Mathematics for Computer Science

First Edition

revised June 4, 2009, 1142 minutes

Prof. Albert R Meyer
Massachussets Institute of Technology

Creative Commons BY NC SA 2009, Prof. Albert R. Meyer.

Chapter 3

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. Now she states two rules:

1. The student at the beginning of the line gets a candy bar.
2. If a student gets candy bar, then the following student in line also gets a candy bar, for every student in the line.

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.

3.1 Ordinary Induction

The reasoning that led us to conclude 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.[1] What's not so obvious is how much mileage we get by using it.

3.1.1 Using Ordinary Induction

Ordinary induction often works directly in proving that some statement about nonnegative integers holds for all of them. For example, here is a classic formula:

[1] But see section **??**.

Theorem 0.1. *For all $n \in \mathbb{N}$,*

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \tag{3.1}$$

The left side of equation (3.1) represents the sum of all the numbers from 1 to n. Here the dots $(\cdots)$ indicate a pattern you're supposed to be able to guess so you can mentally fill in the remaining terms. However, the meaning of this sum is not so obvious in a couple of special cases:

- If $n = 1$, then there is only one term in the summation, and so $1 + 2 + 3 + \cdots + n = 1$. 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.)

We could eliminate 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 on by the expression to the right of the sigma as the variable, i, ranges from 1 to n. Both these summation 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).

Now let's use the induction principle to prove Theorem 0.1. Suppose that we define predicate $P(n)$ to be "$1 + 2 + 3 + \cdots + n = n(n+1)/2$". 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:

- $P(0)$ 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 $0(0+1)/2 = 0$. The second statement is more complicated. But remember the basic plan for proving the validity of any implication: *assume* the statement on the left and then *prove* the statement on the right. In this case, we assume $P(n)$:

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

in order to prove $P(n+1)$:

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

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):

$$\begin{aligned} 1+2+3+\cdots+n+(n+1) &= \frac{n(n+1)}{2} + (n+1) \\ &= \frac{(n+2)(n+1)}{2} \end{aligned}$$

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

3.1.2 A Template for Induction Proofs

The proof of Theorem 0.1 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 claim, 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.

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". (Sometimes the base case will be $n = 1$ or even some larger number, in which case the starting value of n also should be stated.)

4. **Prove that $P(n)$ implies $P(n+1)$ for every nonnegative integer n.** This is called the "inductive step" or "induction 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 many writers leave this step implicit.

Explicitly labeling the *base case* and *inductive step* may make your proofs clearer.

3.1.3 A Clean Writeup

The proof of Theorem 0.1 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. We use induction. The induction hypothesis, $P(n)$, will be equation (3.1).

Base case: $P(0)$ is true, because both sides of equation (3.1) equal zero when $n = 0$.

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

$$\begin{aligned} 1 + 2 + 3 + \cdots + n + (n+1) &= \frac{n(n+1)}{2} + (n+1) && \text{(by induction hypothesis)} \\ &= \frac{(n+1)(n+2)}{2} && \text{(by simple algebra)} \end{aligned}$$

which proves $P(n+1)$.

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

Induction was helpful for *proving the correctness* of this summation formula, but not helpful for *discovering* it in the first place. We'll show you some tricks for finding such formulas in a few weeks.

3.1.4 Powers of Odd Numbers

Fact. The nth power of an odd number is odd, for all nonnegative integers, n.

The proof in Chapter **??** that $\sqrt[n]{2}$ is irrational used this "obvious" fact. Instead of taking it for granted, we can prove this fact by induction. The proof will require a simple Lemma.

Lemma. *The product of two odd numbers is odd.*

To prove the Lemma, note that the odd numbers are, by definition, the numbers of the form $2k + 1$ where k is an integer. But

$$(2k+1)(2k'+1) = 2(2kk' + k + k') + 1,$$

so the product of two odd numbers also has the form of an odd number, which proves the Lemma.

Now we will prove the Fact using the induction hypothesis

$$P(n) ::= \text{if } a \text{ is an odd integer, then so is } a^n.$$

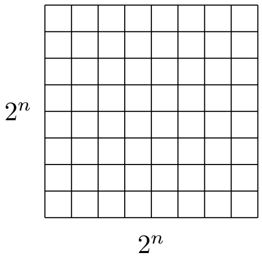
The base case $P(0)$ holds because $a^0 = 1$, and 1 is odd.

For the inductive step, suppose $n \geq 0$, a is an odd number and $P(n)$ holds. So a^n is an odd number. Therefore, $a^{n+1} = a^n a$ is a product of odd numbers, and by the Lemma a^{n+1} is also odd. This proves $P(n+1)$, and we conclude by induction that $P(n)$ holds for nonnegative integers n.

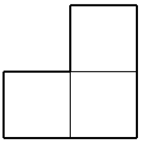
3.1.5 Courtyard Tiling

Induction served purely as a proof technique in the preceding examples. But induction sometimes can serve as a more general reasoning tool.

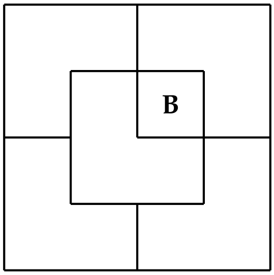
MIT recently constructed the Stata Center which houses the Computer Science and AI Laboratory. During development, the project went further and further over budget, and there were some radical fundraising ideas. One rumored plan was to install a big courtyard with dimensions $2^n \times 2^n$:



One of the central squares would be occupied by a statue of a wealthy potential donor. Let's call him "Bill". (In the special case $n = 0$, the whole courtyard consists of a single central square; otherwise, there are four central squares.) A complication was that the building's unconventional architect, Frank Gehry, supposedly insisted that only special L-shaped tiles be used:



A courtyard meeting these constraints exists, at least for $n = 2$:



For larger values of n, is there a way to tile a $2^n \times 2^n$ courtyard with L-shaped tiles and a statue in the center? Let's try to prove that this is so.

Theorem 0.2. *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.

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

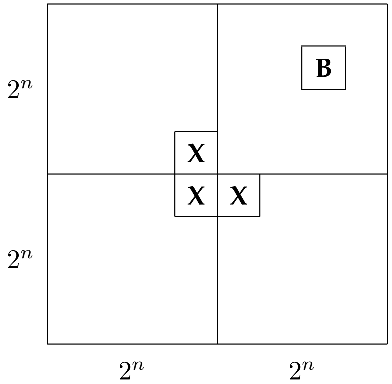
When this happens, your first fallback should be to look for a *stronger* induction hypothesis; that is, one which implies your previous hypothesis. For example, we could make $P(n)$ the proposition that for *every* location of Bill in a $2^n \times 2^n$ courtyard, there exists a tiling of the remainder.

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 $P(n) \longrightarrow P(n+1)$, 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:



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 $n \geq 0$. The theorem follows as a special case. □

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 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"[2]. 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 is easy!

3.1.6 A Faulty Induction Proof

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

[2] 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 two weeks.

False Theorem 0.3. *In every set of $n \geq 1$ horses, all 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.

False 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}, and likewise h_{n+1} is the same color as the remaining horses besides h_1. So h_1 and h_{n+1} are the same color. That is, horses $h_1, h_2, \ldots, h_{n+1}$ 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 error in this argument is in the sentence that begins, "So h_1 and h_{n+1} are the same color." The "..." notation creates the impression that there are some remaining horses besides h_1 and h_{n+1}. However, this is not true when $n = 1$. In that case, the first set is just h_1 and the second is h_2, and there are no remaining horses besides them. 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 horses of a different color.

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 6.042 exam.

Class Problems

Problem 1. Use induction to prove that

$$1^3 + 2^3 + \cdots + n^3 = \left(\frac{n(n+1)}{2}\right)^2. \tag{3.5}$$

for all $n \geq 1$.
Remember to formally

1. Declare proof by induction.
2. Identify the induction hypothesis $P(n)$.
3. Establish the base case.
4. Prove that $P(n) \Rightarrow P(n+1)$.
5. Conclude that $P(n)$ holds for all $n \geq 1$.

as in the five part template.

Problem 2. **(a)** Prove by induction that a $2^n \times 2^n$ courtyard with a 1×1 statue of Bill in *any position* can be covered with L-shaped tiles.

(b) *(Discussion Question)* In part (a) we saw that it can be easier to prove a stronger theorem. Does this surprise you? How would you explain this phenomenon?

Problem 3. Here is another exciting 6.042 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 follows:

Stack Heights										**Score**
$\underline{10}$										
5	$\underline{5}$									25 points
$\underline{5}$	3	2								6
$\underline{4}$	3	2	1							4
2	$\underline{3}$	2	1	2						4
$\underline{2}$	2	2	1	2	1					2
1	$\underline{2}$	2	1	2	1	1				1
1	1	$\underline{2}$	1	2	1	1	1			1
1	1	1	1	$\underline{2}$	1	1	1	1		1
1	1	1	1	1	1	1	1	1	1	1
								Total Score	=	45 points

On each line, the underlined stack is divided in the next step.

(a) Can you find a better strategy? Experiment with a few strategies, and before looking at the next page, see if your team can guess what's going on.

(b) As you may have guessed, the strategy is irrelevant: the score is determined solely by the number of boxes. Confirm this using strong induction to prove that the predicate

$$S(n) ::= \text{every way of unstacking } n+1 \text{ blocks gives a score of } (n+1)n/2$$

holds for all $n \in \mathbb{N}$.

Problem 4. Week 4 Notes contain a proof by induction that:

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

But now we're going to prove a *contradictory* theorem!

False Theorem. *For all $n \geq 0$,*

$$2 + 3 + 4 + \cdots + n = \frac{n(n+1)}{2}$$

Proof. We use induction. Let $P(n)$ be the proposition that $2+3+4+\cdots+n = n(n+1)/2$.
Base case: $P(0)$ is true, since both sides of the equation are equal to zero. (Recall that a sum with no terms is zero.)
Inductive step: Now we must show that $P(n)$ implies $P(n+1)$ for all $n \geq 0$. So suppose that $P(n)$ is true; that is, $2+3+4+\cdots+n = n(n+1)/2$. Then

we can reason as follows:

$$\begin{aligned} 2+3+4+\cdots+n+(n+1) &= \big[2+3+4+\cdots+n\big]+(n+1) \\ &= \frac{n(n+1)}{2}+(n+1) \\ &= \frac{(n+1)(n+2)}{2} \end{aligned}$$

Above, we group some terms, use the assumption $P(n)$, and then simplify. This shows that $P(n)$ implies $P(n+1)$. By the principle of induction, $P(n)$ is true for all $n \in \mathbb{N}$. □

Where exactly is the error in this proof?

3.2 Strong Induction

A useful variant of induction is called *strong induction*. Strong Induction and Ordinary Induction are used for exactly the same thing: proving that a predicate $P(n)$ is true for all $n \in \mathbb{N}$.

Principle of Strong Induction. Let $P(n)$ be a predicate. If

- $P(0)$ is true, and
- for all $n \in \mathbb{N}$, $P(0), P(1), \ldots, P(n)$ *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 $P(0), P(1), \ldots,$ and $P(n)$ are *all* true when you go to prove $P(n+1)$. These extra assumptions can only make your job easier.

3.2.1 Products of Primes

As a first example, we'll use strong induction to prove one of those familiar facts that is almost, but maybe not entirely, obvious:

Lemma 4.1. *Every integer greater than 1 is a product of primes.*

Note that, by convention, any number is considered to be a product consisting of one term, namely itself. In particular, every prime is considered to be a product whose terms are all primes.

Proof. We will prove Lemma 4.1 by strong induction, letting the induction hypothesis, $P(n)$, be

$$n + 2 \text{ is a product of primes.}$$

So Lemma 4.1 will follow if we prove that $P(n)$ holds for all $n \geq 0$.

Base Case: $P(0)$ is true because $0 + 2$ is prime, and so is a product of primes by convention.

Inductive step: Suppose that $n \geq 0$ and that $i + 2$ is a product of primes for every nonnegative integer $i < n+1$. We must show that $P(n+1)$ holds, namely, that $n + 3$ is also a product of primes. We argue by cases:

If $n + 3$ is itself prime, then it is a product of primes by convention, so $P(n + 1)$ holds in this case.

Otherwise, $n+3$ is not prime, which by definition means $n+3 = km$ for some integers k, m such that $2 \leq k, m < n + 3$. Now $0 \leq k - 2 < n + 1$, so by strong induction hypothesis, we know that $(k - 2) + 2 = k$ is a product of primes. Likewise, m is a product of primes. it follows immediately that $km = n + 3$ 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 nonnegative integers, n.

□

Despite the name, strong induction is actually no more powerful than ordinary induction: any theorem that can be proved with strong induction could also be proved with ordinary induction (using a slightly more complicated induction hypothesis). But strong induction can make some proofs a bit easier. On the other hand, if $P(n)$ is easily sufficient to prove $P(n + 1)$, then it's better to use ordinary induction for simplicity.

3.2.2 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 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:

> If $n \geq 8$, then there is a collection of coins whose value is n Strongs.

Notice that $P(n)$ is an implication. When the hypothesis of an implication is false, we know the whole implication is true. In this situation, the implication is said to be *vacuously* true. So $P(n)$ will be vacuously true whenever $n < 8$.[3]

We now proceed with the induction proof:

Base case: $P(0)$ is vacuously true.

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

Case ($n + 1 < 8$): $P(n + 1)$ is vacuously true in this case.

Case ($n + 1 = 8$): $P(8)$ holds because the Inductians can use one 3Sg coin and one fiveSg coins.

Case ($n + 1 = 9$): Use a three 3Sg coins.

Case ($n + 1 = 10$): Use two 5Sg coins.

Case ($n + 1 \geq 11$): Then $n \geq (n + 1) - 3 \geq 8$, so by the strong induction hypothesis, the Inductians can make change for $(n + 1) - 3$ Strong. Now by adding a 3Sg coin, they can make change for $(n + 1)$Sg.

So in any case, $P(n + 1)$ is true, and we conclude by strong induction that for all $n \geq 8$, the Inductians can make change for n Strong.

□

3.2.3 Unstacking

Here is another exciting 6.042 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.

[3] Another approach that avoids these vacuous cases is to define

$$P'(n) ::= \text{there is a collection of coins whose value is } n + 8 \text{ Strongs}$$

and prove that $P'(n)$ holds for all $n \geq 0$.

Then the game might proceed as follows:

Stack Heights										**Score**
$\underline{10}$										
5	$\underline{5}$									25 points
$\underline{5}$	3	2								6
$\underline{4}$	3	2	1							4
2	$\underline{3}$	2	1	2						4
$\underline{2}$	2	2	1	2	1					2
1	$\underline{2}$	2	1	2	1	1				1
1	1	$\underline{2}$	1	2	1	1	1			1
1	1	1	1	$\underline{2}$	1	1	1	1		1
1	1	1	1	1	1	1	1	1	1	1
						Total Score			=	45 points

On each line, the underlined stack is divided in the next step. Can you find a better strategy?

Analyzing the Game

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!

Theorem 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 is exactly the same as 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 $P(1), \dots, P(n)$ 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 $P(1), \dots, P(n)$ imply $P(n+1)$ for all $n \geq 1$. So assume that $P(1), \dots, P(n)$ 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:

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

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

Therefore, the claim is true by strong induction. □

Problem 5. Define the *potential*, $p(S)$, of a stack, S, of blocks to be $k(k+1)/2$ where k is the number of blocks in S. Define the potential, $p(A)$, of a set, A, of stacks to be the sum of the potentials of the stacks in A.
Generalize Theorem 4.2 to show that for any set, A, of stacks, if a sequence of moves starting with A leads to another set, B, of stacks, then the score for this sequence of moves is $p(A) - p(B)$.

Class Problems

3.3 Induction versus Well Ordering

The induction axiom looks nothing like the Well Ordering Principle, but these two proof methods are closely related. In fact, we can take any Induction proof and reformat it into a Well Ordering proof.

Here's how: suppose that we have a proof by Induction with induction hypothesis $P(n)$. Then we start a Well Ordering proof by assuming the set of counterexamples to P is nonempty. Then by Well Ordering there is a smallest counterexample, s, that is, a smallest s such that $P(s)$ is false.

Now we use the proof of $P(0)$ that was part of the Induction proof to conclude that s must be greater than 0. Also since s is the smallest counterexample, we can conclude that $P(s-1)$ must be true. At this point we reuse the proof of the inductive step in the Induction proof, which shows that since $P(s-1)$ true, then $P(s)$ is also true. This contradicts the assumption that $P(s)$ is false, so we have the contradiction needed to complete the Well Ordering Proof that $P(n)$ holds for all $n \in \mathbb{N}$.

Problem 6. Conversely, use Strong Induction to prove the Well Ordering Principle. *Hint:* Prove that if a set of nonnegative integers contains an integer, n, then it has a smallest element.

Mathematicians commonly use the Well Ordering Principle because it can lead to shorter proofs than induction. On the other hand, well ordering proofs typically involve proof by contradiction, so using it is not always the best approach. The choice of method is really a matter of style—but style does matter.

Class Problems

3.4 Structural Induction

3.4.1 Recursive Data Types

Recursive data types play a central role in programming. From a Mathematical point of view, recursive data types are what induction is about. Recursive data types are specified by *recursive definitions* that say how to build something from its parts. These definitions have two parts:

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

3.4.2 Strings of Parentheses

Let prns be the set of all strings of parentheses. For example, the following two strings are in prns:

$$())((((())) \quad \text{and} \quad ((())())() \tag{3.6}$$

Since we're just starting to study recursive data, just for practice we'll formulate prns as a recursive data type,

Definition 6.1. The data type, prns, of strings of parentheses is defined recursively:

- **Base case:** The *empty string*, λ, is in prns.
- **Constructor case:** If $s \in$ prns, then $s)$ and $s($ are in prns.

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

A string, $s \in$ prns, a called a *matched string* if its parentheses "match up" in the usual way. For example, the left hand string above is not matched because its second right parenthesis does not have a matching left parenthesis. The string on the right is matched.

One precise way to determine if a string is matched is to start with 0 and read the string from left to right, adding 1 to the count for each left parenthesis and subtracting 1 from the count for each right parenthesis. For example, here are the counts for the two strings above

$$\begin{array}{ccccccccccccc} & (&) &) & (& (& (& (& (&) &) &) &) \\ 0 & 1 & 0 & -1 & 0 & 1 & 2 & 3 & 4 & 3 & 2 & 1 & 0 \end{array}$$

$$\begin{array}{ccccccccccc} & (& (& (&) &) & (&) &) & (&) \\ 0 & 1 & 2 & 3 & 2 & 1 & 2 & 1 & 0 & 1 & 0 \end{array}$$

A string has a *good count* if its running count never goes negative and ends with 0. So the second string above has a good count, but the first one does not because its count went negative at the third step.

Definition 6.2. Let

$$\text{GoodCount} ::= \{s \in \text{prns} \mid s \text{ has a good count}\}.$$

The matched strings can now be characterized precisely as this set of strings with good counts. But it turns out to be really useful to characterize the matched strings in another way as well, namely, as a recursive data type:

Definition 6.3. Recursively define the set, RecMatch, of strings as follows:

- **Base case:** $\lambda \in \text{RecMatch}$.
- **Constructor case:** If $s, t \in \text{RecMatch}$, then

$$(s)t \in \text{RecMatch}.$$

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

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

$$(\lambda)\lambda = () \in \text{RecMatch}$$

by the Constructor case. So now,

$$\begin{aligned} (\lambda)() = ()() &\in \text{RecMatch} & &(\text{letting } s = \lambda, t = ()) \\ (())\lambda = (()) &\in \text{RecMatch} & &(\text{letting } s = (), t = \lambda) \\ (())() &\in \text{RecMatch} & &(\text{letting } s = (), t = ()) \end{aligned}$$

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

Quickie: Verify that $((())())() \in \text{RecMatch}$.

It may not be obvious, but $\text{RecMatch} = \text{GoodCount}$. We'll confirm this later.

3.4.3 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 6.4. The set, Aexp, of *Arithmetic expressions* in the variable, x, is defined recursively as follows:

- **Base cases:**

  1. The variable, x, is in Aexp.
  2. The arabic numeral, $\mathtt{k}$, for any nonnegative integer, k, is in Aexp.

- **Constructor cases:** If $e, f \in$ 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

$$((\mathsf{3} * (x * x)) + ((\mathsf{2} * x) + \mathsf{1})). \tag{3.7}$$

These parentheses and $*$'s clutter up examples, so we'll often use simpler expressions like "$3x^2 + 2x + 1$" instead of (3.7). But it's important to recognize that $3x^2 + 2x + 1$ is not an Aexp; it's an *abbreviation* for an Aexp.

3.4.4 The Nonnegative Integers

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

Definition 6.5. The set, $\mathbb{N}$, is a data type defined recursivly as:

- $0 \in \mathbb{N}$.
- If $n \in \mathbb{N}$, then the *successor*, $n + 1$, of n is in $\mathbb{N}$.

3.4.5 Structural Induction on Recursive Data Types

Structural induction is a method for proving some property, P, of 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 the recursively defined matched strings always have an equal number of left and right parentheses. To do this, define a predicate, P, on strings $s \in$ prns:

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

Proof. We'll prove that $P(s)$ holds for all $s \in \text{RecMatch}$ by structural induction on the definition that $s \in \text{RecMatch}$, using $P(s)$ as the induction hypothesis.

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

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 parentheses in s and t. So the number of left parentheses 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 parentheses in s is n_s, and likewise, the number of right parentheses in t is n_t. So the number of right parentheses in r is $1+n_s+n_t$, which is the same as the number of left parentheses. This proves $P(r)$. We conclude by structural induction that $P(s)$ holds for all $s \in \text{RecMatch}$. □

Problem 7. (a) Give an easy proof using structural induction that every recursively defined match string has a good count. That is

$$\text{RecMatch} \subseteq \text{GoodCount}\,.$$

(b) Conversely, prove by induction on the length of strings that the every string with a good count has a recursive definition, that is,

$$\text{GoodCount} \subseteq \text{RecMatch}\,.$$

By the way, you should notice that ordinary induction is simply the special case of structural induction on the recursive Definition 6.5 of $\mathbb{N}$!

3.4.6 Functions on Recursively-defined Data Types

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, from the recursive definition of the set, RecMatch, of strings of matched parentheses, we define:

Definition 7.6. The *depth*, $d(s)$, of a string, $s \in \text{RecMatch}$, is defined recursively by the rules:

- $d(\lambda) ::= 0.$
- $d((s)t) ::= \max\{d(s)+1, d(t)\}$

Warning: 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 yet another definition of the matched strings.

Example 7.7. Define the set, $M \subseteq$ prns 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.

Problem 8. Give an easy proof by structural induction that $M = \text{RecMatch}$.

Since $M = \text{RecMatch}$, and the definition of M seems more straightforward, why didn't we use it? Because the definition of M is ambiguous, while the trickier definition of RecMatch is unambiguous. Does this ambiguity matter? Yes it does. For suppose we defined

$$\begin{aligned} f(\lambda) &::= 1, \\ f(\ (s)\ ) &::= 1 + f(s), \\ f(st) &::= (f(s)+1)\cdot(f(t)+1) \qquad\qquad \text{for } st \neq \lambda. \end{aligned}$$

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 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 $t = a$.

Now by these rules, $f(a) = 2$, and $f(b) = (2+1)(2+1) = 9$. This means that $f(c) = f(bb) = (9+1)(9+1) = 100$.

But also $f(ba) = (9+1)(2+1) = 27$, so that $f(c) = f(ba\,a) = (27+1)(2+1) = 84$.

The outcome is that $f(c)$ is defined to be both 100 and 84, which shows that the rules defining f are inconsistent.

On the other hand, structural induction remains a sound proof method even for ambiguous recursive definitions, which is why it was easy to prove that $M = \text{RecMatch}$.

3.4.7 Recursive Functions on Nonnegative Integers

Definition 6.5 of the nonnegative integers as a recursive data type justifies the familiar recursive definitions of functions on the nonnegative integers. Here are some examples.

The Factorial function. This function is often written "$n!$." You will see a lot of it later in the term. Here we'll use the notation fac(n):

- fac$(0) ::= 1$.
- fac$(n+1) ::= (n+1) \cdot$ fac(n) for $n \geq 0$.

The Fibonacci numbers. Fibonacci numbers arose out of an effort 800 years ago to model population growth. They have a continuing fan club of people captivated by their extraordinary properties. The nth Fibonacci number, fib, can be defined recursively by:

$$\begin{aligned} \text{fib}(0) &::= 0, \\ \text{fib}(1) &::= 1, \\ \text{fib}(n) &::= \text{fib}(n-1) + \text{fib}(n-2) \qquad \text{for } n \geq 2. \end{aligned}$$

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

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

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

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.

$$f_1(n) ::= 2 + f_1(n-1). \tag{3.8}$$

This "definition" has no base case. If some function, f_1, satisfied (3.8), so would a function obtained by adding a constant to the value of f_1. So equation (3.8) does not uniquely define an f_1.

$$f_2(n) ::= \begin{cases} 0, & \text{if } n = 0, \\ f_2(n+1) & \text{otherwise.} \end{cases} \tag{3.9}$$

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 (3.9) 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 (3.9) as defining a *partial* function, f_2, that is undefined everywhere but 0.

$$f_3(n) ::= \begin{cases} 0, & \text{if } n \text{ is divisible by 2,} \\ 1, & \text{if } n \text{ is divisible by 3,} \\ 2, & \text{otherwise.} \end{cases} \tag{3.10}$$

This "definition" is inconsistent: it requires $f_3(6) = 0$ and $f_3(6) = 1$, so (3.10) doesn't define anything.

A Mysterious Function

Mathematicians have been wondering about this function specification for a while:

$$f_4(n) ::= \begin{cases} 1, & \text{if } n \le 1, \\ f_4(n/2) & \text{if } n > 1 \text{ is even,} \\ f_4(3n+1) & \text{if } n > 1 \text{ is odd.} \end{cases} \tag{3.11}$$

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 (3.11), 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 (3.11) equals 1 for all n up to over a billion.

Quick exercise: Why does the constant function 1 satisfy (3.11)?

3.4.8 Evaluation and Substitution with Aexp's

Evaluating Aexp's

Since the only variable in an Aexp is x, the value of an Aexp is determined by the value of x. For example, if the value of x is 3, then the value of $3x^2 + 2x + 1$ is obviously 34. In general, given any Aexp, e, and an integer value, n, for the variable, x, we can evaluate e to finds its value, $\text{eval}(e, n)$. It's easy, and useful, to specify this evaluation process with a recursive definition.

Definition 8.8. The *evaluation function*, $\text{eval} : \text{Aexp} \times \mathbb{Z} \to \mathbb{Z}$, is defined recursively on expressions, $e \in \text{Aexp}$, 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 $\mathtt{k}$]

     $$\text{eval}(\mathtt{k}, n) ::= k.$$

     (The value of the numeral $\mathtt{k}$ 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 $--(e_1)$]

     $$\text{eval}(--(e_1), n) ::= -\,\text{eval}(e_1, n).$$

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

$$\begin{aligned}
\text{eval}((\mathtt{3} + (x * x)), 2) &= \text{eval}(\mathtt{3}, 2) + \text{eval}((x * x), 2) && \text{(by Def 8.8.3)}\\
&= 3 + \text{eval}((x * x), 2) && \text{(by Def 8.8.2)}\\
&= 3 + (\text{eval}(x, 2) \cdot \text{eval}(x, 2)) && \text{(by Def 8.8.4)}\\
&= 3 + (2 \cdot 2) && \text{(by Def 8.8.1)}\\
&= 3 + 4 = 7.
\end{aligned}$$

Substituting into Aexp's

Substituting expressions for variables is a standard, important operation. For example the result of substituting the expression $3x$ for x in the $(x(x-1))$ would be $(3x(3x-1)$. We'll use the general notation $\text{subst}(f, e)$ for the the result of substituting an Aexp, f, for each of the x's in an Aexp, e. For instance,

$$\text{subst}(3x, x(x-1)) = 3x(3x-1).$$

This substitution function has a simple recursive definition:

Definition 8.9. The *substitution function* from Aexp × Aexp to Aexp is defined recursively on expressions, $e \in$ Aexp, as follows. Let f be any Aexp.

- **Base cases:**

  1. Case[e is x]

  $$\text{subst}(f, x) ::= f.$$

  (The result of substituting f for the variable, x, is just f.)

  2. Case[e is $\texttt{k}$]

  $$\text{subst}(f, \texttt{k}) ::= \texttt{k}.$$

  (The numeral, k, has no x's in it to substitute for.)

- **Constructor cases:**

  3. Case[e is $(e_1 + e_2)$]

  $$\text{subst}(f, (e_1 + e_2))) ::= (\text{subst}(f, e_1) + \text{subst}(f, e_2)).$$

  4. Case[e is $(e_1 * e_2)$]

  $$\text{subst}(f, (e_1 * e_2))) ::= (\text{subst}(f, e_1) * \text{subst}(f, e_2)).$$

  5. Case[e is $--(e_1)$]

  $$\text{subst}(f, --(e_1)) ::= --(\text{subst}(f, e_1)).$$

Here's how the recursive definition of the substitution function would find the result of substituting $3x$ for x in the $x(x-1)$:

$$\begin{aligned}
\text{subst}(3x, (x(x-1))) &= \text{subst}(3x, (x * (x + --(1)))) && \text{(unabbreviating)}\\
&= (\text{subst}(3x, x) * \text{subst}(3x, (x + --(1)))) && \text{(by Def 8.9 4)}\\
&= (3x * \text{subst}(3x, (x + --(1)))) && \text{(by Def 8.9 1)}\\
&= (3x * (\text{subst}(3x, x) + \text{subst}(3x, --(1)))) && \text{(by Def 8.9 3)}\\
&= (3x * (3x + --(\text{subst}(3x, 1)))) && \text{(by Def 8.9 1 \& 5)}\\
&= (3x * (3x + --(1))) && \text{(by Def 8.9 2)}\\
&= 3x(3x-1) && \text{(abbreviation)}
\end{aligned}$$

Now suppose we have to find the value of $\text{subst}(3x, (x(x-1)))$ when $x = 2$. There are two approaches.

First, we could actually do the substitution above to get $3x(3x-1)$, and then we could evaluate $3x(3x-1)$ when $x = 2$, that is, we could recursively calculate $\text{eval}(3x(3x-1), 2)$ to get the final value 30. In programming jargon, this would be called evaluation using the *Substitution Model*. Tracing through the steps in the evaluation, we find that the Substitution Model requires two substitutions for occurrences of x and 5 integer operations: 3 integer multiplications, 1 integer addition, and 1 integer negative operation. Note that in this Substitution Model the multiplication $3 \cdot 2$ was performed twice to get the value of 6 for each of the two occurrences of $3x$.

The other approach is called evaluation using the *Environment Model*. Namely, we evaluate $3x$ when $x = 2$ using just 1 multiplication to get the value 6. Then we evaluate $x(x-1)$ when x has this value 6 to arrive at the value $6 \cdot 5 = 30$. So the Environment Model requires 2 variable lookups and only 4 integer operations: 1 multiplication to find the value of $3x$, another multiplication to find the value $6 \cdot 5$, along with 1 integer addition and 1 integer negative operation.

So the Environment Model approach of calculating

$$\text{eval}(x(x-1), \text{eval}(3x, 2))$$

instead of the Substitution Model approach of calculating

$$\text{eval}(\text{subst}(3x, x(x-1)), 2)$$

is faster. But how do we know that these final values reached by these two approaches always agree? We can prove this easily by structural induction on the definitions of the two approaches. More precisely, what we want to prove is

Theorem 8.10. *For all expressions $e, f \in$ Aexp and $n \in \mathbb{Z}$,*

$$\text{eval}(\text{subst}(f, e), n) = \text{eval}(e, \text{eval}(f, n)). \tag{3.12}$$

Proof. The proof is by structural induction on e.[4]

Base cases:

- Case[e is x]

  The left hand side of equation (3.12) equals $\text{eval}(f, n)$ by this base case in Definition 8.9 of the substitution function, and the right hand side also equals $\text{eval}(f, n)$ by this base case in Definition 8.8 of eval.

- Case[e is $\texttt{k}$].

  The left hand side of equation (3.12) equals $\texttt{k}$ by this base case in Definitions 8.9 and 8.8 of the substitution and evaluation functions. Likewise, the right hand side equals $\texttt{k}$ by two applications of this base case in the Definition 8.8 of eval.

[4] This is an example of why it's useful to notify the reader what the induction variable is —in this case it isn't n.

Constructor cases:

- Case[e is $(e_1 + e_2)$]

  By the structural induction hypothesis (3.12), we may assume that for all $f \in$ Aexp and $n \in \mathbb{Z}$,

  $$\text{eval}(\text{subst}(f, e_i), n) = \text{eval}(e_i, \text{eval}(f, n)) \tag{3.13}$$

  for $i = 1, 2$. We wish to prove that

  $$\text{eval}(\text{subst}(f, (e_1 + e_2)), n) = \text{eval}((e_1 + e_2), \text{eval}(f, n)) \tag{3.14}$$

  But the left hand side of (3.14) equals

  $$\text{eval}(\ (\text{subst}(f, e_1) + \text{subst}(f, e_2)),\ n)$$

  by Definition 8.9.3 of substitution into a sum expression. But this equals

  $$\text{eval}(\text{subst}(f, e_1), n) + \text{eval}(\text{subst}(f, e_2), n)$$

  by Definition 8.8.3 of eval for a sum expression. By induction hypothesis (3.13), this in turn equals

  $$\text{eval}(e_1, \text{eval}(f, n)) + \text{eval}(e_2, \text{eval}(f, n)).$$

  Finally, this last expression equals the right hand side of (3.14) by Definition 8.8.3 of eval for a sum expression. This proves (3.14) in this case.

- e is $(e_1 * e_2)$. Similar.

- e is $-(e_1)$. Even easier.

This covers all the constructor cases, and so completes the proof by structural induction.

□

Class Problems

Problem 9. The Elementary 18.01 Functions (F18's) are the set of functions of one real variable defined recursively as follows:
Base cases:

- The identity function, id$(x) ::= x$ is an F18,

- any constant function is an F18,
- the sine function is an F18,

Constructor cases:
If f, g are F18's, then so are

1. $f + g$, fg, e^g (the constant e),
2. the inverse function f^{-1},
3. the composition $f \circ g$.

(a) Prove that the function $1/x$ is an F18.

Warning: Don't confuse $1/x = x^{-1}$ with the inverse, id^{-1} of the identity function $\text{id}(x)$. The inverse id^{-1} is equal to id.

(b) Prove by Structural Induction on this definition that the Elementary 18.01 Functions are *closed under taking derivatives*. That is, show that if $f(x)$ is an F18, then so is $f' ::= df/dx$. (Just work out 2 or 3 of the most interesting constructor cases; you may skip the less interesting ones.)

Problem 10. Here is a simple recursive definition of the set, E, of even integers:

Definition. Base case: $0 \in E$.
Constructor cases: If $n \in E$, then so are $n + 2$ and $-n$.

Provide similar simple recursive definitions of the following sets:

(a) The set $S ::= \left\{2^k 3^m 5^n \mid k, m, n \in \mathbb{N}\right\}$.

(b) The set $T ::= \left\{2^k 3^{2k+m} 5^{m+n} \mid k, m, n \in \mathbb{N}\right\}$.

(c) The set $L ::= \left\{(a, b) \in \mathbb{Z}^2 \mid 3 \mid (a - b)\right\}$.

Let L' be the set defined by the recursive definition you gave for L in the previous part. Now if you did it right, then $L' = L$, but maybe you made a mistake. So let's check that you got the definition right.

(d) Prove by structural induction on your definition of L' that

$$L' \subseteq L.$$

(e) *Optional: come back to this part only if you finish all the remaining problems.* Confirm that you got the definition right by proving that

$$L \subseteq L'.$$

(f) *Optional: come back to this part only if you finish all the remaining problems.* Give an *unambiguous* recursive definition of L.

Problem 11. Let p be the string $()$. A string of parentheses is said to be *erasable* iff it can be reduced to the empty string by repeatedly erasing occurrences of p. For example, here's how to erase the string $((())())()$:

$$((())())() \to (()) \to () \to \lambda.$$

On the other hand the string $())((((())$ is not erasable because when we try to erase, we get stuck at $)((($:

$$())((((()) \to)(((() \to)(((\not\to$$

Let Erasable be the set of erasable strings of parentheses. Let RecMatch be the recursive data type of strings of *matched* parentheses. (The definition of RecMatch is repeated in the Appendix.)

(a) Use structural induction to prove that

$$\text{RecMatch} \subseteq \text{Erasable}\,.$$

(b) Supply the missing parts of the following proof that

$$\text{Erasable} \subseteq \text{RecMatch}\,.$$

Proof. We prove by induction on the length, n, of strings, x, that if $x \in \text{Erasable}$, then $x \in \text{RecMatch}$. The induction predicate is

$$P(n) ::= \forall x \in \text{Erasable}\,.\,[|x| \leq n \quad \text{IMPLIES} \quad x \in \text{RecMatch}]$$

Base case:

What is the base case? Prove that P is true in this case.

Inductive step: To prove $P(n+1)$, suppose $|x| \leq n+1$ and $x \in \text{Erasable}$. We need only show that $x \in \text{RecMatch}$. Now if $|x| < n+1$, then the induction hypothesis, $P(n)$, implies that $x\,\text{RecMatch}$, so we only have to deal with x of length exactly $n+1$.

Let's say that a string y is an *erase* of a string z iff y is the result of erasing a single occurrence of p in z.

Since $x \in \text{Erasable}$ and has positive length, there must be an erase, $y \in \text{Erasable}$, of x. So $|y| = n-1$, and since $y \in \text{Erasable}$, we may assume by induction hypothesis that $y \in \text{RecMatch}$.

Now we argue by cases:

Case [y is the empty string].

Prove that $x \in \text{RecMatch}$ in this case.

Case [$y = (s)t$ for some strings $s, t \in \text{RecMatch}$.] Now we argue by subcases.

- **Subcase** [x is of the form $(s')t$ where s is an erase of s'].
  Since $s \in \text{RecMatch}$, it is erasable by part (b), which implies that $s' \in \text{Erasable}$. But $|s'| < |x|$, so by induction hypothesis, we may assume that $s' \in \text{RecMatch}$. This shows that x is the result of the constructor step of RecMatch, and therefore $x \in \text{RecMatch}$.
- **Subcase** [x is of the form $(s)t'$ where t is an erase of t'].
  Prove that $x \in \text{RecMatch}$ **in this subcase.**
- **Subcase**[$x = p(s)t$].
  Prove that $x \in \text{RecMatch}$ **in this subcase.**

The proofs of the remaining subcases are just like this last one. **List these remaining subcases.**

This completes the proof by induction on n, so we conclude that $P(n)$ holds for all $n \in \mathbb{N}$. Therefore $x \in \text{RecMatch}$ for every string $x \in \text{Erasable}$. That is,

$$\text{Erasable} \subseteq \text{RecMatch} \text{ and hence } \text{Erasable} = \text{RecMatch}\,.$$

□

Problem 12.

Definition 12.11. The recursive data type, binary-2PTG, of *binary trees* with leaf labels, L is defined recursively as follows:

- **Base case:** $\langle \texttt{leaf}, l \rangle \in$ binary-2PTG, for all $l \in L$.
- **Constructor case:** If $G_1, G_2 \in$ binary-2PTG, then

$$\langle \texttt{bintree}, G_1, G_2 \rangle \in \text{binary-2PTG}.$$

The *size*, $|G|$, of $G \in$ binary-2PTG is defined recursively on this definition by:

- **Base case:**

$$|\langle \texttt{leaf}, l \rangle| ::= ::= l, \quad \text{for all} l \in L.$$

- **Constructor case:**

$$|\langle \texttt{bintree}, G_1, G_2 \rangle| ::= |G_1| + |G_2| + 1.$$

For example, for the size of the binary-2PTG, G, pictured in Figure 3.1, is 7.

(a) Write out (using angle brackets and labels `bintree`, `leaf`, etc.) the binary-2PTG, G, pictured in Figure 3.1.

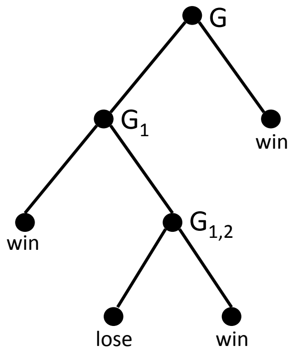


Figure 3.1: *A picture of a binary tree w.*

The value of flatten(G) for $G \in$ binary-2PTG is the sequence of labels in L of the leaves of G. For example, for the binary-2PTG, G, pictured in Figure 3.1,

$$\text{flatten}(G) = (\texttt{win}, \texttt{lose}, \texttt{win}, \texttt{win}).$$

(b) Give a recursive definition of flatten. (You may use the operation of *concatenation* (append) of two sequences.)

(c) Prove by structural induction on the definitions of flatten and size that

$$2 \cdot \text{length}(\text{flatten}(G)) = |G| + 1. \tag{3.15}$$

3.5 Games as a Recursive Data Type

Chess, Checkers, and Tic-Tac-Toe are examples of *two-person terminating games of perfect information*, —2PTG's for short. These are games in which two players alternate moves that depend only on the visible board position or state of the game. "Perfect information" means that the players know the complete state of the game at each move. (Most card games are *not* games of perfect information because neither player can see the other's hand.) "Terminating" means that play cannot go on forever —it must end after a finite number of moves.[5]

[5] Since board positions can repeat in chess and checkers, termination is enforced by rules that prevent any position from being repeated more than a fixed number of times. So the

We will define 2PTG's as a recursive data type. To see how this will work, let's use the game of Tic-Tac-Toe as an example.

3.5.1 Tic-Tac-Toe

Tic-Tac-Toe is a game for young children. There are two players who alternately write the letters "X" and "O" in the empty boxes of a 3×3 grid. Three copies of the same letter filling a row, column, or diagonal of the grid is called a *tic-tac-toe*, and the first player who gets a tic-tac-toe of their letter wins the game.

We're now going give a precise mathematical definition of the Tic-Tac-Toe *game tree* as a recursive data type. Here's the idea behind the definition: at any point in the game, the "board position" is the pattern of X's and O's on the 3×3 grid. From any such Tic-Tac-Toe pattern, there are a number of next patterns that might result from a move. For example, from the initial empty grid, there are nine possible next patterns, each with a single X in some grid cell and the other eight cells empty. From any of these patterns, there are eight possible next patterns gotten by placing an O in an empty cell. These move possibilities are given by the game tree for Tic-Tac-Toe indicated in Figure 3.2.

Definition 12.1. A Tic-Tac-Toe *pattern* is a 3×3 grid each of whose 9 cells contains either the single letter, X, the single letter, O, or is empty.

A pattern, Q, is a *possible next pattern after* P, providing P has no tic-tac-toes and

- if P has an equal number of X's and O's, and Q is the same as P except that a cell that was empty in P has an X in Q, or

- if P has one more X than O's, and Q is the same as P except that a cell that was empty in P has an O in Q.

If P is a Tic-Tac-Toe pattern, and P has no next patterns, then the *terminated Tic-Tac-Toe game trees* at P are

- $$\langle P, \langle \texttt{win} \rangle \rangle ,$$
  if P has a tic-tac-toe of X's.

- $$\langle P, \langle \texttt{lose} \rangle \rangle ,$$
  if P has a tic-tac-toe of O's.

- $$\langle P, \langle \texttt{tie} \rangle \rangle ,$$
  otherwise.

"state" of these games is the board position *plus* a record of how many times positions have been reached.

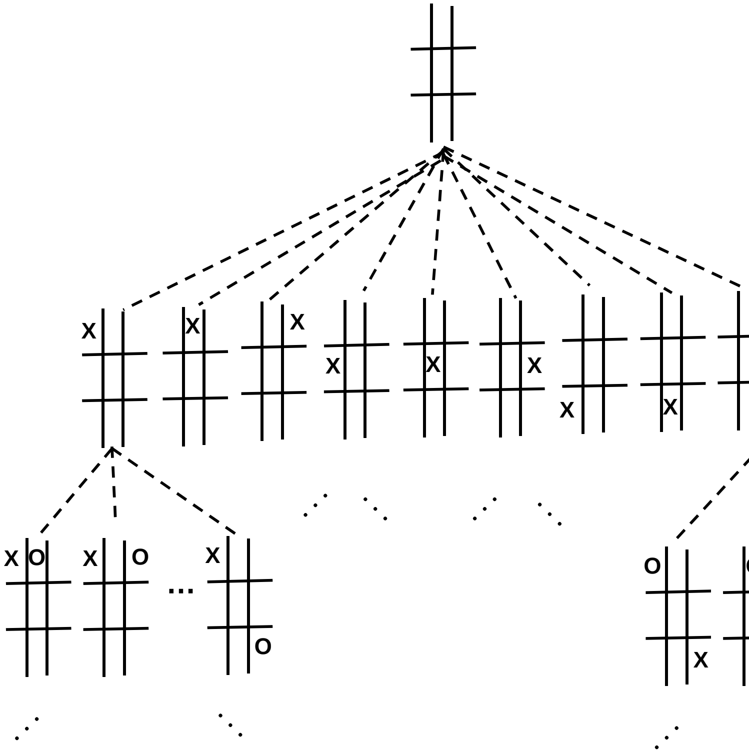


Figure 3.2: The Top of the Game Tree for Tic-Tac-Toe.

The *Tic-Tac-Toe game trees starting at P* are defined recursively:

Base Case: A terminated Tic-Tac-Toe game tree at P is a Tic-Tac-Toe game tree starting at P.

Constructor case: If P is a non-terminated Tic-Tac-Toe pattern, then the Tic-Tac-Toe game tree starting at P consists of P and the set of all games trees starting at possible next patterns after P.

For example, if

$$P_0 = \begin{array}{c|c|c} O & X & O \\ \hline X & O & X \\ \hline X & & \end{array}$$

$$Q_1 = \begin{array}{c|c|c} O & X & O \\ \hline X & O & X \\ \hline X & & O \end{array}$$

$$Q_2 = \begin{array}{c|c|c} O & X & O \\ \hline X & O & X \\ \hline X & O & \end{array}$$

$$R = \begin{array}{c|c|c} O & X & O \\ \hline X & O & X \\ \hline X & O & X \end{array}$$

the game tree starting at P_0 is pictured in Figure 3.3.

Game trees are usually pictured in this way with the starting pattern (referred to as the "root" of the tree) at the top and lines connecting the root to the game trees that start at each possible next pattern. The "leaves" at the bottom of the tree (trees grow upside down in Computer Science) correspond to terminated games. A path from the root to a leaf describes a complete *play* of the game. (In English, "game" can be used in two senses: first we can say that Chess is a game, and second we can play a game of Chess. The first usage refers to the data type of Chess game trees, and the second usage refers to a "play.")

3.5.2 Infinite Tic-Tac-Toe Games

At any point in a Tic-Tac-Toe game, there are at most nine possible next patterns, and no play can continue for more than nine moves. But suppose we consider an *n-game Tic-Tac-Toe tournament* where the tournament winner is the one who wins more of $n > 0$ individual Tic-Tac-Toe games. (If they each win an equal number of individual games, then the tournament is a tie.)

Now we can consolidate all these tournaments into a single game we can call *Tournament-Tic-Tac-Toe*: the first player in Tournament-Tic-Tac-Toe chooses any integer $n > 0$, and then the players play an n-game tournament. Now there are infinitely many possible first moves: the first player can choose $n = 1$, or $n = 2$, or $n = 3$, or $\ldots$. But still, it's obvious that

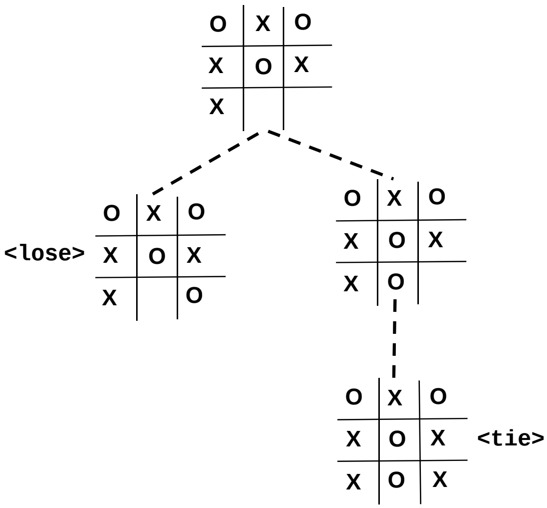


Figure 3.3: Game Tree for the Tic-Tac-Toe game starting at P_0.

every possible play of Tournament-Tic-Tac-Toe is finite, because after the first player chooses a value for n, the game can't continue for more than $9n$ moves. So it's not possible to keep playing forever even though the game tree is infinite.

This isn't very hard to understand, but there is an important difference between any given n-game tournament and Tournament-Tic-Tac-Toe: even though every play of Tournament-Tic-Tac-Toe must come to an end, there is no longer any bound on how many moves it might be before the game ends —a play might end after 9 moves, or $9(2001)$ moves, or $9(10^{10}+1)$ moves; it just can't continue forever.

With Tournament-Tic-Tac-Toe recognized as a 2PTG, we can go on to Tournament^2-Tic-Tac-Toe where the first player chooses a number, $m > 0$, of Tournament-Tic-Tac-Toe games to play, and the second player acts as the first player in each of the m Tournament-Tic-Tac-Toe games to be played. Then, of course, there's Tournament^3-Tic-Tac-Toe. . . .

3.5.3 Two Person Terminating Games

Familiar games like Tic-Tac-Toe, Checkers, and Chess can all end in ties, but for simplicity we'll only consider win/lose games —no "everybody wins"-type games at MIT. `:-)`. But everything we show about win/lose games will extend easily to games with ties.

Like Tic-Tac-Toe, the idea behind the definition of 2PTG's as a recursive data type is that making a move in a 2PTG leads to the start of a subgame. For Tic-Tac-Toe, we used the patterns and the rules of Tic-Tac-Toe to determine the next patterns. But once we have a complete game tree, we don't really need the pattern labels: the root of a game tree itself can play the role of a "board position" with its possible "next positions" determined by the roots of its subtrees. This leads to the following very simple —perhaps deceptively simple —general definition.

Definition 12.2. The 2PTG, *game trees for two-person terminating games of perfect information* are defined recursively as follows:

- **Base cases:**
$$\begin{aligned} \langle \texttt{leaf}, \texttt{win} \rangle &\in \text{2PTG, and} \\ \langle \texttt{leaf}, \texttt{lose} \rangle &\in \text{2PTG.} \end{aligned}$$

- **Constructor case:** If $\mathcal{G}$ is a nonempty set of 2PTG's, then G is a 2PTG, where
$$G ::= \langle \texttt{tree}, \mathcal{G} \rangle .$$

  The games trees in $\mathcal{G}$ are called the possible *next moves* from G.

These games are called "terminating" because, even though a 2PTG may be a (very) infinite datum like Tournament^2-Tic-Tac-Toe, every play of a 2PTG must terminate. This is something we can now prove, after we give a precise definition of "play":

Definition 12.3. A *play* of a 2PTG, G, is a (potentially infinite) sequence of 2PTG's starting with G and such that if G_1 and G_2 are consecutive 2PTG's in the play, then G_2 is a possible next move of G_1.

If a 2PTG has no infinite play, it is called a *terminating* game.

Theorem 12.4. *Every 2PTG is terminating.*

Proof. By structural induction on the definition of a 2PTG, G, with induction hypothesis

$$G \text{ is terminating.}$$

Base case: If $G = \langle \texttt{leaf}, \texttt{win} \rangle$ or $G = \langle \texttt{leaf}, \texttt{lose} \rangle$ then the only possible play of G is the length one sequence consisting of G. Hence G terminates.

Constructor case: For $G = \langle \texttt{tree}, \mathcal{G} \rangle$, we must show that G is terminating, given the Induction Hypothesis that *every* $G' \in \mathcal{G}$ is terminating.

But any play of G is, by definition, a sequence starting with G and followed by a play starting with some $G_0 \in \mathcal{G}$. But G_0 is terminating, so the play starting at G_0 is finite, and hence so is the play starting at G.

This completes the structural induction, proving that every 2PTG, G, is terminating. □

3.5.4 Game Strategies

A key question about a game is whether a player has a winning strategy. A *strategy* for a player in a game specifies which move the player should make at any point in the game. A *winning* strategy ensures that the player will win no matter what moves the other player makes.

In Tic-Tac-Toe for example, most elementary school children figure out strategies for both players that each ensure that the game ends with no tic-tac-toes, that is, it ends in a tie. Of course the first player can win if his opponent plays childishly, but not if the second player follows the proper strategy. In more complicated games like Checkers or Chess, it's not immediately clear that anyone has a winning strategy, even if we agreed to count ties as wins for the second player.

But structural induction makes it easy to prove that in any 2PTG, *somebody* has the winning strategy!

Theorem 12.5. ***Fundamental Theorem for Two-Person Games:*** *For every two-person terminating game of perfect information, there is a winning strategy for one of the players.*

Proof. The proof is by structural induction on the definition of a 2PTG, G. The induction hypothesis is that there is a winning strategy for G.

Base cases:

1. $G = \langle \texttt{leaf}, \texttt{win} \rangle$. Then the first player has the winning strategy: "make the winning move."

2. $G = \langle \texttt{leaf}, \texttt{lose} \rangle$. Then the second player has a winning strategy: "Let the first player make the losing move."

Constructor case: Suppose $G = \langle \texttt{tree}, \mathcal{G} \rangle$. By structural induction, we may assume that some player has a winning strategy for each $G' \in \mathcal{G}$. There are two cases to consider:

- some $G_0 \in \mathcal{G}$ has a winning strategy for its second player. Then the first player in G has a winning strategy: make the move to G_0 and then follow the second player's winning strategy in G_0.

- every $G' \in \mathcal{G}$ has a winning strategy for its first player. Then the second player in G has a winning strategy: if the first player's move in G is to $G_0 \in \mathcal{G}$, then follow the winning strategy for the first player in G_0.

So in any case, one of the players has a winning strategy for G, which completes the proof of the constructor case.

It follows by structural induction that there is a winning strategy for every 2PTG, G. □

Notice that although Theorem 12.5 guarantees a winning strategy, its proof gives no clue which player has it. For most familiar 2PTG's like Checkers, Chess, Go, . . . , no one knows which player has a winning strategy.

Class Problems

3.6 Induction in Computer Science

Induction is a powerful and widely applicable proof technique, which is why we've devoted this entire set of Notes to it. Ordinary induction is specially helpful in the study of computation. Why? Well, ordinary induction on nonnegative integers is a "one step at a time" proof method. Computations also evolve "one step at a time." Structural induction then goes beyond natural number counting and offers a powerful handle on recursive computation.

In fact, structural induction is theoretically more powerful than ordinary induction. However, it's only more powerful when it comes to reasoning about infinite data types —like infinite trees, for example —so this greater power doesn't matter in practice. What does matter is that for recursively defined data types, structural induction is a simple and natural approach. It is a technique every Computer Scientist should embrace.

3.7 Problems

Homework Problems

Miniquiz Problems