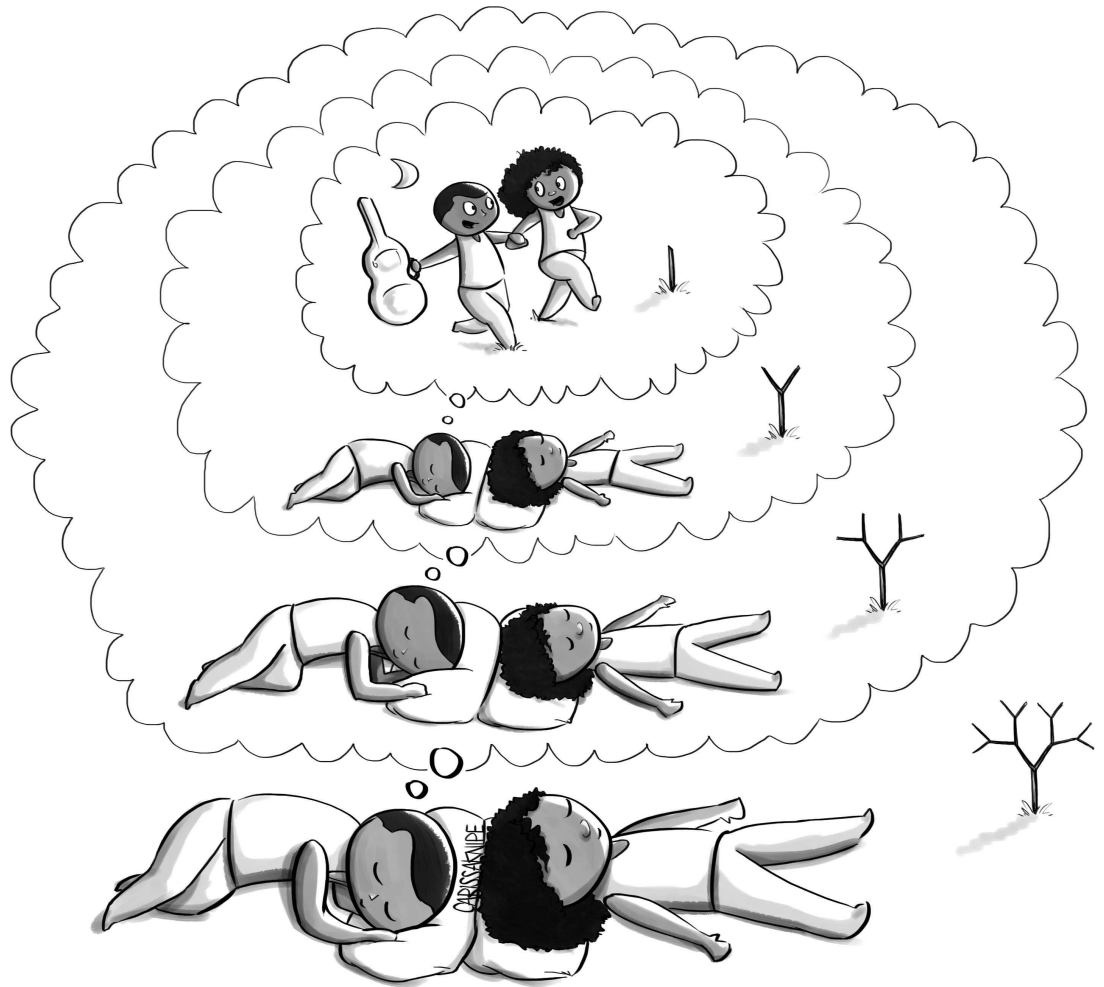


# 5

## 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

Each problem that I solved became a rule which served afterwards to solve other problems.

---

René Descartes (1596–1650)

*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  $n$ -element array  $A$  by sorting the left half of  $A$  and the right half of  $A$  and merging the results together; 1-element arrays are trivially sorted. (That’s *merge sort*.) 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 a trivial empty data structure can store an empty set of keys. (That’s a *binary search tree*.) And many other things are best understood recursively: factorials, the Fibonacci numbers, fractals (see Figure 5.1), and finding the median element of an unsorted array, for example.

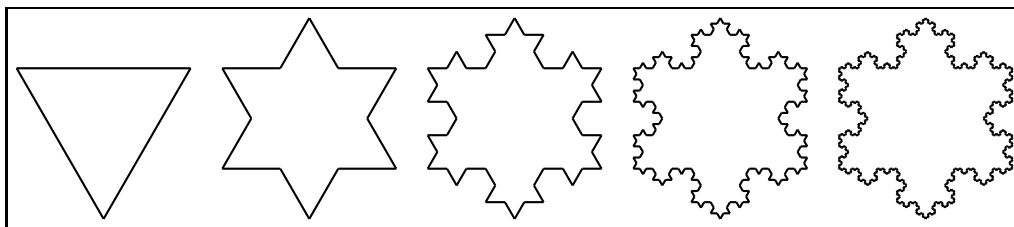
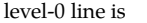



Figure 5.1: The Von Koch Snowflake fractal, shown at levels  $\{0, 1, 2, 3, 4\}$ . A level- $\ell$  snowflake consists of three level- $\ell$  lines. A level-0 line is ; a level- $\ell$  line consists of four level- $(\ell - 1)$  lines arranged in the shape .

*Mathematical induction* is a technique for proofs that is directly analogous to recursion: to prove that  $P(n)$  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 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  $P(0)$ ,  $P(1)$ ,  $\dots$ , 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)

### 5.2.1 An Overview of Proofs by Mathematical Induction

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*). 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 the following:

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  $P(5)$  from a base case and inductive case)**

Problem: 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  $P(5)$ ?

Solution: Here is a proof of  $P(5)$ , using the base case once and the inductive case five times. (At each stage we make use of *modus ponens*—which, as a reminder, states that from  $p \Rightarrow q$  and  $p$ , we can conclude  $q$ .)

We know $P(0)$	<i>base case</i>	(5.1)
and we know $P(0) \Rightarrow P(1)$	<i>inductive case, with <math>n = 1</math></i>	(5.2)
and thus we can conclude $P(1)$ .	<i>(5.1), (5.2), and modus ponens</i>	(5.3)
We know $P(1) \Rightarrow P(2)$	<i>inductive case, with <math>n = 2</math></i>	(5.4)
and thus we can conclude $P(2)$ .	<i>(5.3), (5.4), and modus ponens</i>	(5.5)
We know $P(2) \Rightarrow P(3)$	<i>inductive case, with <math>n = 3</math></i>	(5.6)
and thus we can conclude $P(3)$ .	<i>(5.5), (5.6), and modus ponens</i>	(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)

We know  $P(4) \Rightarrow P(5)$  *inductive case, with  $n = 5$*  (5.10)  
 and thus we can conclude  $P(5)$ . (5.9), (5.10), and *modus ponens* (5.11)

This sequence of inferences established that  $P(5)$  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.

<p><i>Dominoes falling:</i> We have an infinitely long line of dominoes, numbered <math>0, 1, 2, \dots, n, \dots</math>. To convince someone that the <math>n</math>th domino falls over, you can convince them that</p> <ul style="list-style-type: none"> <li>the 0th domino falls over, and</li> <li>whenever one domino falls over, the next domino falls over too.</li> </ul> <p>(One domino falls, and they keep on falling. Thus, for any <math>n \geq 0</math>, the <math>n</math>th domino falls.)</p>
<p><i>Climbing a ladder:</i> We have a ladder with rungs numbered <math>0, 1, 2, \dots, n, \dots</math>. To convince someone that a climber climbing the ladder reaches the <math>n</math>th rung, you can convince them that</p> <ul style="list-style-type: none"> <li>the climber steps onto rung #0.</li> <li>if the climber steps onto one rung, then she also steps onto the next rung.</li> </ul> <p>(The climber starts to climb, and the climber never stops climbing. Thus, for any <math>n \geq 0</math>, the climber reaches the <math>n</math>th rung.)</p>
<p><i>Whispering down the alley:</i> We have an infinitely long line of people, with the people numbered <math>0, 1, 2, \dots, n, \dots</math>. To argue that everyone in the line learns a secret, we can argue that</p> <ul style="list-style-type: none"> <li>person #0 learns the secret.</li> <li>if person #<math>n</math> learns the secret, then she tells person #<math>(n + 1)</math> the secret.</li> </ul> <p>(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 <math>n \geq 0</math>, the <math>n</math>th person learns the secret.)</p>
<p><i>Falling into the depths of despair:</i> 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:</p> <ul style="list-style-type: none"> <li>the Evil Villain is pushed into the pit (that is, She is in the Pit zero meters below the surface).</li> <li>if someone is in the Pit at a depth of <math>n</math> meters beneath the surface, then She falls to depth <math>n + 1</math> meters beneath the surface.</li> </ul> <p>(The Villain starts to fall, and if the Villain has fallen to a certain depth then She falls another meter further. Thus, for any <math>n \geq 0</math>, the Evil Villain eventually reaches depth <math>n</math> in the Pit.)</p>

Figure 5.2: Some analogies to make mathematical induction more intuitive.

**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; hence 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 a simple arithmetic property:

##### Theorem 5.1 (A formula for the sum of powers of two)

For any nonnegative integer  $n$ , we have

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

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. Here is our first example of a proof by induction:

##### Example 5.2 (A proof of Theorem 5.1)

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

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

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

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

*Problem-solving tip:*  
Do this kind of plausibility check, and test out a claim for small values of  $n$  before you try to prove it. Often the process of testing small examples either reveals a misunderstanding of the claim or helps you see why the claim is true in general.

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):

$$\begin{array}{r} \phantom{+} \begin{array}{cccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \\ + \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

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*.

#### 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.*

**Step #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  $n$ .

**Step #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.

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

... **Step #3a:** state the inductive hypothesis.

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

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

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.

We must prove  $P(n)$ .

... Step #3b: state what we need to prove.

... Step #3c: prove it, making use of the inductive hypothesis and stating where it was used.

$$\begin{aligned}
 \sum_{i=0}^n (-1)^i &= \left[ \sum_{i=0}^{n-1} (-1)^i \right] + (-1)^n && \text{definition of summations} \\
 &= \begin{cases} 1 + (-1)^n & \text{if } n-1 \text{ is even} \\ 0 + (-1)^n & \text{if } n-1 \text{ is odd.} \end{cases} && \text{inductive hypothesis} \\
 &= \begin{cases} 1 + (-1)^n & \text{if } n \text{ is odd} \\ 0 + (-1)^n & \text{if } n \text{ is even.} \end{cases} && n \text{ is odd} \Leftrightarrow n-1 \text{ is even} \\
 &= \begin{cases} 1 + -1 & \text{if } n \text{ is odd} \\ 0 + 1 & \text{if } n \text{ is even.} \end{cases} && (-1)^n = \pm 1, \text{ depending on whether } n \text{ is even; see Exercise 5.3.} \\
 &= \begin{cases} 0 & \text{if } n \text{ is odd} \\ 1 & \text{if } n \text{ is even.} \end{cases}
 \end{aligned}$$

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

*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. You should ensure that when you write an inductive proof, you include each of these steps. These steps are summarized in Figure 5.3.

Checklist for a proof by mathematical induction:

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.

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

#### THE SUM OF THE FIRST $n$ INTEGERS

We'll do another simple example of an inductive proof of an arithmetic property, by showing that the sum of the integers between 0 and  $n$  is  $\frac{n(n+1)}{2}$ . (For example, for  $n = 4$  we have  $0 + 1 + 2 + 3 + 4 = 10 = \frac{4(4+1)}{2}$ .) Here's a proof:

##### Example 5.4 (Sum of the first $n$ integers)

**Problem:** Show that  $0 + 1 + \cdots + n$  is  $\frac{n(n+1)}{2}$ , for any integer  $n \geq 0$ .



**Solution:** 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  $\sum_{i=0}^0 i = 0$  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 that  $\sum_{i=0}^n i = \frac{n(n+1)}{2}$ . Here is the proof:

$$\begin{aligned} \sum_{i=0}^n i &= \left[ \sum_{i=0}^{n-1} i \right] + n && \text{definition of summations} \\ &= \frac{(n-1)((n-1)+1)}{2} + n && \text{inductive hypothesis} \\ &= \frac{(n-1)n + 2n}{2} && \text{putting terms over common denominator} \\ &= \frac{n(n-1+2)}{2} && \text{factoring} \\ &= \frac{n(n+1)}{2}. \end{aligned}$$

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

*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!

**Taking it further:** While the summation that we analyzed in Example 5.4 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  $k$ th element backward until it's in position). The total number of swaps that are done in the  $k$ th 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 Example 5.4 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:



**Example 5.5 (Vertices in a Von Koch Line)**

**Problem:** 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  $(\ell - 1)$ , arranged in the shape  $\sim$ . (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  $\ell$ . Prove 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  $\ell$ . Let's start with a few small examples, based on Figure 5.4:

- a level-0 line has 2 endpoints (and 1 segment).
- a level-1 line has 5 endpoints (and 4 segments): the two at the far left and far right, plus the three in the start, middle, and end of the “bump” in the center.
- a level-2 line—after some tedious counting in the picture in Figure 5.4—turns out to have 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- $\ell$  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- $\ell$  line contains  $4^\ell$  lines of level 0. The number of endpoints that we observe is  $2 = 4^0 + 1$ , 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  $\ell$ . 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  $\ell$  has  $4^\ell + 1$  endpoints.

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

**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  $\ell$  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:

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

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

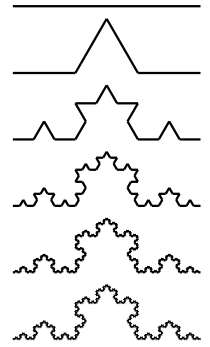


Figure 5.4: Von Koch lines of level  $0, 1, \dots, 5$ . (A Von Koch snowflake consists of three Von Koch lines, all of the same level, arranged in a triangle; see Figure 5.1.)

## A NOTE AND TWO VARIATIONS ON THE INDUCTIVE TEMPLATE

The basic idea of induction is simple: 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  $P(n)$ .

**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.

*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!

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  $n$ th domino falls because the  $(n-1)$ st domino fell into it” or as “the  $n$ th domino falls and therefore knocks over the  $(n+1)$ st domino.”

In the remainder of this section, we'll give some more examples of proofs by mathematical induction, following the template of Figure 5.3. While the examples that we've used so far have almost all related to summations, the same style of inductive proof can be used for a wide variety of claims. We'll encounter many inductive proofs throughout the book, and you'll find inductive proofs ubiquitous throughout computer science. We'll start with some more summation-based proofs, and then move on to inductive proofs of some other types of statements.

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

We'll now introduce three types of summations that arise frequently in computer science: *geometric* sequences  $(1, 2, 4, 8, 16, \dots)$ ; *arithmetic* sequences  $(2, 4, 6, 8, 10, \dots)$ ; and the *harmonic* sequence  $(1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \dots)$ . Summations involving all of these types of sequences can be analyzed inductively, and we'll address all three of them here and in the exercises. (The statements we'll prove are both useful facts to know about geometric/arithmetic/harmonic sequences, and good practice with induction.)

## GEOMETRIC SERIES

**Definition 5.2 (Geometric sequences and series)**

A geometric sequence is a sequence of numbers where each number is generated by multiplying the previous entry by a fixed ratio  $\alpha \in \mathbb{R}$ , starting from an initial value  $x_0$ .

(Thus the sequence is  $\langle x_0, x_0 \cdot \alpha, x_0 \cdot \alpha^2, x_0 \cdot \alpha^3, \dots \rangle$ .) A geometric series or geometric sum is  $\sum_{i=0}^n x_0 \alpha^i$ .

Examples include  $\langle 2, 4, 8, 16, 32, \dots \rangle$ ; or  $\langle 1, \frac{1}{3}, \frac{1}{9}, \frac{1}{27}, \dots \rangle$ ; or  $\langle 1, 1, 1, 1, \dots \rangle$ .

It turns out that there is a relatively simple formula expressing the sum of the first  $n$  terms of a geometric sequence:

**Theorem 5.2 (Analysis of geometric series)**

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

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

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

(For simplicity, we stated Theorem 5.2 without reference to  $x_0$ . Because we can pull a constant multiplicative factor out of a summation, we can use the theorem to conclude that  $\sum_{i=0}^n x_0 \alpha^i = x_0 \cdot \sum_{i=0}^n \alpha^i = x_0 \cdot \frac{\alpha^{n+1} - 1}{\alpha - 1}$ .)

We will be able to prove Theorem 5.2 using a proof by mathematical induction:

**Example 5.6 (Geometric series)**

*Proof of Theorem 5.2.* Consider a fixed real number  $\alpha$  with  $\alpha \neq 1$ , and let  $P(n)$  denote the property that

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

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

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

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

$$\sum_{i=0}^{n-1} \alpha^i = \frac{\alpha^n - 1}{\alpha - 1},$$

and we must prove  $P(n)$ . Here is the proof:

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

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

*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 5.2. For the former, Theorem 5.2 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 5.2 addressing *infinite* geometric sums will turn out to be useful later, so we'll state it now. (You can skip over the proof if you don't know calculus, or if you haven't thought about calculus recently.)

### Corollary 5.3

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

1.  $\sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$ , and
2. For all  $n \geq 0$ , we have  $1 \leq f(n) \leq \frac{1}{1-\alpha}$ .

*Proof.* The proof of (1) requires calculus. Theorem 5.2 says that  $f(n) = \frac{\alpha^{n+1}-1}{\alpha-1}$ , and we take the limit as  $n \rightarrow \infty$ . Because  $\alpha < 1$ , we have that  $\lim_{n \rightarrow \infty} \alpha^{n+1} = 0$ . Thus as  $n \rightarrow \infty$  the numerator  $\alpha^{n+1} - 1$  tends to  $-1$ , and the entire ratio tends to  $1/(1-\alpha)$ .

For (2), observe that  $\sum_{i=0}^n \alpha^i$  is definitely greater than or equal to  $\sum_{i=0}^0 \alpha^i$  (because  $\alpha \geq 0$  and so the latter results by eliminating  $n$  nonnegative terms from the former). Similarly,  $\sum_{i=0}^n \alpha^i$  is definitely less than or equal to  $\sum_{i=0}^{\infty} \alpha^i$ . Thus:

$$\begin{aligned} f(n) = \sum_{i=0}^n \alpha^i &\geq \sum_{i=0}^0 \alpha^i = \alpha^0 = 1 \\ f(n) = \sum_{i=0}^n \alpha^i &\leq \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}. \end{aligned}$$

□

## ARITHMETIC SERIES

### Definition 5.3 (Arithmetic sequences and series)

An arithmetic sequence is a sequence of numbers where each number is generated by adding a fixed step-size  $\alpha \in \mathbb{R}$  to the previous number in the sequence. The first entry in the sequence is some initial value  $x_0 \in \mathbb{R}$ . (Thus the sequence is  $\langle x_0, x_0 + \alpha, x_0 + 2\alpha, x_0 + 3\alpha, \dots \rangle$ .) An arithmetic series or sum is  $\sum_{i=0}^n (x_0 + i\alpha)$ .

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

## HARMONIC SERIES

### Definition 5.4 (Harmonic series)

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

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

Giving a precise equation for the value of  $H_n$  requires a bit more work, but we can very easily prove upper and lower bounds on  $H_n$  by induction. (If you've had calculus, then there's a simple way for you to approximate the value of  $H_n$ , as

$$H_n = \sum_{x=1}^n \frac{1}{x} \approx \int_{x=1}^n \frac{1}{x} dx = \ln n.$$

But we'll do a calculus-free version here.) We will be able to prove the following, which captures the value of  $H_n$  to within a factor of 2, at least when  $n$  is a power of 2:

**Theorem 5.4 (Bounds on the  $(2^k)$ th harmonic number)**

For any integer  $k \geq 0$ , we have  $k+1 \geq H_{2^k} \geq \frac{k}{2} + 1$ .

The name “harmonic” comes from music: when a note at frequency  $f$  is played, *overtone*s of that note—other high-intensity frequencies—can be heard at frequencies  $2f, 3f, 4f, \dots$ . The *wavelengths* of the corresponding sound waves are  $\frac{1}{f}, \frac{1}{2f}, \frac{1}{3f}, \frac{1}{4f}, \dots$

We'll prove half of Theorem 5.4 (namely  $k+1 \geq H_{2^k}$ ) by induction in Example 5.7, leaving the other half to the exercises. We will also leave to the exercises a proof of 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  $H_{2^k} = 1 = k+1$ .

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

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

Thus we've proven that  $H_{2^k} \leq k+1$ —that is, we've proven  $P(k)$ . This proof completes the inductive case, and the theorem follows.  $\square$

The proof in Example 5.7 is perhaps the first time in this chapter in which we needed some serious insight and creativity to establish the inductive case. The structure of a proof 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.

### 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? It turns out that the latter algorithm is faster, and we can prove this fact (with a small caveat for small  $n$ ) by induction:

#### 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 5.5.) Indeed,  $2^3 < 3^2$ . But the relationship appears to begin 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  $n \geq 4$ .

**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

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

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

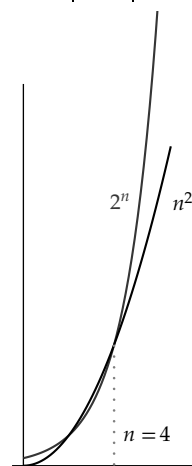


Figure 5.5: Small values of  $2^n$  and  $n^2$ , and a plot of the functions.



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

**Taking it further:** In analyzing the efficiency of algorithms, we will frequently have to do the type of comparison that we just completed, 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 takes requires testing  $2^n$  subsequences of  $X$ . But there's a cleverer approach to solving this problem using an algorithmic design technique called *dynamic programming* (see p. 959 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 a simple example, for the natural recursive algorithm to compute factorials (see Figure 5.6):

```
fact(n):
1: if n = 1 then
2:   return 1
3: else
4:   return n · fact(n - 1)
```

Figure 5.6: Pseudocode for factorial: given  $n \in \mathbb{Z}^{\geq 1}$ , we wish to compute the value of  $n!$ .

#### Example 5.9 (Factorial)

Consider the recursive algorithm **fact** in Figure 5.6. For a positive integer  $n$ , let  $P(n)$  denote the property that **fact**( $n$ ) =  $n!$ . We'll prove by induction on  $n$  that, indeed,  $P(n)$  holds for all integers  $n \geq 1$ .

**base case ( $n = 1$ ):** Observe that **fact**(1) returns 1 immediately. And  $1! = 1$  by definition. Thus  $P(1)$  holds.

**inductive case ( $n \geq 2$ ):** We assume the inductive hypothesis  $P(n - 1)$ , namely that **fact**( $n - 1$ ) returns  $(n - 1)!$ . We want to prove that **fact**( $n$ ) returns  $n!$ . But this claim is easy to see:

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

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.)

**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 the discussion on p. 517.

## DIVISIBILITY

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

### 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:

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

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.  $\square$

*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!

*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<sup>1</sup>—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 clear; 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.

Here's an example of a sketch of a proof of correctness of Insertion Sort (Figure 5.7) using loop invariants. (Many proofs using loop invariants would proceed with more formal detail.) We claim that the property

$P(k) := A[1 \dots k+1]$  is sorted after completing  $k$  iterations of the outer **while** loop

is true for all  $k \geq 0$ . (That is,  $P$  is a loop invariant for the outer **while** loop.)

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

For the inductive case ( $k \geq 1$ ), we assume the inductive hypothesis  $P(k-1)$ —that is,  $A[1 \dots k]$  was sorted before the  $k$ th iteration. The  $k$ th iteration of the loop executed lines 2–7, so we must show that the execution of these lines extended the sorted segment  $A[1 \dots k]$  to  $A[1 \dots k+1]$ . A formal proof of this claim would use another loop invariant, like

$Q(j) := \text{both } A[1 \dots j-1] \text{ and } A[j \dots i] \text{ are sorted, and } A[j-1] < A[j+1]$

but for this proof sketch we'll be satisfied by concluding the desired conclusion by inspection of the algorithm's code.  $\square$

Because  $P(n-1)$  is true (after  $n-1$  iterations of the loop), we know that  $A[1 \dots (n-1)+1] = A[1 \dots n]$  is sorted, as desired.

Loop invariants can also be extremely valuable as part of the development of programs. For example, many people end up struggling to correctly write binary search—but by writing down loop invariants before actually writing the code, it's actually easy. If we think about the property

*if  $x$  is in  $A$ , then  $x$  is one of  $A[lo, \dots, hi]$*

as a loop invariant as we write the program, binary search becomes much easier to get right. 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.

<sup>1</sup> Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics XIX*, American Mathematical Society, pages 19–32, 1967; and C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–585, October 1969.

```
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
```

Figure 5.7: Insertion Sort.

```
binarySearch(A[1...n], x):
//output: is x in the sorted array A?
1: lo := 1
2: hi := n
3: while lo ≤ hi:
4:   middle := ⌊(lo+hi)/2⌋
5:   if A[middle] = x then
6:     return True
7:   else if A[middle] > x then
8:     hi := middle - 1
9:   else
10:    lo := middle + 1
11: return False
```

Figure 5.8: Binary Search.

## 5.2.4 Exercises

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

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

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

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

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

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

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

5.7 In a typical optical camera lens, the light that enters the lens (through the opening called the *aperture*) is controlled by a collection of movable blades that can be adjusted inward to narrow the area through which light can pass. (There are two effects of narrowing this opening: first, the amount of light entering the lens is reduced, darkening the resulting image; and, second, the *depth of field*—the range of distances from the lens at which objects are in focus in the image—increases.) Although some lenses allow continuous adjustment to their openings, many have a sequence of so-called *stops*: discrete steps by which the aperture narrows. (See Figure 5.9.) These steps are called *f-stops* (the “f” is short for “focal”), and they are denoted with some unusual notation that you’ll unwind in this exercise. 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, you might use a lens that’s 50mm long and that has a 25mm diameter, which yields an *f-stop* of  $50\text{mm}/25\text{mm} = 2$ .) One can also “stop down” a lens from this fastest setting by adjusting the blades to shrink the diameter of the aperture, as described above. (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  $50\text{mm}/12.5\text{mm} = 4$ .)

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

$$d_{i+1} := \frac{d_i}{\sqrt{2}}, \text{ for any } i \geq 0.$$

Give a closed-form expression for  $d_n$ —that is, give a nonrecursive numerical expression whose value is equal to  $d_n$  (where your expression involves only real numbers and the variable  $n$ ). Prove your answer correct by induction on  $n$ . Also 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 5.9?)

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

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

5.10 Let  $\alpha \in \mathbb{R}$  and let  $n \in \mathbb{Z}^{\geq 0}$ , and consider the arithmetic sequence  $\langle x_0, x_0 + \alpha, x_0 + 2\alpha, \dots \rangle$ . (Recall that each entry in an arithmetic sequence is a fixed amount more than the previous entry. Three examples are  $\langle 1, 3, 5, 7, 9, \dots \rangle$ , with  $x_0 = 1$  and  $\alpha = 2$ ;  $\langle 25, 20, 15, 10, \dots \rangle$ , with  $x_0 = 25$  and  $\alpha = -5$ ; and  $\langle 5, 5, 5, 5, \dots \rangle$ , with  $x_0 = 5$  and  $\alpha = 0$ .) An *arithmetic sum* or *arithmetic series* is the sum of an arithmetic sequence. For the arithmetic sequence  $\langle x_0, x_0 + \alpha, x_0 + 2\alpha, \dots \rangle$ , formulate and prove correct by induction a formula expressing the value of the arithmetic series

$$\sum_{i=0}^n (x_0 + i\alpha).$$

(Hint: note that  $\sum_{i=0}^n i\alpha = \alpha \sum_{i=0}^n i = \frac{\alpha n(n+1)}{2}$ , by Example 5.4.)

5.11 In chess, 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 ( $\langle r \pm 1, c \pm 2 \rangle$ ), or two rows over and one column up/down ( $\langle r \pm 2, c \pm 1 \rangle$ ). (See Figure 5.10.) A *knight’s walk* is a sequence of legal moves, starting from a square of your choice, that visits *every* square of the board. Prove by induction that there exists a knight’s walk for any  $n$ -by- $n$  chessboard for any  $n \geq 4$ . (A *knight’s tour* is a knight’s walk that visits every square *only* once. It turns out that knight’s tours exist for all even  $n \geq 6$ , but you don’t need to prove this fact.)

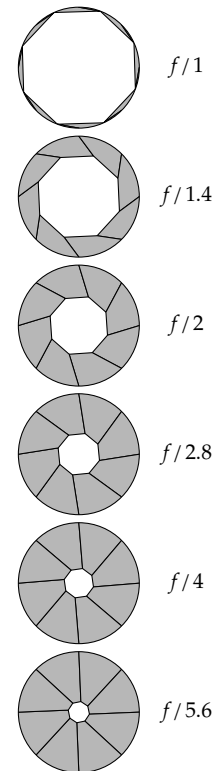


Figure 5.9: A particular lens of a camera, shown at several different *f-stops*. These configurations are only an approximation—the real blades are shaped somewhat differently than is shown here.

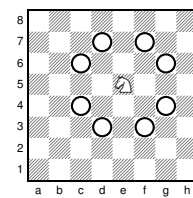


Figure 5.10: A chess board. The knight can move to any of the marked positions.

**5.12** (programming required) In a programming language of your choice, implement your proof from Exercise 5.11 as a recursive algorithm that *computes* a knight's walk in an  $n$ -by- $n$  chessboard.

**5.13** In chess, a rook at position  $\langle r, c \rangle$  can move in a straight line either horizontally or vertically (to  $\langle r \pm x, c \rangle$  or  $\langle r, c \pm x \rangle$ , for any integer  $x$ ). (See Figure 5.11.) A *rook's tour* is a sequence of legal moves, starting from a square of your choice, that visits *every* square of the board *once and only once*. Prove by induction that there exists a rook's tour for any  $n$ -by- $n$  chessboard for any  $n \geq 1$ .

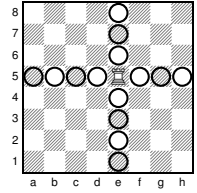


Figure 5.11: A rook can move to any of the positions marked with a circle.

Figure 5.12 shows three different fractals. One is the Von Koch snowflake (Figure 5.12(a)), which we've already seen: 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  $\ell$  consists of four Von Koch lines of size  $(s/3)$  and level  $(\ell - 1)$  arranged in the shape  $\sim$ ; a Von Koch snowflake of size  $s$  and level  $\ell$  consists of a triangle of three Von Koch lines of size  $s$  and level  $\ell$ .

The other two fractals in Figure 5.12 are new. Figure 5.12(b) shows the Sierpinski triangle: a Sierpinski triangle of level 0 and size  $s$  is an equilateral triangle of side length  $s$ ; a Sierpinski triangle of level  $(\ell + 1)$  is three Sierpinski triangles of level  $\ell$  and side length  $s/2$  arranged in a triangle. Figure 5.12(c) shows a related fractal called the Sierpinski carpet, recursively formed from 8 smaller Sierpinski carpets (arranged in a 3-by-3 grid with a hole in the middle); the base case is just a filled square.

Suppose that we draw each of these fractals at level  $\ell$  and with size 1. What is the perimeter of each of these fractals? (By "perimeter," we mean the total length of all boundaries separating regions inside the figure from regions outside—which includes, for example, the boundary of the "hole" in the Sierpinski carpet. For the Sierpinski fractals as drawn here, the perimeter is precisely the length of lines separating colored-in from uncolored-in regions.) In each case, conjecture a formula and prove your answer correct by induction.

**5.14** Von Koch snowflake      **5.15** Sierpinski triangle      **5.16** Sierpinski carpet

Draw each of these fractals at level  $\ell$  and with size 1. What is the enclosed area of each of these fractals? (Again, for the Sierpinski fractals as drawn here, the enclosed area is precisely the area of the colored-in regions.)

**5.17** Von Koch snowflake      **5.18** Sierpinski triangle      **5.19** Sierpinski carpet

In the last few exercises, you computed the fractals' perimeter/area at level  $\ell$ . But what if we continued the fractal-expansion process forever? What are the area and perimeter of an infinite-level fractal? (Hint: use Corollary 5.3.)

**5.20** Von Koch snowflake      **5.21** Sierpinski triangle      **5.22** Sierpinski carpet

The Von Koch snowflake is named after Helge von Koch, a 19th/20th-century Swedish mathematician; the Sierpinski triangle/carpet are named after Waclaw Sierpiński, a 20th-century Polish mathematician.

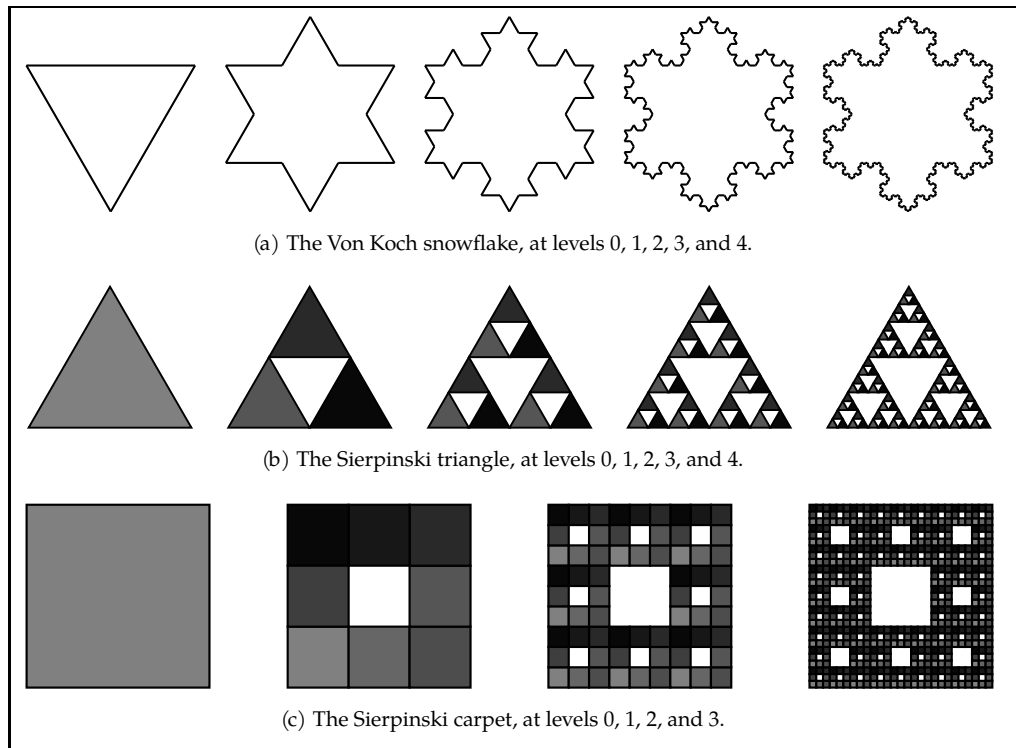


Figure 5.12: Three fractals: the Von Koch snowflake, the Sierpinski triangle, and the Sierpinski carpet.

**5.23** (*programming required*) Write a recursive function `sierpinskiTriangle(level, length, x, y)`, in a language of your choice, to draw a Sierpinski triangle of side length `length` at level `level` with bottom-left coordinate  $(x, y)$ . (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 returns both the *total length* and *total area* of the triangles that it draws. Use your function to verify some small cases of Exercises 5.15 and 5.18.

**5.24** (*programming required*) Write a recursive function `sierpinskiCarpet(level, length, x, y)`, in a programming language of your choice, to draw a Sierpinski carpet. (See Exercise 5.23 for the meaning of the parameters.) Write your function so that—in addition to drawing the fractal—it also returns the *area* of the boxes that it encloses. Use your function to verify some small cases of your answer to Exercise 5.19.

**5.25** An *n*-by-*n* magic square is an *n*-by-*n* grid into which the numbers  $1, 2, \dots, n^2$  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, a 3-by-3 magic square is shown in Figure 5.13. Conjecture and prove a formula for what the sum of each row / column / diagonal must be in an *n*-by-*n* magic square.

4	9	2
3	5	7
8	1	6

Figure 5.13: A 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 5.4, which states that  $k+1 \geq H_{2^k} \geq \frac{k}{2} + 1$  for any integer  $k \geq 0$ .

**5.26** 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 5.4: show by induction that  $H_{2^k} \geq \frac{k}{2} + 1$  for any integer  $k \geq 0$ .

**5.27** Generalize Theorem 5.4 to numbers that aren't necessarily exact powers of 2. Specifically, prove that  $\log n + 2 \geq H_n \geq (\log n - 1)/2 + 1$  for any real number  $n \geq 1$ . (Hint: use Theorem 5.4.)

**5.28** 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  $f(n) \leq g(n)$  hold “for sufficiently large *n*.” That is, identify an integer *k* and then prove (by induction on *n*) that  $f(n) \leq g(n)$  for all integers  $n \geq k$ .

**5.29**  $2^n \leq n!$

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

**5.31**  $3n \leq n^2$

**5.32**  $n^3 \leq 2^n$

**5.33** Prove that, for any nonnegative integer *n*, the algorithm `odd?(n)` returns True if and only if *n* is odd. (See Figure 5.14.)

**5.34** Prove that the algorithm `sum(n, m)` returns  $\sum_{i=n}^m i$  (again see Figure 5.14) for any  $m \geq n$ . (Hint: perform induction on the value of  $m - n$ .)

**5.35** Describe how your proof from Exercise 5.34 would change if Line 4 from the `sum` algorithm in Figure 5.14 were changed to return  $m + \text{sum}(n, m-1)$  instead of  $n + \text{sum}(n+1, m)$ .

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

**5.37** 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.)

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

**5.39** 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  $00 \dots 0$ , 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, for  $n = 3$ , we have  $000 \rightarrow 001 \rightarrow 010 \rightarrow 011 \rightarrow 100 \rightarrow 101 \rightarrow 110 \rightarrow 111$ , which has a total of  $3 + 1 + 2 + 1 + 3 + 1 + 2 + 1 = 14$  bit flips. Prove your answer.

**5.40** To protect my backyard from my neighbor, a biology professor who is sometimes a little over-friendly, 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 5.15.) 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)

```

Figure 5.14: Two algorithms.

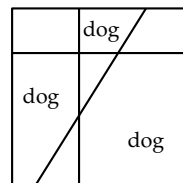
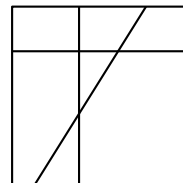


Figure 5.15: A configuration of fences, and a valid way to deploy my dogs.



### 5.3 Strong Induction

It's not true that life is one damn thing after another; it is one damn thing over and over.

---

Edna St. Vincent Millay (1892–1950)

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:  $P(0), P(1), P(2), \dots$ , 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 5.5 (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  $[P(0) \wedge P(1) \wedge \dots \wedge P(n-1)] \Rightarrow P(n)$ .

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  $P(0) \wedge P(1) \wedge \dots \wedge P(n-1)$ ; 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. (Strong induction is actually equivalent to weak induction; anything that can be proven with one can also be proven with the other. See Exercises 5.75–5.76.)

#### 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 5.16 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.

*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.

```

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

```

Figure 5.16: A simple parity algorithm.

#### Example 5.11 (The correctness of **parity**)

**Claim:** For any nonnegative integer  $n \geq 0$ ,

$$\text{parity}(n) = n \bmod 2.$$

*Proof.* Write  $P(n)$  to denote the property that  $\text{parity}(n) = n \bmod 2$ . 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, **parity**(0) returns 0 in Line 2, and, indeed,  $0 \bmod 2 = 0$ . Similarly, we have **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  $P(0) \wedge P(1) \wedge \dots \wedge P(n-1)$ . Namely, assume that

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

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

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

where  $(n-2) \bmod 2 = n \bmod 2$  by Definition 2.9. (Note that the inductive hypothesis applies for  $k := n-2$  because  $n \geq 2$  and therefore  $0 \leq n-2 < n$ .)  $\square$



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.

### 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 called *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 5.5 (Prime Factorization Theorem)

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

The prime factorization theorem is also sometimes called the *Fundamental Theorem of Arithmetic*.

While proving the *uniqueness* requires a bit more work (see Section 7.3.3), we can give a proof using strong induction to show that a prime factorization *exists*.

#### Example 5.12 (Prime factorization)

Let  $P(n)$  denote the first part of Theorem 5.5, namely the claim

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

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.) 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:

- If  $n$  is prime, then there’s nothing to do: define  $p_1 := n$ , and we’re done immediately. (We’ve written  $n$  as the product of 1 prime number.)
- 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  $a \notin \{1, n\}$ ; it should be easy 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

$$a = q_1 \cdot q_2 \cdot \dots \cdot q_\ell \quad \text{and} \quad b = r_1 \cdot r_2 \cdot \dots \cdot r_m \quad (*)$$

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

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

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, as shown in Figure 5.17. (Notice that there's some magic in the "algorithm," in the sense that Line 7 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 7. The second part of Theorem 5.5, 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.)

```

primeFactor(n):
1: if  $n = 1$  then
2:   return  $\langle \rangle$                                 or " $P(1)$  is true!"
3: else
4:   if  $n$  is prime then
5:     return  $\langle n \rangle$                                 or " $P(n)$  is true!"
6:   else
7:     find factors  $a, b$  where  $2 \leq a \leq n - 1$  and  $2 \leq b \leq n - 1$  such that  $n = a \cdot b$ .
8:      $\langle q_1, \dots, q_k \rangle := \mathbf{primeFactor}(a)$ 
9:      $\langle r_1, \dots, r_m \rangle := \mathbf{primeFactor}(b)$ 
10:    return  $\langle q_1, \dots, q_k, r_1, \dots, r_m \rangle$     or " $P(n)$  is true, because  $P(a) \wedge P(b)$ !"

```

Figure 5.17: The proof of Example 5.12, interpreted as a recursive algorithm.

### 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 5.18 for an example. Here we will show that an arbitrary convex polygon can be decomposed into a collection of nonoverlapping triangles.

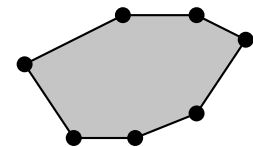


Figure 5.18: A polygon. The dots are called *vertices*; the lines connecting them are the *sides*; and the shaded region (excluding the boundary) is the *interior*.

#### Example 5.13 (Decomposing a polygon into triangles)

**Problem:** Prove the following claim:

**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.

(For an example, and an outline of a possible proof, see Figure 5.19.)

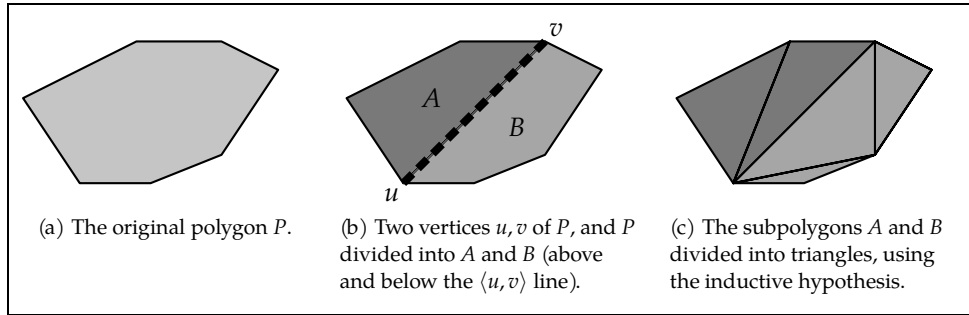


Figure 5.19: An example of the recursive decomposition of a polygon into interior-disjoint triangles.

**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: 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  $Q(3), Q(4), \dots, Q(k - 1)$ .) 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  $\ell$  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  $\ell$  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 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  $Q(3) \wedge \dots \wedge Q(k - 1) \Rightarrow Q(k)$  for any  $k \geq 4$ , which completes the proof by strong induction.

**Taking it further:** The style of *triangulation* from Example 5.13 has particularly 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. 528 for more discussion.

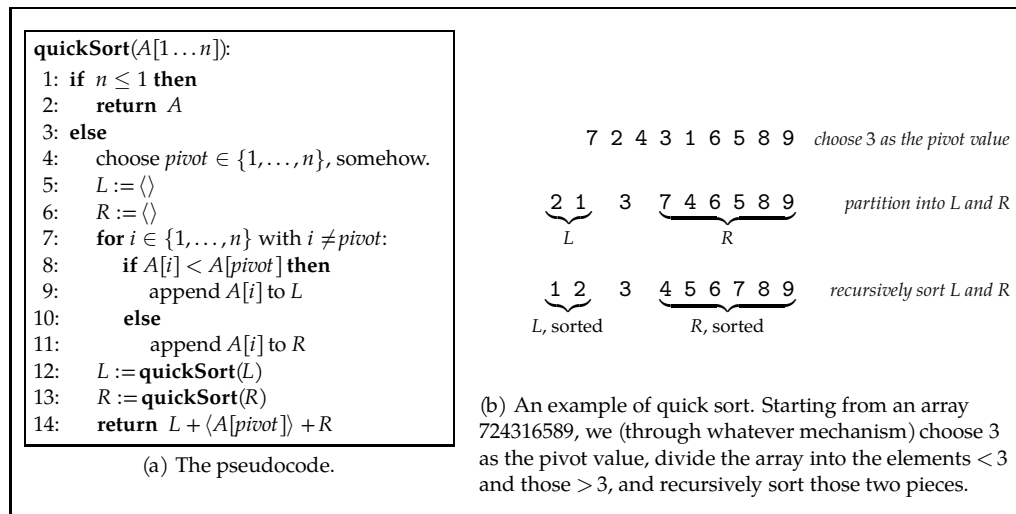


Figure 5.20: Quick Sort: pseudocode, and an example.

## PROVING ALGORITHMS CORRECT: QUICK SORT

We've now seen a proof of correctness by strong induction for a simple 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.

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 5.20(a) for the algorithm.) For example, suppose we wish to sort all 43 U.S. Presidents by birthday. (Grover Cleveland will appear only once.) Barack Obama's birthday is August 4th. If we choose him as the pivot, then Quick Sort would first divide all the other presidents into two lists, of those with pre-August 4th and post-August 4th birthdays,

$\text{before}[1 \dots 23] := \langle \text{George Washington [February 22nd]}, \dots, \text{George W. Bush [July 6th]} \rangle$   
 $\text{after}[1 \dots 19] := \langle \text{John Adams [October 30th]}, \dots, \text{Bill Clinton [August 19th]} \rangle,$

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

$\text{before in sorted order} \quad \text{Barack Obama} \quad \text{after in sorted order}$   
 $\text{pres}[1], \dots, \text{pres}[23], \quad \text{pres}[24], \quad \text{pres}[25], \dots, \text{pres}[43]$

(See Figure 5.20(b) for another example of Quick Sort.)

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**( $A$ ) re-

Even without two Grover Cleveland entries in the array, the simplifying assumption that we're making about distinct elements actually doesn't apply for the U.S. Presidents: James Polk and Warren Harding were both born on November 2nd. (Think about how you'd modify the proof that follows to handle duplicates.)

turns 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:

**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  $A[1 \dots n]$  be an arbitrary array with distinct elements. Let  $\text{pivot} \in \{1, \dots, n\}$  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  $A[\text{pivot}]$ . (See Figure 5.21.)

*Case 1:*  $x = A[\text{pivot}]$ . 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:*  $y = A[\text{pivot}]$ . 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:*  $x < A[\text{pivot}]$  and  $y < A[\text{pivot}]$ . Then both  $x$  and  $y$  appear in  $L$ . Because  $A[\text{pivot}]$  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:*  $x > A[\text{pivot}]$  and  $y > A[\text{pivot}]$ . 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:*  $x < A[\text{pivot}]$  and  $y > A[\text{pivot}]$ . 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:*  $x > A[\text{pivot}]$  and  $y < A[\text{pivot}]$ . 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.  $\square$

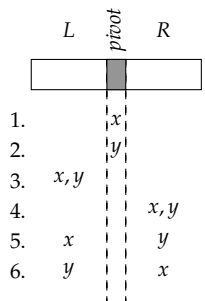


Figure 5.21: The cases of the proof in Example 5.14.

**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. 529 for a discussion of *maximum heaps*.



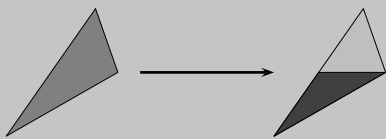
## COMPUTER SCIENCE CONNECTIONS

## 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 algorithms for displaying these scenes—approximates the three-dimensional shapes of the objects in the scene using triangles instead of arbitrary shapes. 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.

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. 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:



Note that there are many different ways to subdivide a given triangle into two 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.<sup>2</sup> (See Figure 5.22.)

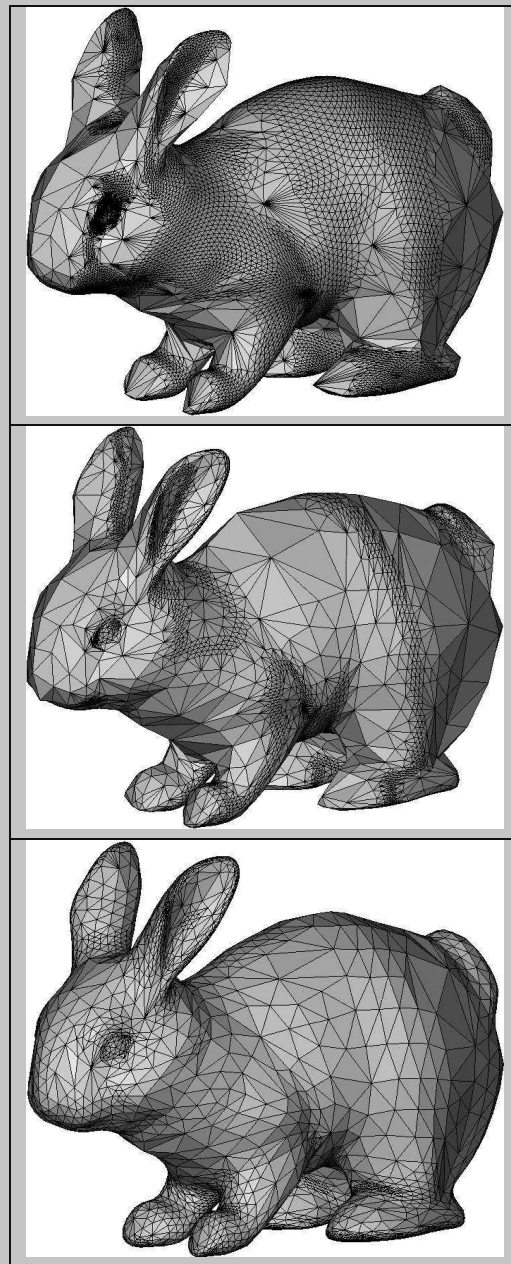


Figure 5.22: Three strategies for refining a triangulation of a rabbit. Reprinted, with permission, from:

<sup>2</sup> Tobias Isenberg, Knut Hartmann, and Henry König. Interest value driven adaptive subdivision. In *Simulation and Visualisation (SimVis)*, pages 139–149. SCS European Publishing House, 2003.

## 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 5.23): 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 5.24.

It's easy to check that the topmost node (the *root*) of the maximum heap in Figure 5.24 has the highest priority. Heaps are designed so that the root of the tree contains the node with the highest priority—but this claim requires proof. Here is a proof by induction:

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

**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  $\ell$  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  $\ell$ .

**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  $P(1), \dots, P(\ell - 1)$ .

Let  $x$  be the priority of the root of the tree. 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  $\ell - 1$  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  $z \leq x$ .  $\square$

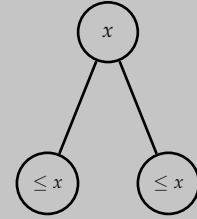


Figure 5.23: The maximum heap property. For a node with value  $x$ , the children must have values  $\leq x$ .

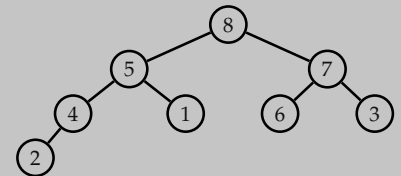
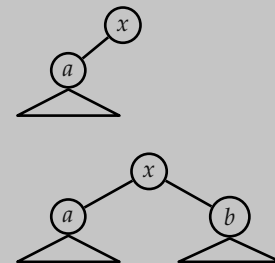


Figure 5.24: A maximum heap.





## 5.3.3 Exercises

**5.41** In Example 5.11, we showed the correctness of the **parity** function (see Figure 5.25)—that is, for any  $n \geq 0$ , we have that  $\text{parity}(n) = n \bmod 2$ . Prove by strong induction on  $n$  that the *depth* of the recursion (that is, the total number of calls to **parity** made) for **parity**( $n$ ) is  $1 + \lfloor n/2 \rfloor$ .

**5.42** Consider the algorithm in Figure 5.25, which finds the binary representation of a given integer  $n \geq 0$ . For example, **toBinary**(13) =  $\langle 1, 1, 0, 1 \rangle$ , and  $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 0 + 1 = 13$ .

Prove the correctness of **toBinary** by strong induction—that is, prove that for any  $n \geq 0$ , we have  $\sum_{i=0}^k b_i 2^i = n$  where **toBinary**( $n$ ) =  $\langle b_k, \dots, b_0 \rangle$ .

Your proof of the correctness of **toBinary**( $n$ ) establishes that any nonnegative integer can be represented in binary. Now you'll show that this binary representation is unique—or, at least, unique up to leading zeros. (For example, we can represent 7 in binary as 111 or 0111 or 00111, but only 111 has no leading zeros.)

**5.43** 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  $a := \langle a_k, a_{k-1}, \dots, a_0 \rangle$  and  $b := \langle b_k, b_{k-1}, \dots, b_0 \rangle$  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  $i \in \{k, k-1, \dots, 0\}$  we have  $a_i = b_i$ .

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

In Chapter 7, we'll talk in a great deal more detail about modular arithmetic, and we'll discuss a more general algorithm for converting from one base to another on p. 714. In Chapter 7, we'll do most of the computation iteratively; here you'll fill in a few pieces recursively.

**5.44** Generalize the **parity**( $n$ ) algorithm to **remainder**( $n, k$ ) to recursively compute the number  $r \in \{0, 1, \dots, k-1\}$  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 5.25. Prove your algorithm correct using strong induction on  $n$ .

**5.45** 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**( $n, k$ ) =  $\langle b_\ell, b_{\ell-1}, \dots, b_0 \rangle$  with each  $b_i \in \{0, 1, \dots, k-1\}$ , then  $n = \sum_{i=0}^{\ell} k^i b_i$ .

**5.46** 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  $b \geq 0$ .)

**5.47** 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.

**5.48** 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 called the Cis-Patriarchal Pet Shop 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 can choose to remove either one or two cards. A player wins by taking the last credit card. (See Figure 5.26.)

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.

Consider the following modifications of the game from Exercise 5.48. The two players start with  $n$  cards on the table, as before. Determine who wins the modified game: conjecture a condition on  $n$  that describes precisely when the first player can guarantee a win under the stated modification, and prove your answer.

**5.49** Let  $k \geq 2$  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.

**5.50** As in the original game, players can take only 1 or 2 cards per turn—but the player who is forced to take the last card *loses* (instead of winning by managing to take the last card).

```

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

```

```

toBinary( $n$ ):        //assume that  $n \geq 0$  is an integer.
1: if  $n \leq 1$  then
2:   return  $\langle n \rangle$ 
3: else
4:    $\langle b_k, \dots, b_0 \rangle := \text{toBinary}(\lfloor n/2 \rfloor)$ 
5:    $x := \text{parity}(n)$ 
6:   return  $\langle b_k, \dots, b_0, x \rangle$ 

```

Figure 5.25: A reminder of the parity algorithm (from Figure 5.16), and an algorithm to convert an integer to binary.

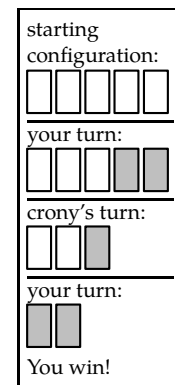


Figure 5.26: You start with  $n = 5$  cards on the table, and you make the first move. You win because you took the last card.

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, ... (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  $n$ :

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

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

**5.53**  $\sum_{i=1}^n f_i = f_{n+2} - 1$

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

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

**5.56** For a  $k$ -by- $k$  matrix  $M$ , the matrix  $M^n$  is also  $k$ -by- $k$ , and its value is the result of the  $n$ -fold multiplication of  $M$  by itself:  $MM \cdots M$ . Or we can define matrix exponentiation recursively:  $M^0 := I$  (the  $k$ -by- $k$  identity matrix), and  $M^{n+1} := M \cdot M^n$ . With this definition in mind, prove the following identity:

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

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 Fibonacci numbers are a much more famous cousin of the Lucas numbers; the Lucas numbers follow the same recursive definition as the Fibonacci numbers, but start from a different pair of base cases.) Prove the following facts about the Lucas numbers, by induction (weak or strong, as appropriate) on  $n$ :

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

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

**5.59**  $(L_n)^2 = 5(f_n)^2 + 4(-1)^n$

(Hint: for Exercise 5.59, 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  $L_n L_{n-1}$  and  $f_n f_{n-1}$ , 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.) Prove the following facts about the Jacobsthal numbers by induction (weak or strong, as appropriate) on  $n$ :

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

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

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

The Fibonacci numbers are named after Leonardo of Pisa (also sometimes known as Leonardo Bonacci or just as Fibonacci), a 13th-century Italian mathematician.

The Lucas numbers and Jacobsthal numbers are named after Édouard Lucas, a 19th-century French mathematician, and Ernst Jacobsthal, a 20th-century German mathematician, respectively.

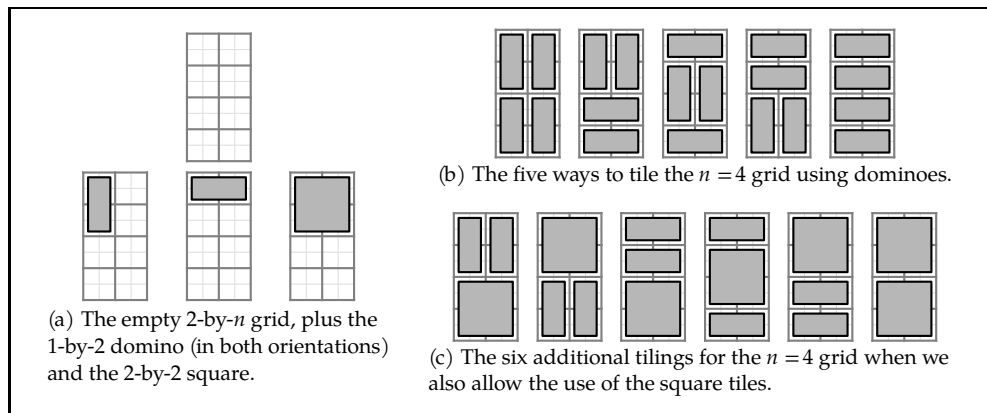


Figure 5.27: A tiling problem, using 1-by-2 dominoes and 2-by-2 squares.

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.

**5.63** 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 5.27.) 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.)

**5.64** Suppose that you run out of squares, so you can now only use dominoes for tiling. (See Figure 5.27(b).) How does your answer to the last 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. Here's the definition:

$$s_1 := 1 \qquad s_2 := 0 \qquad \text{for } n \geq 3, s_n := \underbrace{s_{n-1} \circ s_{n-2}}_{\text{the concatenation of } s_{n-1} \text{ and } s_{n-2}}.$$

For example, we have  $s_3 = s_2 \circ s_1 = 01$  and  $s_4 = s_3 \circ s_2 = 010$  and  $s_5 = s_4 \circ s_3 = 01001$  and  $s_6 = s_5 \circ s_4 = 01001010$ . 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). Here you'll prove a few slightly simpler properties, using strong induction on  $n$ :

**5.65** The number of bits in  $s_n$  is precisely  $f_n$  (the  $n$ th Fibonacci number).

**5.66** The string  $s_n$  does not contain two consecutive 1s or three consecutive 0s.

**5.67** 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 quantity  $\#0(s_n) - \#1(s_n)$  is a Fibonacci number.

**5.68** (programming required) The reason that  $s_n$  is called the “Fibonacci word 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  $i$ th symbol in  $s_n$ , for  $i = 1, 2, \dots, |s_n|$ : if the symbol is 1 then do not turn; if the symbol is a 0 and  $i$  is even, turn  $90^\circ$  to the right; and if the symbol is a 0 and  $i$  is odd, turn  $90^\circ$  to the left. In any case, proceed in your current direction by one unit. (See Figure 5.28.) Write a program to draw a bitstring using these rules; then implement the recursive definition of the Fibonacci word fractal and “draw” the strings  $s_1, s_2, \dots, s_{16}$ . (For efficiency's sake, you may want to compute  $s_n$  with a loop instead of recursively; see Figure 6.41 in Chapter 6 for some ideas.)

**5.69** The sum of the interior angles of any triangle is  $180^\circ$ . 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 5.29.)

**5.70** A *diagonal* of a polygon is a line that connects two non-adjacent vertices. (See Figure 5.29.) 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.)

**5.71** Prove that the recursive binary search algorithm shown in Figure 5.30 is correct. That is, prove that the following condition is true, by strong induction on  $n$ : For any sorted array  $A[1 \dots n]$ , **binarySearch**( $A, x$ ) returns true if and only if  $x \in A$ .

**5.72** Prove by weak induction on the quantity  $(n + m)$  that the **merge** algorithm in Figure 5.30 satisfies the following property for any  $n \geq 0$  and  $m \geq 0$ : given any two sorted arrays  $X[1 \dots n]$  and  $Y[1 \dots m]$  as input, the output of **merge**( $X, Y$ ) is a sorted array containing all elements of  $X$  and all elements of  $Y$ .

**5.73** Prove by strong induction on  $n$  that **mergeSort**( $A[1 \dots n]$ ), shown in Figure 5.30, indeed sorts its input.

**5.74** 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  $\{\langle 1, 2, 3 \rangle, \langle 1, 3, 2 \rangle, \langle 2, 1, 3 \rangle, \langle 2, 3, 1 \rangle, \langle 3, 1, 2 \rangle, \langle 3, 2, 1 \rangle\}$ , in some order.) Prove your algorithm correct by induction.

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  $P$ .)

**5.75** 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.

**5.76** 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.

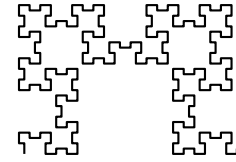


Figure 5.28: The Fibonacci word fractal  $s_{14}$ , visualized as in Exercise 5.68.

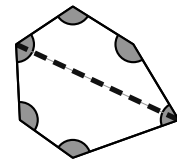


Figure 5.29: The interior angles and a diagonal for a polygon.

```

binarySearch( $A[1 \dots n], x$ ):
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( $A[1 \dots middle - 1], x$ )
8: else
9:   return binarySearch( $A[middle + 1 \dots n], x$ )

merge( $X[1 \dots n], Y[1 \dots m]$ ):
1: if  $n = 0$  then
2:   return  $Y$ 
3: else if  $m = 0$  then
4:   return  $X$ 
5: else if  $X[1] < Y[1]$  then
6:   return  $X[1]$  followed by merge( $X[2 \dots n], Y$ )
7: else
8:   return  $Y[1]$  followed by merge( $X, Y[2 \dots m]$ )

mergeSort( $A[1 \dots n]$ ):
1: if  $n = 1$  then
2:   return  $A$ 
3: else
4:    $L := \text{mergeSort}(A[1 \dots \lfloor \frac{n}{2} \rfloor])$ 
5:    $R := \text{mergeSort}(A[\lfloor \frac{n}{2} \rfloor + 1 \dots n])$ 
6:   return merge( $L, R$ )

```

Figure 5.30: Binary Search, Merge, and Merge Sort, recursively.

## 5.4 Recursively Defined Structures and Structural Induction

When a thing is done, it's done. Don't look back. Look forward to your next objective.

---

George C. Marshall (1880–1959)

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 5.31 for an example. (The slashed line in Figure 5.31 represents the null value.) Here is a recursive definition of a linked list:

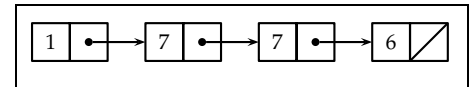


Figure 5.31: An example linked list.

#### 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 5.31 shows the linked list that consists of 1 followed by the linked list containing 7, 7, and 6 (which is a linked list consisting of 7 followed by a linked list containing 7 and 6, which is a linked list consisting of 7 followed by the linked list containing 6, which is ...). That is, Figure 5.31 shows the linked list  $\langle 1, \langle 7, \langle 7, \langle 6, \langle \rangle \rangle \rangle \rangle$ .

## BINARY TREES

We can also recursively define a *binary tree* (see Section 11.4.2). Again, deferring the discussion of why binary trees are useful to a course on data structures, 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 5.32 for an example; here the root node 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:

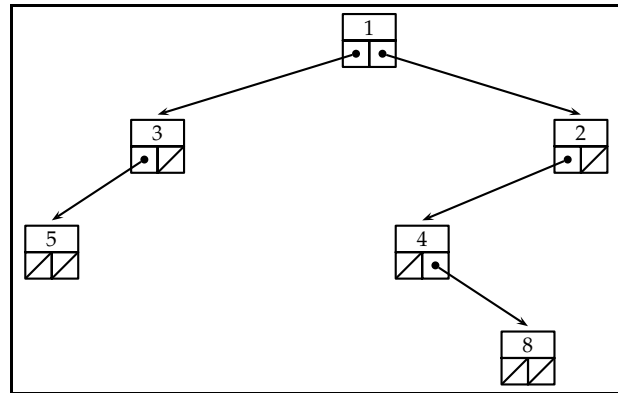


Figure 5.32: An example binary tree.

**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*  $T_\ell$ , and a *right subtree*  $T_r$ , where  $x$  is an arbitrary value and  $T_\ell$  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;
}
  
```

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. (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.)

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 5.32, 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. 529 (for heaps) and p. 1160 (for binary search trees) for more discussion.



## 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.  $\neg E$ , where  $E$  is an arithmetic expression; or
3.  $E \odot F$ , where  $E$  and  $F$  are arithmetic expressions and  $\odot \in \{+, -, \cdot, /\}$  is an *operator*.

**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  $f : X \rightarrow \{\text{True}, \text{False}\}$  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:

- the truth value of  $x \in X$  under  $f$  is  $f(x)$ ;
- 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. 543 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*  $T_\ell$ , and a *right subtree*  $T_r$ , where  $T_\ell$  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 establishes the claim.

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

**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  $leaves(B) \leq internals(B) + 1$ . We must prove  $P(n)$ .

We'll handle the case  $n = 1$  separately. (See Figure 5.33(a).) 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 5.33(b).) Observe that the tree  $T$  must have been generated by (a) generating a left subtree  $T_\ell$  using some number  $\ell$  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  $x$ ,  $T_\ell$ , 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  $T_\ell$  and  $T_r$ , and thus

$$leaves(T_\ell) \leq internals(T_\ell) + 1 \quad (1)$$

$$leaves(T_r) \leq internals(T_r) + 1. \quad (2)$$

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

$$leaves(T) = leaves(T_\ell) + leaves(T_r) \quad \text{by (3)}$$

$$\leq internals(T_\ell) + 1 + internals(T_r) + 1 \quad \text{by (1) and (2)}$$

$$= internals(T) + 1. \quad \text{by (4)}$$

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

An (abbreviated) reminder of the recursive definition of a binary tree:

**Rule #1:**  $null$  is a binary tree;

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

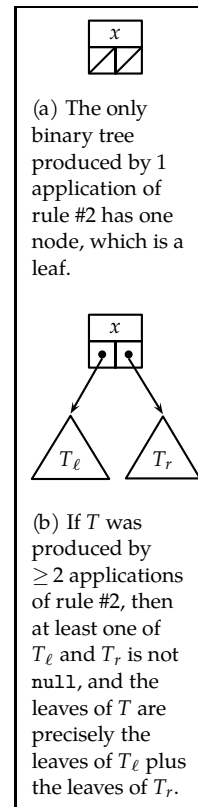


Figure 5.33: Illustrations of the inductive case for Example 5.19.

## 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. (More precisely, a set of objects is well ordered if there’s a “least” element among any collection of those objects.) 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-length 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:** 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 5.6 (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  $y_1, y_2, \dots, y_k \in S$  by an inductive case in the definition of  $S$ , prove that  $P(y_1) \wedge P(y_2) \wedge \dots \wedge P(y_k) \Rightarrow P(x)$ .

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  $\text{leaves}(T) \leq \text{internals}(T) + 1$ .*Proof.* Let  $P(T)$  denote the property that  $\text{leaves}(T) \leq \text{internals}(T) + 1$  for a binary tree  $T$ . We proceed by structural induction on the form of  $T$ .**base case** ( $T = \text{null}$ ): Then  $\text{leaves}(T) = \text{internals}(T) = 0$ , and indeed  $0 \leq 0 + 1$ .**inductive case** ( $T$  has root  $x$ , left subtree  $T_\ell$ , and right subtree  $T_r$ ): We assume the inductive hypotheses  $P(T_\ell)$  and  $P(T_r)$ , namely

$$\text{leaves}(T_\ell) \leq \text{internals}(T_\ell) + 1 \quad (1)$$

$$\text{leaves}(T_r) \leq \text{internals}(T_r) + 1. \quad (2)$$

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

$$\text{leaves}(T) = \text{leaves}(T_\ell) + \text{leaves}(T_r) \quad (3)$$

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

Putting together these facts, we have

$$\text{leaves}(T) = \text{leaves}(T_\ell) + \text{leaves}(T_r) \quad \text{by (3)}$$

$$\leq \text{internals}(T_\ell) + 1 + \text{internals}(T_r) + 1 \quad \text{by (1) and (2)}$$

$$= \text{internals}(T) + 1. \quad \text{by (4)}$$

Thus  $P(n)$  holds, which completes the proof.  $\square$ **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  $\neg$  AND  $\wedge$** First, we'll give a formal proof using structural induction of the claim that any propositional logic statement can be expressed using  $\neg$  and  $\wedge$  as the only logical connectives. (See Exercise 4.68.)**Example 5.21 (All of propositional logic using  $\neg$  and  $\wedge$ )****Claim:** For any logical proposition  $\varphi$  using the connectives  $\{\neg, \wedge, \vee, \Rightarrow\}$ , there exists a proposition using only  $\{\neg, \wedge\}$  that is logically equivalent to  $\varphi$ .

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

**base case:**  $\varphi$  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:**  $\varphi$  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:**  $\varphi$  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  $P_2 \equiv Q_2$ :

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

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  $\varphi$ , so the claim follows.  $\square$

**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: a well-formed formula is a variable, or the negation of a well-formed formula, or the conjunction of a pair of well-formed formulas ( $\text{wff} * \text{wff}$ ), 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. See Figure 5.34 for both the recursive definition of the `wff` datatype and the recursive function `simplify`, which takes an arbitrary `wff` as input, and produces a `wff` that uses only `And` and `Not` as output.

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

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)));
```

Figure 5.34: Well-formed formulas in ML.

## CONJUNCTIVE AND DISJUNCTIVE NORMAL FORMS

Here is another example of a proof by structural induction based on propositional logic, to establish Theorems 3.1 and 3.2, that any proposition is logically equivalent to one that's in conjunctive or disjunctive normal form.

(Recall that a proposition  $\varphi$  is in *conjunctive normal form* (CNF) if  $\varphi$  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  $\varphi$  is in *disjunctive normal form* (DNF) if  $\varphi$  is the disjunction of one or more *clauses*, where each clause is the conjunction of one or more literals.)

**Theorem 5.6 (CNF/DNF suffice)**

Let  $\varphi$  be a Boolean formula that uses the connectives  $\{\wedge, \vee, \neg, \Rightarrow\}$ . Then:

1. there exists  $\varphi_{\text{dnf}}$  in disjunctive normal form so that  $\varphi$  and  $\varphi_{\text{dnf}}$  are logically equivalent.
2. there exists  $\varphi_{\text{cnf}}$  in conjunctive normal form so that  $\varphi$  and  $\varphi_{\text{cnf}}$  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.

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  $\varphi$  contains only the connectives  $\{\neg, \wedge\}$ . Let  $C(\varphi)$  and  $D(\varphi)$ , respectively, denote the property that  $\varphi$  is logically equivalent to a CNF proposition and a DNF proposition, respectively. We now proceed by structural induction on the form of  $\varphi$ —which now can only be a variable, negation, or conjunction—to show that  $C(\varphi) \wedge D(\varphi)$  holds for any proposition  $\varphi$ .

**base case:**  $\varphi$  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:**  $\varphi$  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  $P_{\text{cnf}}$  and  $P_{\text{dnf}}$  such that  $P \equiv P_{\text{cnf}} \equiv P_{\text{dnf}}$ , where  $P_{\text{cnf}}$  is in CNF and  $P_{\text{dnf}}$  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  $P_{\text{cnf}}$  is of the form  $P_{\text{cnf}} = \bigwedge_{i=1}^n c_i$ , where  $c_i$  is a clause of the form  $c_i = \bigvee_{j=1}^{m_i} c_{ij}$ , where  $c_{ij}$  is a variable or its negation. Therefore we have

*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  $\varphi$  can be expressed in DNF (Theorem 5.6.1) by induction! But I do know how to prove that any proposition  $\varphi$  can be expressed in *both* DNF *and* CNF by induction, as is done in Example 5.22.

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

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  $P_{\text{dnf}}$  instead of  $P_{\text{cnf}}$ .

**inductive case II:  $\varphi$  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  $P_{\text{cnf}}$  and  $Q_{\text{cnf}}$  and DNF propositions  $P_{\text{dnf}}$  and  $Q_{\text{dnf}}$  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  $P_{\text{cnf}}$  and  $Q_{\text{cnf}}$  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  $P_{\text{dnf}}$  and  $Q_{\text{dnf}}$  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

$$\begin{aligned}
P \wedge Q &\equiv P_{\text{dnf}} \wedge Q \equiv \left( \bigvee_{i=1}^n c_i \right) \wedge Q && \text{inductive hypothesis } D(P) \text{ and definition of DNF} \\
&\equiv \bigvee_{i=1}^n (c_i \wedge Q) && \text{distributivity of } \vee \text{ over } \wedge \\
&\equiv \bigvee_{i=1}^n \left( c_i \wedge \bigvee_{j=1}^m d_j \right) && \text{inductive hypothesis } D(Q) \text{ and definition of DNF} \\
&\equiv \bigvee_{i=1}^n \bigvee_{j=1}^m (c_i \wedge d_j). && \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}$$

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.  $\square$



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:

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

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 5.7 (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 identical:

- they have precisely the same base case: prove  $P(0)$ ; and
- 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 5.35.)

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. 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 " $\rightarrow$ " means "can be rewritten as" and "|" means "or.")

$$\begin{array}{ll} S \rightarrow p \mid q \mid r & \text{S can be a propositional variable ...} \\ \mid \neg S & \text{... or the negation of a proposition ...} \\ \mid S \vee S \mid S \wedge S \mid S \Rightarrow S & \text{... or the } \wedge / \vee / \Rightarrow \text{ of two propositions.} \end{array}$$

An expression  $\varphi$  is a valid proposition over the variables  $\{p, q, r\}$  if and only if  $\varphi$  can be generated by a finite-length 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:

$$S \rightarrow S \vee S \rightarrow S \vee p \rightarrow \neg S \vee p \rightarrow \neg p \vee p.$$

(We used the rule  $S \rightarrow p$  twice, the rule  $S \rightarrow \neg S$  once, and the rule  $S \rightarrow S \vee S$  once.) The parse tree corresponding to this sequence of rule applications is shown in Figure 5.36(a).

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, shown in Figure 5.36(b):

$$S \rightarrow \neg S \rightarrow \neg S \vee S \rightarrow \neg p \vee S \rightarrow \neg p \vee p.$$

The parse tree in Figure 5.36(b) corresponds to  $\neg(p \vee p)$  instead of  $(\neg p) \vee p$ , which is the correct "order of operations" because  $\neg$  binds tighter than  $\vee$ .

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.<sup>3</sup>

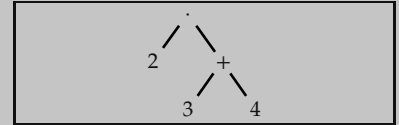


Figure 5.35: A parse tree for the arithmetic expression  $2 \cdot (3 + 4)$ .

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.

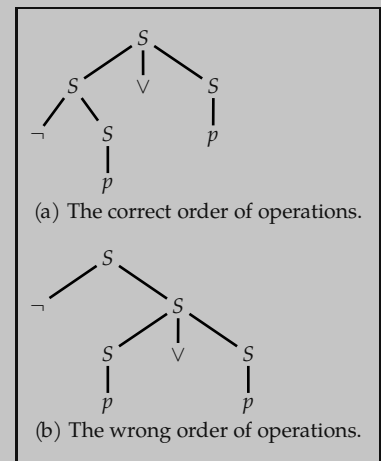


Figure 5.36: Two parse trees for  $\neg p \vee p$ .

More on context-free grammars and parsing, and their relationship to compilers and interpreters, can be found in books like

<sup>3</sup> Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Prentice Hall, 2nd edition, 2006; Dexter Kozen. *Automata and Computability*. Springer, 1997; and Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 3rd edition, 2012.

## 5.4.5 Exercises

**5.77** Let  $L$  be a linked list (as defined in Example 5.15). Prove by structural induction on  $L$  that **length**( $L$ ) returns the number of elements contained in  $L$ . (See Figure 5.37 for the algorithm.)

**5.78** Let  $L$  be a linked list containing integers. Prove by structural induction on  $L$  that **sum**( $L$ ) returns the sum of the numbers contained in  $L$ . (See Figure 5.37 for the algorithm.)

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

1.  $\langle x, \langle \rangle \rangle$ ; or
2.  $\langle x, \langle y, L \rangle \rangle$  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  $z \geq x$ .

A string of balanced parentheses (with a close parenthesis that matches every open parenthesis, and appears to its right) is one of the following:

1. the empty string (consisting of zero characters);
2. a string  $[ S ]$  where  $S$  is a string of balanced parentheses; or
3. a string  $S_1 S_2$  where  $S_1$  and  $S_2$  are both strings of balanced parentheses.

For example,  $[ [] ] []$  is a string of balanced parentheses, using Rule 3 on  $[ [] ]$  and  $[]$ . (Note that  $[]$  is a string of balanced parentheses using Rule 2 on the empty string (Rule 1), and therefore  $[ [] ]$  is by using Rule 2 on  $[]$ .)

**5.80** Prove by structural induction that every string of balanced parentheses according to this definition has exactly the same number of open parentheses as close parentheses.

**5.81** Prove by structural induction that any prefix of a string of balanced parentheses according to this definition has at least as many open parentheses as it does close parentheses.

**5.82** Recall from Definition 5.16 that we defined a *binary tree* as

1. an empty tree, denoted by **null**; or
2. a root node  $x$ , a left subtree  $T_\ell$ , and a right subtree  $T_r$ , where  $x$  is an arbitrary value and  $T_\ell$  and  $T_r$  are both binary trees.

Recall further that a *leaf* of a binary tree  $T$  is a node in  $T$  whose left subtree and right subtree are both **null**. Prove by structural induction that the algorithm **countLeaves**( $T$ ) in Figure 5.38 returns the number of leaves in a binary tree  $T$ .

**5.83** Recall that 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. 1160.) That is, a BST is either:

1. an empty tree, denoted by **null**; or
2. a root node  $x$ , a left subtree  $T_\ell$  where all elements are less than  $x$ , and a right subtree  $T_r$ , where all elements are greater than  $x$ , and  $T_\ell$  and  $T_r$  are both BSTs.

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

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

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**.)

**5.84** Give a recursive definition of a heap.

**5.85** 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.)

**5.86** 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.)

```
length(L): //assume L is a linked list.
1: if L = ⟨⟩ then
2:   return 0
3: else if L = ⟨x, L'⟩ then
4:   return 1 + length(L')
```

```
sum(L): //assume L is a linked list containing integers.
1: if L = ⟨⟩ then
2:   return 0
3: else if L = ⟨x, L'⟩ then
4:   return x + sum(L')
```

Figure 5.37: Two algorithms on linked lists.

```
countLeaves(T):
1: if T = null then
2:   return 0
3: else
4:   TL, TR := the left and right subtrees of T
5:   if TL = TR = null then
6:     return 1
7:   else
8:     return countLeaves(TL) + countLeaves(TR)
```

Figure 5.38: An algorithm to count leaves in a binary tree.

A 2–3 tree is a data structure, similar in spirit to a binary search tree (see Exercise 5.83)—or, more precisely, 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.

**5.87** Formally, a 2–3 tree of height  $h$  is one of the following:

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

Prove by structural induction that a 2–3 tree of height  $h$  has at least  $2^h$  leaves and at most  $3^h$  leaves. (Therefore a 2–3 tree that contains  $n$  leaf nodes has height between  $\log_3 n$  and  $\log_2 n$ .)

**5.88** 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 5.39 in your proofs:

**5.89** 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.

**5.90** 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.

**5.91** Let  $a_1, a_2, \dots, a_k$  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 5.89.)

In Chapter 2, 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 (Definition 2.5). 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  $n$ .

**5.92** Using the associativity/commutativity/identity/zero properties in Figure 5.39, 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.

**5.93** Using the facts in Figure 5.39 and Exercise 5.92, 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.

Recall Example 5.18, in which we defined a well-formed formula (a “wff”) of propositional logic as a variable; the negation ( $\neg$ ) of a wff; or the conjunction/disjunction/implication ( $\wedge$ ,  $\vee$ , and  $\Rightarrow$ ) of two wffs. Assuming we allow the corresponding new connective in the following exercises as part of a wff, give a proof using structural induction (see Example 5.21 for an example) that any wff is logically equivalent to one using only  $\neg$  and  $\wedge$ .

**5.94** Sheffer stroke  $|$ , where  $p | q \equiv \neg(p \wedge q)$  **5.95** Peirce’s arrow  $\downarrow$ , where  $p \downarrow q \equiv \neg(p \vee q)$   
(programming required) In the programming language ML (see Figure 5.34 for more), write a program to translate an arbitrary statement of propositional logic into a logically equivalent statement that has the following special form. (In other words, implement the proof of Exercises 5.94 and 5.95 as a recursive function.)

**5.96**  $|$  is the only logical connective

**5.97**  $\downarrow$  is the only logical connective

**5.98** 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.)

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

**5.100** Let  $\#0(s)$  and  $\#1(s)$  denote the number of 0s and 1s in a bitstring  $s$ , respectively. Using your recursive definition from the previous exercise, prove by structural induction that, for any palindromic bitstring  $s$ , the value of  $\#0(s) \cdot \#1(s)$  is an even number.

$(a + b) + c = a + (b + c)$	Associativity of Addition
$a + b = b + a$	Commutativity of Addition
$a + 0 = 0 + a = a$	Additive Identity
$(a \cdot b) \cdot c = a \cdot (b \cdot c)$	Associativity of Multiplication
$a \cdot b = b \cdot a$	Commutativity of Multiplication
$a \cdot 1 = 1 \cdot a = a$	Multiplicative Identity
$a \cdot 0 = 0 \cdot a = 0$	Multiplicative Zero

Figure 5.39: A few elementary-school facts about addition and multiplication.

## 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 on 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:

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

(If  $\alpha = 1$ , then  $\sum_{i=0}^n \alpha^i = n+1$ .) 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  $[P(0) \wedge P(1) \dots \wedge P(n-1)] \Rightarrow P(n)$ . 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  $p_1, p_2, \dots, p_k$  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  $y_1, y_2, \dots, y_k \in S$  by an inductive case in the definition of  $S$ , prove that  $P(y_1) \wedge P(y_2) \dots \wedge P(y_k) \Rightarrow P(x)$ .

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  $1 + 2 + \dots + n = \frac{n(n+1)}{2}$ .

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

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

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

#### 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  $[P(0) \wedge P(1) \dots \wedge P(n-1)] \Rightarrow P(n)$ .

2. The prime factorization theorem: let  $n \in \mathbb{Z}^{\geq 1}$  be a positive integer. Then there exist  $k \geq 0$  prime numbers  $p_1, p_2, \dots, p_k$  such that  $n = \prod_{i=1}^k p_i$ . Furthermore, up to reordering, the prime numbers  $p_1, p_2, \dots, p_k$  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  $y_1, y_2, \dots, y_k \in S$  by an inductive case in the definition of  $S$ , we have that  $P(y_1) \wedge P(y_2) \dots \wedge P(y_k) \Rightarrow P(x)$ .