3   Induction

Now that you understand the basics of how to prove that a proposition is true, it is time to equip you with the most powerful methods we have for establishing truth: the Well Ordering Principle, the Induction Rule, and Strong Induction. These methods are especially useful when you need to prove that a predicate is true for all natural numbers.

Although the three methods look and feel different, it turns out that they are equivalent in the sense that a proof using any one of the methods can be automatically reformatted so that it becomes a proof using any of the other methods. The choice of which method to use is up to you and typically depends on whichever seems to be the easiest or most natural for the problem at hand.

3.1   The Well Ordering Principle

> Every *nonempty* set of *nonnegative integers* has a *smallest* element.

This statement is known as The *Well Ordering Principle*. Do you believe it? Seems sort of obvious, right? But notice how tight it is: it requires a *nonempty* set—it's false for the empty set which has *no* smallest element because it has no elements at all! And it requires a set of *nonnegative* integers—it's false for the set of *negative* integers and also false for some sets of nonnegative *rationals*—for example, the set of positive rationals. So, the Well Ordering Principle captures something special about the nonnegative integers.

3.1.1   Well Ordering Proofs

While the Well Ordering Principle may seem obvious, it's hard to see offhand why it is useful. But in fact, it provides one of the most important proof rules in discrete mathematics.

In fact, looking back, we took the Well Ordering Principle for granted in proving that $\sqrt{2}$ is irrational. That proof assumed that for any positive integers m and n, the fraction m/n can be written in *lowest terms*, that is, in the form m'/n' where m' and n' are positive integers with no common factors. How do we know this is always possible?

Suppose to the contrary[1] that there were $m, n \in \mathbb{Z}^+$ such that the fraction m/n cannot be written in lowest terms. Now let C be the set of positive integers that are numerators of such fractions. Then $m \in C$, so C is nonempty. Therefore, by Well Ordering, there must be a smallest integer, $m_0 \in C$. So by definition of C, there is an integer $n_0 > 0$ such that

$$\text{the fraction } \frac{m_0}{n_0} \text{ cannot be written in lowest terms.}$$

This means that m_0 and n_0 must have a common factor, $p > 1$. But

$$\frac{m_0/p}{n_0/p} = \frac{m_0}{n_0},$$

so any way of expressing the left hand fraction in lowest terms would also work for m_0/n_0, which implies

$$\text{the fraction } \frac{m_0/p}{n_0/p} \text{ cannot be in written in lowest terms either.}$$

So by definition of C, the numerator, m_0/p, is in C. But $m_0/p < m_0$, which contradicts the fact that m_0 is the smallest element of C.

Since the assumption that C is nonempty leads to a contradiction, it follows that C must be empty. That is, that there are no numerators of fractions that can't be written in lowest terms, and hence there are no such fractions at all.

We've been using the Well Ordering Principle on the sly from early on!

3.1.2   Template for Well Ordering Proofs

More generally, to prove that "$P(n)$ is true for all $n \in \mathbb{N}$" using the Well Ordering Principle, you can take the following steps:

- Define the set, C, of *counterexamples* to P being true. Namely, define[2]

$$C ::= \{n \in \mathbb{N} \mid P(n) \text{ is false}\}.$$

- Use a proof by contradiction and assume that C is nonempty.

- By the Well Ordering Principle, there will be a smallest element, n, in C.

- Reach a contradiction (somehow)—often by showing how to use n to find another member of C that is smaller than n. (This is the open-ended part of the proof task.)

- Conclude that C must be empty, that is, no counterexamples exist. QED

---

[1] This means that you are about to see an informal proof by contradiction.

[2] As we learned in Section 2.6.2, the notation $\{n \mid P(n) \text{ is false}\}$ means "the set of all elements n, for which $P(n)$ is false."

3.1. The Well Ordering Principle

3.1.3   Examples

Let's use this this template to prove

Theorem 3.1.1.
$$1 + 2 + 3 + \cdots + n = n(n+1)/2 \tag{3.1}$$

for all nonnegative integers, n.

First, we better address of a couple of ambiguous special cases before they trip us up:

- If $n = 1$, then there is only one term in the summation, and so $1 + 2 + 3 + \cdots + n$ is just the term 1. Don't be misled by the appearance of 2 and 3 and the suggestion that 1 and n are distinct terms!

- If $n \leq 0$, then there are no terms at all in the summation. By convention, the sum in this case is 0.

So while the dots notation is convenient, you have to watch out for these special cases where the notation is misleading! (In fact, whenever you see the dots, you should be on the lookout to be sure you understand the pattern, watching out for the beginning and the end.)

We could have eliminated the need for guessing by rewriting the left side of (3.1) with *summation notation*:

$$\sum_{i=1}^{n} i \qquad \text{or} \qquad \sum_{1 \leq i \leq n} i.$$

Both of these expressions denote the sum of all values taken by the expression to the right of the sigma as the variable, i, ranges from 1 to n. Both expressions make it clear what (3.1) means when $n = 1$. The second expression makes it clear that when $n = 0$, there are no terms in the sum, though you still have to know the convention that a sum of no numbers equals 0 (the *product* of no numbers is 1, by the way).

OK, back to the proof:

Proof. By contradiction and use of the Well Ordering Principle. Assume that the theorem is *false*. Then, some nonnegative integers serve as *counterexamples* to it. Let's collect them in a set:

$$C ::= \{n \in \mathbb{N} \mid 1 + 2 + 3 + \cdots + n \neq \frac{n(n+1)}{2}\}.$$

By our assumption that the theorem admits counterexamples, C is a nonempty set of nonnegative integers. So, by the Well Ordering Principle, C has a minimum element, call it c. That is, c is the *smallest counterexample* to the theorem.

Since c is the smallest counterexample, we know that (3.1) is false for $n = c$ but true for all nonnegative integers $n < c$. But (3.1) is true for $n = 0$, so $c > 0$. This means $c - 1$ is a nonnegative integer, and since it is less than c, equation (3.1) is true for $c - 1$. That is,

$$1 + 2 + 3 + \cdots + (c - 1) = \frac{(c - 1)c}{2}.$$

But then, adding c to both sides we get

$$1 + 2 + 3 + \cdots + (c - 1) + c = \frac{(c - 1)c}{2} + c = \frac{c^2 - c + 2c}{2} = \frac{c(c + 1)}{2},$$

which means that (3.1) does hold for c, after all! This is a contradiction, and we are done. ∎

Here is another result that can be proved using Well Ordering. It will be useful in Chapter 4 when we study number theory and cryptography.

Theorem 3.1.2. *Every natural number can be factored as a product of primes.*

Proof. By contradiction and Well Ordering. Assume that the theorem is false and let C be the set of all integers greater than one that cannot be factored as a product of primes. We assume that C is not empty and derive a contradiction.

If C is not empty, there is a least element, $n \in C$, by Well Ordering. The n can't be prime, because a prime by itself is considered a (length one) product of primes and no such products are in C.

So n must be a product of two integers a and b where $1 < a, b < n$. Since a and b are smaller than the smallest element in C, we know that $a, b \notin C$. In other words, a can be written as a product of primes $p_1 p_2 \cdots p_k$ and b as a product of primes $q_1 \cdots q_l$. Therefore, $n = p_1 \cdots p_k q_1 \cdots q_l$ can be written as a product of primes, contradicting the claim that $n \in C$. Our assumption that C is not empty must therefore be false. ∎

3.2   Ordinary Induction

Induction is by far the most powerful and commonly-used proof technique in discrete mathematics and computer science. In fact, the use of induction is a defining

characteristic of *discrete*—as opposed to *continuous*—mathematics. To understand how it works, suppose there is a professor who brings to class a bottomless bag of assorted miniature candy bars. She offers to share the candy in the following way. First, she lines the students up in order. Next she states two rules:

1. The student at the beginning of the line gets a candy bar.

2. If a student gets a candy bar, then the following student in line also gets a candy bar.

Let's number the students by their order in line, starting the count with 0, as usual in computer science. Now we can understand the second rule as a short description of a whole sequence of statements:

- If student 0 gets a candy bar, then student 1 also gets one.

- If student 1 gets a candy bar, then student 2 also gets one.

- If student 2 gets a candy bar, then student 3 also gets one.

$$\vdots$$

Of course this sequence has a more concise mathematical description:

If student n gets a candy bar, then student $n + 1$ gets a candy bar, for all nonnegative integers n.

So suppose you are student 17. By these rules, are you entitled to a miniature candy bar? Well, student 0 gets a candy bar by the first rule. Therefore, by the second rule, student 1 also gets one, which means student 2 gets one, which means student 3 gets one as well, and so on. By 17 applications of the professor's second rule, you get your candy bar! Of course the rules actually guarantee a candy bar to *every* student, no matter how far back in line they may be.

3.2.1   A Rule for Ordinary Induction

The reasoning that led us to conclude that every student gets a candy bar is essentially all there is to induction.

The Principle of Induction.

Let $P(n)$ be a predicate. If

- $P(0)$ is true, and

- $P(n)$ IMPLIES $P(n + 1)$ for all nonnegative integers, n,

then

- $P(m)$ is true for all nonnegative integers, m.

Since we're going to consider several useful variants of induction in later sections, we'll refer to the induction method described above as *ordinary induction* when we need to distinguish it. Formulated as a proof rule, this would be

Rule. *Induction Rule*

$$\frac{P(0), \quad \forall n \in \mathbb{N}.\ P(n)\ \text{IMPLIES}\ P(n + 1)}{\forall m \in \mathbb{N}.\ P(m)}$$

This general induction rule works for the same intuitive reason that all the students get candy bars, and we hope the explanation using candy bars makes it clear why the soundness of the ordinary induction can be taken for granted. In fact, the rule is so obvious that it's hard to see what more basic principle could be used to justify it.[3] What's not so obvious is how much mileage we get by using it.

3.2.2   A Familiar Example

Ordinary induction often works directly in proving that some statement about nonnegative integers holds for all of them. For example, here is the formula for the sum of the nonnegative integers that we already proved (equation (3.1)) using the Well Ordering Principle:

Theorem 3.2.1. *For all $n \in \mathbb{N}$,*

$$1 + 2 + 3 + \cdots + n = \frac{n(n + 1)}{2} \tag{3.2}$$

This time, let's use the Induction Principle to prove Theorem 3.2.1.

Suppose that we define predicate $P(n)$ to be the equation (3.2). Recast in terms of this predicate, the theorem claims that $P(n)$ is true for all $n \in \mathbb{N}$. This is great, because the induction principle lets us reach precisely that conclusion, provided we establish two simpler facts:

---

[3]But see section 3.2.7.

- $P(0)$ is true.

- For all $n \in \mathbb{N}$, $P(n)$ IMPLIES $P(n + 1)$.

So now our job is reduced to proving these two statements. The first is true because $P(0)$ asserts that a sum of zero terms is equal to $0(0 + 1)/2 = 0$, which is true by definition. The second statement is more complicated. But remember the basic plan for proving the validity of any implication from Section 2.3: *assume* the statement on the left and then *prove* the statement on the right. In this case, we assume $P(n)$ in order to prove $P(n + 1)$, which is the equation

$$1 + 2 + 3 + \cdots + n + (n + 1) = \frac{(n + 1)(n + 2)}{2}. \tag{3.3}$$

These two equations are quite similar; in fact, adding $(n + 1)$ to both sides of equation (3.2) and simplifying the right side gives the equation (3.3):

$$1 + 2 + 3 + \cdots + n + (n + 1) = \frac{n(n + 1)}{2} + (n + 1)$$
$$= \frac{(n + 2)(n + 1)}{2}$$

Thus, if $P(n)$ is true, then so is $P(n + 1)$. This argument is valid for every non-negative integer n, so this establishes the second fact required by the induction principle. Therefore, the induction principle says that the predicate $P(m)$ is true for all nonnegative integers, m, so the theorem is proved.

3.2.3 A Template for Induction Proofs

The proof of Theorem 3.2.1 was relatively simple, but even the most complicated induction proof follows exactly the same template. There are five components:

1. **State that the proof uses induction.** This immediately conveys the overall structure of the proof, which helps the reader understand your argument.

2. **Define an appropriate predicate $P(n)$.** The eventual conclusion of the induction argument will be that $P(n)$ is true for all nonnegative n. Thus, you should define the predicate $P(n)$ so that your theorem is equivalent to (or follows from) this conclusion. Often the predicate can be lifted straight from the proposition that you are trying to prove, as in the example above. The predicate $P(n)$ is called the *induction hypothesis*. Sometimes the induction hypothesis will involve several variables, in which case you should indicate which variable serves as n.

Chapter 3   Induction

3. **Prove that** $P(0)$ **is true.** This is usually easy, as in the example above. This part of the proof is called the *base case* or *basis step*.

4. **Prove that** $P(n)$ **implies** $P(n + 1)$ **for every nonnegative integer** n**.** This is called the *inductive step*. The basic plan is always the same: assume that $P(n)$ is true and then use this assumption to prove that $P(n+1)$ is true. These two statements should be fairly similar, but bridging the gap may require some ingenuity. Whatever argument you give must be valid for every nonnegative integer n, since the goal is to prove the implications $P(0) \to P(1)$, $P(1) \to P(2)$, $P(2) \to P(3)$, etc. all at once.

5. **Invoke induction.** Given these facts, the induction principle allows you to conclude that $P(n)$ is true for all nonnegative n. This is the logical capstone to the whole argument, but it is so standard that it's usual not to mention it explicitly.

Always be sure to explicitly label the *base case* and the *inductive step*. It will make your proofs clearer, and it will decrease the chance that you forget a key step (such as checking the base case).

3.2.4   A Clean Writeup

The proof of Theorem 3.2.1 given above is perfectly valid; however, it contains a lot of extraneous explanation that you won't usually see in induction proofs. The writeup below is closer to what you might see in print and should be prepared to produce yourself.

Proof of Theorem 3.2.1. We use induction. The induction hypothesis, $P(n)$, will be equation (3.2).

Base case: $P(0)$ is true, because both sides of equation (3.2) equal zero when $n = 0$.

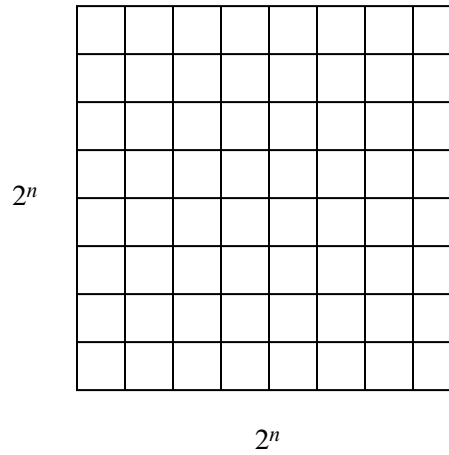**Inductive step:** Assume that $P(n)$ is true, where n is any nonnegative integer. Then

$$1 + 2 + 3 + \cdots + n + (n + 1) = \frac{n(n + 1)}{2} + (n + 1) \quad \text{(by induction hypothesis)}$$
$$= \frac{(n + 1)(n + 2)}{2} \qquad \text{(by simple algebra)}$$

which proves $P(n + 1)$.

So it follows by induction that $P(n)$ is true for all nonnegative n. ∎

Figure 3.1   A $2^n \times 2^n$ courtyard for $n = 3$.

Induction was helpful for *proving the correctness* of this summation formula, but not helpful for *discovering* it in the first place. Tricks and methods for finding such formulas will be covered in Part III of the text.

3.2.5   A More Challenging Example

During the development of MIT's famous Stata Center, as costs rose further and further beyond budget, there were some radical fundraising ideas. One rumored plan was to install a big courtyard with dimensions $2^n \times 2^n$ (as shown in Figure 3.1 for the case where $n = 3$) and to have one of the central squares[4] be occupied by a statue of a wealthy potential donor (who we will refer to as "Bill", for the purposes of preserving anonymity). A complication was that the building's unconventional architect, Frank Gehry, was alleged to require that only special L-shaped tiles (show in Figure 3.2) be used for the courtyard. It was quickly determined that a courtyard meeting these constraints exists, at least for $n = 2$. (See Figure 3.3.) But what about for larger values of n? Is there a way to tile a $2^n \times 2^n$ courtyard with L-shaped tiles around a statue in the center? Let's try to prove that this is so.

Theorem 3.2.2. *For all $n \geq 0$ there exists a tiling of a $2^n \times 2^n$ courtyard with Bill in a central square.*
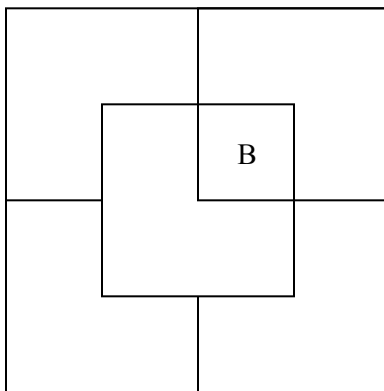
Proof. (doomed attempt) The proof is by induction. Let $P(n)$ be the proposition that there exists a tiling of a $2^n \times 2^n$ courtyard with Bill in the center.

---

[4]In the special case $n = 0$, the whole courtyard consists of a single central square; otherwise, there are four central squares.
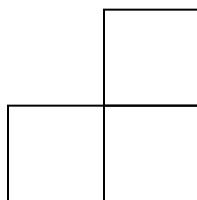
Figure 3.2   The special L-shaped tile.

Figure 3.3   A tiling using L-shaped tiles for $n = 2$ with Bill in a center square.

Base case: $P(0)$ is true because Bill fills the whole courtyard.

Inductive step: Assume that there is a tiling of a $2^n \times 2^n$ courtyard with Bill in the center for some $n \geq 0$. We must prove that there is a way to tile a $2^{n+1} \times 2^{n+1}$ courtyard with Bill in the center .... ∎

Now we're in trouble! The ability to tile a smaller courtyard with Bill in the center isn't much help in tiling a larger courtyard with Bill in the center. We haven't figured out how to bridge the gap between $P(n)$ and $P(n+1)$.

So if we're going to prove Theorem 3.2.2 by induction, we're going to need some *other* induction hypothesis than simply the statement about *n* that we're trying to prove.

When this happens, your first fallback should be to look for a *stronger* induction hypothesis; that is, one which implies your previous hypothesis. For example, we could make $P(n)$ the proposition that for *every* location of Bill in a $2^n \times 2^n$ courtyard, there exists a tiling of the remainder.

This advice may sound bizarre: "If you can't prove something, try to prove something grander!" But for induction arguments, this makes sense. In the inductive step, where you have to prove $P(n)$ IMPLIES $P(n+1)$, you're in better shape because you can *assume* $P(n)$, which is now a more powerful statement. Let's see how this plays out in the case of courtyard tiling.

Proof (successful attempt). The proof is by induction. Let $P(n)$ be the proposition that for every location of Bill in a $2^n \times 2^n$ courtyard, there exists a tiling of the remainder.

**Base case:** $P(0)$ is true because Bill fills the whole courtyard.

Inductive step: Assume that $P(n)$ is true for some $n \geq 0$; that is, for every location of Bill in a $2^n \times 2^n$ courtyard, there exists a tiling of the remainder. Divide the $2^{n+1} \times 2^{n+1}$ courtyard into four quadrants, each $2^n \times 2^n$. One quadrant contains Bill (**B** in the diagram below). Place a temporary Bill (**X** in the diagram) in each of the three central squares lying outside this quadrant as shown in Figure 3.4.

Now we can tile each of the four quadrants by the induction assumption. Replacing the three temporary Bills with a single L-shaped tile completes the job. This proves that $P(n)$ implies $P(n+1)$ for all $n \geq 0$. Thus $P(m)$ is true for all $n \in \mathbb{N}$, and the theorem follows as a special case where we put Bill in a central square. ∎

This proof has two nice properties. First, not only does the argument guarantee that a tiling exists, but also it gives an algorithm for finding such a tiling. Second, we have a stronger result: if Bill wanted a statue on the edge of the courtyard, away from the pigeons, we could accommodate him!

Strengthening the induction hypothesis is often a good move when an induction proof won't go through. But keep in mind that the stronger assertion must actually
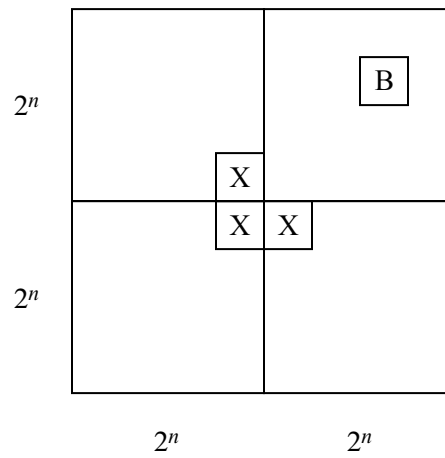
Figure 3.4   Using a stronger inductive hypothesis to prove Theorem 3.2.2.

be *true*; otherwise, there isn't much hope of constructing a valid proof! Sometimes finding just the right induction hypothesis requires trial, error, and insight. For example, mathematicians spent almost twenty years trying to prove or disprove the conjecture that "Every planar graph is 5-choosable"[5]. Then, in 1994, Carsten Thomassen gave an induction proof simple enough to explain on a napkin. The key turned out to be finding an extremely clever induction hypothesis; with that in hand, completing the argument was easy!

3.2.6   A Faulty Induction Proof

If we have done a good job in writing this text, right about now you should be thinking, "Hey, this induction stuff isn't so hard after all—just show $P(0)$ is true and that $P(n)$ implies $P(n + 1)$ for any number n." And, you would be right, although sometimes when you start doing induction proofs on your own, you can run into trouble. For example, we will now attempt to ruin your day by using induction to "prove" that all horses are the same color. And just when you thought it was safe to skip class and work on your robot program instead. Bummer!

False Theorem. *All horses are the same color.*

Notice that no n is mentioned in this assertion, so we're going to have to reformulate it in a way that makes an n explicit. In particular, we'll (falsely) prove that

---

[5]5-choosability is a slight generalization of 5-colorability. Although every planar graph is 4-colorable and therefore 5-colorable, not every planar graph is 4-choosable. If this all sounds like nonsense, don't panic. We'll discuss graphs, planarity, and coloring in Part II of the text.

3.2. Ordinary Induction

False Theorem 3.2.3. *In every set of $n \geq 1$ horses, all the horses are the same color.*

This a statement about all integers $n \geq 1$ rather ≥ 0, so it's natural to use a slight variation on induction: prove $P(1)$ in the base case and then prove that $P(n)$ implies $P(n+1)$ for all $n \geq 1$ in the inductive step. This is a perfectly valid variant of induction and is *not* the problem with the proof below.

Bogus proof. The proof is by induction on n. The induction hypothesis, $P(n)$, will be

$$\text{In every set of } n \text{ horses, all are the same color.} \tag{3.4}$$

Base case: ($n = 1$). $P(1)$ is true, because in a set of horses of size 1, there's only one horse, and this horse is definitely the same color as itself.

Inductive step: Assume that $P(n)$ is true for some $n \geq 1$. That is, assume that in every set of n horses, all are the same color. Now consider a set of $n+1$ horses:

$$h_1, h_2, \ldots, h_n, h_{n+1}$$

By our assumption, the first n horses are the same color:

$$\underbrace{h_1, h_2, \ldots, h_n}_{\text{same color}}, h_{n+1}$$

Also by our assumption, the last n horses are the same color:

$$h_1, \underbrace{h_2, \ldots, h_n, h_{n+1}}_{\text{same color}}$$

So h_1 is the same color as the remaining horses besides h_{n+1}—that is, h_2, ..., h_n)—and likewise h_{n+1} is the same color as the remaining horses besides h_1–h_2, ..., h_n. Since h_1 and h_{n+1} are the same color as h_2, ..., h_n, horses h_1, h_2, ..., h_{n+1} must all be the same color, and so $P(n+1)$ is true. Thus, $P(n)$ implies $P(n+1)$.

By the principle of induction, $P(n)$ is true for all $n \geq 1$. ∎

We've proved something false! Is math broken? Should we all become poets? No, this proof has a mistake.

The first error in this argument is in the sentence that begins "So h_1 is the same color as the remaining horses besides h_{n+1}—$h_2, \ldots, h_n$)..."

The "..." notation in the expression "h_1, h_2, ..., h_n, h_{n+1}" creates the impression that there are some remaining horses (namely h_2, ..., h_n) besides h_1 and h_{n+1}. However, this is not true when $n = 1$. In that case, $h_1, h_2, \ldots, h_n, h_{n+1} =$

h_1, h_2 and there are no remaining horses besides h_1 and h_{n+1}. So h_1 and h_2 need not be the same color!

This mistake knocks a critical link out of our induction argument. We proved $P(1)$ and we *correctly* proved $P(2) \longrightarrow P(3)$, $P(3) \longrightarrow P(4)$, etc. But we failed to prove $P(1) \longrightarrow P(2)$, and so everything falls apart: we can not conclude that $P(2)$, $P(3)$, etc., are true. And, of course, these propositions are all false; there are sets of n non-uniformly-colored horses for all $n \geq 2$.

Students sometimes claim that the mistake in the proof is because $P(n)$ is false for $n \geq 2$, and the proof assumes something false, namely, $P(n)$, in order to prove $P(n + 1)$. You should think about how to explain to such a student why this claim would get no credit on a Math for Computer Science exam.

3.2.7   Induction versus Well Ordering

The Induction Rule looks nothing like the Well Ordering Principle, but these two proof methods are closely related. In fact, as the examples above suggest, we can take any Well Ordering proof and reformat it into an Induction proof. Conversely, it's equally easy to take any Induction proof and reformat it into a Well Ordering proof.

So what's the difference? Well, sometimes induction proofs are clearer because they resemble recursive procedures that reduce handling an input of size $n + 1$ to handling one of size n. On the other hand, Well Ordering proofs sometimes seem more natural, and also come out slightly shorter. The choice of method is really a matter of style and is up to you.

3.3   Invariants

One of the most important uses of induction in computer science involves proving that a program or process preserves one or more desirable properties as it proceeds. A property that is preserved through a series of operations or steps is known as an *invariant*. Examples of desirable invariants include properties such as a variable never exceeding a certain value, the altitude of a plane never dropping below 1,000 feet without the wingflaps and landing gear being deployed, and the temperature of a nuclear reactor never exceeding the threshold for a meltdown.

We typically use induction to prove that a proposition is an invariant. In particular, we show that the proposition is true at the beginning (this is the base case) and that if it is true after t steps have been taken, it will also be true after step $t + 1$ (this is the inductive step). We can then use the induction principle to conclude that the

proposition is indeed an invariant, namely, that it will always hold.

3.3.1   A Simple Example: The Diagonally-Moving Robot

Invariants are useful in systems that have a *start state* (or *starting configuration*) and a well-defined series of *steps* during which the system can change state.[6] For example, suppose that you have a robot that can walk across diagonals on an infinite 2-dimensional grid. The robot starts at position $(0, 0)$ and at each step it moves up or down by 1 unit vertically and left or right by 1 unit horizontally. To be clear, the robot must move by exactly 1 unit in each dimension during each step, since it can only traverse diagonals.

In this example, the *state* of the robot at any time can be specified by a coordinate pair (x, y) that denotes the robot's position. The *start state* is $(0, 0)$ since it is given that the robot starts at that position. After the first step, the robot could be in states $(1, 1)$, $(1, -1)$, $(-1, 1)$, or $(-1, -1)$. After two steps, there are 9 possible states for the robot, including $(0, 0)$.

Can the robot ever reach position $(1, 0)$?

After playing around with the robot for a bit, it will become apparent that the robot will never be able to reach position $(1, 0)$. This is because the robot can only reach positions (x, y) for which $x + y$ is even. This crucial observation quickly leads to the formulation of a predicate

$$P(t) :: \text{if the robot is in state } (x, y) \text{ after } t \text{ steps, then } x + y \text{ is even}$$

which we can prove to be an invariant by induction.

Theorem 3.3.1. *The sum of robot's coordinates is always even.*

Proof. We will prove that P is an invariant by induction.

$P(0)$ is true since the robot starts at $(0, 0)$ and $0 + 0$ is even.

Assume that $P(t)$ is true for the inductive step. Let (x, y) be the position of the robot after t steps. Since $P(t)$ is assumed to be true, we know that $x + y$ is even. There are four cases to consider for step $t + 1$, depending on which direction the robot moves.

Case 1 The robot moves to $(x + 1, y + 1)$. Then the sum of the coordinates is $x + y + 2$, which is even, and so $P(t + 1)$ is true.

Case 2 The robot moves to $(x + 1, y - 1)$. The the sum of the coordinates is $x + y$, which is even, and so $P(t + 1)$ is true.

---

[6]Such systems are known as state machines and we will study them in greater detail in Chapter 8.

Chapter 3   Induction

Case 3 The robot moves to $(x - 1, y + 1)$. The the sum of the coordinates is $x + y$, as with Case 2, and so $P(t + 1)$ is true.

Case 4 The robot moves to $(x - 1, y - 1)$. The the sum of the coordinates is $x + y - 2$, which is even, and so $P(t + 1)$ is true.

In every case, $P(t + 1)$ is true and so we have proved $P(t)$ IMPLIES $P(t + 1)$ and so, by induction, we know that $P(t)$ is true for all $t \geq 0$. ■

Corollary 3.3.2. *The robot can never reach position* $(1, 0)$.

Proof. By Theorem 3.3.1, we know the robot can only reach positions with coordinates that sum to an even number, and thus it cannot reach position $(1, 0)$. ■

Since this was the first time we proved that a predicate was an invariant, we were careful to go through all four cases in gory detail. As you become more experienced with such proofs, you will likely become more brief as well. Indeed, if we were going through the proof again at a later point in the text, we might simply note that the sum of the coordinates after step $t + 1$ can be only $x + y$, $x + y + 2$ or $x + y - 2$ and therefore that the sum is even.

3.3.2   The Invariant Method

In summary, if you would like to prove that some property NICE holds for every step of a process, then it is often helpful to use the following method:

- Define $P(t)$ to be the predicate that NICE holds immediately after step t.

- Show that $P(0)$ is true, namely that NICE holds for the start state.

- Show that
$$\forall t \in \mathbb{N}. \ P(t) \text{ IMPLIES } P(t + 1),$$
  namely, that for any $t \geq 0$, if NICE holds immediately after step t, it must also hold after the following step.

3.3.3   A More Challenging Example: The 15-Puzzle

In the late 19th century, Noyes Chapman, a postmaster in Canastota, New York, invented the 15-puzzle[7], which consisted of a 4×4 grid containing 15 numbered blocks in which the 14-block and the 15-block were out of order. The objective was to move the blocks one at a time into an adjacent hole in the grid so as to eventually
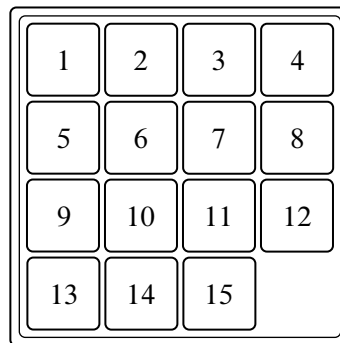
---

[7]Actually, there is a dispute about who really invented the 15-puzzle. Sam Lloyd, a well-known puzzle designer, claimed to be the inventor, but this claim has since been discounted.

3.3. Invariants

(a)   (b)
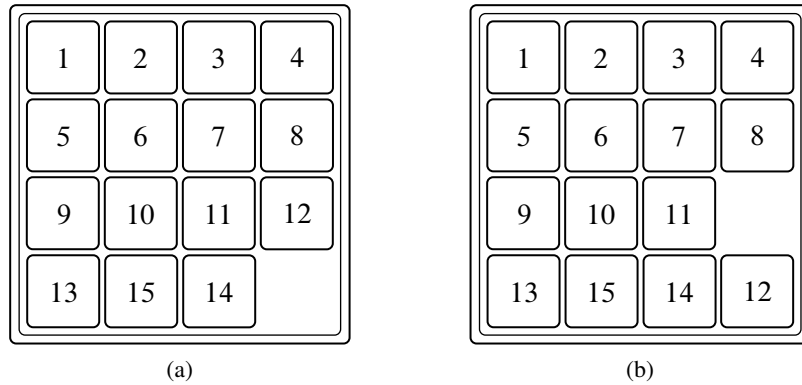
Figure 3.5   The 15-puzzle in its starting configuration (a) and after the 12-block is moved into the hole below (b).


Figure 3.6   The desired final configuration for the 15-puzzle. Can it be achieved by only moving one block at a time into an adjacent hole?

get all 15 blocks into their natural order. A picture of the 15-puzzle is shown in Figure 3.5 along with the configuration after the 12-block is moved into the hole below. The desired final configuration is shown in Figure 3.6.

The 15-puzzle became very popular in North America and Europe and is still sold in game and puzzle shops today. Prizes were offered for its solution, but it is doubtful that they were ever awarded, since it is impossible to get from the configuration in Figure 3.5(a) to the configuration in Figure 3.6 by only moving one block at a time into an adjacent hole. The proof of this fact is a little tricky so we have left it for you to figure out on your own! Instead, we will prove that the analogous task for the much easier 8-puzzle cannot be performed. Both proofs, of course, make use of the Invariant Method.

(a)   (b)   (c)
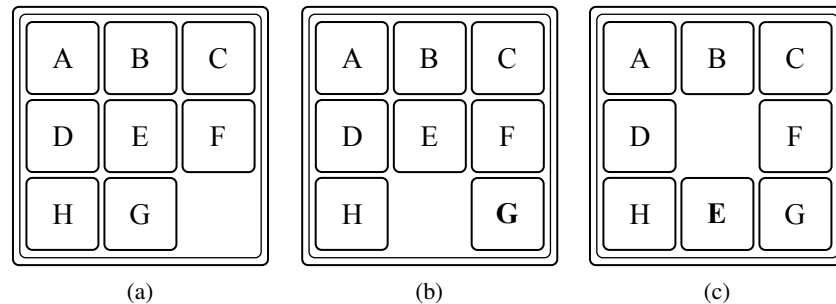
Figure 3.7 The 8-Puzzle in its initial configuration (a) and after one (b) and two (c) possible moves.

3.3.4 The 8-Puzzle

In the 8-Puzzle, there are 8 lettered tiles (A–H) and a blank square arranged in a 3×3 grid. Any lettered tile adjacent to the blank square can be slid into the blank. For example, a sequence of two moves is illustrated in Figure 3.7.

In the initial configuration shown in Figure 3.7(a), the G and H tiles are out of order. We can find a way of swapping G and H so that they are in the right order, but then other letters may be out of order. Can you find a sequence of moves that puts these two letters in correct order, but returns every other tile to its original position? Some experimentation suggests that the answer is probably "no," and we will prove that is so by finding an invariant, namely, a property of the puzzle that is always maintained, no matter how you move the tiles around. If we can then show that putting all the tiles in the correct order would violate the invariant, then we can conclude that the puzzle cannot be solved.

Theorem 3.3.3. *No sequence of legal moves transforms the configuration in Figure 3.7(a) into the configuration in Figure 3.8.*

We'll build up a sequence of observations, stated as lemmas. Once we achieve a critical mass, we'll assemble these observations into a complete proof of Theorem 3.3.3.

Define a *row move* as a move in which a tile slides horizontally and a *column move* as one in which the tile slides vertically. Assume that tiles are read top-to-bottom and left-to-right like English text, that is, the *natural order*, defined as follows: So when we say that two tiles are "out of order", we mean that the larger letter precedes the smaller letter in this natural order.

Our difficulty is that one pair of tiles (the G and H) is out of order initially. An immediate observation is that row moves alone are of little value in addressing this

3.3. Invariants
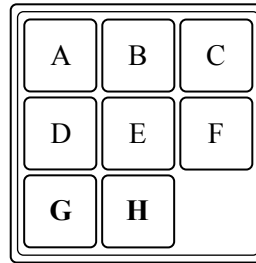
Figure 3.8   The desired final configuration of the 8-puzzle.
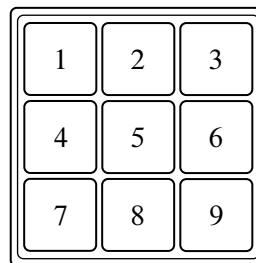
problem:

Lemma 3.3.4. *A row move does not change the order of the tiles.*

Proof. A row move moves a tile from cell i to cell $i + 1$ or vice versa. This tile does not change its order with respect to any other tile. Since no other tile moves, there is no change in the order of any of the other pairs of tiles. ∎

Let's turn to column moves. This is the more interesting case, since here the order can change. For example, the column move in Figure 3.9 changes the relative order of the pairs (G, H) and (G, E).

Lemma 3.3.5. *A column move changes the relative order of exactly two pairs of tiles.*

Proof. Sliding a tile down moves it after the next two tiles in the order. Sliding a tile up moves it before the previous two tiles in the order. Either way, the relative order changes between the moved tile and each of the two tiles it crosses. The relative order between any other pair of tiles does not change. ∎

These observations suggest that there are limitations on how tiles can be swapped. Some such limitation may lead to the invariant we need. In order to reason about swaps more precisely, let's define a term referring to a pair of items that are out of order:
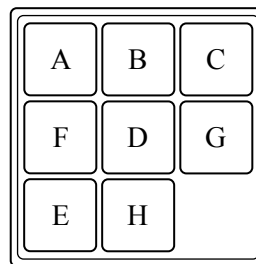
(a)   (b)
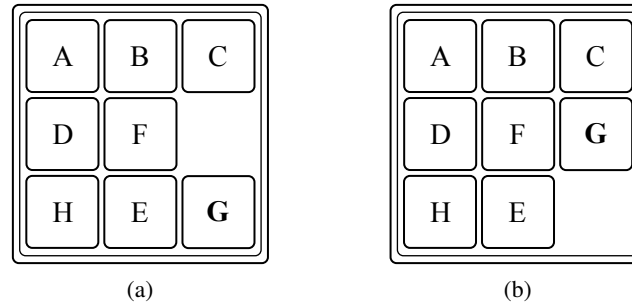
Figure 3.9   An example of a column move in which the G-tile is moved into the adjacent hole above. In this case, G changes order with E and H.

Definition 3.3.6. A pair of letters L_1 and L_2 is an *inversion* if L_1 precedes L_2 in the alphabet, but L_1 appears after L_2 in the puzzle order.

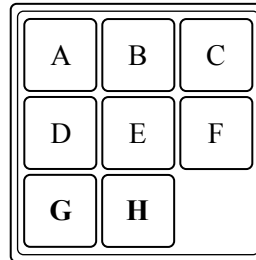For example, in the puzzle below, there are three inversions: (D, F), (E, F), (E, G).

There is exactly one inversion (G, H) in the start state:
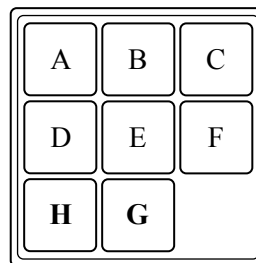
3.3. Invariants

There are no inversions in the end state:

Let's work out the effects of row and column moves in terms of inversions.

Lemma 3.3.7. *During a move, the number of inversions can only increase by 2, decrease by 2, or remain the same.*

Proof. By Lemma 3.3.4, a row move does not change the order of the tiles, and so a row move does not change the number of inversions.

By Lemma 3.3.5, a column move changes the relative order of exactly 2 pairs of tiles. There are three cases: If both pairs were originally in order, then the number of inversions after the move goes up by 2. If both pairs were originally inverted, then the number of inversions after the move goes down by 2. If one pair was originally inverted and the other was originally in order, then the number of inversions stays the same (since changing the former pair makes the number of inversions smaller by 1, and changing the latter pair makes the number of inversions larger by 1). ∎

We are almost there. If the number of inversions only changes by 2, then what about the parity of the number of inversions? (The "parity" of a number refers to whether the number is even or odd. For example, 7 and 5 have odd parity, and 18 and 0 have even parity.)

Since adding or subtracting 2 from a number does not change its parity, we have the following corollary to Lemma 3.3.7:

Corollary 3.3.8. *Neither a row move nor a column move ever changes the parity of the number of inversions.*

Now we can bundle up all these observations and state an *invariant*, that is, a property of the puzzle that never changes, no matter how you slide the tiles around.

Lemma 3.3.9. *In every configuration reachable from the configuration shown in Figure 3.7(a), the* parity *of the number of inversions is odd.*

Chapter 3 Induction

Proof. We use induction. Let $P(n)$ be the proposition that after n moves from the above configuration, the parity of the number of inversions is odd.

Base case: After zero moves, exactly one pair of tiles is inverted (G and H), which is an odd number. Therefore $P(0)$ is true.

Inductive step: Now we must prove that $P(n)$ implies $P(n+1)$ for all $n \geq 0$. So assume that $P(n)$ is true; that is, after n moves the parity of the number of inversions is odd. Consider any sequence of $n+1$ moves $m_1, \ldots, m_{n+1}$. By the induction hypothesis $P(n)$, we know that the parity after moves $m_1, \ldots, m_n$ is odd. By Corollary 3.3.8, we know that the parity does not change during m_{n+1}. Therefore, the parity of the number of inversions after moves $m_1, \ldots, m_{n+1}$ is odd, so we have that $P(n+1)$ is true.

By the principle of induction, $P(n)$ is true for all $n \geq 0$. ∎

The theorem we originally set out to prove is restated below. With our invariant in hand, the proof is simple.

Theorem. *No sequence of legal moves transforms the board below on the left into the board below on the right.*
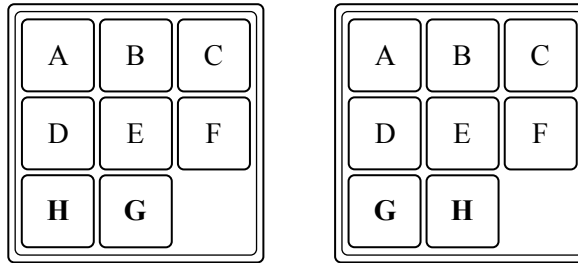
Proof. In the target configuration on the right, the total number of inversions is zero, which is even. Therefore, by Lemma 3.3.9, the target configuration is un-reachable. ∎

3.4   Strong Induction

Strong induction is a variation of ordinary induction that is useful when the predicate $P(n+1)$ naturally depends on $P(a)$ for values of $a < n$. As with ordinary induction, strong induction is useful to prove that a predicate $P(n)$ is true for all $n \in \mathbb{N}$.

3.4.1   A Rule for Strong Induction

> **Principle of Strong Induction.**   Let $P(n)$ be a predicate. If
>
> - $P(0)$ is true, and
>
> - for all $n \in \mathbb{N}$, $P(0)$, $P(1)$, …, $P(n)$ *together* imply $P(n+1)$,
>
> then $P(n)$ is true for all $n \in \mathbb{N}$.

The only change from the ordinary induction principle is that strong induction allows you to assume more stuff in the inductive step of your proof! In an ordinary induction argument, you assume that $P(n)$ is true and try to prove that $P(n+1)$ is also true. In a strong induction argument, you may assume that $P(0)$, $P(1)$, …, and $P(n)$ are *all* true when you go to prove $P(n+1)$. These extra assumptions can only make your job easier. Hence the name: *strong* induction.

Formulated as a proof rule, strong induction is

Rule. *Strong Induction Rule*

$$\frac{P(0), \quad \forall n \in \mathbb{N}. \ \big(P(0) \text{ AND } P(1) \text{ AND } \dots \text{ AND } P(m)\big) \text{ IMPLIES } P(n+1)]}{\forall m \in \mathbb{N}. \ P(m)}$$

The template for strong induction proofs is identical to the template given in Section 3.2.3 for ordinary induction except for two things:

- you should state that your proof is by strong induction, and

- you can assume that $P(0)$, $P(1)$, …, $P(n)$ are all true instead of only $P(n)$ during the inductive step.

3.4.2   Some Examples

Products of Primes

As a first example, we'll use strong induction to re-prove Theorem 3.1.2 which we previously proved using Well Ordering.

Lemma 3.4.1. *Every integer greater than 1 is a product of primes.*

Proof. We will prove Lemma 3.4.1 by strong induction, letting the induction hypothesis, $P(n)$, be

$$n \text{ is a product of primes.}$$

So Lemma 3.4.1 will follow if we prove that $P(n)$ holds for all $n \geq 2$.

Base Case: ($n = 2$) $P(2)$ is true because 2 is prime, and so it is a length one product of primes by convention.

Inductive step: Suppose that $n \geq 2$ and that i is a product of primes for every integer i where $2 \leq i < n + 1$. We must show that $P(n + 1)$ holds, namely, that $n + 1$ is also a product of primes. We argue by cases:

If $n + 1$ is itself prime, then it is a length one product of primes by convention, and so $P(n + 1)$ holds in this case.

Otherwise, $n + 1$ is not prime, which by definition means $n + 1 = km$ for some integers k, m such that $2 \leq k, m < n + 1$. Now by the strong induction hypothesis, we know that k is a product of primes. Likewise, m is a product of primes. It follows immediately that $km = n$ is also a product of primes. Therefore, $P(n + 1)$ holds in this case as well.

So $P(n + 1)$ holds in any case, which completes the proof by strong induction that $P(n)$ holds for all $n \geq 2$.

∎

Making Change

The country Inductia, whose unit of currency is the Strong, has coins worth 3Sg (3 Strongs) and 5Sg. Although the Inductians have some trouble making small change like 4Sg or 7Sg, it turns out that they can collect coins to make change for any number that is at least 8 Strongs.

Strong induction makes this easy to prove for $n + 1 \geq 11$, because then $(n + 1) - 3 \geq 8$, so by strong induction the Inductians can make change for exactly $(n + 1) - 3$ Strongs, and then they can add a 3Sg coin to get $(n + 1)$Sg. So the only thing to do is check that they can make change for all the amounts from 8 to 10Sg, which is not too hard to do.

Here's a detailed writeup using the official format:

Proof. We prove by strong induction that the Inductians can make change for any amount of at least 8Sg. The induction hypothesis, $P(n)$ will be:

There is a collection of coins whose value is $n + 8$ Strongs.

Base case: $P(0)$ is true because a 3Sg coin together with a 5Sg coin makes 8Sg.

Inductive step: We assume $P(m)$ holds for all $m \leq n$, and prove that $P(n + 1)$ holds. We argue by cases:

Case ($n + 1 = 1$): We have to make $(n + 1) + 8 = 9$Sg. We can do this using three 3Sg coins.

Case ($n + 1 = 2$): We have to make $(n + 1) + 8 = 10$Sg. Use two 5Sg coins.

3.4. Strong Induction

Stack Heights										**Score**
10										
5	5									25 points
5	3	2								6
4	3	2	1							4
2	3	2	1	2						4
2	2	2	1	2	1					2
1	2	2	1	2	1	1				1
1	1	2	1	2	1	1	1			1
1	1	1	1	2	1	1	1	1		1
1	1	1	1	1	1	1	1	1	1	1
							Total Score	=		45 points

Figure 3.10 An example of the stacking game with $n = 10$ boxes. On each line, the underlined stack is divided in the next step.

Case ($n + 1 \geq 3$): Then $0 \leq n - 2 \leq n$, so by the strong induction hypothesis, the Inductians can make change for $n - 2$ Strong. Now by adding a 3Sg coin, they can make change for $(n + 1)$Sg.

Since $n \geq 0$, we know that $n + 1 \geq 1$ and thus that the three cases cover every possibility. Since $P(n + 1)$ is true in every case, we can conclude by strong induction that for all $n \geq 0$, the Inductians can make change for $n + 8$ Strong. That is, they can make change for any number of eight or more Strong.

∎

The Stacking Game

Here is another exciting game that's surely about to sweep the nation!

You begin with a stack of n boxes. Then you make a sequence of moves. In each move, you divide one stack of boxes into two nonempty stacks. The game ends when you have n stacks, each containing a single box. You earn points for each move; in particular, if you divide one stack of height $a + b$ into two stacks with heights a and b, then you score ab points for that move. Your overall score is the sum of the points that you earn for each move. What strategy should you use to maximize your total score?

As an example, suppose that we begin with a stack of $n = 10$ boxes. Then the game might proceed as shown in Figure 3.10. Can you find a better strategy?

Let's use strong induction to analyze the unstacking game. We'll prove that your score is determined entirely by the number of boxes—your strategy is irrelevant!

Chapter 3   Induction

Theorem 3.4.2. *Every way of unstacking n blocks gives a score of $n(n-1)/2$ points.*

There are a couple technical points to notice in the proof:

- The template for a strong induction proof mirrors the template for ordinary induction.

- As with ordinary induction, we have some freedom to adjust indices. In this case, we prove $P(1)$ in the base case and prove that $P(1), \ldots, P(n)$ imply $P(n+1)$ for all $n \geq 1$ in the inductive step.

Proof. The proof is by strong induction. Let $P(n)$ be the proposition that every way of unstacking n blocks gives a score of $n(n-1)/2$.

Base case: If $n = 1$, then there is only one block. No moves are possible, and so the total score for the game is $1(1-1)/2 = 0$. Therefore, $P(1)$ is true.

Inductive step: Now we must show that $P(1), \ldots, P(n)$ imply $P(n+1)$ for all $n \geq 1$. So assume that $P(1), \ldots, P(n)$ are all true and that we have a stack of $n + 1$ blocks. The first move must split this stack into substacks with positive sizes a and b where $a + b = n + 1$ and $0 < a, b \leq n$. Now the total score for the game is the sum of points for this first move plus points obtained by unstacking the two resulting substacks:

$$
\begin{aligned}
\text{total score} &= (\text{score for 1st move}) \\
&\quad + (\text{score for unstacking } a \text{ blocks}) \\
&\quad + (\text{score for unstacking } b \text{ blocks}) \\
&= ab + \frac{a(a-1)}{2} + \frac{b(b-1)}{2} \qquad\qquad \text{by } P(a) \text{ and } P(b) \\
&= \frac{(a+b)^2 - (a+b)}{2} = \frac{(a+b)((a+b)-1)}{2} \\
&= \frac{(n+1)n}{2}
\end{aligned}
$$

This shows that $P(1), P(2), \ldots, P(n)$ imply $P(n+1)$.

Therefore, the claim is true by strong induction. ∎

3.4.3   Strong Induction versus Induction

Is strong induction really "stronger" than ordinary induction? It certainly looks that way. After all, you can assume a lot more when proving the induction step. But actually, any proof using strong induction can be reformatted into a proof using ordinary induction—you just need to use a "stronger" induction hypothesis.

Which method should you use? Whichever you find easier. But whichever method you choose, be sure to state the method up front so that the reader can understand and more easily verify your proof.

3.5 Structural Induction

Up to now, we have focussed on induction over the natural numbers. But the idea of induction is far more general—it can be applied to a much richer class of sets. In particular, it is especially useful in connection with sets or data types that are defined recursively.

3.5.1 Recursive Data Types

Recursive data types play a central role in programming. They are specified by *recursive definitions* that say how to build something from its parts. *Recursive* definitions have two parts:

- **Base case(s)** that don't depend on anything else.

- **Constructor case(s)** that depend on previous cases.

Let's see how this works in a couple of examples: Strings of brackets and expression evaluation.

Example 1: Strings of Brackets

Let brkts be the set of all sequences (or strings) of square brackets. For example, the following two strings are in brkts:

$$[\,]\,]\,[\,[\,[\,[\,]\,] \quad \text{and} \quad [\,[\,[\,]\,]\,[\,]\,]\,[\,] \tag{3.5}$$

Definition 3.5.1. The set brkts of strings of brackets can be defined recursively as follows:

- **Base case:** The *empty string*, λ, is in brkts.

- **Constructor case:** If $s \in$ brkts, then $s\,]$ and $s\,[$ are in brkts.

Here, we're writing $s\,]$ to indicate the string that is the sequence of brackets (if any) in the string s, followed by a right bracket; similarly for $s\,[$.

A string $s \in$ brkts is called a *matched string* if its brackets can be "matched up" in the usual way. For example, the left hand string in 3.5 is not matched because its second right bracket does not have a matching left bracket. The string on the right is matched. The set of matched strings can be defined recursively as follows.

Chapter 3   Induction

Definition 3.5.2. Recursively define the set, RecMatch, of strings as follows:

- **Base case:** $\lambda \in$ RecMatch.

- **Constructor case:** If $s, t \in$ RecMatch, then

$$[\, s \,]\, t \in \text{RecMatch}.$$

Here we're writing $[\, s \,]\, t$ to indicate the string that starts with a left bracket, followed by the sequence of brackets (if any) in the string s, followed by a right bracket, and ending with the sequence of brackets in the string t.

Using this definition, we can see that $\lambda \in$ RecMatch by the Base case, so

$$[\lambda]\lambda = [\,] \in \text{RecMatch}$$

by the Constructor case. So now,

$$[\lambda][\,] = [\,][\,] \in \text{RecMatch} \qquad (\text{letting } s = \lambda, t = [\,])$$
$$[[\,]]\lambda = [[\,]] \in \text{RecMatch} \qquad (\text{letting } s = [\,], t = \lambda)$$
$$[[\,]][\,] \in \text{RecMatch} \qquad (\text{letting } s = [\,], t = [\,])$$

are also strings in RecMatch by repeated applications of the Constructor case.

In general, RecMatch will contain precisely the strings with matching brackets. This is because the constructor case is, in effect, identifying the bracket that matches the leftmost bracket in any string. Since that matching bracket is unique, this method of constructing RecMatch gives a unique way of constructing any string with matched brackets. This will turn out to be important later when we talk about ambiguity.

Strings with matched brackets arise in the area of expression parsing. A brief history of the advances in this field is provided in the box on the next page.

Example 2: Arithmetic Expressions

Expression evaluation is a key feature of programming languages, and recognition of expressions as a recursive data type is a key to understanding how they can be processed.

To illustrate this approach we'll work with a toy example: arithmetic expressions like $3x^2 + 2x + 1$ involving only one variable, "x." We'll refer to the data type of such expressions as Aexp. Here is its definition:

Definition 3.5.3. The set Aexp is defined recursively as follows:

- **Base cases:**

3.5. Structural Induction

Expression Parsing

During the early development of computer science in the 1950's and 60's, creation of effective programming language compilers was a central concern. A key aspect in processing a program for compilation was expression parsing. The problem was to take in an expression like

$$x + y * z^2 \div y + 7$$

and *put in* the brackets that determined how it should be evaluated—should it be

$$[[x + y] * z^2 \div y] + 7, \text{ or,}$$
$$x + [y * z^2 \div [y + 7]], \text{ or,}$$
$$[x + [y * z^2]] \div [y + 7],$$

or ... ?

The Turing award (the "Nobel Prize" of computer science) was ultimately bestowed on Robert Floyd, for, among other things, being discoverer of a simple program that would insert the brackets properly.

In the 70's and 80's, this parsing technology was packaged into high-level compiler-compilers that automatically generated parsers from expression grammars. This automation of parsing was so effective that the subject stopped demanding attention and largely disappeared from the computer science curriculum by the 1990's.

1. The variable, x, is in Aexp.

2. The arabic numeral, k, for any nonnegative integer, k, is in Aexp.

- **Constructor cases:** If e, f ∈ Aexp, then

3. $(e + f) \in$ Aexp. The expression $(e + f)$ is called a *sum*. The Aexp's e and f are called the *components* of the sum; they're also called the *summands*.

4. $(e * f) \in$ Aexp. The expression $(e * f)$ is called a *product*. The Aexp's e and f are called the *components* of the product; they're also called the *multiplier* and *multiplicand*.

5. $-(e) \in$ Aexp. The expression $-(e)$ is called a *negative*.

Notice that Aexp's are fully parenthesized, and exponents aren't allowed. So the Aexp version of the polynomial expression $3x^2 + 2x + 1$ would officially be written as

$$((3 * (x * x)) + ((2 * x) + 1)). \tag{3.6}$$

These parentheses and $*$'s clutter up examples, so we'll often use simpler expressions like "$3x^2 + 2x + 1$" instead of (3.6). But it's important to recognize that $3x^2 + 2x + 1$ is not an Aexp; it's an *abbreviation* for an Aexp.

3.5.2 Structural Induction on Recursive Data Types

Structural induction is a method for proving that some property, P, holds for all the elements of a recursively-defined data type. The proof consists of two steps:

- Prove P for the base cases of the definition.

- Prove P for the constructor cases of the definition, assuming that it is true for the component data items.

A very simple application of structural induction proves that (recursively-defined) matched strings always have an equal number of left and right brackets. To do this, define a predicate, P, on strings $s \in$ brkts:

$$P(s) ::= \quad s \text{ has an equal number of left and right brackets.}$$

Theorem 3.5.4. *$P(s)$ holds for all $s \in$ RecMatch.*

Proof. By structural induction on the definition that $s \in$ RecMatch, using $P(s)$ as the induction hypothesis.

Base case: $P(\lambda)$ holds because the empty string has zero left and zero right brackets.

Constructor case: For $r = [\,s\,]\,t$, we must show that $P(r)$ holds, given that $P(s)$ and $P(t)$ holds. So let n_s, n_t be, respectively, the number of left brackets in s and t. So the number of left brackets in r is $1 + n_s + n_t$.

Now from the respective hypotheses $P(s)$ and $P(t)$, we know that the number of right brackets in s is n_s, and likewise, the number of right brackets in t is n_t. So the number of right brackets in r is $1 + n_s + n_t$, which is the same as the number of left brackets. This proves $P(r)$. We conclude by structural induction that $P(s)$ holds for all $s \in$ RecMatch. ∎

3.5.3 Functions on Recursively-defined Data Types

A Quick Review of Functions

A *function* assigns an element of one set, called the *domain*, to elements of another set, called the *codomain*. The notation

$$f : A \to B$$

indicates that f is a function with domain, A, and codomain, B. The familiar notation "$f(a) = b$" indicates that f assigns the element $b \in B$ to a. Here b would be called the *value* of f at *argument a*.

Functions are often defined by formulas as in:

$$f_1(x) ::= \frac{1}{x^2}$$

where x is a real-valued variable, or

$$f_2(y, z) ::= y\,10\,yz$$

where y and z range over binary strings, or

$$f_3(x, n) ::= \text{ the pair } (n, x)$$

where n ranges over the nonnegative integers.

A function with a finite domain could be specified by a table that shows the value of the function at each element of the domain. For example, a function $f_4(P, Q)$ where P and Q are propositional variables is specified by:

P	Q	$f_4(P, Q)$
T	T	T
T	F	F
F	T	T
F	F	T

Notice that f_4 could also have been described by a formula:

$$f_4(P, Q) ::= [P \text{ IMPLIES } Q].$$

A function might also be defined by a procedure for computing its value at any element of its domain, or by some other kind of specification. For example, define $f_5(y)$ to be the length of a left to right search of the bits in the binary string y until a 1 appears, so

$$f_5(0010) = 3,$$
$$f_5(100) = 1,$$
$$f_5(0000) \text{ is undefined.}$$

Notice that f_5 does not assign a value to a string of just 0's. This illustrates an important fact about functions: they need not assign a value to every element in the domain. In fact this came up in our first example $f_1(x) = 1/x^2$, which does not assign a value to 0. So in general, functions may be *partial functions*, meaning that there may be domain elements for which the function is not defined. If a function is defined on every element of its domain, it is called a *total function*.

It's often useful to find the set of values a function takes when applied to the elements in *a set* of arguments. So if $f : A \to B$, and S is a subset of A, we define $f(S)$ to be the set of all the values that f takes when it is applied to elements of S. That is,

$$f(S) ::= \{b \in B \mid f(s) = b \text{ for some } s \in S\}.$$

For example, if we let $[r, s]$ denote the interval from r to s on the real line, then $f_1([1, 2]) = [1/4, 1]$.

For another example, let's take the "search for a 1" function, f_5. If we let X be the set of binary words which start with an even number of 0's followed by a 1, then $f_5(X)$ would be the odd nonnegative integers.

Applying f to a set, S, of arguments is referred to as "applying f pointwise to S", and the set $f(S)$ is referred to as the *image* of S under f.[8] The set of values that arise from applying f to all possible arguments is called the *range* of f. That is,

$$\text{range}(f) ::= f(\text{domain}(f)).$$

---

[8]There is a picky distinction between the function f which applies to elements of A and the function which applies f pointwise to subsets of A, because the domain of f is A, while the domain of pointwise-f is $\mathcal{P}(A)$. It is usually clear from context whether f or pointwise-f is meant, so there is no harm in overloading the symbol f in this way.

3.5. Structural Induction

Recursively-Defined Functions

Functions on recursively-defined data types can be defined recursively using the same cases as the data type definition. Namely, to define a function, f, on a recursive data type, define the value of f for the base cases of the data type definition, and then define the value of f in each constructor case in terms of the values of f on the component data items.

For example, consider the function

$$\text{eval} : \text{Aexp} \times \mathbb{Z} \to \mathbb{Z},$$

which evaluates any expression in Aexp using the value n for x. It is useful to express this function with a recursive definition as follows:

Definition 3.5.5. The *evaluation function*, eval : Aexp $\times \mathbb{Z} \to \mathbb{Z}$, is defined recursively on expressions, $e \in$ Aexp, as follows. Let n be any integer.

- **Base cases:**

    1. Case[e is x]
    $$\text{eval}(x, n) ::= n.$$

    (The value of the variable, x, is given to be n.)

    2. Case[e is k]
    $$\text{eval}(\text{k}, n) ::= k.$$

    (The value of the numeral k is the integer k, no matter what value x has.)

- **Constructor cases:**

    3. Case[e is $(e_1 + e_2)$]
    $$\text{eval}((e_1 + e_2), n) ::= \text{eval}(e_1, n) + \text{eval}(e_2, n).$$

    4. Case[e is $(e_1 * e_2)$]
    $$\text{eval}((e_1 * e_2), n) ::= \text{eval}(e_1, n) \cdot \text{eval}(e_2, n).$$

    5. Case[e is $-(e_1)$]
    $$\text{eval}(-(e_1), n) ::= -\text{eval}(e_1, n).$$

Chapter 3 Induction

For example, here's how the recursive definition of eval would arrive at the value of $3 + x^2$ when x is 2:

$$
\begin{aligned}
\text{eval}((3 + (x * x)), 2) &= \text{eval}(3, 2) + \text{eval}((x * x), 2) & \text{(by Def 3.5.5.3)} \\
&= 3 + \text{eval}((x * x), 2) & \text{(by Def 3.5.5.2)} \\
&= 3 + (\text{eval}(x, 2) \cdot \text{eval}(x, 2)) & \text{(by Def 3.5.5.4)} \\
&= 3 + (2 \cdot 2) & \text{(by Def 3.5.5.1)} \\
&= 3 + 4 = 7.
\end{aligned}
$$

A Second Example

We next consider the function on matched strings that specifies the depth of the matched brackets in any string. This function can be specified recursively as follows:

Definition 3.5.6. The *depth* $d(s)$ of a string $s \in \text{RecMatch}$ is defined recursively by the rules:

- $d(\lambda) ::= 0$.

- $d([\,s\,]\,t) ::= \max\{d(s) + 1, d(t)\}$

Ambiguity

When a recursive definition of a data type allows the same element to be constructed in more than one way, the definition is said to be *ambiguous*. A function defined recursively from an ambiguous definition of a data type will not be well-defined unless the values specified for the different ways of constructing the element agree.

We were careful to choose an *un*ambiguous definition of RecMatch to ensure that functions defined recursively on the definition would always be well-defined. As an example of the trouble an ambiguous definition can cause, let's consider another definition of the matched strings.

Definition 3.5.7. Define the set, $M \subseteq \texttt{brkts}$ recursively as follows:

- **Base case:** $\lambda \in M$,

- **Constructor cases:** if $s, t \in M$, then the strings $[\,s\,]$ and st are also in M.

By using structural induction, it is possible to prove that $M = \text{RecMatch}$. Indeed, the definition of M might even seem like a more natural way to define the set

of matched strings than the definition of RecMatch. But the definition of M is ambiguous, while the (perhaps less natural) definition of RecMatch is unambiguous. Does this ambiguity matter? Yes, it can. For example, suppose we defined

$$f(\lambda) ::= 1,$$
$$f(\,[\,s\,]\,) ::= 1 + f(s),$$
$$f(st) ::= (f(s) + 1) \cdot (f(t) + 1) \qquad \text{for } st \neq \lambda.$$

Let a be the string $[\,[\,]\,]$ $\in M$ built by two successive applications of the first M constructor starting with λ. Next let

$$b ::= aa$$
$$= [\,[\,]\,][\,[\,]\,]$$

and

$$c ::= bb$$
$$= [\,[\,]\,][\,[\,]\,][\,[\,]\,][\,[\,]\,]$$

each be built by successive applications of the second M constructor starting with a.

Alternatively, we can build ba from the second constructor with $s = b$ and $t = a$, and then get to c using the second constructor with $s = ba$ and $t = a$.

By applying these rules to the first way of constructing c, $f(a) = 2$, $f(b) = (2+1)(2+1) = 9$, and $f(c) = f(bb) = (9+1)(9+1) = 100$. Using the second way of constructing c, we find that $f(ba) = (9+1)(2+1) = 27$ and $f(c) = f(ba\,a) = (27+1)(2+1) = 84$. The outcome is that $f(c)$ is defined to be both 100 and 84, which shows that the rules defining f are inconsistent.

Note that structural induction remains a sound proof method even for ambiguous recursive definitions, which is why it is easy to prove that $M = $ RecMatch.

3.5.4 Recursive Functions on $\mathbb{N}$—Structural Induction versus Ordinary Induction

The nonnegative integers can be understood as a recursive data type.

Definition 3.5.8. The set, $\mathbb{N}$, is a data type defined recursivly as:

- **Base Case:** $0 \in \mathbb{N}$.

- **Constructor Case:** If $n \in \mathbb{N}$, then the *successor*, $n + 1$, of n is in $\mathbb{N}$.

Chapter 3   Induction

This means that ordinary induction is a special case of structural induction on the recursive Definition 3.5.8. Conversely, most proofs based on structural induction that you will encounter in computer science can also be reformatted into proofs that use only ordinary induction. The decision as to which technique to use is up to you, but it will often be the case that structural induction provides the easiest approach when you are dealing with recursive data structures or functions.

Definition 3.5.8 also justifies the familiar recursive definitions of functions on the nonnegative integers. Here are some examples.

The Factorial Function

The factorial function is often written "$n!$." You will be seeing it a lot in Parts III and IV of this text. For now, we'll use the notation $\mathrm{fac}(n)$ and define it recursively as follows:

- **Base Case:** $\mathrm{fac}(0) ::= 1$.

- **Constructor Case:** $\mathrm{fac}(n + 1) ::= (n + 1) \cdot \mathrm{fac}(n)$ for $n \geq 0$.

The Fibonacci numbers.

Fibonacci numbers arose out of an effort 800 years ago to model population growth. We will study them at some length in Part III. The nth Fibonacci number, $\mathrm{fib}(n)$, can be defined recursively by:

- **Base Cases:** $\mathrm{fib}(0) ::= 0$ and $\mathrm{fib}(1) ::= 1$

- **Constructor Case:** $\mathrm{fib}(n) ::= \mathrm{fib}(n - 1) + \mathrm{fib}(n - 2)$ for $n \geq 2$.

Here the recursive step starts at $n = 2$ with base cases for $n = 0$ and $n = 1$. This is needed since the recursion relies on two previous values.

What is $\mathrm{fib}(4)$? Well, $\mathrm{fib}(2) = \mathrm{fib}(1) + \mathrm{fib}(0) = 1$, $\mathrm{fib}(3) = \mathrm{fib}(2) + \mathrm{fib}(1) = 2$, so $\mathrm{fib}(4) = 3$. The sequence starts out $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$.

Sum-notation

Let "$S(n)$" abbreviate the expression "$\sum_{i=1}^{n} f(i)$." We can recursively define $S(n)$ with the rules

- **Base Case:** $S(0) ::= 0$.

- **Constructor Case:** $S(n + 1) ::= f(n + 1) + S(n)$ for $n \geq 0$.

3.5. Structural Induction

Ill-formed Function Definitions

There are some blunders to watch out for when defining functions recursively. Below are some function specifications that resemble good definitions of functions on the nonnegative integers, but they aren't.

Definition 3.5.9.

$$f_1(n) ::= 2 + f_1(n-1). \tag{3.7}$$

This "definition" has no base case. If some function, f_1, satisfied (3.7), so would a function obtained by adding a constant to the value of f_1. So equation (3.7) does not uniquely define an f_1.

Definition 3.5.10.

$$f_2(n) ::= \begin{cases} 0, & \text{if } n = 0, \\ f_2(n+1) & \text{otherwise.} \end{cases} \tag{3.8}$$

This "definition" has a base case, but still doesn't uniquely determine f_2. Any function that is 0 at 0 and constant everywhere else would satisfy the specification, so (3.8) also does not uniquely define anything.

In a typical programming language, evaluation of $f_2(1)$ would begin with a recursive call of $f_2(2)$, which would lead to a recursive call of $f_2(3)$, ... with recursive calls continuing without end. This "operational" approach interprets (3.8) as defining a *partial* function, f_2, that is undefined everywhere but 0.

Definition 3.5.11.

$$f_3(n) ::= \begin{cases} 0, & \text{if } n \text{ is divisible by 2,} \\ 1, & \text{if } n \text{ is divisible by 3,} \\ 2, & \text{otherwise.} \end{cases} \tag{3.9}$$

This "definition" is inconsistent: it requires $f_3(6) = 0$ and $f_3(6) = 1$, so (3.9) doesn't define anything.

A Mysterious Function

Mathematicians have been wondering about the following function specification for many years:

$$f_4(n) ::= \begin{cases} 1, & \text{if } n \leq 1, \\ f_4(n/2) & \text{if } n > 1 \text{ is even,} \\ f_4(3n+1) & \text{if } n > 1 \text{ is odd.} \end{cases} \tag{3.10}$$

Chapter 3   Induction

For example, $f_4(3) = 1$ because

$$f_4(3) ::= f_4(10) ::= f_4(5) ::= f_4(16) ::= f_4(8) ::= f_4(4) ::= f_4(2) ::= f_4(1) ::= 1.$$

The constant function equal to 1 will satisfy (3.10), but it's not known if another function does too. The problem is that the third case specifies $f_4(n)$ in terms of f_4 at arguments larger than n, and so cannot be justified by induction on $\mathbb{N}$. It's known that any f_4 satisfying (3.10) equals 1 for all n up to over a billion.

6.042J / 18.062J Mathematics for Computer Science
Fall 2010