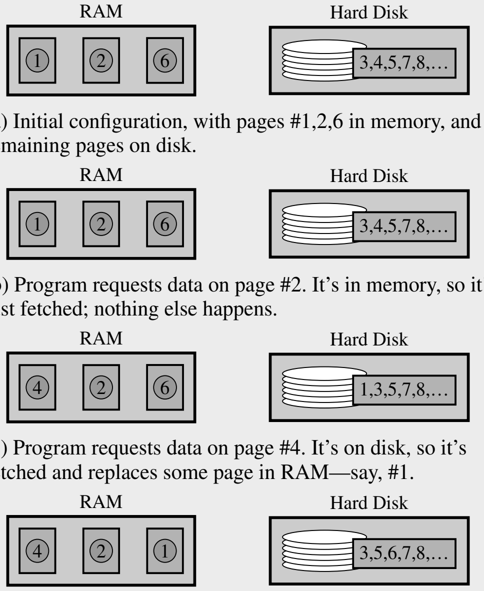
There are some very efficient *randomized* algorithms for primality testing that are actually used in real cryptosystems, including one called the *Miller–Rabin test* [89, 105]. This randomized algorithm performs a (randomly chosen) test that all prime numbers pass and most composite numbers fail; repeating this operation with many different randomly chosen tests decreases the probability of getting a wrong answer to an arbitrarily small number. (See p. 7-53.) And more recently, researchers gave the first theoretically efficient algorithm for primality testing that's not randomized [4]. In other words, like multiplication (and probably unlike factoring), primality testing is "easy."

Computer Science Connections

Other Uncomputable Problems (That You Might Care About)

The Halting Problem (Theorem 4.36) may seem like a purely abstract problem, and therefore one that doesn't matter in the real world—sure, it'd be nice to have an infinite-loop detector in your Python interpreter or Java compiler, but would it just be a vaguely helpful feature for students in Intro CS classes but nobody else? The answer is a resounding no: while the Halting Problem itself may seem obscure, there are many uncomputable problems that, if solved, would vastly improve operating systems or compilers, among other things. But they're uncomputable, and therefore the desired improvements cannot be made.

Here's one example. Modern operating systems use *virtual memory* for their applications. The physical computer has a limited amount of physical memory—say, 64 gigabytes of RAM—that applications can use. But the operating system "pretends" that it has a much larger amount of memory, so that the word processor, web browser, Java compiler, and solitaire game can *each* act as though they had even more than 64 gigabytes of memory that they don't have to share. Memory (both virtual and real) is divided into chunks of a fixed size, called *pages*. The operating system stores pages that are actively in use in physical memory (RAM), and relegates some of the not-currently-used pages to the hard drive. At every point in time, the operating system's *paging system* decides which pages to leave in physical memory, and which pages to "eject" to the hard drive when there are too many in RAM. (This idea is the same as what you do when you're cooking several dishes in a kitchen with limited counter space: you have to relegate some of the not-currently-being-prepared ingredients to the fridge. And at every moment that the counter's full, you have to decide which ingredients to leave on the counter, and which to "eject" to the fridge.) See Figure 4.33.



(a) Initial configuration, with pages #1,2,6 in memory, and remaining pages on disk.

(b) Program requests data on page #2. It's in memory, so it's just fetched; nothing else happens.

(c) Program requests data on page #4. It's on disk, so it's fetched and replaces some page in RAM—say, #1.

(d) Program requests data on page #1. It's on disk, so it's fetched and replaces some page in RAM—say, #6.

Figure 4.33 A sample sequence of memory fetches in a paged memory system.

Here's a problem that a paging system would love to solve: given a page p of memory that an application has used, will that application ever access the contents of p again? Let's call this problem willBeUsedAgain. When the paging system needs to eject a page, ideally it would eject a page that's a no-instance of willBeUsedAgain, because it will never have to bring that page back into physical memory. (When you're out of counter space, you would of course prefer to put away some ingredient that you're done using.)

Unfortunately for operating system designers, willBeUsedAgain is uncomputable. There's a very quick proof, based on the uncomputability of the Halting Problem. Consider the following algorithm:

- run the Python function f on the input x.
- if $f(x)$ terminates, then access some memory from page p.

This algorithm accesses page p if and only if $\langle f, x \rangle$ is a yes-instance of the Halting Problem.

Therefore *if we could give an algorithm to solve the* willBeUsedAgain *problem, then we could give an algorithm to solve the Halting Problem.* (Just check whether p will be used again; the answer that that question tells you whether f halts on x!) But we already know that we can't give an algorithm to solve the Halting Problem. If $p \Rightarrow q$ and $\neg q$, then we can conclude $\neg p$; therefore willBeUsedAgain is uncomputable.

EXERCISES

Figure 4.34 shows the truth tables for all 16 *different binary logical operators, with each column named if it's a logical operator that we've already seen. A set S of binary operators is said to be* universal *if every binary logical operation can be expressed using some combination of the operators in S. Formally, a set S is universal if, for every Boolean expression φ over variables $p_1, \ldots, p_k$, there exists a Boolean expression ψ that is logically equivalent to φ where ψ uses only the variables $p_1, \ldots, p_k$ and the logical connectives in S.*

4.66 Prove that the set $\{\vee, \wedge, \Rightarrow, \neg\}$ is universal. *(Hint: to do so, you need to show that, for each column $\boxed{1}$ through $\boxed{16}$ of Figure 4.34, you can build a Boolean expression φ_i over the variables p and q that uses only the operators $\{\vee, \wedge, \Rightarrow, \neg\}$, and such that φ_i is logically equivalent to $p \, \boxed{i} \, q$.)*

4.67 Prove that the set $\{\vee, \wedge, \neg\}$ is universal. *(Hint: once you've done Exercise 4.66, all you have to do is show that you can express $\Rightarrow$ using $\{\vee, \wedge, \neg\}$.)*

4.68 Prove that $\{\vee, \neg\}$ and $\{\wedge, \neg\}$ are both universal.

4.69 Prove that the set $\{|\}$—the set containing just the Sheffer stroke, that is, *nand*—is universal.

4.70 Prove that the singleton set $\{\downarrow\}$ is universal.

4.71 Prove that the set $\{\wedge, \vee\}$ is *not* universal. *(Hint: what happens under the all-true truth assignment?)*

4.72 Let φ be a fully quantified proposition of predicate logic. Prove that φ is logically equivalent to a fully quantified proposition ψ in which *all quantifiers are at the outermost level of* ψ. In other words, the proposition ψ must be of the form

$$ {}^{\forall}\!/_{\exists}\, x_1 : \ {}^{\forall}\!/_{\exists}\, x_2 : \ \cdots {}^{\forall}\!/_{\exists}\, x_k : \ P(x_1, x_2, \ldots, x_k), $$

where each ${}^{\forall}\!/_{\exists}$ is either a universal or existential quantifier. (The transformation that you performed in Exercise 3.185 put Goldbach's Conjecture in this special form.) *(Hint: you might find the results from Exercises 4.66–4.71 helpful. Using these results, you can assume that φ has a very particular form.)*

4.73 Prove that, for any integer $n \geq 1$, there is an n-variable logical proposition φ in conjunctive normal form such that the truth-table translation to DNF (from Theorem 4.27) yields an DNF proposition with exponentially more clauses than φ has.

4.74 Prove that the area of a right triangle with legs x and y is $xy/2$.

4.75 Use Figure 4.35 as an outline to give a different proof of the Pythagorean Theorem.

4.76 Exercise 4.47 asked you to prove (via algebra) the *Arithmetic Mean–Geometric Mean inequality*: for $x, y \in \mathbb{R}^{\geq 0}$, we have that $\sqrt{xy} \leq \frac{x+y}{2}$. Here you'll reprove the result geometrically. Suppose that $x \geq y$, and draw two circles of radius x and y tangent to each other, and tangent to a horizontal line. See Figure 4.36. Considering the right triangle shown in that diagram, and using the Pythagorean Theorem and the fact that the hypotenuse is the longest side of a right triangle, prove the result again.

Let $x, y \in \mathbb{R}^2$ be two points in the plane. As usual, denote their coordinates by x_1 and x_2, and y_1 and y_2, respectively. The Euclidean distance *between these points is the length of the line that connects them: $\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$. The* Manhattan distance

		$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$	$\boxed{5}$	$\boxed{6}$	$\boxed{7}$	$\boxed{8}$	$\boxed{9}$	$\boxed{10}$	$\boxed{11}$	$\boxed{12}$	$\boxed{13}$	$\boxed{14}$	$\boxed{15}$	$\boxed{16}$
p	q	True	$p \vee q$	$p \Leftarrow q$	p	$p \Rightarrow q$	q	$p \Leftrightarrow q$	$p \wedge q$	$p \mid q$	$p \oplus q$	$\neg q$		$\neg p$		$p \downarrow q$	False
T	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*
T	*F*	*T*	*T*	*T*	*T*	*F*	*F*	*F*	*F*	*T*	*T*	*T*	*T*	*F*	*F*	*F*	*F*
F	*T*	*T*	*T*	*F*	*F*	*T*	*T*	*F*	*F*	*T*	*T*	*F*	*F*	*T*	*T*	*F*	*F*
F	*F*	*T*	*F*	*T*	*F*	*T*	*F*	*T*	*F*	*T*	*F*	*T*	*F*	*T*	*F*	*T*	*F*

Figure 4.34 The full set of binary logical operators.

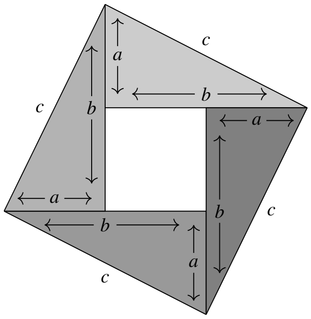


Figure 4.35 A diagram illustrating another way to prove the Pythagorean Theorem.

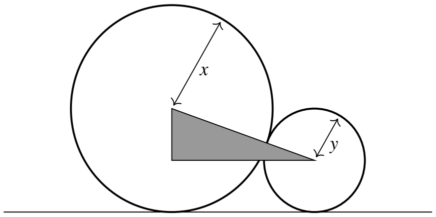


Figure 4.36 Using the Pythagorean Theorem for the Arithmetic Mean/Geometric Mean inequality.

between them is $|x_1 - y_1| + |x_2 - y_2|$: the number of blocks that you would have to walk "over" plus the number that you'd have to walk "up" to get from one point to the other. Denote these distances by $d_{euclidean}$ and $d_{manhattan}$.

4.77 Prove that $d_{\text{euclidean}}(x, y) \leq d_{\text{manhattan}}(x, y)$ for any two points x, y.

4.78 Prove that there exists a constant a such that both

- $d_{\text{manhattan}}(x, y) \leq a \cdot d_{\text{euclidean}}(x, y)$ for all points x and y; and
- there exist points x^*, y^* such that $d_{\text{manhattan}}(x^*, y^*) = a \cdot d_{\text{euclidean}}(x^*, y^*)$

4.79 On p. 2-63, we discussed rotation matrices for points in two dimensions. Specifically, given a point $\langle x, y \rangle \in \mathbb{R}^2$ and an angle $\theta \in \mathbb{R}$, we claimed that rotating the given point by the given angle around the origin $\langle 0, 0 \rangle$ resulted in the point

$$\langle x \cos\theta - y \sin\theta, x \sin\theta + y \cos\theta \rangle.$$

Prove that this formula is correct. You may find two facts from trigonometry (which you may use without proof) useful:

$$\sin(\alpha + \beta) = \sin\alpha \cos\beta + \cos\alpha \sin\beta \qquad \text{and} \qquad \cos(\alpha + \beta) = \cos\alpha \cos\beta - \sin\alpha \sin\beta.$$

A positive integer n is called a perfect number *if it is equal to the sum of all positive integer factors $1 \leq k < n$ of n. For example, the number 14 is not perfect: the numbers less than 14 that evenly divide 14 are $\{1, 2, 7\}$, but $1 + 2 + 7 = 10 \neq 14$.*

4.80 Prove that at least one perfect number exists.

4.81 Prove that, for any prime integer p, the positive integer p^2 is not a perfect number.

4.82 Let $n \geq 10$ be any positive integer. Prove that the set $\{n, n+1, \ldots, n+5\}$ contains at most two prime numbers.

4.83 Let n be any positive integer. Prove or disprove: any set of ten consecutive positive integers $\{n, n+1, \ldots, n+9\}$ contains at least one prime number.

4.84 *(Thanks to the NPR radio show Car Talk, from which I learned this exercise.)* Imagine a junior high school, with 100 lockers, numbered 1 through 100. All lockers are initially closed. There are 100 students, each of whom—brimming with teenage angst—rampages through the hallway and systematically slams some of them shut and yanks some of them open. Specifically, in round $i = 1, 2, \ldots, 100$, student $\#i$ changes the state of every ith locker: if the door is open, then it's slammed shut; if the door is closed, then it's opened. (So student #1 opens them all, student #2 closes all the even-numbered lockers, etc.) Which lockers are open after this whole process is over? Prove your answer.

4.85 Theorem 4.32 established this claim: *A positive integer $n \geq 2$ is evenly divisible by some other integer $k \in \{2, 3, \ldots, \lceil \sqrt{n} \rceil\}$ if and only if n is composite.* If we delete the word "other," this claim becomes false. Prove that this modified claim is false.

4.86 Prove that the unmodified claim (retaining the word "other") remains true if the bounds on k are changed from $k \in \{2, 3, \ldots, \lceil\sqrt{n}\rceil\}$ to $k \in \{\lceil\sqrt{n}\rceil, \ldots, n-1\}$.

4.87 Prove that the bound *cannot* be changed from $k \in \{2, 3, \ldots, \lceil\sqrt{n}\rceil\}$ to $k \in \{\lfloor\sqrt{n}/2\rfloor, \ldots, \lfloor 3\sqrt{n}/2\rfloor\}$. That is, prove that the following claim is false: *A positive integer $n \geq 2$ is evenly divisible by some other integer $k \in \{\lfloor\sqrt{n}/2\rfloor, \ldots, \lfloor 3\sqrt{n}/2\rfloor\}$ if and only if n is composite.*

4.88 Let n be any positive integer, and let p_n denote the smallest prime number that evenly divides n. Prove that there are infinite number of integers n such that $p_n \geq \sqrt{n}$. (This fact establishes that we cannot change the bound in the aforementioned theorem to anything smaller than $\sqrt{n}$.)

4.5 Common Errors in Proofs

Mistakes were made.

Ron Ziegler (1939–2003)
Press Secretary for President Richard Nixon during Watergate,
in an apology to *The Washington Post* (1973)

We've now spent considerable time establishing a catalogue of proof techniques that you can use to prove theorems, along with some examples of these techniques in action. We'll close this chapter with a brief overview of some common *flaws* in proofs, so that you can avoid them in your own work (and be on the lookout for them in the work of others). Recall that a proof consists of a sequence of logical inferences, deriving new facts from assumptions or previously established facts. A *valid* inference is one whose conclusion is always true as long as the facts that it relies on were true. (That is, a valid step never creates a false statement from true ones.) An *invalid* inference is one in which the conclusion can be false *even if the premises are all true.* An invalid argument can also be called a *logical fallacy,* a *fallacious argument,* or just a *fallacy.* In a correct proof, of course, every step is valid.

Taking it further: Some of the most famous disasters in the history of computer science have come from some bugs that arose because of an erroneous understanding of some property of a system—and a lack of valid proof of correctness for the system. These bugs have been costly, with both lives and many dollars lost. See p. 4-82 for a few highlights/lowlights.

Here are a few examples of a single logical inference, some of which might be fallacious:

Problem-solving tip: To make the logical structure of an argument clearer, consider an abstract form of the argument in which you use variables to name the atomic propositions.

Example 4.32: Some (valid and invalid) logical inferences.

Here are several inferences. In each case, there are two premises, and a conclusion that is claimed to follow logically from those premises. Which of these inferences are valid, and which are fallacies?

1 *Premise A.* All people belch sometimes.
Premise B. Sister Mary Kenneth Keller was a person.
Conclusion. Therefore, Sister Mary Kenneth Keller belched sometimes.

2 *Premise A.* All software is buggy.
Premise B. Windows is a piece of software.
Conclusion. Therefore, Windows is buggy.

3 *Premise A.* If you handed in an exam without your name on it, then you got a zero.
Premise B. You handed in an exam without your name on it.
Conclusion. Therefore, you got a zero.

4 *Premise A.* If you handed in an exam without your name on it, then you got a zero.
Premise B. You handed in an exam with your name on it.
Conclusion. Therefore, you didn't get a zero.

Solution. We abstract away from buggy software and belching people by rewriting these arguments in purely logical form:

1 Assume $a \in S$ and assume $\forall x \in S : P(x)$. Conclude $P(a)$.
2 Assume $a \in S$ and assume $\forall x \in S : P(x)$. Conclude $P(a)$.
3 Assume $p \Rightarrow q$ and assume p. Conclude q.
4 Assume $p \Rightarrow q$ and assume $\neg p$. Conclude $\neg q$.

In this format, we see first that (1) and (2) are actually the same logical argument (with different meanings for the symbols), and they're both valid. Argument (3) is precisely an invocation of Modus Ponens (see Chapter 3), and it's valid. But (4) is a fallacy: the fact that $p \Rightarrow q$ and $\neg p$ is consistent with either q or $\neg q$, so in particular when $p =$ False and $q =$ True the premises are true but the conclusion is false.

Taking it further: Sister Mary Kenneth Keller (1913–1985) earned a Ph.D. in computer science from the University of Wisconsin in 1965. Hers was one of two such degrees awarded that day in the U.S., tying her as one of the first two CS Ph.D. recipients in the country. (She'd previously become a nun, in 1940.) Her Ph.D. work was on symbolic differentiation, and she was also a major contributor to the early work on the BASIC programming language. And, despite all of that, she—if you believe the premises of Example 4.32—belched sometimes.

Each of these examples purports to convince its reader of its conclusion, *under the assumption* that the premises are true. Valid arguments will convince any (reasonable) reader that their conclusion follows from their premises. Fallacious arguments are buggy; a vigilant reader will not accept the conclusion of a fallacious argument even if she accepts the premises.

Taking it further: Here's a useful way to think about validity and fallacy. An argument with premises $p_1, p_2, \ldots, p_k$ and conclusion c is valid if and only if $p_1 \wedge p_2 \wedge \cdots \wedge p_k \Rightarrow c$ is a theorem. If there is a circumstance in which $p_1 \wedge p_2 \wedge \cdots \wedge p_k \Rightarrow c$ is false—in other words, where the premises $p_1 \wedge p_2 \wedge \cdots \wedge p_k$ are all true but the conclusion c is false—then the argument is fallacious.

Your main job in proofs is simple: avoid fallacies! But that can be harder than it sounds. The remainder of this section is devoted to a few types of common mistakes in proofs—that is, some common types of fallacies.

A broken proof

The most common mistake in a purported proof is simple but insidious: a single statement is alleged to follow logically from previous statements, but it doesn't. Here's a somewhat subtle example:

Problem-solving tip: The kind of mistake in Example 4.33, in which there's a single step that doesn't follow from the previous step, can sometimes be difficult to sniff out. But it's the kind of bug that you can spot by simply being überskeptical of everything that's written in a purported proof.

Example 4.33: Figure out what's wrong with this logic.
Find the error in this purported proof, and give a counterexample to the claim.

False Theorem: Let $F_n = \left\{k \in \mathbb{Z}^{\geq 1} : k \mid n\right\}$ denote the factors of an integer $n \geq 2$. Then $|F_n|$ is even.

Proof. Let $F_{\text{small}} \subseteq F_n$ be the set of factors of n that are less than $\sqrt{n}$. Let $F_{\text{big}} \subseteq F_n$ be the set of factors that are greater than $\sqrt{n}$. Observe that every $d \in F_{\text{small}}$ has a unique entry $\frac{n}{d}$ corresponding to it in F_{big}. Therefore $|F_{\text{small}}| = |F_{\text{big}}|$. Let $k = |F_{\text{small}}| = |F_{\text{big}}|$. Note that k is an integer. Thus F_n contains precisely k elements less than $\sqrt{n}$ and k elements greater than $\sqrt{n}$, and so $|F_n| = 2k$, which is an even number. □

Solution. The problem comes right at the end of the proof, in the last sentence: *Thus F_n contains precisely k elements less than $\sqrt{n}$ and k elements greater than $\sqrt{n}$, and so $|F_n| = 2k$.*

The problem is that this statement discounts the possibility that $\sqrt{n}$ itself might be in F_n. (Note that $\sqrt{n}$ is in *neither* F_{small} nor F_{big} as they were defined.) For an integer n that's a perfect square, we have that $\sqrt{n} \in F_n$, and therefore $|F_n| = 2k + 1$. For example, the integer 9 is a counterexample to the claim, because $F_9 = \{1, 3, 9\}$ and $|F_9| = 3$.

But while an error of this form—one step in the proof that doesn't actually follow from the previously established facts—may be the most common type of bug in a proof, there are some other, more structural errors that can arise. Most of these structural errors come from errors of propositional logic—namely by proving a new proposition that's *not* in fact logically equivalent to the given proposition. Here are a few of these types of flawed reasoning.

Fallacy: proving true

We are considering a claim φ. We proceed as follows: we assume φ, and (correctly) prove True under that assumption. (Usually, for some reason, the "proof" writer puts a little check mark in their alleged proof at this point: ✓.) What can we conclude about φ? The answer is: *absolutely nothing!* The reason: we've proven that $\varphi \Rightarrow$ True, but *anything implies true.* (Both True $\Rightarrow$ True and False $\Rightarrow$ True are true implications.) Here's a classical example of a bogus proof that uses this fallacious reasoning:

Example 4.34: Figure out what's wrong with this logic.
Find the error in this purported proof.

False Theorem: $1 = 0$.

Proof. Suppose that $1 = 0$. Then:

$$\begin{aligned} && 1 &= 0 \\ \text{therefore, multiplying both sides by 0} && 0 \cdot 1 &= 0 \cdot 0 \\ \text{and therefore,} && 0 &= 0. \quad \checkmark \end{aligned}$$

And, indeed, $0 = 0$. Thus the assumption that $1 = 0$ was correct, and the theorem follows. □

Solution. We have merely shown that $(1 = 0) \Rightarrow (0 = 0)$, which does not say anything about the truth or falsity of $1 = 0$; anything implies true.

Writing tip: When you're trying to prove that two quantities a and b are equal, it's generally preferable to manipulate a until it equals b, rather than "meeting in the middle" by manipulating both sides of the equation until you reach a line in which the two sides are equal. The "manipulate a until it equals b" style of argument makes it clear to the reader that you are proving $a = b$ rather than proving $(a = b) \Rightarrow$ True.

Fallacy: affirming the consequent

We are considering a claim φ. We prove (correctly) that $\varphi \Rightarrow \psi$, and we prove (correctly) that ψ. We then conclude φ. (Recall that ψ is the *consequent* of the implication $\varphi \Rightarrow \psi$, and we have "affirmed" it by proving ψ.) This "proof" is wrong because it confuses necessary and sufficient conditions: when we prove $\varphi \Rightarrow \psi$, we've shown that *one way* for ψ to be true is for φ to be true. But there might be other reasons that φ is true! Here's an example of a fallacious argument that uses this bogus logic:

Example 4.35: Figure out what's wrong with this logic.
Find the error in this argument:

Premise A. If it's raining, then the computer burning will be postponed.

Premise B. The computer burning was postponed.

Conclusion. Therefore, it's raining.

Solution. This fallacious argument is an example of affirming the consequent. The first premise here merely says that the computer burning will be postponed if it rains; it does not say that rain is the only reason that the burning could be postponed. There may be many other reasons why the burning might be delayed: for example, the inability to find a match, the sudden vigilance of the health and safety office, or a last-minute stay of execution by the owner of the computer.

Fallacy: denying the hypothesis

Denying the hypothesis is a closely related fallacy to affirming the consequent: we prove (correctly) that $\psi \Rightarrow \varphi$, and we prove (correctly) that $\neg\psi$; we then (fallaciously) conclude $\neg\varphi$. This logic is buggy for essentially the same reason as affirming the consequent. (In fact, denying the hypothesis is the contrapositive of affirming the consequent—and therefore a fallacy too, because it's logically equivalent to a fallacy.)

The implication $\psi \Rightarrow \varphi$ means that one way of φ being true is for ψ to be true, but it does *not* mean that there is no other way for φ to be true. Here's an example of a fallacious argument of this type:

Example 4.36: Figure out what's wrong with this logic.
Find the error in this argument:

Premise A. If you have resolved the P-versus-NP question, then you are famous.

Premise B. You have not resolved the P-versus-NP question.

Conclusion. Therefore, you are not famous.

Solution. This fallacious argument is an example of denying the hypothesis. The first premise says that one way to be famous is to resolve the P-versus-NP question (see p. 3-32 for a brief description of this problem), but it does not say that resolving the P-versus-NP question is the only way to be famous. For example, you could be famous by being the President of the United States or by founding Google.

Fallacy: false dichotomy

A *false dichotomy* or *false dilemma* is a fallacious argument in which two nonexhaustive alternatives are presented as exhaustive (without acknowledgement that there are any unmentioned alternatives).

Example 4.37: False Dichotomy.
The flawed step in Example 4.33 can be interpreted as a false dilemma: implicitly, that proof relied on the assertion that if k evenly divides n, then

$$k \in F_{\text{small}} = \left\{\text{factors of } n \text{ that are less than } \sqrt{n}\right\} \text{ or}$$
$$k \in F_{\text{big}} = \left\{\text{factors of } n \text{ that are greater than } \sqrt{n}\right\}.$$

But of course the third unmentioned possibility is that $k = \sqrt{n}$.

(The classical false dichotomy, often found in political rhetoric, is "either you're with us or you're against us": actually, you might be neutral on the issue, and therefore neither "with" nor "against" us!)

Fallacy: begging the question

We wish to prove a proposition φ. A purported proof of φ that *begs the question* is one that assumes φ along the way. That is, the "proof" assumes precisely the thing that it purports to prove, and thus actually proves $\varphi \Rightarrow \varphi$. Although this type of fallacious reasoning sounds ridiculous, the assumption of the desired result can be very subtle; you must be vigilant to catch this type of error. Here's an example of a fallacious argument of this kind:

Example 4.38: Figure out what's wrong with this logic.
Find the error in this proof:

False Theorem: Let n be a positive integer such that $n + n^2$ is even. Then n is odd.

Proof. Assume the antecedent—that is, assume that $n + n^2$ is even. Let k be the integer such that $n = 2k + 1$. Then

$$\begin{aligned} n + n^2 &= 2k + 1 + (2k + 1)^2 \\ &= 2k + 1 + 4k^2 + 4k + 1 \\ &= 4k^2 + 6k + 2 \\ &= 2 \cdot (2k^2 + 3k + 1), \end{aligned}$$

which is even because it is equal to 2 times an integer. But $n^2 = (2k + 1)^2 = 4k^2 + 4k + 1$ is odd (because $4k^2$ and $4k$ are both even). Therefore

$$n = \underset{\substack{\text{As argued above,} \\ n + n^2 = 2 \cdot (2k^2 + 3k + 1), \\ \text{which is an even number.}}}{n + n^2} - \underset{\substack{\text{As argued above,} \\ n^2 = 4k^2 + 4k + 1, \\ \text{which is an odd number.}}}{n^2} .$$

An even number less an odd number is an odd number, which implies that n must be odd too. □

Solution. The problem comes very early in the "proof," in the sentence *Let k be the integer such that $n = 2k+1$*. But this statement implicitly assumes that n is an odd integer; an integer k such that $n = 2k+1$ exists *only if* n is odd. So the proof begs the question: it assumes that n is odd, and—after some algebraic shenanigans—concludes that n is odd.

Problem-solving tip: Even without identifying the specific bug in Example 4.38, we could notice that there's something fishy by doing the post-proof plausibility check to make sure that all premises were actually used. The "proof" states that it is assuming the antecedent, but we actually *derived* the fact that $n + n^2$ is even. So we never used that assumption in the "proof." (In fact, $n + n^2$ is even for *any* positive integer n.) But, because we didn't use the assumption, the same proof works just as well without it as an assumption, so we could use the same "proof" to establish this claim instead:

Patently False Theorem: *Let n be a positive integer. Then n is odd.*

Given that this new claim is obviously false, there *must* be a bug in the proof. The only challenge is to find that bug.

Other fallacies

We have discussed a reasonably large collection of logical fallacies into which some less-than-careful or less-than-scrupulous proof writers may fall. It is always your job to be vigilant—both when reading proofs written by others, and in developing your own proofs—to avoid fallacious reasoning.

But there are many other types of flaws in arguments that more typically arise in informal contexts; these are the kinds of flawed arguments that are—sadly—often used in politics. (Some of them have analogues in more mathematical settings, too.)

> This listing is just a brief outline of some of the many invalid techniques of persuasion/propaganda; a much more extensive and thorough list is maintained by Gary Curtis at `http://www.fallacyfiles.org/`. You might also be interested in books that catalogue fallacious techniques of argument, such as [101].

Here are a few examples of other types of fallacies that you may encounter in "real-world" arguments:

Confusing correlation and causation. Phenomena A and B are said to be *(positively) correlated* if they occur together more often than their individual frequencies would predict. (See Chapter 10.) But just because A and B are correlated does *not* mean that one *causes* the other! For example, the user population of TikTok is much younger than is the population at large. We could say, correctly, that *Being young is correlated with using TikTok.* But *Using TikTok makes you young* is an obviously absurd conclusion. (Some correlation-versus-causation mistakes are subtler; your reaction to *Being young makes you use TikTok* is probably less virulent, but it is equally unsupported by the facts that we've cited here.) Always be wary when attempting to infer causal relationships!

Ad hominem attacks. An *ad hominem* (Latin *ad hominem*: "to the man") attack ignores the logical argument and speaks to the arguer: *Bob doesn't know the difference between contrapositive and converse, and he says that n is prime. So n must be composite.*

Equivocation (also known as "shifting language"). This type of argument relies on changes in the meanings of the words/variables in an argument. This shift can be grammatical: *Time waits for no man*, and *no man is an island*; therefore, *time waits for an island.* Or it can be in the semantics of a particular word: 1024 *is a prime example of an exact power of two*, and *prime numbers are evenly divisible only by* 1 *and themselves*; therefore, 1024 *is not divisible by* 4. A similar type of fallacy can also occur when a variable in a proof is introduced to mean two different things.

Computer Science Connections

The Cost of Missing Proofs: Some Famous Bugs in CS

There's an apocryphal story that the first use of the word "bug" to refer to a flaw in a computer system was in the 1940s: Grace Hopper, a rear admiral in the US Navy and a pioneer in early programming, found a moth (a literal, physical moth) jamming a piece of computer equipment and causing a malfunction. (The story is true, but the *Oxford English Dictionary* reports uses of "bug" to refer to a technological fault dating back to Thomas Edison in the late 1800s.) But there are many other stories of bugs that are both more important and more true. When a computer system "almost" works—when there's no proof that it works correctly in all circumstances—there can be grave repercussions, in dollars and lives lost. Here are a few of the most famous, and most costly, bugs in history:

The Pentium division bug. In 1994, Thomas Nicely, at the time a math professor at Lynchburg College, discovered a hardware bug in Intel's new Pentium chip that caused incorrect results when some floating-point numbers were divided by certain other floating-point numbers. The flaw resulted from a lookup table for the division operation that was missing a handful of entries. Although the range of numbers that were incorrectly divided was limited, the resulting brouhaha led to a full Pentium recall and about $500 million (in 1994 dollars) in losses for Intel.

The Ariane 5 rocket. The European Space Agency's rocket, carrying a $400 million (in 1996 dollars) payload of satellites, exploded 40 seconds into its first flight, in 1996. The rocket had engaged its self-destruct system, which was correctly triggered when it strayed from its intended trajectory. But the altered trajectory was caused by a sequence of errors, including an *integer overflow* error: the rocket's velocity was too big to fit into the 16-bit variable that was being used to store it. (An Ariane 5 rocket was much faster than the Ariane 4 rockets for which the code was originally developed.) Embarrassingly, the overflow caused a subsystem to output a diagnostic error code that was interpreted as navigation data. More embarrassingly still, this entire subsystem played no role in navigation after liftoff, and would have caused no harm if it were just turned off.

The Therac-25. The Therac-25 was a medical device in use in the mid-1980s that treated tumors with a focused beam of radiation. The device fired a concentrated X-ray beam of extremely high dosage into a diffuser that would reduce the beam's intensity to the desired levels before it was directed at the patient. But it turned out that a particularly fast touch-typing operator could cause the high-intensity beam to be fired without the diffuser in place: hitting `enter` at the precise moment that an internal variable reset to zero caused the undiffused beam to be fired. (This kind of bug is called a *race condition*, in which the output of a system depends crucially on the precise timing of events like operator input.) At least five patients were killed by radiation overdoses.

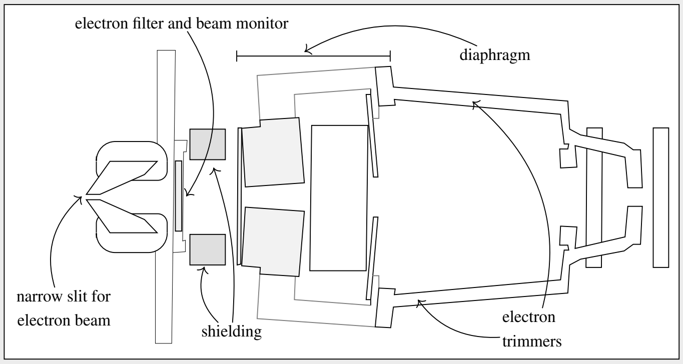

electron filter and beam monitor
diaphragm
narrow slit for electron beam
shielding
electron trimmers

Figure 4.37 A schematic drawing of a Therac-25, adapted from the article "Therac 25: A New Medical Accelerator Concept" [124].

The Pentium, Ariane 5, and Therac-25 bugs are among the most well-known and worst "bug-like" errors in the history of CS. For a list of one person's view of the ten worst bugs in history, including these three and some other sordid tales, see [49]. (There are other kinds of pernicious problems with software systems that you wouldn't necessarily call "bugs"—maybe "ethical lapses" is a better descriptor—that are deeply problematic in other ways. For example, a system's impacts may be different for different kinds of users, perhaps reinforcing or exacerbating existing gender-, race-, or socioeconomics-based biases.) For more on the particular bugs described above: you can read more about the Pentium issue and its aftermath in [100]; the official report on the Ariane 5 is available as [80]; and an excellent analysis of what went wrong with Therac-25 can be found in two books by Nancy Levenson [77, 78].

EXERCISES

Identify whether the following arguments are valid or fallacious. Justify your answers.

4.89 *Premise A:* Every programming language that uses garbage collection is slow.

Premise B: C does not use garbage collection.

Conclusion: Therefore, C is slow.

4.90 *Premise A:* If a piece of software is written well, then it was built with its user in mind.

Premise B: The Firefox web browser is a piece of software that was written with its user in mind.

Conclusion: Therefore, the Firefox web browser is written well.

4.91 *Premise A:* If a processor overheats while multiplying, then it overheats while computing square roots.

Premise B: The xMax processor does not overheat while computing square roots.

Conclusion: Therefore, the xMax processor does not overheat while multiplying.

4.92 *Premise A:* Every data structure is either slow at insertions or lookups.

Premise B: The data structure called the *Hackmatack tree* is slow at insertions.

Conclusion: Therefore, the Hackmatack tree is slow at lookups.

4.93 *Premise A:* Every web server has an IP address.

Premise B: `www.cia.gov` is a web server.

Conclusion: Therefore, `www.cia.gov` has an IP address.

4.94 *Premise A:* If a computer system is hacked, then there was user error or the system had a design flaw.

Premise B: A computer at NASA was hacked.

Premise C: That computer did not have a design flaw.

Conclusion: Therefore, there was user error.

In the next few problems, you will be presented with a false claim and a bogus proof of that false claim. For each, you'll be asked to (a) identify the precise error in the proof, and (b) give a counterexample to the claim. (Note that saying why the claim is false does not address (a) in the slightest—it would be possible to give a bogus proof of a true claim!)

4.95 State precisely what's wrong with the following proof.

> **False Claim 1.** Let n be a positive integer and let $p, q \in \mathbb{Z}^{\geq 2}$, where p and q are prime. If n is evenly divisible by both p and q, then n is also evenly divisible by pq.
>
> *Bogus proof.* Because $p \mid n$, there exists a positive integer k such that $n = pk$. Thus, by assumption, we know that $q \mid pk$. Because p and q are both prime, we know that p does not evenly divide q, and thus the only way that $q \mid pk$ can hold is if $q \mid k$. Hence $k = q\ell$ for some positive integer ℓ, and thus $n = pk = pq\ell$. Therefore $pq \mid n$. □

4.96 Give a counterexample to False Claim 1 from Exercise 4.95.

4.97 State precisely what's wrong with the following proof.

> **False Claim 2.** 721 is prime.
>
> *Bogus proof.* In Example 4.7, we proved that $n! + 1$ is not evenly divisible by any k satisfying $2 \leq k \leq n$. Observe that $6! = 720$. Therefore, $721 = 6! + 1$ isn't evenly divisible by any integer between 2 and 720 inclusive, and therefore 721 is prime. □

4.98 *Without using a calculator*, disprove False Claim 2 from Exercise 4.97.

4.99 *Without using a calculator*, find an integer n such that $n! + 1$ *is* prime.

4.100 State precisely what's wrong with the following proof.

> **False Claim 3.** $\frac{\sqrt{2}}{4}$ and $\frac{8}{\sqrt{2}}$ are both rational.
>
> *Bogus proof.* In Example 4.11, we proved that if x and y are rational then xy is rational too. Here, let $x = \frac{\sqrt{2}}{4}$ and $y = \frac{8}{\sqrt{2}}$. Then $xy = \frac{\sqrt{2}}{4} \cdot \frac{8}{\sqrt{2}} = \frac{8\sqrt{2}}{4\sqrt{2}} = 2$. So $xy = 2$ is rational, and x and y are too. □

4.101 Prove that $\frac{8}{\sqrt{2}}$ isn't rational (which shows that False Claim 3 from Exercise 4.100 is false).

4.102 State precisely what's wrong with the following proof.

> **False Claim 4.** Let n be any integer. Then $12 \mid n$ if and only if $12 \mid n^2$.
>
> *Bogus proof.* We'll follow the same outline as in Example 4.18. We proceed by mutual implication.
>
> First, assume that $12 \mid n$. Then, by definition, there exists an integer k such that $n = 12k$. Therefore $n^2 = (12k)^2 = 12 \cdot (12k^2)$. Thus $12 \mid n^2$ too.
>
> Second, we must show the converse: if $12 \mid n^2$, then $12 \mid n$. We prove the contrapositive. Assume that $12 \nmid n$. Then there exist integers k and $r \in \{1, \ldots, 11\}$ such that $n = 12k + r$. Therefore $n^2 = (12k + r)^2 = 144k^2 + 24kr + r^2 = 12(12k^2 + 2kr) + r^2$. Because $r < 12$, adding r^2 to a multiple of 12 does not result in another multiple of 12. Thus $12 \nmid n^2$. □

4.103 Disprove False Claim 4 from Exercise 4.102.

4.104 State precisely what's wrong with the following proof.

> **False Claim 5.** $\sqrt{4}$ is irrational.
>
> *Bogus proof.* We'll follow the same outline as Example 4.20. Our proof is by contradiction.
>
> Assume that $\sqrt{4}$ is rational. Therefore, there exist integers n and $d \neq 0$ such that $\frac{n}{d} = \sqrt{4}$, where n and d have no common divisors.
>
> Squaring both sides yields that $\frac{n^2}{d^2} = 4$, and therefore that $n^2 = 4d^2$. Because $4d^2$ is divisible by 4, we know that n^2 is divisible by 4. Therefore, by the same logic as in Example 4.18, we have that n is itself divisible by 4.
>
> Because n is divisible by 4, there exists an integer k such that $n = 4k$, which implies that $n^2 = 16k^2$. Thus $n^2 = 16k^2$ and $n^2 = 4d^2$, so $d^2 = 4k^2$. Hence d^2 is divisible by four.
>
> But now we have a contradiction: we assumed that $\frac{n}{d}$ was in lowest terms, but we have now shown that n^2 and d^2 are both divisible by 4, and therefore both n and d must be even! Thus the original assumption was false, and $\sqrt{4}$ is irrational. □

4.105 State precisely what's wrong with the following proof.

False Claim 6. $3 \leq 2$.

Bogus proof. Let x and y be arbitrary nonnegative numbers. Because $y \geq 0$ implies $-y \leq y$, we can add x to both sides of this inequality to get

$$x - y \leq x + y. \tag{1}$$

Similarly, adding $y - 3x$ to both sides of $-x \leq x$ yields

$$y - 4x \leq y - 2x. \tag{2}$$

Observe that whenever $a \leq b$ and $c \leq d$, we know that $ac \leq bd$. So we can combine (1) and (2) to get

$$(x - y)(y - 4x) \leq (x + y)(y - 2x). \tag{3}$$

Multiplying out and then combining like terms, we have

$$xy - 4x^2 - y^2 + 4xy \leq xy - 2x^2 + y^2 - 2xy, \text{ and} \tag{4}$$

$$6xy \leq 2x^2 + 2y^2. \tag{5}$$

This calculation was valid for any $x, y \geq 0$. For $x = y = \sqrt{1/2}$, we have $xy = x^2 = y^2 = \left(\sqrt{1/2}\right)^2 = \frac{1}{2}$. Plugging into (5), we have

$$\tfrac{6}{2} \leq \tfrac{2}{2} + \tfrac{2}{2}. \tag{6}$$

In other words, we have $3 \leq 2$. □

Computer vision *is the subfield of computer science devoted to developing algorithms that can "understand" images. One of the more controversial applications of computer vision is in* facial recognition, *which can be used for some very positive applications but also in some ethically dubious ones too. One of the most ethically concerning properties of many widely deployed facial recognition algorithms is that their accuracy rates can be very different across different demographic groups, particularly based on race: Black people's faces are systematically less accurately recognized by these algorithms than the faces of white people. (See p. 8-19 for more on this kind of* algorithmic bias.*) These differential accuracies are absolutely real, but it turns out that there are some subtleties to calculating accuracy rates, which can lead to some counterintuitive conclusions, as you'll explore in the next two exercises. (Although some applications, like facial recognition, raise major ethical questions regarding fairness and privacy, there are other uses tremendously useful applications of computer vision that seem to be much more clearly world-improving. Deployed applications of computer vision include: assessing the condition of crops using aerial imagery from drones to help a farmer better allocate irrigation and fertilizer; doing a first-cut diagnosis on medical images to categorize x-rays or MRIs into "normal" or "possibly anomalous" clusters (the latter being examined by human experts); and automating the sorting of most packages in a post office facility by using optical character recognition on the address labels.)*

4.106 Suppose that we have two disjoint groups of people, say B (Black) and W (White). Further, suppose that we have two algorithms, $\mathcal{F}$ and $\mathcal{G}$, that we have employed on two different cameras that are being used to allow/deny access to a workplace. Suppose that algorithm $\mathcal{F}$ is deployed on Camera I. It makes the correct decision on 75% of the people in W at Camera I and 60% of the people in B at Camera I. (That is, when a person from W arrives at Camera I, algorithm $\mathcal{F}$ correctly decides whether to grant her access 75% of the time.) Algorithm $\mathcal{G}$, deployed at Camera II, makes the correct decision on 70% of the people in W and 50% of the people in B. The following claim seems obvious, because Algorithm $\mathcal{F}$ performed better for both members of B and W. But the claim is false, as you'll show! (The falsehood of this claim (for example, in the scenario in Exercise 4.107) is called *Simpson's Paradox* because the behavior is so counterintuitive.) State precisely where the purported proof goes wrong.

False Claim 7. Algorithm $\mathcal{F}$ is right a higher fraction of the time (overall, combining both sets of people) than Algorithm $\mathcal{G}$.

Bogus proof. Observe that Algorithm $\mathcal{F}$ had a better success probability than $\mathcal{G}$ with members of B, and also had a better success probability with members of W. Therefore Algorithm $\mathcal{F}$ was right a higher fraction of the time (in total, for all people—that is, members of $B \cup W$—than Algorithm $\mathcal{G}$). □

4.107 Consider the scenario described in Exercise 4.106. Suppose that there were 100 members of W and 100 members of B who went by Camera I. Suppose that 1000 members of W and 100 members of B went by Camera II. Calculate the success rate for Algorithm $\mathcal{F}$ at Camera I, over all people. Do the same for Algorithm $\mathcal{G}$ at Camera II.

4.108 Here is a (nonobviously) bogus proof of the (obviously) bogus claim that $0 = 1$. Identify precisely the flaw in the argument.

Proof that $0 = 1$. Consider the four shapes in Figure 4.38a, and the two arrangements thereof in Figure 4.38b. The area of the triangle in the first configuration is $\frac{13 \cdot 5}{2} = \frac{65}{2}$, as it forms a right triangle with height 5 and base 13. But the second configuration also forms a right triangle with height 5 and base 13 as well, and therefore it too has area $\frac{65}{2}$. But the second configuration has one unfilled square in the triangle, and thus we have

$$\begin{aligned}0 &= \tfrac{65}{2} - \tfrac{65}{2}\\ &= \text{area of the second bounding triangle} - \text{area of the first bounding triangle}\\ &= (1 + \text{area of four constituent shapes}) - (\text{area of four constituent shapes})\\ &= 1.\end{aligned}$$

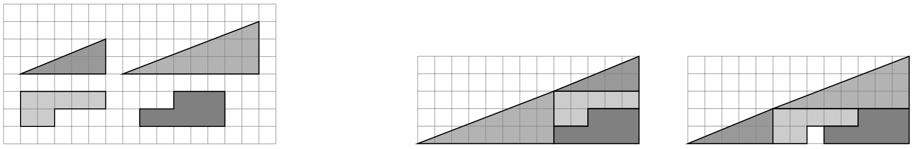
□

4.109 The following two statements are theorems from geometry that you may recall from high school:

- the angles of a triangle sum to precisely 180°.
- if the three angles of triangle T_1 are precisely equal to the three angles of T_2, then T_1 and T_2 are *similar*, and their sides are in the same ratios. (That is, if the side lengths of T_1 are a, b, c and the side lengths of T_2 are x, y, z, then $\frac{a}{x} = \frac{b}{y} = \frac{c}{z}$.)

These statements are theorems, but they're used in the following utterly bogus "proof" of the Pythagorean Theorem (actually one that was published, in 1896!). State precisely what's wrong with the following purported proof of the Pythagorean Theorem.

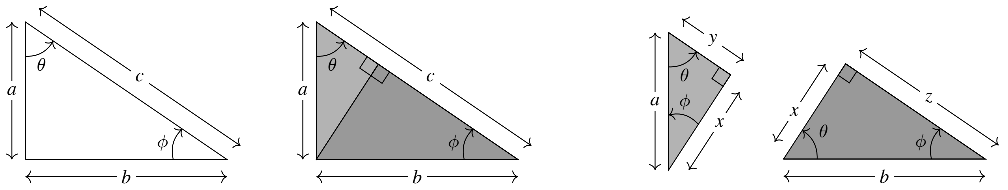
Bogus proof. Consider an arbitrary right triangle. Let the two legs and hypotenuse, respectively, have length a, b, and c, and let the angles between the legs and the hypotenuse be given by θ and $\phi = 90° - \theta$. (See Figure 4.39a.) Draw a line perpendicular to the hypotenuse to the opposite vertex, dividing the interior of the triangle into two separate sections, which are shaded with different colors in Figure 4.39b. Observe that the unlabeled angle within the smaller shaded interior triangle must be $\phi = 90° - \theta$, because the other angles of the smaller shaded interior triangle are (just like for the enclosing triangle) 90° and θ. Similarly, the unlabeled angle within the larger shaded interior triangle must be θ. Therefore we have three similar triangles, all with angles 90°,



(a) The shapes.

(b) Two configurations.

Figure 4.38 Some shapes and their arrangements, for Exercise 4.108.



(a) The original triangle.

(b) A perpendicular line added from the hypotenuse to the opposite vertex.

(c) The original triangle, split into two smaller right triangles. Note that $c = y + z$.

Figure 4.39 Diagrams for Exercise 4.109.

θ, and ϕ. Call the lengths of the previously unnamed sides x, y, and z as in Figure 4.39c. Now we can assemble our known facts. By assumption,

$$a^2 = x^2 + y^2, \qquad b^2 = x^2 + z^2, \text{ and } \qquad (y+z)^2 = a^2 + b^2,$$

which we can combine to yield

$$(y+z)^2 = 2x^2 + y^2 + z^2. \tag{1}$$

Expanding $(y+z)^2 = y^2 + 2yz + z^2$ and subtracting common terms from both sides, we have

$$2yz = 2x^2, \tag{2}$$

which, dividing both sides by two, yields

$$yz = x^2. \tag{3}$$

But (3) is immediate: we know that

$$\tfrac{x}{y} = \tfrac{z}{x} \tag{4}$$

because the two shaded triangles are similar, and therefore the two triangles have the same ratio of the length of the hypotenuse to the length of the longer leg. Multiplying both sides of (4) by xy gives us $x^2 = yz$, as desired. □

4.6 Chapter at a Glance

Error-Correcting Codes

Although the main purpose of this section was to introduce proofs, here's a brief summary of the definitions and results about error-correcting and error-detecting codes, too.

A *code* is a set $\mathcal{C} \subseteq \{0,1\}^n$, where $|\mathcal{C}| = 2^k$ for some integer $1 \leq k \leq n$. A *message* is an element of $\{0,1\}^k$; the elements of $\mathcal{C}$ are called *codewords.* Thinking of an error as a single bit being flipped from 0 to 1 or from 1 to 0, consider any codeword $c \in \mathcal{C}$ and for any sequence of up to ℓ errors applied to c to produce c'. The code $\mathcal{C}$ can *detect* $\ell \geq 0$ errors if we can always correctly report "error" or "no error," and can *correct* ℓ errors if we can always correctly identify that c was the original codeword.

The *Hamming distance* between strings $x, y \in \{0,1\}^n$, denoted $\Delta(x, y)$, is the number of positions i in which $x_i \neq y_i$. The *minimum distance* of a code $\mathcal{C}$ is the smallest Hamming distance between two distinct codewords of $\mathcal{C}$. The *rate* of a code with k-bit messages and n-bit codewords is k/n. If the minimum distance of a code $\mathcal{C}$ is $2t + 1$ for an integer t, then $\mathcal{C}$ can detect $2t$ errors and correct t errors.

The Repetition_ℓ *code* creates codewords via the ℓ-fold repetition of the message. This code has rate $1/\ell$ and minimum distance ℓ. The *Hamming code* creates 7-bit codewords from 4-bit messages by adding three different parity bits to the message. This code has rate $4/7$ and minimum distance 3. Any code with messages of length 4 and minimum distance 3 has codewords of length ≥ 7. (Thus the Hamming code has the best possible rate among all such codes.) We can prove this result via a "sphere-packing" argument and a proof by contradiction.

Proofs and Proof Techniques

A *proof* of a claim φ is a convincing argument that φ is true. (A proof should be written with its audience in mind.) A variety of useful proof techniques can be employed to prove a given claim φ:

- *direct proof:* we prove φ by repeatedly inferring new facts from known facts to eventually conclude φ. (Sometimes we divide a proof into multiple cases, or "assume the antecedent," where we prove $p \Rightarrow q$ by assuming p and deriving q.)

You may also prove φ by proving a claim logically equivalent to φ:

- *proof by contrapositive:* to prove $p \Rightarrow q$, we instead prove $\neg q \Rightarrow \neg p$.
- *proof by contradiction* (or *reductio ad absurdum*): to prove φ, we instead prove that $\neg\varphi \Rightarrow$ False—that is, we prove that $\neg\varphi$ leads to an absurdity.

We say that $y \in S$ with $\neg P(y)$ is a *counterexample* to the claim $\forall x \in S : P(x)$. A *proof by construction* of the claim $\exists x \in S : P(x)$ proceeds by constructing a particular $y \in S$ and proving that $P(y)$. A *nonconstructive proof* establishes $\exists x \in S : P(x)$ without giving an explicit $y \in S$ for which $P(x)$—for example, by proving $\exists x \in S : P(x)$ by contradiction.

The process of developing a proof requires persistence, open-mindedness, and creativity. Here's a helpful three-step plan to use when developing a new proof: (1) understand what you're trying to do (checking definitions and small examples); (2) do it (by trying the proof techniques catalogued here, and thinking about analogies from similar problems that you've solved previously); and (3) think about what you've done (reflecting on and trying to improve your proof). Remember that writing a proof is a form of writing! Be kind to your reader.

Some Examples of Proofs

We can use these proof techniques to establish a wide variety of facts—about arithmetic, propositional logic, geometry, prime numbers, and computability. For more extensive examples, see Section 4.4. We'll highlight one result: there are problems that we can formally define, but that cannot be solved by any computer program; these problems (including the *Halting Problem*) are called *uncomputable.*

Common Errors in Proofs

A *valid* inference is one whose conclusion is always true as long as the facts that it relies on were true. An *invalid* inference is one in which the conclusion can be false *even if the premises are all true.* An invalid, or fallacious, argument can also be called a *logical fallacy* or just a *fallacy.* In a correct proof, of course, every step is valid.

Perhaps the most common error in a proof is simply asserting that a fact φ follows from previously established facts, when actually φ is not implied by those facts. Other common types of fallacious reasoning are structural errors that involve purporting to prove a statement φ, but instead proving a statement that is not logically equivalent to φ. (For example, the *fallacy of proving true*: a "proof" of φ that assumes φ and proves True. But $\varphi \Rightarrow$ True is true regardless of the truth of φ, so this purported proof proves nothing.) Be vigilant; do not let anyone—yourself or others!—get away with fallacious reasoning.

Key Terms and Results

Key Terms

Error-Correcting Codes

- Hamming distance
- code, message, codeword
- error-detecting/correcting code
- minimum distance, rate
- repetition code
- Hamming code

Proofs and Proof Techniques

- proof
- proof techniques:
  - direct proof
  - proof by contrapositive
  - proof by contradiction
- counterexample
- constructive/nonconstructive proof

Some Examples of Proofs

- conjunctive/disjunctive normal form
- uncomputability
- the Halting Problem

Valid and Fallacious Arguments

- valid argument
- fallacious/invalid argument; fallacy
- fallacy: proving true
- fallacy: affirming the consequent
- fallacy: denying the hypothesis
- fallacy: false dichotomy
- fallacy: begging the question

Key Results

Error-Correcting Codes

1. If the minimum distance of a code $\mathcal{C}$ is $2t+1$ for an integer $t \geq 0$, then $\mathcal{C}$ can detect $2t$ errors and correct t errors.
2. For 4-bit messages and minimum distance 3, there exist codes with rate $\frac{1}{3}$ (such as the REPETITION$_3$ code) and with rate $\frac{4}{7}$ (such as the Hamming code), but not with rate better than $\frac{4}{7}$.

Proofs and Proof Techniques

1. You can prove a claim φ with a direct proof, or by instead proving a different claim that is logically equivalent to φ. Examples include proofs by contrapositive and proofs by contradiction.
2. A useful three-step process for developing proofs is: (1) understand what you're trying to do; (2) do it; and (3) think about what you've done. All three steps are important, and doing each will help with the other steps.
3. Writing a proof is a form of writing.

Some Examples of Proofs

1. All logical propositions are equivalent to propositions in conjunctive/disjunctive normal form, or using only *nand*.
2. There are infinitely many prime numbers.
3. There are problems that can be specified completely formally that are uncomputable (that is, cannot be solved by any computer program). The Halting Problem is one example.

Valid and Fallacious Arguments

1. There are many common mistakes in proofs that are centered on several types of fallacious reasoning. These fallacies are essentially all the result of purporting to prove a statement φ by instead proving a statement ψ, where ψ fails to be logically equivalent to φ.