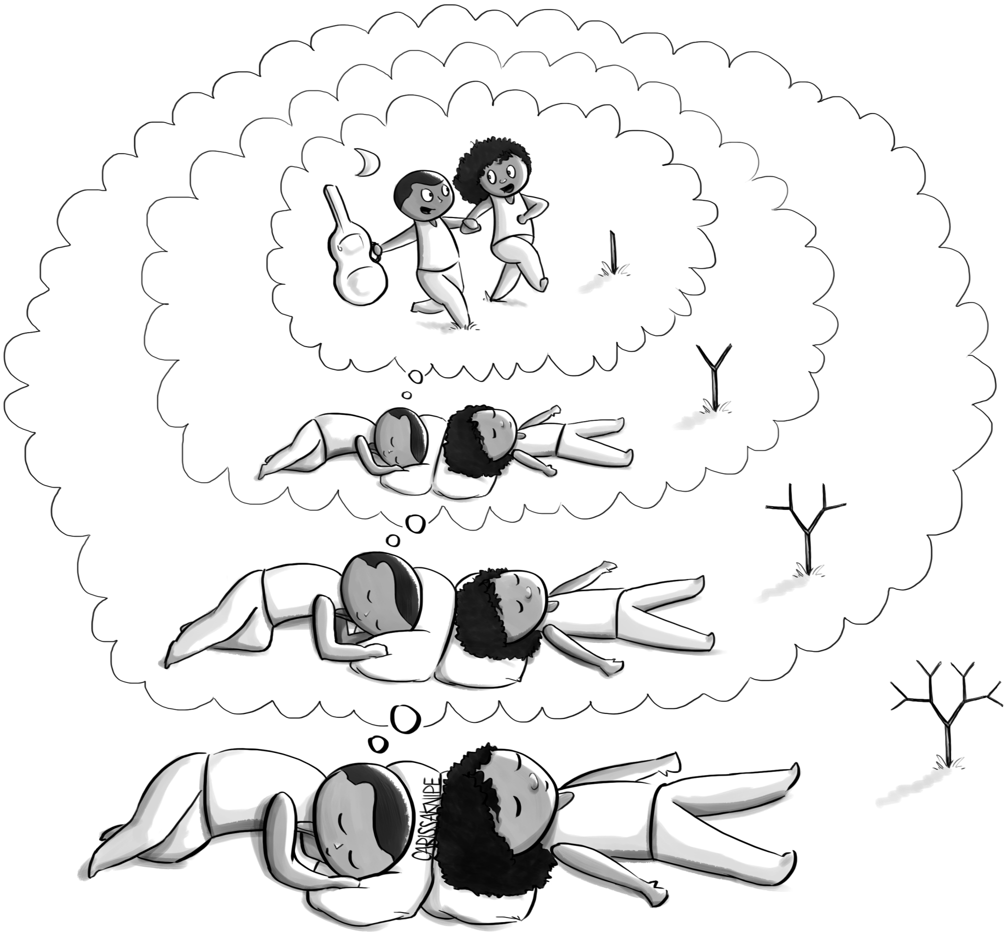
5 Mathematical Induction



In which our heroes wistfully dream about having dreams about dreaming about a very simple and pleasant world in which no one sleeps at all.

5.1 Why You Might Care

> *Chaque vérité que je trouvais étant une règle qui me servait après à en trouver d'autres.*
> Each truth discovered was a rule available in the discovery of subsequent ones.
>
> René Descartes (1596–1650)
> *Le Discours de la méthode [Discourse on the Method]* (1637)

Recursion is a powerful technique in computer science. If we can express a solution to problem X in terms of solutions to smaller instances of the same problem X—and we can solve X directly for the "smallest" inputs—then we can solve X for all inputs. There are many examples. We can sort an array A by sorting the left half of A and the right half of A and merging the results together; 1-element arrays are by definition already sorted. (That's *merge sort.*) We can search for an element x in a sorted list A by comparing x to the middle element of A and, if they don't match, searching for x in the appropriate half of A, left or right; an empty list A by definition does not contain x. (That's *binary search.*) We can build an efficient data structure for storing and searching a set of keys by selecting one of those keys k, and building two such data structures for keys $< k$ and for keys $> k$; to search for a key x, we compare x to k and search for x in the appropriate substructure. And an empty data structure can store an empty set of keys. (That's a *binary search tree.*) And many other things are best understood recursively, too: factorials, the Fibonacci numbers, fractals (see Figure 5.1), and finding the median element of an unsorted array, for example.

Mathematical induction is a technique for proofs that is directly analogous to recursion: to prove that a property $P(n)$ holds for all nonnegative integers n, we prove that $P(0)$ is true, and we prove that for an arbitrary $n \geq 1$, if $P(n-1)$ is true, then $P(n)$ is true too. The proof of $P(0)$ is called the *base case,* and the proof that $P(n-1) \Rightarrow P(n)$ is called the *inductive case.* In the same way that a recursive solution to a

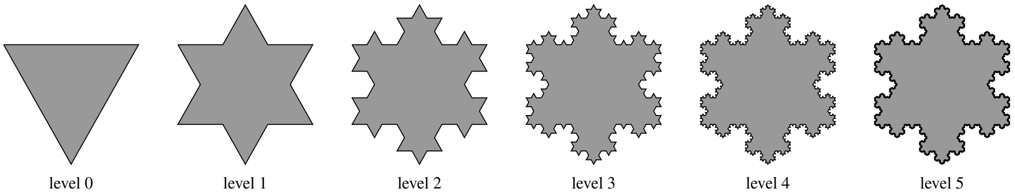


Figure 5.1 The Von Koch Snowflake fractal, shown at several different levels. A level-ℓ snowflake consists of three level-ℓ lines. A level-0 line is ——— ; a level-ℓ line consists of four level-$(\ell - 1)$ lines arranged in the shape ‗/\‗.

problem relies on solutions to a smaller instance of the same problem, an inductive proof of a claim relies on proofs of a smaller instance of the same claim.

A full understanding of recursion depends on a thorough understanding of mathematical induction. And many other applications of mathematical induction will arise throughout the book: analyzing the running time of algorithms, counting the number of bitstrings that have a particular form, and many others.

In this chapter, we will introduce mathematical induction, including a few variations and extensions of this proof technique. We will start with the "vanilla" form of proofs by mathematical induction (Section 5.2). We will then introduce *strong induction* (Section 5.3), a form of proof by induction in which the proof of $P(n)$ in the inductive case may rely on the truth of all of $P(0)$, $P(1)$, …, and $P(n-1)$ instead of just on $P(n-1)$. Finally, we will turn to *structural induction* (Section 5.4), a form of inductive proof that operates directly on recursively defined structures like linked lists, binary trees, or well-formed formulas of propositional logic.

5.2 Proofs by Mathematical Induction

> So if you find nothing in the corridors open the doors, if you find nothing behind these doors there are more floors, and if you find nothing up there, don't worry, just leap up another flight of stairs. As long as you don't stop climbing, the stairs won't end, under your climbing feet they will go on growing upwards.
>
> Franz Kafka (1883–1924)
> *Fürsprecher [Advocates]* (c. 1922)

The *principle of mathematical induction* says the following: to prove that a statement $P(n)$ is true for all nonnegative integers n, we can prove that P "starts being true" (the *base case*) and that P "never stops being true" (the *inductive case*). We'll start with the formal definition of a proof based on this principle, and then consider a wide variety of examples.

5.2.1 An Overview of Proofs by Mathematical Induction

Formally, a proof by mathematical induction proceeds as follows:

Definition 5.1: Proof by mathematical induction.
Suppose that we want to prove that $P(n)$ holds for all $n \in \mathbb{Z}^{\geq 0}$. To give a *proof by mathematical induction* of $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$, we prove two things:

1 the *base case:* prove $P(0)$.
2 the *inductive case*: for every $n \geq 1$, prove $P(n-1) \Rightarrow P(n)$.

When we've proven both the base case and the inductive case as in Definition 5.1, we have established that $P(n)$ holds for all $n \in \mathbb{Z}^{\geq 0}$. Here's an example to illustrate how the base case and inductive case combine to establish this fact:

Example 5.1: Proving $P(4)$ from a base case and inductive case.
Suppose we've proven both the base case ($P(0)$) and the inductive case ($P(n-1) \Rightarrow P(n)$, for any $n \geq 1$) as in Definition 5.1. Why do these two facts establish that $P(n)$ holds for all $n \in \mathbb{Z}^{\geq 0}$? For example, why do they establish $P(4)$?

Solution. Here is a proof of $P(4)$, using the base case once and the inductive case four times. (At each stage we use *Modus Ponens*—which, as a reminder, states that from $p \Rightarrow q$ and p, we can conclude q.)

$$\text{We know } P(0) \qquad \textit{base case} \tag{5.1}$$

$$\text{and we know } P(0) \Rightarrow P(1) \qquad \textit{inductive case, with } n = 1 \tag{5.2}$$

$$\text{and thus we can conclude } P(1). \qquad \textit{(5.1), (5.2), and Modus Ponens} \tag{5.3}$$

$$\text{We know } P(1) \Rightarrow P(2) \qquad \textit{inductive case, with } n = 2 \tag{5.4}$$

and thus we can conclude $P(2)$. *(5.3), (5.4), and Modus Ponens* (5.5)

We know $P(2) \Rightarrow P(3)$ *inductive case, with $n = 3$* (5.6)

and thus we can conclude $P(3)$. *(5.5), (5.6), and Modus Ponens* (5.7)

We know $P(3) \Rightarrow P(4)$ *inductive case, with $n = 4$* (5.8)

and thus we can conclude $P(4)$. *(5.7), (5.8), and Modus Ponens* (5.9)

This sequence of inferences established that $P(4)$ is true. We can use the same technique to prove that $P(n)$ holds for an arbitrary integer $n \geq 0$, using the base case once and the inductive case n times.

The principle of mathematical induction is as simple as in Example 5.1—we apply the base case to get started, and then repeatedly apply the inductive case to conclude $P(n)$ for any larger n—but there are several analogies that can help to make proofs by mathematical induction more intuitive; see Figure 5.2.

Taking it further: "Mathematical induction" is somewhat unfortunately named because its name collides with a distinction made by philosophers between two types of reasoning. *Deductive* reasoning is the use of logic (particularly rules of inference) to reach conclusions—what computer scientists would call a *proof*. A proof by mathematical induction is an example of deductive reasoning. For a philosopher, though, *inductive reasoning* is the type of reasoning that draws conclusions from empirical observations. If you've seen a few hundred ravens in your life, and every one that you've seen is black, then you might conclude *All ravens are black*. Of course, it might turn out that your conclusion is false, because you haven't happened upon any of the albino ravens that exist in the world; what philosophers call inductive reasoning leads to conclusions that may turn out to be false.

A first example: summing powers of two

Let's use mathematical induction to prove an arithmetic property:

Theorem 5.2: A formula for the sum of powers of two.

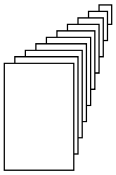
For any nonnegative integer n, we have that $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$.

As a plausibility check, let's test the given formula for some small values of n:

$$\begin{array}{lll}
n = 1: & 2^0 + 2^1 = 1 + 2 = 3 & 2^2 - 1 = 3 \\
n = 2: & 2^0 + 2^1 + 2^2 = 1 + 2 + 4 = 7 & 2^3 - 1 = 7 \\
n = 3: & 2^0 + 2^1 + 2^2 + 2^3 = 1 + 2 + 4 + 8 = 15 & 2^4 - 1 = 15
\end{array}$$

These small examples all check out, so it's reasonable to try to prove the claim. Let's do it, with our first example of a proof by induction:

Problem-solving tip: A plausibility check (testing out a claim for small values of n) can be very helpful before attempting a proof. Often small examples can either reveal a misunderstanding of the claim *or* help you see why the claim is true in general.



Dominoes falling: We have an infinitely long line of dominoes, numbered $0, 1, 2, \ldots, n, \ldots$. To convince someone that the nth domino falls over, you can convince them that

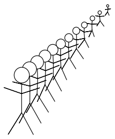
- the 0th domino falls over, and
- whenever one domino falls over, the next domino falls over too.

(One domino falls, and they keep on falling. Thus, for any $n \geq 0$, the nth domino falls.)

Climbing a ladder: We have a ladder with rungs numbered $0, 1, 2, \ldots, n, \ldots$. To convince someone that a climber climbing the ladder reaches the nth rung, you can convince them that

- the climber steps onto rung #0.
- if the climber steps onto one rung, then she also steps onto the next rung.

(The climber starts to climb, and the climber never stops climbing. Thus, for any $n \geq 0$, the climber reaches the nth rung.)



Whispering down the alley: We have an infinitely long line of people, with the people numbered $0, 1, 2, \ldots, n, \ldots$. To argue that everyone in the line learns a secret, we can argue that

- person #0 learns the secret.
- if person #n learns the secret, then she tells person #$(n+1)$ the secret.

(The person at the front of the line learns the secret, and everyone who learns it tells the secret to the next person in line. Thus, for any $n \geq 0$, the nth person learns the secret.)

Falling into the depths of despair: Consider the Pit of Infinite Despair, which is filled with nothing but despair and goes infinitely far down beneath the surface of the earth. (The Pit does not respect physics.) Suppose that:

- the Evil Villain is pushed into the pit (that is, She is in the Pit zero meters below the surface).
- if someone is in the Pit at a depth of n meters beneath the surface, then She falls to depth $n+1$ meters beneath the surface.

(The Villain starts to fall, and if the Villain has fallen to a certain depth then She falls another meter further. Thus, for any $n \geq 0$, the Evil Villain eventually reaches depth n in the Pit.)

Figure 5.2 Some analogies to make mathematical induction more intuitive.

① A clear statement of the claim to be proven—that is, a clear definition of the property $P(n)$ that will be proven true for all $n \geq 0$—and a statement that the proof is by induction, including specifically identifying the variable n upon which induction is being performed. (Some claims involve multiple variables, and it can be confusing if you aren't clear about which is the variable upon which you are performing induction.)

② A statement and proof of the base case—that is, a proof of $P(0)$.

③ A statement and proof of the inductive case—that is, a proof of $P(n-1) \Rightarrow P(n)$, for a generic value of $n \geq 1$. The proof of the inductive case should include all of the following:

ⓐ a statement of the inductive hypothesis $P(n-1)$.
ⓑ a statement of the claim $P(n)$ that needs to be proven.
ⓒ a proof of $P(n)$, which at some point makes use of the assumed inductive hypothesis.

Figure 5.3 A checklist of the steps required for a proof by mathematical induction.

Example 5.2: A proof of Theorem 5.2.

Let $P(n)$ denote the property

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

We'll prove that $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$ by induction on n.

Base case ($n = 0$). We must prove $P(0)$. That is, we must prove $\sum_{i=0}^{0} 2^i = 2^{0+1} - 1$. But this fact is easy to prove, because both sides are equal to 1: $\sum_{i=0}^{0} 2^i = 2^0 = 1$, and $2^{0+1} - 1 = 2 - 1 = 1$.

Inductive case ($n \geq 1$). We must prove that $P(n-1) \Rightarrow P(n)$, for an arbitrary integer $n \geq 1$. We prove this implication by assuming the antecedent—namely, we assume $P(n-1)$ and prove $P(n)$. The assumption $P(n-1)$ is

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

We can now prove $P(n)$—under the assumption $(*)$—by showing that the left-hand and right-hand sides of $P(n)$ are equal:

$$\begin{aligned}
\sum_{i=0}^{n} 2^i &= \left[\sum_{i=0}^{n-1} 2^i\right] + 2^n && \textit{by the definition of summations}\\
&= \left[2^{(n-1)+1} - 1\right] + 2^n && \textit{by } (*) \textit{, a.k.a. by the assumption that } P(n-1)\\
&= 2^n - 1 + 2^n && \textit{by algebraic manipulation}\\
&= 2 \cdot 2^n - 1\\
&= 2^{n+1} - 1.
\end{aligned}$$

We've thus shown that $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$—in other words, we've proven $P(n)$.

We've proven the base case $P(0)$ and the inductive case $P(n-1) \Rightarrow P(n)$, so by the principle of mathematical induction we have shown that $P(n)$ holds for all $n \in \mathbb{Z}^{\geq 0}$.

Taking it further: In case the inductive proof doesn't feel 100% natural, here's another way to make the result from Example 5.2 intuitive: think about binary representations of numbers. Written in binary, the number $\sum_{i=0}^{n} 2^i$ will look like $11 \cdots 111$, with $n+1$ ones. What happens when we add 1 to, say, 11111111 (= 255)? It's a colossal sequence of carrying (as $1 + 1 = 0$, carrying the 1 to the next place), resulting in 100000000 (= 256). In other words, $2^{n+1} - 1$ is written in binary as a sequence of $n+1$ ones—that is, $2^{n+1} - 1 = \sum_{i=0}^{n} 2^i$.

Example 5.2 follows the standard outline of a proof by mathematical induction. We will *always* prove the inductive case $P(n-1) \Rightarrow P(n)$ by assuming the antecedent $P(n-1)$ and proving $P(n)$. The assumed antecedent $P(n-1)$ in the inductive case of the proof is called the *inductive hypothesis*. (You may see "inductive hypothesis" abbreviated as *IH*.)

Warning! $P(n)$ denotes a *proposition*—that is, $P(n)$ is either true or false. (We're proving that, in fact, it's true for every n.) Despite its apparent temptation to people new to inductive proofs, it is nonsensical to treat $P(n)$ as a number.

A second example, and a template for proofs by induction

Here's another proof by induction, with the parts of the proof carefully labeled:

Example 5.3: Summing powers of -1.

Claim. For any integer $n \geq 0$, we have that $\sum_{i=0}^{n} (-1)^i = \begin{cases} 1 & \text{if } n \text{ is even} \\ 0 & \text{if } n \text{ is odd.} \end{cases}$

Proof. ① *Clearly state the claim to be proven. Clearly state that the proof will be by induction, and clearly state the variable upon which induction will be performed.*

Let $P(n)$ denote the property

$$\sum_{i=0}^{n} (-1)^i = \begin{cases} 1 & \text{if } n \text{ is even} \\ 0 & \text{if } n \text{ is odd.} \end{cases}$$

We'll prove that $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$ by induction on n.

② *State and prove the base case.*

Base case ($n = 0$). We must prove $P(0)$. But $\sum_{i=0}^{0} (-1)^i = (-1)^0 = 1$, and 0 is even.

③ *State and prove the inductive case. Within the statement and proof of the inductive case . . .*

ⓐ *. . . state the inductive hypothesis.*

Inductive case ($n \geq 1$). We assume the inductive hypothesis $P(n-1)$, namely

$$\sum_{i=0}^{n-1} (-1)^i = \begin{cases} 1 & \text{if } n-1 \text{ is even} \\ 0 & \text{if } n-1 \text{ is odd.} \end{cases}$$

ⓑ *. . . state what we need to prove.*

We must prove $P(n)$.

ⓒ *. . . prove it, making use of the inductive hypothesis and stating where it was used.*

$$\begin{aligned}
\sum_{i=0}^{n} (-1)^i &= \left[\sum_{i=0}^{n-1} (-1)^i \right] + (-1)^n && \textit{definition of summations} \\
&= \begin{cases} 1 + (-1)^n & \text{if } n-1 \text{ is even} \\ 0 + (-1)^n & \text{if } n-1 \text{ is odd.} \end{cases} && \textit{inductive hypothesis} \\
&= \begin{cases} 1 + (-1)^n & \text{if } n \text{ is odd} \\ 0 + (-1)^n & \text{if } n \text{ is even.} \end{cases} && \textit{n is odd} \Leftrightarrow \textit{n} - 1 \textit{ is even}
\end{aligned}$$

$$= \begin{cases} 1 + -1 & \text{if } n \text{ is odd} \\ 0 + 1 & \text{if } n \text{ is even.} \end{cases}$$ *$(-1)^n = \pm 1$, depending on whether n is even; see Exercise 5.3.*

$$= \begin{cases} 0 & \text{if } n \text{ is odd} \\ 1 & \text{if } n \text{ is even.} \end{cases}$$

Thus we have proven $P(n)$, and the theorem follows. □

Writing tip: In the inductive case of a proof of an equality—like Example 5.3—start from the left-hand side of the equality and manipulate it until you derive the right-hand side of the equality *exactly*. If you work from both sides simultaneously, you're at risk of the fallacy of proving true—or at least the appearance of that fallacy!

We can treat the labeled pieces of Example 5.3 as a checklist for writing proofs by induction. (See Figure 5.3.) You should ensure that when you write an inductive proof, you include each of these steps.

The sum of the first n integers

We'll do another example of an inductive proof of an arithmetic property:

Theorem 5.3: Sum of the first n nonnegative integers.

For any integer $n \geq 0$, we have that $\sum_{i=0}^{n} i = 0 + 1 + \cdots + n$ is equal to $\frac{n(n+1)}{2}$.

(For example, for $n = 4$ we have $0 + 1 + 2 + 3 + 4 = 10 = \frac{4(4+1)}{2}$.)

Example 5.4: Sum of the first n integers: proving Theorem 5.3.

First, we must phrase this problem in terms of a property $P(n)$ that we'll prove true for every $n \geq 0$. For a particular integer n, let $P(n)$ denote the claim that

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}.$$

We will prove that $P(n)$ holds for all integers $n \geq 0$ by induction on n.

Base case ($n = 0$). Note that $\sum_{i=1}^{0} i = 0$ and $\frac{0(0+1)}{2} = 0$ too. Thus $P(0)$ follows.

Inductive case ($n \geq 1$). Assume the inductive hypothesis $P(n-1)$, namely

$$\sum_{i=0}^{n-1} i = \frac{(n-1)((n-1)+1)}{2}.$$

We must prove $P(n)$—that is, we must prove $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$. Here is the proof:

$$\sum_{i=0}^{n} i = \left[\sum_{i=0}^{n-1} i \right] + n$$ *definition of summations*

$$= \frac{(n-1)((n-1)+1)}{2} + n$$ *inductive hypothesis*

$$
\begin{aligned}
&= \frac{(n-1)n + 2n}{2} && \textit{putting terms over common denominator} \\
&= \frac{n(n-1+2)}{2} && \textit{factoring} \\
&= \frac{n(n+1)}{2}.
\end{aligned}
$$

Thus we've shown $P(n)$ assuming $P(n-1)$, which completes the proof.

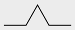
Taking it further: While the summation that we analyzed in Theorem 5.3 may seem like a purely arithmetic example, it also has direct applications in CS—particularly in the *analysis of algorithms.* Chapter 6 is devoted to this topic, and there's much more there, but here's a brief preview. A basic step in analyzing an algorithm is counting how many steps that algorithm takes, for an input of arbitrary size. One particular example is Insertion Sort, which sorts an n-element array by repeatedly ensuring that the first k elements of the array are in sorted order (by swapping the kth element backward until it's in position). The total number of swaps that are done in the kth iteration can be as high as $k-1$—so the total number of swaps can be as high as $\sum_{k=1}^{n} k-1 = \sum_{i=0}^{n-1} i$. Thus Theorem 5.3 tells us that Insertion Sort can require as many as $n(n-1)/2$ swaps.

Generating a conjecture: segments in a fractal

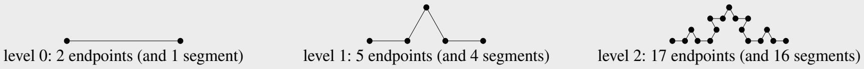
In the inductive proofs that we've seen thus far, we were given a problem statement that described exactly what property we needed to prove. Solving these problems "just" requires proving the base case and the inductive case—which may or may not be *easy,* but at least we know what we're trying to prove! In other problems, though, you may also have to first figure out what you're going to prove, and *then* prove it. Obviously this task is generally harder. Here's one example of such a proof, about the Von Koch snowflake fractal from Figure 5.1:

Problem-solving tip: Your first task in giving a proof by induction is to identify the property $P(n)$ that you'll prove true for every integer $n \geq 0$. Sometimes the property is given to you more or less directly and sometimes you'll have to formulate it yourself, but in any case you need to identify the precise property you're going to prove before you can prove it!

Example 5.5: Vertices in a Von Koch Line.

A Von Koch line of level 0 is a straight line segment; a Von Koch line of level $\ell \geq 1$ consists of four Von Koch lines of level $(\ell - 1)$ in the shape . (See Figure 5.4.) Conjecture a formula for the number of *vertices* (that is, the number of segment endpoints) in a Von Koch line of level ℓ. Prove the correctness of your formula by induction.

Solution. Our first task is to formulate a conjecture for the number of vertices in a Von Koch line of level ℓ. Let's start with a few small examples, and some tedious counting in the pictures in Figure 5.4:



level 0: 2 endpoints (and 1 segment) level 1: 5 endpoints (and 4 segments) level 2: 17 endpoints (and 16 segments)

There are a few ways to think about this pattern. Here's one that turns out to be helpful: a level-ℓ line contains 4 lines of level $(\ell - 1)$, so it contains 16 lines of level $(\ell - 2)$. And thus, expanding it all the way out, the level-ℓ line contains 4^{ℓ} lines of level 0. The number of endpoints that we observe is $2 = 4^0 + 1$,

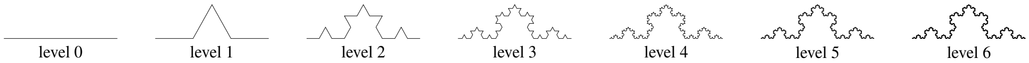


Figure 5.4 Von Koch lines of several levels. (A Von Koch snowflake consists of three Von Koch lines, all of the same level, arranged in a triangle; see Figure 5.1.)

then $5 = 4^1 + 1$, then $17 = 4^2 + 1$. (Why the "+1?" Each segment starts where the previous segment ended—so there is one more endpoint than segment, because of the last segment's second endpoint.)

So it looks like there are $4^\ell + 1$ endpoints in a Von Koch line of level ℓ. Let's turn this observation into a formal claim, with an inductive proof:

Claim: For any $\ell \geq 0$, a Von Koch line of level ℓ has $4^\ell + 1$ endpoints.

Proof. Let $P(\ell)$ denote the claim that a Von Koch line of level ℓ has $4^\ell + 1$ endpoints. We'll prove that $P(\ell)$ holds for all integers $\ell \geq 0$ by induction on ℓ.

Base case ($\ell = 0$). We must prove $P(0)$. By definition, a Von Koch line of level 0 is a single line segment, which has 2 endpoints. Indeed, $4^0 + 1 = 1 + 1 = 2$.

Inductive case ($\ell \geq 1$). We assume the inductive hypothesis, namely $P(\ell - 1)$, and we must prove $P(\ell)$. The key observation is that a Von Koch line of level ℓ consists of four Von Koch lines of level $(\ell - 1)$—and the last endpoint of line #1 is identical to the first endpoint of line #2; the last endpoint of #2 is the first of #3, and the last endpoint of #3 is the first of #4. Therefore there are three endpoints that are shared among the four lines of level $(\ell - 1)$. Thus:

$$\begin{aligned}
&\text{the number of endpoints in a Von Koch line of level } \ell \\
&= 4 \cdot \big[\text{the number of endpoints in a Von Koch line of level } (\ell - 1)\big] - 3 && \textit{by the above discussion} \\
&= 4 \cdot \left[4^{\ell-1} + 1\right] - 3 && \textit{by the inductive hypothesis} \\
&= 4^\ell + 4 - 3 && \textit{multiplying through} \\
&= 4^\ell + 1. && \textit{algebra}
\end{aligned}$$

Thus $P(\ell)$ follows, completing the proof. □

A note and two variations on the inductive template

The basic idea of induction is not too hard: the reason that $P(n)$ holds is that $P(n - 1)$ held, and the reason that $P(n - 1)$ held is that $P(n - 2)$ held—and so forth, until eventually the proof finally rests on $P(0)$, the base case. A proof by induction can sometimes look superficially like it's circular reasoning—that we're assuming precisely the thing that we're trying to prove. *But it's not!* In the inductive case, we're assuming $P(n - 1)$ and proving $P(n)$—we are *not* assuming $P(n)$ and proving $P(n)$.

Warning! If you do not use the inductive hypothesis $P(n-1)$ in the proof of $P(n)$, then something is wrong—or, at least, your proof is not actually a proof by induction!

Taking it further: The superficial appearance of circularity in a proof by induction is equivalent to the superficial appearance that a recursive function in a program will run forever. (A recursive function f *will* run forever if calling f on n results in f calling itself on n again! That's the same circularity that would happen if we assumed $P(n)$ and proved $P(n)$.) The correspondence between these aspects of induction and recursion should be no surprise; induction and recursion are essentially the same thing. In fact, it's not too hard to write a recursive function that "implements" an inductive proof by outputting a step-by-step argument establishing $P(n)$ for an arbitrary n, as in Example 5.1.

Our proofs so far have shown $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$ by proving $P(0)$ as a base case. If we instead want to prove $\forall n \in \mathbb{Z}^{\geq k} : P(n)$ for some integer k, we can prove $P(k)$ as the base case, and then prove the inductive case $P(n-1) \Rightarrow P(n)$ for all $n \geq k+1$.

Another variation in writing inductive proofs relates to the statement of the inductive case. We've proven $P(0)$ and $P(n-1) \Rightarrow P(n)$ for arbitrary $n \geq 1$. Some writers prefer to prove $P(0)$ and $P(n) \Rightarrow P(n+1)$ for arbitrary $n \geq 0$. The difference is merely a reindexing, not a substantive difference: it's just a matter of whether one thinks of induction as "the nth domino falls because the $(n-1)$st domino fell into it" or as "the nth domino falls and therefore knocks over the $(n+1)$st domino."

5.2.2 Some Numerical Examples: Geometric, Arithmetic, and Harmonic Series

In the remainder of this section, we'll give some more examples of proofs by mathematical induction, following the examples so far, and checklist of Figure 5.3. We'll start here with some more summation-based proofs. (The statements we'll prove in this section are intended to serve both as useful facts to know about particular numerical sequences and as good practice with induction.) We'll move on to inductive proofs of some other types of statements—inductive proofs are ubiquitous throughout computer science, not just in analyzing summations—in Section 5.2.3.

Geometric series

A *geometric sequence* of numbers is one in the ratio between consecutive numbers is always the same: for example, each pair of consecutive numbers in the sequence $1, 2, 4, 8, 16, \ldots$ differs by a factor of two.

Definition 5.4: Geometric sequences and series.
A *geometric sequence* is a sequence of numbers where each entry is generated by multiplying the previous entry by a fixed ratio $r \in \mathbb{R}$, starting from an initial value a. (Thus the sequence is $\langle a, ar, ar^2, ar^3, \ldots \rangle$.) A *geometric series* or *geometric sum* is $\sum_{i=0}^{n} ar^i$.

Examples of geometric sequences include $\langle 2, 4, 8, 16, 32, \ldots \rangle$; or $\langle 1, \frac{1}{3}, \frac{1}{9}, \frac{1}{27}, \ldots \rangle$; or $\langle 1, 1, 1, 1, 1, \ldots \rangle$. There turns out to be a relatively simple formula for the sum of the first n terms of a geometric sequence:

Theorem 5.5: Analysis of geometric series.

Let $r \in \mathbb{R}$ where $r \neq 1$, and let $n \in \mathbb{Z}^{\geq 0}$. Then $\sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r-1}$. (If $r = 1$, then $\sum_{i=0}^{n} r^i = n+1$.)

(For simplicity, we stated Theorem 5.5 without reference to the initial value a. Because we can pull a constant multiplicative factor out of a summation, the theorem allows us to conclude that $\sum_{i=0}^{n} ar^i = a \cdot \sum_{i=0}^{n} r^i = a \cdot \frac{r^{n+1}-1}{r-1}$.) We will be able to prove Theorem 5.5 using a proof by mathematical induction:

Example 5.6: Geometric series.

Proof of Theorem 5.5. Consider a fixed real number r with $r \neq 1$, and let $P(n)$ denote the property that

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r-1}.$$

We'll prove that $P(n)$ holds for all integers $n \geq 0$ by induction on n.

Base case ($n = 0$). Note that $\sum_{i=0}^{0} r^i = r^0$ and $\frac{r^{0+1}-1}{r-1}$ both equal 1. Thus $P(0)$ holds.

Inductive case ($n \geq 1$). We assume the inductive hypothesis $P(n-1)$, which tells us that $\sum_{i=0}^{n-1} r^i = \frac{r^n-1}{r-1}$. We must prove $P(n)$. Here is a proof:

$$\begin{aligned}
\sum_{i=0}^{n} r^i &= r^n + \sum_{i=0}^{n-1} r^i && \textit{definition of summation}\\
&= r^n + \tfrac{r^n-1}{r-1} && \textit{inductive hypothesis}\\
&= \tfrac{r^n(r-1)+r^n-1}{r-1} && \textit{putting the fractions over a common denominator}\\
&= \tfrac{r^{n+1}-r^n+r^n-1}{r-1} && \textit{multiplying out}\\
&= \tfrac{r^{n+1}-1}{r-1}. && \textit{simplifying}
\end{aligned}$$

Thus $P(n)$ holds, and the theorem follows. □

Problem-solving tip: The inductive cases of many inductive proofs follow the same pattern: first, we use some kind of structural definition to "pull apart" the statement about n into something kind of statement about $n-1$ (plus some "leftover" other stuff), then apply the inductive hypothesis to simplify the $n-1$ part. We then manipulate the result of using the inductive hypothesis plus the leftovers to get the desired equation.

Notice that Examples 5.2 and 5.3 were both special cases of Theorem 5.5. For the former, Theorem 5.5 tells us that $\sum_{i=0}^{n} 2^i = \frac{2^{n+1}-1}{2-1} = 2^{n+1} - 1$; for the latter, this theorem tells us that

$$\sum_{i=0}^{n} (-1)^i = \frac{(-1)^{n+1}-1}{-1-1} = \frac{1-(-1)^{n+1}}{2} = \begin{cases} \frac{1-(-1)}{2} = 1 & \text{if } n \text{ is even} \\ \frac{1-1}{2} = 0 & \text{if } n \text{ is odd.} \end{cases}$$

A corollary of Theorem 5.5 addressing *infinite* geometric sums will turn out to be useful later:

Corollary 5.6. Let $r \in \mathbb{R}$ where $0 \leq r < 1$, and define $f(n) = \sum_{i=0}^{n} r^i$. Then:

1 $\sum_{i=0}^{\infty} r^i = \frac{1}{1-r}$, and

2 For all $n \geq 0$, we have $1 \leq f(n) \leq \frac{1}{1-r}$.

Taking it further: The proof of Corollary 5.6 relies on calculus, so we won't dwell on it here. But, if you're interested, here's the idea. We'll start with the proof of (1), which is the part that requires calculus. Theorem 5.5 says that $f(n) = \frac{r^{n+1}-1}{r-1}$, and we take the limit as $n \to \infty$. Because $r < 1$, we have that $\lim_{n\to\infty} r^{n+1} = 0$. Thus as $n \to \infty$ the numerator $r^{n+1} - 1$ tends to -1, and the entire ratio tends to $1/(1-r)$. To prove (2), observe that $\sum_{i=0}^{n} r^i$ is definitely greater than or equal to $\sum_{i=0}^{0} r^i$ (because $r \geq 0$ and so the latter results by eliminating n nonnegative terms from the former). Similarly, $\sum_{i=0}^{n} r^i$ is definitely less than or equal to $\sum_{i=0}^{\infty} r^i$. Thus we have $f(n) = \sum_{i=0}^{n} r^i \geq \sum_{i=0}^{0} r^i = r^0 = 1$ and, similarly, we have $f(n) = \sum_{i=0}^{n} r^i \leq \sum_{i=0}^{\infty} r^i = \frac{1}{1-r}$.

Arithmetic series

An *arithmetic sequence* of numbers in which the *difference* between consecutive numbers is always the same: for example, consecutive numbers in the sequence $2, 4, 6, 8, 10, \ldots)$ always differ by two. (The definition for a geometric sequence was about *ratios* instead of differences.)

Definition 5.7: Arithmetic sequences and series.
An *arithmetic sequence* is a sequence of numbers where each number is generated by adding a fixed step-size $r \in \mathbb{R}$ to the previous number in the sequence. The first entry in the sequence is some initial value $a \in \mathbb{R}$. (Thus the sequence is $\langle a, a+d, a+2d, a+3d, \ldots \rangle$.) An *arithmetic series* or *sum* is $\sum_{i=0}^{n}(a+id)$.

Examples include $\langle 2, 4, 6, 8, 10, \ldots \rangle$; or $\langle 1, \frac{1}{3}, -\frac{1}{3}, -1, -\frac{5}{3}, \ldots \rangle$; or $\langle 1, 1, 1, 1, 1, \ldots \rangle$. You'll prove a general formula for an arithmetic sum in the exercises.

Harmonic series

The *harmonic* sequence is defined in terms of the reciprocals of the positive integers $(1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \ldots)$:

Definition 5.8: Harmonic series.
A *harmonic series* is the sum of a sequence of numbers whose kth number is $\frac{1}{k}$. The *nth harmonic number* is defined by $H_n = \sum_{k=1}^{n} \frac{1}{k}$.

Thus $H_1 = 1$ and $H_2 = 1 + \frac{1}{2} = 1.5$ and $H_3 = 1 + \frac{1}{2} + \frac{1}{3} \approx 1.8333$ and $H_4 = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} \approx 2.0833$.

The name "harmonic" comes from music: when a note at frequency f is played, *overtones* of that note—other high-intensity frequencies—can be heard at frequencies $2f, 3f, 4f, \ldots$. The *wavelengths* of the corresponding sound waves are $\frac{1}{f}, \frac{1}{2f}, \frac{1}{3f}, \frac{1}{4f}, \ldots$. (If you've had calculus: you can approximate the value of H_n using an integral, writing $H_n = \sum_{x=1}^{n} \frac{1}{x} \approx \int_{x=1}^{n} \frac{1}{x}\, dx = \ln n$. But we'll do a calculus-free analysis here.)

Giving a precise equation for the value of H_n requires more work than for geometric or arithmetic series, but we can prove upper and lower bounds on H_n by induction a bit more easily. We will be able to show the following, which captures the value of H_n to within a factor of 2:

Theorem 5.9: Bounds on the (2^k)th harmonic number.
For any integer $k \geq 0$, we have $k + 1 \geq H_{2^k} \geq \frac{k}{2} + 1$.

(See Figure 5.5.) We'll prove half of Theorem 5.9 (namely $k + 1 \geq H_{2^k}$) by induction in Example 5.7, leaving the other half to the exercises. (Theorem 5.9 only talks about the case in which n is a power of 2; we'll also leave to the exercises upper and lower bounds for H_n when n is not an exact power of 2.)

Example 5.7: Inductive proof that $k + 1 \geq H_{2^k}$.
Proof. Let $P(k)$ denote the property that $k+1 \geq H_{2^k}$. We'll use induction on k to prove that $P(k)$ holds for all integers $k \geq 0$.

Base case ($k = 0$). We have that $H_{2^k} = H_{2^0} = H_1 = 1$, and $k + 1 = 0 + 1 = 1$ as well. Therefore $H_{2^k} = 1 = k + 1$—that is, $P(0)$ holds.

Inductive case ($k \geq 1$). Let $k \geq 1$ be an integer. We must prove $P(k)$—that is, we must prove $k+1 \geq H_{2^k}$. To do so, we assume the inductive hypothesis $P(k-1)$, namely that $k \geq H_{2^{k-1}}$. Consider H_{2^k}:

$$
\begin{aligned}
H_{2^k} &= \textstyle\sum_{i=1}^{2^k} \frac{1}{i} && \textit{definition of the harmonic numbers} \\
&= \left[\textstyle\sum_{i=1}^{2^{k-1}} \frac{1}{i}\right] + \left[\textstyle\sum_{i=2^{k-1}+1}^{2^k} \frac{1}{i}\right] && \textit{splitting the summation into two parts: } i \leq 2^{k-1} \textit{ and } i > 2^{k-1} \\
&= H_{2^{k-1}} + \left[\textstyle\sum_{i=2^{k-1}+1}^{2^k} \frac{1}{i}\right] && \textit{definition of the harmonic numbers, again} \\
&\leq H_{2^{k-1}} + \left[\textstyle\sum_{i=2^{k-1}+1}^{2^k} \frac{1}{2^{k-1}}\right] && \textit{every term in the summation } \textstyle\sum_{i=2^{k-1}+1}^{2^k} \frac{1}{i} \textit{ is smaller than } \frac{1}{2^{k-1}} \\
&\leq H_{2^{k-1}} + 2^{k-1} \cdot \frac{1}{2^{k-1}} && \textit{there are } 2^{k-1} \textit{ terms in the summation} \\
&= H_{2^{k-1}} + 1 && \tfrac{1}{x} \cdot x = 1 \textit{ for any } x \neq 0 \\
&\leq k + 1. && \textit{inductive hypothesis}
\end{aligned}
$$

Thus we've proven that $H_{2^k} \leq k + 1$—that is, we've proven $P(k)$. The theorem follows. □

The proof in Example 5.7 is perhaps the first time in which we needed some serious insight and creativity to establish the inductive case. (I don't think that the "split the summation" step was in any way an obvious thing to do, nor was the step of bounding $\frac{1}{i} \leq \frac{1}{2^{k-1}}$ for the second-half summation.) The structure of a proof

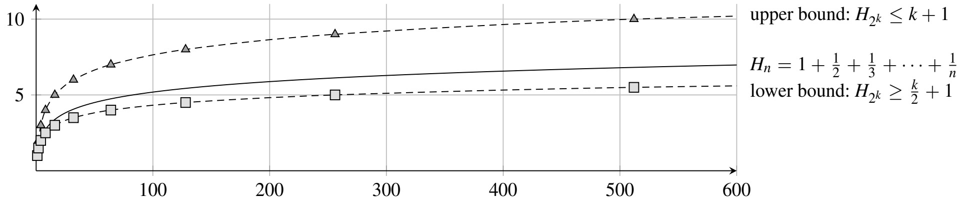


Figure 5.5 A visualization of Theorem 5.9, showing that $k + 1 \geq H(2^k) \geq \frac{k}{2} + 1$.

by induction is rigid—we must prove a base case $P(0)$; we must prove an inductive case $P(n-1) \Rightarrow P(n)$—but that doesn't make the entire proof totally formulaic. (The proof of the inductive case must use the inductive hypothesis at some point, so its statement gives you a little guidance for the kinds of manipulations to try.) Just as with all the other proof techniques that we explored in Chapter 4, a proof by induction can require you to *think*—and all of strategies that we discussed in Chapter 4 may be helpful to deploy.

Taking it further: Induction as a proof technique is the analogue to recursion as a programming technique, in the sense that both of them "call" themselves on smaller inputs. But they are also related in the sense that we just discussed, regarding the proof in Example 5.7: there can be some surprising, creative ways of using both to solve seemingly difficult problems. Here's another example: when you want to communicate with someone secretly, the computationally inspired thing to do is to use cryptography to encrypt your messages before you send them. (Perhaps you'd use RSA; see Section 7.5.) But if you want to keep secret not just the *contents* of your messages, but also *the very fact that you're communicating* with the other person, then there is a clever recursive use of encryption that can be deployed. See p. 5-22 for a summary of *onion routing* (and the *Tor* system), along with a brief discussion of steganography, which has a related motivation.

5.2.3 Some More Examples

We'll close this section with a few more examples of proofs by mathematical induction, but we'll focus on things other than analyzing summations. Some of these examples are still about arithmetic properties, but they should at least hint at the breadth of possible statements that we might be able to prove by induction.

Comparing algorithms: which is faster?

Suppose that we have two different candidate algorithms that solve a problem related to a set S with n elements—a *brute-force algorithm* that tries all 2^n possible subsets of S, and a second algorithm that computes the solution by looking at only n^2 subsets of S. Which would be faster to use? The latter algorithm is (massively!) faster, and we can prove that it's faster (with a small caveat for small n) by induction:

n	2^n	n^2
0	1	0
1	2	1
2	4	4
3	8	9
4	16	16
5	32	25
6	64	36
7	128	49

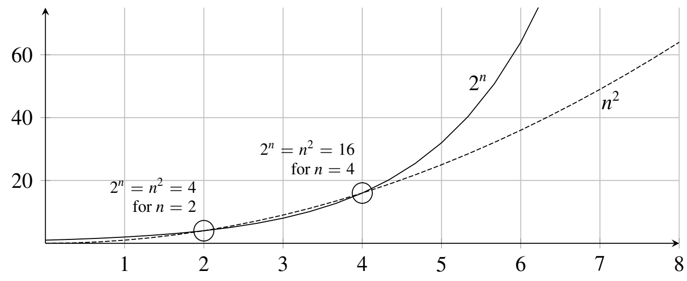


Figure 5.6 Small values of 2^n and n^2, and a plot of the functions.

Example 5.8: 2^n vs. n^2.

We'd like to prove that $2^n \geq n^2$ for all integers $n \geq 0$—but it turns out not to be true! (See Figure 5.6.) Indeed, $2^3 < 3^2$. But the relationship appears to hold starting at $n = 4$. Let's prove it, by induction:

Claim: For all integers $n \geq 4$, we have $2^n \geq n^2$.

Proof. Let $P(n)$ denote the property $2^n \geq n^2$. We'll use induction on n to prove that $P(n)$ holds for all $n \geq 4$. (Thus our base case will be for $n = 4$, and our inductive case for $n \geq 5$.)

Base case ($n = 4$). For $n = 4$, we have $2^n = 16 = n^2$, so the inequality $P(4)$ holds.

Inductive case ($n \geq 5$). Assume the inductive hypothesis $P(n-1)$—that is, assume $2^{n-1} \geq (n-1)^2$. We must prove $P(n)$. For $n \geq 4$, note that $n^2 \geq 4n$ (by multiplying both sides of the inequality $n \geq 4$ by n). Thus $n^2 - 4n \geq 0$, and so

$$\begin{aligned}
2^n &= 2 \cdot (2^{n-1}) && \textit{definition of exponentiation} \\
&\geq 2 \cdot (n-1)^2 && \textit{inductive hypothesis} \\
&= 2n^2 - 4n + 2 && \textit{multiplying out} \\
&= n^2 + (n^2 - 4n) + 2 && \textit{rearranging} \\
&\geq n^2 + 0 + 2 && \textit{by the above discussion, we have } n^2 - 4n \geq 0 \\
&> n^2.
\end{aligned}$$

Thus we have shown $2^n > n^2$, which completes the proof of the inductive case. The claim follows. □

Taking it further: In analyzing the efficiency of algorithms, we will frequently have to do the type of comparison in Example 5.8, to compare the amount of time consumed by one algorithm versus another. Chapter 6 discusses this type of comparison in much greater detail, but here's one example of this sort. Let X be a sequence. A *subsequence* of X results from selecting some of the entries in X—for example, `TURING` is a subsequence of `OUTSOURCING`. For two sequences X and Y, a *common subsequence* is a subsequence of both X and Y. The *longest common subsequence* of X and Y is, naturally, the common subsequence of X and Y that's longest. (For example, `TURING` is the longest common subsequence of `DISTURBINGLY` and `OUTSOURCING`.)

Given two sequences X and Y of length n, we can find the longest common subsequence fairly easily by testing *every possible subsequence of X* to see whether it's also a subsequence of Y. This brute-force solution tests 2^n subsequences of X. But there's a cleverer approach to solving this problem using an algorithmic design technique called *dynamic programming* (see p. 9-73 or a textbook on algorithms) that avoids redoing the same computation—here, testing the same sequence of letters to see if it appears in Y—more than once. The dynamic programming algorithm for longest common subsequence requires only about n^2 steps.

Proving algorithms correct: factorial

We just gave an example of using a proof by induction to analyze the efficiency of an algorithm, but we can also use mathematical induction to prove the *correctness* of a recursive algorithm. (That is, we'd like to show that a recursive algorithm always returns the desired output.) Here's an example, for the natural recursive algorithm to compute factorials (see Figure 5.7):

Taking it further: While induction is much more closely related to recursive algorithms than nonrecursive algorithms, we can also prove the correctness of an iterative algorithm using induction. The basic idea is to consider a statement, called a *loop invariant*, about the correct behavior of a loop; we can prove inductively that a loop invariant starts out true and stays true throughout the execution of the algorithm. See p. 5-20.

Example 5.9: Factorial.

Consider the recursive algorithm **fact** in Figure 5.7. For a positive integer n, let $P(n)$ denote the property that $\textbf{fact}(n) = n!$. We'll prove by induction on n that, indeed, $P(n)$ holds for all integers $n \geq 1$.

Base case $(n = 1)$. $P(1)$ is true because $\textbf{fact}(1)$ returns 1 immediately, and $1! = 1$ by definition.

Inductive case $(n \geq 2)$. We assume the inductive hypothesis $P(n-1)$, namely that $\textbf{fact}(n-1)$ returns $(n-1)!$. We want to prove that $\textbf{fact}(n)$ returns $n!$. But this claim follows directly:

$$\begin{aligned}
\textbf{fact}(n) &= n \cdot \textbf{fact}(n-1) && \textit{by inspection of the algorithm} \\
&= n \cdot (n-1)! && \textit{by the inductive hypothesis} \\
&= n! && \textit{by definition of } !
\end{aligned}$$

and therefore the claim holds by induction.

In fact, induction and recursion are basically the same thing: recursion "works" by leveraging a solution to a smaller instance of a problem to solve a larger instance of the same problem; a proof by induction "works" by leveraging a proof of a smaller instance of a claim to prove a larger instance of the same claim. (Actually, one common use of induction is to analyze the efficiency of a recursive algorithm. We'll discuss this type of analysis in great depth in Section 6.4.)

Divisibility

We'll close this section with one more numerical example, about divisibility:

Writing tip: Example 5.10 illustrates why it is crucial to state clearly the variable upon which induction is being performed. This statement involves two variables, k and n, but we're performing induction on only one of them!

```
fact($n$):
if $n = 1$ then
    return 1
else
    return $n \cdot$ fact($n - 1$)
```

Figure 5.7 Pseudocode for factorial: given $n \in \mathbb{Z}^{\geq 1}$, we wish to compute the value of $n!$.

Example 5.10: $k^n - 1$ is evenly divisible by $k - 1$.

Claim: For any $n \geq 0$ and $k \geq 2$, we have that $k^n - 1$ is evenly divisible by $k - 1$.

(For example, $7^n - 1$ is always divisible by 6, as in $7 - 1$, $49 - 1$, and $343 - 1$. And $k^2 - 1$ is always divisible by $k - 1$; in fact, factoring $k^2 - 1$ yields $k^2 - 1 = (k - 1)(k + 1)$.)

Proof. We'll proceed by induction on n. That is, let $P(n)$ denote the claim

For all integers $k \geq 2$, we have that $k^n - 1$ is evenly divisible by $k - 1$.

We will prove that $P(n)$ holds for all integers $n \geq 0$ by induction on n.

Base case ($n = 0$). For any k, we have $k^n - 1 = k^0 - 1 = 1 - 1 = 0$. And 0 is evenly divisible by any positive integer, including $k - 1$. Thus $P(0)$ holds.

Inductive case ($n \geq 1$). We assume the inductive hypothesis $P(n - 1)$, and we need to prove $P(n)$. Let $k \geq 2$ be an arbitrary integer. Then:

$$\begin{aligned} k^n - 1 &= k^n - k + k - 1 && \textit{antisimplification: } x = x + k - k. \\ &= k \cdot (k^{n-1} - 1) + k - 1 && \textit{factoring} \end{aligned}$$

By the inductive hypothesis, $k^{n-1} - 1$ is evenly divisible by $k - 1$. In other words, by the definition of divisibility, there exists a nonnegative integer a such that $a \cdot (k - 1) = k^{n-1} - 1$. Therefore

$$\begin{aligned} k^n - 1 &= k \cdot a \cdot (k - 1) + k - 1 \\ &= (k - 1) \cdot (k \cdot a + 1). \end{aligned}$$

Because $k \cdot a + 1$ is a nonnegative integer, $(k - 1) \cdot (k \cdot a + 1)$ is by definition evenly divisible by $k - 1$. Thus $k^n - 1 = (k - 1) \cdot (k \cdot a + 1)$ is evenly divisible by $k - 1$. Our k was arbitrary, so $P(n)$ follows. □

Problem-solving tip: In inductive proofs, try to massage the expression in question into something—*anything!*—that matches the form of the inductive hypothesis. Here, the "antisimplification" step is obviously true but seems completely bizarre. Why did we do it? Our only hope in the inductive case is to somehow make use of the inductive hypothesis. Here, the inductive hypothesis tells us something about $k^{n-1} - 1$—so a good strategy is to transform $k^n - 1$ into an expression involving $k^{n-1} - 1$, plus some leftover stuff.

Computer Science Connections

Loop Invariants

In Example 5.9, we saw how to use a proof by induction to establish that a recursive algorithm correctly solves a particular problem. But proving the correctness of *iterative* algorithms seems different. An approach—pioneered in the 1960s by Robert Floyd and C. A. R. Hoare [48, 55]—is based on *loop invariants*, and can be used to analyze nonrecursive algorithms. A *loop invariant* for a loop L is logical property P such that (i) P is true before L is first executed; and (ii) if P is true at the beginning of an iteration of L, then P is true after that iteration of L. The parallels to induction are pretty direct; property (i) is the base case, and property (ii) is the inductive case. Together, they ensure that P is always true, and in particular P is true when the loop terminates.

Let's look at Insertion Sort as an example of using loop invariants. (See Figure 5.8.) The basic idea will be to define a property that describes the partial (and growing) state of correctness of the algorithm as it executes. Specifically, let's define $P(k)$ to denote the condition that *the subarray* $A[1 \ldots k+1]$ *is sorted after completing k iterations of the outer* **while** *loop*. We claim that $P(k)$ is true for all $k \geq 0$—in other words, that P is a loop invariant for the outer **while** loop. Let's prove it:

```
insertionSort(A[1...n]):
1 i := 2
2 while i ≤ n:
3     j := i
4     while j > 1 and A[j] > A[j − 1]:
5         swap A[j] and A[j − 1]
6         j := j − 1
7     i := i + 1
```

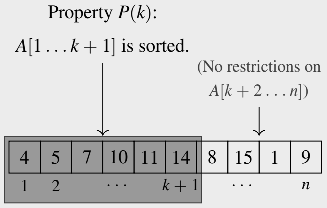

Property $P(k)$:
$A[1 \ldots k+1]$ is sorted.
(No restrictions on $A[k+2 \ldots n]$)
4
5
7
10
11
14
8
15
1
9
1
2
· · ·
$k+1$
· · ·
n

Figure 5.8 Insertion Sort, and the loop invariant (if we've completed k iterations of the outer loop, then $P(k)$ holds—here illustrated for $k = 5$).

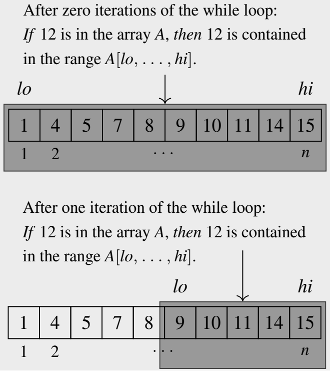
Proof (sketch). For the base case ($k = 0$), we've completed zero iterations—that is, we have only executed Line 1. But $A[1 \ldots k+1]$ is then vacuously sorted, because it contains only the lone element $A[1]$.

For the inductive case ($k \geq 1$), assume the inductive hypothesis $P(k-1)$—that is, that $A[1 \ldots k]$ was sorted before the kth iteration. The kth iteration of the loop executed Lines 2–7, so we must persuade ourselves that the execution of these lines extends the sorted segment $A[1 \ldots k]$ to $A[1 \ldots k+1]$. And it does! (A formal proof of this claim would use *another* loop invariant to describe the behavior of the inner loop—for example, letting $Q(j)$ denote the property *both* $A[1 \ldots j-1]$ *and* $A[j \ldots i]$ *are sorted, and* $A[j-1] < A[j+1]$. But we'll leave out those details in this sketch.) □

Because the above argument establishes that $P(n-1)$ is true, we know that $A[1 \ldots (n-1)+1]$ (in other words, all of A) is sorted after $n-1$ iterations of the loop, exactly as desired.

Loop invariants are valuable as part of the development of programs, too. For example, many people end up struggling to correctly write binary search (it's all too easy to have an infinite loop or an early-stopping bug)—but by writing down a loop invariant and thinking about it as we write the

```
binarySearch(A[1...n], x):
Output: Is x in the sorted array A?
lo := 1
hi := n
while lo ≤ hi:
    middle := ⌊(lo+hi)/2⌋
    if A[middle] = x then
        return True
    else if A[middle] > x then
        hi := middle − 1
    else
        lo := middle + 1
return False
```


After zero iterations of the while loop:
If 12 is in the array A, *then* 12 is contained in the range $A[lo, \ldots, hi]$.
lo
hi
1
4
5
7
8
9
10
11
14
15
1
2
· · ·
n
After one iteration of the while loop:
If 12 is in the array A, *then* 12 is contained in the range $A[lo, \ldots, hi]$.
lo
hi
1
4
5
7
8
9
10
11
14
15
1
2
· · ·
n

code, it becomes a lot easier to get right. (See Figure 5.9.) Many programming languages allow programmers to use *assertions* to state logical conditions that they believe to always be true at a particular point in the code. A simple `assert(`P`)` statement can help a programmer identify bugs earlier in the development process and avoid a great deal of debugging trauma later.

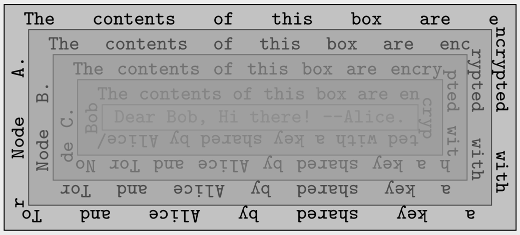
Computer Science Connections

Onion Routing, Steganography, and Conversations whose Very Existence is Sensitive

The typical story introducing cryptography goes like this: two people (Alice and Bob) want to communicate securely, but the only channel over which they are able to share information is insecure; an eavesdropper (Eve) can listen to everything that they say. Some remarkable cryptographic protocols (like RSA; see Section 7.5 for much, much more) have been developed in this context: Bob can publicly post some information about how to communicate with him, and Alice can use that public information to encrypt a message to Bob so that he can easily decode it, but Eve can't.

Cryptographic systems like RSA aim for *confidentiality* of communication: Eve remains fully ignorant of the contents of whatever message Alice sent to Bob. But what is *not* secret is *the fact that Alice sent a secret message to Bob.* That can be a problem: if you work at the Kremlin, the very fact that you just sent a message to Ottawa may be damning, regardless of contents.

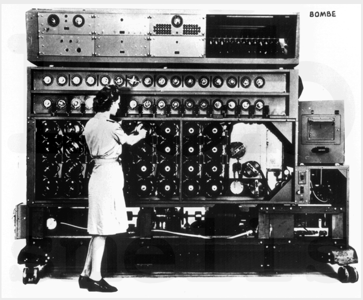
If you're willing to reveal *that* you're communicating (but you want to keep secret *with whom* you are communicating) there's a relevant—and widely used—tool called *Tor* ("the onion router"). Tor is a communication system that combines cryptographic protocols in a recursive way: the basic idea of onion routing is to add multiple "layers" of encryption to a message, which is transmitted across multiple hops—and a layer of encryption is peeled away by each subsequent recipient, like the layers of an onion. (And so an argument for the correctness of this system relies on induction.) If Alice wants to send a message m to Bob, she first identifies a sequence of waypoints—say, Tor nodes A, B, and C—and agrees on a encryption key with each of these intermediaries (using the Diffie–Hellman protocol; see p. 7-67). She computes a quadruply encrypted message $e_A(e_B(e_C(e_{Bob}(m))))$, where the subscript indicates with whom she shares that particular encryption key, and routes it to Bob via A to B to C. See Figure 5.10. (Alice tags each layer of the encryption with routing information: A knows that B is the next recipient [but A doesn't know who C or Bob are], and so forth) The point of these intermediate layers is that no participant knows the identity of both Alice and Bob, and therefore, unless many nodes in the system are operated by the same malicious entity, the identity of a Tor user's communication partner is not revealed. (As long as A, B, and C are simultaneously serving as intermediaries for many different Alices and Bobs, it's not tractable to "connect the dots.")



The contents of this box are encrypted with a key shared by Alice and Tor Node A.
The contents of this box are encrypted with a key shared by Alice and Tor Node B.
The contents of this box are encrypted with a key shared by Alice and Tor Node C.
The contents of this box are encrypted with a key shared by Alice/Bob
Dear Bob, Hi there! --Alice.

Figure 5.10 A message for Bob, in Tor.

Another option: a system for *steganography* (Greek: *stego* hidden/covered + *graphy* writing), which seeks to conceal that communication is happening at all, rather than concealing the recipient or the contents. (Naturally, these ideas can be combined: you could encrypt a message which you then steganographically transmit.) One possibility is to embed a hidden message by altering the least-significant bit of each pixel in an image—rounding each pixel value to an odd or even



Original image from the National Security Agency photograph "WAVE Demonstrating Bombe" (1944)

(This image is adjusted five times more prominently than necessary, to make the message, perhaps, just barely visible in a few places.)

Figure 5.11 An image with a hidden message in the least-significant bits of the pixel values.

value, as in ■ (190) vs. ■ (191), to convey a 0 or a 1—in a way that's all but undetectable visually. (You could even post your photograph publicly, with the idea that only the intended recipient would know to look for a embedded message.) See Figure 5.11.

(You can learn more about Tor in the original paper describing it [39]—including about ways to mitigate risks of malicious Tor nodes, and traffic analysis [an adversary might observe the rate at which particular individuals send and receive data, to try to match up who's talking with whom]. And, while there are plenty of good reasons for wanting a system that, like Tor, allows individuals to communicate in an essentially untraceable way, there are some serious legal and ethical questions, too: truly private communication leads to the possibility of using the system for illegal, unethical, and other awful purposes.)

EXERCISES

Prove that the following claims hold for all integers $n \geq 0$, by induction on n:

5.1 $\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

5.2 $\sum_{i=0}^{n} i^3 = \frac{n^4+2n^3+n^2}{4}$

5.3 $(-1)^n = \begin{cases} 1 & \text{if } n \text{ is even} \\ -1 & \text{if } n \text{ is odd} \end{cases}$

5.4 $\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$

5.5 $\sum_{i=1}^{n} \frac{2}{i(i+2)} = \frac{3}{2} - \frac{1}{n+1} - \frac{1}{n+2}$

5.6 $\sum_{i=1}^{n} i \cdot (i!) = (n+1)! - 1$

In an optical camera, light enters the lens through the opening called the aperture. *The aperture is controlled by a collection of movable blades, which can be adjusted inward to narrow the area through which light can pass. Although some lenses allow continuous adjustment to their openings, many have a sequence of so-called* stops: *discrete steps by which the aperture narrows, called f-*stops *(the "f" is short for "focal"). (See Figure 5.12.)*

The names of f-stops are a little strange, and you'll unwind them here. The "fastest" f-stop for a lens measures the ratio of two numbers: the focal length of the lens divided by the diameter of the aperture of the lens. (For example, a lens that's 50mm *long and has a* 25mm *diameter yields an f-stop of* $\frac{50\text{mm}}{25\text{mm}} = 2$.*) You can also "stop down" a lens from this fastest setting by adjusting the blades to shrink the aperture, as shown in Figure 5.12. (For example, for the* 50mm*-long lens with a* 25mm *diameter, you might reduce the diameter to* 12.5mm*, which yields an f-stop of* $\frac{50\text{mm}}{12.5\text{mm}} = 4$.*)*

5.7 Consider a camera lens with a 50mm focal length, and let $d_0 = 50\text{mm}$ denote the diameter of the lens's aperture diameter. "Stopping down" the lens by one step causes the lens's aperture diameter to shrink by a factor of $\frac{1}{\sqrt{2}}$—that is, the next-smaller aperture diameter for a diameter d_i is defined as $d_{i+1} = \frac{d_i}{\sqrt{2}}$ for any $i \geq 0$. Give a closed-form expression for d_n—that is, a nonrecursive numerical expression whose value is equal to d_n. (Your expression can involve real numbers and the variable n.) Prove your answer correct by induction.

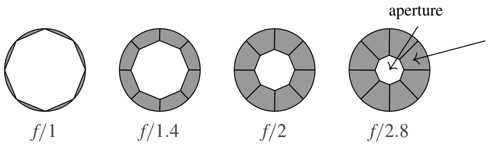
5.8 Using your solution to Exercise 5.7, give a closed-form expression for two further quantities:

- the "light-gathering" area (that is, the area of the aperture) of the lens when its diameter is set to d_n.
- the f-stop f_n of the lens when its diameter is set to d_n.

Using your formula for f_n, can you explain the f-stop names from Figure 5.12?

5.9 What is the sum of the first n odd positive integers? Formulate a conjecture by trying a few examples (for example, what are $1+3$ and $1+3+5$ and $1+3+5+7$, for $n=2$, $n=3$, and $n=4$?). Then prove your answer by induction.

5.10 What is the sum of the first n even positive integers? Prove your answer by induction.


aperture
a blade (note: blade shape is a little different in real lenses)
$f/1$
$f/1.4$
$f/2$
$f/2.8$

Narrowing the aperture has two effects:
(1) it reduces the amount of light entering the lens, which darkens the resulting image, and
(2) it increases the *depth of field* (the range of distances from the lens at which objects are in focus in the image).

Figure 5.12 A particular lens of a camera, at several different f-stops. (Not pictured: $f/4, f/5.6, f/8, f/11, \ldots$.)

5.11 Formulate and prove correct by induction an expression for the sum of the first n *negative* powers of two—that is, $\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^n}$. (If you write this number in binary, it looks like 0.1111111, with a total of n ones.)

5.12 Consider an arithmetic sequence $\langle x, x+r, x+2r, \ldots \rangle$, with $x \in \mathbb{R}$ as the first entry and $r \in \mathbb{R}$ as the amount by which each entry increases over the previous one. (See Definition 5.7. For example, $\langle 1, 3, 5, \ldots \rangle$ has $x = 1$ and $r = 2$; $\langle 25, 20, 15, \ldots \rangle$ has $x = 25$ and $r = -5$; and $\langle 5, 5, 5, \ldots \rangle$ has $x = 5$ and $r = 0$.) An *arithmetic sum* or *arithmetic series* is the sum of an arithmetic sequence. For the arithmetic sequence $\langle x, x+r, x+2r, \ldots \rangle$, formulate and prove correct by induction a formula expressing the sum of the first $n+1$ terms: that is, for the arithmetic series $\sum_{i=0}^{n}(x+ir)$. *(Hint: note that* $\sum_{i=0}^{n} ir = r\sum_{i=0}^{n} i = \frac{rn(n+1)}{2}$*, by Theorem 5.3.)*

In chess, a walk *for a particular piece is a sequence of legal moves for that piece, starting from a square of your choice, that visits* every *square of the board. A* tour *is a walk that visits every square* only *once. (See Figure 5.13.)*

5.13 Prove by induction that there exists a knight's walk of an n-by-n chessboard for any $n \geq 4$. (It turns out that knight's *tours* exist for all even $n \geq 6$, but you don't need to prove this fact.)

5.14 *(programming required.)* In a programming language of your choice, implement your proof from Exercise 5.13 as a recursive algorithm that *computes* a knight's walk in an n-by-n chessboard.

5.15 Prove by induction that there exists a rook's tour for any n-by-n chessboard for any $n \geq 1$.

Figure 5.14 shows three different fractals. Recall that the perimeter *of a shape is the total length of all boundaries separating regions inside the figure from regions outside. (For example, the perimeter includes boundary of the "hole" in the Sierpiński carpet. In fact, for the Sierpiński fractals as drawn here, the perimeter is precisely* the length of lines separating colored-in from uncolored-in regions *and the enclosed area is precisely* the area of the colored-in regions.*) Suppose that we draw each of these fractals at level* ℓ *and with size* 1*. For each of the following, conjecture a formula and prove your answer correct by induction.*

5.16 perimeter of the Von Koch snowflake

5.17 perimeter of the Sierpiński triangle

5.18 perimeter of the Sierpiński carpet

5.19 enclosed area of the Von Koch snowflake

5.20 enclosed area of the Sierpiński triangle

5.21 enclosed area of the Sierpiński carpet

In the last few exercises, you computed the fractals' perimeter and area at level ℓ*. But what if we continued the fractal-expansion process forever? What are the area and perimeter of an* infinite-level *fractal? (Hint: use Corollary 5.6.)*

5.22 perimeter of the infinite Von Koch snowflake

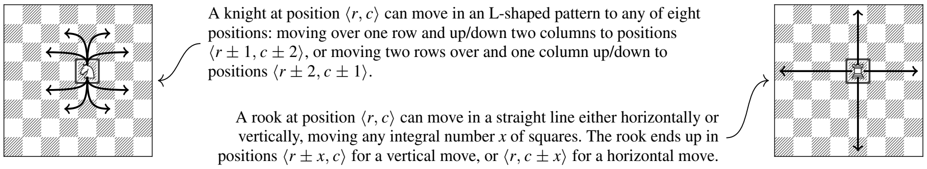
5.23 perimeter of the infinite Sierpiński triangle

5.24 perimeter of the infinite Sierpiński carpet

5.25 enclosed area of the infinite Von Koch snowflake

5.26 enclosed area of the infinite Sierpiński triangle

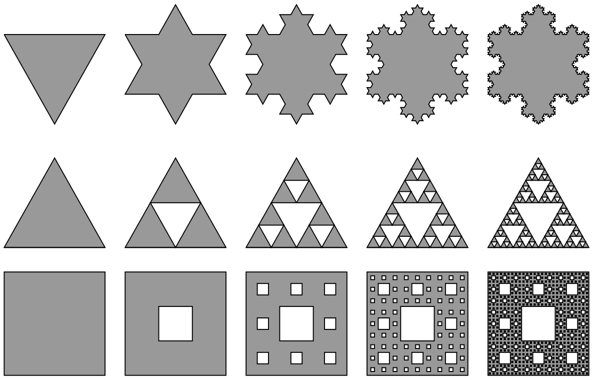
5.27 enclosed area of the infinite Sierpiński carpet



A knight at position $\langle r, c \rangle$ can move in an L-shaped pattern to any of eight positions: moving over one row and up/down two columns to positions $\langle r \pm 1, c \pm 2 \rangle$, or moving two rows over and one column up/down to positions $\langle r \pm 2, c \pm 1 \rangle$.

A rook at position $\langle r, c \rangle$ can move in a straight line either horizontally or vertically, moving any integral number x of squares. The rook ends up in positions $\langle r \pm x, c \rangle$ for a vertical move, or $\langle r, c \pm x \rangle$ for a horizontal move.

Figure 5.13 A chess board, and the legal moves for a knight (left) or rook (right).



A *Von Koch line* of size s and level 0 is just a straight line segment; a Von Koch line of size s and level ℓ consists of four Von Koch lines of size $\frac{s}{3}$ and level $\ell-1$ arranged in the shape ⎯/\⎯; a *Von Koch snowflake* of size s and level ℓ consists of a triangle of three Von Koch lines of size s and level ℓ.

A *Sierpiński triangle* of level 0 and size s is an equilateral triangle of side length s; a Sierpiński triangle of level $\ell+1$ is three Sierpiński triangles of level ℓ and size $\frac{s}{2}$ arranged in a triangle.

A *Sierpiński carpet* of level ℓ and size s is formed from 8 Sierpiński carpets of level $\ell-1$ and size $\frac{s}{3}$ (arranged in a 3-by-3 grid with a hole in the middle); a Sierpiński carpet of level 0 and side length s is a filled s-by-s square.

Figure 5.14 Three fractals—named after the Swedish mathematician Helge von Koch (1870–1924) and the Polish mathematician Wacław Sierpiński (1882–1969)—drawn at levels $0, 1, \ldots, 4$.

5.28 *(programming required.)* Write a recursive function `sierpinski_triangle(level, length, x, y)` in a language of your choice to draw a Sierpiński triangle with the given side length and level with bottom-left coordinate $\langle \mathtt{x}, \mathtt{y} \rangle$. (You'll need to use some kind of graphics package with line-drawing capability.) Write your function so that—in addition to drawing the fractal—it also returns both the *total length* and *total area* of the triangles that it draws. Use your function to verify some small cases of Exercises 5.17 and 5.20.

5.29 *(programming required.)* Write a recursive function `sierpinski_carpet(level, length, x, y)` to draw a Sierpiński carpet. (See Exercise 5.28.) Extend your function to also return the *area* of the boxes that it encloses, and use it to verify some small cases of your answer to Exercise 5.21.

5.30 An *n-by-n magic square* is an n-by-n grid into which the numbers $1, 2, \ldots, n^2$ are placed, once each. The "magic" is that each *row*, *column*, and *diagonal* must be filled with numbers that have the same sum. For example, you can build a 3-by-3 magic square with the top row `[4, 9, 2]`, middle row `[3, 5, 7]`, and bottom row `[8, 1, 6]`. (Check!) Conjecture and prove a formula for what the sum of each row/column/diagonal must be in an n-by-n magic square.

Recall from Section 5.2.2 the harmonic numbers, *where $H_n = \sum_{i=1}^{n} \frac{1}{i}$ is the sum of the reciprocals of the first n positive integers. Further recall Theorem 5.9, which states that $k+1 \geq H_{2^k} \geq \frac{k}{2}+1$ for any integer $k \geq 0$.*

5.31 In Example 5.7, we proved that $k+1 \geq H_{2^k}$. Using the same type of reasoning as in the example, complete the proof of Theorem 5.9: show by induction that $H_{2^k} \geq \frac{k}{2}+1$ for any integer $k \geq 0$.

5.32 Generalize Theorem 5.9 to numbers that aren't necessarily exact powers of 2. Specifically, prove that

$$\log n + 2 \geq H_n \geq \frac{\log n - 1}{2} + 1$$

for any real number $n \geq 1$. *(Hint: use Theorem 5.9.)*

5.33 Prove *Bernoulli's inequality:* let $x \geq -1$ be an arbitrary real number. Prove by induction on n that

$$(1+x)^n \geq 1 + nx$$

for any positive integer n.

Prove that the following inequalities hold "for sufficiently large n." That is, identify an integer k and then prove (by induction on n) that the stated relationship holds for all integers $n \geq k$.

5.34 $2^n \leq n!$

5.35 $b^n \leq n!$, for an arbitrary integer $b \geq 1$

5.36 $3n \leq n^2$

5.37 $n^3 \leq 2^n$

5.38 Prove that, for any nonnegative integer n, the algorithm **odd?**(n) returns True if and only if n is odd. (See Figure 5.15.)

5.39 For the algorithm **sum** in Figure 5.15 (the non-alternative version), prove that **sum**(n, m) returns $\sum_{i=n}^{m} i$ for any $m \geq n$. *(Hint: perform induction on the value of $m - n$.)*

5.40 Modify your proof from Exercise 5.39 for the alternative version of **sum** in Figure 5.15 (which differs only in Line 4).

5.41 Prove by induction on n that $8^n - 3^n$ is divisible by 5 for any nonnegative integer n.

5.42 Conjecture a formula for the value of $9^n \bmod 10$, and prove it correct by induction on n. *(Hint: try computing $9^n \bmod 10$ for a few small values of n to generate your conjecture.)*

5.43 As in the previous exercise, conjecture a formula for the value of $2^n \bmod 7$, and prove it correct.

5.44 Suppose that we count, in binary, using an n-bit counter that goes from 0 to $2^n - 1$. There are 2^n different steps along the way: the initial step of $00 \cdots 0$, and then $2^n - 1$ increment steps, each of which causes at least one bit to be flipped. What is the *average* number of bit flips that occur per step? (Count the first step as changing all n bits.) For example,

$$\underline{000} \to 00\underline{1} \to 0\underline{10} \to 01\underline{1} \to \underline{100} \to 10\underline{1} \to 1\underline{10} \to 11\underline{1}$$

for $n = 3$, which has a total of $3 + 1 + 2 + 1 + 3 + 1 + 2 + 1 = 14$ bit flips. Prove your answer.

5.45 To protect my backyard from my neighbor, a biology professor who is sometimes a little overfriendly, I have acquired a large army of vicious robotic dogs. Unfortunately the robotic dogs in this batch are very jealous, and they must be separated by fences—in fact, they can't even *face* each other directly through a fence. So I have built a collection of n fences to separate my backyard into polygonal regions, where each fence completely crosses my yard (that is, it goes from property line to property line, possibly crossing other fences). I wish to deploy my robotic dogs to satisfy the following property:

> For *any* two polygonal regions that share a boundary (that is, are separated by a fence segment), one of the two regions has exactly one robotic dog and the other region has zero robotic dogs.

(See Figure 5.16.) Prove by induction on n that this condition is satisfiable for *any* collection of n fences.

```
odd?(n):
1 if n = 0 then
2     return False
3 else
4     return not odd?(n − 1)
```

```
sum(n, m):
1 if n = m then
2     return m
3 else
4     return n + sum(n + 1, m)
```

```
sum(n, m): // ALTERNATIVE
1 if n = m then
2     return m
3 else
4     return m + sum(n, m − 1)
```

Figure 5.15 Two algorithms (and one variation).

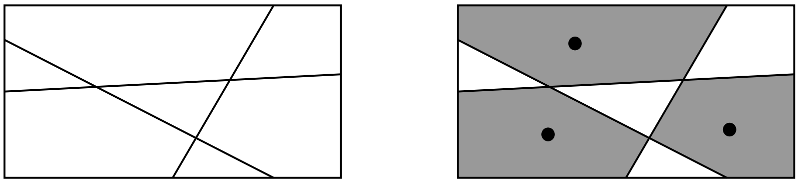


Figure 5.16 A configuration of fences, and a valid way to deploy my dogs (the dots in the shaded regions).

5.3 Strong Induction

> Step by step, little by little, bit by bit—that is the way to wealth, that is the way to wisdom, that is the way to glory. Pounds are the sons, not of pounds, but of pence.
>
> Charles Buxton (1823–1871)
> *Notes of Thought* (1873)

In the proofs by induction in Section 5.2, we established the claim $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$ by proving $P(0)$ [the base case] and proving that $P(n-1) \Rightarrow P(n)$ [the inductive case]. But let's think again about what happens in an inductive proof, as we build up facts about $P(n)$ for ever-increasing values of n. (Glance at Example 5.1 again.)

1. We prove $P(0)$.
2. We prove $P(0) \Rightarrow P(1)$, so we conclude $P(1)$, using Fact #1.

Now we wish to prove $P(2)$. In a proof by induction like those from Section 5.2, we'd proceed as follows:

3. We prove $P(1) \Rightarrow P(2)$, so we conclude $P(2)$, using Fact #2.

In a *proof by strong induction*, we allow ourselves to make use of more assumptions: namely, we know that $P(1)$ *and* $P(0)$ when we're trying to prove $P(2)$. (By way of contrast, we'll refer to proofs like those from Section 5.2 as using *weak* induction.) In a proof by strong induction, we proceed as follows instead:

3′. We prove $P(0) \wedge P(1) \Rightarrow P(2)$, so we conclude $P(2)$, using Fact #1 and Fact #2.

In a proof by strong induction, in the inductive case we prove $P(n)$ by assuming n different inductive hypotheses: $P(0)$, $P(1)$, $P(2)$, …, and $P(n-1)$. Or, less formally: in the inductive case of a proof by weak induction, we show that *if P "was true last time" then it's still true this time;* in the inductive case of a proof by strong induction, we show that *if P "has been true up until now" then it's still true this time.*

5.3.1 A Definition and a First Example

Here is the formal definition of a proof by strong induction:

Definition 5.10: Proof by strong induction.
Suppose that we want to prove that $P(n)$ holds for all $n \in \mathbb{Z}^{\geq 0}$. To give a *proof by strong induction* of $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$, we prove the following:

1 the *base case:* prove $P(0)$.
2 the *inductive case*: for every $n \geq 1$, prove $[P(0) \wedge P(1) \wedge \cdots \wedge P(n-1)] \Rightarrow P(n)$.

Generally speaking, using strong induction makes sense when the "reason for" $P(n)$ is that $P(k)$ is true for more than one index $k \leq n-1$, or that $P(k)$ is true for some index $k \leq n-2$. (For weak induction, the "reason for" $P(n)$ is that $P(n-1)$ is true.)

Strong induction makes the inductive case easier to prove than weak induction, because the claim that we need to show—that is, $P(n)$—is the same, but we get to use more assumptions in strong induction: in strong induction, we've assumed all of $P(0) \wedge P(1) \wedge \ldots \wedge P(n-1)$; in weak induction, we've assumed only $P(n-1)$. We can always ignore those extra assumptions, so it's never harder to prove something by strong induction than with weak induction.

Writing tip: While anything that can be proven using weak induction can also be proven using strong induction, you should still use the tool that's best suited to the job—generally, the one that makes the argument easiest to understand.

(Strong induction is actually equivalent to weak induction in its power; anything that can be proven with one can also be proven with the other. See Exercises 5.81–5.82.)

A first example: a simple algorithm for parity

In the rest of this section, we'll give several examples of proofs by strong induction. We'll start here with a proof of correctness for a blazingly simple algorithm that computes the parity of a positive integer. (Recall that the *parity* of n is the "evenness" or "oddness" of n.) See Figure 5.17 for the **parity** algorithm.

We've already used (weak) induction to prove the correctness of recursive algorithms that, given an input of size n, call themselves on an input of size $n-1$. (That's how we proved the correctness of the factorial algorithm **fact** from Example 5.9.) But for recursive algorithms that call themselves on smaller inputs but not necessarily of size $n-1$, like **parity**, we can use strong induction to prove their correctness.

Example 5.11: The correctness of **parity**.

Claim: For any nonnegative integer $n \geq 0$, we have that $\textbf{parity}(n) = n \bmod 2$.

Proof. Write $P(n)$ to denote the property that $\textbf{parity}(n) = n \bmod 2$. We proceed by strong induction on n to show that $P(n)$ holds for all $n \geq 0$:

Base cases ($n = 0$ and $n = 1$). By inspection of the algorithm, $\textbf{parity}(0)$ returns 0 in Line 2, and, indeed, $0 \bmod 2 = 0$. Similarly, we have $\textbf{parity}(1) = 1$, and $1 \bmod 2 = 1$ too. Thus $P(0)$ and $P(1)$ hold.

Inductive case ($n \geq 2$). Assume the inductive hypothesis $P(0) \wedge P(1) \wedge \cdots \wedge P(n-1)$—that is, assume

$$\text{for any integer } 0 \leq k < n, \text{ we have } \textbf{parity}(k) = k \bmod 2.$$

We must prove $P(n)$—that is, we must prove $\textbf{parity}(n) = n \bmod 2$:

$$\begin{aligned} \textbf{parity}(n) &= \textbf{parity}(n-2) && \textit{by inspection (specifically because } n \geq 2 \textit{ and by Line 4)} \\ &= (n-2) \bmod 2 && \textit{by the inductive hypothesis } P(n-2) \end{aligned}$$

```
parity($n$):    // assume that $n \geq 0$ is an integer
if $n \leq 1$ then
    return $n$
else
    return parity($n - 2$)
```

Figure 5.17 An algorithm that computes whether n is odd or even.

$$= n \bmod 2,$$

where $(n - 2) \bmod 2 = n \bmod 2$ by Definition 2.11. (Note that the inductive hypothesis applies for $k = n - 2$ because $n \geq 2$ and therefore $0 \leq n - 2 < n$.) □

There are two things to note about the proof in Example 5.11. First, using strong induction instead of weak induction made sense because the inductive case relied on $P(n - 2)$ to prove $P(n)$; we did *not* show $P(n - 1) \Rightarrow P(n)$. Second, we needed two base cases: the "reason" that $P(1)$ holds is *not* that $P(-1)$ was true. (In fact, $P(-1)$ is false—**parity**(-1) isn't equal to 1! Think about why.) The inductive case of the proof in Example 5.11 does not correctly apply for $n = 1$, and therefore we had to handle that case separately, as a second base case.

5.3.2 Some Further Examples of Strong Induction

We'll continue this section with several more examples of proofs by strong induction. We'll first turn to a proof about *prime factorization* of integers, and then look at one geometric and one algorithmic claim.

Prime factorization

Recall that an integer $n \geq 2$ is *prime* if the only positive integers that evenly divide it are 1 and n itself. It's a basic fact about numbers that any positive integer can be uniquely expressed as the product of primes:

Theorem 5.11: Prime Factorization Theorem.
Let $n \in \mathbb{Z}^{\geq 1}$ be any positive integer. Then there exist $k \geq 0$ prime numbers $p_1, p_2, \ldots, p_k$ such that $n = \prod_{i=1}^{k} p_i$. Furthermore, up to reordering, the primes $p_1, p_2, \ldots, p_k$ are unique.

(The prime factorization theorem is also sometimes called the *Fundamental Theorem of Arithmetic*.) While proving the *uniqueness* requires a bit more work (we'll prove it in Section 7.3.3), we can give a proof using strong induction to show that a prime factorization *exists*.

Example 5.12: A prime factorization exists.
Let $P(n)$ denote the first part of Theorem 5.11, namely the claim

$$\textit{there exist } k \geq 0 \textit{ prime numbers } p_1, p_2, \ldots, p_k \textit{ such that } n = \prod_{i=1}^{k} p_i.$$

We will prove that $P(n)$ holds for any integer $n \geq 1$, by strong induction on n.

Base case ($n = 1$). Recall that the product of zero multiplicands is 1. (See Section 2.2.7 for a reminder of the definition.) Thus we can write n as the product of *zero* prime numbers. Thus $P(1)$ holds.

Inductive case ($n \geq 2$). We assume the inductive hypothesis—namely, we assume that $P(n')$ holds for any positive integer n' where $1 \leq n' \leq n - 1$. We must prove $P(n)$. There are two cases:

Case I: n is prime. If n is prime, then there's nothing to do: we can write n as the product of one prime number, namely n itself. Formally, let $k = 1$ and $p_1 = n$, and we're done immediately.

Case II: n is not prime. If n is not prime, then by definition n can be written as the product $n = a \cdot b$, for positive integers a and b satisfying $2 \leq a \leq n-1$ and $2 \leq b \leq n-1$. (The definition of (non)primality says that $n = a \cdot b$ for $a \notin \{1, n\}$; you should be able to convince yourself that neither a nor b can be smaller than 2 or larger than $n - 1$.) By the inductive hypotheses $P(a)$ and $P(b)$, we have

$$a = q_1 \cdot q_2 \cdot \; \cdots \; \cdot q_\ell \qquad \text{and} \qquad b = r_1 \cdot r_2 \cdot \; \cdots \; \cdot r_m \tag{$*$}$$

for prime numbers $q_1, \ldots, q_\ell$ and $r_1, \ldots, r_m$. By $(*)$ and the fact that $n = a \cdot b$,

$$n = q_1 \cdot q_2 \cdot \; \cdots \; \cdot q_\ell \cdot r_1 \cdot r_2 \cdot \; \cdots \; \cdot r_m.$$

Because each q_i and r_i is prime, we have now written n as the product of $\ell + m$ prime numbers, and $P(n)$ holds. The theorem follows.

Taking it further: As with any inductive proof, it may be useful to view the proof from Example 5.12 as a recursive algorithm. Here's a sketch of the algorithm **primeFactor**(n) implicitly defined by this proof:

- if $n = 1$, then return $\langle\rangle$. *(The base case: $P(1)$ is true.)*
- if $n > 1$ and n is prime, then return $\langle n \rangle$. *(The inductive case, part I: if $n \geq 2$ is prime, then $P(n)$ is true.)*
- otherwise $n > 1$ and n is not prime. In this case: *(The inductive case, part II: if $n \geq 2$ is not prime, then $P(n)$ is true.)*
  - Find factors a and b where $2 \leq a \leq n - 1$ and $2 \leq b \leq n - 1$ such that $n = a \cdot b$. $(*)$
  - Recursively compute $\langle q_1, ..., q_k \rangle :=$ **primeFactor**(a).
  - Recursively compute $\langle r_1, ..., r_m \rangle :=$ **primeFactor**(b).
  - Return $\langle q_1, ..., q_k, r_1, ..., r_m \rangle$.

(Notice that there's some magic in the "algorithm," in the sense that Line $(*)$ doesn't tell us *how* to find the values of a and b—but we do know that such values exist, by definition.) We can think of the inductive case of an inductive proof as "making a recursive call" to a proof for a smaller input. For example, **primeFactor**(2) returns $\langle 2 \rangle$ and **primeFactor**(5) returns $\langle 5 \rangle$, because both 2 and 5 are prime. For another example, the result of **primeFactor**(10) is $\langle 2, 5 \rangle$, because 10 is not prime, but we can write $10 = 2 \cdot 5$ and **primeFactor**(2) returns $\langle 2 \rangle$ and **primeFactor**(5) returns $\langle 5 \rangle$. The result of **primeFactor**(70) could be $\langle 7, 2, 5 \rangle$, because 70 is not prime, but we can write $70 = 7 \cdot 10$ and **primeFactor**(7) returns $\langle 7 \rangle$ and **primeFactor**(10) returns $\langle 2, 5 \rangle$. Or **primeFactor**(70) could be $\langle 7, 5, 2 \rangle$ because $70 = 35 \cdot 2$, and **primeFactor**(35) returns $\langle 7, 5 \rangle$ and **primeFactor**(2) returns $\langle 2 \rangle$. (Which ordering of the values is the output depends on the magic of Line $(*)$. The second part of Theorem 5.11, about the uniqueness of the prime factorization, says that it is only the ordering of these numbers that depends on the magic; the numbers themselves must be the same.)

Triangulating a polygon

We'll now turn to a proof by strong induction about a geometric question, instead of a numerical one. A *convex polygon* is, informally, the points "inside" a set of n vertices: imagine stretching a giant rubber band around n points in the plane; the polygon is defined as the set of all points contained inside the rubber band. See Figure 5.18a for an example. Here we will show that an arbitrary convex polygon can be decomposed into a collection of nonoverlapping triangles.

Example 5.13: Decomposing a polygon into triangles.
Prove the following claim. (For an example, and an outline of a possible proof, see Figure 5.18.)

Claim: Any convex polygon P with $k \geq 3$ vertices can be decomposed into a set of $k-2$ triangles whose interiors do not overlap.

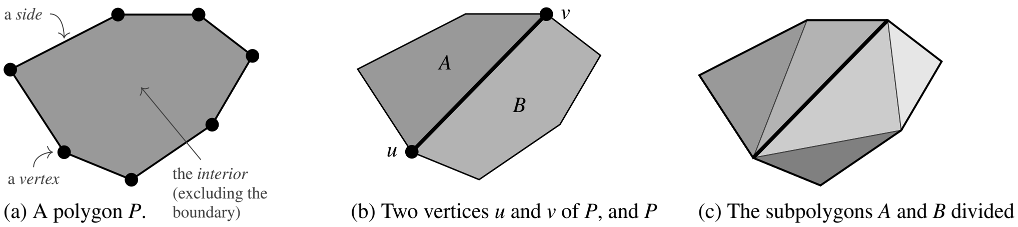
Solution. Let $Q(k)$ denote the claim that any k-vertex polygon can be decomposed into a set of $k-2$ interior-disjoint triangles. We'll give a proof by strong induction on k that $Q(k)$ holds for all $k \geq 3$. (Note that strong induction isn't strictly necessary to prove this claim; we could give an alternative proof using weak induction.)

Base case ($k = 3$). There's nothing to do: any 3-vertex polygon P is itself a triangle, so the collection $\{P\}$ is a set of $k-2=1$ triangles whose interiors do not intersect (vacuously, because there is only one triangle). Thus $Q(3)$ holds.

Inductive case ($k \geq 4$). We assume the inductive hypothesis, which states: any convex polygon with $3 \leq \ell < k$ vertices can be decomposed into a set of $\ell - 2$ interior-disjoint triangles. (That is, we assume $Q(3), Q(4), \ldots, Q(k-1)$.) We must prove $Q(k)$.

Let P be an arbitrary k-vertex polygon. Let u and v be any two nonadjacent vertices of P. (Because $k \geq 4$, such a pair exists.) Define A as the "above the $\langle u, v\rangle$ line" piece of P and B as the "below the $\langle u, v\rangle$ line" piece of P. Notice that $P = A \cup B$, both A and B are convex, and the interiors of A and B are disjoint. Let ℓ be the number of vertices in A. Observe that $\ell \geq 3$ and $\ell < k$ because u and v are nonadjacent. Also observe that B contains precisely $k - \ell + 2$ vertices. (The "$+2$" is because vertices u and v appear in both A and B.) Note that both $3 \leq \ell \leq k-1$ and $3 \leq k - \ell + 2 \leq k-1$, so we can apply the inductive hypothesis to both ℓ and $k - \ell + 2$.

Therefore, by the inductive hypothesis $Q(\ell)$, the polygon A is decomposable into a set S of $\ell - 2$ interior-disjoint triangles. Again by the inductive hypothesis $Q(k - \ell + 2)$, the polygon B is decomposable into a set T of $k - \ell + 2 - 2 = k - \ell$ interior-disjoint triangles. Furthermore because A and B are interior



(a) A polygon P.

(b) Two vertices u and v of P, and P divided into A and B (above and below the u-to-v line).

(c) The subpolygons A and B divided into triangles, using the inductive hypothesis.

Figure 5.18 An example of the recursive decomposition of a polygon into interior-disjoint triangles.

disjoint, the triangles of $S \cup T$ all have disjoint interiors. Thus P itself can be decomposed into the union of these two sets of triangles, yielding a total of $\ell - 2 + k - \ell = k - 2$ interior-disjoint triangles.

We've shown both $Q(3)$ and $Q(3) \wedge \cdots \wedge Q(k-1) \Rightarrow Q(k)$ for any $k \geq 4$, which completes the proof by strong induction.

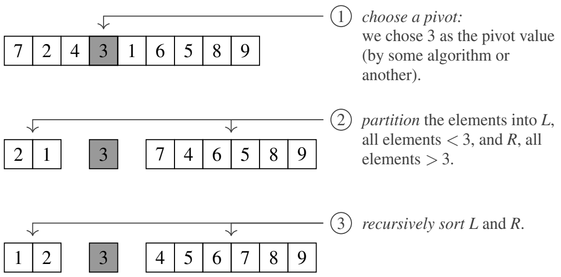
Taking it further: The style of *triangulation* from Example 5.13 has important implications in computer graphics, in which we seek to render representations of complicated real-world scenes using computational techniques. In many computer graphics applications, complex surfaces are decomposed into small triangular regions, which are then rendered individually. See p. 5-36.

Proving algorithms correct: Quick Sort

We've now seen a proof of correctness by strong induction for a small (4-line) recursive algorithm (for parity), and proofs of somewhat more complicated non-algorithmic properties. Here we'll prove the correctness of a somewhat more complicated algorithm—the recursive sorting algorithm called *Quick Sort*—again using strong induction.

```
quickSort($A[1 \dots n]$):
if $n \leq 1$ then
    return $A$
else
    choose $pivot \in \{1, \dots, n\}$, somehow.
    $L := \langle\rangle$
    $R := \langle\rangle$
    for $i \in \{1, \dots, n\}$ with $i \neq pivot$:
        if $A[i] < A[pivot]$ then
            append $A[i]$ to $L$
        else
            append $A[i]$ to $R$
    $L :=$ quickSort($L$)
    $R :=$ quickSort($R$)
    return $L + \langle A[pivot] \rangle + R$
```

(a) The pseudocode.


7 2 4 3 1 6 5 8 9
① *choose a pivot:* we chose 3 as the pivot value (by some algorithm or another).
2 1 3 7 4 6 5 8 9
② *partition* the elements into L, all elements < 3, and R, all elements > 3.
1 2 3 4 5 6 7 8 9
③ *recursively sort* L *and* R.

(b) An example of quick sort.

Figure 5.19 Quick Sort: pseudocode, and an example.

The idea of the Quick Sort algorithm is to select a *pivot* value x from an input array A; we then partition the elements of A into those less than x (which we then sort recursively), then x itself, and finally the elements of A greater than x (which we again sort recursively). We also need a base case: an input array with fewer than 2 elements is already sorted. (See Figure 5.19a for the algorithm.)

For example, suppose we wish to sort by birthday the 53 winners of the Nobel Prize for Literature from the second half of the 20th century. Toni Morrison's birthday is February 18th. If we choose her as the pivot, then Quick Sort would first divide all 52 other winners into two lists, of those with pre–February 18th birthdays and those with post–February 18th birthdays,

$$\begin{aligned} \mathit{before}[1 \dots 4] &:= \langle \text{Boris Pasternak (1958) [February 10th]}, \dots, \text{Gao Xingjian (2000) [January 4th]} \rangle \\ \mathit{after}[1 \dots 48] &:= \langle \text{Bertrand Russell (1950) [May 18th]}, \dots, \text{Günter Grass (1999) [October 16th]} \rangle, \end{aligned}$$

and then recursively sort *before* and *after*. Then the final sorted list will be

$$\underbrace{\mathit{before} \text{ in sorted order}}_{\mathit{winner}[1], \dots, \mathit{winner}[4],} \qquad \underbrace{\text{Toni Morrison}}_{\mathit{winner}[5],} \qquad \underbrace{\mathit{after} \text{ in sorted order}}_{\mathit{winner}[6], \dots, \mathit{winner}[53]}$$

While the efficiency of Quick Sort depends crucially on *how* we choose the pivot value (see Chapter 6), the correctness of the algorithm holds regardless of that choice. For simplicity, we will prove that Quick Sort correctly sorts its input under the assumption that all the elements of the input array A are distinct. (The more general case, in which there may be duplicate elements, is conceptually no harder, but is a bit more tedious.) It is easy to see by inspection of the algorithm that **quickSort**(A) returns a reordering of the input array A—that is, Quick Sort neither deletes or inserts elements. Thus the real work is to prove that Quick Sort returns a sorted array.

Taking it further: There were 53 winners between 1950—the year in which Bertrand Russell, of Russell's Paradox fame (see p. 2-31), won the award for his writings on philosophy—and 2000, when Gao Xingjian won the prize. As it happens, the simplifying assumption that we're making about distinct elements doesn't apply for this set of Nobelists: Samuel Beckett (1969) and Seamus Heaney (1995) share a birthday, as do Naguib Mahfouz (1988) and Aleksandr Solzhenitsyn (1970). Thank about how you would modify the proof in Example 5.14 to handle duplicates.

Example 5.14: Correctness of Quick Sort.

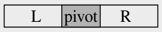
Claim: For any array A with distinct elements, **quickSort**(A) returns a sorted array.

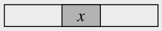
Proof. Let $P(n)$ denote the claim that **quickSort**$(A[1 \dots n])$ returns a sorted array for any n-element array A with distinct elements. We'll prove $P(n)$ for every $n \geq 0$, by strong induction on n.

Base cases ($n = 0$ and $n = 1$). Both $P(0)$ and $P(1)$ are trivial: any array of length 0 or 1 is sorted.

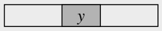
Inductive case ($n \geq 2$). We assume the inductive hypothesis $P(0), \dots, P(n-1)$: for any array $B[1 \dots k]$ with distinct elements and length $k < n$, **quickSort**(B) returns a sorted array. We must prove $P(n)$.

Let $A[1 \ldots n]$ be an arbitrary array with distinct elements. Let $pivot \in \{1, \ldots, n\}$ be arbitrary. We must prove that x appears before y in **quickSort**(A) if and only if $x < y$. We proceed by cases, based on the relationship between x, y, and $A[pivot]$.

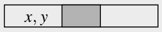


Case 1  *: $x = A[pivot]$.*

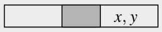
The elements appearing after x in **quickSort**(A) are precisely the elements of R. And R is exactly the set of elements greater than x. Thus x appears before y if and only if y appears in R, which occurs if and only if $x < y$.

Case 2  *: $y = A[pivot]$.*

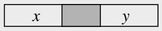
Analogously to Case 1, x appears before y if and only if x appears in L, which occurs if and only if $x < y$.

Case 3  *: $x < A[pivot]$ and $y < A[pivot]$.*

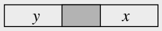
Then both x and y appear in L. Because $A[pivot]$ does *not* appear in L, we know that L contains at most $n - 1$ elements, all of which are distinct because they're a subset of the distinct elements of A. Thus the inductive hypothesis $P(|L|)$ says that x appears before y in **quickSort**(L) if and only if $x < y$. And x appears before y in **quickSort**(A) if and only if x appears before y in **quickSort**(L).

Case 4  *: $x > A[pivot]$ and $y > A[pivot]$.*

Then both x and y appear in R. An analogous argument to Case 3 shows that x appears before y if and only if $x < y$.

Case 5  *: $x < A[pivot]$ and $y > A[pivot]$.*

It is immediate both that x appears before y (because x is in L and y is in R) and that $x < y$.

Case 6  *: $x > A[pivot]$ and $y < A[pivot]$.*

It is immediately apparent that x does not appear before y and that $x \not< y$.

In all six cases, we have established that $x < y$ if and only if x appears before y in the output array; furthermore, the cases are exhaustive. The claim follows. □

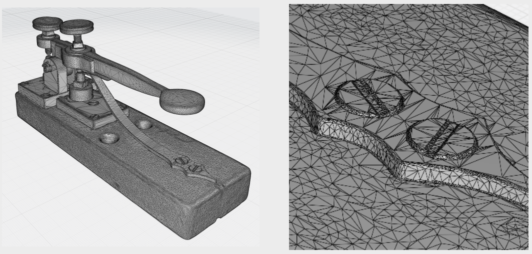
Taking it further: In addition to proofs of correctness for algorithms, like the one for **quickSort** that we just gave, strong induction is crucial in analyzing the efficiency of recursive algorithms; we'll see many examples in Section 6.4. And strong induction can also be fruitfully applied to understanding (and designing!) data structures—for example, see p. 5-38 for a discussion of *maximum heaps*.

Computer Science Connections

Triangulation, Computer Graphics, and 3D Surfaces

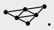
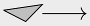
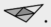
Here is a typical problem in computer graphics: we are given a three-dimensional *scene* consisting of a collection of objects of various shapes and sizes, and we must render a two-dimensional *image* that is a visual display of the scene. (Computer graphics uses a lot of matrix computation to facilitate the *projection* of a 3-dimensional shape onto a 2-dimensional surface.)

A typical approach—to simplify and speed the relevant algorithms—approximates the three-dimensional shapes of the objects in the scene using triangles instead of arbitrary shapes. (See Figure 5.20.) Triangles are the easiest shape to process computationally: the "real" triangle in the scene can be specified completely by three 3-dimensional points corresponding to the vertices; and the rendered shape in the image is still a triangle specified completely by 2-dimensional points corresponding to the vertices' projections onto the image. Specialized hardware called a *graphics processing unit (GPU)* makes these computations extremely fast on many modern computers.

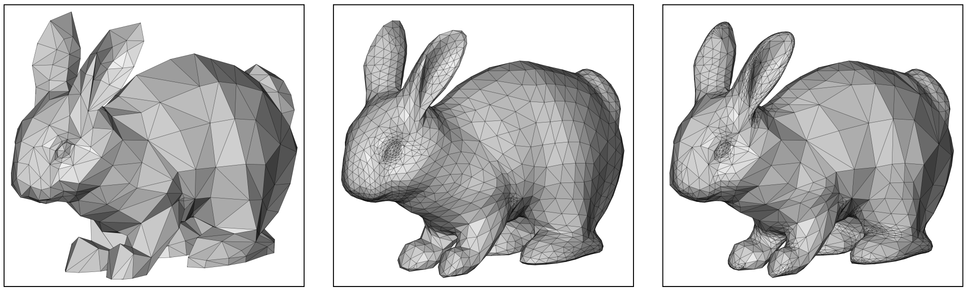


Images generated using a 3D model from the Smithsonian Institution's National Museum of American History.

Figure 5.20 A 3D model of an 1844 telegraph key (made by Alfred Vail based on a design by Samuel Morse), and a close-up of the screws in the bottom-right corner. The full model contains 150,000 triangles.

We can approximate any three-dimensional shape arbitrarily well using a collection of triangles, and we can *refine* a triangulation by dividing splitting one triangle into two pieces, and then properly rendering each constituent triangle: for example, we can take the triangle , and split it into four subtriangles by placing new vertices at the midpoint of each edge; we can then connect up the resulting 6 points (3 original + 3 new) to produce . When rendering a scene, we might compute a single color c that best represents the color of a given triangle in the real scene, and then display a solid c-colored (projected) triangle in the image; thus our refined triangulation can have more color variation than the single triangle, as in  $\longrightarrow$ . Note that there are many different ways to subdivide a given triangle into separate triangles. Which subdivision we pick might depend on the geometry of the scene; for example, we might try to make the subtriangles roughly similar in size, or maximally different in color.

The larger the number of triangles we use, the better the match between the real 3-dimensional shape and the triangulated approximation. But, of course, the more triangles we use, the more computation must be done (and the slower the rendering will be). By identifying particularly important triangles—for example, those whose colors vary particularly widely, or those at a particularly steep angle to their neighbors, or those whose angles to the viewer are particularly extreme—we can selectively refine "the most important parts" of the triangulation to produce higher quality images. (See Figure 5.21.)



Images by Tobias Isenberg; printed with permission under CC BY-SA 4.0 https://creativecommons.org/licenses/by-sa/4.0/.

Figure 5.21 Two strategies for refining a triangulation of a rabbit: one based on dividing each triangle four smaller ones, and one that chooses which triangles to refine based on local geometric properties. (The adaptive strategy on the far right produces a smaller image file, but one that is more accurate in the most interesting parts of the image.) See [62].

Computer Science Connections

Max Heaps

When we design data structures to support particular operations, it is often the case that we wish to maintain some properties in the way that the data are stored. Here's one example, for an implementation of *priority queues,* that we'll establish using a proof by mathematical induction. A priority queue is a data structure that stores a set of jobs, each of which has a *priority*; we wish to be able to insert new jobs (with specified priorities) and identify/extract the existing job with the highest priority.

A *maximum heap* is one way of implementing priority queues. A maximum heap is a binary tree—see Section 5.4 or Chapter 11—in which every node stores a job with an associated priority. Every node in the tree satisfies the *(maximum) heap property* (see Figure 5.22a): the priority of node u must be greater than or equal to the priorities of each of u's children. (A heap must also satisfy another property, being "nearly complete"—intuitively, a heap has no "missing nodes" except in the bottommost layer; this "nearly complete" property is what guarantees that heaps implement priority queues very efficiently.) An example of a heap is shown in Figure 5.22b. It's easy to check that the topmost node (the *root*) of the maximum heap in Figure 5.22b has the highest priority. Heaps are designed so that the root of the tree contains the node with the highest priority—that's a property that ensures that particular operations related to priority queues can be carried out very quickly using heaps. Let's prove this fact by induction:

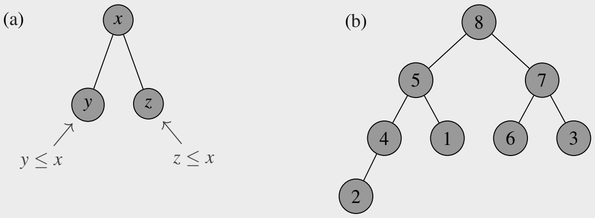


Figure 5.22 Maximum heaps: (a) the heap property, which says that, for any node with value x, the children of that node must have values $\leq x$; and (b) an example.

Claim: In any binary tree in which every node satisfies the maximum heap property, the node with the highest priority is the root of the tree.

Proof. We'll proceed by strong induction on the number of layers of nodes in the tree. (This proof is an example of a situation in which it's not immediately clear upon what quantity to perform induction, but once we've chosen the quantity well, the proof itself is fairly easy.) Let $P(\ell)$ denote the claim *In any tree containing ℓ layers of nodes, in which every node satisfies the maximum heap property, the node with the highest priority is the root of the tree.* We will prove that $P(\ell)$ holds for all $\ell \geq 1$ by strong induction on ℓ.

Base case ($\ell = 1$). The tree has only one level—that is, the root *is* the only node in the tree. Thus, vacuously, the root has the highest priority, because there are no other nodes.

Inductive case ($\ell \geq 2$). We assume the inductive hypothesis $P(1), \ldots, P(\ell - 1)$. Let x be the priority of the root of the tree. (See Figure 5.23 for the cases.) If the root has only one child, say with priority a, then by the inductive hypothesis every element y beneath a satisfies $y \leq a$. (There are at most $\ell - 1$

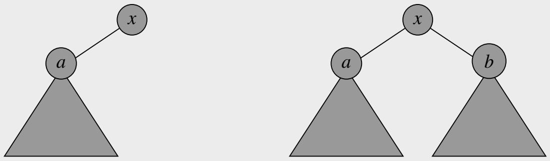


Figure 5.23 The two possibilities for the inductive case of the proof: either x has one child (here, a) or x has two children (here, a and b). (Because the tree has $\ell \geq 2$ levels, the root x has at least one child.)

layers in the tree beneath a, so the inductive hypothesis applies.) By the heap property, we know $a \leq x$, and thus every element y satisfies $y \leq x$. If the root has a second child, say with priority b, then by the inductive hypothesis every element z beneath b satisfies $z \leq b$. (There are at most $\ell - 1$ layers in the tree beneath b, so the inductive hypothesis applies again.) Again, by the heap property, we have $b \leq x$, so every element z satisfies $z \leq x$. □

EXERCISES

5.46 In Example 5.11, we showed the correctness of the **parity** function (see Figure 5.24)—that is, for any $n \geq 0$, we have that **parity**$(n) = n \bmod 2$. Prove by strong induction on n that the *depth* of the recursion (that is, the total number of calls to **parity** made) is $1 + \lfloor \frac{n}{2} \rfloor$.

5.47 Consider the **toBinary** algorithm in Figure 5.25, which finds the binary representation of a given integer $n \geq 0$. For example, **toBinary**(13) returns $\langle 1, 1, 0, 1 \rangle$ (and $13 = 8 + 4 + 0 + 1$ is written as 1101 in binary). Prove the correctness of **toBinary** by strong induction: prove that for any $n \geq 0$, we have $\sum_{i=0}^{k} b_i 2^i = n$, where **toBinary**$(n) = \langle b_k, \ldots, b_0 \rangle$.

5.48 Exercise 5.47 establishes that any nonnegative integer can be represented in binary. Now, let's show that this binary representation is *unique*—or, at least, unique if we ignore the presence or absence of leading zeros. (For example, we can represent 7 in binary as 111 or 0111 or 00111.) To do so, prove that every nonnegative integer n that can be represented as a k-bit string is *uniquely* represented as a k-bit bitstring. In other words, prove the following claim, for any integer $k \geq 1$:

> Let $a = \langle a_k, a_{k-1}, \ldots, a_0 \rangle$ and $b = \langle b_k, b_{k-1}, \ldots, b_0 \rangle$ be two k-bit sequences.
> If $\sum_{i=0}^{k} a_i 2^i = \sum_{i=0}^{k} b_i 2^i$, then for all $i \in \{k, k-1, \ldots, 0\}$ we have $a_i = b_i$.

Your proof should be by (weak) induction on k.

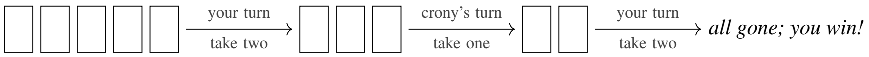
In Chapter 7, we'll talk in detail about modular arithmetic, including a more general algorithm for converting from one base to another (p. 7-16). There, we'll do most of the computation iteratively; here you'll fill in a few pieces recursively.

5.49 Generalize the **parity**(n) algorithm to **remainder**(n, k) to recursively compute the number $r \in \{0, 1, \ldots, k-1\}$ such that **remainder**$(n, k) = n \bmod k$. Assume that $k \geq 1$ and $n \geq 0$ are both integers, and follow the same algorithmic outline as in Figure 5.24. Prove your algorithm correct using strong induction on n.

5.50 Generalize the **toBinary**(n) algorithm to **baseConvert**(n, k) to recursively convert the integer n to base k. Assume that $k \geq 2$ and $n \geq 0$ are both integers, and follow the same algorithmic outline as in Figure 5.25. Prove using strong induction on n that if **baseConvert**$(n, k) = \langle b_\ell, b_{\ell-1}, \ldots, b_0 \rangle$ with each $b_i \in \{0, 1, \ldots, k-1\}$, then $n = \sum_{i=0}^{\ell} k^i b_i$.

5.51 *(programming required.)* Implement your **remainder** and **baseConvert** algorithms, in a language of your choice.

You are sitting around the table with a crony you're in cahoots with. You and the crony decide to play the following silly game. (The two of you run a store that sells nothing but vicious robotic dogs. The loser of the game has to clean up the yard where the dogs roam—not a pleasant chore—so the stakes are high.) We start with n $\in \mathbb{Z}^{\geq 1}$ stolen credit cards on a table. The two players take turns removing cards from the table. In a single turn, a player chooses to remove either one or two cards. A player wins by taking the last card. For example:



5.52 Prove (by strong induction on n) that if n is divisible by three, then the second player to move can guarantee a win, and if n is not divisible by three, then the first player to move can guarantee a win.

```
parity(n):    // assume that $n \geq 0$ is an integer
if $n \leq 1$ then
    return $n$
else
    return parity($n - 2$)
```

Figure 5.24 A reminder of **parity** (from Figure 5.17).

```
toBinary(n):    // assume that $n \geq 0$ is an integer
if $n \leq 1$ then
    return $\langle n \rangle$
else
    $\langle b_k, \ldots, b_0 \rangle$ := toBinary($\lfloor \frac{n}{2} \rfloor$)
    $x$ := parity($n$)
    return $\langle b_k, \ldots, b_0, x \rangle$
```

Figure 5.25 An algorithm to convert an integer to binary.

5.53 Determine who wins the "Take k" modification of the game from Exercise 5.52: conjecture a condition on n that describes precisely when the first player can guarantee a win under the stated modification, and prove your answer. Here's the variant of the game: Let $k \geq 1$ be any integer. As in the original game, the player who takes the last card wins—but each player is now allowed to take *any number of cards between* 1 *and* k in any single move. (The original game was Take-2.) Your answer can depend both on k and the starting number n of cards.

5.54 Repeat for the "Don't Go Last" variant: As in the original game, players can take only 1 or 2 cards per turn—but the player who takes the last card *loses* (instead of winning by managing to take the last card).

5.55 Prove by strong induction on n that, for every integer $n \geq 4$, it is possible to make n dollars using only two- and five-dollar bills. (That is, prove that any integer $n \geq 4$ can be written as $n = 2a + 5b$ for some integer $a \geq 0$ and some integer $b \geq 0$.)

5.56 Consider a sport in which teams can score two types of goals, worth either 3 points or 7 points. For example, Team Vikings might (theoretically speaking) score 32 points by accumulating, in succession, 3, 7, 3, 7, 3, 3, 3, and 3 points. Find the smallest possible n_0 such that, for any $n \geq n_0$, a team can score exactly n points in a game. Prove your answer correct by strong induction.

Define the Fibonacci numbers *by the sequence $f_1 = 1$, $f_2 = 1$, and $f_n = f_{n-1} + f_{n-2}$ for $n \geq 3$. Thus the first several Fibonacci numbers are* $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$*. (The Fibonacci numbers are named after Leonardo of Pisa, also sometimes known as Leonardo Bonacci or just as Fibonacci, a 13th-century Italian mathematician. We'll see a lot more about the Fibonacci numbers in Section 6.4.) Prove each of the following statements by induction—weak or strong, as appropriate—on n:*

5.57 $f_n \bmod 2 = 0$ if and only if $n \bmod 3 = 0$. (That is, every third Fibonacci number is even.)

5.58 $f_n \bmod 3 = 0$ if and only if $n \bmod 4 = 0$.

5.59 $\displaystyle\sum_{i=1}^{n} f_i = f_{n+2} - 1$

5.60 $\displaystyle\sum_{i=1}^{n} (f_i)^2 = f_n \cdot f_{n+1}$

5.61 Prove *Cassini's identity:* $f_{n-1} \cdot f_{n+1} - (f_n)^2 = (-1)^n$ for any $n \geq 2$.

5.62 For a k-by-k matrix M, the matrix M^n is also k-by-k, and its value is the result of multiplying M by itself n times: $MM \cdots M$. We can define matrix exponentiation recursively: $M^0 = I$ (the k-by-k identity matrix), and $M^{n+1} = M \cdot M^n$. Using this recursive definition, prove the following identity concerning the Fibonacci numbers:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} f_n \\ f_{n-1} \end{bmatrix} \quad \text{for any } n \geq 2.$$

You may use the *associativity* of matrix multiplication in your answer: for any matrices A, B, and C of the appropriate dimensions, we have $A(BC) = (AB)C$.

Define the Lucas numbers *as $L_1 = 1$, $L_2 = 3$, and $L_n = L_{n-1} + L_{n-2}$ for $n \geq 3$. (The Lucas numbers are a less famous cousin of the Fibonacci numbers; they follow the same recursive definition as the Fibonaccis, but starting from a different pair of base cases. They're named after Édouard Lucas, a 19th-century French mathematician.) Prove the following facts about the Lucas numbers, by induction (weak or strong, as appropriate) on n:*

5.63 $L_n = f_n + 2f_{n-1}$

5.64 $f_n = \frac{L_{n-1} + L_{n+1}}{5}$

5.65 $(L_n)^2 = 5(f_n)^2 + 4(-1)^n$ *(Hint: you may need to conjecture a second property relating Lucas and Fibonacci numbers to complete the proof of the given property $P(n)$—specifically, try to formulate a property $Q(n)$ relating L_nL_{n-1} and f_nf_{n-1}, and prove $P(n) \wedge Q(n)$ with a single proof by strong induction.)*

Define the Jacobsthal numbers *as $J_1 = 1$, $J_2 = 1$, and $J_n = J_{n-1} + 2J_{n-2}$ for $n \geq 3$. (Thus the Jacobsthal numbers are a more distant relative of the Fibonacci numbers: they have the same base case, but a different recursive definition. They're named after*

Ernst Jacobsthal, a 20th-century German mathematician.) Prove the following facts about the Jacobsthal numbers by induction (weak or strong, as appropriate) on n:

5.66 $J_n = 2J_{n-1} + (-1)^{n-1}$, for all $n \geq 2$.

5.67 $J_n = \frac{2^n - (-1)^n}{3}$

5.68 $J_n = 2^{n-1} - J_{n-1}$, for all $n \geq 2$.

The next two problems are previews of Chapter 9, where we'll talk about how to count *the size of sets (often, sets that are described in somewhat complicated ways). You should be able to attack these problems without the detailed results from Chapter 9, but feel free to glance ahead to Section 9.2 if you'd like.*

5.69 You are given a 2-by-n grid that you must tile, using either 1-by-2 *dominoes* or 2-by-2 *squares*. The dominoes can be arranged either vertically or horizontally. (See Figure 5.26.) Prove by strong induction on n that the number of different ways of tiling the 2-by-n grid is precisely J_{n+1}. (Be careful: it's easy to accidentally count some configurations twice—for example, make sure that you count only once the tiling of a 2-by-3 grid that uses three horizontal dominoes.)

5.70 Suppose that you run out of squares, so you can now only use dominoes for tiling. (See Figure 5.26a.) How does your answer to the previous exercise change? How many different tilings of a 2-by-n grid are there now? Prove your answer.

The Fibonacci word fractal *defines a sequence of bitstrings using a similar recursive description to the Fibonacci numbers. See Figure 5.27 for the definition and a few examples. It turns out that if we delete the last two bits from s_n, the resulting string is a palindrome (reading the same back-to-front and front-to-back).*

5.71 Prove by strong induction that the number of bits in s_n is precisely f_n (the nth Fibonacci number).

5.72 Prove by strong induction that the string s_n does not contain two consecutive 1s or three consecutive 0s.

5.73 Let $\#0(x)$ and $\#1(x)$ denote the number of 0s and 1s in a bitstring x, respectively. Show that, for all $n \geq 3$, the amount by which the number of zeroes exceeds the number of ones—that is, the quantity $\#0(s_n) - \#1(s_n)$—is a Fibonacci number.

5.74 *(programming required.)* The reason that s_n is called a *fractal* is that it's possible to visualize these "words" (strings) as a geometric fractal by interpreting 0s and 1s as "turn" and "go straight," respectively. Specifically, here's the algorithm: start pointing east. For the ith symbol in s_n, for $i = 1, 2, \ldots, |s_n|$: if the symbol is 1 then do not turn; if the symbol is a 0 and i is even, turn 90° to the right; and if the symbol is a 0 and i is odd, turn 90° to the left. In any case, proceed in your current direction by one unit. (See Figure 5.27.) Write a program to draw a bitstring using these rules. Then implement the definition of the Fibonacci word fractal and "draw" the strings $s_1, s_2, \ldots, s_{16}$. (For efficiency's sake, you may want to compute s_n with a loop instead of recursively; see Figure 6.33 in Chapter 6 for some ideas.)

5.75 The sum of the interior angles of any triangle is 180°. Now, using this fact and induction, prove that any polygon with $k \geq 3$ vertices has interior angles that sum to $180k - 360$ degrees. (See Figure 5.28.)

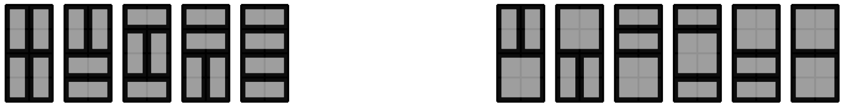
5.76 A *diagonal* of a polygon is a line that connects two non-adjacent vertices. (Again, see Figure 5.28.) How many diagonals are there in a triangle? A quadrilateral? A pentagon? Formulate a conjecture for the number $d(k)$ of diagonals in a k-gon, and prove your formula correct by induction. *(Hint: consider lopping off a triangle from the polygon.)*

5.77 Prove that the recursive binary search algorithm shown in Figure 5.29 is correct. That is, prove the following condition, by strong induction on n: *For any sorted array $A[1 \ldots n]$,* **binarySearch**(A, x) *returns true if and only if $x \in A$.*

5.78 Prove by weak induction on the total length of the inputs (that is, on the quantity $n + m$) that the **merge** algorithm in Figure 5.29 satisfies the following property for any $n \geq 0$ and $m \geq 0$: given any two sorted arrays $X[1 \ldots n]$ and $Y[1 \ldots m]$ as input, the output of **merge**(X, Y) is a sorted array containing all elements of X and all elements of Y.

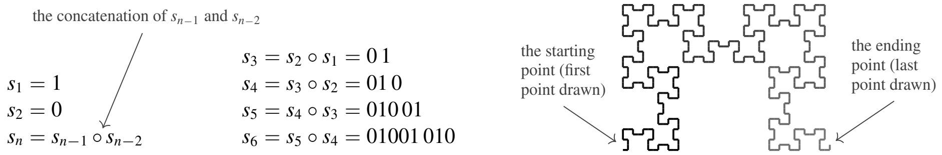
5.79 Prove by strong induction on n that **mergeSort**$(A[1 \ldots n])$, shown in Figure 5.29, indeed sorts its input.

5.80 Give a recursive algorithm to compute a list of all permutations of a given set S. (That is, compute a list of all possible orderings of the elements of S. For example, **permutations**$(\{1, 2, 3\})$ should return $\langle 1, 2, 3 \rangle$, $\langle 1, 3, 2 \rangle$, $\langle 2, 1, 3 \rangle$, $\langle 2, 3, 1 \rangle$, $\langle 3, 1, 2 \rangle$, and $\langle 3, 2, 1 \rangle$, in some order.) Prove your algorithm correct by induction.



(a) The five ways to tile using dominoes ($n = 4$). (b) The six additional tilings when we also allow squares.

Figure 5.26 How many ways are there to tile an 2-by-n grid using 1-by-2 *dominoes* and 2-by-2 *squares?*



the concatenation of s_{n-1} and s_{n-2}

$$s_1 = 1$$
$$s_2 = 0$$
$$s_n = s_{n-1} \circ s_{n-2}$$

(a) The definition.

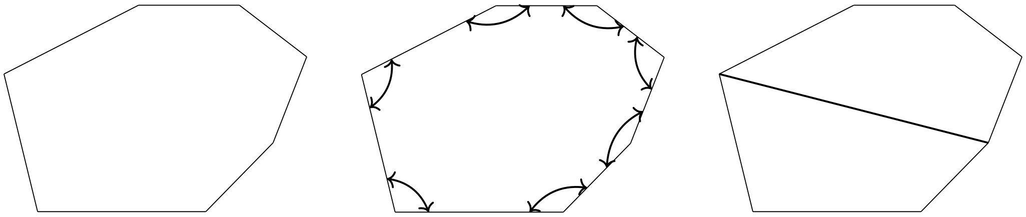
$$s_3 = s_2 \circ s_1 = 0\,1$$
$$s_4 = s_3 \circ s_2 = 01\,0$$
$$s_5 = s_4 \circ s_3 = 010\,01$$
$$s_6 = s_5 \circ s_4 = 01001\,010$$

(b) Some examples.

the starting point (first point drawn)
the ending point (last point drawn)

(c) A visualization of s_{14}, as in Exercise 5.74.

Figure 5.27 The Fibonacci word fractal, which defines a sequence of bitstrings using recursion.



(a) A polygon. (b) The interior angles. (c) One of the diagonals.

Figure 5.28 A polygon, its interior angles, and one of its diagonals.

Prove that weak induction, as defined in Section 5.2, and strong induction are equivalent. (Hint: in one of these two exercises, you will have to use a different predicate than P.)

5.81 Suppose that you've written a proof of $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$ by weak induction. I'm in an evil mood, and I declare that you aren't allowed to prove anything by weak induction. Explain how to adapt your weak-induction proof to prove $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$ using strong induction.

5.82 Now suppose that, obeying my new Draconian rules, you have written a proof of $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$ by *strong* induction. In a doubly evil mood, I tell you that now you can only use weak induction to prove things. Explain how to adapt your strong-induction proof to prove $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$ using weak induction.

```
binarySearch($A[1 \ldots n], x$):
1 if $n \leq 0$ then
2     return False
3 $middle := \lfloor \frac{1+n}{2} \rfloor$
4 if $A[middle] = x$ then
5     return True
6 else if $A[middle] > x$ then
7     return binarySearch($A[1 \ldots middle - 1], x$)
8 else
9     return binarySearch($A[middle + 1 \ldots n], x$)
```

Note: for the sake of efficiency, it's important to make sure that Line 7 and Line 9 do not copy half of the array *A*. (As written, they appear to.) Usually, you'd accomplish these steps efficiently by maintaining one array, and passing *indices* into that array (rather than the array itself) as parameters to **binarySearch**. Writing it out with the extra parameters makes this algorithm a lot harder to read, so this pseudocode errs on the side of readability at the cost of leaving out an important implementation detail.

```
merge($X[1 \ldots n], Y[1 \ldots m]$):
if $n = 0$ then
    return $Y$
else if $m = 0$ then
    return $X$
else if $X[1] < Y[1]$ then
    return $X[1]$ followed by merge($X[2 \ldots n], Y$)
else
    return $Y[1]$ followed by merge($X, Y[2 \ldots m]$)
```

```
mergeSort($A[1 \ldots n]$):
if $n = 1$ then
    return $A$
else
    $L :=$ mergeSort($A[1 \ldots \lfloor \frac{n}{2} \rfloor]$)
    $R :=$ mergeSort($A[\lfloor \frac{n}{2} \rfloor + 1 \ldots n]$)
    return merge($L, R$)
```

Figure 5.29 Binary Search, Merge, and Merge Sort, recursively.

5.4 Recursively Defined Structures and Structural Induction

The vermin only teaze and pinch
Their foes superior by an inch.
So, naturalists observe, a flea
Has smaller fleas that on him prey;
And these have smaller still to bite 'em,
And so proceed *ad infinitum.*
Thus every poet, in his kind,
Is bit by him that comes behind.

Jonathan Swift (1667–1745)
"On Poetry: a Rhapsody" (1733)

In the proofs that we have written so far in this chapter, we have performed induction on an *integer*: the number that's the input to an algorithm, the number of vertices of a polygon, the number of elements in an array. In this section, we will address proofs about *recursively defined structures*, instead of about integers, using a version of induction called *structural induction* that proceeds over the defined structure itself, rather than just using numbers.

5.4.1 Recursively Defined Structures

A recursively defined structure, just like a recursive algorithm, is a structure defined in terms of one or more *base cases* and one or more *inductive cases*. Any data type that can be understood as either a trivial instance of the type or as being built up from a smaller instance (or smaller instances) of that type can be expressed in this way. For example, basic data structures like a *linked list* and a *binary tree* can be defined recursively. So too can well-formed sentences of a formal language—languages like Python, or propositional logic—among many other examples. In this section, we'll give recursive definitions for some of these examples.

Linked lists

A *linked list* is a commonly used data structure in which we store a sequence of elements (just like the sequences from Section 2.4). The reasons that linked lists are useful are best left to a data structures course, but here is a brief synopsis of what a linked list actually is. Each element in the list, called a *node*, stores a data value and a "pointer" to the rest of the list. A special value, often called `null`, represents the empty list; the last node in the list stores this value as its pointer to represent that there are no further elements in the list. See Figure 5.30a for an example. Here is a recursive definition of a linked list:

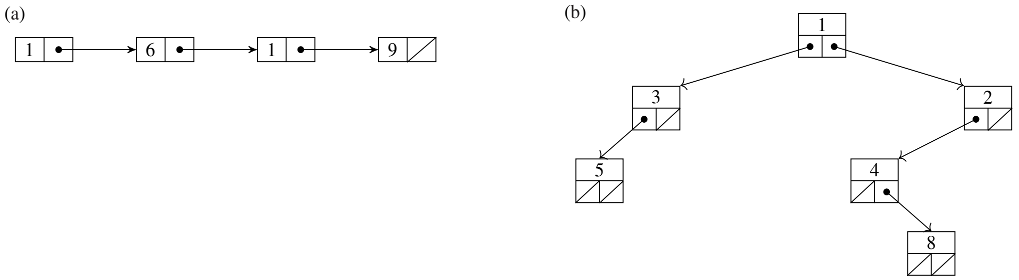


Figure 5.30 An example of (a) a linked list and (b) a binary tree. The slashed line represents the `null` value.

Example 5.15: Linked list.
A *linked list* is either:

1 $\langle\rangle$, known as the *empty list*; or

2 $\langle x, L\rangle$, where x is an arbitrary element and L is a linked list.

For example, Figure 5.30a shows the linked list that consists of 1 followed by the linked list containing 6, 1, and 9 (which is a linked list consisting of 6 followed by a linked list containing 1 and 9, which is …). In other words, Figure 5.30a shows the linked list $\langle 1, \langle 6, \langle 1, \langle 9, \langle\rangle\rangle\rangle\rangle\rangle$.

Binary trees

We can also recursively define a *binary tree* (see Section 11.4.2). We'll again defer the discussion of why binary trees are useful, but here is a quick summary of what they are. Like a linked list, a binary tree is a collection of nodes that store data values and "pointers" to other nodes. Unlike a linked list, a node in a binary tree stores *two* pointers to other nodes (or `null`, representing an empty binary tree). These two pointers are to the *left child* and *right child* of the node. The *root* node is the one at the very top of the tree. See Figure 5.30b for an example; the root node of this tree stores the value 1, and has a left child (the binary tree with root 3) and a right child (the binary tree with root 2). Here is a recursive definition:

Example 5.16: Binary trees.
A *binary tree* is either:

1 the empty tree, denoted by `null`; or

2 a *root node* x, a *left subtree* T_ℓ, and a *right subtree* T_r, where x is an arbitrary value and T_ℓ and T_r are both binary trees.

Taking it further: In many programming languages, we can explicitly define data types that echo these recursive definitions, where the base case is a trivial instance of the data structure (often `nil` or `None` or `null`). In C, for example, we can define a binary tree with integer-valued nodes as:

```
struct binaryTree {
  int root;
  struct binaryTree *leftSubtree;
  struct binaryTree *rightSubtree;
}
```

In C, the symbol $*$ means that we're storing a *reference*, or *pointer*, to the subtrees, rather than the subtrees themselves, in the data structure.

The base case—an empty binary tree—is `NULL`; the inductive case—a binary tree with a root node—has a value stored as its root, and then two binary trees (possibly empty) as its left and right subtrees.

Define the *leaves* of a binary tree T to be those nodes contained in T whose left subtree and right subtree are both `null`. Define the *internal nodes* of T to be all nodes that are not leaves. In Figure 5.30b, for example, the leaves are the nodes 5 and 8, and the internal nodes are $\{1, 2, 3, 4\}$.

Taking it further: Binary trees with certain additional properties turn out to be very useful ways of organizing data for efficient access. For example, a *binary search tree* is a binary tree in which each node stores a "key," and the tree is organized so that, for any node u, the key at node u is larger than all the keys in u's left subtree and smaller than all the keys in u's right subtree. (For example, we might store the email address of a student as a key; the tree is then organized alphabetically.) Another special type of a binary search tree is a *heap*, in which each node's key is larger than all the keys in its subtrees. These two data structures are very useful in making certain common operations very efficient; see p. 5-38 (for heaps) and p. 11-73 (for binary search trees).

Sentences in a language

In addition to data structures, we can also define *sentences* in a language using a recursive definition—for example, arithmetic expressions of the type that are understood by a simple calculator; or propositions (as in Chapter 3's propositional logic):

Example 5.17: Arithmetic expressions.
An *arithmetic expression* is any of the following:

1 any integer n;
2 $-E$, where E is an arithmetic expression; or
3 $E + F$, $E - F$, $E \cdot F$, or E/F, where E and F are arithmetic expressions.

Example 5.18: Sentences of propositional logic.
A *sentence of propositional logic* (also known as a *well-formed formula*, or *wff*) over the propositional variables X is one of the following:

1 x, for some $x \in X$;
2 $\neg P$, where P is a wff over X; or
3 $P \vee Q$, $P \wedge Q$, or $P \Rightarrow Q$, where P and Q are wffs over X.

We implicitly used the recursive definition of logical propositions from Example 5.18 throughout Chapter 3, but using this recursive definition explicitly allows us to express a number of concepts more concisely. For example, consider a truth assignment $f : X \to \{\text{True}, \text{False}\}$ that assigns True or False to each variable in

X. Then the truth value of a proposition over X under the truth assignment f can be defined recursively for each case of the definition: (1) the truth value of $x \in X$ under f is $f(x)$; (2) the truth value of $\neg P$ under f is True if the truth value of P under f is False, and the truth value of $\neg P$ under f is False if the truth value of P under f is True; and so forth.

Taking it further: Linguists interested in syntax spend a lot of energy constructing recursive definitions (like those in Examples 5.17 and 5.18) of grammatical sentences of English. But one can also give a recursive definition for non-natural languages: in fact, another structure that can be defined recursively is *the grammar of a programming language itself.* As such, this type of recursive approach to defining (and processing) a grammar plays a key role not just in linguistics but also in computer science. See the discussion on p. 5-56 for more.

5.4.2 Structural Induction

The recursively defined structures from Section 5.4.1 are particularly amenable to inductive proofs. For example, recall from Example 5.16 that a binary tree is one of the following: (1) the empty tree, denoted by `null`; or (2) a *root node* x, a *left subtree* T_ℓ, and a *right subtree* T_r, where T_ℓ and T_r are both binary trees. To prove that some property P is true of all binary trees T, we can use (strong) induction on the number n of applications of rule #2 from the definition. Here is an example of such a proof:

Example 5.19: Internal nodes vs. leaves in binary trees.
Recall that a *leaf* in a binary tree is a node whose left and right subtrees are both empty; an *internal node* is any non-leaf node. Write $leaves(T)$ and $internals(T)$ to denote the number of leaves and internal nodes in a binary tree T, respectively.

Claim: In any binary tree T, we have $leaves(T) \leq internals(T) + 1$.

Proof. We proceed by strong induction on the number of applications of rule #2 used to generate T. Specifically, let $P(n)$ denote the property that $leaves(T) \leq internals(T) + 1$ holds *for any binary tree T generated by n applications of rule #2*; we'll prove that $P(n)$ holds for all $n \geq 0$, which proves the claim.

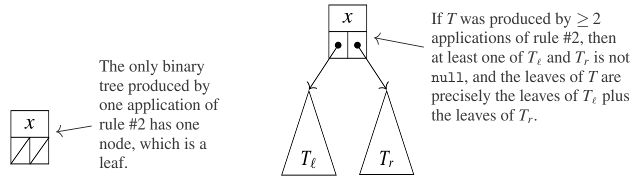
Base case ($n = 0$). The only binary tree generated with 0 applications of rule #2 is the empty tree `null`. Indeed, $leaves(\texttt{null}) = internals(\texttt{null}) = 0$, and $0 \leq 0 + 1$.

Definition 5.16, in brief:

Rule #1: `null` is a binary tree.

Rule #2: if T_ℓ and T_r are binary trees, then $\langle x, T_\ell, T_r \rangle$ is a binary tree.

(a) What binary trees are.



The only binary tree produced by one application of rule #2 has one node, which is a leaf.

If T was produced by ≥ 2 applications of rule #2, then at least one of T_ℓ and T_r is not `null`, and the leaves of T are precisely the leaves of T_ℓ plus the leaves of T_r.

(b) One application of rule #2.

(c) Two or more applications of rule #2.

Figure 5.31 Illustrations of the inductive case for Example 5.19.

Inductive case ($n \geq 1$). Assume the inductive hypothesis $P(0) \wedge P(1) \wedge \cdots \wedge P(n-1)$: for any binary tree B generated using $k < n$ applications of rule #2, we have $\mathit{leaves}(B) \leq \mathit{internals}(B) + 1$. We must prove $P(n)$.

We'll handle the case $n = 1$ separately. (See Figure 5.31b.) The only way to make a binary tree T using one application of rule #2 is to use rule #1 for both of T's subtrees, so T must contain only one node (which is itself a leaf). Then T contains 1 leaf and 0 internal nodes, and indeed $1 \leq 0 + 1$.

Otherwise $n \geq 2$. (See Figure 5.31c.) Observe that the tree T must have been generated by (a) generating a left subtree T_ℓ using some number ℓ of applications of rule #2; (b) generating a right subtree T_r using some number r of applications of rule #2; and then (c) applying rule #2 to a root node x, T_ℓ, and T_r to produce T. Therefore $r + \ell + 1 = n$, and therefore $r < n$ and $\ell < n$. Ergo, we can apply the inductive hypothesis to both T_ℓ and T_r, and thus

$$\mathit{leaves}(T_\ell) \leq \mathit{internals}(T_\ell) + 1 \tag{1}$$

$$\mathit{leaves}(T_r) \leq \mathit{internals}(T_r) + 1. \tag{2}$$

Also observe that, because $r + \ell + 1 = n \geq 2$, either $T_r \neq \texttt{null}$ or $T_\ell \neq \texttt{null}$, or both. Thus the leaves of T are the leaves of T_ℓ and T_r, and internal nodes of T are the internal nodes of T_ℓ and T_r plus the root x (which cannot be a leaf because at least one of T_ℓ and T_r is not empty). Therefore

$$\mathit{leaves}(T) = \mathit{leaves}(T_\ell) + \mathit{leaves}(T_r) \tag{3}$$

$$\mathit{internals}(T) = \mathit{internals}(T_\ell) + \mathit{internals}(T_r) + 1. \tag{4}$$

Putting together these facts, we have

$$\begin{aligned}
\mathit{leaves}(T) &= \mathit{leaves}(T_\ell) + \mathit{leaves}(T_r) && \textit{by (3)} \\
&\leq \mathit{internals}(T_\ell) + 1 + \mathit{internals}(T_r) + 1 && \textit{by (1) and (2)} \\
&= \mathit{internals}(T) + 1. && \textit{by (4)}
\end{aligned}$$

Thus $P(n)$ holds, which completes the proof. □

Structural induction: the idea

The proof in Example 5.19 is perfectly legitimate, but there is another approach that we can use for recursively defined structures, called *structural induction*. The basic idea is to perform induction *on the structure of an object itself* rather than on some integer: instead of a case for $n = 0$ and a case for $n \geq 1$, in a proof by structural induction our cases correspond directly to the cases of the recursive structural definition.

For structural induction to make sense, we must impose some restrictions on the recursive definition. Specifically, the set of structures defined must be *well ordered,* which intuitively ensures that every invocation of the inductive case of the definition "makes progress" toward the base case(s) of the definition. For the type of recursive definitions that we're considering—where there are base cases in the definition, and all instances of the structure are produced by a finite sequence of applications of the inductive rules in the definition—structural induction is a valid technique to prove facts about the recursively defined structure.

> **Taking it further:** A set of objects is called well ordered if there's a "least" element among any collection of those objects. More formally, a set S of structures is *well ordered* if there exists a "smaller than" relationship $\prec$ between elements of S such that, for any nonempty $T \subseteq S$, there exists a *minimal element* m in T—that is, there exists $m \in T$ such that no $x \in T$ satisfies $x \prec m$. (There might be more than one least element in T.) For example, the set $\mathbb{Z}^{\geq 0}$ is well ordered, using the normal $\leq$ relationship. However, the set $\mathbb{R}$ is not well ordered: for example, the set $\{x \in \mathbb{R} : x > 2\}$ has no smallest element using $\leq$. But the set of binary trees *is* well ordered; the relation $\prec$ is "is a subtree of." One can prove that a set S is well ordered if and only if a proof by mathematical induction is valid on a set S (where the base cases are the minimal elements of S, and to prove $P(x)$ we assume the inductive hypotheses $P(y)$ for any $y \prec x$).

Proofs by structural induction

Here is the formal definition of a proof by structural induction:

Definition 5.12: Proof by structural induction.
Suppose that we want to prove that $P(x)$ holds for every $x \in S$, where S is the (well-ordered) set of structures generated by a recursive definition, and P is some property. To give a *proof by structural induction* of $\forall x \in S : P(x)$, we prove the following:

1. *Base cases*: for every x defined by a base case in the definition of S, prove $P(x)$.
2. *Inductive cases*: for every x defined in terms of $y_1, y_2, \ldots, y_k \in S$ by an inductive case in the definition of S, prove that $P(y_1) \wedge P(y_2) \wedge \cdots \wedge P(y_k) \Rightarrow P(x)$.

In a proof by structural induction, we can view both base cases and inductive cases in the same light: each case assumes that the recursively constructed subpieces of a structure x satisfy the stated property, and we prove that x itself also satisfies the property. For a base case, the point is just that there *are no* recursively constructed pieces, so we actually are not making any assumption.

Notice that a proof by structural induction is identical in form to a proof by strong induction *on the number of applications of the inductive-case rules used to generate the object.* For example, we can immediately rephrase the proof in Example 5.19 to use structural induction instead. While the structure of the proof is identical, structural induction can streamline the proof and make it easier to read:

Example 5.20: Internal nodes vs. leaves in binary trees, take II.

Claim: In any binary tree T, we have $\textit{leaves}(T) \leq \textit{internals}(T) + 1$.

Proof. Let $P(T)$ denote the property that $\textit{leaves}(T) \leq \textit{internals}(T) + 1$ for a binary tree T. We proceed by structural induction on the form of T.

Base case ($T = \texttt{null}$). Then $\textit{leaves}(T) = \textit{internals}(T) = 0$, and indeed $0 \leq 0 + 1$.

Inductive case (T has root x, left subtree T_ℓ, and right subtree T_r). We assume the inductive hypotheses $P(T_\ell)$ and $P(T_r)$, namely

$$leaves(T_\ell) \le internals(T_\ell) + 1 \quad (1)$$
$$leaves(T_r) \le internals(T_r) + 1. \quad (2)$$

- If x is itself a leaf, then $T_\ell = T_r = \texttt{null}$, and therefore $leaves(T) = 1$ and $internals(T) = 0$, and indeed $1 \le 0 + 1$.
- Otherwise x is not a leaf, and either $T_r \neq \texttt{null}$ or $T_\ell \neq \texttt{null}$, or both. Thus the leaves of T are the leaves of T_ℓ and T_r, and internal nodes of T are the internal nodes of T_ℓ and T_r plus the root x. Therefore

$$leaves(T) = leaves(T_\ell) + leaves(T_r) \quad (3)$$
$$internals(T) = internals(T_\ell) + internals(T_r) + 1. \quad (4)$$

Putting together these facts, we have

$$\begin{aligned} leaves(T) &= leaves(T_\ell) + leaves(T_r) && \text{by (3)} \\ &\le internals(T_\ell) + 1 + internals(T_r) + 1 && \text{by (1) and (2)} \\ &= internals(T) + 1. && \text{by (4)} \end{aligned}$$

Thus $P(n)$ holds, which completes the proof. □

5.4.3 Some More Examples of Structural Induction: Propositional Logic

We'll finish this section with two more proofs by structural induction, about propositional logic—using Example 5.18's recursive definition.

Propositional logic using only ¬ and ∧

First, we'll give a formal proof using structural induction of the claim that any propositional logic statement can be expressed using ¬ and ∧ as the only logical connectives. (See Exercise 4.68.)

Example 5.21: All of propositional logic using ¬ and ∧.

Claim: For any logical proposition φ using the connectives $\{\neg, \wedge, \vee, \Rightarrow\}$, there exists a proposition using only $\{\neg, \wedge\}$ that is logically equivalent to φ.

Proof. For a logical proposition φ, let $A(\varphi)$ denote the property that there exists a $\{\neg, \wedge\}$-only proposition logically equivalent to φ. We'll prove by structural induction on φ that $A(\varphi)$ holds for any well-formed formula φ (see Example 5.18):

Base case: φ is a variable, say $\varphi = x$. The proposition x uses no connectives—and thus is vacuously $\{\neg, \wedge\}$-only—and is obviously logically equivalent to itself. Thus $A(x)$ follows.

Inductive case I: φ is a negation, say $\varphi = \neg P$. We assume the inductive hypothesis $A(P)$. We must prove $A(\neg P)$. By the inductive hypothesis, there is a $\{\neg, \wedge\}$-only proposition Q such that $Q \equiv P$. Consider the proposition $\neg Q$. Because $Q \equiv P$, we have that $\neg Q \equiv \neg P$, and $\neg Q$ contains only the connectives $\{\neg, \wedge\}$. Thus $\neg Q$ is a $\{\neg, \wedge\}$-only proposition logically equivalent to $\neg P$. Thus $A(\neg P)$ follows.

Inductive case II: φ is a conjunction, disjunction, or implication, say $\varphi = P_1 \wedge P_2$, $\varphi = P_1 \vee P_2$, or $\varphi = P_1 \Rightarrow P_2$. We assume the inductive hypotheses $A(P_1)$ and $A(P_2)$—that is, we assume there are $\{\neg, \wedge\}$-only propositions Q_1 and Q_2 with $Q_1 \equiv P_1$ and $Q_2 \equiv P_2$. We must prove $A(P_1 \wedge P_2)$, $A(P_1 \vee P_2)$, and $A(P_1 \Rightarrow P_2)$. Consider the propositions $Q_1 \wedge Q_2$, $\neg(\neg Q_1 \wedge \neg Q_2)$, and $\neg(Q_1 \wedge \neg Q_2)$. By De Morgan's Law, and the facts that $x \Rightarrow y \equiv \neg(x \wedge \neg y)$, $P_1 \equiv Q_1$, and $P_2 \equiv Q_2$:

$$\begin{array}{lclcl} Q_1 \wedge Q_2 & \equiv & Q_1 \wedge Q_2 & \equiv & P_1 \wedge P_2 \\ \neg(\neg Q_1 \wedge \neg Q_2) & \equiv & Q_1 \vee Q_2 & \equiv & P_1 \vee P_2 \\ \neg(Q_1 \wedge \neg Q_2) & \equiv & Q_1 \Rightarrow Q_2 & \equiv & P_1 \Rightarrow P_2 \end{array}$$

Because Q_1 and Q_2 are $\{\neg, \wedge\}$-only, our three propositions are $\{\neg, \wedge\}$-only as well; therefore $A(P_1 \wedge P_2)$, $A(P_1 \vee P_2)$, and $A(P_1 \Rightarrow P_2)$ follow.

We've shown that $A(\varphi)$ holds for any proposition φ, so the claim follows. □

Taking it further: In the programming language ML, among others, a programmer can use both recursive definitions *and* a form of recursion that mimics structural induction. For example, we can give a simple implementation of the recursive definition of a well-formed formula from Example 5.18:

```
datatype wff = Variable of string
             | Not of wff
             | And of (wff * wff)
             | Or of (wff * wff)
             | Implies of (wff * wff);
```

"A well-formed formula is a variable (whose name is a `string`), or the negation of a well-formed formula, or the conjunction of a pair of well-formed formulas (`wff * wff`), or . . . .")

In ML, we can also write a function that mimics the structure of the proof in Example 5.21, using ML's capability of *pattern matching* function arguments. Here is an implementation of the recursive function that takes an arbitrary `wff` as input, and produces an equivalent `wff` using only `And` and `Not` as output.

```
fun simplify (Variable var)     =  Variable var
  | simplify (Not P)            =  Not(simplify P)
  | simplify (And (P1, P2))     =  And(simplify P1, simplify P2)
  | simplify (Or (P1, P2))      =  Not(And(Not(simplify P1), Not(simplify P2)))
  | simplify (Implies (P1, P2)) =  Not(And(simplify P1, Not(simplify P2)));
```

Conjunctive and Disjunctive Normal Forms

Here is another example of a proof by structural induction based on propositional logic, to establish Theorems 3.18 and 3.19, that any proposition is logically equivalent to one that's in conjunctive or disjunctive normal form. (Recall that 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 *literal* is a Boolean variable or the negation of a Boolean variable. 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.)

Theorem 5.13: CNF/DNF suffice.
Let φ be a Boolean formula that uses the connectives $\{\wedge, \vee, \neg, \Rightarrow\}$. Then:

1 there exists φ_{dnf} in disjunctive normal form so that φ and φ_{dnf} are logically equivalent.

2 there exists φ_{cnf} in conjunctive normal form so that φ and φ_{cnf} are logically equivalent.

Perhaps bizarrely, it will turn out to be easier to prove that any proposition is logically equivalent to *both* one in CNF *and* one in DNF than to prove either claim on its own. So we will prove both parts of the theorem simultaneously, by structural induction.

Problem-solving tip: Suppose we want to prove $\forall x : P(x)$ by induction. Here's a problem-solving strategy that's highly counterintuitive: it is sometimes easier to prove a *stronger* statement $\forall x : P(x) \wedge Q(x)$. It seems bizarre that trying to prove *more* than what we want is easier—but the advantage arises because the inductive hypothesis is a more powerful assumption! For example, I don't know how to prove that any proposition φ can be expressed in DNF (Theorem 5.13.1) by induction! But I do know how to prove that any proposition φ can be expressed in *both DNF and CNF* by induction, as is done in Example 5.22.

We'll make use of some handy notation in this proof: analogous to summation and product notation, we write $\bigwedge_{i=1}^{n} p_i$ to denote $p_1 \wedge p_2 \wedge \cdots \wedge p_n$, and similarly $\bigvee_{i=1}^{n} p_i$ means $p_1 \vee p_2 \vee \cdots \vee p_n$. Here is the proof:

Example 5.22: Conjunctive/disjunctive normal form.
Proof. We start by simplifying the task: we use Example 5.21 to ensure that φ contains only the connectives $\{\neg, \wedge\}$. Let $C(\varphi)$ and $D(\varphi)$, respectively, denote the property that φ is logically equivalent to a CNF proposition and a DNF proposition, respectively. We now proceed by structural induction on the form of φ—which now can only be a variable, negation, or conjunction—to show that $C(\varphi) \wedge D(\varphi)$ holds for any proposition φ.

Base case: φ is a variable, say $\varphi = x$. We're done immediately; a single variable is actually in both CNF and DNF. We simply choose $\varphi_{\text{dnf}} = \varphi_{\text{cnf}} = x$. Thus $C(x)$ and $D(x)$ follow immediately.

Inductive case I: φ is a negation, say $\varphi = \neg P$. We assume the inductive hypothesis $C(P) \wedge D(P)$—that is, we assume that there are propositions P_{cnf} and P_{dnf} such that $P \equiv P_{\text{cnf}} \equiv P_{\text{dnf}}$, where P_{cnf} is in CNF and P_{dnf} is in DNF. We must show $C(\neg P)$ and $D(\neg P)$.

We'll first show $D(\neg P)$—that is, that $\neg P$ can be rewritten in DNF. By the definition of conjunctive normal form, we know that the proposition P_{cnf} is of the form $P_{\text{cnf}} = \bigwedge_{i=1}^{n} c_i$, where c_i is a clause of the form $c_i = \bigvee_{j=1}^{m_i} c_i^j$, where c_i^j is a variable or its negation. Therefore we have

$$\neg P \overset{\substack{\text{inductive hypothesis } C(P)\\ \downarrow}}{\equiv} \neg P_{\text{cnf}} \overset{\substack{\text{definition of CNF}\\ \downarrow}}{\equiv} \neg\left[\bigwedge_{i=1}^{n}\left(\bigvee_{j=1}^{m_i} c_j^i\right)\right] \overset{\substack{\text{De Morgan's Law}\\ \downarrow}}{\equiv} \left[\bigvee_{i=1}^{n} \neg\left(\bigvee_{j=1}^{m_i} c_j^i\right)\right] \overset{\substack{\text{De Morgan's Law}\\ \downarrow}}{\equiv} \left[\bigvee_{i=1}^{n}\left(\bigwedge_{j=1}^{m_i} \neg c_j^i\right)\right].$$

Once we delete double negations (that is, if $c_i^j = \neg x$, then we write $\neg c_i^j$ as x rather than as $\neg\neg x$), this last proposition is in DNF, so $D(\neg P)$ follows.

The construction to show $C(\neg P)$—that is, to give an CNF proposition logically equivalent to $\neg P$—is strictly analogous; the only change to the argument is that we start from P_{dnf} instead of P_{cnf}.

Inductive case II: φ is a conjunction, say $P \wedge Q$. We assume the inductive hypotheses $C(P) \wedge D(P)$ and $C(Q) \wedge D(Q)$—that is, we assume that there are CNF propositions P_{cnf} and Q_{cnf} and DNF propositions P_{dnf} and Q_{dnf} such that $P \equiv P_{\text{cnf}} \equiv P_{\text{dnf}}$ and $Q \equiv Q_{\text{cnf}} \equiv Q_{\text{dnf}}$. We must show $C(P \wedge Q)$ and $D(P \wedge Q)$.

The argument for $C(P \wedge Q)$ is the easier of the two: we have propositions P_{cnf} and Q_{cnf} in CNF where $P_{\text{cnf}} \equiv P$ and $Q_{\text{cnf}} \equiv Q$. Thus $P \wedge Q \equiv P_{\text{cnf}} \wedge Q_{\text{cnf}}$—and the conjunction of two CNF formulas is itself in CNF. So $C(P \wedge Q)$ follows.

We have to work a little harder to prove $D(P \wedge Q)$. Recall that, by the inductive hypothesis, there are propositions P_{dnf} and Q_{dnf} in DNF, where $P \equiv P_{\text{dnf}}$ and $Q \equiv Q_{\text{dnf}}$. By the definition of DNF, these propositions have the form $P_{\text{dnf}} = \bigvee_{i=1}^{n} c_i$ and $Q_{\text{dnf}} = \bigvee_{j=1}^{m} d_j$, where every c_i and d_j is a clause that is a conjunction of literals. Therefore

$$P \wedge Q \overset{\substack{\text{inductive hypothesis } D(P) \text{ and definition of DNF}\\ \downarrow}}{\equiv} \left(\bigvee_{i=1}^{n} c_i\right) \wedge Q \overset{\substack{\text{distributivity of } \vee \text{ over } \wedge\\ \downarrow}}{\equiv} \bigvee_{i=1}^{n} (c_i \wedge Q) \overset{\substack{\text{inductive hypothesis } D(Q) \text{ and definition of DNF}\\ \downarrow}}{\equiv} \bigvee_{i=1}^{n}\left(c_i \wedge \bigvee_{i=j}^{m} d_j\right) \overset{\substack{\text{distributivity of } \vee \text{ over } \wedge\\ \downarrow}}{\equiv} \bigvee_{i=1}^{n}\bigvee_{j=1}^{m} (c_i \wedge d_j)\,.$$

Because every c_i and d_j is a conjunction of literals, $c_i \wedge d_j$ is too, and thus this last proposition is in DNF! So $D(P \wedge Q)$ follows—as does the theorem. □

The construction for a conjunction $P \wedge Q$ in Theorem 5.22 is a little tricky, so let's illustrate it with a small example:

Example 5.23: An example of the construction from Example 5.22.

Suppose that we are trying to transform a proposition $\varphi \wedge \psi$ into DNF. Suppose that we have (recursively) computed $\varphi_{\text{dnf}} = (p \wedge t) \vee q$ and $\psi_{\text{dnf}} = r \vee (s \wedge t)$. Then the construction from Example 5.22 lets us construct a proposition equivalent to $\varphi \wedge \psi$ as:

$$\begin{aligned}\varphi \wedge \psi \equiv \varphi_{\text{dnf}} \wedge \psi_{\text{dnf}} &\equiv \Big[\underbrace{(p \wedge t)}_{c_1} \vee \underbrace{(q)}_{c_2}\Big] \wedge \Big[\underbrace{(r)}_{d_1} \vee \underbrace{(s \wedge t)}_{d_2}\Big] \\ &\equiv \Big[\underbrace{(p \wedge t)}_{c_1} \wedge \Big[\underbrace{(r) \vee (s \wedge t)}_{d_1 \vee d_2}\Big]\Big] \vee \Big[\underbrace{(q)}_{c_2} \wedge \Big[\underbrace{(r) \vee (s \wedge t)}_{d_1 \vee d_2}\Big]\Big]\end{aligned}$$

$$\equiv \Big[\underbrace{(p \wedge t \wedge r)}_{c_1 \wedge d_1} \vee \underbrace{(p \wedge t \wedge s \wedge t)}_{c_1 \wedge d_2} \Big] \vee \Big[\underbrace{(q \wedge r)}_{c_2 \wedge d_1} \vee \underbrace{(q \wedge s \wedge t)}_{c_2 \wedge d_2} \Big].$$

Then the construction yields $(p \wedge t \wedge r) \vee (p \wedge t \wedge s \wedge t) \vee (q \wedge r) \vee (q \wedge s \wedge t)$ as the DNF proposition equivalent to $\varphi \wedge \psi$.

5.4.4 The Integers, Recursively Defined

Before we end the section, we'll close our discussion of recursively defined structures and structural induction with one more potentially interesting observation. Although the basic form of induction in Section 5.2 appears fairly different, that basic form of induction can actually be seen as structural induction, too. The key is to view the nonnegative integers $\mathbb{Z}^{\geq 0}$ as defined recursively:

Definition 5.14: Nonnegative integers, recursively defined.
A *nonnegative integer* is either:

1 *zero*, denoted by 0; or

2 the *successor* of a nonnegative integer, denoted by $s(x)$ for a nonnegative integer x.

Under this definition, a proof of $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$ by structural induction and a proof of $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$ by weak induction are literally identical. First, they have precisely the same base case: prove $P(0)$. And, second, they have precisely the same inductive case: prove $P(n) \Rightarrow P(s(n))$—or, in other words, prove that $P(n) \Rightarrow P(n+1)$.

Computer Science Connections

Grammars, Parsing, and Ambiguity

In *interpreters* and *compilers*—systems that translate input source code written in a programming language like Python, Java, or C into a machine-executable format—a key initial step is to *parse* the input into a format that represents its structure. (A similar step occurs in systems designed to perform natural language processing.) The structured representation of such an expression is called a *parse tree*, in which the leaves of the tree correspond to the base cases of the recursive structural definition, and the internal nodes correspond to the inductive cases of the definition. We can then use the parse tree for whatever purpose we desire: evaluating arithmetic expressions, simplifying propositional logic, or any other manipulation. (See Figure 5.32.)

In this setting, a recursively defined structure is written as a *context-free grammar (CFG)*. A grammar consists of a set of *rules* that can be used to generate a particular example of this defined structure. (This type of grammar is called *context free* because the rules defined by the grammar can be used any time—that is, without regard to the context in which the symbol on the left-hand side of the rule appears. There are also "context-sensitive" grammars, which remove this restriction; they tend to be more complicated to think about.) We'll take the definition of propositions over the variables $\{p, q, r\}$ (Example 5.18) as a running example. Here is a CFG for propositions, following that definition precisely. (Here "→" means "can be rewritten as" and "|" means "or.")

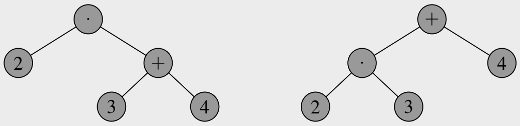


Figure 5.32 Parse trees for two arithmetic expressions: $2 \cdot (3+4)$ and $(2 \cdot 3) + 4$.

$$S \quad\to\quad \underbrace{p \;\mid\; q \;\mid\; r}_{\text{S can be a propositional variable}} \;\mid\; \underbrace{\neg S}_{\text{or S can be the negation of a proposition}} \;\mid\; \underbrace{S \vee S \;\mid\; S \wedge S \;\mid\; S \Rightarrow S}_{\text{or S can be the $\wedge$ or $\vee$ or $\Rightarrow$ of two propositions.}} .$$

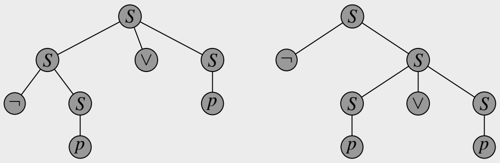
An expression is a valid proposition over the variables $\{p, q, r\}$ if and only if it can be generated by a finite sequence of applications of the rewriting rules in the grammar. For example, $\neg p \vee p$ is a valid proposition over $\{p, q, r\}$, because we can generate it as follows (the rule from the grammar that we've invoked in each step is shown above the arrow):

$$S \xrightarrow{S \to S \vee S} S \vee S \xrightarrow{S \to p} S \vee p \xrightarrow{S \to \neg S} \neg S \vee p \xrightarrow{S \to p} \neg p \vee p. \tag{1}$$

The parse tree corresponding to this sequence of rule applications is shown in Figure 5.33a. A complication that arises with the grammar given above is that it is *ambiguous*: the same proposition can be produced using a fundamentally different sequence of rule applications, which gives rise to a different parse tree:

$$S \to \neg S \to \neg S \vee S \to \neg p \vee S \to \neg p \vee p. \tag{2}$$

The parse tree corresponding to this second derivation is shown in Figure 5.33b. This second parse tree corresponds to the expression $\neg(p \vee p)$ instead of $(\neg p) \vee p$, which is the correct "order of operations" because $\neg$ binds tighter than $\vee$.



(a) The correct order of operations, corresponding to derivation (1).

(b) The wrong order of operations, corresponding to derivation (2).

Figure 5.33 Two parse trees for $\neg p \vee p$.

It's bad news if the grammar of a programming language is ambiguous, because certain valid code is then "allowed" to be interpreted in more than one way. (The classic example is the attachment of `else` clauses: in code like `if P then if Q then X else Y`, when should `Y` be executed? When `P` is true and `Q` is false? Or when `P` is false?) Thus programming language designers develop unambiguous grammars that reflect the desired behavior. (For more on context-free grammars and parsing, and the relationship between context-free grammars and compilers/interpreters, see [5, 73, 120].)

EXERCISES

Consider linked lists as defined in Example 5.15. See Figure 5.34 for some algorithms operating on linked lists.

5.83 Let L be a linked list. Prove by structural induction on L that **length**(L) returns the number of elements contained in L.

5.84 Prove by structural induction that **contains**(L, x) returns True if and only if x is one of the elements in L.

5.85 Prove by structural induction that **sum**(L) returns the sum of the elements of L. (Assume L contains only numbers.)

5.86 Here's a variant of the definition of linked lists in Example 5.15, where we insist that the elements be in increasing order. Define a *nonempty sorted list* as one of the following:

- $\langle x, \langle\rangle\rangle$; or
- $\langle x, \langle y, L\rangle\rangle$ where $x \leq y$ and $\langle y, L\rangle$ is a nonempty sorted list.

Prove by structural induction that in a nonempty sorted list $\langle x, L\rangle$, every element z in L satisfies $z \geq x$.

Intuitively, a string of parentheses is unbalanced if there's an open parenthesis that's never closed (as in `[[]`*) or if there's a close parenthesis that doesn't close anything (as in* `[]]`*). Formally, define a* string of balanced parentheses *as any of the following:*

- *the empty string (consisting of zero characters);*
- *a string* `[` S `]` *where S is a string of balanced parentheses; or*
- *a string S_1S_2 where S_1 and S_2 are both strings of balanced parentheses.*

For example, `[[]][]` *is a string of balanced parentheses, using the third rule on* `[[]]` *and* `[]`*. (And* `[]` *is a string of balanced parentheses using the second rule on the empty string. Thus* `[[]]` *is, too, using the second rule on* `[]`*.)*

5.87 Prove by structural induction that every string of balanced parentheses has exactly the same number of `[`s as it does `]`s.

5.88 Prove by structural induction that any *prefix* of a string of balanced parentheses has at least as many `[`s as it does `]`s.

5.89 Prove by structural induction that the algorithm **countLeaves**(T) in Figure 5.35 returns the number of leaves in a binary tree T.

5.90 A *binary search tree (BST)* is a binary tree in which each node stores a "key," and, for any node u, the key at node u is larger than all keys in u's left subtree and smaller than all the keys in u's right subtree. (See p. 11-73.) That is, a *BST* is either:

- an empty tree, denoted by `null`; or
- a *root node* x, a *left subtree* T_ℓ where all elements are less than x, and a *right subtree* T_r, where all elements are greater than x, and T_ℓ and T_r are both BSTs.

```
length(L):   // L is any linked list
if $L = \langle\rangle$ then
    return 0
else if $L = \langle x, L'\rangle$ then
    return $1 + \textbf{length}(L')$
```

```
contains(L, x):   // L is any linked list
if $L = \langle\rangle$ then
    return False
else if $L = \langle y, L'\rangle$ then
    return $x = y$ or $\textbf{contains}(L', x)$
```

```
sum(L):   // L contains only numbers
if $L = \langle\rangle$ then
    return 0
else if $L = \langle x, L'\rangle$ then
    return $x + \textbf{sum}(L')$
```

Figure 5.34 Three algorithms on linked lists.

```
countLeaves(T):
if $T = $ null then
    return 0
else
    $T_L, T_R :=$ the left and right subtrees of $T$
    if $T_L = T_R = $ null then
        return 1
    else
        return $\textbf{countLeaves}(T_L) + \textbf{countLeaves}(T_R)$
```

A reminder of Definition 5.16: a *binary tree* is either

- an empty tree, denoted by `null`; or
- a *root node* x, a *left subtree* T_ℓ, and a *right subtree* T_r, where x is an arbitrary value and T_ℓ and T_r are both binary trees.

(Recall that a *leaf* of a binary tree T is a node in T whose left subtree and right subtree are both `null`.)

Figure 5.35 A reminder of the definition of a binary tree, and an algorithm to count leaves in one.

Prove that the smallest element in a nonempty BST is the bottommost leftmost node—that is, prove that

$$\text{the smallest element in a BST with root } x \text{ and left subtree } T_\ell = \begin{cases} x & \text{if } T_\ell = \texttt{null} \\ \text{the smallest element in } T_\ell & \text{if } T_\ell \neq \texttt{null}. \end{cases}$$

A heap is a binary tree where each node stores a priority, *and in which every node satisfies the* heap property: *the priority of a node u must be greater than or equal to the priorities of the roots of both of u's subtrees. (The restriction only applies for a subtree that is not* `null`*.) See p. 5-38.*

5.91 Give a recursive definition of a heap.

5.92 Prove by structural induction that every heap is empty, or that no element of the heap is larger than its root node. (That is, the root is a maximum element.)

5.93 Prove by structural induction that every heap is empty, or it has a leaf u such that u is no larger than any node in the heap. (That is, the leaf u is a minimum element.)

A 2–3 tree is a data structure that is similar in spirit to a binary search tree (see Exercise 5.90—more precisely, a 2–3 tree is analogous to a balanced *form of BST, which is guaranteed to support fast operations like insertions, lookups, and deletions). The name "2–3 tree" comes from the fact that each internal node in the tree must have precisely 2 or 3 children; no node has a single child. Furthermore, all leaves in a 2–3 tree must be at the same "level" of the tree. Formally, a 2–3 tree of height h is one of the following:*

- *a single node (in which case $h = 0$, and the node is called a* leaf*); or*
- *a node with 2 subtrees, both of which are 2–3 trees of height $h - 1$; or*
- *a node with 3 subtrees, all three of which are 2–3 trees of height $h - 1$.*

5.94 Prove by structural induction that a 2–3 tree of height h has at least 2^h leaves.

5.95 Prove by structural induction that a 2–3 tree of height h has at most 3^h leaves. (This result plus Exercise 5.94 tells us that a 2–3 tree that contains n leaf nodes has height between $\log_3 n$ and $\log_2 n$.)

5.96 A 2–3–4 tree is a similar data structure to a 2–3 tree, except that a tree can be a single node or a node with 2, 3, or 4 subtrees. Give a formal recursive definition of a 2–3–4 tree, and prove that a 2–3–4 tree of height h has at least 2^h leaves and at most 4^h leaves.

The next few exercises give recursive definitions of some familiar arithmetic operations which are usually defined nonrecursively. In each, you're asked to prove a familiar property by structural induction. Think carefully when you choose the quantity upon which to perform induction, and don't skip any steps in your proof! You may use the elementary-school facts about addition and multiplication from Figure 5.36 in your proofs.

5.97 Let's define an *even number* as either (i) 0, or (ii) $2 + k$, where k is an even number. Prove by structural induction that the sum of any two even numbers is an even number.

5.98 Let's define a *power of two* as either (i) 1, or (ii) $2 \cdot k$, where k is a power of two. Prove by structural induction that the product of any two powers of two is itself a power of two.

5.99 Let $a_1, a_2, \ldots, a_k$ all be even numbers, for an arbitrary integer $k \geq 0$. Prove that $\sum_{i=1}^{k} a_i$ is also an even number. *(Hint: use weak induction and Exercise 5.97.)*

$(a + b) + c = a + (b + c)$ *Associativity of Addition*

$a + b = b + a$ *Commutativity of Addition*

$(a \cdot b) \cdot c = a \cdot (b \cdot c)$ *Associativity of Multiplication*

$a \cdot b = b \cdot a$ *Commutativity of Multiplication*

$a + 0 = 0 + a = a$ *Additive Identity*

$a \cdot 1 = 1 \cdot a = a$ *Multiplicative Identity*

$a \cdot 0 = 0 \cdot a = 0$ *Multiplicative Zero*

Figure 5.36 A few elementary-school facts about addition and multiplication.

In Chapter 2 (Definition 2.5), we defined b^n (for a base $b \in \mathbb{R}$ and an exponent $n \in \mathbb{Z}^{\geq 0}$) as denoting the result of multiplying b by itself n times. As an alternative to that definition of exponentiation, we could instead give a recursive *definition with integer exponents: $b^0 = 1$ and $b^{n+1} = b \cdot b^n$, for any nonnegative integer n.*

5.100 Using the facts in Figure 5.36, prove by induction that $b^m b^n = b^{m+n}$ for any integers $n \geq 0$ and $m \geq 0$. Don't skip any steps.

5.101 Using the facts in Figure 5.36 and Exercise 5.100, prove by induction that $(b^m)^n = b^{mn}$ for any integers $n \geq 0$ and $m \geq 0$. Again, don't skip any steps.

5.102 Suppose that we extend the well-formed formula ("wff") definition from Example 5.18 to allow another logical connective: the Sheffer stroke $|$, where $p \mid q \equiv \neg(p \wedge q)$. Give a proof using structural induction (see Example 5.21 for an example) that any wff is logically equivalent to one using Sheffer stroke as the only connective.

5.103 *(programming required.)* Write a program in the programming language ML (see p. 5-52) to translate an arbitrary statement of propositional logic into a logically equivalent statement in which $|$ is the only logical connective. (In other words, implement the proof of Exercise 5.102 as a recursive function.)

5.104 Repeat Exercise 5.102 for Peirce's arrow $\downarrow$, where $p \downarrow q \equiv \neg(p \vee q)$.

5.105 *(programming required.)* Repeat Exercise 5.103 with $\downarrow$ as the only logical connective (using Exercise 5.104).

5.106 Call a logical proposition *truth-preserving* if the proposition is true under the all-true truth assignment. (That is, a proposition is truth-preserving if and only if the first row of its truth table is True.) Prove the following claim by structural induction on the form of the proposition:

> Any logical proposition that uses only the logical connectives $\vee$ and $\wedge$ is truth-preserving.

(A solution to this exercise yields a rigorous solution to Exercise 4.71—there are propositions that cannot be expressed using only $\wedge$ and $\vee$. Explain.)

5.107 A *palindrome* is a string that reads the same front-to-back as it does back-to-front—for example, `RACECAR` or (ignoring spaces and punctuation) `A MAN, A PLAN, A CANAL---PANAMA!` or 10011001. Give a recursive definition of the set of palindromic bitstrings.

5.108 Let $\#0(s)$ and $\#1(s)$ denote the number of 0s and 1s in a bitstring s, respectively. Using structural induction and your definition from Exercise 5.107, prove that, for any palindromic bitstring s, the value of $[\#0(s)] \cdot [\#1(s)]$ is an even number.

5.5 Chapter at a Glance

Proofs by Mathematical Induction

Suppose that we want to prove that a property $P(n)$ holds for all $n \in \mathbb{Z}^{\geq 0}$. To give a *proof by mathematical induction* of the claim $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$, we prove the *base case* $P(0)$, and we prove the *inductive case*: for every $n \geq 1$, we have $P(n-1) \Rightarrow P(n)$.

When writing an inductive proof of the claim $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$, include each of the following steps:

1 A clear statement of the claim to be proven—that is, a clear definition of the property $P(n)$ that will be proven true for all $n \geq 0$—and a statement that the proof is by induction, including specifically identifying the variable n upon which induction is being performed. (Some claims involve multiple variables, and it can be confusing if you aren't clear about which is the variable upon which you are performing induction.)

2 A statement and proof of the base case—that is, a proof of $P(0)$.

3 A statement and proof of the inductive case—that is, a proof of $P(n-1) \Rightarrow P(n)$, for a generic value of $n \geq 1$. The proof of the inductive case should include all of the following:

a a statement of the inductive hypothesis $P(n-1)$.

b a statement of the claim $P(n)$ that needs to be proven.

c a proof of $P(n)$, which at some point makes use of the assumed inductive hypothesis $P(n-1)$.

We can use a proof by mathematical induction to establish arithmetic properties, like a formula for the sum of the nonnegative integers up to n—that is, $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ for any integer $n \geq 0$—or a formula for a geometric series:

$$\text{if } r \in \mathbb{R} \text{ where } r \neq 1, \text{ and } n \in \mathbb{Z}^{\geq 0}, \text{ then } \sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r-1}.$$

(If $r = 1$, then $\sum_{i=0}^{n} r^i = n+1$.) We can also use proofs by mathematical induction to prove the correctness of algorithms, particularly recursive algorithms.

Strong Induction

Suppose that we want to prove that $P(n)$ holds for all $n \in \mathbb{Z}^{\geq 0}$. To give a *proof by strong induction* of $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$, we prove the *base case* $P(0)$, and we prove the *inductive case*: for every $n \geq 1$, we have $[P(0) \wedge P(1) \ldots \wedge P(n-1)] \Rightarrow P(n)$. Strong induction is actually completely equivalent to weak induction; anything that can be proven with one can also be proven with the other.

Generally speaking, using strong induction makes sense when the "reason" that $P(n)$ is true is that $P(k)$ is true for more than one value of $k < n$ (or a single value of $k < n$ with $k \neq n-1$). (For weak induction, the reason that $P(n)$ is true is just $P(n-1)$.) We can use strong induction to prove many claims, including part of the Prime Factorization Theorem: if $n \in \mathbb{Z}^{\geq 1}$ is a positive integer, then there exist $k \geq 0$ prime numbers $p_1, p_2, \ldots, p_k$ such that $n = \prod_{i=1}^{k} p_i$.

Recursively Defined Structures and Structural Induction

A recursively defined structure, just like a recursive algorithm, is a structure defined in terms of one or more *base cases* and one or more *inductive cases*. Any data type that can be understood as either a trivial instance of the type or as being built up from a smaller instance (or smaller instances) of that type can be expressed in this way. The set of structures defined is *well ordered* if, intuitively, every invocation of the inductive case of the definition "makes progress" toward the base case(s) of the definition (and, more formally, that every nonempty subset of those structures has a "least" element).

Suppose that we want to prove that $P(x)$ holds for every $x \in S$, where S is the (well-ordered) set of structures generated by a recursive definition. To give a *proof by structural induction* of $\forall x \in S : P(x)$, we prove the following:

1 *Base cases*: for every x defined by a base case in the definition of S, prove $P(x)$.

2 *Inductive cases*: for every x defined in terms of $y_1, y_2, \ldots, y_k \in S$ by an inductive case in the definition of S, prove that $P(y_1) \wedge P(y_2) \ldots \wedge P(y_k) \Rightarrow P(x)$.

The form of a proof by structural induction that $\forall x \in S : P(x)$ for a well-ordered set of structures S is identical to the form of a proof using strong induction. Specifically, the proof by structural induction looks like a proof by strong induction of the claim $\forall n \in \mathbb{Z}^{\geq 0} : Q(n)$, where $Q(n)$ denotes the property "for any structure $x \in S$ that is generated using n applications of the inductive-case rules in the definition of S, we have $P(x)$."

Key Terms and Results

Key Terms

Proofs by Mathematical Induction

- proof by mathematical induction
- base case
- inductive case
- inductive hypothesis
- geometric series
- arithmetic series
- harmonic series

Strong Induction

- strong induction
- prime factorization

Recursively Defined Structures and Structural Induction

- recursively defined structures
- structural induction
- well-ordered set

Key Results

Proofs by Mathematical Induction

1 Suppose that we want to prove that $P(n)$ holds for all $n \in \mathbb{Z}^{\geq 0}$. To give a *proof by mathematical induction* of $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$, we prove the following:

a the *base case* $P(0)$.

b the *inductive case*: for every $n \geq 1$, we have $P(n-1) \Rightarrow P(n)$.

2 For any integer $n \geq 0$, we have $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$.

3 Let $r \in \mathbb{R}$ where $r \neq 1$, and let $n \in \mathbb{Z}^{\geq 0}$. Then

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1} - 1}{r - 1}.$$

(If $r = 1$, then $\sum_{i=0}^{n} r^i = n + 1$.)

Strong Induction

1 Suppose that we want to prove that $P(n)$ holds for all $n \in \mathbb{Z}^{\geq 0}$. To give a *proof by strong induction* of $\forall n \in \mathbb{Z}^{\geq 0} : P(n)$, we prove the following:

a the *base case* $P(0)$.

b the *inductive case*: for every $n \geq 1$, we have $[P(0) \wedge P(1) \ldots \wedge P(n-1)] \Rightarrow P(n)$.

2 The prime factorization theorem: let $n \in \mathbb{Z}^{\geq 1}$ be a positive integer. Then there exist $k \geq 0$ prime numbers $p_1, p_2, \ldots, p_k$ such that $n = \prod_{i=1}^{k} p_i$. Furthermore, up to reordering, the prime numbers $p_1, p_2, \ldots, p_k$ are unique.

Recursively Defined Structures and Structural Induction

1 To give a *proof by structural induction* of $\forall x \in S : P(x)$, we prove the following:

a the *base cases*: for every x defined by a base case in the definition of S, we have that $P(x)$.

b the *inductive cases*: for every x defined in terms of $y_1, y_2, \ldots, y_k \in S$ by an inductive case in the definition of S, we have that $P(y_1) \wedge P(y_2) \ldots \wedge P(y_k) \Rightarrow P(x)$.