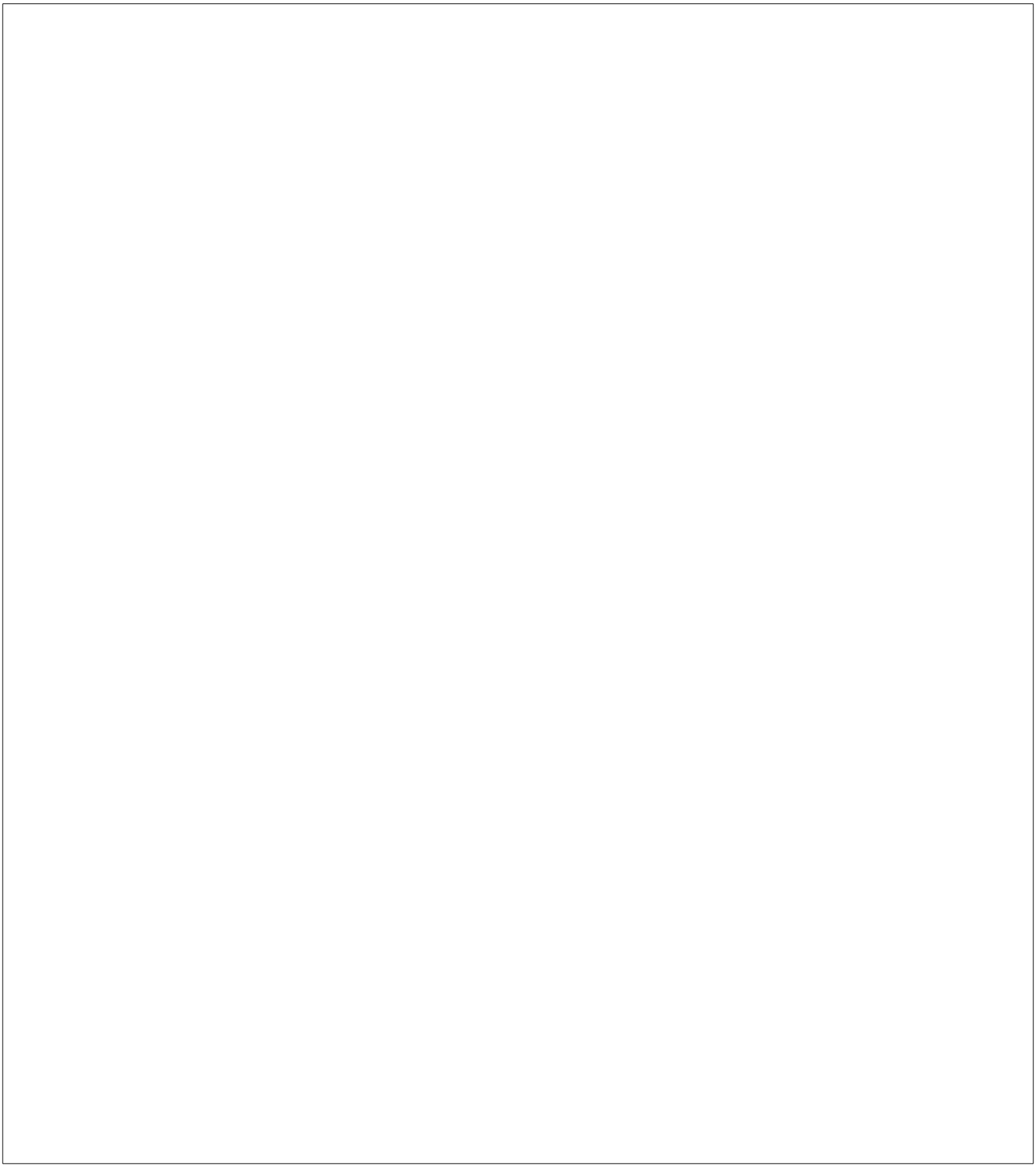
Mathematics for Computer Science

revised Wednesday 8th September, 2010, 00:40

Eric Lehman
Google Inc.

F Thomson Leighton
Department of Mathematics and CSAIL, MIT
Akamai Technologies

Albert R Meyer
Massachusets Institute of Technology

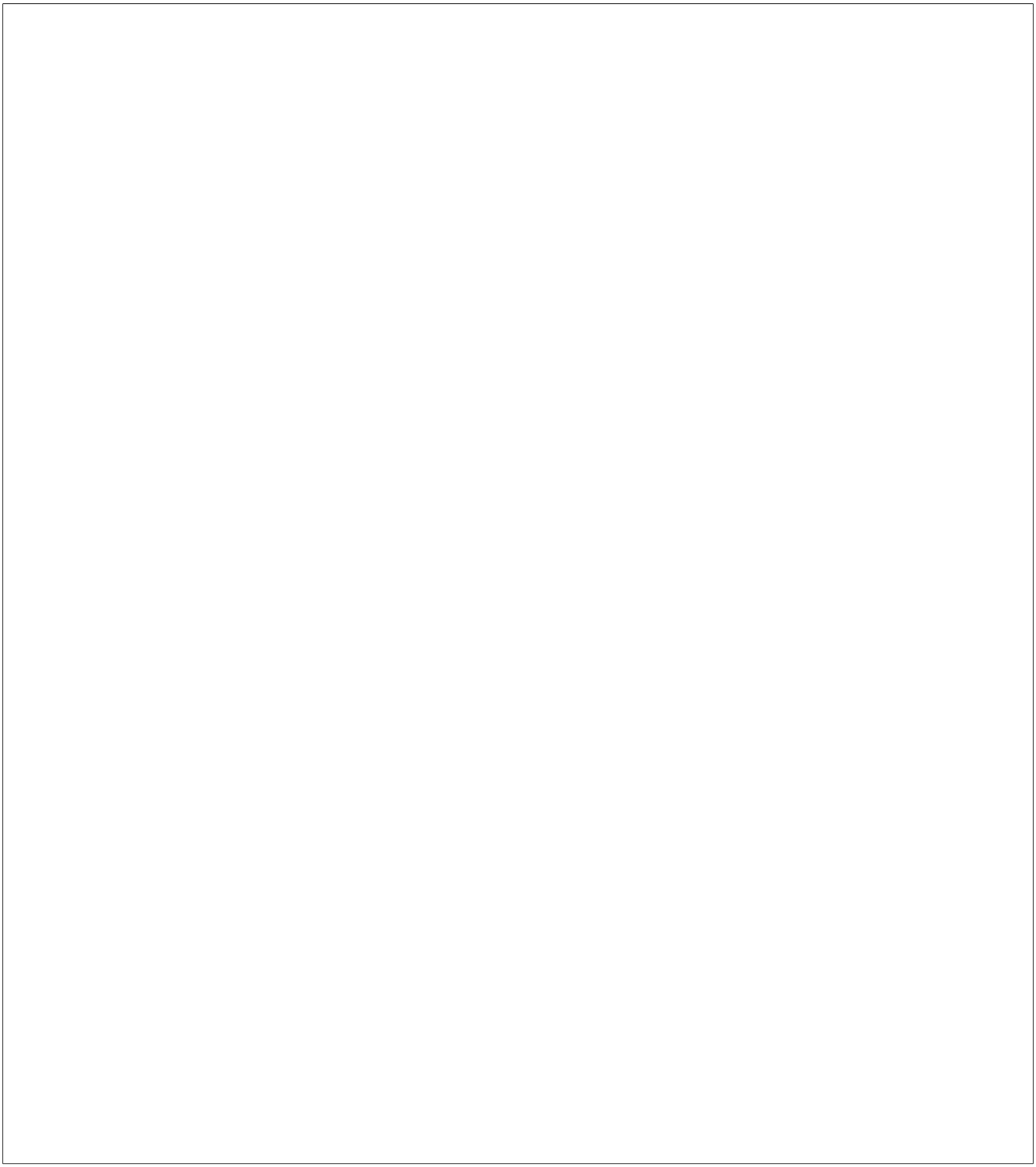


Contents

II Structures

III Counting

I Proofs



Introduction

This text explains how to use mathematical models and methods to analyze problems that arise in computer science. The notion of a proof plays a central role in this work.

Simply put, a proof is a method of establishing truth. Like beauty, "truth" sometimes depends on the eye of the beholder, and it should not be surprising that what constitutes a proof differs among fields. For example, in the judicial system, *legal* truth is decided by a jury based on the allowable evidence presented at trial. In the business world, *authoritative* truth is specified by a trusted person or organization, or maybe just your boss. In fields such as physics and biology, *scientific* truth[1] is confirmed by experiment. In statistics, *probable* truth is established by statistical analysis of sample data.

Philosophical proof involves careful exposition and persuasion typically based on a series of small, plausible arguments. The best example begins with "Cogito ergo sum," a Latin sentence that translates as "I think, therefore I am." It comes from the beginning of a 17th century essay by the mathematician/philosopher, René Descartes, and it is one of the most famous quotes in the world: do a web search on the phrase and you will be flooded with hits.

Deducing your existence from the fact that you're thinking about your existence is a pretty cool and persuasive-sounding idea. However, with just a few more lines of argument in this vein, Descartes goes on to conclude that there is an infinitely beneficent God. Whether or not you believe in a beneficent God, you'll probably agree that any very short proof of God's existence is bound to be far-fetched. So

[1] Actually, only scientific *falsehood* can be demonstrated by an experiment—when the experiment fails to behave as predicted. But no amount of experiment can confirm that the *next* experiment won't fail. For this reason, scientists rarely speak of truth, but rather of *theories* that accurately predict past, and anticipated future, experiments.

even in masterful hands, this approach is not reliable.

Mathematics has its own specific notion of "proof."

Definition. A *mathematical proof* of a *proposition* is a chain of *logical deductions* leading to the proposition from a base set of *axioms*.

The three key ideas in this definition are highlighted: proposition, logical deduction, and axiom. These three ideas are explained in the following chapters, beginning with propositions in Chapter 1. We will then provide *lots* of examples of proofs and even some examples of "false proofs" (that is, arguments that look like a proof but that contain missteps, or deductions that aren't so logical when examined closely). False proofs are often even more important as examples than correct proofs, because they are uniquely helpful with honing your skills at making sure each step of a proof follows logically from prior steps.

Creating a good proof is a lot like creating a beautiful work of art. In fact, mathematicians often refer to really good proofs as being "elegant" or "beautiful." As with any endeavor, it will probably take a little practice before your fellow students use such praise when referring to your proofs, but to get you started in the right direction, we will provide templates for the most useful proof techniques in Chapters 2 and 3. We then apply these techniques in Chapter 4 to establish some important facts about numbers; facts that form the underpinning of one of the world's most widely-used cryptosystems.

1 Propositions

Definition. A *proposition* is a statement that is either true or false.

For example, both of the following statements are propositions. The first is true and the second is false.

Proposition 1.0.1. *2 + 3 = 5.*

Proposition 1.0.2. *1 + 1 = 3.*

Being true or false doesn't sound like much of a limitation, but it does exclude statements such as, "Wherefore art thou Romeo?" and "Give me an *A*!".

Unfortunately, it is not always easy to decide if a proposition is true or false, or even what the proposition means. In part, this is because the English language is riddled with ambiguities. For example, consider the following statements:

1. "You may have cake, or you may have ice cream."
2. "If pigs can fly, then you can understand the Chebyshev bound."
3. "If you can solve any problem we come up with, then you get an *A* for the course."
4. "Every American has a dream."

What *precisely* do these sentences mean? Can you have both cake and ice cream or must you choose just one dessert? If the second sentence is true, then is the Chebyshev bound incomprehensible? If you can solve some problems we come up with but not all, then do you get an *A* for the course? And can you still get an *A* even if you can't solve any of the problems? Does the last sentence imply that all Americans have the same dream or might some of them have different dreams?

Some uncertainty is tolerable in normal conversation. But when we need to formulate ideas precisely—as in mathematics and programming—the ambiguities inherent in everyday language can be a real problem. We can't hope to make an exact argument if we're not sure exactly what the statements mean. So before we start into mathematics, we need to investigate the problem of how to talk about mathematics.

To get around the ambiguity of English, mathematicians have devised a special mini-language for talking about logical relationships. This language mostly uses ordinary English words and phrases such as "or", "implies", and "for all". But

mathematicians endow these words with definitions more precise than those found in an ordinary dictionary. Without knowing these definitions, you might sometimes get the gist of statements in this language, but you would regularly get misled about what they really meant.

Surprisingly, in the midst of learning the language of mathematics, we'll come across the most important open problem in computer science—a problem whose solution could change the world.

1.1 Compound Propositions

In English, we can modify, combine, and relate propositions with words such as "not", "and", "or", "implies", and "if-then". For example, we can combine three propositions into one like this:

If all humans are mortal **and** all Greeks are human, **then** all Greeks are mortal.

For the next while, we won't be much concerned with the internals of propositions—whether they involve mathematics or Greek mortality—but rather with how propositions are combined and related. So we'll frequently use variables such as P and Q in place of specific propositions such as "All humans are mortal" and "$2 + 3 = 5$". The understanding is that these variables, like propositions, can take on only the values **T** (true) and **F** (false). Such true/false variables are sometimes called *Boolean variables* after their inventor, George—you guessed it—Boole.

1.1.1 NOT, AND, and OR

We can precisely define these special words using *truth tables*. For example, if P denotes an arbitrary proposition, then the truth of the proposition "NOT(P)" is defined by the following truth table:

P	NOT(P)
T	**F**
F	**T**

The first row of the table indicates that when proposition P is true, the proposition "NOT(P)" is false. The second line indicates that when P is false, "NOT(P)" is true. This is probably what you would expect.

In general, a truth table indicates the true/false value of a proposition for each possible setting of the variables. For example, the truth table for the proposition

"P AND Q" has four lines, since the two variables can be set in four different ways:

P	Q	P AND Q
T	**T**	**T**
T	**F**	**F**
F	**T**	**F**
F	**F**	**F**

According to this table, the proposition "P AND Q" is true only when P and Q are both true. This is probably the way you think about the word "and."

There is a subtlety in the truth table for "P OR Q":

P	Q	P OR Q
T	**T**	**T**
T	**F**	**T**
F	**T**	**T**
F	**F**	**F**

The third row of this table says that "P OR Q" is true even if *both* P and Q are true. This isn't always the intended meaning of "or" in everyday speech, but this is the standard definition in mathematical writing. So if a mathematician says, "You may have cake, or you may have ice cream," he means that you *could* have both.

If you want to exclude the possibility of both having and eating, you should use "exclusive-or" (XOR):

P	Q	P XOR Q
T	**T**	**F**
T	**F**	**T**
F	**T**	**T**
F	**F**	**F**

1.1.2 IMPLIES

The least intuitive connecting word is "implies." Here is its truth table, with the lines labeled so we can refer to them later.

P	Q	P IMPLIES Q	
T	**T**	**T**	(tt)
T	**F**	**F**	(tf)
F	**T**	**T**	(ft)
F	**F**	**T**	(ff)

Let's experiment with this definition. For example, is the following proposition true or false?

"If the Riemann Hypothesis is true, then $x^2 \geq 0$ for every real number x."

The Riemann Hypothesis is a famous unresolved conjecture in mathematics —no one knows if it is true or false. But that doesn't prevent you from answering the question! This proposition has the form P IMPLIES Q where the *hypothesis*, P, is "the Riemann Hypothesis is true" and the *conclusion*, Q, is "$x^2 \geq 0$ for every real number x". Since the conclusion is definitely true, we're on either line (tt) or line (ft) of the truth table. Either way, the proposition as a while is *true*!

One of our original examples demonstrates an even stranger side of implications.

"If pigs can fly, then you can understand the Chebyshev bound."

Don't take this as an insult; we just need to figure out whether this proposition is true or false. Curiously, the answer has *nothing* to do with whether or not you can understand the Chebyshev bound. Pigs cannot fly, so we're on either line (ft) or line (ff) of the truth table. In both cases, the proposition is *true*!

In contrast, here's an example of a false implication:

"If the moon shines white, then the moon is made of white cheddar."

Yes, the moon shines white. But, no, the moon is not made of white cheddar cheese. So we're on line (tf) of the truth table, and the proposition is false.

The truth table for implications can be summarized in words as follows:

An implication is true exactly when the if-part is false or the then-part is true.

This sentence is worth remembering; a large fraction of all mathematical statements are of the if-then form!

1.1.3 IFF

Mathematicians commonly join propositions in one additional way that doesn't arise in ordinary speech. The proposition "P if and only if Q" asserts that P and Q are logically equivalent; that is, either both are true or both are false.

P	Q	P IFF Q
T	**T**	**T**
T	**F**	**F**
F	**T**	**F**
F	**F**	**T**

For example, the following if-and-only-if statement is true for every real number x:

$$x^2 - 4 \geq 0 \quad \text{iff} \quad |x| \geq 2$$

For some values of x, *both* inequalities are true. For other values of x, *neither* inequality is true . In every case, however, the proposition as a whole is true.

1.1.4 Notation

Mathematicians have devised symbols to represent words like "AND" and "NOT". The most commonly-used symbols are summarized in the table below.

English	**Symbolic Notation**
NOT(P)	$\neg P$ (alternatively, $\overline{P}$)
P AND Q	$P \wedge Q$
P OR Q	$P \vee Q$
P IMPLIES Q	$P \longrightarrow Q$
if P then Q	$P \longrightarrow Q$
P IFF Q	$P \longleftrightarrow Q$

For example, using this notation, "If P AND NOT(Q), then R" would be written:

$$(P \wedge \overline{Q}) \longrightarrow R$$

This symbolic language is helpful for writing complicated logical expressions compactly. But words such as "OR" and "IMPLIES" generally serve just as well as the symbols $\vee$ and $\longrightarrow$, and their meaning is easy to remember. We will use the prior notation for the most part in this text, but you can feel free to use whichever convention is easiest for you.

1.1.5 Logically Equivalent Implications

Do these two sentences say the same thing?

If I am hungry, then I am grumpy.
If I am not grumpy, then I am not hungry.

We can settle the issue by recasting both sentences in terms of propositional logic. Let P be the proposition "I am hungry", and let Q be "I am grumpy". The first sentence says "P IMPLIES Q" and the second says "NOT(Q) IMPLIES NOT(P)". We can compare these two statements in a truth table:

P	Q	P IMPLIES Q	NOT(Q) IMPLIES NOT(P)
T	**T**	**T**	**T**
T	**F**	**F**	**F**
F	**T**	**T**	**T**
F	**F**	**T**	**T**

Sure enough, the columns of truth values under these two statements are the same, which precisely means they are equivalent. In general, "NOT(Q) IMPLIES NOT(P)"

is called the *contrapositive* of the implication "P IMPLIES Q." And, as we've just shown, the two are just different ways of saying the same thing.

In contrast, the *converse* of "P IMPLIES Q" is the statement "Q IMPLIES P". In terms of our example, the converse is:

If I am grumpy, then I am hungry.

This sounds like a rather different contention, and a truth table confirms this suspicion:

P	Q	P IMPLIES Q	Q IMPLIES P
T	**T**	**T**	**T**
T	**F**	**F**	**T**
F	**T**	**T**	**F**
F	**F**	**T**	**T**

Thus, an implication *is* logically equivalent to its contrapositive but is *not* equivalent to its converse.

One final relationship: an implication and its converse together are equivalent to an iff statement. For example,

If I am grumpy, then I am hungry, AND
if I am hungry, then I am grumpy.

are equivalent to the single statement:

I am grumpy IFF I am hungry.

Once again, we can verify this with a truth table:

P	Q	(P IMPLIES Q)	(Q IMPLIES P)	(P IMPLIES Q) AND (Q IMPLIES P)	P IFF Q
T	**T**	**T**	**T**	**T**	**T**
T	**F**	**F**	**T**	**F**	**F**
F	**T**	**T**	**F**	**F**	**F**
F	**F**	**T**	**T**	**T**	**T**

1.2 Propositional Logic in Computer Programs

Propositions and logical connectives arise all the time in computer programs. For example, consider the following snippet, which could be either C, C++, or Java:

```
if ( x > 0 || (x <= 0 && y > 100) )
    ⋮
```

(further instructions)

The symbol `||` denotes "OR", and the symbol `&&` denotes "AND". The *further instructions* are carried out only if the proposition following the word `if` is true. On closer inspection, this big expression is built from two simpler propositions. Let A be the proposition that `x > 0`, and let B be the proposition that `y > 100`. Then we can rewrite the condition "A OR (NOT(A) AND B)". A truth table reveals that this complicated expression is logically equivalent to "A OR B".

A	B	A OR (NOT(A) AND B)	A OR B
T	**T**	**T**	**T**
T	**F**	**T**	**T**
F	**T**	**T**	**T**
F	**F**	**F**	**F**

This means that we can simplify the code snippet without changing the program's behavior:

```
if ( x > 0 || y > 100 )
    ⋮
```
(further instructions)

Rewriting a logical expression involving many variables in the simplest form is both difficult and important. Simplifying expressions in software can increase the speed of your program. Chip designers face a similar challenge—instead of minimizing `&&` and `||` symbols in a program, their job is to minimize the number of analogous physical devices on a chip. The payoff is potentially enormous: a chip with fewer devices is smaller, consumes less power, has a lower defect rate, and is cheaper to manufacture.

1.3 Predicates and Quantifiers

1.3.1 Propositions with Infinitely Many Cases

Most of the examples of propositions that we have considered thus far have been straightforward in the sense that it has been relatively easy to determine if they are true or false. At worse, there were only a few cases to check in a truth table. Unfortunately, not all propositions are so easy to check. That is because some propositions may involve a large or infinite number of possible cases. For example, consider the following proposition involving prime numbers. (A *prime* is an integer greater than 1 that is divisible only by itself and 1. For example, 2, 3, 5, 7, and 11

are primes, but 4, 6, and 9 are not. A number greater than 1 that is not prime is said to be *composite*.)

Proposition 1.3.1. *For every nonnegative integer, n, the value of $n^2 + n + 41$ is prime.*

It is not immediately clear whether this proposition is true or false. In such circumstances, it is tempting to try to determine its veracity by computing the value of[1]

$$p(n) ::= n^2 + n + 41. \tag{1.1}$$

for several values of n and then checking to see if they are prime. If any of the computed values is not prime, then we will know that the proposition is false. If all the computed values are indeed prime, then we might be tempted to conclude that the proposition is true.

We begin the checking by evaluating $p(0) = 41$, which is prime. $p(1) = 43$ is also prime. So is $p(2) = 47$, $p(3) = 53$, ..., and $p(20) = 461$, all of which are prime. Hmmm... It is starting to look like $p(n)$ is a prime for every nonnegative integer n. In fact, continued checking reveals that $p(n)$ is prime for all $n \leq 39$. The proposition certainly does seem to be true.

But $p(40) = 40^2 + 40 + 41 = 41 \cdot 41$, which is not prime. So it's *not* true that the expression is prime *for all* nonnegative integers, and thus the proposition is false!

Although surprising, this example is not as contrived or rare as you might suspect. As we will soon see, there are many examples of propositions that seem to be true when you check a few cases (or even many), but which turn out to be false. The key to remember is that you can't check a claim about an infinite set by checking a finite set of its elements, no matter how large the finite set.

Propositions that involve *all* numbers are so common that there is a special notation for them. For example, Proposition 1.3.1 can also be written as

$$\forall n \in \mathbb{N}.\ p(n) \text{ is prime.} \tag{1.2}$$

Here the symbol $\forall$ is read "for all". The symbol $\mathbb{N}$ stands for the set of *nonnegative integers*, namely, 0, 1, 2, 3, ...(ask your instructor for the complete list). The symbol "$\in$" is read as "is a member of," or "belongs to," or simply as "is in". The period after the $\mathbb{N}$ is just a separator between phrases.

Here is another example of a proposition that, at first, seems to be true but which turns out to be false.

[1] The symbol ::= means "equal by definition." It's always ok to simply write "=" instead of ::=, but reminding the reader that an equality holds by definition can be helpful.

Proposition 1.3.2. *$a^4 + b^4 + c^4 = d^4$ has no solution when a, b, c, d are positive integers.*

Euler (pronounced "oiler") conjectured this proposition to be true in 1769. It was checked by humans and then by computers for many values of a, b, c, and d over the next two centuries. Ultimately the proposition was proven false in 1987 by Noam Elkies. The solution he found was $a = 95800, b = 217519, c = 414560, d = 422481$. No wonder it took 218 years to show the proposition is false!

In logical notation, Proposition 1.3.2 could be written,

$$\forall a \in \mathbb{Z}^+ \, \forall b \in \mathbb{Z}^+ \, \forall c \in \mathbb{Z}^+ \, \forall d \in \mathbb{Z}^+ . \, a^4 + b^4 + c^4 \neq d^4.$$

Here, $\mathbb{Z}^+$ is a symbol for the positive integers. Strings of $\forall$'s are usually abbreviated for easier reading, as follows:

$$\forall a, b, c, d \in \mathbb{Z}^+ . \, a^4 + b^4 + c^4 \neq d^4.$$

The following proposition is even nastier.

Proposition 1.3.3. *$313(x^3 + y^3) = z^3$ has no solution when $x, y, z \in \mathbb{Z}^+$.*

This proposition is also false, but the smallest counterexample values for x, y, and z have more than 1000 digits! Even the world's largest computers would not be able to get that far with brute force. Of course, you may be wondering why anyone would care whether or not there is a solution to $313(x^3 + y^3) = z^3$ where x, y, and z are positive integers. It turns out that finding solutions to such equations is important in the field of elliptic curves, which turns out to be important to the study of factoring large integers, which turns out (as we will see in Chapter 4) to be important in cracking commonly-used cryptosystems, which is why mathematicians went to the effort to find the solution with thousands of digits.

Of course, not all propositions that have infinitely many cases to check turn out to be false. The following proposition (known as the "Four-Color Theorem") turns out to be true.

Proposition 1.3.4. *Every map can be colored with 4 colors so that adjacent[2] regions have different colors.*

The proof of this proposition is difficult and took over a century to perfect. Along the way, many incorrect proofs were proposed, including one that stood for 10 years

[2]Two regions are adjacent only when they share a boundary segment of positive length. They are not considered to be adjacent if their boundaries meet only at a few points.

in the late 19th century before the mistake was found. An extremely laborious proof was finally found in 1976 by mathematicians Appel and Haken, who used a complex computer program to categorize the four-colorable maps; the program left a few thousand maps uncategorized, and these were checked by hand by Haken and his assistants—including his 15-year-old daughter. There was a lot of debate about whether this was a legitimate proof: the proof was too big to be checked without a computer, and no one could guarantee that the computer calculated correctly, nor did anyone have the energy to recheck the four-colorings of the thousands of maps that were done by hand. Within the past decade, a mostly intelligible proof of the Four-Color Theorem was found, though a computer is still needed to check the colorability of several hundred special maps.[3]

In some cases, we do not know whether or not a proposition is true. For example, the following simple proposition (known as Goldbach's Conjecture) has been heavily studied since 1742 but we still do not know if it is true. Of course, it has been checked by computer for many values of n, but as we have seen, that is not sufficient to conclude that it is true.

Proposition 1.3.5 (Goldbach)**.** *Every even integer n greater than 2 is the sum of two primes.*

While the preceding propositions are important in mathematics, computer scientists are often interested in propositions concerning the "correctness" of programs and systems, to determine whether a program or system does what it's supposed to do. Programs are notoriously buggy, and there's a growing community of researchers and practitioners trying to find ways to prove program correctness. These efforts have been successful enough in the case of CPU chips that they are now routinely used by leading chip manufacturers to prove chip correctness and avoid mistakes like the notorious Intel division bug in the 1990's.

Developing mathematical methods to verify programs and systems remains an active research area. We'll consider some of these methods later in the text.

1.3.2 Predicates

A *predicate* is a proposition whose truth depends on the value of one or more variables. Most of the propositions above were defined in terms of predicates. For example,

"n is a perfect square"

[3] See http://www.math.gatech.edu/~thomas/FC/fourcolor.html
The story of the Four-Color Proof is told in a well-reviewed popular (non-technical) book: "Four Colors Suffice. How the Map Problem was Solved." *Robin Wilson*. Princeton Univ. Press, 2003, 276pp. ISBN 0-691-11533-8.

is a predicate whose truth depends on the value of n. The predicate is true for $n = 4$ since four is a perfect square, but false for $n = 5$ since five is not a perfect square.

Like other propositions, predicates are often named with a letter. Furthermore, a function-like notation is used to denote a predicate supplied with specific variable values. For example, we might name our earlier predicate P:

$$P(n) ::= \text{“}n \text{ is a perfect square”}$$

Now $P(4)$ is true, and $P(5)$ is false.

This notation for predicates is confusingly similar to ordinary function notation. If P is a predicate, then $P(n)$ is either *true* or *false*, depending on the value of n. On the other hand, if p is an ordinary function, like n^2+n, then $p(n)$ is a *numerical quantity*. **Don't confuse these two!**

1.3.3 Quantifiers

There are a couple of assertions commonly made about a predicate: that it is *sometimes* true and that it is *always* true. For example, the predicate

$$\text{“}x^2 \geq 0\text{”}$$

is always true when x is a real number. On the other hand, the predicate

$$\text{“}5x^2 - 7 = 0\text{”}$$

is only sometimes true; specifically, when $x = \pm\sqrt{7/5}$.

There are several ways to express the notions of "always true" and "sometimes true" in English. The table below gives some general formats on the left and specific examples using those formats on the right. You can expect to see such phrases hundreds of times in mathematical writing!

Always True

For all n, $P(n)$ is true.	For all $x \in \mathbb{R}$, $x^2 \geq 0$.
$P(n)$ is true for every n.	$x^2 \geq 0$ for every $x \in \mathbb{R}$.

Sometimes True

There exists an n such that $P(n)$ is true.	There exists an $x \in \mathbb{R}$ such that $5x^2 - 7 = 0$.
$P(n)$ is true for some n.	$5x^2 - 7 = 0$ for some $x \in \mathbb{R}$.
$P(n)$ is true for at least one n.	$5x^2 - 7 = 0$ for at least one $x \in \mathbb{R}$.

All these sentences quantify how often the predicate is true. Specifically, an assertion that a predicate is always true, is called a *universally quantified* statement.

An assertion that a predicate is sometimes true, is called an *existentially quantified* statement.

Sometimes English sentences are unclear about quantification:

"If you can solve any problem we come up with, then you get an A for the course."

The phrase "you can solve any problem we can come up with" could reasonably be interpreted as either a universal or existential statement. It might mean:

"You can solve *every* problem we come up with,"

or maybe

"You can solve *at least one* problem we come up with."

In the preceding example, the quantified phrase appears inside a larger if-then statement. This is quite normal; quantified statements are themselves propositions and can be combined with AND, OR, IMPLIES, etc., just like any other proposition.

1.3.4 More Notation

There are symbols to represent universal and existential quantification, just as there are symbols for "AND" ($\wedge$), "IMPLIES" ($\longrightarrow$), and so forth. In particular, to say that a predicate, $P(x)$, is true for all values of x in some set, D, we write:

$$\forall x \in D.\ P(x) \tag{1.3}$$

The *universal quantifier* symbol $\forall$ is read "for all," so this whole expression (1.3) is read "For all x in D, $P(x)$ is true." Remember that upside-down "A" stands for "**A**ll."

To say that a predicate $P(x)$ is true for at least one value of x in D, we write:

$$\exists x \in D.\ P(x) \tag{1.4}$$

The *existential quantifier* symbol $\exists$, is read "there exists." So expression (1.4) is read, "There exists an x in D such that $P(x)$ is true." Remember that backward "E" stands for "**E**xists."

The symbols $\forall$ and $\exists$ are always followed by a variable—typically with an indication of the set the variable ranges over—and then a predicate, as in the two examples above.

As an example, let Probs be the set of problems we come up with, Solves(x) be the predicate "You can solve problem x", and G be the proposition, "You get an A for the course." Then the two different interpretations of

> "If you can solve any problem we come up with, then you get an *A* for the course."

can be written as follows:

$$(\forall x \in \text{Probs. Solves}(x)) \text{ IMPLIES } G,$$

or maybe

$$(\exists x \in \text{Probs. Solves}(x)) \text{ IMPLIES } G.$$

1.3.5 Mixing Quantifiers

Many mathematical statements involve several quantifiers. For example, *Goldbach's Conjecture* states:

> "Every even integer greater than 2 is the sum of two primes."

Let's write this more verbosely to make the use of quantification clearer:

> For every even integer n greater than 2, there exist primes p and q such that $n = p + q$.

Let Evens be the set of even integers greater than 2, and let Primes be the set of primes. Then we can write Goldbach's Conjecture in logic notation as follows:

$$\underbrace{\forall n \in \text{Evens.}}_{\substack{\text{for every even} \\ \text{integer } n > 2}} \underbrace{\exists p \in \text{Primes } \exists q \in \text{Primes.}}_{\substack{\text{there exist primes} \\ p \text{ and } q \text{ such that}}} n = p + q.$$

The proposition can also be written more simply as

$$\forall n \in \text{Evens}. \exists p, q \in \text{Primes}. \ p + q = n.$$

1.3.6 Order of Quantifiers

Swapping the order of different kinds of quantifiers (existential or universal) usually changes the meaning of a proposition. For example, let's return to one of our initial, confusing statements:

> "Every American has a dream."

This sentence is ambiguous because the order of quantifiers is unclear. Let A be the set of Americans, let D be the set of dreams, and define the predicate $H(a, d)$ to be "American a has dream d." Now the sentence could mean that there is a single dream that every American shares:

$$\exists d \in D. \forall a \in A. \ H(a, d)$$

For example, it might be that every American shares the dream of owning their own home.

Or it could mean that every American has a personal dream:

$$\forall a \in A.\, \exists\, d \in D.\; H(a, d)$$

For example, some Americans may dream of a peaceful retirement, while others dream of continuing practicing their profession as long as they live, and still others may dream of being so rich they needn't think at all about work.

Swapping quantifiers in Goldbach's Conjecture creates a patently false statement; namely that every even number ≥ 2 is the sum of *the same* two primes:

$$\underbrace{\exists\, p, q \in \text{Primes}.}_{\substack{\text{there exist primes} \\ p \text{ and } q \text{ such that}}} \underbrace{\forall n \in \text{Evens}.}_{\substack{\text{for every even} \\ \text{integer } n > 2}} \; n = p + q.$$

1.3.7 Variables Over One Domain

When all the variables in a formula are understood to take values from the same nonempty set, D, it's conventional to omit mention of D. For example, instead of $\forall x \in D\ \exists y \in D.\ Q(x, y)$ we'd write $\forall x \exists y.\ Q(x, y)$. The unnamed nonempty set that x and y range over is called the *domain of discourse*, or just plain *domain*, of the formula.

It's easy to arrange for all the variables to range over one domain. For example, Goldbach's Conjecture could be expressed with all variables ranging over the domain $\mathbb{N}$ as

$$\forall n.\, (n \in \text{Evens}) \ \text{IMPLIES}\ \ (\exists p\, \exists q.\ p \in \text{Primes AND } q \in \text{Primes AND } n = p + q).$$

1.3.8 Negating Quantifiers

There is a simple relationship between the two kinds of quantifiers. The following two sentences mean the same thing:

> It is not the case that everyone likes to snowboard.
>
> There exists someone who does not like to snowboard.

In terms of logic notation, this follows from a general property of predicate formulas:

$$\text{NOT}\,(\forall x.\ P(x)) \quad \text{is equivalent to} \quad \exists x.\ \text{NOT}(P(x)).$$

Similarly, these sentences mean the same thing:

There does not exist anyone who likes skiing over magma.

Everyone dislikes skiing over magma.

We can express the equivalence in logic notation this way:

$$\text{NOT}\,(\exists x.\, P(x)) \;\text{ IFF }\; \forall x.\, \text{NOT}(P(x)). \tag{1.5}$$

The general principle is that *moving a "not" across a quantifier changes the kind of quantifier.*

1.4 Validity

A propositional formula is called *valid* when it evaluates to **T** no matter what truth values are assigned to the individual propositional variables. For example, the propositional version of the Distributive Law is that P AND $(Q$ OR $R)$ is equivalent to $(P$ AND $Q)$ OR $(P$ AND $R)$. This is the same as saying that

$$[P \text{ AND } (Q \text{ OR } R)] \text{ IFF } [(P \text{ AND } Q) \text{ OR } (P \text{ AND } R)] \tag{1.6}$$

is valid. This can be verified by checking the truth table for P AND $(Q$ OR $R)$ and $(P$ AND $Q)$ OR $(P$ AND $R)$:

P	Q	R	P AND $(Q$ OR $R)$	$(P$ AND $Q)$ OR $(P$ AND $R)$
T	**T**	**T**	**T**	**T**
T	**T**	**F**	**T**	**T**
T	**F**	**T**	**T**	**T**
T	**F**	**F**	**F**	**F**
F	**T**	**T**	**F**	**F**
F	**T**	**F**	**F**	**F**
F	**F**	**T**	**F**	**F**
F	**F**	**F**	**F**	**F**

The same idea extends to predicate formulas, but to be valid, a formula now must evaluate to true no matter what values its variables may take over any unspecified domain, and no matter what interpretation a predicate variable may be given. For example, we already observed that the rule for negating a quantifier is captured by the valid assertion (1.5).

Another useful example of a valid assertion is

$$\exists x \forall y.\, P(x, y) \text{ IMPLIES } \forall y \exists x.\, P(x, y). \tag{1.7}$$

Here's an explanation why this is valid:

Let D be the domain for the variables and P_0 be some binary predicate[4] on D. We need to show that if

$$\exists x \in D\ \forall y \in D.\ P_0(x, y) \tag{1.8}$$

holds under this interpretation, then so does

$$\forall y \in D\ \exists x \in D.\ P_0(x, y). \tag{1.9}$$

So suppose (1.8) is true. Then by definition of $\exists$, this means that some element $d_0 \in D$ has the property that

$$\forall y \in D.\ P_0(d_0, y).$$

By definition of $\forall$, this means that

$$P_0(d_0, d)$$

is true for all $d \in D$. So given any $d \in D$, there is an element in D, namely, d_0, such that $P_0(d_0, d)$ is true. But that's exactly what (1.9) means, so we've proved that (1.9) holds under this interpretation, as required.

We hope this is helpful as an explanation, although purists would not really want to call it a "proof." The problem is that with something as basic as (1.7), it's hard to see what more elementary axioms are ok to use in proving it. What the explanation above did was translate the logical formula (1.7) into English and then appeal to the meaning, in English, of "for all" and "there exists" as justification.

In contrast to (1.7), the formula

$$\forall y \exists x.\ P(x, y) \text{ IMPLIES } \exists x \forall y.\ P(x, y). \tag{1.10}$$

is *not* valid. We can prove this by describing an interpretation where the hypothesis, $\forall y \exists x.\ P(x, y)$, is true but the conclusion, $\exists x \forall y.\ P(x, y)$, is not true. For example, let the domain be the integers and $P(x, y)$ mean $x > y$. Then the hypothesis would be true because, given a value, n, for y we could, for example, choose the value of x to be $n + 1$. But under this interpretation the conclusion asserts that there is an integer that is bigger than all integers, which is certainly false. An interpretation like this which falsifies an assertion is called a *counter model* to the assertion.

[4] That is, a predicate that depends on two variables.

1.5 Satisfiability

A proposition is **satisfiable** if some setting of the variables makes the proposition true. For example, P AND $\overline{Q}$ is satisfiable because the expression is true if P is true or Q is false. On the other hand, P AND $\overline{P}$ is not satisfiable because the expression as a whole is false for both settings of P. But determining whether or not a more complicated proposition is satisfiable is not so easy. How about this one?

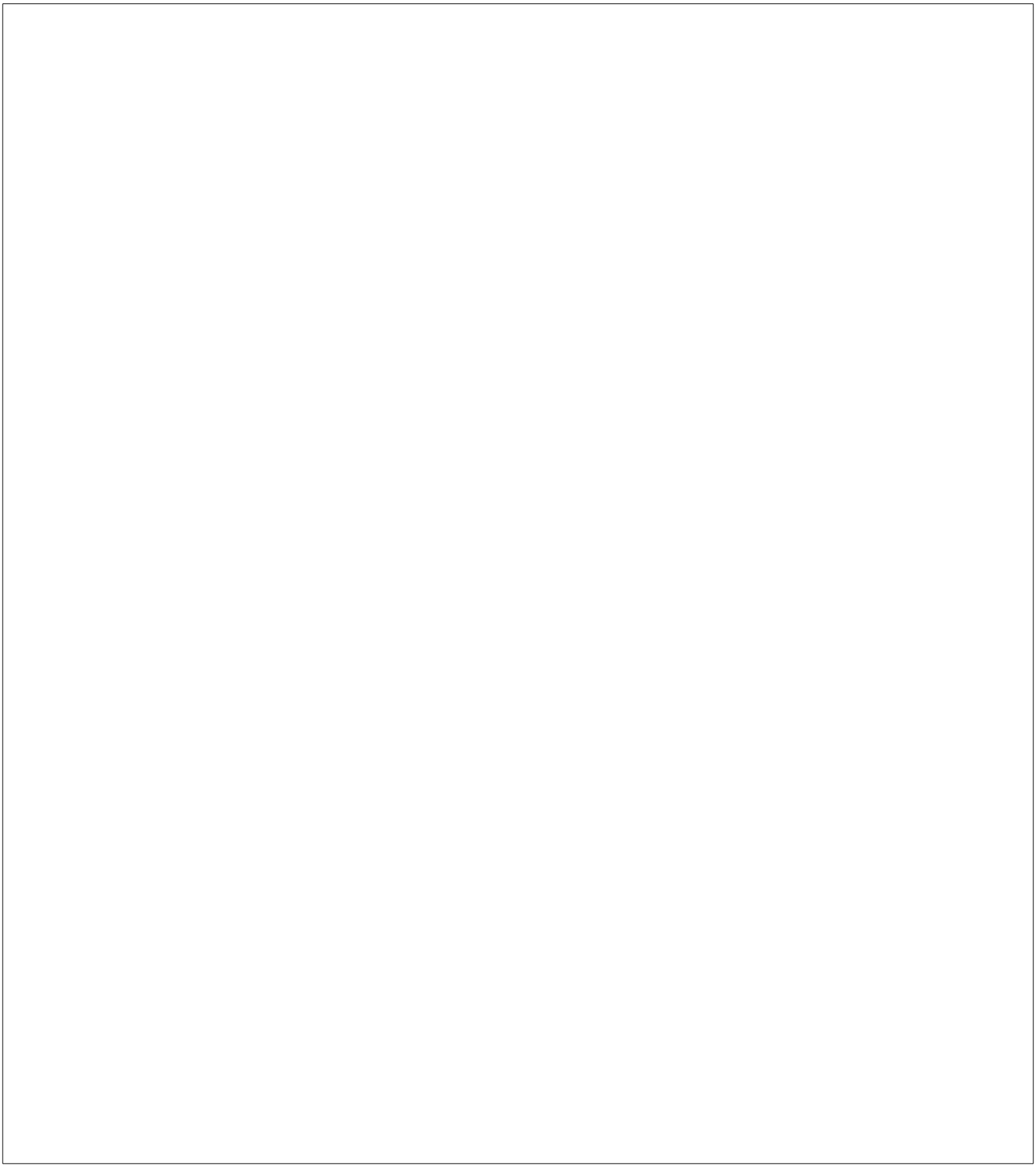
$$(P \text{ OR } Q \text{ OR } R) \text{ AND } (\overline{P} \text{ OR } \overline{Q}) \text{ AND } (\overline{P} \text{ OR } \overline{R}) \text{ AND } (\overline{R} \text{ OR } \overline{Q})$$

The general problem of deciding whether a proposition is satisfiable is called *SAT*. One approach to SAT is to construct a truth table and check whether or not a **T** ever appears. But this approach is not very efficient; a proposition with n variables has a truth table with 2^n lines, so the effort required to decide about a proposition grows exponentially with the number of variables. For a proposition with just 30 variables, that's already over a billion lines to check!

Is there a more *efficient* solution to SAT? In particular, is there some, presumably very ingenious, procedure that determines in a number of steps that grows *polynomially*—like n^2 or n^{14}—instead of exponentially, whether any given proposition is satisfiable or not? No one knows. And an awful lot hangs on the answer. An efficient solution to SAT would immediately imply efficient solutions to many, many other important problems involving packing, scheduling, routing, and circuit verification, among other things. This would be wonderful, but there would also be worldwide chaos. Decrypting coded messages would also become an easy task (for most codes). Online financial transactions would be insecure and secret communications could be read by everyone.

Recently there has been exciting progress on *sat-solvers* for practical applications like digital circuit verification. These programs find satisfying assignments with amazing efficiency even for formulas with millions of variables. Unfortunately, it's hard to predict which kind of formulas are amenable to sat-solver methods, and for formulas that are NOT satisfiable, sat-solvers generally take exponential time to verify that.

So no one has a good idea how to solve SAT in polynomial time, or how to prove that it can't be done—researchers are completely stuck. The problem of determining whether or not SAT has a polynomial time solution is known as the "**P** vs. **NP**" problem. It is the outstanding unanswered question in theoretical computer science. It is also one of the seven Millenium Problems: the Clay Institute will award you $1,000,000 if you solve the **P** vs. **NP** problem.



2 Patterns of Proof

2.1 The Axiomatic Method

The standard procedure for establishing truth in mathematics was invented by Euclid, a mathematician working in Alexandria, Egypt around 300 BC. His idea was to begin with five *assumptions* about geometry, which seemed undeniable based on direct experience. For example, one of the assumptions was "There is a straight line segment between every pair of points." Propositions like these that are simply accepted as true are called *axioms*.

Starting from these axioms, Euclid established the truth of many additional propositions by providing "proofs". A *proof* is a sequence of logical deductions from axioms and previously-proved statements that concludes with the proposition in question. You probably wrote many proofs in high school geometry class, and you'll see a lot more in this course.

There are several common terms for a proposition that has been proved. The different terms hint at the role of the proposition within a larger body of work.

- Important propositions are called *theorems*.
- A *lemma* is a preliminary proposition useful for proving later propositions.
- A *corollary* is a proposition that follows in just a few logical steps from a lemma or a theorem.

The definitions are not precise. In fact, sometimes a good lemma turns out to be far more important than the theorem it was originally used to prove.

Euclid's axiom-and-proof approach, now called the *axiomatic method*, is the foundation for mathematics today. In fact, just a handful of axioms, collectively called Zermelo-Frankel Set Theory with Choice (*ZFC*), together with a few logical deduction rules, appear to be sufficient to derive essentially all of mathematics.

2.1.1 Our Axioms

The ZFC axioms are important in studying and justifying the foundations of mathematics, but for practical purposes, they are much too primitive. Proving theorems in ZFC is a little like writing programs in byte code instead of a full-fledged programming language—by one reckoning, a formal proof in ZFC that $2 + 2 = 4$ requires more than 20,000 steps! So instead of starting with ZFC, we're going to

take a *huge* set of axioms as our foundation: we'll accept all familiar facts from high school math!

This will give us a quick launch, but you may find this imprecise specification of the axioms troubling at times. For example, in the midst of a proof, you may find yourself wondering, "Must I prove this little fact or can I take it as an axiom?" Feel free to ask for guidance, but really there is no absolute answer. Just be up front about what you're assuming, and don't try to evade homework and exam problems by declaring everything an axiom!

2.1.2 Logical Deductions

Logical deductions or *inference rules* are used to prove new propositions using previously proved ones.

A fundamental inference rule is *modus ponens*. This rule says that a proof of P together with a proof that P IMPLIES Q is a proof of Q.

Inference rules are sometimes written in a funny notation. For example, *modus ponens* is written:

Rule 2.1.1.

$$\frac{P, \quad P \text{ IMPLIES } Q}{Q}$$

When the statements above the line, called the *antecedents*, are proved, then we can consider the statement below the line, called the *conclusion* or *consequent*, to also be proved.

A key requirement of an inference rule is that it must be *sound*: any assignment of truth values that makes all the antecedents true must also make the consequent true. So if we start off with true axioms and apply sound inference rules, everything we prove will also be true.

You can see why modus ponens is a sound inference rule by checking the truth table of P IMPLIES Q. There is only one case where P and P IMPLIES Q are both true, and in that case Q is also true.

P	Q	$P \longrightarrow Q$
F	**F**	**T**
F	**T**	**T**
T	**F**	**F**
T	**T**	**T**

There are many other natural, sound inference rules, for example:

Rule 2.1.2.

$$\frac{P \text{ IMPLIES } Q, \quad Q \text{ IMPLIES } R}{P \text{ IMPLIES } R}$$

Rule 2.1.3.

$$\frac{P \text{ IMPLIES } Q, \quad \text{NOT}(Q)}{\text{NOT}(P)}$$

Rule 2.1.4.

$$\frac{\text{NOT}(P) \text{ IMPLIES NOT}(Q)}{Q \text{ IMPLIES } P}$$

On the other hand,

Non-Rule.

$$\frac{\text{NOT}(P) \text{ IMPLIES NOT}(Q)}{P \text{ IMPLIES } Q}$$

is *not* sound: if P is assigned **T** and Q is assigned **F**, then the antecedent is true and the consequent is not.

Note that a propositional inference rule is sound precisely when the conjunction (AND) of all its antecedents implies its consequent.

As with axioms, we will not be too formal about the set of legal inference rules. Each step in a proof should be clear and "logical"; in particular, you should state what previously proved facts are used to derive each new conclusion.

2.1.3 Proof Templates

In principle, a proof can be *any* sequence of logical deductions from axioms and previously proved statements that concludes with the proposition in question. This freedom in constructing a proof can seem overwhelming at first. How do you even *start* a proof?

Here's the good news: many proofs follow one of a handful of standard templates. Each proof has it own details, of course, but these templates at least provide you with an outline to fill in. In the remainder of this chapter, we'll go through several of these standard patterns, pointing out the basic idea and common pitfalls and giving some examples. Many of these templates fit together; one may give you a top-level outline while others help you at the next level of detail. And we'll show you other, more sophisticated proof techniques in Chapter 3.

The recipes that follow are very specific at times, telling you exactly which words to write down on your piece of paper. You're certainly free to say things your own way instead; we're just giving you something you *could* say so that you're never at a complete loss.

2.2 Proof by Cases

Breaking a complicated proof into cases and proving each case separately is a useful and common proof strategy. In fact, we have already implicitly used this strategy when we used truth tables to show that certain propositions were true or valid. For example, in section 1.1.5, we showed that an implication P IMPLIES Q is equivalent to its contrapositive NOT(Q) IMPLIES NOT(P) by considering all 4 possible assignments of **T** or **F** to P and Q. In each of the four cases, we showed that P IMPLIES Q is true if and only if NOT(Q) IMPLIES NOT(P) is true. For example, if $P = \mathbf{T}$ and $Q = \mathbf{F}$, then both P IMPLIES Q and NOT(Q) IMPLIES NOT(P) are false, thereby establishing that (P IMPLIES Q)IFF(NOT(Q) IMPLIES NOT(P)) is true for this case. If a proposition is true in every possible case, then it is true.

Proof by cases works in much more general environments than propositions involving Boolean variables. In what follows, we will use this approach to prove a simple fact about acquaintances. As background, we will assume that for any pair of people, either they have met or not. If every pair of people in a group has met, we'll call the group a *club*. If every pair of people in a group has not met, we'll call it a group of *strangers*.

Theorem. *Every collection of 6 people includes a club of 3 people or a group of 3 strangers.*

Proof. The proof is by case analysis[1]. Let x denote one of the six people. There are two cases:

1. Among the other 5 people besides x, at least 3 have met x.
2. Among the other 5 people, at least 3 have not met x.

Now we have to be sure that at least one of these two cases must hold,[2] but that's easy: we've split the 5 people into two groups, those who have shaken hands with x and those who have not, so one of the groups must have at least half the people.

Case 1: Suppose that at least 3 people have met x.

This case splits into two subcases:

[1] Describing your approach at the outset helps orient the reader. Try to remember to always do this.

[2] Part of a case analysis argument is showing that you've covered all the cases. Often this is obvious, because the two cases are of the form "P" and "not P". However, the situation above is not stated quite so simply.

Case 1.1: Among the people who have met x, none have met each other. Then the people who have met x are a group of at least 3 strangers. So the Theorem holds in this subcase.

Case 1.2: Among the people who have met x, some pair have met each other. Then that pair, together with x, form a club of 3 people. So the Theorem holds in this subcase.

This implies that the Theorem holds in Case 1.

Case 2: Suppose that at least 3 people have not met x.

This case also splits into two subcases:

Case 2.1: Among the people who have not met x, every pair has met each other. Then the people who have not met x are a club of at least 3 people. So the Theorem holds in this subcase.

Case 2.2: Among the people who have not met x, some pair have not met each other. Then that pair, together with x, form a group of at least 3 strangers. So the Theorem holds in this subcase.

This implies that the Theorem also holds in Case 2, and therefore holds in all cases. ■

2.3 Proving an Implication

Propositions of the form "If P, then Q" are called *implications*. This implication is often rephrased as "P IMPLIES Q" or "$P \longrightarrow Q$".

Here are some examples of implications:

- (Quadratic Formula) If $ax^2 + bx + c = 0$ and $a \neq 0$, then

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

- (Goldbach's Conjecture) If n is an even integer greater than 2, then n is a sum of two primes.
- If $0 \leq x \leq 2$, then $-x^3 + 4x + 1 > 0$.

There are a couple of standard methods for proving an implication.

2.3.1 Method #1: Assume P is true

When proving P IMPLIES Q, there are two cases to consider: P is true and P is false. The case when P is false is easy since, by definition, **F** IMPLIES Q is true no matter what Q is. This case is so easy that we usually just forget about it and start right off by assuming that P is true when proving an implication, since this is the only case that is interesting. Hence, in order to prove that P IMPLIES Q:

1. Write, "Assume P."
2. Show that Q logically follows.

For example, we will use this method to prove

Theorem 2.3.1. *If* $0 \le x \le 2$*, then* $-x^3 + 4x + 1 > 0$.

Before we write a proof of this theorem, we have to do some scratchwork to figure out why it is true.

The inequality certainly holds for $x = 0$; then the left side is equal to 1 and $1 > 0$. As x grows, the $4x$ term (which is positive) initially seems to have greater magnitude than $-x^3$ (which is negative). For example, when $x = 1$, we have $4x = 4$, but $-x^3 = -1$. In fact, it looks like $-x^3$ doesn't begin to dominate $4x$ until $x > 2$. So it seems the $-x^3 + 4x$ part should be nonnegative for all x between 0 and 2, which would imply that $-x^3 + 4x + 1$ is positive.

So far, so good. But we still have to replace all those "seems like" phrases with solid, logical arguments. We can get a better handle on the critical $-x^3 + 4x$ part by factoring it, which is not too hard:

$$-x^3 + 4x = x(2 - x)(2 + x)$$

Aha! For x between 0 and 2, all of the terms on the right side are nonnegative. And a product of nonnegative terms is also nonnegative. Let's organize this blizzard of observations into a clean proof.

Proof. Assume $0 \le x \le 2$. Then x, $2-x$, and $2+x$ are all nonnegative. Therefore, the product of these terms is also nonnegative. Adding 1 to this product gives a positive number, so:

$$x(2 - x)(2 + x) + 1 > 0$$

Multiplying out on the left side proves that

$$-x^3 + 4x + 1 > 0$$

as claimed. ■

There are a couple points here that apply to all proofs:

- You'll often need to do some scratchwork while you're trying to figure out the logical steps of a proof. Your scratchwork can be as disorganized as you like—full of dead-ends, strange diagrams, obscene words, whatever. But keep your scratchwork separate from your final proof, which should be clear and concise.
- Proofs typically begin with the word "Proof" and end with some sort of doohickey like □ or ■ or "q.e.d". The only purpose for these conventions is to clarify where proofs begin and end.

Potential Pitfall

For the purpose of proving an implication P IMPLIES Q, it's OK, and typical, to begin by assuming P. But when the proof is over, it's no longer OK to assume that P holds! For example, Theorem 2.3.1 has the form "if P, then Q" with P being "$0 \leq x \leq 2$" and Q being "$-x^3 + 4x + 1 > 0$," and its proof began by assuming that $0 \leq x \leq 2$. But of course this assumption does not always hold. Indeed, if you were going to prove another result using the variable x, it could be disastrous to have a step where you assume that $0 \leq x \leq 2$ just because you assumed it as part of the proof of Theorem 2.3.1.

2.3.2 Method #2: Prove the Contrapositive

We have already seen that an implication "P IMPLIES Q" is logically equivalent to its *contrapositive*

$$\text{NOT}(Q) \text{ IMPLIES } \text{NOT}(P).$$

Proving one is as good as proving the other, and proving the contrapositive is sometimes easier than proving the original statement. Hence, you can proceed as follows:

1. Write, "We prove the contrapositive:" and then state the contrapositive.
2. Proceed as in Method #1.

For example, we can use this approach to prove

Theorem 2.3.2. *If r is irrational, then $\sqrt{r}$ is also irrational.*

Recall that rational numbers are equal to a ratio of integers and irrational numbers are not. So we must show that if r is *not* a ratio of integers, then $\sqrt{r}$ is also *not* a ratio of integers. That's pretty convoluted! We can eliminate both *not*'s and make the proof straightforward by considering the contrapositive instead.

Proof. We prove the contrapositive: if $\sqrt{r}$ is rational, then r is rational.

Assume that $\sqrt{r}$ is rational. Then there exist integers a and b such that:

$$\sqrt{r} = \frac{a}{b}$$

Squaring both sides gives:

$$r = \frac{a^2}{b^2}$$

Since a^2 and b^2 are integers, r is also rational. ■

2.4 Proving an "If and Only If"

Many mathematical theorems assert that two statements are logically equivalent; that is, one holds if and only if the other does. Here is an example that has been known for several thousand years:

> Two triangles have the same side lengths if and only if two side lengths and the angle between those sides are the same in each triangle.

The phrase "if and only if" comes up so often that it is often abbreviated "iff".

2.4.1 Method #1: Prove Each Statement Implies the Other

The statement "P IFF Q" is equivalent to the two statements "P IMPLIES Q" and "Q IMPLIES P". So you can prove an "iff" by proving *two* implications:

1. Write, "We prove P implies Q and vice-versa."
2. Write, "First, we show P implies Q." Do this by one of the methods in Section 2.3.
3. Write, "Now, we show Q implies P." Again, do this by one of the methods in Section 2.3.

2.4.2 Method #2: Construct a Chain of IFFs

In order to prove that P is true iff Q is true:

1. Write, "We construct a chain of if-and-only-if implications."
2. Prove P is equivalent to a second statement which is equivalent to a third statement and so forth until you reach Q.

This method sometimes requires more ingenuity than the first, but the result can be a short, elegant proof, as we see in the following example.

Theorem 2.4.1. *The standard deviation of a sequence of values $x_1, \ldots, x_n$ is zero iff all the values are equal to the mean.*

Definition. The *standard deviation* of a sequence of values $x_1, x_2, \ldots, x_n$ is defined to be:

$$\sqrt{\frac{(x_1-\mu)^2+(x_2-\mu)^2+\cdots+(x_n-\mu)^2}{n}} \tag{2.1}$$

where μ is the *mean* of the values:

$$\mu ::= \frac{x_1+x_2+\cdots+x_n}{n}$$

As an example, Theorem 2.4.1 says that the standard deviation of test scores is zero if and only if everyone scored exactly the class average. (We will talk a lot more about means and standard deviations in Part IV of the book.)

Proof. We construct a chain of "iff" implications, starting with the statement that the standard deviation (2.1) is zero:

$$\sqrt{\frac{(x_1-\mu)^2+(x_2-\mu)^2+\cdots+(x_n-\mu)^2}{n}} = 0. \tag{2.2}$$

Since zero is the only number whose square root is zero, equation (2.2) holds iff

$$(x_1-\mu)^2+(x_2-\mu)^2+\cdots+(x_n-\mu)^2 = 0. \tag{2.3}$$

Squares of real numbers are always nonnegative, and so every term on the left hand side of equation (2.3) is nonnegative. This means that (2.3) holds iff

$$\text{Every term on the left hand side of (2.3) is zero.} \tag{2.4}$$

But a term $(x_i-\mu)^2$ is zero iff $x_i = \mu$, so (2.4) is true iff

$$\text{Every } x_i \text{ equals the mean.}$$

■

2.5 Proof by Contradiction

In a *proof by contradiction* or *indirect proof*, you show that if a proposition were false, then some false fact would be true. Since a false fact can't be true, the proposition had better not be false. That is, the proposition really must be true.

Proof by contradiction is *always* a viable approach. However, as the name suggests, indirect proofs can be a little convoluted. So direct proofs are generally preferable as a matter of clarity.

Method: In order to prove a proposition P by contradiction:

1. Write, "We use proof by contradiction."
2. Write, "Suppose P is false."
3. Deduce something known to be false (a logical contradiction).
4. Write, "This is a contradiction. Therefore, P must be true."

As an example, we will use proof by contradiction to prove that $\sqrt{2}$ is irrational. Recall that a number is *rational* if it is equal to a ratio of integers. For example, $3.5 = 7/2$ and $0.1111\cdots = 1/9$ are rational numbers.

Theorem 2.5.1. *$\sqrt{2}$ is irrational.*

Proof. We use proof by contradiction. Suppose the claim is false; that is, $\sqrt{2}$ is rational. Then we can write $\sqrt{2}$ as a fraction n/d where n and d are positive integers. Furthermore, let's take n and d so that n/d is in *lowest terms* (that is, so that there is no number greater than 1 that divides both n and d).

Squaring both sides gives $2 = n^2/d^2$ and so $2d^2 = n^2$. This implies that n is a multiple of 2. Therefore n^2 must be a multiple of 4. But since $2d^2 = n^2$, we know $2d^2$ is a multiple of 4 and so d^2 is a multiple of 2. This implies that d is a multiple of 2.

So the numerator and denominator have 2 as a common factor, which contradicts the fact that n/d is in lowest terms. So $\sqrt{2}$ must be irrational. ■

Potential Pitfall

A proof of a proposition P by contradiction is really the same as proving the implication **T** IMPLIES P by contrapositive. Indeed, the contrapositive of T IMPLIES P is NOT(P) IMPLIES **F**. As we saw in Section 2.3.2, such a proof would be begin by assuming NOT(P) in an effort to derive a falsehood, just as you do in a proof by contradiction.

No matter how you think about it, it is important to remember that when you start by assuming NOT(P), you will derive conclusions along the way that are not necessarily true. (Indeed, the whole point of the method is to derive a falsehood.) This means that you cannot rely on intermediate results after a proof by contradiction is completed (for example, that n is even after the proof of Theorem 2.5.1). There was not much risk of that happening in the proof of Theorem 2.5.1, but when you are doing more complicated proofs that build up from several lemmas, some of which utilize a proof by contradiction, it will be important to keep track of which propositions only follow from a (false) assumption in a proof by contradiction.

2.6 Proofs about Sets

Sets are simple, flexible, and everywhere. You will find some set mentioned in nearly every section of this text. In fact, we have already talked about a lot of sets: the set of integers, the set of real numbers, and the set of positive even numbers, to name a few.

In this section, we'll see how to prove basic facts about sets. We'll start with some definitions just to make sure that you know the terminology and that you are comfortable working with sets.

2.6.1 Definitions

Informally, a *set* is a bunch of objects, which are called the *elements* of the set. The elements of a set can be just about anything: numbers, points in space, or even other sets. The conventional way to write down a set is to list the elements inside curly-braces. For example, here are some sets:

$$\begin{array}{lcll} A & = & \{\text{Alex, Tippy, Shells, Shadow}\} & \text{dead pets} \\ B & = & \{\text{red, blue, yellow}\} & \text{primary colors} \\ C & = & \{\{a,b\},\{a,c\},\{b,c\}\} & \text{a set of sets} \end{array}$$

This works fine for small finite sets. Other sets might be defined by indicating how to generate a list of them:

$$D = \{1, 2, 4, 8, 16, \ldots\} \qquad \text{the powers of 2}$$

The order of elements is not significant, so $\{x, y\}$ and $\{y, x\}$ are the same set written two different ways. Also, any object is, or is not, an element of a given

set—there is no notion of an element appearing more than once in a set.[3] So writing $\{x, x\}$ is just indicating the same thing twice, namely, that x is in the set. In particular, $\{x, x\} = \{x\}$.

The expression $e \in S$ asserts that e is an element of set S. For example, $32 \in D$ and blue $\in B$, but Tailspin $\notin A$—yet.

Some Popular Sets

Mathematicians have devised special symbols to represent some common sets.

symbol	set	elements
$\emptyset$	the empty set	none
$\mathbb{N}$	nonnegative integers	$\{0, 1, 2, 3, \ldots\}$
$\mathbb{Z}$	integers	$\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$
$\mathbb{Q}$	rational numbers	$\frac{1}{2},\ -\frac{5}{3},\ 16,$ etc.
$\mathbb{R}$	real numbers	$\pi,\ e,\ -9,\ \sqrt{2},$ etc.
$\mathbb{C}$	complex numbers	$i,\ \frac{19}{2},\ \sqrt{2} - 2i,$ etc.

A superscript "$+$" restricts a set to its positive elements; for example, $\mathbb{R}^+$ denotes the set of positive real numbers. Similarly, $\mathbb{R}^-$ denotes the set of negative reals.

Comparing and Combining Sets

The expression $S \subseteq T$ indicates that set S is a *subset* of set T, which means that every element of S is also an element of T (it could be that $S = T$). For example, $\mathbb{N} \subseteq \mathbb{Z}$ and $\mathbb{Q} \subseteq \mathbb{R}$ (every rational number is a real number), but $\mathbb{C} \not\subseteq \mathbb{Z}$ (not every complex number is an integer).

As a memory trick, notice that the $\subseteq$ points to the smaller set, just like a $\leq$ sign points to the smaller number. Actually, this connection goes a little further: there is a symbol $\subset$ analogous to $<$. Thus, $S \subset T$ means that S is a subset of T, but the two are *not* equal. So $A \subseteq A$, but $A \not\subset A$, for every set A.

There are several ways to combine sets. Let's define a couple of sets for use in examples:

$$X ::= \{1, 2, 3\}$$
$$Y ::= \{2, 3, 4\}$$

- The *union* of sets X and Y (denoted $X \cup Y$) contains all elements appearing in X or Y or both. Thus, $X \cup Y = \{1, 2, 3, 4\}$.

[3] It's not hard to develop a notion of *multisets* in which elements can occur more than once, but multisets are not ordinary sets.

- The *intersection* of X and Y (denoted $X \cap Y$) consists of all elements that appear in *both* X and Y. So $X \cap Y = \{2, 3\}$.
- The *set difference* of X and Y (denoted $X - Y$) consists of all elements that are in X, but not in Y. Therefore, $X - Y = \{1\}$ and $Y - X = \{4\}$.

The Complement of a Set

Sometimes we are focused on a particular domain, D. Then for any subset, A, of D, we define $\overline{A}$ to be the set of all elements of D *not* in A. That is, $\overline{A} ::= D - A$. The set $\overline{A}$ is called the *complement* of A.

For example, when the domain we're working with is the real numbers, the complement of the positive real numbers is the set of negative real numbers together with zero. That is,

$$\overline{\mathbb{R}^+} = \mathbb{R}^- \cup \{0\}.$$

It can be helpful to rephrase properties of sets using complements. For example, two sets, A and B, are said to be *disjoint* iff they have no elements in common, that is, $A \cap B = \emptyset$. This is the same as saying that A is a subset of the complement of B, that is, $A \subseteq \overline{B}$.

Cardinality

The *cardinality* of a set A is the number of elements in A and is denoted by $|A|$. For example,

$$|\emptyset| = 0,$$
$$|\{1, 2, 4\}| = 3, \text{ and}$$
$$|\mathbb{N}| \text{ is infinite.}$$

The Power Set

The set of all the subsets of a set, A, is called the *power set*, $\mathcal{P}(A)$, of A. So $B \in \mathcal{P}(A)$ iff $B \subseteq A$. For example, the elements of $\mathcal{P}(\{1, 2\})$ are $\emptyset, \{1\}, \{2\}$ and $\{1, 2\}$.

More generally, if A has n elements, then there are 2^n sets in $\mathcal{P}(A)$. In other words, if A is finite, then $|\mathcal{P}(A)| = 2^{|A|}$. For this reason, some authors use the notation 2^A instead of $\mathcal{P}(A)$ to denote the power set of A.

Sequences

Sets provide one way to group a collection of objects. Another way is in a *sequence*, which is a list of objects called *terms* or *components*. Short sequences

are commonly described by listing the elements between parentheses; for example, (a, b, c) is a sequence with three terms.

While both sets and sequences perform a gathering role, there are several differences.

- The elements of a set are required to be distinct, but terms in a sequence can be the same. Thus, (a, b, a) is a valid sequence of length three, but $\{a, b, a\}$ is a set with two elements—not three.
- The terms in a sequence have a specified order, but the elements of a set do not. For example, (a, b, c) and (a, c, b) are different sequences, but $\{a, b, c\}$ and $\{a, c, b\}$ are the same set.
- Texts differ on notation for the *empty sequence*; we use λ for the empty sequence and $\emptyset$ for the empty set.

Cross Products

The product operation is one link between sets and sequences. A *product of sets*, $S_1 \times S_2 \times \cdots \times S_n$, is a new set consisting of all sequences where the first component is drawn from S_1, the second from S_2, and so forth. For example, $\mathbb{N} \times \{a, b\}$ is the set of all pairs whose first element is a nonnegative integer and whose second element is an a or a b:

$$\mathbb{N} \times \{a, b\} = \{(0, a), (0, b), (1, a), (1, b), (2, a), (2, b), \dots\}$$

A product of n copies of a set S is denoted S^n. For example, $\{0, 1\}^3$ is the set of all 3-bit sequences:

$$\{0, 1\}^3 = \{(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)\}$$

2.6.2 Set Builder Notation

An important use of predicates is in *set builder notation*. We'll often want to talk about sets that cannot be described very well by listing the elements explicitly or by taking unions, intersections, etc., of easily-described sets. Set builder notation often comes to the rescue. The idea is to define a *set* using a *predicate*; in particular, the set consists of all values that make the predicate true. Here are some examples of set builder notation:

$$A ::= \{n \in \mathbb{N} \mid n \text{ is a prime and } n = 4k + 1 \text{ for some integer } k\}$$
$$B ::= \{x \in \mathbb{R} \mid x^3 - 3x + 1 > 0\}$$
$$C ::= \{a + bi \in \mathbb{C} \mid a^2 + 2b^2 \leq 1\}$$

The set A consists of all nonnegative integers n for which the predicate

"n is a prime and $n = 4k + 1$ for some integer k"

is true. Thus, the smallest elements of A are:

$$5, 13, 17, 29, 37, 41, 53, 57, 61, 73, \ldots.$$

Trying to indicate the set A by listing these first few elements wouldn't work very well; even after ten terms, the pattern is not obvious! Similarly, the set B consists of all real numbers x for which the predicate

$$x^3 - 3x + 1 > 0$$

is true. In this case, an explicit description of the set B in terms of intervals would require solving a cubic equation. Finally, set C consists of all complex numbers $a + bi$ such that:

$$a^2 + 2b^2 \leq 1$$

This is an oval-shaped region around the origin in the complex plane.

2.6.3 Proving Set Equalities

Two sets are defined to be equal if they contain the same elements. That is, $X = Y$ means that $z \in X$ if and only if $z \in Y$, for all elements, z. (This is actually the first of the ZFC axioms.) So set equalities can often be formulated and proved as "iff" theorems. For example:

Theorem 2.6.1 (*Distributive Law* for Sets)**.** *Let A, B, and C be sets. Then:*

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C) \tag{2.5}$$

Proof. The equality (2.5) is equivalent to the assertion that

$$z \in A \cap (B \cup C) \quad \text{iff} \quad z \in (A \cap B) \cup (A \cap C) \tag{2.6}$$

for all z. This assertion looks very similar to the Distributive Law for AND and OR that we proved in Section 1.4 (equation 1.6). Namely, if P, Q, and R are propositions, then

$$[P \text{ AND } (Q \text{ OR } R)] \text{ IFF } [(P \text{ AND } Q) \text{ OR } (P \text{ AND } R)] \tag{2.7}$$

Using this fact, we can now prove (2.6) by a chain of iff's:

$z \in A \cap (B \cup C)$

iff	$(z \in A)$ AND $(z \in B \cup C)$	(def of $\cap$)
iff	$(z \in A)$ AND $(z \in B$ OR $z \in C)$	(def of $\cup$)
iff	$(z \in A$ AND $z \in B)$ OR $(z \in A$ AND $z \in C)$	(equation 2.7)
iff	$(z \in A \cap B)$ OR $(z \in A \cap C)$	(def of $\cap$)
iff	$z \in (A \cap B) \cup (A \cap C)$	(def of $\cup$) ■

Many other set equalities can be derived from other valid propositions and proved in an analogous manner. In particular, propositions such as P, Q and R are replaced with sets such as A, B, and C, AND ($\wedge$) is replaced with intersection ($\cap$), OR ($\vee$) is replaced with union ($\cup$), NOT is replaced with complement (for example, $\overline{P}$ would become $\overline{A}$), and IFF becomes set equality ($=$). Of course, you should always check that any alleged set equality derived in this manner is indeed true.

2.6.4 Russell's Paradox and the Logic of Sets

Reasoning naively about sets can sometimes be tricky. In fact, one of the earliest attempts to come up with precise axioms for sets by a late nineteenth century logician named Gotlob *Frege* was shot down by a three line argument known as *Russell's Paradox*:[4] This was an astonishing blow to efforts to provide an axiomatic foundation for mathematics.

Russell's Paradox

Let S be a variable ranging over all sets, and define

$$W ::= \{S \mid S \notin S\}.$$

So by definition, for any set S,

$$S \in W \text{ iff } S \notin S.$$

In particular, we can let S be W, and obtain the contradictory result that

$$W \in W \text{ iff } W \notin W.$$

A way out of the paradox was clear to Russell and others at the time: *it's unjustified to assume that W is a set*. So the step in the proof where we let S be W has no justification, because S ranges over sets, and W may not be a set. In fact, the paradox implies that W had better not be a set!

But denying that W is a set means we must *reject* the very natural axiom that every mathematically well-defined collection of elements is actually a set. So the problem faced by Frege, Russell and their colleagues was how to specify *which*

[4] Bertrand *Russell* was a mathematician/logician at Cambridge University at the turn of the Twentieth Century. He reported that when he felt too old to do mathematics, he began to study and write about philosophy, and when he was no longer smart enough to do philosophy, he began writing about politics. He was jailed as a conscientious objector during World War I. For his extensive philosophical and political writing, he won a Nobel Prize for Literature.

well-defined collections are sets. Russell and his fellow Cambridge University colleague Whitehead immediately went to work on this problem. They spent a dozen years developing a huge new axiom system in an even huger monograph called *Principia Mathematica*.

Over time, more efficient axiom systems were developed and today, it is generally agreed that, using some simple logical deduction rules, essentially all of mathematics can be derived from the Axioms of Zermelo-Frankel Set Theory with Choice (ZFC). We are *not* going to be working with these axioms in this course, but just in case you are interested, we have included them as a sidebar below.

The ZFC axioms avoid Russell's Paradox because they imply that no set is ever a member of itself. Unfortunately, this does not necessarily mean that there are not other paradoxes lurking around out there, just waiting to be uncovered by future mathematicians.

ZFC Axioms

Extensionality. Two sets are equal if they have the same members. In formal logical notation, this would be stated as:

$$(\forall z.\ (z \in x \text{ IFF } z \in y)) \text{ IMPLIES } x = y.$$

Pairing. For any two sets x and y, there is a set, $\{x, y\}$, with x and y as its only elements:

$$\forall x, y.\ \exists u.\ \forall z.\ [z \in u \text{ IFF } (z = x \text{ OR } z = y)]$$

Union. The union, u, of a collection, z, of sets is also a set:

$$\forall z.\ \exists u \forall x.\ (\exists y.\ x \in y \text{ AND } y \in z) \text{ IFF } x \in u.$$

Infinity. There is an infinite set. Specifically, there is a nonempty set, x, such that for any set $y \in x$, the set $\{y\}$ is also a member of x.

Subset. Given any set, x, and any proposition $P(y)$, there is a set containing precisely those elements $y \in x$ for which $P(y)$ holds.

Power Set. All the subsets of a set form another set:

$$\forall x.\ \exists p.\ \forall u.\ u \subseteq x \text{ IFF } u \in p.$$

Replacement. Suppose a formula, ϕ, of set theory defines the graph of a function, that is,

$$\forall x, y, z.\ [\phi(x, y) \text{ AND } \phi(x, z)] \text{ IMPLIES } y = z.$$

Then the image of any set, s, under that function is also a set, t. Namely,

$$\forall s\, \exists t\, \forall y.\, [\exists x.\, \phi(x, y) \text{ IFF } y \in t].$$

Foundation. There cannot be an infinite sequence

$$\cdots \in x_n \in \cdots \in x_1 \in x_0$$

of sets each of which is a member of the previous one. This is equivalent to saying every nonempty set has a "member-minimal" element. Namely, define

$$\text{member-minimal}(m, x) ::= [m \in x \text{ AND } \forall y \in x.\, y \notin m].$$

Then the Foundation axiom is

$$\forall x.\, x \neq \emptyset \text{ IMPLIES } \exists m.\, \text{member-minimal}(m, x).$$

Choice. Given a set, s, whose members are nonempty sets no two of which have any element in common, then there is a set, c, consisting of exactly one element from each set in s.

$$\begin{array}{lll} \exists y \forall z \forall w & ((z \in w \text{ AND } w \in x) \text{ IMPLIES} & \\ & \exists v \exists u (\exists t ((u \in w \text{ AND } & w \in t) \quad \text{AND}(u \in t \text{ AND } t \in y)) \\ & & \text{IFF} u = v)) \end{array}$$

2.7 *Good* Proofs in Practice

One purpose of a proof is to establish the truth of an assertion with absolute certainty. Mechanically checkable proofs of enormous length or complexity can accomplish this. But humanly intelligible proofs are the only ones that help someone understand the subject. Mathematicians generally agree that important mathematical results can't be fully understood until their proofs are understood. That is why proofs are an important part of the curriculum.

To be understandable and helpful, more is required of a proof than just logical correctness: a good proof must also be clear. Correctness and clarity usually go together; a well-written proof is more likely to be a correct proof, since mistakes are harder to hide.

In practice, the notion of proof is a moving target. Proofs in a professional research journal are generally unintelligible to all but a few experts who know all the terminology and prior results used in the proof. Conversely, proofs in the first weeks of an introductory course like *Mathematics for Computer Science* would be regarded as tediously long-winded by a professional mathematician. In fact, what we accept as a good proof later in the term will be different than what we consider to be a good proof in the first couple of weeks of this course. But even so, we can offer some general tips on writing good proofs:

State your game plan. A good proof begins by explaining the general line of reasoning. For example, "We use case analysis" or "We argue by contradiction."

Keep a linear flow. Sometimes proofs are written like mathematical mosaics, with juicy tidbits of independent reasoning sprinkled throughout. This is not good. The steps of an argument should follow one another in an intelligible order.

A proof is an essay, not a calculation. Many students initially write proofs the way they compute integrals. The result is a long sequence of expressions without explanation, making it very hard to follow. This is bad. A good proof usually looks like an essay with some equations thrown in. Use complete sentences.

Avoid excessive symbolism. Your reader is probably good at understanding words, but much less skilled at reading arcane mathematical symbols. So use words where you reasonably can.

Revise and simplify. Your readers will be grateful.

Introduce notation thoughtfully. Sometimes an argument can be greatly simplified by introducing a variable, devising a special notation, or defining a new term. But do this sparingly since you're requiring the reader to remember all that new stuff. And remember to actually *define* the meanings of new variables, terms, or notations; don't just start using them!

Structure long proofs. Long programs are usually broken into a hierarchy of smaller procedures. Long proofs are much the same. Facts needed in your proof that are easily stated, but not readily proved are best pulled out and proved in preliminary lemmas. Also, if you are repeating essentially the same argument over and over, try to capture that argument in a general lemma, which you can cite repeatedly instead.

Be wary of the "obvious". When familiar or truly obvious facts are needed in a proof, it's OK to label them as such and to not prove them. But remember

that what's obvious to you, may not be—and typically is not—obvious to your reader.

Most especially, don't use phrases like "clearly" or "obviously" in an attempt to bully the reader into accepting something you're having trouble proving. Also, go on the alert whenever you see one of these phrases in someone else's proof.

Finish. At some point in a proof, you'll have established all the essential facts you need. Resist the temptation to quit and leave the reader to draw the "obvious" conclusion. Instead, tie everything together yourself and explain why the original claim follows.

The analogy between good proofs and good programs extends beyond structure. The same rigorous thinking needed for proofs is essential in the design of critical computer systems. When algorithms and protocols only "mostly work" due to reliance on hand-waving arguments, the results can range from problematic to catastrophic. An early example was the Therac 25, a machine that provided radiation therapy to cancer victims, but occasionally killed them with massive overdoses due to a software race condition. A more recent (August 2004) example involved a single faulty command to a computer system used by United and American Airlines that grounded the entire fleet of both companies—and all their passengers!

It is a certainty that we'll all one day be at the mercy of critical computer systems designed by you and your classmates. So we really hope that you'll develop the ability to formulate rock-solid logical arguments that a system actually does what you think it does!

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.

3.1.3 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):

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

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 $P(m)$ 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.

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

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

which proves $P(n+1)$.

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

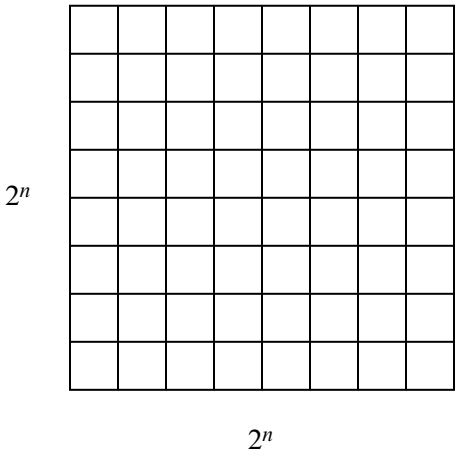


Figure 3.1 A $2^n \times 2^n$ courtyard for $n = 3$.

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.

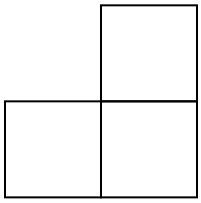


Figure 3.2 The special L-shaped tile.

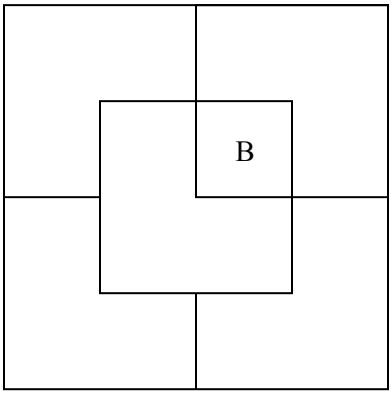


Figure 3.3 A tiling using L-shaped tiles for $n = 2$ with Bill in a center square.

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

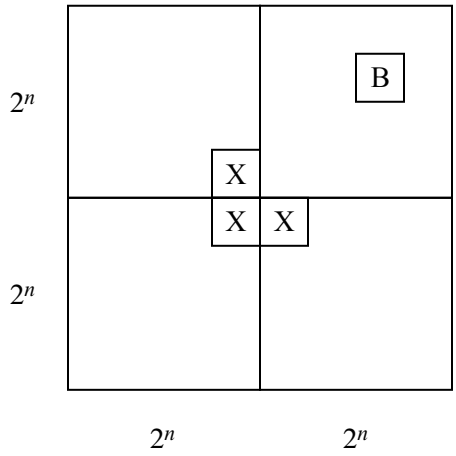


Figure 3.4 Using a stronger inductive hypothesis to prove Theorem 3.2.2.

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.

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, $h_2, \ldots, h_n$)—and likewise h_{n+1} is the same color as the remaining horses besides h_1–$h_2, \ldots, h_n$. Since h_1 and h_{n+1} are the same color as $h_2, \ldots, h_n$, 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 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 "$h_1,\ h_2,\ \ldots,\ h_n,\ h_{n+1}$" creates the impression that there are some remaining horses (namely $h_2, \ldots, h_n$) 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.

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

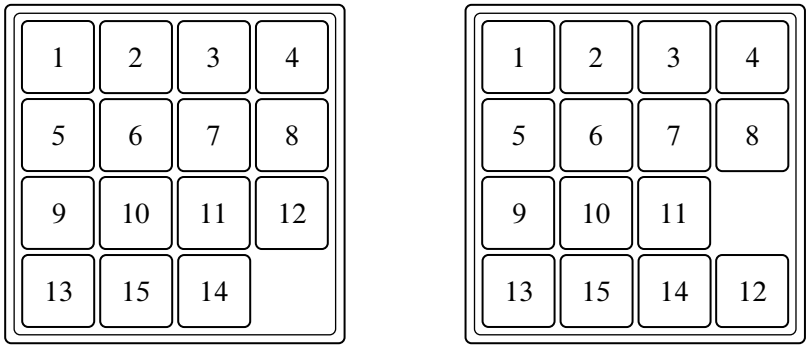
In summary, if you would like to prove that some property NICE holds for every step of a process, then it is often helpful to use the following method:

- 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.



(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).

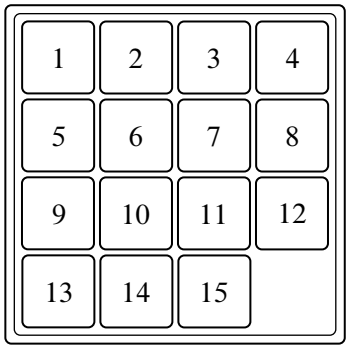


Figure 3.6 The desired final configuration for the 15-puzzle. Can it be achieved by only moving one block at a time into an adjacent hole?

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.

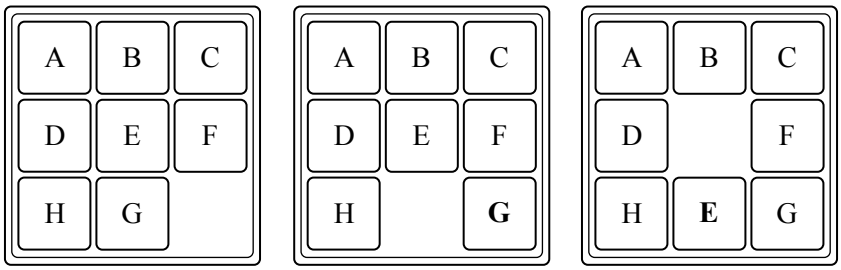


Figure 3.7 The 8-Puzzle in its initial configuration (a) and after one (b) and two (c) possible moves.

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

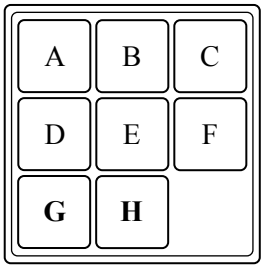
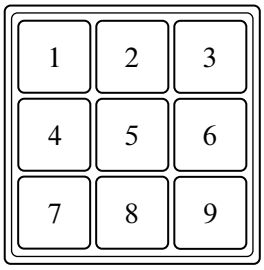


Figure 3.8 The desired final configuration of the 8-puzzle.



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:

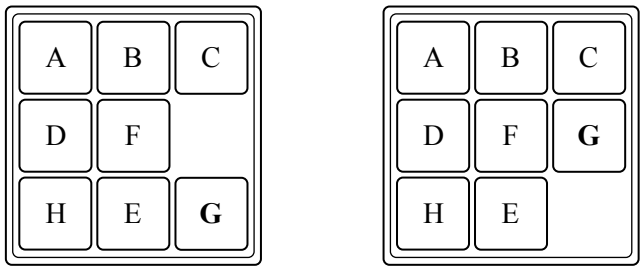
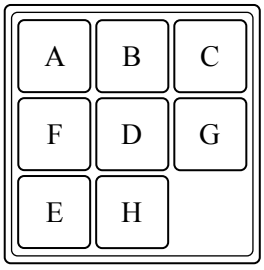


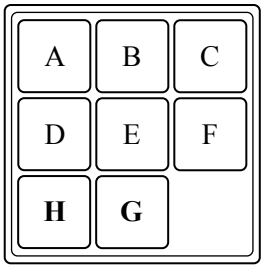
Figure 3.9 An example of a column move in which the G-tile is moved into the adjacent hole above. In this case, G changes order with E and H.

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.

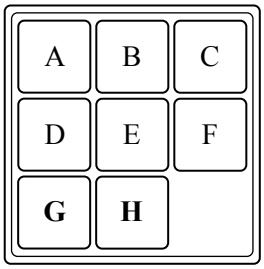
For example, in the puzzle below, there are three inversions: (D, F), (E, F), (E, G).



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



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.

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.

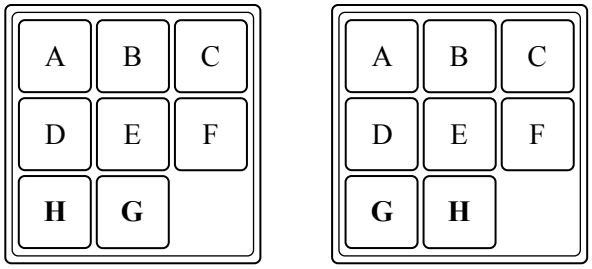
Base case: After zero moves, exactly one pair of tiles is inverted (G and H), which is an odd number. Therefore $P(0)$ is true.

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 unreachable. ■

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***

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

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.

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

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!

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:

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

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`:

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

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.

Definition 3.5.2. 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 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 $\lambda \in \text{RecMatch}$ 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.

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:**

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, $\mathtt{k}$, 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

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

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 $s \in \mathtt{brkts}$:

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

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

$$f_2(y, z) ::= y10yz$$

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

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

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.

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]

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

     (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:

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

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 $s \in$ 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 $M =$ 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

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

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

$$\begin{aligned} b &::= aa \\ &= [\,[\,]\,][\,[\,]\,] \end{aligned}$$

and

$$\begin{aligned} c &::= bb \\ &= [\,[\,]\,][\,[\,]\,][\,[\,]\,][\,[\,]\,] \end{aligned}$$

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 $\mathbb{N}$.

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 $\text{fac}(n)$ and define it recursively as follows:

- **Base Case:** $\text{fac}(0) ::= 1$.
- **Constructor Case:** $\text{fac}(n+1) ::= (n+1) \cdot \text{fac}(n)$ 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, $\text{fib}(n)$, can be defined recursively by:

- **Base Cases:** $\text{fib}(0) ::= 0$ and $\text{fib}(1) ::= 1$
- **Constructor Case:** $\text{fib}(n) ::= \text{fib}(n-1) + \text{fib}(n-2)$ 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 $\text{fib}(4)$? Well, $\text{fib}(2) = \text{fib}(1) + \text{fib}(0) = 1$, $\text{fib}(3) = \text{fib}(2) + \text{fib}(1) = 2$, so $\text{fib}(4) = 3$. 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 $n \geq 0$.

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:

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

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.

4 Number Theory

Number theory is the study of the integers. *Why* anyone would want to study the integers is not immediately obvious. First of all, what's to know? There's 0, there's 1, 2, 3, and so on, and, oh yeah, -1, -2, ... Which one don't you understand? Second, what practical value is there in it? The mathematician G. H. Hardy expressed pleasure in its impracticality when he wrote:

> [Number theorists] may be justified in rejoicing that there is one science, at any rate, and that their own, whose very remoteness from ordinary human activities should keep it gentle and clean.

Hardy was specially concerned that number theory not be used in warfare; he was a pacifist. You may applaud his sentiments, but he got it wrong: Number Theory underlies modern cryptography, which is what makes secure online communication possible. Secure communication is of course crucial in war—which may leave poor Hardy spinning in his grave. It's also central to online commerce. Every time you buy a book from Amazon, check your grades on WebSIS, or use a PayPal account, you are relying on number theoretic algorithms.

Number theory also provides an excellent environment for us to practice and apply the proof techniques that we developed in Chapters 2 and 3.

Since we'll be focusing on properties of the integers, we'll adopt the default convention in this chapter that *variables range over the set of integers*, $\mathbb{Z}$.

4.1 Divisibility

The nature of number theory emerges as soon as we consider the *divides* relation

$$a \text{ divides } b \quad \text{iff} \quad ak = b \text{ for some } k.$$

The notation, $a \mid b$, is an abbreviation for "a divides b." If $a \mid b$, then we also say that b is a *multiple* of a. A consequence of this definition is that every number divides zero.

This seems simple enough, but let's play with this definition. The Pythagoreans, an ancient sect of mathematical mystics, said that a number is *perfect* if it equals the sum of its positive integral divisors, excluding itself. For example, $6 = 1 + 2 + 3$ and $28 = 1 + 2 + 4 + 7 + 14$ are perfect numbers. On the other hand, 10 is not perfect because $1+2+5 = 8$, and 12 is not perfect because $1+2+3+4+6 = 16$.

Euclid characterized all the *even* perfect numbers around 300 BC. But is there an *odd* perfect number? More than two thousand years later, we still don't know! All numbers up to about 10^{300} have been ruled out, but no one has proved that there isn't an odd perfect number waiting just over the horizon.

So a half-page into number theory, we've strayed past the outer limits of human knowledge! This is pretty typical; number theory is full of questions that are easy to pose, but incredibly difficult to answer.[1] For example, several such problems are shown in the box on the following page. Interestingly, we'll see that computer scientists have found ways to turn some of these difficulties to their advantage.

4.1.1 Facts about Divisibility

The lemma below states some basic facts about divisibility that are *not* difficult to prove:

Lemma 4.1.1. *The following statements about divisibility hold.*

1. *If $a \mid b$, then $a \mid bc$ for all c.*
2. *If $a \mid b$ and $b \mid c$, then $a \mid c$.*
3. *If $a \mid b$ and $a \mid c$, then $a \mid sb + tc$ for all s and t.*
4. *For all $c \neq 0$, $a \mid b$ if and only if $ca \mid cb$.*

Proof. We'll prove only part 2.; the other proofs are similar.

Proof of 2: Assume $a \mid b$ and $b \mid c$. Since $a \mid b$, there exists an integer k_1 such that $ak_1 = b$. Since $b \mid c$, there exists an integer k_2 such that $bk_2 = c$. Substituting ak_1 for b in the second equation gives $(ak_1)k_2 = c$. So $a(k_1k_2) = c$, which implies that $a \mid c$. ■

4.1.2 When Divisibility Goes Bad

As you learned in elementary school, if one number does *not* evenly divide another, you get a "quotient" and a "remainder" left over. More precisely:

Theorem 4.1.2 (Division Theorem)**.** [3] *Let n and d be integers such that $d > 0$. Then there exists a unique pair of integers q and r, such that*

$$n = q \cdot d + r \text{ AND } 0 \leq r < d. \tag{4.1}$$

[1] *Don't Panic*—we're going to stick to some relatively benign parts of number theory. These super-hard unsolved problems rarely get put on problem sets.

[3] This theorem is often called the "Division Algorithm," even though it is not what we would call an algorithm. We will take this familiar result for granted without proof.

Famous Conjectures in Number Theory

Fermat's Last Theorem There are no positive integers x, y, and z such that

$$x^n + y^n = z^n$$

for some integer $n > 2$. In a book he was reading around 1630, Fermat claimed to have a proof but not enough space in the margin to write it down. Wiles finally gave a proof of the theorem in 1994, after seven years of working in secrecy and isolation in his attic. His proof did not fit in any margin.

Goldbach Conjecture Every even integer greater than two is equal to the sum of two primes[2]. For example, $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5$, etc. The conjecture holds for all numbers up to 10^{16}. In 1939 Schnirelman proved that every even number can be written as the sum of not more than 300,000 primes, which was a start. Today, we know that every even number is the sum of at most 6 primes.

Twin Prime Conjecture There are infinitely many primes p such that $p + 2$ is also a prime. In 1966 Chen showed that there are infinitely many primes p such that $p + 2$ is the product of at most two primes. So the conjecture is known to be *almost* true!

Primality Testing There is an efficient way to determine whether a number is prime. A naive search for factors of an integer n takes a number of steps proportional to $\sqrt{n}$, which is exponential in the *size* of n in decimal or binary notation. All known procedures for prime checking blew up like this on various inputs. Finally in 2002, an amazingly simple, new method was discovered by Agrawal, Kayal, and Saxena, which showed that prime testing only required a polynomial number of steps. Their paper began with a quote from Gauss emphasizing the importance and antiquity of the problem even in his time—two centuries ago. So prime testing is definitely not in the category of infeasible problems requiring an exponentially growing number of steps in bad cases.

Factoring Given the product of two large primes $n = pq$, there is no efficient way to recover the primes p and q. The best known algorithm is the "number field sieve", which runs in time proportional to:

$$e^{1.9(\ln n)^{1/3}(\ln \ln n)^{2/3}}$$

This is infeasible when n has 300 digits or more.

The number q is called the *quotient* and the number r is called the *remainder* of n divided by d. We use the notation $\text{qcnt}(n, d)$ for the quotient and $\text{rem}(n, d)$ for the remainder.

For example, $\text{qcnt}(2716, 10) = 271$ and $\text{rem}(2716, 10) = 6$, since $2716 = 271 \cdot 10 + 6$. Similarly, $\text{rem}(-11, 7) = 3$, since $-11 = (-2) \cdot 7 + 3$. There is a remainder operator built into many programming languages. For example, the expression "32 % 5" evaluates to 2 in Java, C, and C++. However, all these languages treat negative numbers strangely.

4.1.3 Die Hard

Simon: On the fountain, there should be 2 jugs, do you see them? A 5-gallon and a 3-gallon. Fill one of the jugs with exactly 4 gallons of water and place it on the scale and the timer will stop. You must be precise; one ounce more or less will result in detonation. If you're still alive in 5 minutes, we'll speak.
Bruce: Wait, wait a second. I don't get it. Do you get it?
Samuel: No.
Bruce: Get the jugs. Obviously, we can't fill the 3-gallon jug with 4 gallons of water.
Samuel: Obviously.
Bruce: All right. I know, here we go. We fill the 3-gallon jug exactly to the top, right?
Samuel: Uh-huh.
Bruce: Okay, now we pour this 3 gallons into the 5-gallon jug, giving us exactly 3 gallons in the 5-gallon jug, right?
Samuel: Right, then what?
Bruce: All right. We take the 3-gallon jug and fill it a third of the way...
Samuel: No! He said, "Be precise." Exactly 4 gallons.
Bruce: Sh—. Every cop within 50 miles is running his a— off and I'm out here playing kids' games in the park.
Samuel: Hey, you want to focus on the problem at hand?

The preceding script is from the movie *Die Hard 3: With a Vengeance*. In the movie, Samuel L. Jackson and Bruce Willis have to disarm a bomb planted by the diabolical Simon Gruber. Fortunately, they find a solution in the nick of time. (No doubt reading the script helped.) On the surface, *Die Hard 3* is just a B-grade action movie; however, we think the inner message of the film is that everyone should learn at least a little number theory.

Unfortunately, Hollywood never lets go of a gimmick. Although there were no water jug tests in *Die Hard 4: Live Free or Die Hard*, rumor has it that the jugs will

return in future sequels:

Die Hard 5: Die Hardest Bruce goes on vacation and—shockingly—happens into a terrorist plot. To save the day, he must make 3 gallons using 21- and 26-gallon jugs.

Die Hard 6: Die of Old Age Bruce must save his assisted living facility from a criminal mastermind by forming 2 gallons with 899- and 1147-gallon jugs.

Die Hard 7: Die Once and For All Bruce has to make 4 gallons using 3- and 6-gallon jugs.

It would be nice if we could solve all these silly water jug questions at once. In particular, how can one form g gallons using jugs with capacities a and b?

That's where number theory comes in handy.

Finding an Invariant Property

Suppose that we have water jugs with capacities a and b with $b \geq a$. The state of the system is described below with a pair of numbers (x, y), where x is the amount of water in the jug with capacity a and y is the amount in the jug with capacity b. Let's carry out sample operations and see what happens, assuming the b-jug is big enough:

$$\begin{aligned}
(0,0) &\to (a,0) && \text{fill first jug}\\
&\to (0,a) && \text{pour first into second}\\
&\to (a,a) && \text{fill first jug}\\
&\to (2a-b,b) && \text{pour first into second (assuming } 2a \geq b)\\
&\to (2a-b,0) && \text{empty second jug}\\
&\to (0,2a-b) && \text{pour first into second}\\
&\to (a,2a-b) && \text{fill first}\\
&\to (3a-2b,b) && \text{pour first into second (assuming } 3a \geq 2b)
\end{aligned}$$

What leaps out is that at every step, the amount of water in each jug is of the form

$$s \cdot a + t \cdot b \tag{4.2}$$

for some integers s and t. An expression of the form (4.2) is called an *integer linear combination* of a and b, but in this chapter we'll just call it a *linear combination*, since we're only talking integers. So we're suggesting:

Lemma 4.1.3. *Suppose that we have water jugs with capacities a and b. Then the amount of water in each jug is always a linear combination of a and b.*

Lemma 4.1.3 is easy to prove by induction on the number of pourings.

Proof. The induction hypothesis, $P(n)$, is the proposition that after n steps, the amount of water in each jug is a linear combination of a and b.
Base case: ($n = 0$). $P(0)$ is true, because both jugs are initially empty, and $0 \cdot a + 0 \cdot b = 0$.
Inductive step. We assume by induction hypothesis that after n steps the amount of water in each jug is a linear combination of a and b. There are two cases:

- If we fill a jug from the fountain or empty a jug into the fountain, then that jug is empty or full. The amount in the other jug remains a linear combination of a and b. So $P(n+1)$ holds.

- Otherwise, we pour water from one jug to another until one is empty or the other is full. By our assumption, the amount in each jug is a linear combination of a and b before we begin pouring:

$$j_1 = s_1 \cdot a + t_1 \cdot b$$
$$j_2 = s_2 \cdot a + t_2 \cdot b$$

  After pouring, one jug is either empty (contains 0 gallons) or full (contains a or b gallons). Thus, the other jug contains either $j_1 + j_2$ gallons, $j_1 + j_2 - a$, or $j_1 + j_2 - b$ gallons, all of which are linear combinations of a and b. So $P(n+1)$ holds in this case as well.

So in any case, $P(n+1)$ follows, completing the proof by induction. ■

So we have established that the jug problem has an invariant property, namely that the amount of water in every jug is always a linear combination of the capacities of the jugs. This lemma has an important corollary:

Corollary 4.1.4. *Bruce dies.*

Proof. In Die Hard 7, Bruce has water jugs with capacities 3 and 6 and must form 4 gallons of water. However, the amount in each jug is always of the form $3s + 6t$ by Lemma 4.1.3. This is always a multiple of 3 by part 3 of Lemma 4.1.1, so he cannot measure out 4 gallons. ■

But Lemma 4.1.3 isn't very satisfying. We've just managed to recast a pretty understandable question about water jugs into a complicated question about linear combinations. This might not seem like a lot of progress. Fortunately, linear combinations are closely related to something more familiar, namely greatest common divisors, and these will help us solve the water jug problem.

4.2 The Greatest Common Divisor

The *greatest common divisor* of a and b is exactly what you'd guess: the largest number that is a divisor of both a and b. It is denoted by $\gcd(a, b)$. For example, $\gcd(18, 24) = 6$. The greatest common divisor turns out to be a very valuable piece of information about the relationship between a and b and for reasoning about integers in general. So we'll be making lots of arguments about greatest common divisors in what follows.

4.2.1 Linear Combinations and the GCD

The theorem below relates the greatest common divisor to linear combinations. This theorem is *very* useful; take the time to understand it and then remember it!

Theorem 4.2.1. *The greatest common divisor of a and b is equal to the smallest positive linear combination of a and b.*

For example, the greatest common divisor of 52 and 44 is 4. And, sure enough, 4 is a linear combination of 52 and 44:

$$6 \cdot 52 + (-7) \cdot 44 = 4$$

Furthermore, no linear combination of 52 and 44 is equal to a smaller positive integer.

Proof of Theorem 4.2.1. By the Well Ordering Principle, there is a smallest positive linear combination of a and b; call it m. We'll prove that $m = \gcd(a, b)$ by showing both $\gcd(a, b) \leq m$ and $m \leq \gcd(a, b)$.

First, we show that $\gcd(a, b) \leq m$. Now any common divisor of a and b—that is, any c such that $c \mid a$ and $c \mid b$—will divide both sa and tb, and therefore also $sa + tb$ for any s and t. The $\gcd(a, b)$ is by definition a common divisor of a and b, so

$$\gcd(a, b) \mid sa + tb \tag{4.3}$$

for every s and t. In particular, $\gcd(a, b) \mid m$, which implies that $\gcd(a, b) \leq m$.

Now, we show that $m \leq \gcd(a, b)$. We do this by showing that $m \mid a$. A symmetric argument shows that $m \mid b$, which means that m is a common divisor of a and b. Thus, m must be less than or equal to the *greatest* common divisor of a and b.

All that remains is to show that $m \mid a$. By the Division Algorithm, there exists a quotient q and remainder r such that:

$$a = q \cdot m + r \qquad \text{(where } 0 \leq r < m\text{)}$$

Recall that $m = sa + tb$ for some integers s and t. Substituting in for m gives:

$$\begin{aligned} a &= q \cdot (sa + tb) + r, \qquad \text{so} \\ r &= (1 - qs)a + (-qt)b. \end{aligned}$$

We've just expressed r as a linear combination of a and b. However, m is the *smallest positive* linear combination and $0 \leq r < m$. The only possibility is that the remainder r is not positive; that is, $r = 0$. This implies $m \mid a$. ■

Corollary 4.2.2. *An integer is linear combination of a and b iff it is a multiple of* $\gcd(a, b)$.

Proof. By (4.3), every linear combination of a and b is a multiple of $\gcd(a, b)$. Conversely, since $\gcd(a, b)$ is a linear combination of a and b, every multiple of $\gcd(a, b)$ is as well. ■

Now we can restate the water jugs lemma in terms of the greatest common divisor:

Corollary 4.2.3. *Suppose that we have water jugs with capacities a and b. Then the amount of water in each jug is always a multiple of* $\gcd(a, b)$.

For example, there is no way to form 4 gallons using 3- and 6-gallon jugs, because 4 is not a multiple of $\gcd(3, 6) = 3$.

4.2.2 Properties of the Greatest Common Divisor

We'll often make use of some basic gcd facts:

Lemma 4.2.4. *The following statements about the greatest common divisor hold:*

1. *Every common divisor of a and b divides* $\gcd(a, b)$.
2. $\gcd(ka, kb) = k \cdot \gcd(a, b)$ *for all* $k > 0$.
3. *If* $\gcd(a, b) = 1$ *and* $\gcd(a, c) = 1$, *then* $\gcd(a, bc) = 1$.
4. *If* $a \mid bc$ *and* $\gcd(a, b) = 1$, *then* $a \mid c$.
5. $\gcd(a, b) = \gcd(b, \text{rem}(a, b))$.

Here's the trick to proving these statements: translate the gcd world to the linear combination world using Theorem 4.2.1, argue about linear combinations, and then translate back using Theorem 4.2.1 again.

Proof. We prove only parts 3. and 4.

Proof of 3. The assumptions together with Theorem 4.2.1 imply that there exist integers s, t, u, and v such that:

$$sa + tb = 1$$
$$ua + vc = 1$$

Multiplying these two equations gives:

$$(sa + tb)(ua + vc) = 1$$

The left side can be rewritten as $a \cdot (asu + btu + csv) + bc(tv)$. This is a linear combination of a and bc that is equal to 1, so $\gcd(a, bc) = 1$ by Theorem 4.2.1.

Proof of 4. Theorem 4.2.1 says that $\gcd(ac, bc)$ is equal to a linear combination of ac and bc. Now $a \mid ac$ trivially and $a \mid bc$ by assumption. Therefore, a divides *every* linear combination of ac and bc. In particular, a divides $\gcd(ac, bc) = c \cdot \gcd(a, b) = c \cdot 1 = c$. The first equality uses part 2. of this lemma, and the second uses the assumption that $\gcd(a, b) = 1$. ■

4.2.3 Euclid's Algorithm

Part (5) of Lemma 4.2.4 is useful for quickly computing the greatest common divisor of two numbers. For example, we could compute the greatest common divisor of 1147 and 899 by repeatedly applying part (5):

$$\begin{aligned}
\gcd(1147, 899) &= \gcd\big(899, \underbrace{\operatorname{rem}(1147, 899)}_{=248}\big) \\
&= \gcd\big(248, \underbrace{\operatorname{rem}(899, 248)}_{=155}\big) \\
&= \gcd\big(155, \underbrace{\operatorname{rem}(248, 155)}_{=93}\big) \\
&= \gcd\big(93, \underbrace{\operatorname{rem}(155, 93)}_{=62}\big) \\
&= \gcd\big(62, \underbrace{\operatorname{rem}(93, 62)}_{=31}\big) \\
&= \gcd\big(31, \underbrace{\operatorname{rem}(62, 31)}_{=0}\big) \\
&= \gcd(31, 0) \\
&= 31
\end{aligned}$$

The last equation might look wrong, but 31 is a divisor of both 31 and 0 since every integer divides 0.

This process is called *Euclid's algorithm* and it was discovered by the Greeks over 3000 years ago. You can prove that the algorithm always eventually terminates by using induction and the fact that the numbers in each step keep getting smaller until the remainder is 0, whereupon you have computed the GCD. In fact, the numbers are getting smaller quickly (by at least a factor of 2 every two steps) and so Euler's Algorithm is quite fast. The fact that Euclid's Algorithm actually produces the GCD (and not something different) can also be proved by an inductive invariant argument.

The calculation that $\gcd(1147, 899) = 31$ together with Corollary 4.2.3 implies that there is no way to measure out 2 gallons of water using jugs with capacities 1147 and 899, since we can only obtain multiples of 31 gallons with these jugs. This is good news—Bruce won't even survive *Die Hard 6*!

But what about Die Hard 5? Is it possible for Bruce to make 3 gallons using 21- and 26-gallon jugs? Using Euclid's algorithm:

$$\gcd(26, 21) = \gcd(21, 5) = \gcd(5, 1) = 1.$$

Since 3 is a multiple of 1, so we can't *rule out* the possibility that 3 gallons can be formed. On the other hand, we don't know if it can be done either. To resolve the matter, we will need more number theory.

4.2.4 One Solution for All Water Jug Problems

Corollary 4.2.2 says that 3 can be written as a linear combination of 21 and 26, since 3 is a multiple of $\gcd(21, 26) = 1$. In other words, there exist integers s and t such that:

$$3 = s \cdot 21 + t \cdot 26$$

We don't know what the coefficients s and t are, but we do know that they exist.

Now the coefficient s could be either positive or negative. However, we can readily transform this linear combination into an equivalent linear combination

$$3 = s' \cdot 21 + t' \cdot 26 \tag{4.4}$$

where the coefficient s' is positive. The trick is to notice that if we increase s by 26 in the original equation and decrease t by 21, then the value of the expression $s \cdot 21 + t \cdot 26$ is unchanged overall. Thus, by repeatedly increasing the value of s (by 26 at a time) and decreasing the value of t (by 21 at a time), we get a linear combination $s' \cdot 21 + t' \cdot 26 = 3$ where the coefficient s' is positive. Notice that then t' must be negative; otherwise, this expression would be much greater than 3.

Now we can form 3 gallons using jugs with capacities 21 and 26: We simply repeat the following steps s' times:

1. Fill the 21-gallon jug.
2. Pour all the water in the 21-gallon jug into the 26-gallon jug. If at any time the 26-gallon jug becomes full, empty it out, and continue pouring the 21-gallon jug into the 26-gallon jug.

At the end of this process, we must have have emptied the 26-gallon jug exactly $|t'|$ times. Here's why: we've taken $s' \cdot 21$ gallons of water from the fountain, and we've poured out some multiple of 26 gallons. If we emptied fewer than $|t'|$ times, then by (4.4), the big jug would be left with at least $3 + 26$ gallons, which is more than it can hold; if we emptied it more times, the big jug would be left containing at most $3 - 26$ gallons, which is nonsense. But once we have emptied the 26-gallon jug exactly $|t'|$ times, equation (4.4) implies that there are exactly 3 gallons left.

Remarkably, we don't even need to know the coefficients s' and t' in order to use this strategy! Instead of repeating the outer loop s' times, we could just repeat *until we obtain 3 gallons*, since that must happen eventually. Of course, we have to keep track of the amounts in the two jugs so we know when we're done. Here's the

solution that approach gives:

$$
\begin{array}{lllllllll}
(0,0) & \xrightarrow{\text{fill 21}} & (21,0) & \xrightarrow{\text{pour 21 into 26}} & (0,21) & & & & \\
 & \xrightarrow{\text{fill 21}} & (21,21) & \xrightarrow{\text{pour 21 into 26}} & (16,26) & \xrightarrow{\text{empty 26}} & (16,0) & \xrightarrow{\text{pour 21 into 26}} & (0,16) \\
 & \xrightarrow{\text{fill 21}} & (21,16) & \xrightarrow{\text{pour 21 into 26}} & (11,26) & \xrightarrow{\text{empty 26}} & (11,0) & \xrightarrow{\text{pour 21 into 26}} & (0,11) \\
 & \xrightarrow{\text{fill 21}} & (21,11) & \xrightarrow{\text{pour 21 into 26}} & (6,26) & \xrightarrow{\text{empty 26}} & (6,0) & \xrightarrow{\text{pour 21 into 26}} & (0,6) \\
 & \xrightarrow{\text{fill 21}} & (21,6) & \xrightarrow{\text{pour 21 into 26}} & (1,26) & \xrightarrow{\text{empty 26}} & (1,0) & \xrightarrow{\text{pour 21 into 26}} & (0,1) \\
 & \xrightarrow{\text{fill 21}} & (21,1) & \xrightarrow{\text{pour 21 into 26}} & (0,22) & & & & \\
 & \xrightarrow{\text{fill 21}} & (21,22) & \xrightarrow{\text{pour 21 into 26}} & (17,26) & \xrightarrow{\text{empty 26}} & (17,0) & \xrightarrow{\text{pour 21 into 26}} & (0,17) \\
 & \xrightarrow{\text{fill 21}} & (21,17) & \xrightarrow{\text{pour 21 into 26}} & (12,26) & \xrightarrow{\text{empty 26}} & (12,0) & \xrightarrow{\text{pour 21 into 26}} & (0,12) \\
 & \xrightarrow{\text{fill 21}} & (21,12) & \xrightarrow{\text{pour 21 into 26}} & (7,26) & \xrightarrow{\text{empty 26}} & (7,0) & \xrightarrow{\text{pour 21 into 26}} & (0,7) \\
 & \xrightarrow{\text{fill 21}} & (21,7) & \xrightarrow{\text{pour 21 into 26}} & (2,26) & \xrightarrow{\text{empty 26}} & (2,0) & \xrightarrow{\text{pour 21 into 26}} & (0,2) \\
 & \xrightarrow{\text{fill 21}} & (21,2) & \xrightarrow{\text{pour 21 into 26}} & (0,23) & & & & \\
 & \xrightarrow{\text{fill 21}} & (21,23) & \xrightarrow{\text{pour 21 into 26}} & (18,26) & \xrightarrow{\text{empty 26}} & (18,0) & \xrightarrow{\text{pour 21 into 26}} & (0,18) \\
 & \xrightarrow{\text{fill 21}} & (21,18) & \xrightarrow{\text{pour 21 into 26}} & (13,26) & \xrightarrow{\text{empty 26}} & (13,0) & \xrightarrow{\text{pour 21 into 26}} & (0,13) \\
 & \xrightarrow{\text{fill 21}} & (21,13) & \xrightarrow{\text{pour 21 into 26}} & (8,26) & \xrightarrow{\text{empty 26}} & (8,0) & \xrightarrow{\text{pour 21 into 26}} & (0,8) \\
 & \xrightarrow{\text{fill 21}} & (21,8) & \xrightarrow{\text{pour 21 into 26}} & (3,26) & \xrightarrow{\text{empty 26}} & (3,0) & \xrightarrow{\text{pour 21 into 26}} & (0,3)
\end{array}
$$

The same approach works regardless of the jug capacities and even regardless the amount we're trying to produce! Simply repeat these two steps until the desired amount of water is obtained:

1. Fill the smaller jug.
2. Pour all the water in the smaller jug into the larger jug. If at any time the larger jug becomes full, empty it out, and continue pouring the smaller jug into the larger jug.

By the same reasoning as before, this method eventually generates every multiple of the greatest common divisor of the jug capacities—all the quantities we can possibly produce. No ingenuity is needed at all!

4.2.5 The Pulverizer

We have shown that no matter which pair of numbers a and b we are given, there is always a pair of integer coefficients s and t such that

$$\gcd(a,b) = sa + tb.$$

Unfortunately, the proof was *nonconstructive*: it didn't suggest a way for finding such s and t. That job is tackled by a mathematical tool that dates to sixth-century India, where it was called *kuttak*, which means "The Pulverizer". Today, the Pulverizer is more commonly known as "the extended Euclidean GCD algorithm", because it is so close to Euclid's Algorithm.

Euclid's Algorithm for finding the GCD of two numbers relies on repeated application of the equation:

$$\gcd(a,b) = gcd(b, \text{rem}(a,b,)).$$

For example, we can compute the GCD of 259 and 70 as follows:

$$\begin{aligned}
\gcd(259,70) &= \gcd(70,49) && \text{since } \text{rem}(259,70) = 49\\
&= \gcd(49,21) && \text{since } \text{rem}(70,49) = 21\\
&= \gcd(21,7) && \text{since } \text{rem}(49,21) = 7\\
&= \gcd(7,0) && \text{since } \text{rem}(21,7) = 0\\
&= 7.
\end{aligned}$$

The Pulverizer goes through the same steps, but requires some extra bookkeeping along the way: as we compute $\gcd(a,b)$, we keep track of how to write each of the remainders (49, 21, and 7, in the example) as a linear combination of a and b (this is worthwhile, because our objective is to write the last nonzero remainder, which is the GCD, as such a linear combination). For our example, here is this extra bookkeeping:

x	y	$(\text{rem}(x,y))$	$=$	$x - q \cdot y$
259	70	49	$=$	$259 - 3 \cdot 70$
70	49	21	$=$	$70 - 1 \cdot 49$
			$=$	$70 - 1 \cdot (259 - 3 \cdot 70)$
			$=$	$-1 \cdot 259 + 4 \cdot 70$
49	21	7	$=$	$49 - 2 \cdot 21$
			$=$	$(259 - 3 \cdot 70) - 2 \cdot (-1 \cdot 259 + 4 \cdot 70)$
			$=$	$\boxed{3 \cdot 259 - 11 \cdot 70}$
21	7	0		

We began by initializing two variables, $x = a$ and $y = b$. In the first two columns above, we carried out Euclid's algorithm. At each step, we computed $\text{rem}(x,y)$, which can be written in the form $x - q \cdot y$. (Remember that the Division Algorithm says $x = q \cdot y + r$, where r is the remainder. We get $r = x - q \cdot y$ by rearranging terms.) Then we replaced x and y in this equation with equivalent linear combinations of a and b, which we already had computed. After simplifying, we were left

with a linear combination of a and b that was equal to the remainder as desired. The final solution is boxed.

You can prove that the Pulverizer always works and that it terminates by using induction. Indeed, you can "pulverize" very large numbers very quickly by using this algorithm. As we will soon see, its speed makes the Pulverizer a very useful tool in the field of cryptography.

4.3 The Fundamental Theorem of Arithmetic

We now have almost enough tools to prove something that you probably already know.

Theorem 4.3.1 (Fundamental Theorem of Arithmetic)**.** *Every positive integer n can be written in a unique way as a product of primes:*

$$n = p_1 \cdot p_2 \cdots p_j \qquad (p_1 \leq p_2 \leq \cdots \leq p_j)$$

Notice that the theorem would be false if 1 were considered a prime; for example, 15 could be written as $3 \cdot 5$ or $1 \cdot 3 \cdot 5$ or $1^2 \cdot 3 \cdot 5$. Also, we're relying on a standard convention: the product of an empty set of numbers is defined to be 1, much as the sum of an empty set of numbers is defined to be 0. Without this convention, the theorem would be false for $n = 1$.

There is a certain wonder in the Fundamental Theorem, even if you've known it since you were in a crib. Primes show up erratically in the sequence of integers. In fact, their distribution seems almost random:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, \ldots$$

Basic questions about this sequence have stumped humanity for centuries. And yet we know that every natural number can be built up from primes in *exactly one way*. These quirky numbers are the building blocks for the integers.

The Fundamental Theorem is not hard to prove, but we'll need a couple of preliminary facts.

Lemma 4.3.2. *If p is a prime and $p \mid ab$, then $p \mid a$ or $p \mid b$.*

Proof. The greatest common divisor of a and p must be either 1 or p, since these are the only positive divisors of p. If $\gcd(a, p) = p$, then the claim holds, because a is a multiple of p. Otherwise, $\gcd(a, p) = 1$ and so $p \mid b$ by part (4) of Lemma 4.2.4. ■

The Prime Number Theorem

Let $\pi(x)$ denote the number of primes less than or equal to x. For example, $\pi(10) = 4$ because 2, 3, 5, and 7 are the primes less than or equal to 10. Primes are very irregularly distributed, so the growth of π is similarly erratic. However, the Prime Number Theorem gives an approximate answer:

$$\lim_{x\to\infty} \frac{\pi(x)}{x/\ln x} = 1$$

Thus, primes gradually taper off. As a rule of thumb, about 1 integer out of every $\ln x$ in the vicinity of x is a prime.

The Prime Number Theorem was conjectured by Legendre in 1798 and proved a century later by de la Vallee Poussin and Hadamard in 1896. However, after his death, a notebook of Gauss was found to contain the same conjecture, which he apparently made in 1791 at age 15. (You sort of have to feel sorry for all the otherwise "great" mathematicians who had the misfortune of being contemporaries of Gauss.)

In late 2004 a billboard appeared in various locations around the country:

$$\left\{\begin{array}{l}\text{first 10-digit prime found}\\ \text{in consecutive digits of } e\end{array}\right\}\textbf{.com}$$

Substituting the correct number for the expression in curly-braces produced the URL for a Google employment page. The idea was that Google was interested in hiring the sort of people that could and would solve such a problem.

How hard is this problem? Would you have to look through thousands or millions or billions of digits of e to find a 10-digit prime? The rule of thumb derived from the Prime Number Theorem says that among 10-digit numbers, about 1 in

$$\ln 10^{10} \approx 23$$

is prime. This suggests that the problem isn't really so hard! Sure enough, the first 10-digit prime in consecutive digits of e appears quite early:

$$\begin{aligned} e = &2.7182818284590452353602874713526624977572470936999595749669\\ &6762772407663035354759457138217852516642\mathbf{7427466391}9320030\\ &5992181741359662904357290033429526059563073813232862794 34\ldots \end{aligned}$$

A routine induction argument extends this statement to:

Lemma 4.3.3. *Let p be a prime. If $p \mid a_1 a_2 \cdots a_n$, then p divides some a_i.*

Now we're ready to prove the Fundamental Theorem of Arithmetic.

Proof. Theorem 3.1.2 showed, using the Well Ordering Principle, that every positive integer can be expressed as a product of primes. So we just have to prove this expression is unique. We will use Well Ordering to prove this too.

The proof is by contradiction: assume, contrary to the claim, that there exist positive integers that can be written as products of primes in more than one way. By the Well Ordering Principle, there is a smallest integer with this property. Call this integer n, and let

$$\begin{aligned} n &= p_1 \cdot p_2 \cdots p_j \\ &= q_1 \cdot q_2 \cdots q_k \end{aligned}$$

be two of the (possibly many) ways to write n as a product of primes. Then $p_1 \mid n$ and so $p_1 \mid q_1 q_2 \cdots q_k$. Lemma 4.3.3 implies that p_1 divides one of the primes q_i. But since q_i is a prime, it must be that $p_1 = q_i$. Deleting p_1 from the first product and q_i from the second, we find that n/p_1 is a positive integer *smaller* than n that can also be written as a product of primes in two distinct ways. But this contradicts the definition of n as the smallest such positive integer. ■

4.4 Alan Turing

The man pictured in Figure 4.1 is Alan Turing, the most important figure in the history of computer science. For decades, his fascinating life story was shrouded by government secrecy, societal taboo, and even his own deceptions.

At age 24, Turing wrote a paper entitled *On Computable Numbers, with an Application to the Entscheidungsproblem*. The crux of the paper was an elegant way to model a computer in mathematical terms. This was a breakthrough, because it allowed the tools of mathematics to be brought to bear on questions of computation. For example, with his model in hand, Turing immediately proved that there exist problems that no computer can solve—no matter how ingenious the programmer. Turing's paper is all the more remarkable because he wrote it in 1936, a full decade before any electronic computer actually existed.

The word "Entscheidungsproblem" in the title refers to one of the 28 mathematical problems posed by David Hilbert in 1900 as challenges to mathematicians of

Photograph of Alan Turing removed due to copyright restrictions.
Please see: http://en.wikipedia.org/wiki/File:Alan_Turing_photo.jpg

the 20th century. Turing knocked that one off in the same paper. And perhaps you've heard of the "Church-Turing thesis"? Same paper. So Turing was obviously a brilliant guy who generated lots of amazing ideas. But this lecture is about one of Turing's less-amazing ideas. It involved codes. It involved number theory. And it was sort of stupid.

Let's look back to the fall of 1937. Nazi Germany was rearming under Adolf Hitler, world-shattering war looked imminent, and—like us—Alan Turing was pondering the usefulness of number theory. He foresaw that preserving military secrets would be vital in the coming conflict and proposed a way *to encrypt communications using number theory*. This is an idea that has ricocheted up to our own time. Today, number theory is the basis for numerous public-key cryptosystems, digital signature schemes, cryptographic hash functions, and electronic payment systems. Furthermore, military funding agencies are among the biggest investors in cryptographic research. Sorry Hardy!

Soon after devising his code, Turing disappeared from public view, and half a century would pass before the world learned the full story of where he'd gone and what he did there. We'll come back to Turing's life in a little while; for now, let's investigate the code Turing left behind. The details are uncertain, since he never formally published the idea, so we'll consider a couple of possibilities.

4.4.1 Turing's Code (Version 1.0)

The first challenge is to translate a text message into an integer so we can perform mathematical operations on it. This step is not intended to make a message harder to read, so the details are not too important. Here is one approach: replace each letter of the message with two digits ($A = 01$, $B = 02$, $C = 03$, etc.) and string all the digits together to form one huge number. For example, the message "victory" could be translated this way:

	"v	i	c	t	o	r	y"
$\rightarrow$	22	09	03	20	15	18	25

Turing's code requires the message to be a prime number, so we may need to pad the result with a few more digits to make a prime. In this case, appending the digits 13 gives the number 2209032015182513, which is prime.

Here is how the encryption process works. In the description below, m is the unencoded message (which we want to keep secret), m^* is the encrypted message (which the Nazis may intercept), and k is the key.

Beforehand The sender and receiver agree on a secret key, which is a large prime k.

Encryption The sender encrypts the message m by computing:

$$m^* = m \cdot k$$

Decryption The receiver decrypts m^* by computing:

$$\frac{m^*}{k} = \frac{m \cdot k}{k} = m$$

For example, suppose that the secret key is the prime number $k = 22801763489$ and the message m is "victory". Then the encrypted message is:

$$\begin{aligned} m^* &= m \cdot k \\ &= 2209032015182513 \cdot 22801763489 \\ &= 50369825549820718594667857 \end{aligned}$$

There are a couple of questions that one might naturally ask about Turing's code.

1. How can the sender and receiver ensure that m and k are prime numbers, as required?

The general problem of determining whether a large number is prime or composite has been studied for centuries, and reasonably good primality tests were known even in Turing's time. In 2002, Manindra Agrawal, Neeraj Kayal, and Nitin Saxena announced a primality test that is guaranteed to work on a number n in about $(\log n)^{12}$ steps, that is, a number of steps bounded by a twelfth degree polynomial in the length (in bits) of the input, n. This definitively places primality testing way below the problems of exponential difficulty. Amazingly, the description of their breakthrough algorithm was only thirteen lines long!

Of course, a twelfth degree polynomial grows pretty fast, so the Agrawal, *et al.* procedure is of no practical use. Still, good ideas have a way of breeding more good ideas, so there's certainly hope that further improvements will lead to a procedure that is useful in practice. But the truth is, there's no practical need to improve it, since very efficient *probabilistic* procedures for prime-testing have been known since the early 1970's. These procedures have some probability of giving a wrong answer, but their probability of being wrong is so tiny that relying on their answers is the best bet you'll ever make.

2. Is Turing's code secure?

   The Nazis see only the encrypted message $m^* = m \cdot k$, so recovering the original message m requires factoring m^*. Despite immense efforts, no really efficient factoring algorithm has ever been found. It appears to be a fundamentally difficult problem, though a breakthrough someday is not impossible. In effect, Turing's code puts to practical use his discovery that there are limits to the power of computation. Thus, provided m and k are sufficiently large, the Nazis seem to be out of luck!

This all sounds promising, but there is a major flaw in Turing's code.

4.4.2 Breaking Turing's Code

Let's consider what happens when the sender transmits a *second* message using Turing's code and the same key. This gives the Nazis two encrypted messages to look at:

$$m_1^* = m_1 \cdot k \qquad \text{and} \qquad m_2^* = m_2 \cdot k$$

The greatest common divisor of the two encrypted messages, m_1^* and m_2^*, is the secret key k. And, as we've seen, the GCD of two numbers can be computed very efficiently. So after the second message is sent, the Nazis can recover the secret key and read *every* message!

It is difficult to believe a mathematician as brilliant as Turing could overlook such a glaring problem. One possible explanation is that he had a slightly different system in mind, one based on *modular* arithmetic.

4.5 Modular Arithmetic

On page 1 of his masterpiece on number theory, *Disquisitiones Arithmeticae*, Gauss introduced the notion of "congruence". Now, Gauss is another guy who managed to cough up a half-decent idea every now and then, so let's take a look at this one. Gauss said that a is *congruent* to b *modulo* n iff $n \mid (a - b)$. This is written

$$a \equiv b \pmod{n}.$$

For example:

$$29 \equiv 15 \pmod{7} \quad \text{because } 7 \mid (29 - 15).$$

There is a close connection between congruences and remainders:

Lemma 4.5.1 (Congruences and Remainders)**.**

$$a \equiv b \pmod{n} \quad \textit{iff} \quad \text{rem}(a, n) = \text{rem}(b, n).$$

Proof. By the Division Theorem, there exist unique pairs of integers q_1, r_1 and q_2, r_2 such that:

$$\begin{aligned} a &= q_1 n + r_1 && \text{where } 0 \le r_1 < n, \\ b &= q_2 n + r_2 && \text{where } 0 \le r_2 < n. \end{aligned}$$

Subtracting the second equation from the first gives:

$$a - b = (q_1 - q_2)n + (r_1 - r_2) \qquad \text{where } -n < r_1 - r_2 < n.$$

Now $a \equiv b \pmod{n}$ if and only if n divides the left side. This is true if and only if n divides the right side, which holds if and only if $r_1 - r_2$ is a multiple of n. Given the bounds on $r_1 - r_2$, this happens precisely when $r_1 = r_2$, that is, when $\text{rem}(a, n) = \text{rem}(b, n)$. ■

So we can also see that

$$29 \equiv 15 \pmod{7} \quad \text{because } \text{rem}(29, 7) = 1 = \text{rem}(15, 7).$$

This formulation explains why the congruence relation has properties like an equality relation. Notice that even though (mod 7) appears over on the right side, the $\equiv$ symbol, it isn't any more strongly associated with the 15 than with the 29. It would really be clearer to write $29 \equiv_{\text{mod } 7} 15$ for example, but the notation with the modulus at the end is firmly entrenched and we'll stick to it.

We'll make frequent use of the following immediate Corollary of Lemma 4.5.1:

Corollary 4.5.2.

$$a \equiv \text{rem}(a, n) \pmod{n}$$

Still another way to think about congruence modulo n is that it *defines a partition of the integers into n sets so that congruent numbers are all in the same set*. For example, suppose that we're working modulo 3. Then we can partition the integers into 3 sets as follows:

$$\begin{array}{llllllll} \{ & \ldots, & -6, & -3, & 0, & 3, & 6, & 9, & \ldots & \} \\ \{ & \ldots, & -5, & -2, & 1, & 4, & 7, & 10, & \ldots & \} \\ \{ & \ldots, & -4, & -1, & 2, & 5, & 8, & 11, & \ldots & \} \end{array}$$

according to whether their remainders on division by 3 are 0, 1, or 2. The upshot is that when arithmetic is done modulo n there are really only n different kinds of numbers to worry about, because there are only n possible remainders. In this sense, modular arithmetic is a simplification of ordinary arithmetic and thus is a good reasoning tool.

There are many useful facts about congruences, some of which are listed in the lemma below. The overall theme is that *congruences work a lot like equations*, though there are a couple of exceptions.

Lemma 4.5.3 (Facts About Congruences)**.** *The following hold for $n \geq 1$:*

1. $a \equiv a \pmod{n}$
2. $a \equiv b \pmod{n}$ *implies* $b \equiv a \pmod{n}$
3. $a \equiv b \pmod{n}$ *and* $b \equiv c \pmod{n}$ *implies* $a \equiv c \pmod{n}$
4. $a \equiv b \pmod{n}$ *implies* $a + c \equiv b + c \pmod{n}$
5. $a \equiv b \pmod{n}$ *implies* $ac \equiv bc \pmod{n}$
6. $a \equiv b \pmod{n}$ *and* $c \equiv d \pmod{n}$ *imply* $a + c \equiv b + d \pmod{n}$
7. $a \equiv b \pmod{n}$ *and* $c \equiv d \pmod{n}$ *imply* $ac \equiv bd \pmod{n}$

Proof. Parts 1–3. follow immediately from Lemma 4.5.1. Part 4. follows immediately from the definition that $a \equiv b \pmod{n}$ iff $n \mid (a-b)$. Likewise, part 5. follows because if $n \mid (a-b)$ then it divides $(a-b)c = ac - bc$. To prove part 6., assume

$$a \equiv b \pmod{n} \tag{4.5}$$

and

$$c \equiv d \pmod{n}. \tag{4.6}$$

Then

$$\begin{aligned}
a + c &\equiv b + c \pmod{n} && \text{(by part 4. and (4.5)),}\\
c + b &\equiv d + b \pmod{n} && \text{(by part 4. and (4.6)), so}\\
b + c &\equiv b + d \pmod{n} && \text{and therefore}\\
a + c &\equiv b + d \pmod{n} && \text{(by part 3.)}
\end{aligned}$$

Part 7 has a similar proof. ■

4.5.1 Turing's Code (Version 2.0)

In 1940, France had fallen before Hitler's army, and Britain stood alone against the Nazis in western Europe. British resistance depended on a steady flow of supplies brought across the north Atlantic from the United States by convoys of ships. These convoys were engaged in a cat-and-mouse game with German "U-boats"—submarines—which prowled the Atlantic, trying to sink supply ships and starve Britain into submission. The outcome of this struggle pivoted on a balance of information: could the Germans locate convoys better than the Allies could locate U-boats or vice versa?

Germany lost.

But a critical reason behind Germany's loss was made public only in 1974: Germany's naval code, *Enigma*, had been broken by the Polish Cipher Bureau (see http://en.wikipedia.org/wiki/Polish_Cipher_Bureau) and the secret had been turned over to the British a few weeks before the Nazi invasion of Poland in 1939. Throughout much of the war, the Allies were able to route convoys around German submarines by listening in to German communications. The British government didn't explain *how* Enigma was broken until 1996. When it was finally released (by the US), the story revealed that Alan Turing had joined the secret British codebreaking effort at Bletchley Park in 1939, where he became the lead developer of methods for rapid, bulk decryption of German Enigma messages. Turing's Enigma deciphering was an invaluable contribution to the Allied victory over Hitler.

Governments are always tight-lipped about cryptography, but the half-century of official silence about Turing's role in breaking Enigma and saving Britain may be related to some disturbing events after the war. More on that later. Let's get back to number theory and consider an alternative interpretation of Turing's code. Perhaps we had the basic idea right (multiply the message by the key), but erred in using *conventional* arithmetic instead of *modular* arithmetic. Maybe this is what Turing meant:

Beforehand The sender and receiver agree on a large prime p, which may be made public. (This will be the modulus for all our arithmetic.) They also agree on a secret key $k \in \{1, 2, \ldots, p-1\}$.

Encryption The message m can be any integer in the set $\{0, 1, 2, \ldots, p-1\}$; in particular, the message is no longer required to be a prime. The sender encrypts the message m to produce m^* by computing:

$$m^* = \text{rem}(mk, p) \tag{4.7}$$

Decryption (Uh-oh.)

The decryption step is a problem. We might hope to decrypt in the same way as before: by dividing the encrypted message m^* by the key k. The difficulty is that m^* is the *remainder* when mk is divided by p. So dividing m^* by k might not even give us an integer!

This decoding difficulty can be overcome with a better understanding of arithmetic modulo a prime.

4.6 Arithmetic with a Prime Modulus

4.6.1 Multiplicative Inverses

The *multiplicative inverse* of a number x is another number x^{-1} such that:

$$x \cdot x^{-1} = 1$$

Generally, multiplicative inverses exist over the real numbers. For example, the multiplicative inverse of 3 is $1/3$ since:

$$3 \cdot \frac{1}{3} = 1$$

The sole exception is that 0 does not have an inverse.

On the other hand, inverses generally do not exist over the integers. For example, 7 can not be multiplied by another integer to give 1.

Surprisingly, multiplicative inverses do exist when we're working *modulo a prime number*. For example, if we're working modulo 5, then 3 is a multiplicative inverse of 7, since:

$$7 \cdot 3 \equiv 1 \pmod 5$$

(All numbers congruent to 3 modulo 5 are also multiplicative inverses of 7; for example, $7 \cdot 8 \equiv 1 \pmod 5$ as well.) The only exception is that numbers congruent to 0 modulo 5 (that is, the multiples of 5) do not have inverses, much as 0 does not have an inverse over the real numbers. Let's prove this.

Lemma 4.6.1. *If p is prime and k is not a multiple of p, then k has a multiplicative inverse modulo p.*

Proof. Since p is prime, it has only two divisors: 1 and p. And since k is not a multiple of p, we must have $\gcd(p, k) = 1$. Therefore, there is a linear combination of p and k equal to 1:

$$sp + tk = 1$$

Rearranging terms gives:

$$sp = 1 - tk$$

This implies that $p \mid (1 - tk)$ by the definition of divisibility, and therefore $tk \equiv 1 \pmod p$ by the definition of congruence. Thus, t is a multiplicative inverse of k. ■

Multiplicative inverses are the key to decryption in Turing's code. Specifically, we can recover the original message by multiplying the encoded message by the *inverse* of the key:

$$\begin{aligned} m^* \cdot k^{-1} &= \operatorname{rem}(mk, p) \cdot k^{-1} && \text{(the def. (4.7) of } m^*) \\ &\equiv (mk)k^{-1} \pmod p && \text{(by Cor. 4.5.2)} \\ &\equiv m \pmod p. \end{aligned}$$

This shows that m^*k^{-1} is congruent to the original message m. Since m was in the range $0, 1, \ldots, p - 1$, we can recover it exactly by taking a remainder:

$$m = \operatorname{rem}(m^*k^{-1}, p).$$

So all we need to decrypt the message is to find a value of k^{-1}. From the proof of Lemma 4.6.1, we know that t is such a value, where $sp + tk = 1$. Finding t is easy using the Pulverizer.

4.6.2 Cancellation

Another sense in which real numbers are nice is that one can cancel multiplicative terms. In other words, if we know that $m_1 k = m_2 k$, then we can cancel the k's and conclude that $m_1 = m_2$, provided $k \neq 0$. In general, cancellation is *not* valid in modular arithmetic. For example,

$$2 \cdot 3 \equiv 4 \cdot 3 \pmod 6,$$

but canceling the 3's leads to the *false* conclusion that $2 \equiv 4 \pmod 6$. The fact that multiplicative terms can not be canceled is the most significant sense in which congruences differ from ordinary equations. However, this difference goes away if we're working modulo a *prime*; then cancellation is valid.

Lemma 4.6.2. *Suppose p is a prime and k is not a multiple of p. Then*

$$ak \equiv bk \pmod p \quad \text{IMPLIES} \quad a \equiv b \pmod p.$$

Proof. Multiply both sides of the congruence by k^{-1}. ∎

We can use this lemma to get a bit more insight into how Turing's code works. In particular, the encryption operation in Turing's code *permutes the set of possible messages*. This is stated more precisely in the following corollary.

Corollary 4.6.3. *Suppose p is a prime and k is not a multiple of p. Then the sequence:*

$$\text{rem}((1 \cdot k), p), \quad \text{rem}((2 \cdot k), p), \quad \ldots, \quad \text{rem}(((p-1) \cdot k), p)$$

is a permutation[4] *of the sequence:*

$$1, \quad 2, \quad \ldots, \quad (p-1).$$

Proof. The sequence of remainders contains $p - 1$ numbers. Since $i \cdot k$ is not divisible by p for $i = 1, \ldots p - 1$, all these remainders are in the range 1 to $p - 1$ by the definition of remainder. Furthermore, the remainders are all different: no two numbers in the range 1 to $p - 1$ are congruent modulo p, and by Lemma 4.6.2, $i \cdot k \equiv j \cdot k \pmod p$ if and only if $i \equiv j \pmod p$. Thus, the sequence of remainders must contain *all* of the numbers from 1 to $p - 1$ in some order. ∎

[4] A *permutation* of a sequence of elements is a reordering of the elements.

For example, suppose $p = 5$ and $k = 3$. Then the sequence:

$$\underbrace{\text{rem}((1\cdot 3),5)}_{=3},\quad \underbrace{\text{rem}((2\cdot 3),5)}_{=1},\quad \underbrace{\text{rem}((3\cdot 3),5)}_{=4},\quad \underbrace{\text{rem}((4\cdot 3),5)}_{=2}$$

is a permutation of 1, 2, 3, 4. As long as the Nazis don't know the secret key k, they don't know how the set of possible messages are permuted by the process of encryption and thus they can't read encoded messages.

4.6.3 Fermat's Little Theorem

An alternative approach to finding the inverse of the secret key k in Turing's code (about equally efficient and probably more memorable) is to rely on Fermat's Little Theorem, which is much easier than his famous Last Theorem.

Theorem 4.6.4 (Fermat's Little Theorem)**.** *Suppose p is a prime and k is not a multiple of p. Then:*

$$k^{p-1} \equiv 1 \pmod{p}$$

Proof. We reason as follows:

$$\begin{aligned}
(p-1)! &::= 1\cdot 2\cdots(p-1)\\
&= \text{rem}(k,p)\cdot\text{rem}(2k,p)\cdots\text{rem}((p-1)k,p) && \text{(by Cor 4.6.3)}\\
&\equiv k\cdot 2k\cdots(p-1)k \pmod{p} && \text{(by Cor 4.5.2)}\\
&\equiv (p-1)!\cdot k^{p-1} \pmod{p} && \text{(rearranging terms)}
\end{aligned}$$

Now $(p-1)!$ is not a multiple of p because the prime factorizations of $1, 2, \ldots, (p-1)$ contain only primes smaller than p. So by Lemma 4.6.2, we can cancel $(p-1)!$ from the first and last expressions, which proves the claim. ■

Here is how we can find inverses using Fermat's Theorem. Suppose p is a prime and k is not a multiple of p. Then, by Fermat's Theorem, we know that:

$$k^{p-2}\cdot k \equiv 1 \pmod{p}$$

Therefore, k^{p-2} must be a multiplicative inverse of k. For example, suppose that we want the multiplicative inverse of 6 modulo 17. Then we need to compute $\text{rem}(6^{15}, 17)$, which we can do by successive squaring. All the congruences below

hold modulo 17.

$$\begin{aligned} 6^2 &\equiv 36 \equiv 2 \\ 6^4 &\equiv (6^2)^2 \equiv 2^2 \equiv 4 \\ 6^8 &\equiv (6^4)^2 \equiv 4^2 \equiv 16 \\ 6^{15} &\equiv 6^8 \cdot 6^4 \cdot 6^2 \cdot 6 \equiv 16 \cdot 4 \cdot 2 \cdot 6 \equiv 3 \end{aligned}$$

Therefore, $\text{rem}(6^{15}, 17) = 3$. Sure enough, 3 is the multiplicative inverse of 6 modulo 17, since:

$$3 \cdot 6 \equiv 1 \pmod{17}$$

In general, if we were working modulo a prime p, finding a multiplicative inverse by trying every value between 1 and $p-1$ would require about p operations. However, the approach above requires only about $2 \log p$ operations, which is far better when p is large.

4.6.4 Breaking Turing's Code—Again

The Germans didn't bother to encrypt their weather reports with the highly-secure Enigma system. After all, so what if the Allies learned that there was rain off the south coast of Iceland? But, amazingly, this practice provided the British with a critical edge in the Atlantic naval battle during 1941.

The problem was that some of those weather reports had originally been transmitted using Enigma from U-boats out in the Atlantic. Thus, the British obtained both unencrypted reports and the same reports encrypted with Enigma. By comparing the two, the British were able to determine which key the Germans were using that day and could read all other Enigma-encoded traffic. Today, this would be called a *known-plaintext attack*.

Let's see how a known-plaintext attack would work against Turing's code. Suppose that the Nazis know both m and m^* where:

$$m^* \equiv mk \pmod{p}$$

Now they can compute:

$$\begin{aligned} m^{p-2} \cdot m^* &= m^{p-2} \cdot \text{rem}(mk, p) && \text{(def. (4.7) of } m^*) \\ &\equiv m^{p-2} \cdot mk \pmod{p} && \text{(by Cor 4.5.2)} \\ &\equiv m^{p-1} \cdot k \pmod{p} \\ &\equiv k \pmod{p} && \text{(Fermat's Theorem)} \end{aligned}$$

Now the Nazis have the secret key k and can decrypt any message!

This is a huge vulnerability, so Turing's code has no practical value. Fortunately, Turing got better at cryptography after devising this code; his subsequent deciphering of Enigma messages surely saved thousands of lives, if not the whole of Britain.

4.6.5 Turing Postscript

A few years after the war, Turing's home was robbed. Detectives soon determined that a former homosexual lover of Turing's had conspired in the robbery. So they arrested him—that is, they arrested Alan Turing—because homosexuality was a British crime punishable by up to two years in prison at that time. Turing was sentenced to a hormonal "treatment" for his homosexuality: he was given estrogen injections. He began to develop breasts.

Three years later, Alan Turing, the founder of computer science, was dead. His mother explained what happened in a biography of her own son. Despite her repeated warnings, Turing carried out chemistry experiments in his own home. Apparently, her worst fear was realized: by working with potassium cyanide while eating an apple, he poisoned himself.

However, Turing remained a puzzle to the very end. His mother was a devoutly religious woman who considered suicide a sin. And, other biographers have pointed out, Turing had previously discussed committing suicide by eating a poisoned apple. Evidently, Alan Turing, who founded computer science and saved his country, took his own life in the end, and in just such a way that his mother could believe it was an accident.

Turing's last project before he disappeared from public view in 1939 involved the construction of an elaborate mechanical device to test a mathematical conjecture called the Riemann Hypothesis. This conjecture first appeared in a sketchy paper by Bernhard Riemann in 1859 and is now one of the most famous unsolved problem in mathematics.

4.7 Arithmetic with an Arbitrary Modulus

Turing's code did not work as he hoped. However, his essential idea—using number theory as the basis for cryptography—succeeded spectacularly in the decades after his death.

In 1977, Ronald Rivest, Adi Shamir, and Leonard Adleman at MIT proposed a highly secure cryptosystem (called **RSA**) based on number theory. Despite decades of attack, no significant weakness has been found. Moreover, RSA has a major advantage over traditional codes: the sender and receiver of an encrypted mes-

The Riemann Hypothesis

The formula for the sum of an infinite geometric series says:

$$1 + x + x^2 + x^3 + \cdots = \frac{1}{1-x}$$

Substituting $x = \frac{1}{2^s}$, $x = \frac{1}{3^s}$, $x = \frac{1}{5^s}$, and so on for each prime number gives a sequence of equations:

$$1 + \frac{1}{2^s} + \frac{1}{2^{2s}} + \frac{1}{2^{3s}} + \cdots = \frac{1}{1-1/2^s}$$
$$1 + \frac{1}{3^s} + \frac{1}{3^{2s}} + \frac{1}{3^{3s}} + \cdots = \frac{1}{1-1/3^s}$$
$$1 + \frac{1}{5^s} + \frac{1}{5^{2s}} + \frac{1}{5^{3s}} + \cdots = \frac{1}{1-1/5^s}$$
$$\text{etc.}$$

Multiplying together all the left sides and all the right sides gives:

$$\sum_{n=1}^{\infty} \frac{1}{n^s} = \prod_{p \in \text{primes}} \left(\frac{1}{1-1/p^s} \right)$$

The sum on the left is obtained by multiplying out all the infinite series and applying the Fundamental Theorem of Arithmetic. For example, the term $1/300^s$ in the sum is obtained by multiplying $1/2^{2s}$ from the first equation by $1/3^s$ in the second and $1/5^{2s}$ in the third. Riemann noted that every prime appears in the expression on the right. So he proposed to learn about the primes by studying the equivalent, but simpler expression on the left. In particular, he regarded s as a complex number and the left side as a function, $\zeta(s)$. Riemann found that the distribution of primes is related to values of s for which $\zeta(s) = 0$, which led to his famous conjecture:

Definition 4.6.5. *The Riemann Hypothesis*: Every nontrivial zero of the zeta function $\zeta(s)$ lies on the line $s = 1/2 + ci$ in the complex plane.

A proof would immediately imply, among other things, a strong form of the Prime Number Theorem.
Researchers continue to work intensely to settle this conjecture, as they have for over a century. It is another of the Millennium Problems whose solver will earn $1,000,000 from the Clay Institute.

sage need not meet beforehand to agree on a secret key. Rather, the receiver has both a *secret key*, which she guards closely, and a *public key*, which she distributes as widely as possible. The sender then encrypts his message using her widely-distributed public key. Then she decrypts the received message using her closely-held private key. The use of such a *public key cryptography* system allows you and Amazon, for example, to engage in a secure transaction without meeting up beforehand in a dark alley to exchange a key.

Interestingly, RSA does not operate modulo a prime, as Turing's scheme may have, but rather modulo the product of *two* large primes. Thus, we'll need to know a bit about how arithmetic works modulo a composite number in order to understand RSA. Arithmetic modulo an arbitrary positive integer is really only a little more painful than working modulo a prime—though you may think this is like the doctor saying, "This is only going to hurt a little," before he jams a big needle in your arm.

4.7.1 Relative Primality

First, we need a new definition. Integers a and b are *relatively prime* iff $\gcd(a, b) = 1$. For example, 8 and 15 are relatively prime, since $\gcd(8, 15) = 1$. Note that, except for multiples of p, every integer is relatively prime to a prime number p.

Next we'll need to generalize what we know about arithmetic modulo a prime to work modulo an arbitrary positive integer n. The basic theme is that arithmetic modulo n may be complicated, but the integers *relatively prime* to n remain fairly well-behaved. For example, the proof of Lemma 4.6.1 of an inverse for k modulo p extends to an inverse for k relatively prime to n:

Lemma 4.7.1. *Let n be a positive integer. If k is* relatively prime *to n, then there exists an integer k^{-1} such that:*

$$k \cdot k^{-1} \equiv 1 \pmod{n}$$

As a consequence of this lemma, we can cancel a multiplicative term from both sides of a congruence if that term is relatively prime to the modulus:

Corollary 4.7.2. *Suppose n is a positive integer and k is relatively prime to n. If*

$$ak \equiv bk \pmod{n}$$

then

$$a \equiv b \pmod{n}$$

This holds because we can multiply both sides of the first congruence by k^{-1} and simplify to obtain the second.

The following lemma is the natural generalization of Corollary 4.6.3.

Lemma 4.7.3. *Suppose n is a positive integer and k is relatively prime to n. Let $k_1, \ldots, k_r$ denote all the integers relatively prime to n in the range 1 to $n-1$. Then the sequence:*

$$\text{rem}(k_1 \cdot k, n), \quad \text{rem}(k_2 \cdot k, n), \quad \text{rem}(k_3 \cdot k, n), \quad \ldots \quad , \text{rem}(k_r \cdot k, n)$$

is a permutation of the sequence:

$$k_1, \quad k_2, \quad \ldots \quad , k_r.$$

Proof. We will show that the remainders in the first sequence are all distinct and are equal to some member of the sequence of k_j's. Since the two sequences have the same length, the first must be a permutation of the second.

First, we show that the remainders in the first sequence are all distinct. Suppose that $\text{rem}(k_i k, n) = \text{rem}(k_j k, n)$. This is equivalent to $k_i k \equiv k_j k \pmod{n}$, which implies $k_i \equiv k_j \pmod{n}$ by Corollary 4.7.2. This, in turn, means that $k_i = k_j$ since both are between 1 and $n-1$. Thus, none of the remainder terms in the first sequence is equal to any other remainder term.

Next, we show that each remainder in the first sequence equals one of the k_i. By assumption, $\gcd(k_i, n) = 1$ and $\gcd(k, n) = 1$, which means that

$$\begin{aligned} \gcd(n, \text{rem}(k_i k, n)) &= \gcd(k_i k, n) && \text{(by part (5) of Lemma 4.2.4)} \\ &= 1 && \text{(by part (3) of Lemma 4.2.4).} \end{aligned}$$

Since $\text{rem}(k_i k, n)$ is in the range from 0 to $n-1$ by the definition of remainder, and since it is relatively prime to n, it must (by definition of the k_j's) be equal to some k_j. ■

4.7.2 Euler's Theorem

RSA relies heavily on a generalization of Fermat's Theorem known as Euler's Theorem. For both theorems, the exponent of k needed to produce an inverse of k modulo n depends on the number of integers in the set $\{1, 2, \ldots, n\}$ (denoted $[1, n]$) that are relatively prime to n. This value is known as *Euler's ϕ function* (a.k.a. *Euler's totient function*) and it is denoted as $\phi(n)$. For example, $\phi(7) = 6$ since 1, 2, 3, 4, 5, and 6 are all relatively prime to 7. Similarly, $\phi(12) = 4$ since 1, 5, 7, and 11 are the only numbers in $[1, 12]$ that are relatively prime to 12.[5]

If n is prime, then $\phi(n) = n - 1$ since every number less than a prime number is relatively prime to that prime. When n is composite, however, the ϕ function gets a little complicated. The following theorem characterizes the ϕ function for

[5] Recall that $\gcd(n, n) = n$ and so n is never relatively prime to itself.

composite n. We won't prove the theorem in its full generality, although we will give a proof for the special case when n is the product of two primes since that is the case that matters for RSA.

Theorem 4.7.4. *For any number n, if $p_1, p_2, \ldots, p_j$ are the (distinct) prime factors of n, then*

$$\phi(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\ldots\left(1 - \frac{1}{p_j}\right).$$

For example,

$$\begin{aligned}\phi(300) &= \phi(2^2 \cdot 3 \cdot 5^2)\\ &= 300\left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{5}\right)\\ &= 300\left(\frac{1}{2}\right)\left(\frac{2}{3}\right)\left(\frac{4}{5}\right)\\ &= 80.\end{aligned}$$

Corollary 4.7.5. *Let $n = pq$ where p and q are different primes. Then $\phi(n) = (p-1)(q-1)$.*

Corollary 4.7.5 follows easily from Theorem 4.7.4, but since Corollary 4.7.5 is important to RSA and we have not provided a proof of Theorem 4.7.4, we will give a direct proof of Corollary 4.7.5 in what follows.

Proof of Corollary 4.7.5. Since p and q are prime, any number that is not relatively prime to $n = pq$ must be a multiple of p or a multiple of q. Among the numbers $1, 2, \ldots, pq$, there are precisely q multiples of p and p multiples of q. Since p and q are relatively prime, the only number in $[1, pq]$ that is a multiple of both p and q is pq. Hence, there are $p + q - 1$ numbers in $[1, pq]$ that are *not* relatively prime to n. This means that

$$\begin{aligned}\phi(n) &= pq - p - q + 1\\ &= (p-1)(q-1),\end{aligned}$$

as claimed.[6] ■

We can now prove Euler's Theorem:

[6] This proof provides a brief preview of the kinds of counting arguments that we will explore more fully in Part III.

Theorem 4.7.6 (Euler's Theorem)**.** *Suppose n is a positive integer and k is relatively prime to n. Then*

$$k^{\phi(n)} \equiv 1 \pmod{n}$$

Proof. Let $k_1, \ldots, k_r$ denote all integers relatively prime to n such that $0 \leq k_i < n$. Then $r = \phi(n)$, by the definition of the function ϕ. The remainder of the proof mirrors the proof of Fermat's Theorem. In particular,

$$\begin{aligned}
&k_1 \cdot k_2 \cdots k_r \\
&\quad = \operatorname{rem}(k_1 \cdot k, n) \cdot \operatorname{rem}(k_2 \cdot k, n) \cdots \operatorname{rem}(k_r \cdot k, n) && \text{(by Lemma 4.7.3)} \\
&\quad \equiv (k_1 \cdot k) \cdot (k_2 \cdot k) \cdot \cdots \cdot (k_r \cdot k) \pmod{n} && \text{(by Cor 4.5.2)} \\
&\quad \equiv (k_1 \cdot k_2 \cdots k_r) \cdot k^r \pmod{n} && \text{(rearranging terms)}
\end{aligned}$$

Part (3) of Lemma 4.2.4. implies that $k_1 \cdot k_2 \cdots k_r$ is relatively prime to n. So by Corollary 4.7.2, we can cancel this product from the first and last expressions. This proves the claim. ■

We can find multiplicative inverses using Euler's theorem as we did with Fermat's theorem: if k is relatively prime to n, then $k^{\phi(n)-1}$ is a multiplicative inverse of k modulo n. However, this approach requires computing $\phi(n)$. Computing $\phi(n)$ is easy (using Theorem 4.7.4) if we know the prime factorization of n. Unfortunately, finding the factors of n can be hard to do when n is large and so the Pulverizer is often the best approach to computing inverses modulo n.

4.8 The RSA Algorithm

Finally, we are ready to see how the *RSA public key encryption scheme* works. The details are in the box on the next page.

It is not immediately clear from the description of the RSA cryptosystem that the decoding of the encrypted message is, in fact, the original unencrypted message. In order to check that this is the case, we need to show that the decryption $\operatorname{rem}((m')^d, n)$ is indeed equal to the sender's message m. Since $m' = \operatorname{rem}(m^e, n)$, m' is congruent to m^e modulo n by Corollary 4.5.2. That is,

$$m' \equiv m^e \pmod{n}.$$

By raising both sides to the power d, we obtain the congruence

$$(m')^d \equiv m^{ed} \pmod{n}. \tag{4.8}$$

The RSA Cryptosystem

Beforehand The receiver creates a public key and a secret key as follows.

1. Generate two distinct primes, p and q. Since they can be used to generate the secret key, they must be kept hidden.
2. Let $n = pq$.
3. Select an integer e such that $\gcd(e, (p-1)(q-1)) = 1$.
   The *public key* is the pair (e, n). This should be distributed widely.
4. Compute d such that $de \equiv 1 \pmod{(p-1)(q-1)}$. This can be done using the Pulverizer.
   The *secret key* is the pair (d, n). This should be kept hidden!

Encoding Given a message m, the sender first checks that $\gcd(m, n) = 1$.[a] The sender then encrypts message m to produce m' using the public key:

$$m' = \text{rem}(m^e, n).$$

Decoding The receiver decrypts message m' back to message m using the secret key:

$$m = \text{rem}((m')^d, n).$$

[a] It would be very bad if $\gcd(m, n)$ equals p or q since then it would be easy for someone to use the encoded message to compute the secret key If $\gcd(m, n) = n$, then the encoded message would be 0, which is fairly useless. For very large values of n, it is extremely unlikely that $\gcd(m, n) \neq 1$. If this does happen, you should get a new set of keys or, at the very least, add some bits to m so that the resulting message is relatively prime to n.

The encryption exponent e and the decryption exponent d are chosen such that $de \equiv 1 \pmod{(p-1)(q-1)}$. So, there exists an integer r such that $ed = 1 + r(p-1)(q-1)$. By substituting $1 + r(p-1)(q-1)$ for ed in Equation 4.8, we obtain

$$(m')^d \equiv m \cdot m^{r(p-1)(q-1)} \pmod{n}. \tag{4.9}$$

By Euler's Theorem and the assumption that $\gcd(m, n) = 1$, we know that

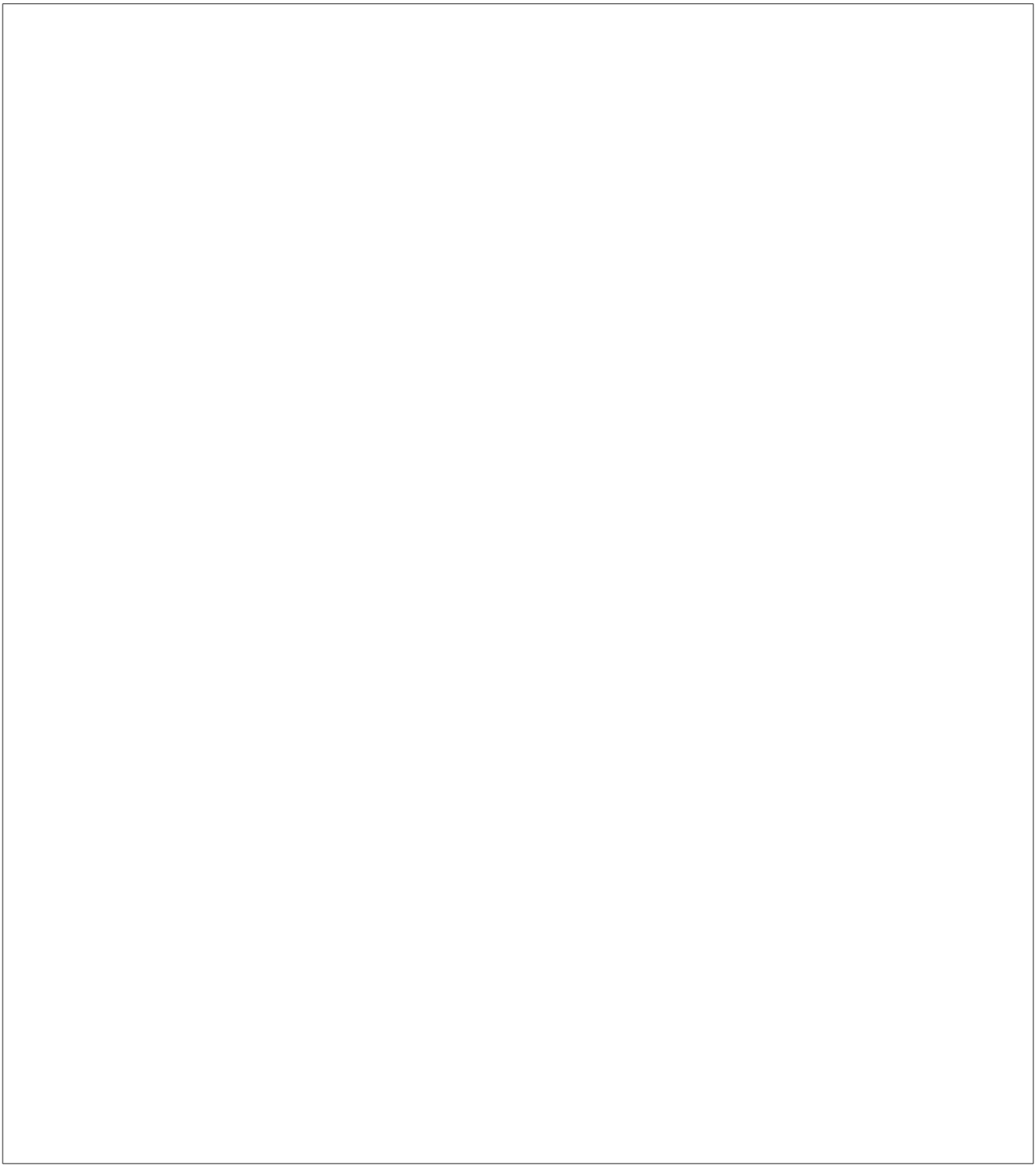
$$m^{\phi(n)} \equiv 1 \pmod{n}.$$

From Corollary 4.7.5, we know that $\phi(n) = (p-1)(q-1)$. Hence,

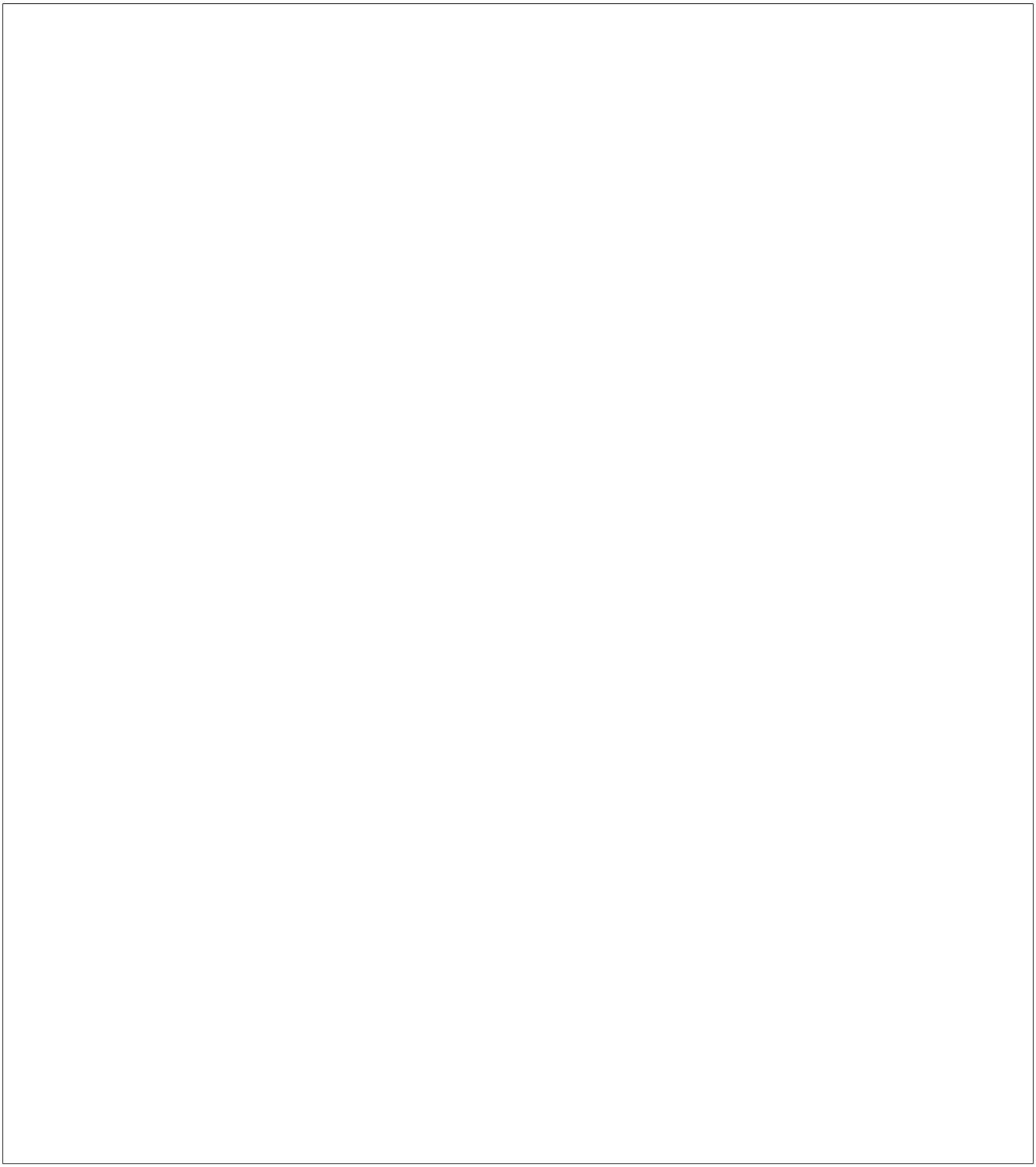
$$\begin{aligned}(m')^d &= m \cdot m^{r(p-1)(q-1)} \pmod{n} \\ &= m \cdot 1^r \pmod{n} \\ &= m \pmod{n}.\end{aligned}$$

Hence, the decryption process indeed reproduces the original message m.

Is it hard for someone without the secret key to decrypt the message? No one knows for sure but it is generally believed that if n is a very large number (say, with a thousand digits), then it is difficult to reverse engineer d from e and n. Of course, it is easy to compute d if you know p and q (by using the Pulverizer) but it is not known how to quickly factor n into p and q when n is very large. Maybe with a little more studying of number theory, you will be the first to figure out how to do it. Although, we should warn you that Gauss worked on it for years without a lot to show for his efforts. And if you do figure it out, you might wind up meeting some serious-looking fellows in black suits. . . .



II Structures



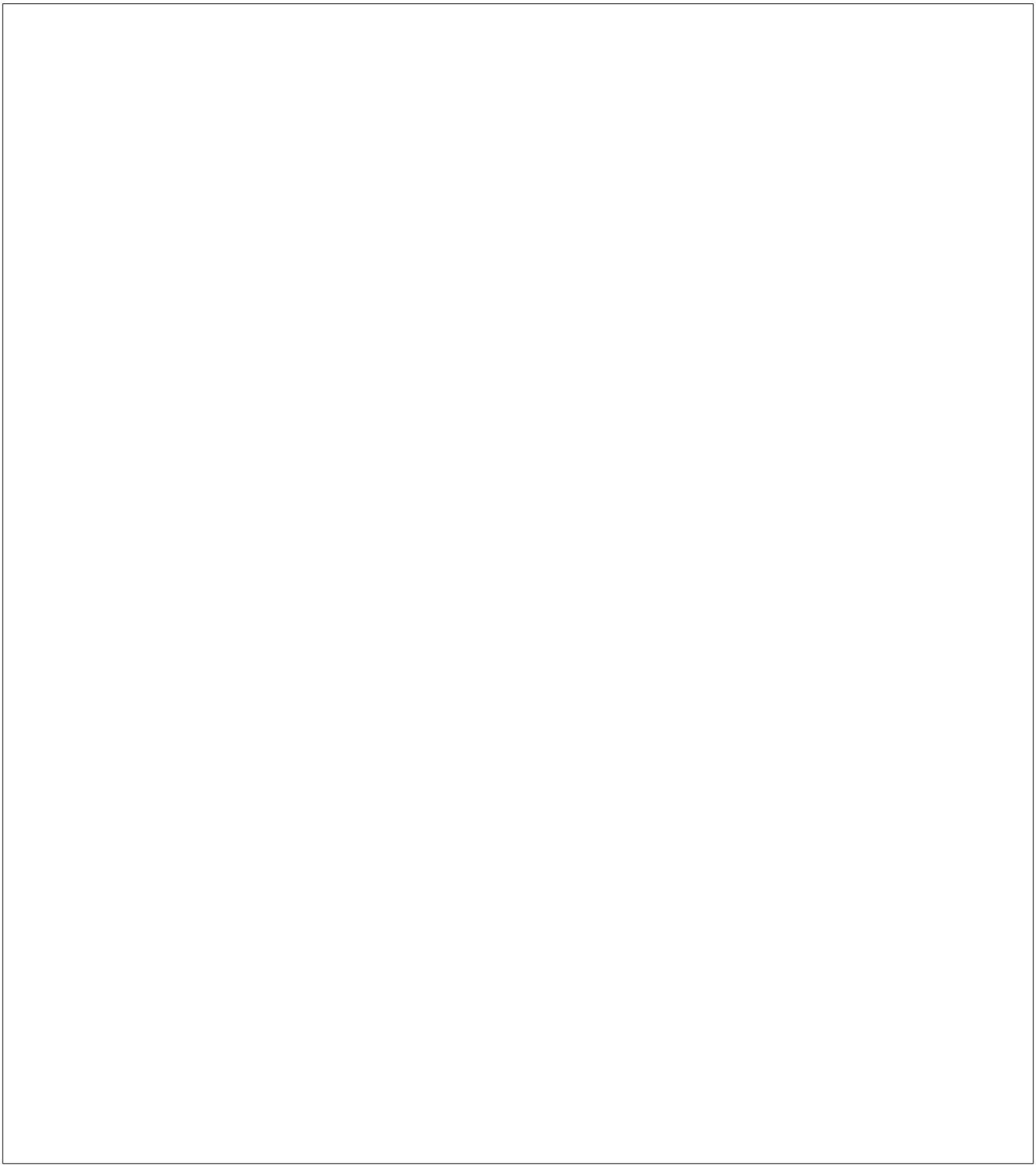
Introduction

Structure is fundamental in computer science. Whether you are writing code, solving an optimization problem, or designing a network, you will be dealing with structure. The better you can understand the structure, the better your results will be. And if you can reason about structure, then you will be in a good position to convince others (and yourself) that your results are worthy.

The most important structure in computer science is a *graph*, also known as a *network*). Graphs provide an excellent mechanism for modeling associations between pairs of objects; for example, two exams that cannot be given at the same time, two people that like each other, or two subroutines that can be run independently. In Chapter 5, we study graphs that represent *symmetric* relationships, like those just mentioned. In Chapter 6, we consider graphs where the relationship is *one-way*; that is, a situation where you can go from x to y but not necessarily vice-versa.

In Chapter 7, we consider the more general notion of a *relation* and we examine important classes of relations such as partially ordered sets. Partially ordered sets arise frequently in scheduling problems.

We conclude in Chapter 8 with a discussion of *state machines*. State machines can be used to model a variety of processes and are a fundamental tool in proving that an algorithm terminates and that it produces the correct output.



5 Graph Theory

Informally, a graph is a bunch of dots and lines where the lines connect some pairs of dots. An example is shown in Figure 5.1. The dots are called *nodes* (or *vertices*) and the lines are called *edges*.

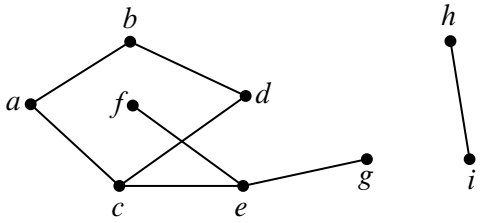


Figure 5.1 An example of a graph with 9 nodes and 8 edges.

Graphs are ubiquitous in computer science because they provide a handy way to represent a relationship between pairs of objects. The objects represent items of interest such as programs, people, cities, or web pages, and we place an edge between a pair of nodes if they are related in a certain way. For example, an edge between a pair of people might indicate that they like (or, in alternate scenarios, that they don't like) each other. An edge between a pair of courses might indicate that one needs to be taken before the other.

In this chapter, we will focus our attention on simple graphs where the relationship denoted by an edge is symmetric. Afterward, in Chapter 6, we consider the situation where the edge denotes a one-way relationship, for example, where one web page points to the other.[1]

5.1 Definitions

5.1.1 Simple Graphs

Definition 5.1.1. A *simple graph* G consists of a nonempty set V, called the *vertices* (aka *nodes*[2]) of G, and a set E of two-element subsets of V. The members of E are called the *edges* of G, and we write $G = (V, E)$.

[1] Two Stanford students analyzed such a graph to become multibillionaires. So, pay attention to graph theory, and who knows what might happen!

[2] We will use the terms vertex and node interchangeably.

The vertices correspond to the dots in Figure 5.1, and the edges correspond to the lines. The graph in Figure 5.1 is expressed mathematically as $G = (V, E)$, where:

$$V = \{a, b, c, d, e, f, g, h, i\}$$
$$E = \{\{a, b\}, \{a, c\}, \{b, d\}, \{c, d\}, \{c, e\}, \{e, f\}, \{e, g\}, \{h, i\}\}.$$

Note that $\{a, b\}$ and $\{b, a\}$ are different descriptions of the same edge, since sets are unordered. In this case, the graph $G = (V, E)$ has 9 nodes and 8 edges.

Definition 5.1.2. Two vertices in a simple graph are said to be *adjacent* if they are joined by an edge, and an edge is said to be *incident* to the vertices it joins. The number of edges incident to a vertex v is called the *degree* of the vertex and is denoted by $\deg(v)$; equivalently, the degree of a vertex is equals the number of vertices adjacent to it.

For example, in the simple graph shown in Figure 5.1, vertex a is adjacent to b and b is adjacent to d, and the edge $\{a, c\}$ is incident to vertices a and c. Vertex h has degree 1, d has degree 2, and $\deg(e) = 3$. It is possible for a vertex to have degree 0, in which case it is not adjacent to any other vertices. A simple graph does not need to have any edges at all —in which case, the degree of every vertex is zero and $|E| = 0$[3] —but it does need to have at least one vertex, that is, $|V| \geq 1$.

Note that simple graphs do *not* have any *self-loops* (that is, an edge of the form $\{a, a\}$) since an edge is defined to be a set of *two* vertices. In addition, there is at most one edge between any pair of vertices in a simple graph. In other words, a simple graph does not contain *multiedges* or *multiple edges*. That is because E is a set. Lastly, and most importantly, simple graphs do not contain *directed edges* (that is, edges of the form (a, b) instead of $\{a, b\}$).

There's no harm in relaxing these conditions, and some authors do, but we don't need self-loops, multiple edges between the same two vertices, or graphs with no vertices, and it's simpler not to have them around. We will consider graphs with directed edges (called *directed graphs* or *digraphs*) at length in Chapter 6. Since we'll only be considering simple graphs in this chapter, we'll just call them "graphs" from now on.

5.1.2 Some Common Graphs

Some graphs come up so frequently that they have names. The *complete graph* on n vertices, denoted K_n, has an edge between every two vertices, for a total of $n(n-1)/2$ edges. For example, K_5 is shown in Figure 5.2.

The *empty graph* has no edges at all. For example, the empty graph with 5 nodes is shown in Figure 5.3.

[3] The *cardinality*, $|E|$, of the set E is the number of elements in E.

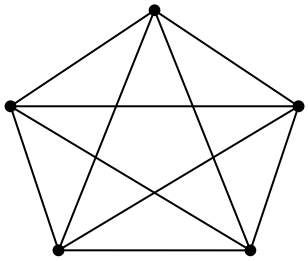


Figure 5.2 The complete graph on 5 nodes, K_5.

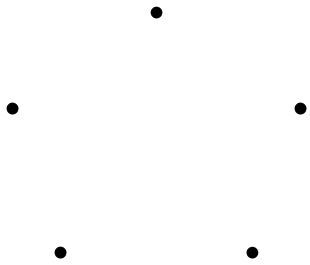


Figure 5.3 The empty graph with 5 nodes.

The n-node graph containing $n-1$ edges in sequence is known as the *line graph* L_n. More formally, $L_n = (V, E)$ where

$$V = \{v_1, v_2, \ldots, v_n\}$$

and

$$E = \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}\}$$

For example, L_5 is displayed in Figure 5.4.

If we add the edge $\{v_n, v_1\}$ to the line graph L_n, we get the graph C_n consisting of a simple cycle. For example, C_5 is illustrated in Figure 5.5.

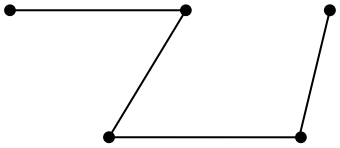


Figure 5.4 The 5-node line graph L_5.

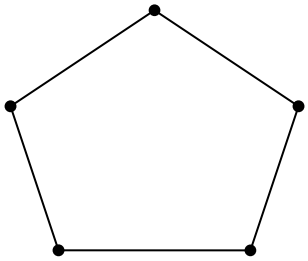


Figure 5.5 The 5-node cycle graph C_5.

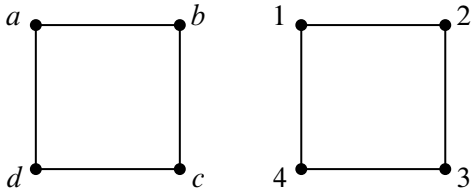


Figure 5.6 Two graphs that are isomorphic to C_4.

5.1.3 Isomorphism

Two graphs that look the same might actually be different in a formal sense. For example, the two graphs in Figure 5.6 are both simple cycles with 4 vertices, but one graph has vertex set $\{a, b, c, d\}$ while the other has vertex set $\{1, 2, 3, 4\}$. Strictly speaking, these graphs are different mathematical objects, but this is a frustrating distinction since the graphs *look the same*!

Fortunately, we can neatly capture the idea of "looks the same" through the notion of graph isomorphism.

Definition 5.1.3. If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are two graphs, then we say that G_1 is *isomorphic* to G_2 iff there exists a *bijection*[4] $f : V_1 \to V_2$ such that for every pair of vertices $u, v \in V_1$:

$$\{u, v\} \in E_1 \quad \text{iff} \quad \{f(u), f(v)\} \in E_2.$$

The function f is called an *isomorphism* between G_1 and G_2.

In other words, two graphs are isomorphic if they are the same up to a relabeling of their vertices. For example, here is an isomorphism between vertices in the two

[4] A bijection $f : V_1 \to V_2$ is a function that associates every node in V_1 with a unique node in V_2 and vice-versa. We will study bijections more deeply in Part III.

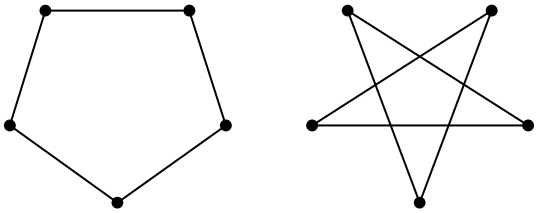


Figure 5.7 Two ways of drawing C_5.

graphs shown in Figure 5.6:

a corresponds to 1	b corresponds to 2
d corresponds to 4	c corresponds to 3.

You can check that there is an edge between two vertices in the graph on the left if and only if there is an edge between the two corresponding vertices in the graph on the right.

Two isomorphic graphs may be drawn very differently. For example, we have shown two different ways of drawing C_5 in Figure 5.7.

Isomorphism preserves the connection properties of a graph, abstracting out what the vertices are called, what they are made out of, or where they appear in a drawing of the graph. More precisely, a property of a graph is said to be *preserved under isomorphism* if whenever G has that property, every graph isomorphic to G also has that property. For example, isomorphic graphs must have the same number of vertices. What's more, if f is a graph isomorphism that maps a vertex, v, of one graph to the vertex, $f(v)$, of an isomorphic graph, then by definition of isomorphism, every vertex adjacent to v in the first graph will be mapped by f to a vertex adjacent to $f(v)$ in the isomorphic graph. This means that v and $f(v)$ will have the same degree. So if one graph has a vertex of degree 4 and another does not, then they can't be isomorphic. In fact, they can't be isomorphic if the number of degree 4 vertices in each of the graphs is not the same.

Looking for preserved properties can make it easy to determine that two graphs are not isomorphic, or to actually find an isomorphism between them if there is one. In practice, it's frequently easy to decide whether two graphs are isomorphic. However, no one has yet found a *general* procedure for determining whether two graphs are isomorphic that is *guaranteed* to run in polynomial time[5] in $|V|$.

Having such a procedure would be useful. For example, it would make it easy to search for a particular molecule in a database given the molecular bonds. On

[5] *I.e.*, in an amount of time that is upper-bounded by $|V|^c$ where c is a fixed number independent of $|V|$.

the other hand, knowing there is no such efficient procedure would also be valuable: secure protocols for encryption and remote authentication can be built on the hypothesis that graph isomorphism is computationally exhausting.

5.1.4 Subgraphs

Definition 5.1.4. A graph $G_1 = (V_1, E_1)$ is said to be a *subgraph* of a graph $G_2 = (V_2, E_2)$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$.

For example, the empty graph on n nodes is a subgraph of L_n, L_n is a subgraph of C_n, and C_n is a subgraph of K_n. Also, the graph $G = (V, E)$ where

$$V = \{g, h, i\} \quad \text{and} \quad E = \{\{h, i\}\}$$

is a subgraph of the graph in Figure 5.1. On the other hand, any graph containing an edge $\{g, h\}$ would not be a subgraph of the graph in Figure 5.1 because the graph in Figure 5.1 does not contain this edge.

Note that since a subgraph is itself a graph, the endpoints of any edge in a subgraph must also be in the subgraph. In other words if $G' = (V', E')$ is a subgraph of some graph G, and $\{v_i, v_j\} \in E'$, then it must be the case that $v_i \in V'$ and $v_j \in V'$.

5.1.5 Weighted Graphs

Sometimes, we will use edges to denote a connection between a pair of nodes where the connection has a *capacity* or *weight*. For example, we might be interested in the capacity of an Internet fiber between a pair of computers, the resistance of a wire between a pair of terminals, the tension of a spring connecting a pair of devices in a dynamical system, the tension of a bond between a pair of atoms in a molecule, or the distance of a highway between a pair of cities.

In such cases, it is useful to represent the system with an *edge-weighted* graph (aka a *weighted graph*). A weighted graph is the same as a simple graph except that we associate a real number (that is, the weight) with each edge in the graph. Mathematically speaking, a weighted graph consists of a graph $G = (V, E)$ and a weight function $w : E \to \mathbb{R}$. For example, Figure 5.8 shows a weighted graph where the weight of edge $\{a, b\}$ is 5.

5.1.6 Adjacency Matrices

There are many ways to represent a graph. We have already seen two ways: you can draw it, as in Figure 5.8 for example, or you can represent it with sets —as in $G = (V, E)$. Another common representation is with an adjacency matrix.

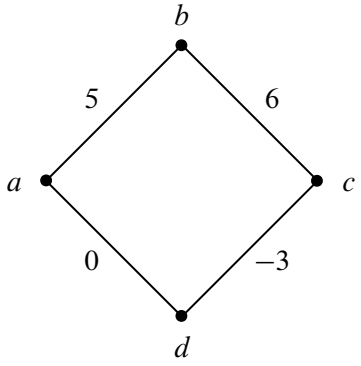


Figure 5.8 A 4-node weighted graph where the edge $\{a, b\}$ has weight 5.

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 5 & 0 & 0 \\ 5 & 0 & 6 & 0 \\ 0 & 6 & 0 & -3 \\ 0 & 0 & -3 & 0 \end{pmatrix}$$

(a) (b)

Figure 5.9 Examples of adjacency matrices. (a) shows the adjacency matrix for the graph in Figure 5.6(a) and (b) shows the adjacency matrix for the weighted graph in Figure 5.8. In each case, we set $v_1 = a$, $v_2 = b$, $v_3 = c$, and $v_4 = d$ to construct the matrix.

Definition 5.1.5. Given an n-node graph $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$, the *adjacency matrix* for G is the $n \times n$ matrix $A_G = \{a_{ij}\}$ where

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E \\ 0 & \text{otherwise.} \end{cases}$$

If G is a weighted graph with edge weights given by $w : E \to \mathbb{R}$, then the adjacency matrix for G is $A_G = \{a_{ij}\}$ where

$$a_{ij} = \begin{cases} w(\{v_i, v_j\}) & \text{if } \{v_i, v_j\} \in E \\ 0 & \text{otherwise.} \end{cases}$$

For example, Figure 5.9 displays the adjacency matrices for the graphs shown in Figures 5.6(a) and 5.8 where $v_1 = a$, $v_2 = b$, $v_3 = c$, and $v_4 = d$.

5.2 Matching Problems

We begin our study of graph theory by considering the scenario where the nodes in a graph represent people and the edges represent a relationship between pairs of people such as "likes", "marries", and so on. Now, you may be wondering what marriage has to do with computer science, and with good reason. It turns out that the techniques we will develop apply to much more general scenarios where instead of matching men to women, we need to match packets to paths in a network, applicants to jobs, or Internet traffic to web servers. And, as we will describe later, these techniques are widely used in practice.

In our first example, we will show how graph theory can be used to debunk an urban legend about sexual practices in America. Yes, you read correctly. So, fasten your seat belt—who knew that math might actually be interesting!

5.2.1 Sex in America

On average, who has more opposite-gender partners: men or women?

Sexual demographics have been the subject of many studies. In one of the largest, researchers from the University of Chicago interviewed a random sample of 2500 Americans over several years to try to get an answer to this question. Their study, published in 1994, and entitled *The Social Organization of Sexuality* found that, on average, men have 74% more opposite-gender partners than women.

Other studies have found that the disparity is even larger. In particular, ABC News claimed that the average man has 20 partners over his lifetime, and the average woman has 6, for a percentage disparity of 233%. The ABC News study, aired on Primetime Live in 2004, purported to be one of the most scientific ever done, with only a 2.5% margin of error. It was called "American Sex Survey: A peek between the sheets." The promotion for the study is even better:

> A ground breaking ABC News "Primetime Live" survey finds a range of eye-popping sexual activities, fantasies and attitudes in this country, confirming some conventional wisdom, exploding some myths—and venturing where few scientific surveys have gone before.

Probably that last part about going where few scientific surveys have gone before is pretty accurate!

Yet again, in August, 2007, the N.Y. Times reported on a study by the National Center for Health Statistics of the U.S. Government showing that men had seven partners while women had four.

Anyway, whose numbers do you think are more accurate, the University of Chicago, ABC News, or the National Center for Health Statistics?—don't answer; this is a setup question like "When did you stop beating your wife?" Using a little graph theory, we will now explain why none of these findings can be anywhere near the truth.

Let's model the question of heterosexual partners in graph theoretic terms. To do this, we'll let G be the graph whose vertices, V, are all the people in America. Then we split V into two separate subsets: M, which contains all the males, and F, which contains all the females.[6] We'll put an edge between a male and a female iff they have been sexual partners. A possible subgraph of this graph is illustrated in Figure 5.10 with males on the left and females on the right.

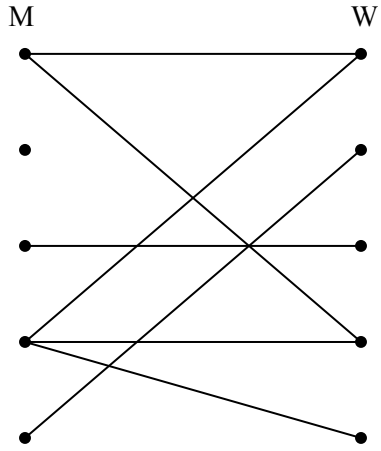


Figure 5.10 A possible subgraph of the sex partners graph.

Actually, G is a pretty hard graph to figure out, let alone draw. The graph is *enormous*: the US population is about 300 million, so $|V| \approx 300M$. In the United States, approximately 50.8% of the populatin is female and 49.2% is male, and so $|M| \approx 147.6M$, and $|F| \approx 152.4M$. And we don't even have trustworthy estimates of how many edges there are, let alone exactly which couples are adjacent. But it turns out that we don't need to know any of this to debunk the sex surveys—we just need to figure out the relationship between the average number of partners per male and partners per female. To do this, we note that every edge is incident to exactly one M vertex and one F vertex (remember, we're only considering male-female relationships); so the sum of the degrees of the M vertices equals the number of edges, and the sum of the degrees of the F vertices equals the

[6] For simplicity, we'll ignore the possibility of someone being both, or neither, a man and a woman.

number of edges. So these sums are equal:

$$\sum_{x \in M} \deg(x) = \sum_{y \in F} \deg(y).$$

If we divide both sides of this equation by the product of the sizes of the two sets, $|M| \cdot |F|$, we obtain

$$\left(\frac{\sum_{x \in M} \deg(x)}{|M|}\right) \cdot \frac{1}{|F|} = \left(\frac{\sum_{y \in F} \deg(y)}{|F|}\right) \cdot \frac{1}{|M|} \tag{5.1}$$

Notice that

$$\frac{\sum_{x \in M} \deg(x)}{|M|}$$

is simply the average degree of a node in M. This is the average number of opposite-gender partners for a male in America. Similarly,

$$\frac{\sum_{x \in F} \deg(x)}{|F|}$$

is the average degree of a node in F, which is the average number of opposite-gender partners for a female in America. Hence, Equation 5.1 implies that on average, an American male has $|F|/|M|$ times as many opposite-gender partners as the average American female.

From the Census Bureau reports, we know that there are slightly more females than males in America; in particular $|F|/|M|$ is about 1.035. So we know that on average, males have 3.5% more opposite-gender partners than females. Of course, this statistic really says nothing about any sex's promiscuity or selectivity. Remarkably, promiscuity is completely irrelevant in this analysis. That is because the ratio of the average number of partners is completely determined by the relative number of males and females. Collectively, males and females have the same number of opposite gender partners, since it takes one of each set for every partnership, but there are fewer males, so they have a higher ratio. This means that the University of Chicago, ABC, and the Federal Government studies are way off. After a huge effort, they gave a totally wrong answer.

There's no definite explanation for why such surveys are consistently wrong. One hypothesis is that males exaggerate their number of partners—or maybe females downplay theirs—but these explanations are speculative. Interestingly, the principal author of the National Center for Health Statistics study reported that she knew the results had to be wrong, but that was the data collected, and her job was to report it.

The same underlying issue has led to serious misinterpretations of other survey data. For example, a few years ago, the Boston Globe ran a story on a survey of the study habits of students on Boston area campuses. Their survey showed that on average, minority students tended to study with non-minority students more than the other way around. They went on at great length to explain why this "remarkable phenomenon" might be true. But it's not remarkable at all—using our graph theory formulation, we can see that all it says is that there are fewer minority students than non-minority students, which is, of course what "minority" means.

The Handshaking Lemma

The previous argument hinged on the connection between a sum of degrees and the number edges. There is a simple connection between these quantities in any graph:

Lemma 5.2.1 (The Handshaking Lemma)**.** *The sum of the degrees of the vertices in a graph equals twice the number of edges.*

Proof. Every edge contributes two to the sum of the degrees, one for each of its endpoints. ■

Lemma 5.2.1 is called the *Handshake Lemma* because if we total up the number of people each person at a party shakes hands with, the total will be twice the number of handshakes that occurred.

5.2.2 Bipartite Matchings

There were two kinds of vertices in the "Sex in America" graph—males and females, and edges only went between the two kinds. Graphs like this come up so frequently that they have earned a special name—they are called *bipartite graphs*.

Definition 5.2.2. A *bipartite graph* is a graph together with a partition of its vertices into two sets, L and R, such that every edge is incident to a vertex in L and to a vertex in R.

The bipartite matching problem is related to the sex-in-America problem that we just studied; only now the goal is to get everyone happily married. As you might imagine, this is not possible for a variety of reasons, not the least of which is the fact that there are more women in America than men. So, it is simply not possible to marry every woman to a man so that every man is married only once.

But what about getting a mate for every man so that every woman is married only once? Is it possible to do this so that each man is paired with a woman that he likes? The answer, of course, depends on the bipartite graph that represents who

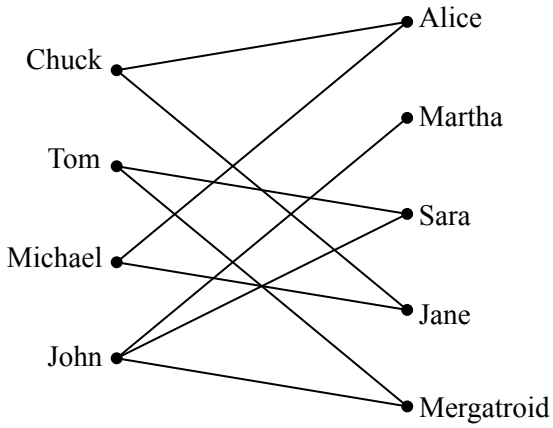


Figure 5.11 A graph where an edge between a man and woman denotes that the man likes the woman.

likes who, but the good news is that it is possible to find natural properties of the who-likes-who graph that completely determine the answer to this question.

In general, suppose that we have a set of men and an equal-sized or larger set of women, and there is a graph with an edge between a man and a woman if the man likes the woman. Note that in this scenario, the "likes" relationship need not be symmetric, since for the time being, we will only worry about finding a mate for each man that he likes.[7] (Later, we will consider the "likes" relationship from the female perspective as well.) For example, we might obtain the graph in Figure 5.11.

In this problem, a *matching* will mean a way of assigning every man to a woman so that different men are assigned to different women, and a man is always assigned to a woman that he likes. For example, one possible matching for the men is shown in Figure 5.12.

The Matching Condition

A famous result known as Hall's Matching Theorem gives necessary and sufficient conditions for the existence of a matching in a bipartite graph. It turns out to be a remarkably useful mathematical tool.

We'll state and prove Hall's Theorem using man-likes-woman terminology. Define *the set of women liked by a given set of men* to consist of all women liked by at least one of those men. For example, the set of women liked by Tom and John in

[7] By the way, we do not mean to imply that marriage should or should not be of a heterosexual nature. Nor do we mean to imply that men should get their choice instead of women. It's just that with bipartite graphs, the edges only connected male nodes to female nodes and there are fewer men in America. So please don't take offense.

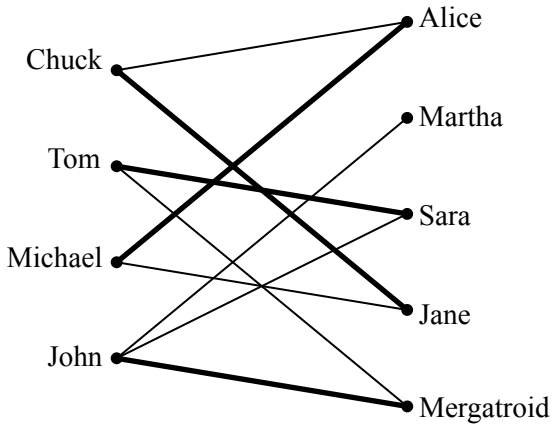


Figure 5.12 One possible matching for the men is shown with bold edges. For example, John is matched with Jane.

Figure 5.11 consists of Martha, Sarah, and Mergatroid. For us to have any chance at all of matching up the men, the following *matching condition* must hold:

Every subset of men likes at least as large a set of women.

For example, we can not find a matching if some set of 4 men like only 3 women. Hall's Theorem says that this necessary condition is actually sufficient; if the matching condition holds, then a matching exists.

Theorem 5.2.3. *A matching for a set of men M with a set of women W can be found if and only if the matching condition holds.*

Proof. First, let's suppose that a matching exists and show that the matching condition holds. Consider an arbitrary subset of men. Each man likes at least the woman he is matched with. Therefore, every subset of men likes at least as large a set of women. Thus, the matching condition holds.

Next, let's suppose that the matching condition holds and show that a matching exists. We use strong induction on $|M|$, the number of men, on the predicate:

$$P(m) ::= \text{for any set of } m \text{ men } M, \text{ if the matching condition holds for } M, \text{ then there is a matching for } M.$$

Base Case ($|M| = 1$): If $|M| = 1$, then the matching condition implies that the lone man likes at least one woman, and so a matching exists.

Inductive Step: We need to show that $P(m)$ IMPLIES $P(m+1)$. Suppose that $|M| = m + 1 \geq 2$.

Case 1: Every proper subset[8] of men likes a *strictly larger* set of women. In this case, we have some latitude: we pair an arbitrary man with a woman he likes and send them both away. The matching condition still holds for the remaining men and women since we have removed only one woman, so we can match the rest of the men by induction.

Case 2: Some proper subset of men $X \subset M$ likes an *equal-size* set of women $Y \subset W$. We match the men in X with the women in Y by induction and send them all away. We can also match the rest of the men by induction if we show that the matching condition holds for the remaining men and women. To check the matching condition for the remaining people, consider an arbitrary subset of the remaining men $X' \subseteq (M - X)$, and let Y' be the set of remaining women that they like. We must show that $|X'| \leq |Y'|$. Originally, the combined set of men $X \cup X'$ liked the set of women $Y \cup Y'$. So, by the matching condition, we know:

$$|X \cup X'| \leq |Y \cup Y'|$$

We sent away $|X|$ men from the set on the left (leaving X') and sent away an equal number of women from the set on the right (leaving Y'). Therefore, it must be that $|X'| \leq |Y'|$ as claimed.

So in both cases, there is a matching for the men, which completes the proof of the Inductive step. The theorem follows by induction. ■

The proof of Theorem 5.2.3 gives an algorithm for finding a matching in a bipartite graph, albeit not a very efficient one. However, efficient algorithms for finding a matching in a bipartite graph do exist. Thus, if a problem can be reduced to finding a matching, the problem can be solved from a computational perspective.

A Formal Statement

Let's restate Theorem 5.2.3 in abstract terms so that you'll not always be condemned to saying, "Now this group of men likes at least as many women..."

Definition 5.2.4. A *matching* in a graph, G, is a set of edges such that no two edges in the set share a vertex. A matching is said to *cover* a set, L, of vertices iff each vertex in L has an edge of the matching incident to it. A matching is said to be *perfect* if every node in the graph is incident to an edge in the matching. In any graph, the set $N(S)$, of *neighbors* of some set, S, of vertices is the set of all vertices adjacent to some vertex in S. That is,

$$N(S) ::= \{ r \mid \{s, r\} \text{ is an edge for some } s \in S \}.$$

[8] Recall that a subset A of B is *proper* if $A \neq B$.

S is called a *bottleneck* if

$$|S| > |N(S)|.$$

Theorem 5.2.5 (Hall's Theorem)**.** *Let G be a bipartite graph with vertex partition L, R. There is matching in G that covers L iff no subset of L is a bottleneck.*

An Easy Matching Condition

The bipartite matching condition requires that *every* subset of men has a certain property. In general, verifying that every subset has some property, even if it's easy to check any particular subset for the property, quickly becomes overwhelming because the number of subsets of even relatively small sets is enormous—over a billion subsets for a set of size 30. However, there is a simple property of vertex degrees in a bipartite graph that guarantees the existence of a matching. Namely, call a bipartite graph *degree-constrained* if vertex degrees on the left are at least as large as those on the right. More precisely,

Definition 5.2.6. A bipartite graph G with vertex partition L, R where $|L| \leq |R|$ is *degree-constrained* if $\deg(l) \geq \deg(r)$ for every $l \in L$ and $r \in R$.

For example, the graph in Figure 5.11 is degree constrained since every node on the left is adjacent to at least two nodes on the right while every node on the right is incident to at most two nodes on the left.

Theorem 5.2.7. *Let G be a bipartite graph with vertex partition L, R where $|L| \leq |R|$. If G is degree-constrained, then there is a matching that covers L.*

Proof. The proof is by contradiction. Suppose that G is degree constrained but that there is no matching that covers L. By Theorem 5.2.5, this means that there must be a bottleneck $S \subseteq L$.

Let d be a value such that $\deg(l) \geq x \geq \deg(r)$ for every $l \in L$ and $r \in R$. Since every edge incident to a node in S is incident to a node in $N(S)$, we know that

$$|N(S)|x \geq |S|x$$

and thus that

$$|N(S)| \geq |S|.$$

This means that S is not a bottleneck, which is a contradiction. Hence G has a matching that covers L. ■

Regular graphs provide a large class of graphs that often arise in practice that are degree constrained. Hence, we can use Theorem 5.2.7 to prove that every regular bipartite graph has a perfect matching. This turns out to be a surprisingly useful result in computer science

Definition 5.2.8. A graph is said to be *regular* if every node has the same degree.

Theorem 5.2.9. *Every regular bipartite graph has a perfect matching.*

Proof. Let G be a regular bipartite graph with vertex partition L, R where $|L| \leq |R|$. Since regular graphs are degree-constrained, we know by Theorem 5.2.7 that there must be a matching in G that covers L. Since G is regular, we also know that $|L| = |R|$ and thus the matching must also cover R. This means that every node in G is incident to an edge in the matching and thus G has a perfect matching. ■

5.2.3 The Stable Marriage Problem

We next consider a version of the bipartite matching problem where there are an equal number of men and women, and where each person has preferences about who they would like to marry. In fact, we assume that each man has a complete list of all the women ranked according to his preferences, with no ties. Likewise, each woman has a ranked list of all of the men.

The preferences don't have to be symmetric. That is, Jennifer might like Brad best, but Brad doesn't necessarily like Jennifer best. The goal is to marry everyone: every man must marry exactly one woman and vice-versa—no polygamy. Moreover, we would like to find a matching between men and women that is *stable* in the sense that there is no pair of people that prefer each other to their spouses.

For example, suppose *every* man likes Angelina best, and every woman likes Brad best, but Brad and Angelina are married to other people, say Jennifer and Billy Bob. Now *Brad and Angelina prefer each other to their spouses*, which puts their marriages at risk: pretty soon, they're likely to start spending late nights together working on problem sets!

This unfortunate situation is illustrated in Figure 5.13, where the digits "1" and "2" near a man shows which of the two women he ranks first second, respectively, and similarly for the women.

More generally, in any matching, a man and woman who are not married to each other and who like each other better than their spouses, is called a *rogue couple*. In the situation shown in Figure 5.13, Brad and Angelina would be a rogue couple.

Having a rogue couple is not a good thing, since it threatens the stability of the marriages. On the other hand, if there are no rogue couples, then for any man and woman who are not married to each other, at least one likes their spouse better than the other, and so they won't be tempted to start an affair.

Definition 5.2.10. A *stable matching* is a matching with no rogue couples.

The question is, given everybody's preferences, how do you find a stable set of marriages? In the example consisting solely of the four people in Figure 5.13, we

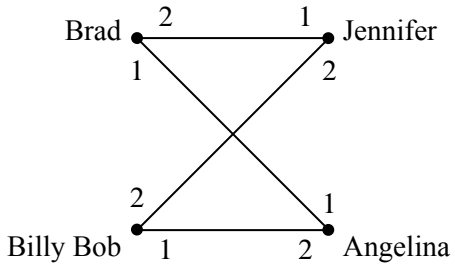


Figure 5.13 Preferences for four people. Both men like Angelina best and both women like Brad best.

could let Brad and Angelina both have their first choices by marrying each other. Now neither Brad nor Angelina prefers anybody else to their spouse, so neither will be in a rogue couple. This leaves Jen not-so-happily married to Billy Bob, but neither Jen nor Billy Bob can entice somebody else to marry them, and so there is a stable matching.

Surprisingly, there is always a stable matching among a group of men and women. The surprise springs in part from considering the apparently similar "buddy" matching problem. That is, if people can be paired off as buddies, regardless of gender, then a stable matching *may not* be possible. For example, Figure 5.14 shows a situation with a love triangle and a fourth person who is everyone's last choice. In this figure Mergatroid's preferences aren't shown because they don't even matter. Let's see why there is no stable matching.

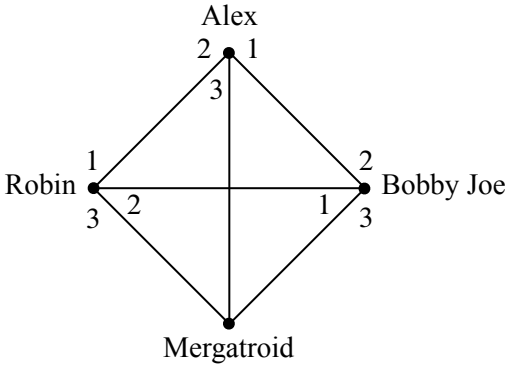


Figure 5.14 Some preferences with no stable buddy matching.

Lemma 5.2.11. *There is no stable buddy matching among the four people in Figure 5.14.*

Proof. We'll prove this by contradiction.

Assume, for the purposes of contradiction, that there is a stable matching. Then there are two members of the love triangle that are matched. Since preferences in the triangle are symmetric, we may assume in particular, that Robin and Alex are matched. Then the other pair must be Bobby-Joe matched with Mergatroid.

But then there is a rogue couple: Alex likes Bobby-Joe best, and Bobby-Joe prefers Alex to his buddy Mergatroid. That is, Alex and Bobby-Joe are a rogue couple, contradicting the assumed stability of the matching. ■

So getting a stable *buddy* matching may not only be hard, it may be impossible. But when mens are only allowed to marry women, and vice versa, then it turns out that a stable matching can always be found.[9]

The Mating Ritual

The procedure for finding a stable matching involves a *Mating Ritual* that takes place over several days. The following events happen each day:

Morning: Each woman stands on her balcony. Each man stands under the balcony of his favorite among the women on his list, and he serenades her. If a man has no women left on his list, he stays home and does his math homework.

Afternoon: Each woman who has one or more suitors serenading her, says to her favorite among them, "We might get engaged. Come back tomorrow." To the other suitors, she says, "No. I will never marry you! Take a hike!"

Evening: Any man who is told by a woman to take a hike, crosses that woman off his list.

Termination condition: When a day arrives in which every woman has at most one suitor, the ritual ends with each woman marrying her suitor, if she has one.

There are a number of facts about this Mating Ritual that we would like to prove:

- The Ritual eventually reaches the termination condition.
- Everybody ends up married.
- The resulting marriages are stable.

There is a Marriage Day

It's easy to see why the Mating Ritual has a terminal day when people finally get married. Every day on which the ritual hasn't terminated, at least one man crosses a woman off his list. (If the ritual hasn't terminated, there must be some woman serenaded by at least two men, and at least one of them will have to cross her off his

[9] Once again, we disclaim any political statement here—its just the way that the math works out.

list). If we start with n men and n women, then each of the n men's lists initially has n women on it, for a total of n^2 list entries. Since no women ever gets added to a list, the total number of entries on the lists decreases every day that the Ritual continues, and so the Ritual can continue for at most n^2 days.

They All Live Happily Every After...

We still have to prove that the Mating Ritual leaves everyone in a stable marriage. To do this, we note one very useful fact about the Ritual: if a woman has a favorite suitor on some morning of the Ritual, then that favorite suitor will still be serenading her the next morning—because his list won't have changed. So she is sure to have today's favorite man among her suitors tomorrow. That means she will be able to choose a favorite suitor tomorrow who is at least as desirable to her as today's favorite. So day by day, her favorite suitor can stay the same or get better, never worse. This sounds like an invariant, and it is.

Definition 5.2.12. Let P be the predicate: For every woman, w, and every man, m, if w is crossed off m's list, then w has a suitor whom she prefers over m.

Lemma 5.2.13. *P is an invariant for The Mating Ritual.*

Proof. By induction on the number of days.

Base Case: In the beginning (that is, at the end of day 0), every woman is on every list—no one has been crossed off and so P is vacuously true.

Inductive Step: Assume P is true at the end of day d and let w be a woman that has been crossed off a man m's list by the end of day $d+1$.

Case 1: w was crossed off m's list on day $d+1$. Then, w must have a suitor she prefers on day $d+1$.

Case 2: w was crossed off m's list prior to day $d+1$. Since P is true at the end of day d, this means that w has a suitor she prefers to m on day d. She therefore has the same suitor or someone she prefers better at the end of day $d+1$.

In both cases, P is true at the end of day $d+1$ and so P must be an invariant. ■

With Lemma 5.2.13 in hand, we can now prove:

Theorem 5.2.14. *Everyone is married by the Mating Ritual.*

Proof. By contradiction. Assume that it is the last day of the Mating Ritual and someone does not get married. Since there are an equal number of men and women,

and since bigamy is not allowed, this means that at least one man (call him Bob) and at least one woman do not get married.

Since Bob is not married, he can't be serenading anybody and so his list must be empty. This means that Bob has crossed every woman off his list and so, by invariant P, every woman has a suitor whom she prefers to Bob. Since it is the last day and every woman still has a suitor, this means that every woman gets married. This is a contradiction since we already argued that at least one woman is *not* married. Hence our assumption must be false and so everyone must be married. ■

Theorem 5.2.15. *The Mating Ritual produces a stable matching.*

Proof. Let Brad and Jen be any man and woman, respectively, that are *not* married to each other on the last day of the Mating Ritual. We will prove that Brad and Jen are not a rogue couple, and thus that all marriages on the last day are stable. There are two cases to consider.

Case 1: Jen is not on Brad's list by the end. Then by invariant P, we know that Jen has a suitor (and hence a husband) that she prefers to Brad. So she's not going to run off with Brad—Brad and Jen cannot be a rogue couple.

Case 2: Jen is on Brad's list. But since Brad is not married to Jen, he must be choosing to serenade his wife instead of Jen, so he must prefer his wife. So he's not going to run off with Jen—once again, Brad and Jenn are not a rogue couple. ■

. . . Especially the Men

Who is favored by the Mating Ritual, the men or the women? The women *seem* to have all the power: they stand on their balconies choosing the finest among their suitors and spurning the rest. What's more, we know their suitors can only change for the better as the Ritual progresses. Similarly, a man keeps serenading the woman he most prefers among those on his list until he must cross her off, at which point he serenades the next most preferred woman on his list. So from the man's perspective, the woman he is serenading can only change for the worse. Sounds like a good deal for the women.

But it's not! The fact is that from the beginning, the men are serenading their first choice woman, and the desirability of the woman being serenaded decreases only enough to ensure overall stability. The Mating Ritual actually does as well as possible for all the men and does the worst possible job for the women.

To explain all this we need some definitions. Let's begin by observing that while The Mating Ritual produces one stable matching, there may be other stable matchings among the same set of men and women. For example, reversing the roles of men and women will often yield a different stable matching among them.

But some spouses might be out of the question in all possible stable matchings. For example, given the preferences shown in Figure 5.13, Brad is just not in the realm of possibility for Jennifer, since if you ever pair them, Brad and Angelina will form a rogue couple.

Definition 5.2.16. Given a set of preference lists for all men and women, one person is in another person's *realm of possible spouses* if there is a stable matching in which the two people are married. A person's *optimal spouse* is their most preferred person within their realm of possibility. A person's *pessimal spouse* is their least preferred person in their realm of possibility.

Everybody has an optimal and a pessimal spouse, since we know there is at least one stable matching, namely, the one produced by the Mating Ritual. Now here is the shocking truth about the Mating Ritual:

Theorem 5.2.17. *The Mating Ritual marries every man to his optimal spouse.*

Proof. By contradiction. Assume for the purpose of contradiction that some man does not get his optimal spouse. Then there must have been a day when he crossed off his optimal spouse—otherwise he would still be serenading (and would ultimately marry) her or some even more desirable woman.

By the Well Ordering Principle, there must be a *first* day when a man (call him "Keith") crosses off his optimal spouse (call her Nicole). According to the rules of the Ritual, Keith crosses off Nicole because Nicole has a preferred suitor (call him Tom), so

$$\text{Nicole prefers Tom to Keith.} \tag{$*$}$$

Since this is the first day an optimal woman gets crossed off, we know that Tom had not previously crossed off his optimal spouse, and so

$$\text{Tom ranks Nicole at least as high as his optimal spouse.} \tag{$**$}$$

By the definition of an optimal spouse, there must be some stable set of marriages in which Keith gets his optimal spouse, Nicole. But then the preferences given in $(*)$ and $(**)$ imply that Nicole and Tom are a rogue couple within this supposedly stable set of marriages (think about it). This is a contradiction. ■

Theorem 5.2.18. *The Mating Ritual marries every woman to her pessimal spouse.*

Proof. By contradiction. Assume that the theorem is not true. Hence there must be a stable set of marriages $\mathcal{M}$ where some woman (call her Nicole) is married to a man (call him Tom) that she likes less than her spouse in The Mating Ritual (call him Keith). This means that

$$\text{Nicole prefers Keith to Tom.} \tag{$+$}$$

By Theorem 5.2.17 and the fact that Nicole and Keith are married in the Mating Ritual, we know that

$$\text{Keith prefers Nicole to his spouse in } \mathcal{M}. \tag{++}$$

This means that Keith and Nicole form a rogue couple in $\mathcal{M}$, which contradicts the stability of $\mathcal{M}$. ■

Applications

The Mating Ritual was first announced in a paper by D. Gale and L.S. Shapley in 1962, but ten years before the Gale-Shapley paper was published, and unknown by them, a similar algorithm was being used to assign residents to hospitals by the National Resident Matching Program (NRMP)[10]. The NRMP has, since the turn of the twentieth century, assigned each year's pool of medical school graduates to hospital residencies (formerly called "internships") with hospitals and graduates playing the roles of men and women. (In this case, there may be multiple women married to one man, a scenario we consider in the problem section at the end of the chapter.). Before the Ritual-like algorithm was adopted, there were chronic disruptions and awkward countermeasures taken to preserve assignments of graduates to residencies. The Ritual resolved these problems so successfully, that it was used essentially without change at least through 1989.[11]

The Internet infrastructure company, Akamai, also uses a variation of the Mating Ritual to assign web traffic to its servers. In the early days, Akamai used other combinatorial optimization algorithms that got to be too slow as the number of servers (over 65,000 in 2010) and requests (over 800 billion per day) increased. Akamai switched to a Ritual-like approach since it is fast and can be run in a distributed manner. In this case, web requests correspond to women and web servers correspond to men. The web requests have preferences based on latency and packet loss, and the web servers have preferences based on cost of bandwidth and collocation.

Not surprisingly, the Mating Ritual is also used by at least one large online dating agency. Even here, there is no serenading going on—everything is handled by computer.

[10] Of course, there is no serenading going on in the hospitals—the preferences are submitted to a program and the whole process is carried out by a computer.

[11] Much more about the Stable Marriage Problem can be found in the very readable mathematical monograph by Dan Gusfield and Robert W. Irving, The Stable Marriage Problem: Structure and Algorithms, MIT Press, Cambridge, Massachusetts, 1989, 240 pp.

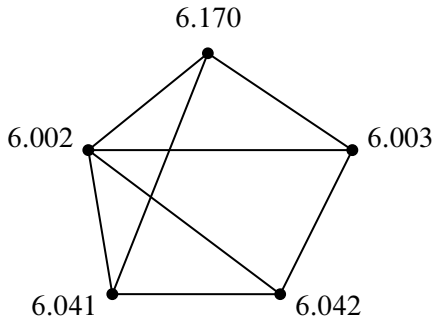


Figure 5.15 A scheduling graph for five exams. Exams connected by an edge cannot be given at the same time.

5.3 Coloring

In Section 5.2, we used edges to indicate an affinity between a pair of nodes. We now consider situations where it is useful to use edges to represent a *conflict* between a pair of nodes. For example, consider the following exam scheduling problem.

5.3.1 An Exam Scheduling Problem

Each term, the MIT Schedules Office must assign a time slot for each final exam. This is not easy, because some students are taking several classes with finals, and (even at MIT) a student can take only one test during a particular time slot. The Schedules Office wants to avoid all conflicts. Of course, you can make such a schedule by having every exam in a different slot, but then you would need hundreds of slots for the hundreds of courses, and the exam period would run all year! So, the Schedules Office would also like to keep exam period short.

The Schedules Office's problem is easy to describe as a graph. There will be a vertex for each course with a final exam, and two vertices will be adjacent exactly when some student is taking both courses. For example, suppose we need to schedule exams for 6.041, 6.042, 6.002, 6.003 and 6.170. The scheduling graph might appear as in Figure 5.15.

6.002 and 6.042 cannot have an exam at the same time since there are students in both courses, so there is an edge between their nodes. On the other hand, 6.042 and 6.170 can have an exam at the same time if they're taught at the same time (which they sometimes are), since no student can be enrolled in both (that is, no student *should* be enrolled in both when they have a timing conflict).

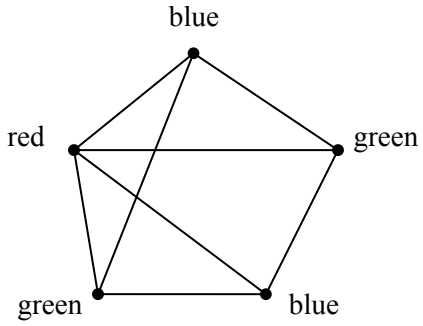


Figure 5.16 A 3-coloring of the exam graph from Figure 5.15.

We next identify each time slot with a color. For example, Monday morning is red, Monday afternoon is blue, Tuesday morning is green, etc. Assigning an exam to a time slot is then equivalent to coloring the corresponding vertex. The main constraint is that *adjacent vertices must get different colors*—otherwise, some student has two exams at the same time. Furthermore, in order to keep the exam period short, we should try to color all the vertices using as *few different colors as possible*. As shown in Figure 5.16, three colors suffice for our example.

The coloring in Figure 5.16 corresponds to giving one final on Monday morning (red), two Monday afternoon (blue), and two Tuesday morning (green). Can we use fewer than three colors? No! We can't use only two colors since there is a triangle in the graph, and three vertices in a triangle must all have different colors.

This is an example of a *graph coloring problem*: given a graph G, assign colors to each node such that adjacent nodes have different colors. A color assignment with this property is called a *valid coloring* of the graph—a "*coloring*," for short. A graph G is *k-colorable* if it has a coloring that uses at most k colors.

Definition 5.3.1. The minimum value of k for which a graph G has a valid k-coloring is called its *chromatic number*, $\chi(G)$.

In general, trying to figure out if you can color a graph with a fixed number of colors can take a long time. It's a classic example of a problem for which no fast algorithms are known. It is easy to check if a coloring works, but it seems really hard to find it. (If you figure out how, then you can get a $1 million Clay prize.)

5.3.2 Degree-Bounded Coloring

There are some simple graph properties that give useful upper bounds on the chromatic number. For example, if the graph is bipartite, then we can color it with 2 colors (one color for the nodes in the "left" set and a second color for the nodes

in the "right" set). In fact, if the graph has any edges at all, then being bipartite is equivalent to being 2-colorable.

Alternatively, if the graph is planar, then the famous 4-Color Theorem says that the graph is 4-colorable. This is a hard result to prove, but we will come close in Section 5.8 where we define planar graphs and prove that they are 5-colorable.

The chromatic number of a graph can also be shown to be small if the vertex degrees of the graph are small. In particular, if we have an upper bound on the degrees of all the vertices in a graph, then we can easily find a coloring with only one more color than the degree bound.

Theorem 5.3.2. *A graph with maximum degree at most k is $(k+1)$-colorable.*

The natural way to try to prove this theorem is to use induction on k. Unfortunately, this approach leads to disaster. It is not that it is impossible, just that it is extremely painful and would ruin your week if you tried it on an exam. When you encounter such a disaster when using induction on graphs, it is usually best to change what you are inducting on. In graphs, typical good choices for the induction parameter are n, the number of nodes, or e, the number of edges.

Proof of Theorem 5.3.2. We use induction on the number of vertices in the graph, which we denote by n. Let $P(n)$ be the proposition that an n-vertex graph with maximum degree at most k is $(k+1)$-colorable.

Base case ($n = 1$): A 1-vertex graph has maximum degree 0 and is 1-colorable, so $P(1)$ is true.

Inductive step: Now assume that $P(n)$ is true, and let G be an $(n+1)$-vertex graph with maximum degree at most k. Remove a vertex v (and all edges incident to it), leaving an n-vertex subgraph, H. The maximum degree of H is at most k, and so H is $(k+1)$-colorable by our assumption $P(n)$. Now add back vertex v. We can assign v a color (from the set of $k+1$ colors) that is different from all its adjacent vertices, since there are at most k vertices adjacent to v and so at least one of the $k+1$ colors is still available. Therefore, G is $(k+1)$-colorable. This completes the inductive step, and the theorem follows by induction. ■

Sometimes $k+1$ colors is the best you can do. For example, in the complete graph, K_n, every one of its n vertices is adjacent to all the others, so all n must be assigned different colors. Of course n colors is also enough, so $\chi(K_n) = n$. In this case, every node has degree $k = n-1$ and so this is an example where Theorem 5.3.2 gives the best possible bound. By a similar argument, we can show that Theorem 5.3.2 gives the best possible bound for *any* graph with degree bounded by k that has K_{k+1} as a subgraph.

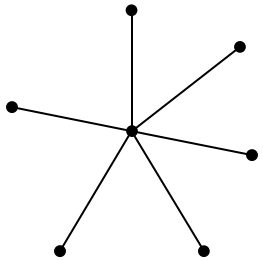


Figure 5.17 A 7-node star graph.

But sometimes $k + 1$ colors is far from the best that you can do. For example, the n-node *star graph* shown in Figure 5.17 has maximum degree $n - 1$ but can be colored using just 2 colors.

5.3.3 Why coloring?

One reason coloring problems frequently arise in practice is because scheduling conflicts are so common. For example, at Akamai, a new version of software is deployed over each of 75,000 servers every few days. The updates cannot be done at the same time since the servers need to be taken down in order to deploy the software. Also, the servers cannot be handled one at a time, since it would take forever to update them all (each one takes about an hour). Moreover, certain pairs of servers cannot be taken down at the same time since they have common critical functions. This problem was eventually solved by making a 75,000-node conflict graph and coloring it with 8 colors—so only 8 waves of install are needed!

Another example comes from the need to assign frequencies to radio stations. If two stations have an overlap in their broadcast area, they can't be given the same frequency. Frequencies are precious and expensive, so you want to minimize the number handed out. This amounts to finding the minimum coloring for a graph whose vertices are the stations and whose edges connect stations with overlapping areas.

Coloring also comes up in allocating registers for program variables. While a variable is in use, its value needs to be saved in a register. Registers can be reused for different variables but two variables need different registers if they are referenced during overlapping intervals of program execution. So register allocation is the coloring problem for a graph whose vertices are the variables; vertices are adjacent if their intervals overlap, and the colors are registers. Once again, the goal is to minimize the number of colors needed to color the graph.

Finally, there's the famous map coloring problem stated in Proposition 1.3.4. The question is how many colors are needed to color a map so that adjacent ter-

ritories get different colors? This is the same as the number of colors needed to color a graph that can be drawn in the plane without edges crossing. A proof that four colors are enough for *planar* graphs was acclaimed when it was discovered about thirty years ago. Implicit in that proof was a 4-coloring procedure that takes time proportional to the number of vertices in the graph (countries in the map). Surprisingly, it's another of those million dollar prize questions to find an efficient procedure to tell if a planar graph really *needs* four colors or if three will actually do the job. (It's always easy to tell if an *arbitrary* graph is 2-colorable.) In Section 5.8, we'll develop enough planar graph theory to present an easy proof that all planar graphs are 5-colorable.

5.4 Getting from A to B in a Graph

5.4.1 Paths and Walks

Definition 5.4.1. A *walk*[12] in a graph, G, is a sequence of vertices

$$v_0, v_1, \ldots, v_k$$

and edges

$$\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}$$

such that $\{v_i, v_{i+1}\}$ is an edge of G for all i where $0 \leq i < k$. The walk is said to *start* at v_0 and to *end* at v_k, and the *length* of the walk is defined to be k. An edge, $\{u, v\}$, is *traversed* n times by the walk if there are n different values of i such that $\{v_i, v_{i+1}\} = \{u, v\}$. A *path* is a walk where all the v_i's are different, that is, $i \neq j$ implies $v_i \neq v_j$. For simplicity, we will refer to paths and walks by the sequence of vertices.[13]

For example, the graph in Figure 5.18 has a length 6 path a, b, c, d, e, f, g. This is the longest path in the graph. Of course, the graph has walks with arbitrarily large lengths; for example, $a, b, a, b, a, b, \ldots$.

The length of a walk or path is the total number of times it traverses edges, which is *one less* than its length as a sequence of vertices. For example, the length 6 path a, b, c, d, e, f, g contains a sequence of 7 vertices.

[12] Some texts use the word *path* for our definition of walk and the term *simple path* for our definition of path.

[13] This works fine for simple graphs since the edges in a walk are completely determined by the sequence of vertices and there is no ambiguity. For graphs with multiple edges, we would need to specify the edges as well as the nodes.

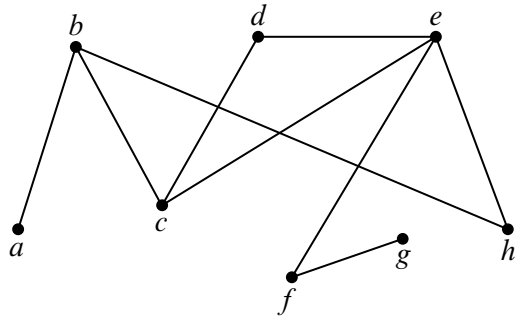


Figure 5.18 A graph containing a path a, b, c, d, e, f, g of length 6.

5.4.2 Finding a Path

Where there's a walk, there's a path. This is sort of obvious, but it's easy enough to prove rigorously using the Well Ordering Principle.

Lemma 5.4.2. *If there is a walk from a vertex u to a vertex v in a graph, then there is a path from u to v.*

Proof. Since there is a walk from u to v, there must, by the Well-ordering Principle, be a minimum length walk from u to v. If the minimum length is zero or one, this minimum length walk is itself a path from u to v. Otherwise, there is a minimum length walk

$$v_0, v_1, \ldots, v_k$$

from $u = v_0$ to $v = v_k$ where $k \geq 2$. We claim this walk must be a path. To prove the claim, suppose to the contrary that the walk is not a path; that is, some vertex on the walk occurs twice. This means that there are integers i, j such that $0 \leq i < j \leq k$ with $v_i = v_j$. Then deleting the subsequence

$$v_{i+1}, \ldots, v_j$$

yields a strictly shorter walk

$$v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_k$$

from u to v, contradicting the minimality of the given walk. ■

Actually, we proved something stronger:

Corollary 5.4.3. *For any walk of length k in a graph, there is a path of length* at most *k with the same endpoints. Moreover, the shortest walk between a pair of vertices is, in fact, a path.*

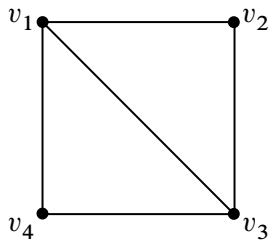


Figure 5.19 A graph for which there are 5 walks of length 3 from v_1 to v_4. The walks are (v_1, v_2, v_1, v_4), (v_1, v_3, v_1, v_4), (v_1, v_4, v_1, v_4), (v_1, v_2, v_3, v_4), and (v_1, v_4, v_3, v_4).

5.4.3 Numbers of Walks

Given a pair of nodes that are connected by a walk of length k in a graph, there are often many walks that can be used to get from one node to the other. For example, there are 5 walks of length 3 that start at v_1 and end at v_4 in the graph shown in Figure 5.19.

There is a surprising relationship between the number of walks of length k between a pair of nodes in a graph G and the kth power of the adjacency matrix A_G for G. The relationship is captured in the following theorem.

Theorem 5.4.4. *Let $G = (V, E)$ be an n-node graph with $V = \{v_1, v_2, \ldots, v_n\}$ and let $A_G = \{a_{ij}\}$ denote the adjacency matrix for G. Let $a_{ij}^{(k)}$ denote the (i, j)-entry of the kth power of A_G. Then the number of walks of length k between v_i and v_j is $a_{ij}^{(k)}$.*

In other words, we can determine the number of walks of length k between any pair of nodes simply by computing the kth power of the adjacency matrix! That's pretty amazing.

For example, the first three powers of the adjacency matrix for the graph in Figure 5.19 are:

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad A^2 = \begin{pmatrix} 3 & 1 & 2 & 1 \\ 1 & 2 & 1 & 2 \\ 2 & 1 & 3 & 1 \\ 1 & 2 & 1 & 2 \end{pmatrix} \quad A^3 = \begin{pmatrix} 4 & 5 & 5 & 5 \\ 5 & 2 & 5 & 2 \\ 5 & 5 & 4 & 5 \\ 5 & 2 & 5 & 2 \end{pmatrix}$$

Sure enough, the $(1, 4)$ coordinate of A^3 is $a_{14}^{(3)} = 5$, which is the number of length 3 walks from v_1 to v_4. And $a_{24}^{(3)} = 2$, which is the number of length 3 walks from v_2 to v_4. By proving the theorem, we'll discover why it is true and thereby uncover the relationship between matrix multiplication and numbers of walks.

Proof of Theorem 5.4.4. The proof is by induction on k. We will let $P(k)$ be the predicate that the theorem is true for k. Let $P_{ij}^{(k)}$ denote the number of walks of length k between v_i and v_j. Then $P(k)$ is the predicate

$$\forall i, j \in [1, n].\ P_{ij}^{(k)} = a_{ij}^{(k)}. \tag{5.2}$$

Base Case ($k = 1$): There are two cases to consider:

Case 1: $\{v_i, v_j\} \in E$. Then $P_{ij}^{(1)} = 1$ since there is precisely one walk of length 1 between v_i and v_j. Moreover, $\{v_i, v_j\} \in E$ means that $a_{ij}^{(1)} = a_{ij} = 1$. So, $P_{ij}^{(1)} = a_{ij}^{(1)}$ in this case.

Case 2: $\{v_i, v_j\} \notin E$. Then $P_{ij}^{(1)} = 0$ since there cannot be any walks of length 1 between v_i and v_j. Moreover, $\{v_i, v_j\} \notin E$ means that $a_{ij} = 0$. So, $P_{ij}^{(1)} = a_{ij}^{(1)}$ in this case as well.

Hence, $P(1)$ must be true.

Inductive Step: Assume $P(k)$ is true. In other words, assume that equation 5.2 holds.

We can group (and thus count the number of) walks of length $k+1$ from v_i to v_j according to the first edge in the walk (call it $\{v_i, v_t\}$). This means that

$$P_{ij}^{(k+1)} = \sum_{t:\{v_i,v_t\}\in E} P_{tj}^{(k)} \tag{5.3}$$

where the sum is over all t such that $\{v_i, v_t\}$ is an edge. Using the fact that $a_{ij} = 1$ if $\{v_i, v_t\} \in E$ and $a_{it} = 0$ otherwise, we can rewrite Equation 5.3 as follows:

$$P_{ij}^{(k+1)} = \sum_{t=1}^{n} a_{it} P_{tj}^{(k)}.$$

By the inductive hypothesis, $P_{tj}^{(k)} = a_{tj}^{(k)}$ and thus

$$P_{ij}^{(k+1)} = \sum_{t=1}^{n} a_{it} a_{tj}^{(k)}.$$

But the formula for matrix multiplication gives that

$$a_{ij}^{(k+1)} = \sum_{t=1}^{n} a_{it} a_{tj}^{(k)}.$$

and so we must have $P_{ij}^{(k+1)} = a_{ij}^{(k+1)}$ for all $i, j \in [1, n]$. Hence $P(k+1)$ is true and the induction is complete. ■

5.4.4 Shortest Paths

Although the connection between the power of the adjacency matrix and the number of walks is cool (at least if you are a mathematician), the problem of counting walks does not come up very often in practice. Much more important is the problem of finding the shortest path between a pair of nodes in a graph.

There is good news and bad news to report on this front. The good news is that it is not very hard to find a shortest path. The bad news is that you can't win one of those million dollar prizes for doing it.

In fact, there are several good algorithms known for finding a Shortest Path between a pair of nodes. The simplest to explain (but not the fastest) is to compute the powers of the adjacency matrix one by one until the value of $a_{ij}^{(k)}$ exceeds 0. That's because Theorem 5.4.4 and Corollary 5.4.3 imply that the length of the shortest path between v_i and v_j will be the smallest value of k for which $a_{ij}^{(k)} > 0$.

Paths in Weighted Graphs

The problem of computing shortest paths in a weighted graph frequently arises in practice. For example, when you drive home for vacation, you usually would like to take the shortest route.

Definition 5.4.5. Given a weighted graph, the length of a path in the graph is the sum of the weights of the edges in the path.

Finding shortest paths in weighted graphs is not a lot harder than finding shortest paths in unweighted graphs. We won't show you how to do it here, but you will study algorithms for finding shortest paths if you take an algorithms course. Not surprisingly, the proof of correctness will use induction.

5.5 Connectivity

Definition 5.5.1. Two vertices in a graph are said to be *connected* if there is a path that begins at one and ends at the other. By convention, every vertex is considered to be connected to itself by a path of length zero.

Definition 5.5.2. A graph is said to be *connected* when every pair of vertices are connected.

5.5.1 Connected Components

Being connected is usually a good property for a graph to have. For example, it could mean that it is possible to get from any node to any other node, or that it is

possible to communicate between any pair of nodes, depending on the application.

But not all graphs are connected. For example, the graph where nodes represent cities and edges represent highways might be connected for North American cities, but would surely not be connected if you also included cities in Australia. The same is true for communication networks like the Internet—in order to be protected from viruses that spread on the Internet, some government networks are completely isolated from the Internet.

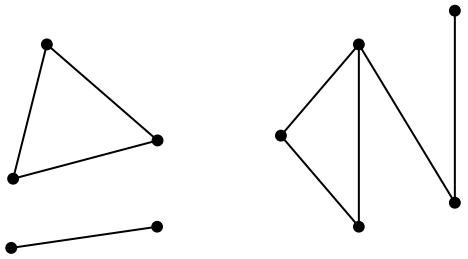


Figure 5.20 One graph with 3 connected components.

For example, the diagram in Figure 5.20 looks like a picture of three graphs, but is intended to be a picture of *one* graph. This graph consists of three pieces (subgraphs). Each piece by itself is connected, but there are no paths between vertices in different pieces. These connected pieces of a graph are called its *connected components*.

Definition 5.5.3. A *connected component* is a subgraph of a graph consisting of some vertex and every node and edge that is connected to that vertex.

So a graph is connected iff it has exactly one connected component. At the other extreme, the empty graph on n vertices has n connected components.

5.5.2 k-Connected Graphs

If we think of a graph as modeling cables in a telephone network, or oil pipelines, or electrical power lines, then we not only want connectivity, but we want connectivity that survives component failure. A graph is called *k-edge connected* if it takes at least k "edge-failures" to disconnect it. More precisely:

Definition 5.5.4. Two vertices in a graph are *k-edge connected* if they remain connected in every subgraph obtained by deleting $k-1$ edges. A graph with at least two vertices is k-edge connected[14] if every two of its vertices are k-edge connected.

[14]The corresponding definition of connectedness based on deleting vertices rather than edges is common in Graph Theory texts and is usually simply called "k-connected" rather than "k-vertex connected."

So 1-edge connected is the same as connected for both vertices and graphs. Another way to say that a graph is k-edge connected is that every subgraph obtained from it by deleting at most $k-1$ edges is connected. For example, in the graph in Figure 5.18, vertices c and e are 3-edge connected, b and e are 2-edge connected, g and e are 1-edge connected, and no vertices are 4-edge connected. The graph as a whole is only 1-edge connected. The complete graph, K_n, is $(n-1)$-edge connected.

If two vertices are connected by k edge-disjoint paths (that is, no two paths traverse the same edge), then they are obviously k-edge connected. A fundamental fact, whose ingenious proof we omit, is Menger's theorem which confirms that the converse is also true: if two vertices are k-edge connected, then there are k edge-disjoint paths connecting them. It even takes some ingenuity to prove this for the case $k = 2$.

5.5.3 The Minimum Number of Edges in a Connected Graph

The following theorem says that a graph with few edges must have many connected components.

Theorem 5.5.5. *Every graph with v vertices and e edges has at least $v - e$ connected components.*

Of course for Theorem 5.5.5 to be of any use, there must be fewer edges than vertices.

Proof. We use induction on the number of edges, e. Let $P(e)$ be the proposition that

> for every v, every graph with v vertices and e edges has at least $v - e$ connected components.

Base case:($e = 0$). In a graph with 0 edges and v vertices, each vertex is itself a connected component, and so there are exactly $v = v - 0$ connected components. So $P(e)$ holds.

Inductive step: Now we assume that the induction hypothesis holds for every e-edge graph in order to prove that it holds for every $(e+1)$-edge graph, where $e \geq 0$. Consider a graph, G, with $e+1$ edges and v vertices. We want to prove that G has at least $v - (e+1)$ connected components. To do this, remove an arbitrary edge $\{a, b\}$ and call the resulting graph G'. By the induction assumption, G' has at least $v - e$ connected components. Now add back the edge $\{a, b\}$ to obtain the original graph G. If a and b were in the same connected component of G', then G has the same connected components as G', so G has at least $v - e > v - (e+1)$ components.

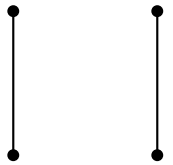


Figure 5.21 A counterexample graph to the False Claim.

Otherwise, if a and b were in different connected components of G', then these two components are merged into one component in G, but all other components remain unchanged, reducing the number of components by 1. Therefore, G has at least $(v-e)-1 = v-(e+1)$ connected components. So in either case, $P(e+1)$ holds. This completes the Inductive step. The theorem now follows by induction. ■

Corollary 5.5.6. *Every connected graph with v vertices has at least $v-1$ edges.*

A couple of points about the proof of Theorem 5.5.5 are worth noticing. First, we used induction on the number of edges in the graph. This is very common in proofs involving graphs, as is induction on the number of vertices. When you're presented with a graph problem, these two approaches should be among the first you consider.

The second point is more subtle. Notice that in the inductive step, we took an arbitrary $(n+1)$-edge graph, threw out an edge so that we could apply the induction assumption, and then put the edge back. You'll see this shrink-down, grow-back process very often in the inductive steps of proofs related to graphs. This might seem like needless effort; why not start with an n-edge graph and add one more to get an $(n+1)$-edge graph? That would work fine in this case, but opens the door to a nasty logical error called *buildup error*.

5.5.4 Build-Up Error

False Claim. *If every vertex in a graph has degree at least 1, then the graph is connected.*

There are many counterexamples; for example, see Figure 5.21.

False proof. We use induction. Let $P(n)$ be the proposition that if every vertex in an n-vertex graph has degree at least 1, then the graph is connected.

Base case: There is only one graph with a single vertex and has degree 0. Therefore, $P(1)$ is vacuously true, since the if-part is false.

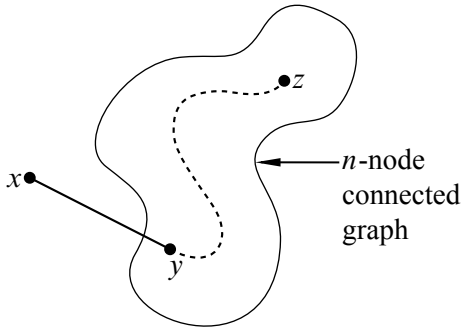


Figure 5.22 Adding a vertex x with degree at least 1 to a connected n-node graph.

Inductive step: We must show that $P(n)$ implies $P(n+1)$ for all $n \geq 1$. Consider an n-vertex graph in which every vertex has degree at least 1. By the assumption $P(n)$, this graph is connected; that is, there is a path between every pair of vertices. Now we add one more vertex x to obtain an $(n+1)$-vertex graph as shown in Figure 5.22.

All that remains is to check that there is a path from x to every other vertex z. Since x has degree at least one, there is an edge from x to some other vertex; call it y. Thus, we can obtain a path from x to z by adjoining the edge $\{x, y\}$ to the path from y to z. This proves $P(n+1)$.

By the principle of induction, $P(n)$ is true for all $n \geq 1$, which proves the theorem ■

Uh-oh...this proof looks fine! Where is the bug? It turns out that the faulty assumption underlying this argument is that *every $(n+1)$-vertex graph with minimum degree 1 can be obtained from an n-vertex graph with minimum degree 1 by adding 1 more vertex.* Instead of starting by considering an arbitrary $(n+1)$-node graph, this proof only considered $(n+1)$-node graphs that you can make by starting with an n-node graph with minimum degree 1.

The counterexample in Figure 5.21 shows that this assumption is false; there is no way to build the 4-vertex graph in Figure 5.21 from a 3-vertex graph with minimum degree 1. Thus the first error in the proof is the statement "This proves $P(n+1)$."

This kind of flaw is known as "build-up error." Usually, build-up error arises from a faulty assumption that every size $n+1$ graph with some property can be "built up" from a size n graph with the same property. (This assumption is correct for some properties, but incorrect for others—such as the one in the argument above.)

One way to avoid an accidental build-up error is to use a "shrink down, grow back" process in the inductive step; that is, start with a size $n+1$ graph, remove a vertex (or edge), apply the inductive hypothesis $P(n)$ to the smaller graph, and then add back the vertex (or edge) and argue that $P(n+1)$ holds. Let's see what would have happened if we'd tried to prove the claim above by this method:

Revised inductive step: We must show that $P(n)$ implies $P(n+1)$ for all $n \geq 1$. Consider an $(n+1)$-vertex graph G in which every vertex has degree at least 1. Remove an arbitrary vertex v, leaving an n-vertex graph G' in which every vertex has degree... uh oh!

The reduced graph G' might contain a vertex of degree 0, making the inductive hypothesis $P(n)$ inapplicable! We are stuck—and properly so, since the claim is false!

Always use shrink-down, grow-back arguments and you'll never fall into this trap.

5.6 Around and Around We Go

5.6.1 Cycles and Closed Walks

Definition 5.6.1. A *closed walk*[15] in a graph G is a sequence of vertices

$$v_0, v_1, \ldots, v_k$$

and edges

$$\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}$$

where v_0 is the same node as v_k and $\{v_i, v_{i+1}\}$ is an edge of G for all i where $0 \leq i < k$. The *length* of the closed walk is k. A closed walk is said to be a *cycle* if $k \geq 3$ and $v_0, v_1, \ldots, v_{k-1}$ are all different.

For example, b, c, d, e, c, b is a closed walk of length 5 in the graph shown in Figure 5.18. It is not a cycle since it contains node c twice. On the other hand, c, d, e, c is a cycle of length 3 in this graph since every node appears just once.

There are many ways to represent the same closed walk or cycle. For example, b, c, d, e, c, b is the same as c, d, e, c, b, c (just starting at node c instead of node b) and the same as b, c, e, d, c, b (just reversing the direction).

[15] Some texts use the word *cycle* for our definition of closed walk and *simple cycle* for our definition of cycle.

Cycles are similar to paths, except that the last node is the first node and the notion of first and last does not matter. Indeed, there are many possible vertex orders that can be used to describe cycles and closed walks, whereas walks and paths have a prescribed beginning, end, and ordering.

5.6.2 Odd Cycles and 2-Colorability

We have already seen that determining the chromatic number of a graph is a challenging problem. There is a special case where this problem is very easy; namely, the case where every cycle in the graph has even length. In this case, the graph is 2-colorable! Of course, this is optimal if the graph has any edges at all. More generally, we will prove

Theorem 5.6.2. *The following properties of a graph are equivalent (that is, if the graph has any one of the properties, then it has all of the properties):*

1. *The graph is bipartite.*
2. *The graph is 2-colorable.*
3. *The graph does not contain any cycles with odd length.*
4. *The graph does not contain any closed walks with odd length.*

Proof. We will show that property 1 IMPLIES property 2, property 2 IMPLIES property 3, property 3 IMPLIES property 4, and property 4 IMPLIES property 1. This will show that all four properties are equivalent by repeated application of Rule 2.1.2 in Section 2.1.2.

1 IMPLIES 2 Assume that $G = (V, E)$ is a bipartite graph. Then V can be partitioned into two sets L and R so that no edge connects a pair of nodes in L nor a pair of nodes in R. Hence, we can use one color for all the nodes in L and a second color for all the nodes in R. Hence $\chi(G) = 2$.

2 IMPLIES 3 Let $G = (V, E)$ be a 2-colorable graph and

$$C ::= v_0, v_1, \ldots, v_k$$

be any cycle in G. Consider any 2-coloring for the nodes of G. Since $\{v_i, v_{i+1}\} \in E$, v_i and v_{i+1} must be differently colored for $0 \leq i < k$. Hence v_0, v_2, v_4, ..., have one color and v_1, v_3, v_5, ..., have the other color. Since C is a cycle, v_k is the same node as v_0, which means they must have the same color, and so k must be an even number. This means that C has even length.

3 IMPLIES 4 The proof is by contradiction. Assume for the purposes of contradiction that G is a graph that does not contain any cycles with odd length (that is, G satisfies Property 3) but that G *does* contain a closed walk with odd length (that is, G does not satisfy Property 4).

Let

$$w ::= v_0, v_1, v_2, \ldots, v_k$$

be the *shortest* closed walk with odd length in G. Since G has no odd-length cycles, w cannot be a cycle. Hence $v_i = v_j$ for some $0 \leq i < j < k$. This means that w is the union of two closed walks:

$$v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_k$$

and

$$v_i, v_{i+1}, \ldots, v_j.$$

Since w has odd length, one of these two closed walks must also have odd length and be shorter than w. This contradicts the minimality of w. Hence 3 IMPLIES 4.

4 IMPLIES 1 Once again, the proof is by contradiction. Assume for the purposes of contradictin that G is a graph without any closed walks with odd length (that is, G satisfies Property 4) but that G is *not* bipartite (that is, G does not satisfy Property 1).

Since G is not bipartite, it must contain a connected component $G' = (V', E')$ that is not bipartite. Let v be some node in V'. For every node $u \in V'$, define

$$\begin{aligned} \text{dist}(u) ::= &\text{ the length of the shortest path from } u \text{ to } v \text{ in } G'. \\ &\text{If } u = v, \text{ the distance is zero.} \end{aligned}$$

Partition V' into sets L and R so that

$$\begin{aligned} L &= \{\, u \mid \text{dist}(u) \text{ is even} \,\}, \\ R &= \{\, u \mid \text{dist}(u) \text{ is odd} \,\}. \end{aligned}$$

Since G' is not bipartite, there must be a pair of adjacent nodes u_1 and u_2 that are both in L or both in R. Let e denote the edge incident to u_1 and u_2.

Let P_i denote a shortest path in G' from u_i to v for $i = 1, 2$. Because u_1 and u_2 are both in L or both in R, it must be the case that P_1 and P_2 both have even length or they both have odd length. In either case, the union of P_1, P_2, and e forms a closed walk with odd length, which is a contradiction. Hence 4 IMPLIES 1. ■

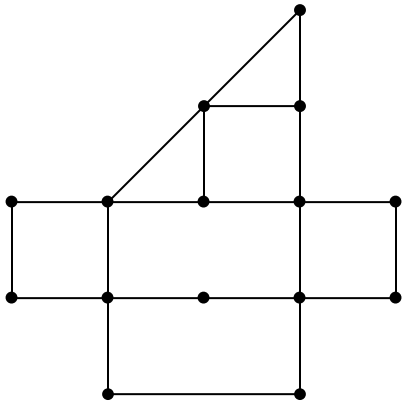


Figure 5.23 A possible floor plan for a museum. Can you find a walk that traverses every edge exactly once?

Theorem 5.6.2 turns out to be useful since bipartite graphs come up fairly often in practice. We'll see examples when we talk about planar graphs in Section 5.8 and when we talk about packet routing in communication networks in Chapter 6.

5.6.3 Euler Tours

Can you walk every hallway in the Museum of Fine Arts *exactly once*? If we represent hallways and intersections with edges and vertices, then this reduces to a question about graphs. For example, could you visit every hallway exactly once in a museum with the floor plan in Figure 5.23?

The entire field of graph theory began when Euler asked whether the seven bridges of Königsberg could all be traversed exactly once—essentially the same question we asked about the Museum of Fine Arts. In his honor, an *Euler walk* is a defined to be a walk that traverses every edge in a graph exactly once. Similarly, an *Euler tour* is an Euler walk that starts and finishes at the same vertex. Graphs with Euler tours and Euler walks both have simple characterizations.

Theorem 5.6.3. *A connected graph has an Euler tour if and only if every vertex has even degree.*

Proof. We first show that if a graph has an Euler tour, then every vertex has even degree. Assume that a graph $G = (V, E)$ has an Euler tour $v_0, v_1, \ldots, v_k$ where $v_k = v_0$. Since every edge is traversed once in the tour, $k = |E|$ and the degree of a node u in G is the number of times that node appears in the sequence $v_0, v_1, \ldots, v_{k-1}$ times two. We multiply by two since if $u = v_i$ for some i where $0 < i < k$, then both $\{v_{i-1}, v_i\}$ and $\{v_i, v_{i+1}\}$ are edges incident to u in G. If $u = v_0 = v_k$,

then both $\{v_{k-1}, v_k\}$ and $\{v_0, v_1\}$ are edges incident to u in G. Hence, the degree of every node is even.

We next show that if the degree of every node is even in a graph $G = (V, E)$, then there is an Euler tour. Let

$$W ::= v_0, v_1, \ldots, v_k$$

be the longest walk in G that traverses *no edge more than once*[16]. W must traverse every edge incident to v_k; otherwise the walk could be extended and W would not be the longest walk that traverses all edges at most once. Moreover, it must be that $v_k = v_0$ and that W is a closed walk, since otherwise v_k would have odd degree in W (and hence in G), which is not possible by assumption.

We conclude the argument with a proof by contradiction. Suppose that W is not an Euler tour. Because G is a connected graph, we can find an edge not in W but incident to some vertex in W. Call this edge $\{u, v_i\}$. But then we can construct a walk W' that is longer than W but that still uses no edge more than once:

$$W' ::= u, v_i, v_{i+1}, \ldots, v_k, v_1, v_2, \ldots, v_i.$$

This contradicts the definition of W, so W must be an Euler tour after all. ■

It is not difficult to extend Theorem 5.6.3 to prove that a connected graph G has an Euler walk if and only if precisely 0 or 2 nodes in G have odd degree. Hence, we can conclude that the graph shown in Figure 5.23 has an Euler walk but not an Euler tour since the graph has precisely two nodes with odd degree.

Although the proof of Theorem 5.6.3 does not explicitly define a method for finding an Euler tour when one exists, it is not hard to modify the proof to produce such a method. The idea is to grow a tour by continually splicing in closed walks until all the edges are consumed.

5.6.4 Hamiltonian Cycles

Hamiltonian cycles are the unruly cousins of Euler tours.

Definition 5.6.4. A *Hamiltonian cycle* in a graph G is a cycle that visits every *node* in G exactly once. Similarly, a *Hamiltonian* path is a path in G that visits every node exactly once.

[16] Did you notice that we are using a variation of the Well Ordering Principle here when we implicitly assume that a longest walk exists? This is ok since the length of a walk where no edge is used more than once is at most $|E|$.

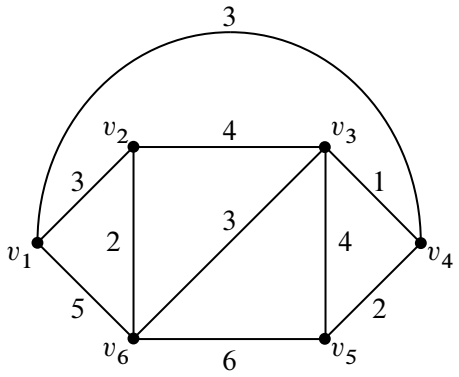


Figure 5.24 A weighted graph. Can you find a cycle with weight 15 that visits every node exactly once?

Although Hamiltonian cycles sound similar to Euler tours—one visits every node once while the other visits every edge once—finding a Hamiltonian cycle can be a lot harder than finding an Euler tour. The same is true for Hamiltonian paths. This is because no one has discovered a simple characterization of all graphs with a Hamiltonian cycle. In fact, determining whether a graph has a Hamiltonian cycle is the same category of problem as the SAT problem of Section 1.5 and the coloring problem in Section 5.3; you get a million dollars for finding an efficient way to determine when a graph has a Hamiltonian cycle—or proving that no procedure works efficiently on all graphs.

5.6.5 The Traveling Salesperson Problem

As if the problem of finding a Hamiltonian cycle is not hard enough, when the graph is weighted, we often want to find a Hamiltonian cycle that has least possible weight. This is a very famous optimization problem known as the Traveling Salesperson Problem.

Definition 5.6.5. Given a weighted graph G, the *weight* of a cycle in G is defined as the sum of the weights of the edges in the cycle.

For example, consider the graph shown in Figure 5.24 and suppose that you would like to visit every node once and finish at the node where you started. Can you find way to do this by traversing a cycle with weight 15?

Needless to say, if you can figure out a fast procedure that finds the optimal cycle for the traveling salesperson, let us know so that we can win a million dollars.

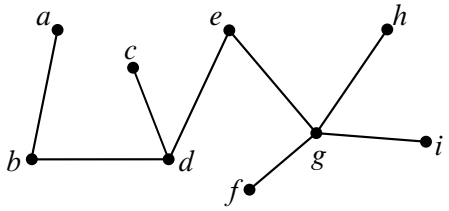


Figure 5.25 A 9-node tree.

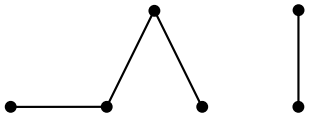


Figure 5.26 A 6-node forest consisting of 2 component trees. Note that this 6-node graph is not itself a tree since it is not connected.

5.7 Trees

As we have just seen, finding good cycles in a graph can be trickier than you might first think. But what if a graph has no cycles at all? Sounds pretty dull. But graphs without cycles (called *acyclic graphs*) are probably the most important graphs of all when it comes to computer science.

5.7.1 Definitions

Definition 5.7.1. A connected acyclic graph is called a *tree*.

For example, Figure 5.25 shows an example of a 9-node tree.

The graph shown in Figure 5.26 is not a tree since it is not connected, but it is a forest. That's because, of course, it consists of a collection of trees.

Definition 5.7.2. If every connected component of a graph G is a tree, then G is a *forest*.

One of the first things you will notice about trees is that they tend to have a lot of nodes with degree one. Such nodes are called *leaves*.

Definition 5.7.3. A *leaf* is a node with degree 1 in a tree (or forest).

For example, the tree in Figure 5.25 has 5 leaves and the forest in Figure 5.26 has 4 leaves.

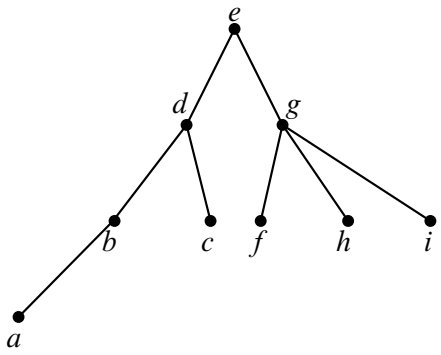


Figure 5.27 The tree from Figure 5.25 redrawn in a leveled fashion, with node E as the root.

Trees are a fundamental data structure in computer science. For example, information is often stored in tree-like data structures and the execution of many recursive programs can be modeled as the traversal of a tree. In such cases, it is often useful to draw the tree in a leveled fashion where the node in the top level is identified as the *root*, and where every edge joins a *parent* to a *child*. For example, we have redrawn the tree from Figure 5.25 in this fashion in Figure 5.27. In this example, node d is a child of node e and a parent of nodes b and c.

In the special case of *ordered binary trees*, every node is the parent of at most 2 children and the children are labeled as being a left-child or a right-child.

5.7.2 Properties

Trees have many unique properties. We have listed some of them in the following theorem.

Theorem 5.7.4. *Every tree has the following properties:*

1. *Any connected subgraph is a tree.*
2. *There is a unique simple path between every pair of vertices.*
3. *Adding an edge between nonadjacent nodes in a tree creates a graph with a cycle.*
4. *Removing any edge disconnects the graph.*
5. *If the tree has at least two vertices, then it has at least two leaves.*
6. *The number of vertices in a tree is one larger than the number of edges.*

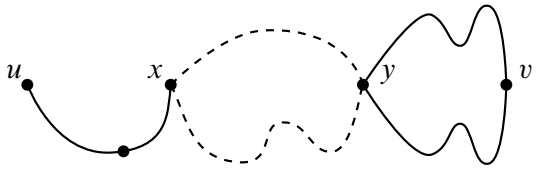


Figure 5.28 If there are two paths between u and v, the graph must contain a cycle.

Proof. 1. A cycle in a subgraph is also a cycle in the whole graph, so any subgraph of an acyclic graph must also be acyclic. If the subgraph is also connected, then by definition, it is a tree.

2. Since a tree is connected, there is at least one path between every pair of vertices. Suppose for the purposes of contradiction, that there are two different paths between some pair of vertices u and v. Beginning at u, let x be the first vertex where the paths diverge, and let y be the next vertex they share. (For example, see Figure 5.28.) Then there are two paths from x to y with no common edges, which defines a cycle. This is a contradiction, since trees are acyclic. Therefore, there is exactly one path between every pair of vertices.

3. An additional edge $\{u, v\}$ together with the unique path between u and v forms a cycle.

4. Suppose that we remove edge $\{u, v\}$. Since the tree contained a unique path between u and v, that path must have been $\{u, v\}$. Therefore, when that edge is removed, no path remains, and so the graph is not connected.

5. Let $v_1, \ldots, v_m$ be the sequence of vertices on a longest path in the tree. Then $m \geq 2$, since a tree with two vertices must contain at least one edge. There cannot be an edge $\{v_1, v_i\}$ for $2 < i \leq m$; otherwise, vertices $v_1, \ldots, v_i$ would from a cycle. Furthermore, there cannot be an edge $\{u, v_1\}$ where u is not on the path; otherwise, we could make the path longer. Therefore, the only edge incident to v_1 is $\{v_1, v_2\}$, which means that v_1 is a leaf. By a symmetric argument, v_m is a second leaf.

6. We use induction on the proposition $P(n) ::=$ there are $n - 1$ edges in any n-vertex tree.

   Base Case ($n = 1$): $P(1)$ is true since a tree with 1 node has 0 edges and $1 - 1 = 0$.

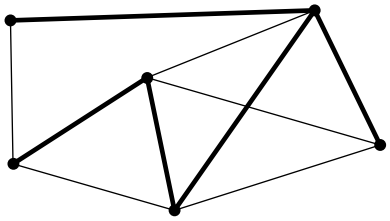


Figure 5.29 A graph where the edges of a spanning tree have been thickened.

Inductive step: Now suppose that $P(n)$ is true and consider an $(n+1)$-vertex tree, T. Let v be a leaf of the tree. You can verify that deleting a vertex of degree 1 (and its incident edge) from any connected graph leaves a connected subgraph. So by part 1 of Theorem 5.7.4, deleting v and its incident edge gives a smaller tree, and this smaller tree has $n-1$ edges by induction. If we re-attach the vertex v and its incident edge, then we find that T has $n = (n+1)-1$ edges. Hence, $P(n+1)$ is true, and the induction proof is complete. ■

Various subsets of properties in Theorem 5.7.4 provide alternative characterizations of trees, though we won't prove this. For example, a *connected* graph with a number of vertices one larger than the number of edges is necessarily a tree. Also, a graph with unique paths between every pair of vertices is necessarily a tree.

5.7.3 Spanning Trees

Trees are everywhere. In fact, every connected graph contains a subgraph that is a tree with the same vertices as the graph. This is a called a *spanning tree* for the graph. For example, Figure 5.29 is a connected graph with a spanning tree highlighted.

Theorem 5.7.5. *Every connected graph contains a spanning tree.*

Proof. By contradiction. Assume there is some connected graph G that has no spanning tree and let T be a connected subgraph of G, with the same vertices as G, and with the smallest number of edges possible for such a subgraph. By the assumption, T is not a spanning tree and so it contains some cycle:

$$\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_k, v_0\}$$

Suppose that we remove the last edge, $\{v_k, v_0\}$. If a pair of vertices x and y was joined by a path not containing $\{v_k, v_0\}$, then they remain joined by that path. On the other hand, if x and y were joined by a path containing $\{v_k, v_0\}$, then they

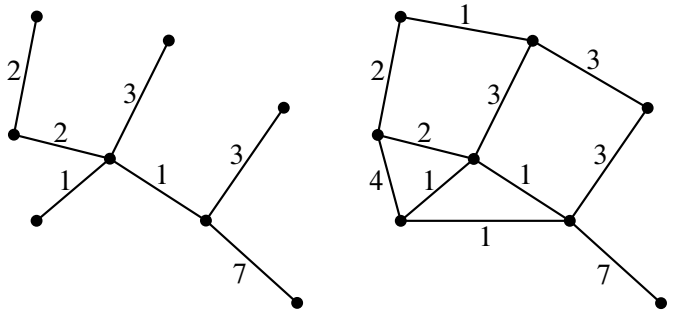


Figure 5.30 A spanning tree (a) with weight 19 for a graph (b).

remain joined by a walk containing the remainder of the cycle. By Lemma 5.4.2, they must also then be joined by a path. So all the vertices of G are still connected after we remove an edge from T. This is a contradiction, since T was defined to be a minimum size connected subgraph with all the vertices of G. So the theorem must be true. ■

5.7.4 Minimum Weight Spanning Trees

Spanning trees are interesting because they connect all the nodes of a graph using the smallest possible number of edges. For example the spanning tree for the 6-node graph shown in Figure 5.29 has 5 edges.

Spanning trees are very useful in practice, but in the real world, not all spanning trees are equally desirable. That's because, in practice, there are often costs associated with the edges of the graph.

For example, suppose the nodes of a graph represent buildings or towns and edges represent connections between buildings or towns. The cost to actually make a connection may vary a lot from one pair of buildings or towns to another. The cost might depend on distance or topography. For example, the cost to connect LA to NY might be much higher than that to connect NY to Boston. Or the cost of a pipe through Manhattan might be more than the cost of a pipe through a cornfield.

In any case, we typically represent the cost to connect pairs of nodes with a weighted edge, where the weight of the edge is its cost. The weight of a spanning tree is then just the sum of the weights of the edges in the tree. For example, the weight of the spanning tree shown in Figure 5.30 is 19.

The goal, of course, is to find the spanning tree with minimum weight, called the min-weight spanning tree (MST for short).

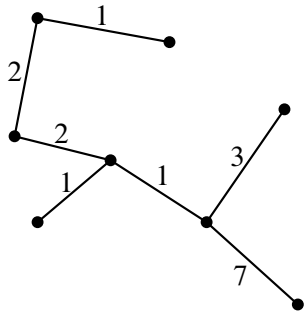


Figure 5.31 An MST with weight 17 for the graph in Figure 5.30(b).

Definition 5.7.6. The *min-weight spanning tree* (MST) of an edge-weighted graph G is the spanning tree of G with the smallest possible sum of edge weights.

Is the spanning tree shown in Figure 5.30(a) an MST of the weighted graph shown in Figure 5.30(b)? Actually, it is not, since the tree shown in Figure 5.31 is also a spanning tree of the graph shown in Figure 5.30(b), and this spanning tree has weight 17.

What about the tree shown in Figure 5.31? Is it an MST? It seems to be, but how do we prove it? In general, how do we find an MST? We could, of course, enumerate all trees, but this could take forever for very large graphs.

Here are two possible algorithms:

Algorithm 1. *Grow a tree one edge at a time by adding the minimum weight edge possible to the tree, making sure that you have a tree at each step.*

Algorithm 2. *Grow a subgraph one edge at a time by adding the minimum-weight edge possible to the subgraph, making sure that you have an acyclic subgraph at each step.*

For example, in the weighted graph we have been considering, we might run Algorithm 1 as follows. We would start by choosing one of the weight 1 edges, since this is the smallest weight in the graph. Suppose we chose the weight 1 edge on the bottom of the triangle of weight 1 edges in our graph. This edge is incident to two weight 1 edges, a weight 4 edge, a weight 7 edge, and a weight 3 edge. We would then choose the incident edge of minimum weight. In this case, one of the two weight 1 edges. At this point, we cannot choose the third weight 1 edge since this would form a cycle, but we can continue by choosing a weight 2 edge. We might end up with the spanning tree shown in Figure 5.32, which has weight 17, the smallest we've seen so far.

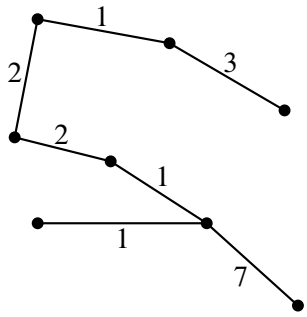


Figure 5.32 A spanning tree found by Algorithm 1.

Now suppose we instead ran Algorithm 2 on our graph. We might again choose the weight 1 edge on the bottom of the triangle of weight 1 edges in our graph. Now, instead of choosing one of the weight 1 edges it touches, we might choose the weight 1 edge on the top of the graph. Note that this edge still has minimum weight, and does not cause us to form a cycle, so Algorithm 2 can choose it. We would then choose one of the remaining weight 1 edges. Note that neither causes us to form a cycle. Continuing the algorithm, we may end up with the same spanning tree in Figure 5.32, though this need not always be the case.

It turns out that both algorithms work, but they might end up with different MSTs. The MST is not necessarily unique—indeed, if all edges of an n-node graph have the same weight ($= 1$), then all spanning trees have weight $n - 1$.

These are examples of greedy approaches to optimization. Sometimes it works and sometimes it doesn't. The good news is that it works to find the MST. In fact, both variations work. It's a little easier to prove it for Algorithm 2, so we'll do that one here.

Theorem 5.7.7. *For any connected, weighted graph G, Algorithm 2 produces an MST for G.*

Proof. The proof is a bit tricky. We need to show the algorithm terminates, that is, that if we have selected fewer than $n - 1$ edges, then we can always find an edge to add that does not create a cycle. We also need to show the algorithm creates a tree of minimum weight.

The key to doing all of this is to show that the algorithm never gets stuck or goes in a bad direction by adding an edge that will keep us from ultimately producing an MST. The natural way to prove this is to show that the set of edges selected at any point is contained in some MST—that is, we can always get to where we need to be. We'll state this as a lemma.

Lemma 5.7.8. *For any $m \geq 0$, let S consist of the first m edges selected by Algorithm 2. Then there exists some MST $T = (V, E)$ for G such that $S \subseteq E$, that is, the set of edges that we are growing is always contained in some MST.*

We'll prove this momentarily, but first let's see why it helps to prove the theorem. Assume the lemma is true. Then how do we know Algorithm 2 can always find an edge to add without creating a cycle? Well, as long as there are fewer than $n-1$ edges picked, there exists some edge in $E - S$ and so there is an edge that we can add to S without forming a cycle. Next, how do we know that we get an MST at the end? Well, once $m = n-1$, we know that S is an MST.

Ok, so the theorem is an easy corollary of the lemma. To prove the lemma, we'll use induction on the number of edges chosen by the algorithm so far. This is very typical in proving that an algorithm preserves some kind of invariant condition—induct on the number of steps taken, that is, the number of edges added.

Our inductive hypothesis $P(m)$ is the following: for any G and any set S of m edges initially selected by Algorithm 2, there exists an MST $T = (V, E)$ of G such that $S \subseteq E$.

For the base case, we need to show $P(0)$. In this case, $S = \emptyset$, so $S \subseteq E$ trivially holds for any MST $T = (V, E)$.

For the inductive step, we assume $P(m)$ holds and show that it implies $P(m+1)$. Let e denote the $(m+1)$st edge selected by Algorithm 2, and let S denote the first m edges selected by Algorithm 2. Let $T^* = (V^*, E^*)$ be the MST such that $S \subseteq E^*$, which exists by the inductive hypothesis. There are now two cases:

Case 1: $e \in E^*$, in which case $S \cup \{e\} \subseteq E^*$, and thus $P(m+1)$ holds.

Case 2: $e \notin E^*$, as illustrated in Figure 5.33. Now we need to find a different MST that contains S and e.

What happens when we add e to T^*? Since T^* is a tree, we get a cycle. (Here we used part 3 of Theorem 5.7.4.) Moreover, the cycle cannot only contains edges in S, since e was chosen so that together with the edges in S, it does not form a cycle. This implies that $\{e\} \cup T^*$ contains a cycle that contains an edge e' of $E^* - S$. For example, such an e' is shown in Figure 5.33.

Note that the weight of e is at most that of e'. This is because Algorithm 2 picks the minimum weight edge that does not make a cycle with S. Since $e' \in T^*$, e' cannot make a cycle with S and if the weight of e were greater than the weight of e', Algorithm 2 would not have selected e ahead of e'.

Okay, we're almost done. Now we'll make an MST that contains $S \cup \{e\}$. Let $T^{**} = (V, E^{**})$ where $E^{**} = (E^* - \{e'\}) \cup \{e\}$, that is, we swap e and e' in T^*.

Claim 5.7.9. *T^{**} is an MST.*

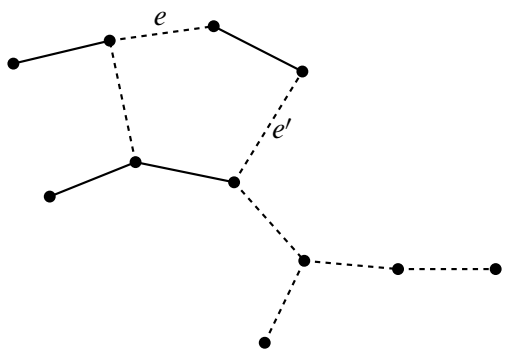


Figure 5.33 The graph formed by adding e to T^*. Edges of S are denoted with solid lines and edges of $E^* - S$ are denoted with dashed lines.

Proof of claim. We first show that T^{**} is a spanning tree. T^{**} is acyclic because it was produced by removing an edge from the only cycle in $T^* \cup \{e\}$. T^{**} is connected since the edge we deleted from $T^* \cup \{e\}$ was on a cycle. Since T^{**} contains all the nodes of G, it must be a spanning tree for G.

Now let's look at the weight of T^{**}. Well, since the weight of e was at most that of e', the weight of T^{**} is at most that of T^*, and thus T^{**} is an MST for G. ■

Since $S \cup \{e\} \subseteq E^{**}$, $P(m+1)$ holds. Thus, Algorithm 2 must eventually produce an MST. This will happens when it adds $n-1$ edges to the subgraph it builds. ■

So now we know for sure that the MST for our example graph has weight 17 since it was produced by Algorithm 2. And we have a fast algorithm for finding a minimum-weight spanning tree for any graph.

5.8 Planar Graphs

5.8.1 Drawing Graphs in the Plane

Suppose there are three dog houses and three human houses, as shown in Figure 5.34. Can you find a route from each dog house to each human house such that no route crosses any other route?

A *quadrapus* is a little-known animal similar to an octopus, but with four arms. Suppose there are five quadrapi resting on the sea floor, as shown in Figure 5.35.

Figure 5.34 Three dog houses and and three human houses. Is there a route from each dog house to each human house so that no pair of routes cross each other?

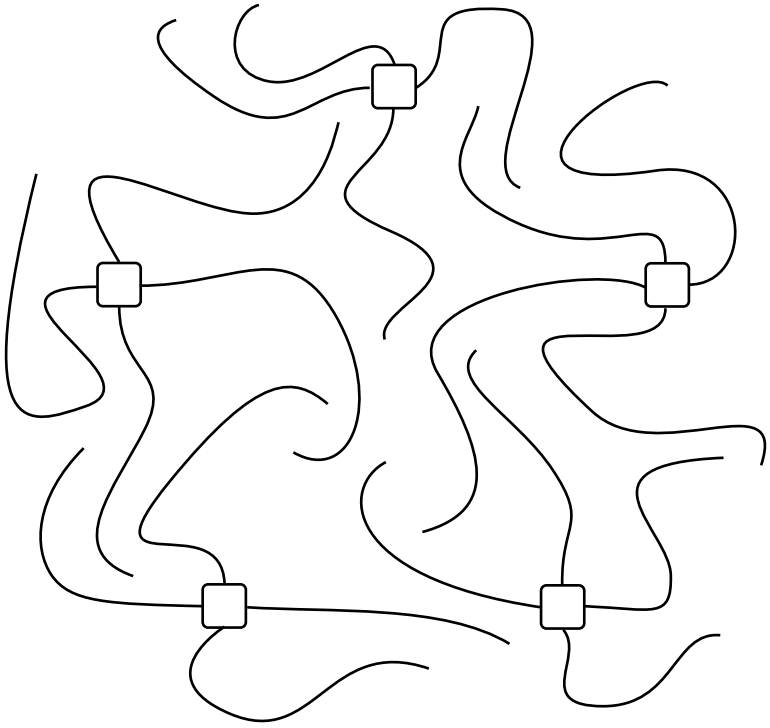


Figure 5.35 Five quadrapi (4-armed creatures).

Can each quadrapus simultaneously shake hands with every other in such a way that no arms cross?

Definition 5.8.1. A *drawing of a graph in the plane* consists of an assignment of vertices to distinct points in the plane and an assignment of edges to smooth, non-self-intersecting curves in the plane (whose endpoints are the nodes incident to the edge). The drawing is *planar* (that is, it is a *planar drawing*) if none of the curves "cross"—that is, if the only points that appear on more than one curve are the vertex points. A *planar graph* is a graph that has a planar drawing.

Thus, these two puzzles are asking whether the graphs in Figure 5.36 are planar; that is, whether they can be redrawn so that no edges cross. The first graph is called the *complete bipartite graph*, $K_{3,3}$, and the second is K_5.

In each case, the answer is, "No—but almost!" In fact, if you remove an edge from either of them, then the resulting graphs *can* be redrawn in the plane so that no edges cross. For example, we have illustrated the planar drawings for each resulting graph in Figure 5.37.

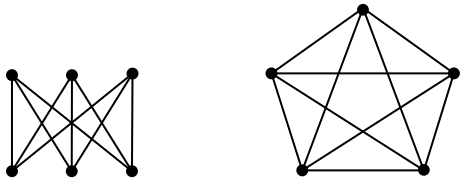


Figure 5.36 $K_{3,3}$ (a) and K_5 (b). Can you redraw these graphs so that no pairs of edges cross?

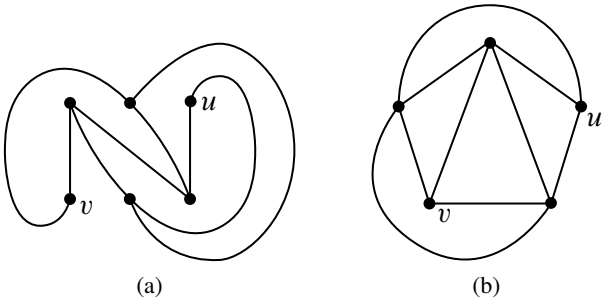


Figure 5.37 Planar drawings of $K_{3,3} - \{u, v\}$ (a) and $K_5 - \{u, v\}$ (b).

Planar drawings have applications in circuit layout and are helpful in displaying graphical data such as program flow charts, organizational charts, and scheduling conflicts. For these applications, the goal is to draw the graph in the plane with as few edge crossings as possible. (See the box on the following page for one such example.)

5.8.2 A Recursive Definition for Planar Graphs

Definition 5.8.1 is perfectly precise but has the challenge that it requires us to work with concepts such as a "smooth curve" when trying to prove results about planar graphs. The trouble is that we have not really laid the groundwork from geometry and topology to be able to reason carefully about such concepts. For example, we haven't really defined what it means for a curve to be smooth—we just drew a simple picture (for example, Figure 5.37) and hoped you would get the idea.

Relying on pictures to convey new concepts is generally not a good idea and can sometimes lead to disaster (or, at least, false proofs). Indeed, it is because of this issue that there have been so many false proofs relating to planar graphs over time.[18] Such proofs usually rely way too heavily on pictures and have way too many statements like,

> As you can see from Figure ABC, it must be that property XYZ holds for all planar graphs.

The good news is that there is another way to define planar graphs that uses only discrete mathematics. In particular, we can define the class of planar graphs as a recursive data type. In order to understand how it works, we first need to understand the concept of a *face* in a planar drawing.

Faces

In a planar drawing of a graph. the curves corresponding to the edges divide up the plane into connected regions. These regions are called the *continuous faces*[19] of the drawing. For example, the drawing in Figure 5.38 has four continuous faces. Face IV, which extends off to infinity in all directions, is called the *outside face*.

Notice that the vertices along the boundary of each of the faces in Figure 5.38 form a cycle. For example, labeling the vertices as in Figure 5.39, the cycles for the face boundaries are

$$abca \qquad abda \qquad bcdb \qquad acda. \tag{5.4}$$

[18] The false proof of the 4-Color Theorem for planar graphs is not the only example.

[19] Most texts drop the word *continuous* from the definition of a face. We need it to differentiate the connected region in the plane from the closed walk in the graph that bounds the region, which we will call a *discrete face*.

When wires are arranged on a surface, like a circuit board or microchip, crossings require troublesome three-dimensional structures. When Steve Wozniak designed the disk drive for the early Apple II computer, he struggled mightily to achieve a nearly planar design:

> For two weeks, he worked late each night to make a satisfactory design. When he was finished, he found that if he moved a connector he could cut down on feedthroughs, making the board more reliable. To make that move, however, he had to start over in his design. This time it only took twenty hours. He then saw another feedthrough that could be eliminated, and again started over on his design. "The final design was generally recognized by computer engineers as brilliant and was by engineering aesthetics beautiful. Woz later said, 'It's something you can only do if you're the engineer and the PC board layout person yourself. That was an artistic layout. The board has virtually no feedthroughs.' "[17]

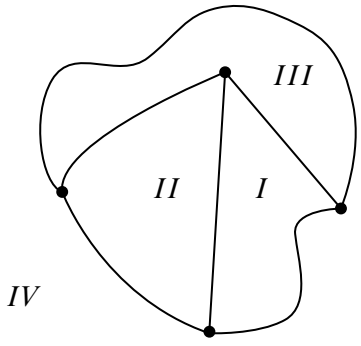


Figure 5.38 A planar drawing with four faces.

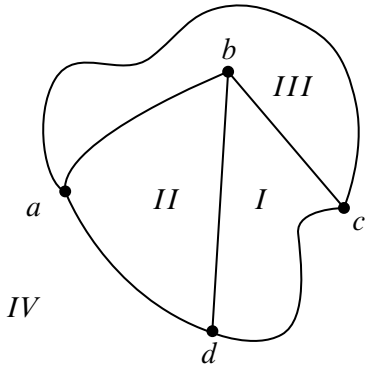


Figure 5.39 The drawing with labeled vertices.

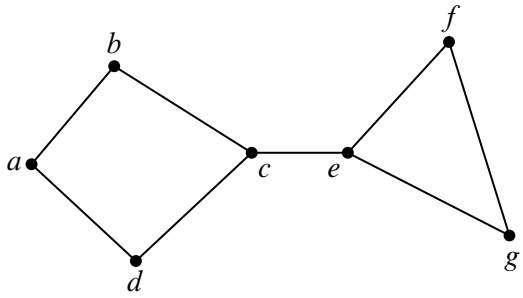


Figure 5.40 A planar drawing with a *bridge*, namely the edge $\{c, e\}$.

These four cycles correspond nicely to the four continuous faces in Figure 5.39. So nicely, in fact, that we can identify each of the faces in Figure 5.39 by its cycle. For example, the cycle $abca$ identifies face III. Hence, we say that the cycles in Equation 5.4 are the *discrete faces* of the graph in Figure 5.39. We use the term "discrete" since cycles in a graph are a discrete data type (as opposed to a region in the plane, which is a continuous data type).

Unfortunately, continuous faces in planar drawings are not always bounded by cycles in the graph—things can get a little more complicated. For example, consider the planar drawing in Figure 5.40. This graph has what we will call a *bridge* (namely, the edge $\{c, e\}$) and the outer face is

$$abcefgecda.$$

This is not a cycle, since it has to traverse the bridge $\{c, e\}$ twice, but it is a closed walk.

As another example, consider the planar drawing in Figure 5.41. This graph has what we will call a *dongle* (namely, the nodes v, x, y, and w, and the edges incident

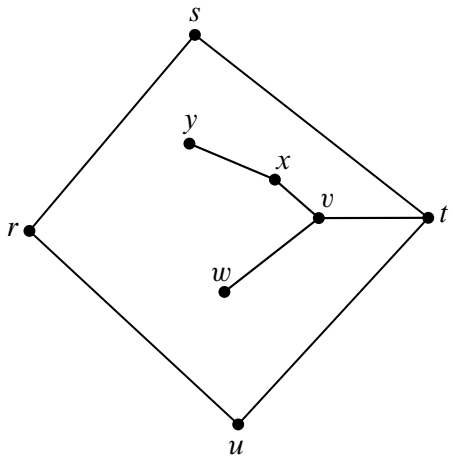


Figure 5.41 A planar drawing with a *dongle*, namely the subgraph with nodes v, w, x, y.

to them) and the inner face is

$$rstvxyxvwvtur.$$

This is not a cycle because it has to traverse *every* edge of the dongle twice—once "coming" and once "going," but once again, it is a closed walk.

It turns out that bridges and dongles are the only complications, at least for connected graphs. In particular, every continuous face in a planar drawing corresponds to a closed walk in the graph. We refer to such closed walks as the *discrete faces* of the drawing.

A Recursive Definition for Planar Embeddings

The association between the continuous faces of a planar drawing and closed walks will allow us to characterize a planar drawing in terms of the closed walks that bound the continuous faces. In particular, it leads us to the discrete data type of *planar embeddings* that we can use in place of continuous planar drawings. Namely, we'll define a planar embedding recursively to be the set of boundary-tracing closed walks that we could get by drawing one edge after another.

Definition 5.8.2. A *planar embedding* of a *connected* graph consists of a nonempty set of closed walks of the graph called the *discrete faces* of the embedding. Planar embeddings are defined recursively as follows:

Base case: If G is a graph consisting of a single vertex v, then a planar embedding of G has one discrete face, namely the length zero closed walk v.

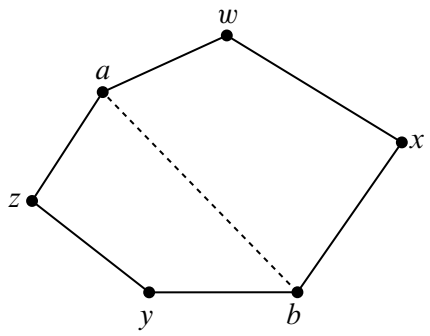


Figure 5.42 The "split a face" case.

Constructor Case (split a face): Suppose G is a connected graph with a planar embedding, and suppose a and b are distinct, nonadjacent vertices of G that appear on some discrete face γ of the planar embedding. That is, γ is a closed walk of the form

$$a \ldots b \ldots a.$$

Then the graph obtained by adding the edge $\{a, b\}$ to the edges of G has a planar embedding with the same discrete faces as G, except that face γ is replaced by the two discrete faces[20]

$$a \ldots ba \quad \text{and} \quad ab \ldots a,$$

as illustrated in Figure 5.42.

Constructor Case (add a bridge): Suppose G and H are connected graphs with planar embeddings and disjoint sets of vertices. Let a be a vertex on a discrete face, γ, in the embedding of G. That is, γ is of the form

$$a \ldots a.$$

Similarly, let b be a vertex on a discrete face, δ, in the embedding of H. So δ is of the form

$$b \cdots b.$$

Then the graph obtained by connecting G and H with a new edge, $\{a, b\}$, has a planar embedding whose discrete faces are the union of the discrete faces of G and

[20] There is a special case of this rule. If G is a line graph beginning with a and ending with b, then the cycles into which γ splits are actually the same. That's because adding edge $\{a, b\}$ creates a simple cycle graph, C_n, that divides the plane into an "inner" and an "outer" region with the same border. In order to maintain the correspondence between continuous faces and discrete faces, we have to allow two "copies" of this same cycle to count as discrete faces.

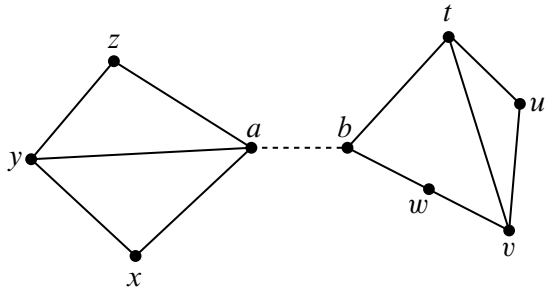


Figure 5.43 The "add a bridge" case.

H, except that faces γ and δ are replaced by one new face

$$a \ldots ab \cdots ba.$$

This is illustrated in Figure 5.43, where the faces of G and H are:

$$G : \{axyza,\ axya,\ ayza\} \qquad H : \{btuvwb,\ btvwb,\ tuvt\},$$

and after adding the bridge $\{a, b\}$, there is a single connected graph with faces

$$\{axyzabtuvwba,\ axya,\ ayza,\ btvwb,\ tuvt\}.$$

Does It Work?

Yes! In general, a graph is planar if and only if each of its connected components has a planar embedding as defined in Definition 5.8.2. Unfortunately, proving this fact requires a bunch of mathematics that we don't cover in this text—stuff like geometry and topology. Of course, that is why we went to the trouble of including Definition 5.8.2—we don't want to deal with that stuff in this text and now that we have a recursive definition for planar graphs, we won't need to. That's the good news.

The bad news is that Definition 5.8.2 looks a lot more complicated than the intuitively simple notion of a drawing where edges don't cross. It seems like it would be easier to stick to the simple notion and give proofs using pictures. Perhaps so, but your proofs are more likely to be complete and correct if you work from the discrete Definition 5.8.2 instead of the continuous Definition 5.8.1.

Where Did the Outer Face Go?

Every planar drawing has an immediately-recognizable outer face—its the one that goes to infinity in all directions. But where is the outer face in a planar embedding?

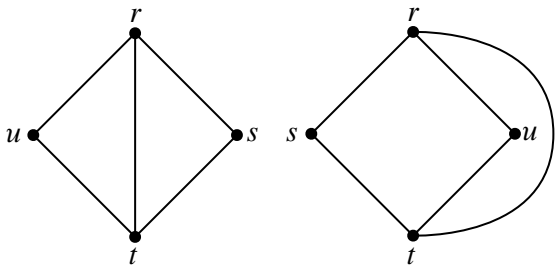


Figure 5.44 Two illustrations of the same embedding.

There isn't one! That's because there really isn't any need to distinguish one. In fact, a planar embedding could be drawn with any given face on the outside. An intuitive explanation of this is to think of drawing the embedding on a *sphere* instead of the plane. Then any face can be made the outside face by "puncturing" that face of the sphere, stretching the puncture hole to a circle around the rest of the faces, and flattening the circular drawing onto the plane.

So pictures that show different "outside" boundaries may actually be illustrations of the same planar embedding. For example, the two embeddings shown in Figure 5.44 are really the same.

This is what justifies the "add a bridge" case in Definition 5.8.2: whatever face is chosen in the embeddings of each of the disjoint planar graphs, we can draw a bridge between them without needing to cross any other edges in the drawing, because we can assume the bridge connects two "outer" faces.

5.8.3 Euler's Formula

The value of the recursive definition is that it provides a powerful technique for proving properties of planar graphs, namely, structural induction. For example, we will now use Definition 5.8.2 and structural induction to establish one of the most basic properties of a connected planar graph; namely, the number of vertices and edges completely determines the number of faces in every possible planar embedding of the graph.

Theorem 5.8.3 (Euler's Formula)**.** *If a connected graph has a planar embedding, then*

$$v - e + f = 2$$

where v is the number of vertices, e is the number of edges, and f is the number of faces.

For example, in Figure 5.38, $|V| = 4$, $|E| = 6$, and $f = 4$. Sure enough, $4 - 6 + 4 = 2$, as Euler's Formula claims.

Proof. The proof is by structural induction on the definition of planar embeddings. Let $P(\mathcal{E})$ be the proposition that $v - e + f = 2$ for an embedding, $\mathcal{E}$.

Base case: ($\mathcal{E}$ is the one-vertex planar embedding). By definition, $v = 1$, $e = 0$, and $f = 1$, so $P(\mathcal{E})$ indeed holds.

Constructor case (split a face): Suppose G is a connected graph with a planar embedding, and suppose a and b are distinct, nonadjacent vertices of G that appear on some discrete face, $\gamma = a \ldots b \cdots a$, of the planar embedding.

Then the graph obtained by adding the edge $\{a, b\}$ to the edges of G has a planar embedding with one more face and one more edge than G. So the quantity $v-e+f$ will remain the same for both graphs, and since by structural induction this quantity is 2 for G's embedding, it's also 2 for the embedding of G with the added edge. So P holds for the constructed embedding.

Constructor case (add bridge): Suppose G and H are connected graphs with planar embeddings and disjoint sets of vertices. Then connecting these two graphs with a bridge merges the two bridged faces into a single face, and leaves all other faces unchanged. So the bridge operation yields a planar embedding of a connected graph with $v_G + v_H$ vertices, $e_G + e_H + 1$ edges, and $f_G + f_H - 1$ faces. Since

$$\begin{aligned}
&(v_G + v_H) - (e_G + e_H + 1) + (f_G + f_H - 1) \\
&\quad = (v_G - e_G + f_G) + (v_H - e_H + f_H) - 2 \\
&\quad = (2) + (2) - 2 && \text{(by structural induction hypothesis)} \\
&\quad = 2,
\end{aligned}$$

$v - e + f$ remains equal to 2 for the constructed embedding. That is, $P(E)$ also holds in this case.

This completes the proof of the constructor cases, and the theorem follows by structural induction. ■

5.8.4 Bounding the Number of Edges in a Planar Graph

Like Euler's formula, the following lemmas follow by structural induction from Definition 5.8.2.

Lemma 5.8.4. *In a planar embedding of a connected graph, each edge is traversed once by each of two different faces, or is traversed exactly twice by one face.*

Lemma 5.8.5. *In a planar embedding of a connected graph with at least three vertices, each face is of length at least three.*

Combining Lemmas 5.8.4 and 5.8.5 with Euler's Formula, we can now prove that planar graphs have a limited number of edges:

Theorem 5.8.6. *Suppose a connected planar graph has $v \geq 3$ vertices and e edges. Then*

$$e \leq 3v - 6.$$

Proof. By definition, a connected graph is planar iff it has a planar embedding. So suppose a connected graph with v vertices and e edges has a planar embedding with f faces. By Lemma 5.8.4, every edge is traversed exactly twice by the face boundaries. So the sum of the lengths of the face boundaries is exactly $2e$. Also by Lemma 5.8.5, when $v \geq 3$, each face boundary is of length at least three, so this sum is at least $3f$. This implies that

$$3f \leq 2e. \tag{5.5}$$

But $f = e - v + 2$ by Euler's formula, and substituting into (5.5) gives

$$\begin{aligned} 3(e - v + 2) &\leq 2e \\ e - 3v + 6 &\leq 0 \\ e &\leq 3v - 6 \end{aligned}$$

■

5.8.5 Returning to K_5 and $K_{3,3}$

Theorem 5.8.6 lets us prove that the quadrapi can't all shake hands without crossing. Representing quadrapi by vertices and the necessary handshakes by edges, we get the complete graph, K_5. Shaking hands without crossing amounts to showing that K_5 is planar. But K_5 is connected, has 5 vertices and 10 edges, and $10 > 3 \cdot 5 - 6$. This violates the condition of Theorem 5.8.6 required for K_5 to be planar, which proves

Corollary 5.8.7. *K_5 is not planar.*

We can also use Euler's Formula to show that $K_{3,3}$ is not planar. The proof is similar to that of Theorem 5.8.6 except that we use the additional fact that $K_{3,3}$ is a bipartite graph.

Theorem 5.8.8. *$K_{3,3}$ is not planar.*

Proof. By contradiction. Assume $K_{3,3}$ is planar and consider any planar embedding of $K_{3,3}$ with f faces. Since $K_{3,3}$ is bipartite, we know by Theorem 5.6.2 that $K_{3,3}$ does not contain any closed walks of odd length. By Lemma 5.8.5, every face has length at least 3. This means that every face in any embedding of $K_{3,3}$ must have length at least 4. Plugging this fact into the proof of Theorem 5.8.6, we find that the sum of the lengths of the face boundaries is exactly $2e$ and at least $4f$. Hence,

$$4f \leq 2e$$

for any bipartite graph.

Plugging in $e = 9$ and $v = 6$ for $K_{3,3}$ in Euler's Formula, we find that

$$f = 2 + e - v = 5.$$

But

$$4 \cdot 5 \not\leq 2 \cdot 9,$$

and so we have a contradiction. Hence $K_{3,3}$ must not be planar. ■

5.8.6 Another Characterization for Planar Graphs

We did not choose to pick on K_5 and $K_{3,3}$ because of their application to dog houses or quadrapi shaking hands. Rather, we selected these graphs as examples because they provide another way to characterize the set of planar graphs.

Theorem 5.8.9 (Kuratowski)**.** *A graph is not planar if and only if it contains K_5 or $K_{3,3}$ as a minor.*

Definition 5.8.10. A *minor* of a graph G is a graph that can be obtained by repeatedly[21] deleting vertices, deleting edges, and merging *adjacent* vertices of G. *Merging* two adjacent vertices, n_1 and n_2 of a graph means deleting the two vertices and then replacing them by a new "merged" vertex, m, adjacent to all the vertices that were adjacent to either of n_1 or n_2, as illustrated in Figure 5.45.

For example, Figure 5.46 illustrates why C_3 is a minor of the graph in Figure 5.46(a). In fact C_3 is a minor of a connected graph G if and only if G is not a tree.

We will not prove Theorem 5.8.9 here, nor will we prove the following handy facts, which are obvious given the continuous Definition 5.8.1, and which can be proved using the recursive Definition 5.8.2.

Lemma 5.8.11. *Deleting an edge from a planar graph leaves another planar graph.*

Corollary 5.8.12. *Deleting a vertex from a planar graph, along with all its incident edges, leaves another planar graph.*

Theorem 5.8.13. *Any subgraph of a planar graph is planar.*

Theorem 5.8.14. *Merging two adjacent vertices of a planar graph leaves another planar graph.*

[21] The three operations can be performed in any order and in any quantities, or not at all.

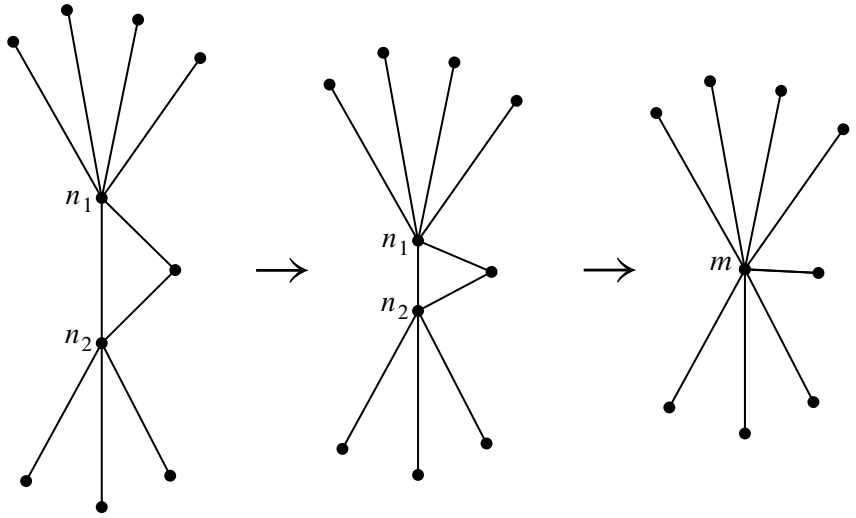


Figure 5.45 Merging adjacent vertices n_1 and n_2 into new vertex, m.

5.8.7 Coloring Planar Graphs

We've covered a lot of ground with planar graphs, but not nearly enough to prove the famous 4-color theorem. But we can get awfully close. Indeed, we have done almost enough work to prove that every planar graph can be colored using only 5 colors. We need only one more lemma:

Lemma 5.8.15. *Every planar graph has a vertex of degree at most five.*

Proof. By contradiction. If every vertex had degree at least 6, then the sum of the vertex degrees is at least $6v$, but since the sum of the vertex degrees equals $2e$, by the Handshake Lemma (Lemma 5.2.1), we have $e \geq 3v$ contradicting the fact that $e \leq 3v - 6 < 3v$ by Theorem 5.8.6. ■

Theorem 5.8.16. *Every planar graph is five-colorable.*

Proof. The proof will be by strong induction on the number, v, of vertices, with induction hypothesis:

> Every planar graph with v vertices is five-colorable.

Base cases ($v \leq 5$): immediate.

Inductive case: Suppose G is a planar graph with $v + 1$ vertices. We will describe a five-coloring of G.

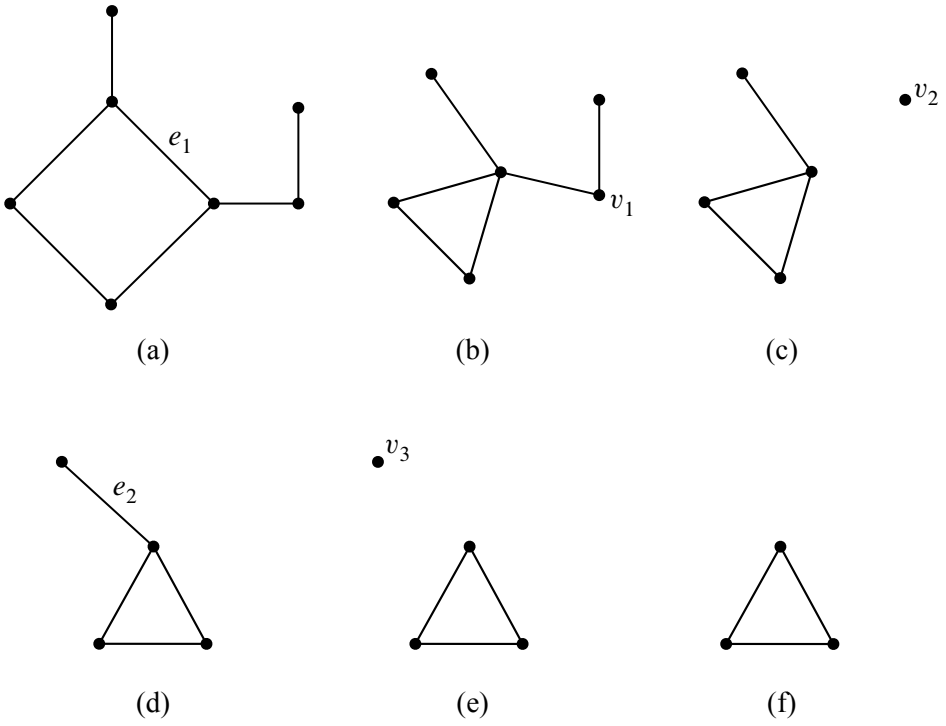


Figure 5.46 One method by which the graph in (a) can be reduced to C_3 (f), thereby showing that C_3 is a minor of the graph. The steps are: merging the nodes incident to e_1 (b), deleting v_1 and all edges incident to it (c), deleting v_2 (d), deleting e_2, and deleting v_3 (f).

First, choose a vertex, g, of G with degree at most 5; Lemma 5.8.15 guarantees there will be such a vertex.

Case 1: ($\deg(g) < 5$): Deleting g from G leaves a graph, H, that is planar by Corollary 5.8.12, and, since H has v vertices, it is five-colorable by induction hypothesis. Now define a five coloring of G as follows: use the five-coloring of H for all the vertices besides g, and assign one of the five colors to g that is not the same as the color assigned to any of its neighbors. Since there are fewer than 5 neighbors, there will always be such a color available for g.

Case 2: ($\deg(g) = 5$): If the five neighbors of g in G were all adjacent to each other, then these five vertices would form a nonplanar subgraph isomorphic to K_5, contradicting Theorem 5.8.13 (since K_5 is not planar). So there must be two neighbors, n_1 and n_2, of g that are not adjacent. Now merge n_1 and g into a new vertex, m. In this new graph, n_2 is adjacent to m, and the graph is planar by Theorem 5.8.14. So we can then merge m and n_2 into a another new vertex, m', resulting in a new graph, G', which by Theorem 5.8.14 is also planar. Since G' has $v - 1$ vertices, it is five-colorable by the induction hypothesis.

Define a five coloring of G as follows: use the five-coloring of G' for all the vertices besides g, n_1 and n_2. Next assign the color of m' in G' to be the color of the neighbors n_1 and n_2. Since n_1 and n_2 are not adjacent in G, this defines a proper five-coloring of G except for vertex g. But since these two neighbors of g have the same color, the neighbors of g have been colored using fewer than five colors altogether. So complete the five-coloring of G by assigning one of the five colors to g that is not the same as any of the colors assigned to its neighbors.

■

5.8.8 Classifying Polyhedra

The Pythagoreans had two great mathematical secrets, the irrationality of $\sqrt{2}$ and a geometric construct that we're about to rediscover!

A *polyhedron* is a convex, three-dimensional region bounded by a finite number of polygonal faces. If the faces are identical regular polygons and an equal number of polygons meet at each corner, then the polyhedron is *regular*. Three examples of regular polyhedra are shown in Figure 5.34: the tetrahedron, the cube, and the octahedron.

We can determine how many more regular polyhedra there are by thinking about planarity. Suppose we took *any* polyhedron and placed a sphere inside it. Then we could project the polyhedron face boundaries onto the sphere, which would give an image that was a planar graph embedded on the sphere, with the images of the

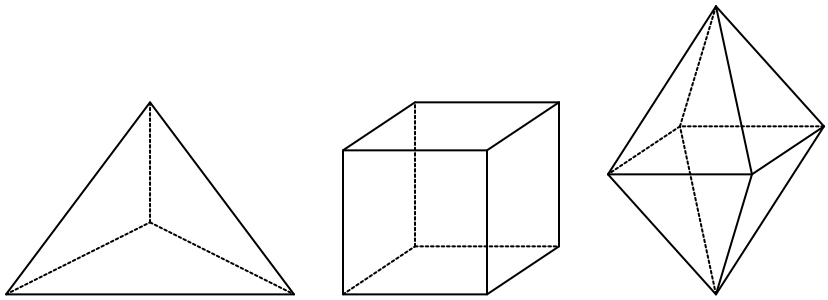


Figure 5.47 The tetrahedron (a), cube (b), and octahedron (c).

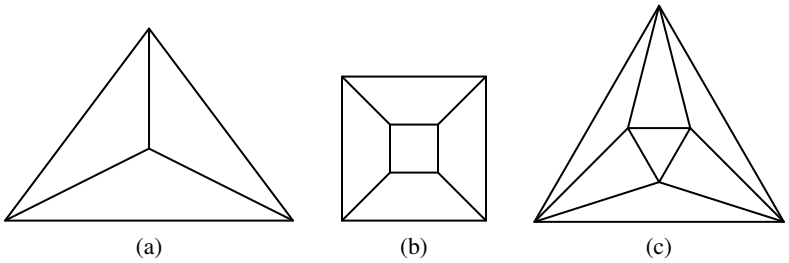


Figure 5.48 Planar embeddings of the tetrahedron (a), cube (b, and octahedron (c).

corners of the polyhedron corresponding to vertices of the graph. We've already observed that embeddings on a sphere are the same as embeddings on the plane, so Euler's formula for planar graphs can help guide our search for regular polyhedra.

For example, planar embeddings of the three polyhedra in Figure 5.34 are shown in Figure 5.48.

Let m be the number of faces that meet at each corner of a polyhedron, and let n be the number of edges on each face. In the corresponding planar graph, there are m edges incident to each of the v vertices. By the Handshake Lemma 5.2.1, we know:

$$mv = 2e.$$

Also, each face is bounded by n edges. Since each edge is on the boundary of two faces, we have:

$$nf = 2e$$

Solving for v and f in these equations and then substituting into Euler's formula

n	m	v	e	f	polyhedron
3	3	4	6	4	tetrahedron
4	3	8	12	6	cube
3	4	6	12	8	octahedron
3	5	12	30	20	icosahedron
5	3	20	30	12	dodecahedron

Figure 5.49 The only possible regular polyhedra.

gives:

$$\frac{2e}{m} - e + \frac{2e}{n} = 2$$

which simplifies to

$$\frac{1}{m} + \frac{1}{n} = \frac{1}{e} + \frac{1}{2} \tag{5.6}$$

Equation 5.6 places strong restrictions on the structure of a polyhedron. Every nondegenerate polygon has at least 3 sides, so $n \geq 3$. And at least 3 polygons must meet to form a corner, so $m \geq 3$. On the other hand, if either n or m were 6 or more, then the left side of the equation could be at most $1/3 + 1/6 = 1/2$, which is less than the right side. Checking the finitely-many cases that remain turns up only five solutions, as shown in Figure 5.49. For each valid combination of n and m, we can compute the associated number of vertices v, edges e, and faces f. And polyhedra with these properties do actually exist. The largest polyhedron, the dodecahedron, was the other great mathematical secret of the Pythagorean sect.

The 5 polyhedra in Figure 5.49 are the only possible regular polyhedra. So if you want to put more than 20 geocentric satellites in orbit so that they *uniformly* blanket the globe—tough luck!

6 Directed Graphs

6.1 Definitions

So far, we have been working with graphs with undirected edges. A *directed edge* is an edge where the endpoints are distinguished—one is the *head* and one is the *tail*. In particular, a directed edge is specified as an ordered pair of vertices u, v and is denoted by (u, v) or $u \to v$. In this case, u is the *tail* of the edge and v is the *head*. For example, see Figure 6.1.

A graph with directed edges is called a *directed graph* or *digraph*.

Definition 6.1.1. A directed graph $G = (V, E)$ consists of a nonempty set of nodes V and a set of directed edges E. Each edge e of E is specified by an ordered pair of vertices $u, v \in V$. A directed graph is *simple* if it has no *loops* (that is, edges of the form $u \to u$) and no multiple edges.

Since we will focus on the case of simple directed graphs in this chapter, we will generally omit the word *simple* when referring to them. Note that such a graph can contain an edge $u \to v$ as well as the edge $v \to u$ since these are different edges (for example, they have a different tail).

Directed graphs arise in applications where the relationship represented by an edge is 1-way or asymmetric. Examples include: a 1-way street, one person likes another but the feeling is not necessarily reciprocated, a communication channel such as a cable modem that has more capacity for downloading than uploading, one entity is larger than another, and one job needs to be completed before another job can begin. We'll see several such examples in this chapter and also in Chapter 7.

Most all of the definitions for undirected graphs from Chapter 5 carry over in a natural way for directed graphs. For example, two directed graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there exists a bijection $f : V_1 \to V_2$ such that for every pair of vertices $u, v \in V_1$,

$$u \to v \in E_1 \qquad \text{IFF} \qquad f(u) \to f(v) \in E_2.$$

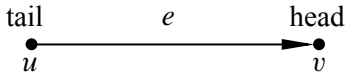


Figure 6.1 A directed edge $e = (u, v)$. u is the tail of e and v is the head of e.

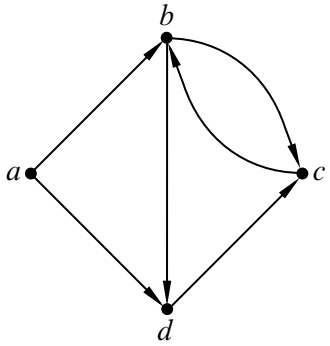


Figure 6.2 A 4-node directed graph with 6 edges.

Directed graphs have adjacency matrices just like undirected graphs. In the case of a directed graph $G = (V, E)$, the adjacency matrix $A_G = \{a_{ij}\}$ is defined so that

$$a_{ij} = \begin{cases} 1 & \text{if } i \to j \in E \\ 0 & \text{otherwise.} \end{cases}$$

The only difference is that the adjacency matrix for a directed graph is not necessarily symmetric (that is, it may be that $A_G^T \neq A_G$).

6.1.1 Degrees

With directed graphs, the notion of degree splits into *indegree* and *outdegree*. For example, indegree$(c) = 2$ and outdegree$(c) = 1$ for the graph in Figure 6.2. If a node has outdegree 0, it is called a *sink*; if it has indegree 0, it is called a *source*. The graph in Figure 6.2 has one source (node a) and no sinks.

6.1.2 Directed Walks, Paths, and Cycles

The definitions for (directed) walks, paths, and cycles in a directed graph are similar to those for undirected graphs except that the direction of the edges need to be consistent with the order in which the walk is traversed.

Definition 6.1.2. A *directed walk* (or more simply, a *walk*) in a directed graph G is a sequence of vertices $v_0, v_1, \ldots, v_k$ and edges

$$v_0 \to v_1, v_1 \to v_2, \ldots, v_{k-1} \to v_k$$

such that $v_{i-1} \to v_i$ is an edge of G for all i where $0 \leq i < k$. A *directed path* (or *path*) in a directed graph is a walk where the nodes in the walk are all different. A *directed closed walk* (or *closed walk*) in a directed graph is a walk

where $v_0 = v_k$. A *directed cycle* (or *cycle*) in a directed graph is a closed walk where all the vertices v_i are different for $0 \leq i < k$.

As with undirected graphs, we will typically refer to a walk in a directed graph by a sequence of vertices. For example, for the graph in Figure 6.2,

- a, b, c, b, d is a walk,
- a, b, d is a path,
- d, c, b, c, b, d is a closed walk, and
- b, d, c, b is a cycle.

Note that b, c, b is also a cycle for the graph in Figure 6.2. This is a cycle of length 2. Such cycles are not possible with undirected graphs.

Also note that

$$c, b, a, d$$

is *not* a walk in the graph shown in Figure 6.2, since $b \to a$ is not an edge in this graph. (You are *not* allowed to traverse edges in the wrong direction as part of a walk.)

A path or cycle in a directed graph is said to be *Hamiltonian* if it visits every node in the graph. For example, a, b, d, c is the only Hamiltonian path for the graph in Figure 6.2. The graph in Figure 6.2 does not have a Hamiltonian cycle.

A walk in a directed graph is said to be *Eulerian* if it contains every edge. The graph shown in Figure 6.2 does not have an Eulerian walk. Can you see why not? (Hint: Look at node a.)

6.1.3 Strong Connectivity

The notion of being connected is a little more complicated for a directed graph than it is for an undirected graph. For example, should we consider the graph in Figure 6.2 to be connected? There is a path from node a to every other node so on that basis, we might answer "Yes." But there is no path from nodes b, c, or d to node a, and so on that basis, we might answer "No." For this reason, graph theorists have come up with the notion of *strong* connectivity for directed graphs.

Definition 6.1.3. A directed graph $G = (V, E)$ is said to be *strongly connected* if for every pair of nodes $u, v \in V$, there is a directed path from u to v (and vice-versa) in G.

For example, the graph in Figure 6.2 is not strongly connected since there is no directed path from node b to node a. But if node a is removed, the resulting graph would be strongly connected.

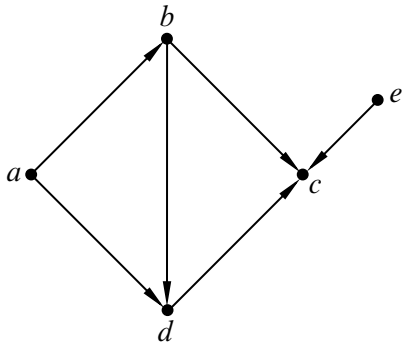


Figure 6.3 A 4-node directed acyclic graph (DAG).

A directed graph is said to be *weakly connected* (or, more simply, *connected*) if the corresponding undirected graph (where directed edges $u \to v$ and/or $v \to u$ are replaced with a single undirected edge $\{u, v\}$ is connected. For example, the graph in Figure 6.2 is weakly connected.

6.1.4 DAGs

If an undirected graph does not have any cycles, then it is a tree or a forest. But what does a directed graph look like if it has no cycles? For example, consider the graph in Figure 6.3. This graph is weakly connected and has no directed cycles but it certainly does not look like a tree.

Definition 6.1.4. A directed graph is called a *directed acyclic graph* (or, *DAG*) if it does not contain any directed cycles.

A first glance, DAGs don't appear to be particularly interesting. But first impressions are not always accurate. In fact, DAGs arise in many scheduling and optimization problems and they have several interesting properties. We will study them extensively in Chapter 7.

6.2 Tournament Graphs

Suppose that n players compete in a round-robin tournament and that for every pair of players u and v, either u beats v or v beats u. Interpreting the results of a round-robin tournament can be problematic—there might be all sorts of cycles where x beats y and y beats z, yet z beats x. Who is the best player? Graph theory does not solve this problem but it can provide some interesting perspectives.

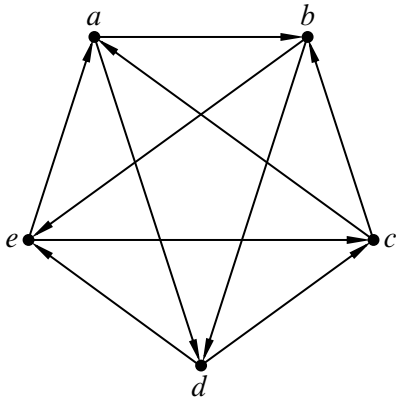


Figure 6.4 A 5-node tournament graph.

The results of a round-robin tournament can be represented with a *tournament graph*. This is a directed graph in which the vertices represent players and the edges indicate the outcomes of games. In particular, an edge from u to v indicates that player u defeated player v. In a round-robin tournament, every pair of players has a match. Thus, in a tournament graph there is either an edge from u to v or an edge from v to u (but not both) for *every* pair of distinct vertices u and v. An example of a tournament graph is shown in Figure 6.4.

6.2.1 Finding a Hamiltonian Path in a Tournament Graph

We're going to prove that in every round-robin tournament, there exists a ranking of the players such that each player lost to the player one position higher. For example, in the tournament corresponding to Figure 6.4, the ranking

$$a > b > d > e > c$$

satisfies this criterion, because b lost to a, d lost to b, e lost to d, and c lost to e. In graph terms, proving the existence of such a ranking amounts to proving that every tournament graph has a Hamiltonian path.

Theorem 6.2.1. *Every tournament graph contains a directed Hamiltonian path.*

Proof. We use strong induction. Let $P(n)$ be the proposition that every tournament graph with n vertices contains a directed Hamiltonian path.

Base case: $P(1)$ is trivially true; every graph with a single vertex has a Hamiltonian path consisting of only that vertex.

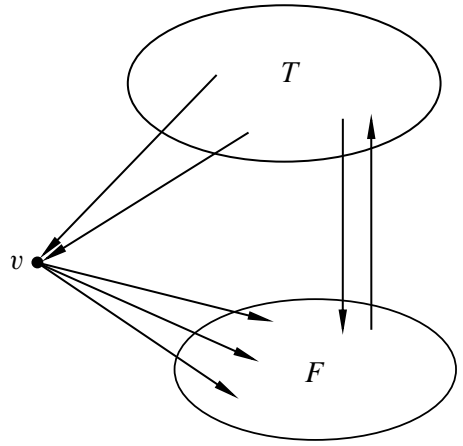


Figure 6.5 The sets T and F in a tournament graph.

Inductive step: For $n \geq 1$, we assume that $P(1), \ldots, P(n)$ are all true and prove $P(n+1)$. Consider a tournament graph $G = (V, E)$ with $n+1$ players. Select one vertex v arbitrarily. Every other vertex in the tournament either has an edge *to* vertex v or an edge *from* vertex v. Thus, we can partition the remaining vertices into two corresponding sets, T and F, each containing at most n vertices, where $T = \{ u \mid u \to v \in E \}$ and $F = \{ u \mid v \to u \in E \}$. For example, see Figure 6.5.

The vertices in T together with the edges that join them form a smaller tournament. Thus, by strong induction, there is a Hamiltonian path within T. Similarly, there is a Hamiltonian path within the tournament on the vertices in F. Joining the path in T to the vertex v followed by the path in F gives a Hamiltonian path through the whole tournament. As special cases, if T or F is empty, then so is the corresponding portion of the path. ■

The ranking defined by a Hamiltonian path is not entirely satisfactory. For example, in the tournament associated with Figure 6.4, notice that the lowest-ranked player, c, actually defeated the highest-ranked player, a.

In practice, players are typically ranked according to how many victories they achieve. This makes sense for several reasons. One not-so-obvious reason is that if the player with the most victories does not beat some other player v, he is guaranteed to have at least beaten a third player who beat v. We'll prove this fact shortly. But first, let's talk about chickens.

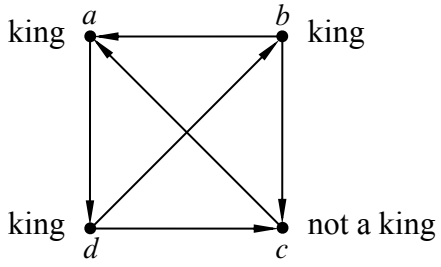


Figure 6.6 A 4-chicken tournament in which chickens a, b, and d are kings.

.

6.2.2 The King Chicken Theorem

Suppose that there are n chickens in a farmyard. Chickens are rather aggressive birds that tend to establish dominance in relationships by pecking. (Hence the term "pecking order.") In particular, for each pair of distinct chickens, either the first pecks the second or the second pecks the first, but not both. We say that chicken u *virtually pecks* chicken v if either:

- Chicken u directly pecks chicken v, or
- Chicken u pecks some other chicken w who in turn pecks chicken v.

A chicken that virtually pecks every other chicken is called a *king chicken*.

We can model this situation with a tournament digraph. The vertices are chickens, and an edge $u \to v$ indicates that chicken u pecks chicken v. In the tournament shown in Figure 6.6, three of the four chickens are kings. Chicken c is not a king in this example since it does not peck chicken b and it does not peck any chicken that pecks chicken b. Chicken a *is* a king since it pecks chicken d, who in turn pecks chickens b and c.

Theorem 6.2.2 (King Chicken Theorem)**.** *The chicken with the largest outdegree in an n-chicken tournament is a king.*

Proof. By contradiction. Let u be a node in a tournament graph $G = (V, E)$ with maximum outdegree and suppose that u is not a king. Let $Y = \{ v \mid u \to v \in E \}$ be the set of chickens that chicken u pecks. Then $\text{outdegree}(u) = |Y|$.

Since u is not a king, there is a chicken $x \notin Y$ (that is, x is not pecked by chicken u) and that is not pecked by any chicken in Y. Since for any pair of chickens, one pecks the other, this means that x pecks u as well as every chicken in Y. This means that

$$\text{outdegree}(x) = |Y| + 1 > \text{outdegree}(u).$$

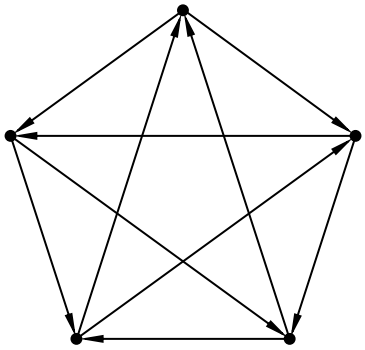


Figure 6.7 A 5-chicken tournament in which every chicken is a king.

But u was assumed to be the node with the largest degree in the tournament, so we have a contradiction. Hence, u must be a king. ■

Theorem 6.2.2 means that if the player with the most victories is defeated by another player x, then at least he/she defeats some third player that defeats x. In this sense, the player with the most victories has some sort of bragging rights over every other player. Unfortunately, as Figure 6.6 illustrates, there can be many other players with such bragging rights, even some with fewer victories. Indeed, for some tournaments, it is possible that every player is a "king." For example, consider the tournament illustrated in Figure 6.7.

6.3 Communication Networks

While reasoning about chickens pecking each other may be amusing (to mathematicians, at least), the use of directed graphs to model communication networks is very serious business. In the context of communication problems, vertices represent computers, processors, or switches, and edges represent wires, fiber, or other transmission lines through which data flows. For some communication networks, like the Internet, the corresponding graph is enormous and largely chaotic. Highly structured networks, such as an array or butterfly, by contrast, find application in telephone switching systems and the communication hardware inside parallel computers.

6.3.1 Packet Routing

Whatever architecture is chosen, the goal of a communication network is to get data from *inputs* to *outputs*. In this text, we will focus on a model in which the data to be communicated is in the form of a *packet*. In practice, a packet would consist of a fixed amount of data, and a message (such as a web page or a movie) would consist of many packets.

For simplicity, we will restrict our attention to the scenario where there is just one packet at every input and where there is just one packet destined for each output. We will denote the number of inputs and output by N and we will often assume that N is a power of two.

We will specify the desired destinations of the packets by a permutation[1] of 0, 1, ..., $N-1$. So a permutation, π, defines a *routing problem*: get a packet that starts at input i to output $\pi(i)$ for $0 \leq i < N$. A *routing* P that *solves* a routing problem π is a set of paths from each input to its specified output. That is, P is a set of paths, P_i, for $i = 0, \ldots, N-1$, where P_i goes from input i to output $\pi(i)$.

Of course, the goal is to get all the packets to their destinations as quickly as possible using as little hardware as possible. The time needed to get the packages to their destinations depends on several factors, such as how many switches they need to go through and how many packets will need to cross the same wire. We will assume that only one packet can cross a wire at a time. The complexity of the hardware depends on factors such as the number of switches needed and the size of the switches.

Let's see how all this works with an example—routing packets on a complete binary tree.

6.3.2 The Complete Binary Tree

One of the simplest structured communications networks is a *complete binary tree*. A complete binary tree with 4 inputs and 4 outputs is shown in Figure 6.8.

In this diagram and many that follow, the squares represent *terminals* (that is, the inputs and outputs), and the circles represent *switches*, which direct packets through the network. A switch receives packets on incoming edges and relays them forward along the outgoing edges. Thus, you can imagine a data packet hopping through the network from an input terminal, through a sequence of switches joined by directed edges, to an output terminal.

Recall that there is a unique simple path between every pair of vertices in a tree. So the natural way to route a packet of data from an input terminal to an output terminal in the complete binary tree is along the corresponding directed path. For

[1] A permutation of a sequence is a reordering of the sequence.

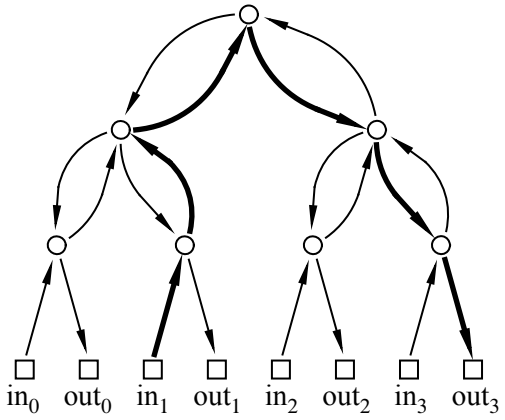


Figure 6.8 A 4-input, 4-output complete binary tree. The squares represent terminals (input and output registers) and the circles represent switches. Directed edges represent communication channels in the network through which data packets can move. The unique path from input 1 to output 3 is shown in bold.

example, the route of a packet traveling from input 1 to output 3 is shown in bold in Figure 6.8.

6.3.3 Network Diameter

The delay between the time that a packet arrives at an input and the time that it reaches its designated output is referred to as *latency* and it is a critical issue in communication networks. If congestion is not a factor, then this delay is generally proportional to the length of the path a packet follows. Assuming it takes one time unit to travel across a wire, and that there are no additional delays at switches, the delay of a packet will be the number of wires it crosses going from input to output.[2]

Generally a packet is routed from input to output using the shortest path possible. The length of this shortest path is the *distance* between the input and output. With a shortest path routing, the worst possible delay is the distance between the input and output that are farthest apart. This is called the *diameter* of the network. In other words, the diameter of a network[3] is the maximum length of any shortest

[2]Latency can also be measured as the number of switches that a packet must pass through when traveling between the most distant input and output, since switches usually have the biggest impact on network speed. For example, in the complete binary tree example, the packet traveling from input 1 to output 3 crosses 5 switches, which is 1 less than the number of edges traversed.

[3]The usual definition of *diameter* for a general graph (simple or directed) is the largest distance between *any* two vertices, but in the context of a communication network, we're only interested in the distance between inputs and outputs, not between arbitrary pairs of vertices.

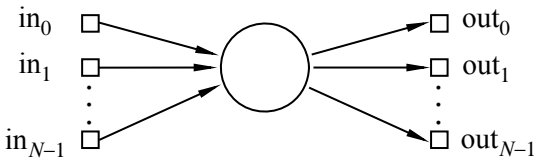


Figure 6.9 A monster $N \times N$ switch.

path between an input and an output. For example, in the complete binary tree shown in Figure 6.8, the distance from input 1 to output 3 is six. No input and output are farther apart than this, so the diameter of this tree is also six.

More generally, the diameter of a complete binary tree with N inputs and outputs is $2 \log N + 2$. (All logarithms in this lecture—and in most of computer science—are base 2.) This is quite good, because the logarithm function grows very slowly. We could connect $2^{20} = 1{,}048{,}576$ inputs and outputs using a complete binary tree and the worst input-output delay for any packet would be this diameter, namely, $2 \log(2^{20}) + 2 = 42$.

6.3.4 Switch Size

One way to reduce the diameter of a network (and hence the latency needed to route packets) is to use larger switches. For example, in the complete binary tree, most of the switches have three incoming edges and three outgoing edges, which makes them 3×3 switches. If we had 4×4 switches, then we could construct a complete *ternary* tree with an even smaller diameter. In principle, we could even connect up all the inputs and outputs via a single monster $N \times N$ switch, as shown in Figure 6.9. In this case, the "network" would consist of a single switch and the latency would be 2.

This isn't very productive, however, since we've just concealed the original network design problem inside this abstract monster switch. Eventually, we'll have to design the internals of the monster switch using simpler components, and then we're right back where we started. So the challenge in designing a communication network is figuring out how to get the functionality of an $N \times N$ switch using fixed size, elementary devices, like 3×3 switches.

6.3.5 Switch Count

Another goal in designing a communication network is to use as few switches as possible. The number of switches in a complete binary tree is $1 + 2 + 4 + 8 + \cdots + N = 2N - 1$, since there is 1 switch at the top (the "root switch"), 2 below it, 4 below those, and so forth. This is nearly the best possible with 3×3 switches,

since at least one switch will be needed for each pair of inputs and outputs.

6.3.6 Congestion

The complete binary tree has a fatal drawback: the root switch is a bottleneck. At best, this switch must handle an enormous amount of traffic: every packet traveling from the left side of the network to the right or vice-versa. Passing all these packets through a single switch could take a long time. At worst, if this switch fails, the network is broken into two equal-sized pieces.

The traffic through the root depends on the routing problem. For example, if the routing problem is given by the identity permutation, $\pi(i) ::= i$, then there is an easy routing P that solves the problem: let P_i be the path from input i up through one switch and back down to output i. On the other hand, if the problem was given by $\pi(i) ::= (N-1) - i$, then in *any* solution P for π, each path P_i beginning at input i must eventually loop all the way up through the root switch and then travel back down to output $(N-1) - i$.

We can distinguish between a "good" set of paths and a "bad" set based on congestion. The *congestion* of a routing, P, is equal to the largest number of paths in P that pass through a single switch. Generally, lower congestion is better since packets can be delayed at an overloaded switch.

By extending the notion of congestion to networks, we can also distinguish between "good" and "bad" networks with respect to bottleneck problems. For each routing problem, π, for the network, we assume a routing is chosen that optimizes congestion, that is, that has the minimum congestion among all routings that solve π. Then the largest congestion that will ever be suffered by a switch will be the maximum congestion among these optimal routings. This "maxi-min" congestion is called the *congestion of the network*.

You may find it helpful to think about max congestion in terms of a value game. You design your spiffy, new communication network; this defines the game. Your opponent makes the first move in the game: she inspects your network and specifies a permutation routing problem that will strain your network. You move second: given her specification, you choose the precise paths that the packets should take through your network; you're trying to avoid overloading any one switch. Then her next move is to pick a switch with as large as possible a number of packets passing through it; this number is her score in the competition. The max congestion of your network is the largest score she can ensure; in other words, it is precisely the max-value of this game.

For example, if your enemy were trying to defeat the complete binary tree, she would choose a permutation like $\pi(i) = (N-1) - i$. Then for *every* packet i, you would be forced to select a path $P_{i,\pi(i)}$ passing through the root switch. Then, your

network	diameter	switch size	# switches	congestion
complete binary tree	$2\log N + 2$	3×3	$2N - 1$	N

Table 6.1 A summary of the attributes of the complete binary tree.

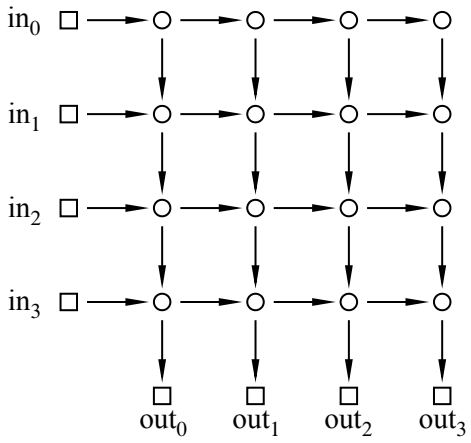


Figure 6.10 A 4×4 2-dimensional array.

enemy would choose the root switch and achieve a score of N. In other words, the max congestion of the complete binary tree is N—which is horrible!

We have summarized the results of our analysis of the complete binary tree in Table 6.1. Overall, the complete binary tree does well in every category except the last—congestion, and that is a killer in practice. Next, we will look at a network that solves the congestion problem, but at a very high cost.

6.3.7 The 2-d Array

An illustration of the $N \times N$ 2-d *array* (also known as the *grid* or *crossbar*) is shown in Figure 6.10 for the case when $N = 4$.

The diameter of the 4×4 2-d array is 8, which is the number of edges between input 0 and output 3. More generally, the diameter of a 2-d array with N inputs and outputs is $2N$, which is much worse than the diameter of the complete binary tree ($2\log N + 2$). On the other hand, replacing a complete binary tree with a 2-d array almost eliminates congestion.

Theorem 6.3.1. *The congestion of an N-input 2-d array is 2.*

Proof. First, we show that the congestion is at most 2. Let π be any permutation. Define a solution, P, for π to be the set of paths, P_i, where P_i goes to the right

network	diameter	switch size	# switches	congestion
complete binary tree	$2\log N + 2$	3×3	$2N - 1$	N
2-D array	$2N$	2×2	N^2	2

Table 6.2 Comparing the N-input 2-d array to the N-input complete binary tree.

from input i to column $\pi(i)$ and then goes down to output $\pi(i)$. In this solution, the switch in row i and column j encounters at most two packets: the packet originating at input i and the packet destined for output j.

Next, we show that the congestion is at least 2. This follows because in any routing problem, π, where $\pi(0) = 0$ and $\pi(N-1) = N-1$, two packets must pass through the lower left switch. ■

The characteristics of the 2-d array are recorded in Table 6.2. The crucial entry in this table is the number of switches, which is N^2. This is a major defect of the 2-d array; a network with $N = 1000$ inputs would require a *million* 2×2 switches! Still, for applications where N is small, the simplicity and low congestion of the array make it an attractive choice.

6.3.8 The Butterfly

The Holy Grail of switching networks would combine the best properties of the complete binary tree (low diameter, few switches) and the array (low congestion). The *butterfly* is a widely-used compromise between the two. A butterfly network with $N = 8$ inputs is shown in Figure 6.11.

The structure of the butterfly is certainly more complicated than that of the complete binary or 2-d array. Let's see how it is constructed.

All the terminals and switches in the network are in N rows. In particular, input i is at the left end of row i, and output i is at the right end of row i. Now let's label the rows in *binary* so that the label on row i is the binary number $b_1 b_2 \ldots b_{\log N}$ that represents the integer i.

Between the inputs and outputs, there are $\log(N) + 1$ levels of switches, numbered from 0 to $\log N$. Each level consists of a column of N switches, one per row. Thus, each switch in the network is uniquely identified by a sequence $(b_1, b_2, \ldots, b_{\log N}, l)$, where $b_1 b_2 \ldots b_{\log N}$ is the switch's row in binary and l is the switch's level.

All that remains is to describe how the switches are connected up. The basic

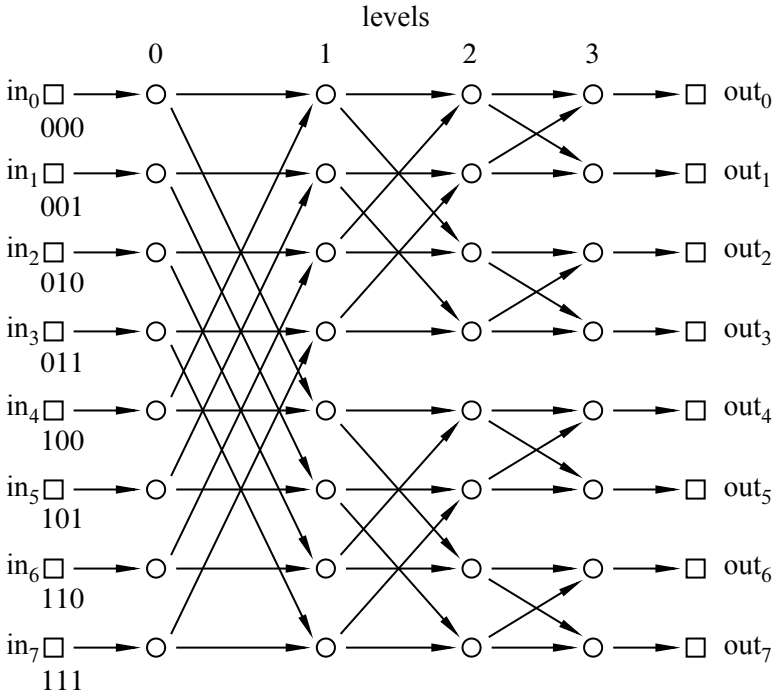


Figure 6.11 An 8-input/output butterfly.

connection pattern is expressed below in a compact notation:

$$(b_1, b_2, \ldots b_{l+1}, \ldots b_{\log N}, l) \begin{array}{l} \nearrow \\ \searrow \end{array} \begin{array}{l} (b_1, b_2, \ldots b_{l+1}, \ldots b_{\log N}, l+1) \\ (b_1, b_2, \ldots \overline{b_{l+1}}, \ldots b_{\log N}, l+1) \end{array}$$

This says that there are directed edges from switch $(b_1, b_2, \ldots, b_{\log N}, l)$ to two switches in the next level. One edges leads to the switch in the *same* row, and the other edge leads to the switch in the row obtained by *inverting* the $(l+1)$st bit b_{l+1}. For example, referring back to the illustration of the size $N = 8$ butterfly, there is an edge from switch $(0, 0, 0, 0)$ to switch $(0, 0, 0, 1)$, which is in the same row, and to switch $(1, 0, 0, 1)$, which is in the row obtained by inverting bit $l + 1 = 1$.

The butterfly network has a recursive structure; specifically, a butterfly of size $2N$ consists of two butterflies of size N and one additional level of switches. Each switch in the additional level has directed edges to a corresponding switch in each of the smaller butterflies. For example, see Figure 6.12.

Despite the relatively complicated structure of the butterfly, there is a simple way to route packets through its switches. In particular, suppose that we want to send a packet from input $x_1 x_2 \ldots x_{\log N}$ to output $y_1 y_2 \ldots y_{\log N}$. (Here we are specifying the input and output numbers in binary.) Roughly, the plan is to "correct" the first bit on the first level, correct the second bit on the second level, and so forth. Thus, the sequence of switches visited by the packet is:

$$\begin{aligned} (x_1, x_2, x_3, \ldots, x_{\log N}, 0) &\to (y_1, x_2, x_3, \ldots, x_{\log N}, 1) \\ &\to (y_1, y_2, x_3, \ldots, x_{\log N}, 2) \\ &\to (y_1, y_2, y_3, \ldots, x_{\log N}, 3) \\ &\to \quad \ldots \\ &\to (y_1, y_2, y_3, \ldots, y_{\log N}, \log N) \end{aligned}$$

In fact, this is the *only* path from the input to the output!

The congestion of the butterfly network is about $\sqrt{N}$. More precisely, the congestion is $\sqrt{N}$ if N is an even power of 2 and $\sqrt{N/2}$ if N is an odd power of 2. The task of proving this fact has been left to the problem section.[4]

A comparison of the butterfly with the complete binary tree and the 2-d array is provided in Table 6.3. As you can see, the butterfly has lower congestion than the complete binary tree. And it uses fewer switches and has lower diameter than the

[4]The routing problems that result in $\sqrt{N}$ congestion do arise in practice, but for most routing problems, the congestion is much lower (around $\log N$), which is one reason why the butterfly is useful in practice.

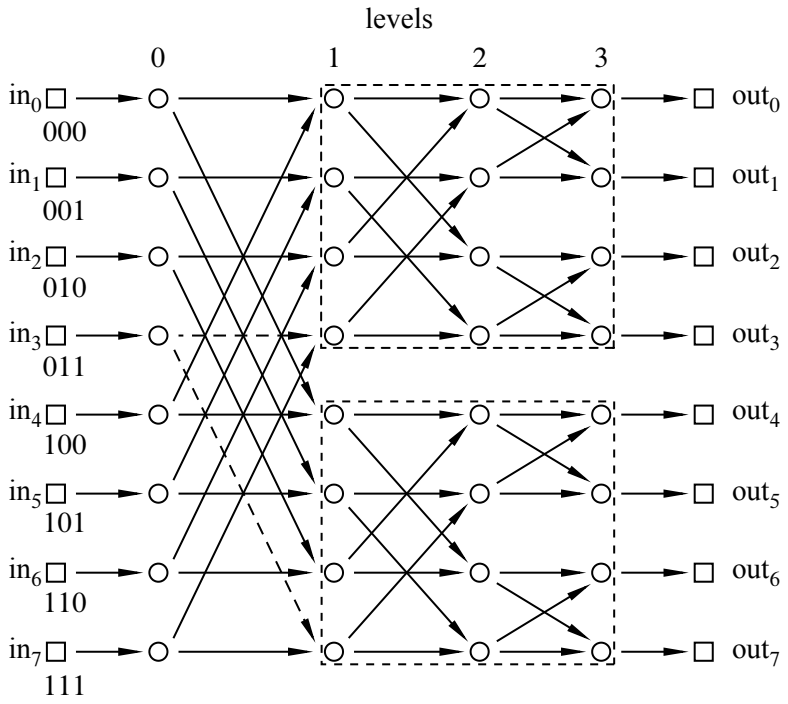


Figure 6.12 An N-input butterfly contains two $N/2$-input butterflies (shown in the dashed boxes). Each switch on the first level is adjacent to a corresponding switch in each of the sub-butterflies. For example, we have used dashed lines to show these edges for the node $(0, 1, 1, 0)$.

network	diameter	switch size	# switches	congestion
complete binary tree	$2\log N + 2$	3×3	$2N - 1$	N
2-D array	$2N$	2×2	N^2	2
butterfly	$\log N + 2$	2×2	$N(\log(N) + 1)$	$\sqrt{N}$ or $\sqrt{N/2}$

Table 6.3 A comparison of the N-input butterfly with the N-input complete binary tree and the N-input 2-d array.

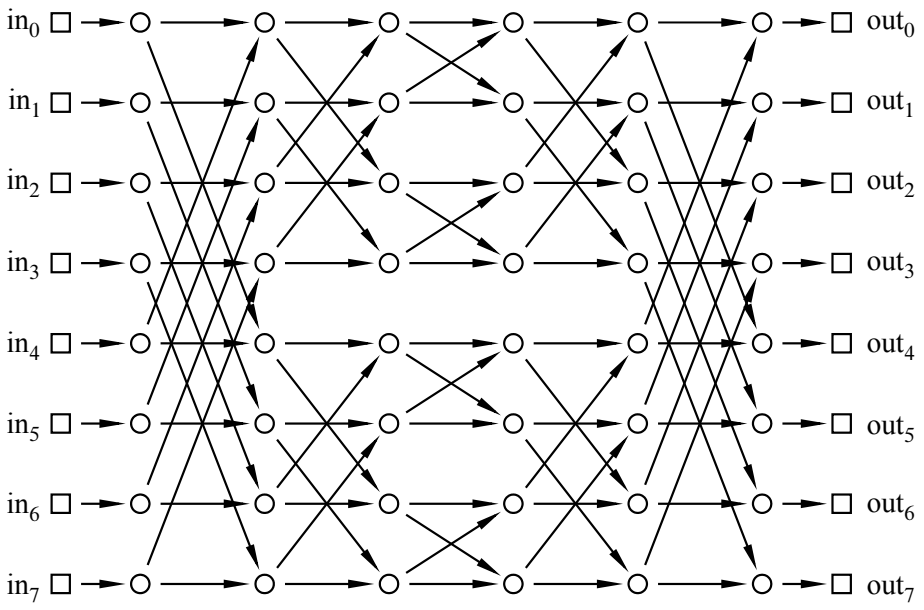


Figure 6.13 The 8-input Beneš network.

array. However, the butterfly does not capture the best qualities of each network, but rather is a compromise somewhere between the two. So our quest for the Holy Grail of routing networks goes on.

6.3.9 Beneš Network

In the 1960's, a researcher at Bell Labs named Václav Beneš had a remarkable idea. He obtained a marvelous communication network with congestion 1 by placing *two* butterflies back-to-back. For example, the 8-input Beneš network is shown in Figure 6.13.

Putting two butterflies back-to-back roughly doubles the number of switches and the diameter of a single butterfly, but it completely eliminates congestion problems! The proof of this fact relies on a clever induction argument that we'll come to in a

network	diameter	switch size	# switches	congestion
complete binary tree	$2\log N + 2$	3×3	$2N - 1$	N
2-D array	$2N$	2×2	N^2	2
butterfly	$\log N + 2$	2×2	$N(\log(N) + 1)$	$\sqrt{N}$ or $\sqrt{N/2}$
Beneš	$2\log N + 1$	2×2	$2N \log N$	1

Table 6.4 A comparison of the N-input Beneš network with the N-input complete binary tree, 2-d array, and butterfly.

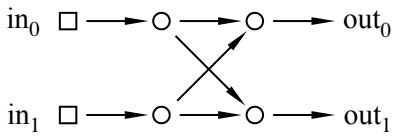


Figure 6.14 The 2-input Beneš network.

moment. Let's first see how the Beneš network stacks up against the other networks we have been studying. As you can see in Table 6.4, the Beneš network has small size and diameter, and completely eliminates congestion. The Holy Grail of routing networks is in hand!

Theorem 6.3.2. *The congestion of the N-input Beneš network is 1 for any N that is a power of 2.*

Proof. We use induction. Let $P(a)$ be the proposition that the congestion of the 2^a-input Beneš network is 1.

Base case ($a = 1$): We must show that the congestion of the 2^1-input Beneš network is 1. The network is shown in Figure 6.14.

There are only two possible permutation routing problems for a 2-input network. If $\pi(0) = 0$ and $\pi(1) = 1$, then we can route both packets along the straight edges. On the other hand, if $\pi(0) = 1$ and $\pi(1) = 0$, then we can route both packets along the diagonal edges. In both cases, a single packet passes through each switch.

Inductive step: We must show that $P(a)$ implies $P(a+1)$ where $a \geq 1$. Thus, we assume that the congestion of a 2^a-input Beneš network is 1 in order to prove that the congestion of a 2^{a+1}-input Beneš network is also 1.

Digression

Time out! Let's work through an example, develop some intuition, and then complete the proof. Notice that inside a Beneš network of size $2N$ lurk two Beneš subnetworks of size N. This follows from our earlier observation that a butterfly

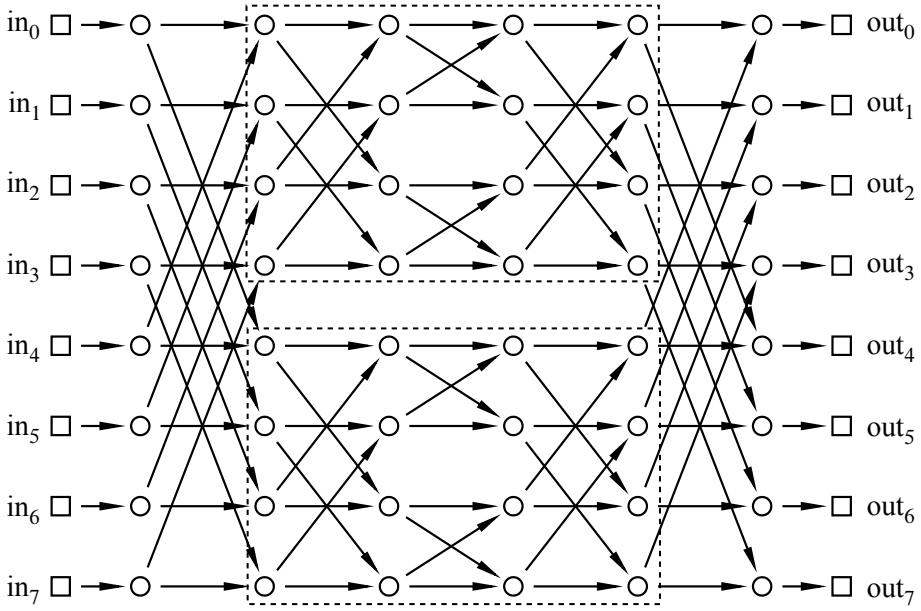


Figure 6.15 A $2N$-input Beneš network contains two N-input Beneš networks—shown here for $N = 4$.

of size $2N$ contains two butterflies of size N. In the Beneš network shown in Figure 6.15 with $N = 8$ inputs and outputs, the two 4-input/output subnetworks are shown in dashed boxes.

By the inductive assumption, the subnetworks can each route an arbitrary permutation with congestion 1. So if we can guide packets safely through just the first and last levels, then we can rely on induction for the rest! Let's see how this works in an example. Consider the following permutation routing problem:

$$\begin{aligned} \pi(0) &= 1 & \qquad \pi(4) &= 3 \\ \pi(1) &= 5 & \qquad \pi(5) &= 6 \\ \pi(2) &= 4 & \qquad \pi(6) &= 0 \\ \pi(3) &= 7 & \qquad \pi(7) &= 2 \end{aligned}$$

We can route each packet to its destination through either the upper subnetwork or the lower subnetwork. However, the choice for one packet may constrain the choice for another. For example, we can not route the packets at inputs 0 and 4 both through the same network since that would cause two packets to collide at a single switch, resulting in congestion. So one packet must go through the upper network and the other through the lower network. Similarly, the packets at inputs 1 and 5,

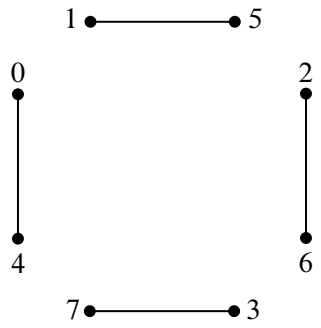


Figure 6.16 The beginnings of a constraint graph for our packet routing problem. Adjacent packets cannot be routed using the same sub-Beneš network.

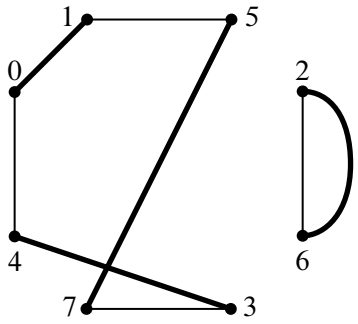


Figure 6.17 The updated constraint graph.

2 and 6, and 3 and 7 must be routed through different networks. Let's record these constraints in a graph. The vertices are the 8 packets (labeled according to their input position). If two packets must pass through different networks, then there is an edge between them. The resulting constraint graph is illustrated in Figure 6.16. Notice that at most one edge is incident to each vertex.

The output side of the network imposes some further constraints. For example, the packet destined for output 0 (which is packet 6) and the packet destined for output 4 (which is packet 2) can not both pass through the same network since that would require both packets to arrive from the same switch. Similarly, the packets destined for outputs 1 and 5, 2 and 6, and 3 and 7 must also pass through different switches. We can record these additional constraints in our constraint graph with gray edges, as is illustrated in Figure 6.17.

Notice that at most one new edge is incident to each vertex. The two lines drawn between vertices 2 and 6 reflect the two different reasons why these packets must be routed through different networks. However, we intend this to be a simple graph;

the two lines still signify a single edge.

Now here's the key insight: *a 2-coloring of the graph corresponds to a solution to the routing problem*. In particular, suppose that we could color each vertex either red or blue so that adjacent vertices are colored differently. Then all constraints are satisfied if we send the red packets through the upper network and the blue packets through the lower network.

The only remaining question is whether the constraint graph is 2-colorable. Fortunately, this is easy to verify:

Lemma 6.3.3. *If the edges of an undirected graph G can be grouped into two sets such that every vertex is incident to at most 1 edge from each set, then the graph is 2-colorable.*

Proof. Since the two sets of edges may overlap, let's call an edge that is in both sets a *doubled edge*. Note that no other edge can be incident to either of the endpoints of a doubled edge, since that endpoint would then be incident to two edges from the same set. This means that doubled edges form connected components with 2 nodes. Such connected components are easily colored with 2 colors and so we can henceforth ignore them and focus on the remaining nodes and edges, which form a simple graph.

By Theorem 5.6.2, we know that if a simple graph has no odd cycles, then it is 2-colorable. So all we need to do is show that every cycle in G has even length. This is easy since any cycle in G must traverse successive edges that alternate from one set to the other. In particular, a closed walk must traverse a path of alternating edges that begins and ends with edges from different sets. This means that the cycle has to be of even length. ■

For example, a 2-coloring of the constraint graph in Figure 6.17 is shown in Figure 6.18. The solution to this graph-coloring problem provides a start on the packet routing problem. We can complete the routing in the two smaller Beneš networks by induction. With this insight in hand, the digression is over and we can now complete the proof of Theorem 6.3.2.

Proof of Theorem 6.3.2 (*continued*). Let π be an arbitrary permutation of $0, 1, \ldots, N-1$. Let G be the graph whose vertices are packet numbers $0, 1, \ldots, N-1$ and whose edges come from the union of these two sets:

$$E_1 ::= \{\{u, v\} \mid |u - v| = N/2\}, \text{ and}$$
$$E_2 ::= \{\{u, w\} \mid |\pi(u) - \pi(w)| = N/2\}.$$

Now any vertex, u, is incident to at most two edges: a unique edge $\{u, v\} \in E_1$ and a unique edge $\{u, w\} \in E_2$. So according to Lemma 6.3.3, there is a 2-coloring for

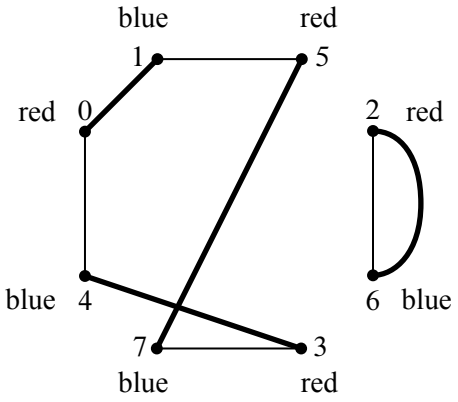
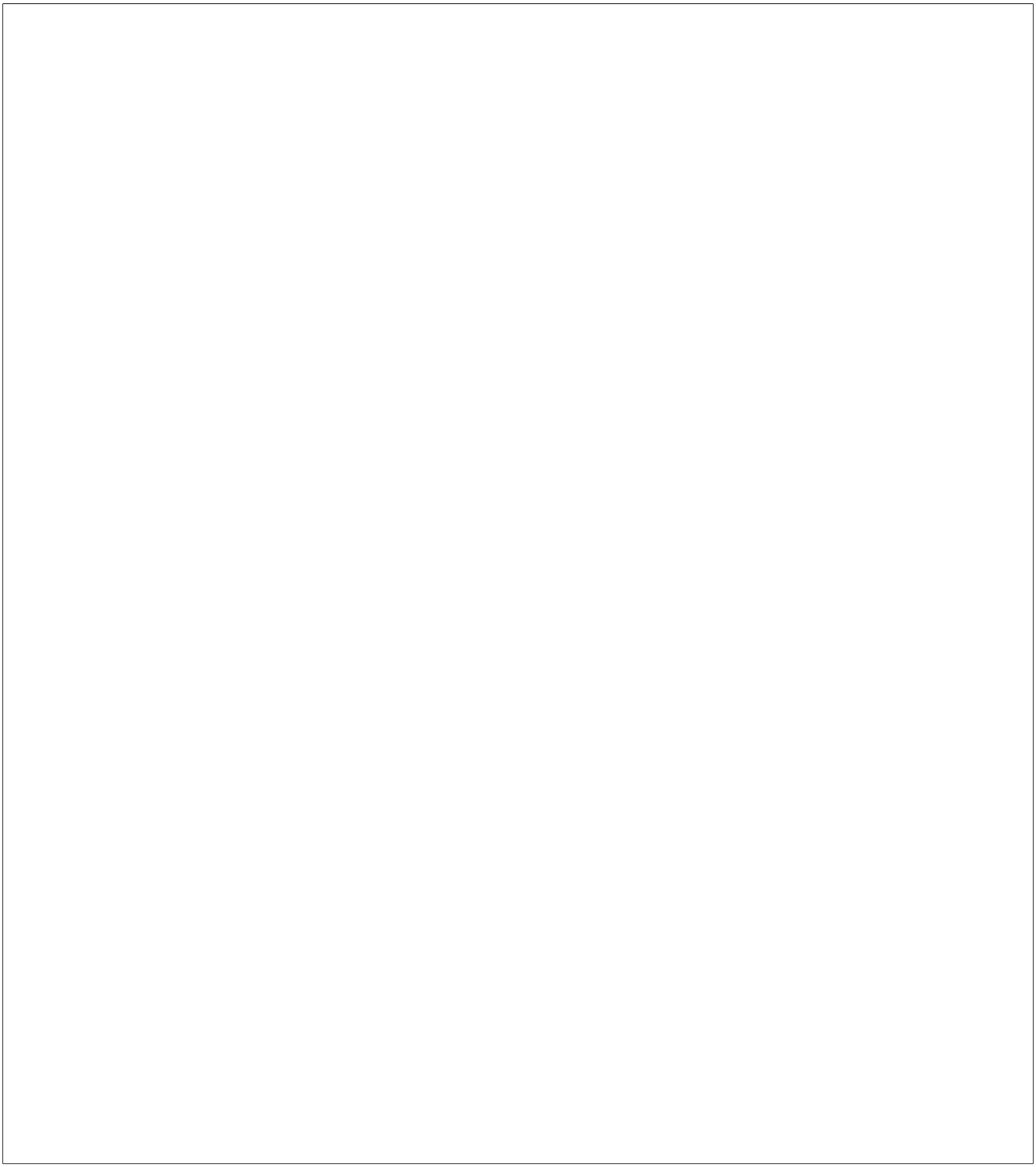


Figure 6.18 A 2-coloring of the constraint graph in Figure 6.17.

the vertices of G. Now route packets of one color through the upper subnetwork and packets of the other color through the lower subnetwork. Since for each edge in E_1, one vertex goes to the upper subnetwork and the other to the lower subnetwork, there will not be any conflicts in the first level. Since for each edge in E_2, one vertex comes from the upper subnetwork and the other from the lower subnetwork, there will not be any conflicts in the last level. We can complete the routing within each subnetwork by the induction hypothesis $P(n)$. ■



7 Relations and Partial Orders

A relation is a mathematical tool for describing associations between elements of sets. Relations are widely used in computer science, especially in databases and scheduling applications. A relation can be defined across many items in many sets, but in this text, we will focus on *binary* relations, which represent an association between two items in one or two sets.

7.1 Binary Relations

7.1.1 Definitions and Examples

Definition 7.1.1. Given sets A and B, a *binary relation* $R : A \to B$ *from*[1] A to B is a subset of $A \times B$. The sets A and B are called the *domain* and *codomain* of R, respectively. We commonly use the notation aRb or $a \sim_R b$ to denote that $(a, b) \in R$.

A relation is similar to a function. In fact, every function $f : A \to B$ is a relation. In general, the difference between a function and a relation is that a relation might associate multiple elements of B with a single element of A, whereas a function can only associate at most one element of B (namely, $f(a)$) with each element $a \in A$.

We have already encountered examples of relations in earlier chapters. For example, in Section 5.2, we talked about a relation between the set of men and the set of women where mRw if man m likes woman w. In Section 5.3, we talked about a relation on the set of MIT courses where $c_1 R c_2$ if the exams for c_1 and c_2 cannot be given at the same time. In Section 6.3, we talked about a relation on the set of switches in a network where $s_1 R s_2$ if s_1 and s_2 are directly connected by a wire that can send a packet from s_1 to s_2. We did not use the formal definition of a relation in any of these cases, but they are all examples of relations.

As another example, we can define an "in-charge-of" relation T from the set of MIT faculty F to the set of subjects in the 2010 MIT course catalog. This relation contains pairs of the form

$$(\langle\text{instructor-name}\rangle, \langle\text{subject-num}\rangle)$$

[1] We also say that the relationship is *between* A and B, or *on* A if $B = A$.

(Meyer,	6.042),
(Meyer,	18.062),
(Meyer,	6.844),
(Leighton,	6.042),
(Leighton,	18.062),
(Freeman,	6.011),
(Freeman,	6.881)
(Freeman,	6.882)
(Freeman,	6.UAT)
(Eng,	6.UAT)
(Guttag,	6.00)

Figure 7.1 Some items in the "in-charge-of" relation T between faculty and subject numbers.

where the faculty member named ⟨instructor-name⟩ is in charge of the subject with number ⟨subject-num⟩. So T contains pairs like those shown in Figure 7.1.

This is a surprisingly complicated relation: Meyer is in charge of subjects with three numbers. Leighton is also in charge of subjects with two of these three numbers—because the same subject, Mathematics for Computer Science, has two numbers (6.042 and 18.062) and Meyer and Leighton are jointly in-charge-of the subject. Freeman is in-charge-of even more subjects numbers (around 20), since as Department Education Officer, he is in charge of whole blocks of special subject numbers. Some subjects, like 6.844 and 6.00 have only one person in-charge. Some faculty, like Guttag, are in-charge-of only one subject number, and no one else is jointly in-charge-of his subject, 6.00.

Some subjects in the codomain, N, do not appear in the list—that is, they are not an element of any of the pairs in the graph of T; these are the Fall term only subjects. Similarly, there are faculty in the domain, F, who do not appear in the list because all their in-charge-of subjects are Fall term only.

7.1.2 Representation as a Bipartite Graph

Every relation $R : A \to B$ can be easily represented as a bipartite graph $G = (V, E)$ by creating a "left" node for each element of A and a "right" node for each element of B. We then create an edge between a left node u and a right node v whenever aRb. Similarly, every bipartite graph (and every partition of the nodes into a "left" and "right" set for which no edge connects a pair of left nodes or a pair of right nodes) determines a relation between the nodes on the left and the nodes on the right.

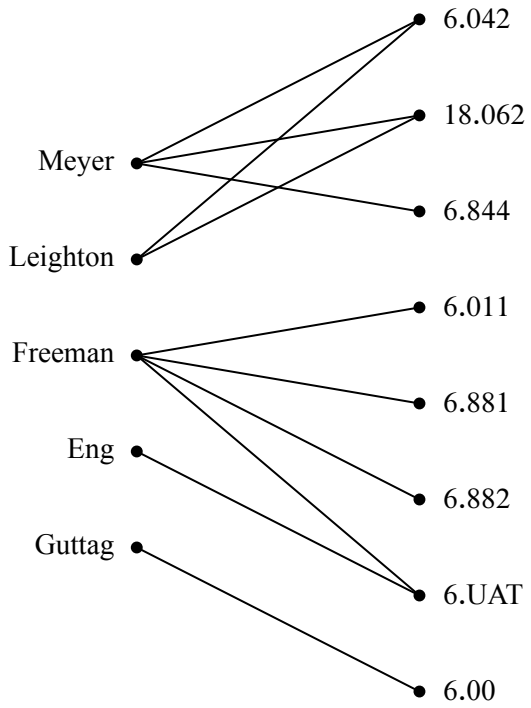


Figure 7.2 Part of the bipartite graph for the "in charge of" relation T from Figure 7.1.

For example, we have shown part of the bipartite graph for the in-charge-of relation from Figure 7.1 in Figure 7.2. In this case, there is an edge between ⟨instructor-name⟩ and ⟨subject-number⟩ if ⟨instructor-name⟩ is in charge of ⟨subject-number⟩.

A relation $R : A \to B$ between finite sets can also be represented as a matrix $A = \{a_{ij}\}$ where

$$a_{ij} = \begin{cases} 1 & \text{if the } i\text{th element of } A \text{ is related to the } j\text{th element of } B \\ 0 & \text{otherwise} \end{cases}$$

for $1 \leq i \leq |A|$ and $1 \leq j \leq |B|$. For example, the matrix for the relation in Figure 7.2 (but restricted to the five faculty and eight subject numbers shown in Figure 7.2, ordering them as they appear top-to-bottom in Figure 7.2) is shown in Figure 7.3.

7.1.3 Relational Images

The idea of the image of a set under a function extends directly to relations.

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 7.3 The matrix for the "in charge of" relation T restricted to the five faculty and eight subject numbers shown in Figure 7.2. The $(3, 4)$ entry of this matrix is 1 since the third professor (Freeman) is in charge of the fourth subject number (6.011).

Definition 7.1.2. The *image* of a set Y under a relation $R : A \to B$, written $R(Y)$, is the set of elements that are related to some element in Y, namely,

$$R(Y) ::= \{ b \in B \mid yRb \text{ for some } y \in Y \}.$$

The image of the domain, $R(A)$, is called the *range* of R.

For example, to find the subject numbers that Meyer is in charge of, we can look for all the pairs of the form

$$(\text{Meyer}, \langle \text{subject-number} \rangle)$$

in the graph of the teaching relation T, and then just list the right-hand sides of these pairs. These right-hand sides are exactly the image $T(\text{Meyer})$, which happens to be $\{6.042, 18.062, 6.844\}$. Similarly, since the domain F is the set of all in-charge faculty, $T(F)$, the range of T, is exactly the set of *all* subjects being taught.

7.1.4 Inverse Relations and Images

Definition 7.1.3. The *inverse* R^{-1} of a relation $R : A \to B$ is the relation from B to A defined by the rule

$$bR^{-1}a \text{ if and only if } aRb.$$

The image of a set under the relation R^{-1} is called the *inverse image* of the set. That is, the inverse image of a set X under the relation R is $R^{-1}(X)$.

Continuing with the in-charge-of example above, we can find the faculty in charge of 6.UAT by taking the pairs of the form

$$(\langle \text{instructor-name} \rangle, 6.\text{UAT})$$

for the teaching relation T, and then just listing the left-hand sides of these pairs; these turn out to be just Eng and Freeman. These left-hand sides are exactly the inverse image of {6.UAT} under T.

7.1.5 Combining Relations

There are at least two natural ways to combine relations to form new relations. For example, given relations $R : B \to C$ and $S : A \to B$, the *composition* of R with S is the relation $(R \circ S) : A \to C$ defined by the rule

$$a(R \circ S)c \quad \text{IFF} \quad \exists b \in B.\ (bRc) \text{ AND } (aSb)$$

where $a \in A$ and $c \in C$.

As a special case, the composition of two functions $f : B \to C$ and $g : A \to B$ is the function $f \circ g : A \to C$ defined by

$$(f \circ g)(a) = f(g(a))$$

for all $a \in A$. For example, if $A = B = C = \mathbb{R}$, $g(x) = x + 1$ and $f(x) = x^2$, then

$$\begin{aligned}(f \circ g)(x) &= (x+1)^2 \\ &= x^2 + 2x + 1.\end{aligned}$$

One can also define the *product* of two relations $R_1 : A_1 \to B_1$ and $R_2 : A_2 \to B_2$ to be the relation $S = R_1 \times R_2$ where

$$S : A_1 \times A_2 \to B_1 \times B_2$$

and

$$(a_1, a_2)S(b_1, b_2) \quad \text{iff} \quad a_1 R_1 b_1 \text{ and } a_2 R_2 b_2.$$

7.2 Relations and Cardinality

7.2.1 Surjective and Injective Relations

There are some properties of relations that will be useful when we take up the topic of counting in Part III because they imply certain relations between the *sizes* of domains and codomains. In particular, we say that a binary relation $R : A \to B$ is

- *surjective* if every element of B is assigned to at least one element of A. More concisely, R is surjective iff $R(A) = B$ (that is, if the range of R is the codomain of R),

- *total* when every element of A is assigned to some element of B. More concisely, R is total iff $A = R^{-1}(B)$,
- *injective* if every element of B is mapped *at most once*, and
- *bijective* if R is total, surjective, injective, and a function[2].

We can illustrate these properties of a relation $R : A \to B$ in terms of the corresponding bipartite graph G for the relation, where nodes on the left side of G correspond to elements of A and nodes on the right side of G correspond to elements of B. For example:

- "R is a function" means that every node on the left is incident to *at most* one edge.
- "R is total" means that *every* node on the left is incident to *at least* one edge. So if R is a function, being total means that every node on the left is incident to exactly one edge.
- "R is surjective" means that *every* node on the right is incident to *at least* one edge.
- "R is injective" means that *every* node on the right is incident to *at most* one edge.
- "R is bijective" means that every node on both sides is incident to *precisely* one edge (that is, there is a perfect matching between A and B).

For example, consider the relations R_1 and R_2 shown in Figure 7.4. R_1 is a total surjective function (every node in the left column is incident to exactly one edge, and every node in the right column is incident to at least one edge), but not injective (node 3 is incident to 2 edges). R_2 is a total injective function (every node in the left column is incident to exactly one edge, and every node in the right column is incident to at most one edge), but not surjective (node 4 is not incident to any edges).

Notice that we need to know what the domain is to determine whether a relation is total, and we need to know the codomain to determine whether it's surjective. For example, the function defined by the formula $1/x^2$ is total if its domain is $\mathbb{R}^+$ but partial if its domain is some set of real numbers that includes 0. It is bijective if its domain and codomain are both $\mathbb{R}^+$, but neither injective nor surjective it is domain and codomain are both $\mathbb{R}$.

[2]These words *surjective*, *injective*, and *bijective* are not very memorable. Some authors use the possibly more memorable phrases *onto* for surjective, *one-to-one* for injective, and *exact correspondence* for bijective.

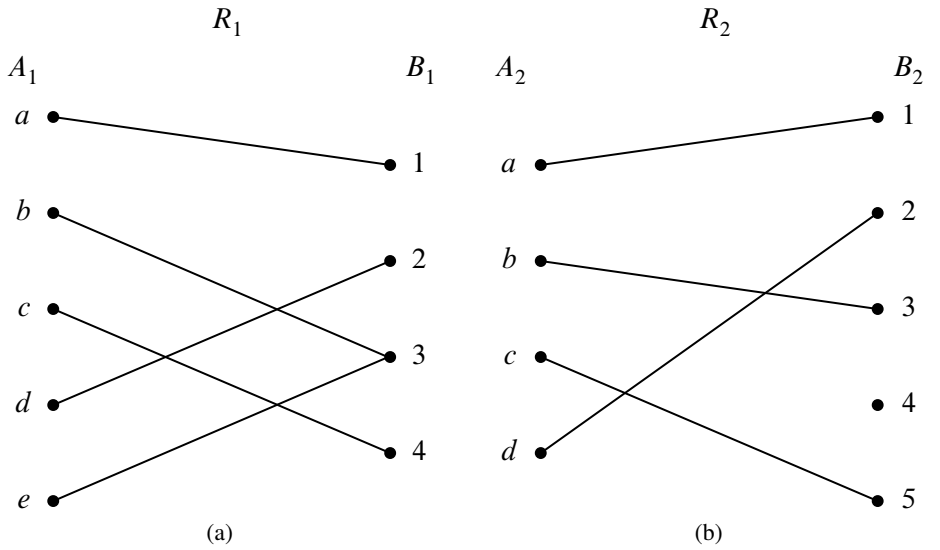


Figure 7.4 Relation $R_1 : A_1 \to B_1$ is shown in (a) and relation $R_2 : A_2 \to B_2$ is shown in (b).

7.2.2 Cardinality

The relational properties in Section 7.2.1 are useful in figuring out the relative sizes of domains and codomains.

If A is a finite set, we use $|A|$ to denote the number of elements in A. This is called the *cardinality* of A. In general, a finite set may have no elements (the empty set), or one element, or two elements, ..., or any nonnegative integer number of elements, so for any finite set, $|A| \in \mathbb{N}$.

Now suppose $R : A \to B$ is a function. Then every edge in the bipartite graph $G = (V, E)$ for R is incident to exactly one element of A, so the number of edges is at most the number of elements of A. That is, if R is a function, then

$$|E| \leq |A|.$$

Similarly, if R is surjective, then every element of B is incident to an edge, so there must be at least as many edges in the graph as the size of B. That is

$$|E| \geq |B|.$$

Combining these inequalities implies that $R : A \to B$ is a surjective function, then $|A| \geq |B|$. This fact and two similar rules relating domain and codomain size to relational properties are captured in the following theorem.

Theorem 7.2.1 (Mapping Rules)**.** *Let A and B be finite sets.*

1. *If there is a surjection from A to B, then $|A| \geq |B|$.*
2. *If there is an injection from A to B, then $|A| \leq |B|$.*
3. *If there is a bijection between A and B, then $|A| = |B|$.*

Mapping rule 2 can be explained by the same kind of reasoning we used for rule 1. Rule 3 is an immediate consequence of the first two mapping rules.

We will see many examples where Theorem 7.2.1 is used to determine the cardinality of a finite set. Later, in Chapter 13, we will consider the case when the sets are infinite and we'll use surjective and injective relations to prove that some infinite sets are "bigger" than other infinite sets.

7.3 Relations on One Set

For the rest of this chapter, we are going to focus on relationships between elements of a single set; that is, relations from a set A to a set B where $A = B$. Thus, a relation on a set A is a subset $R \subseteq A \times A$. Here are some examples:

- Let A be a set of people and the relation R describe who likes whom: that is, $(x, y) \in R$ if and only if x likes y.
- Let A be a set of cities. Then we can define a relation R such that xRy if and only if there is a nonstop flight from city x to city y.
- Let $A = \mathbb{Z}$ and let xRy hold if and only if $x \equiv y \pmod 5$.
- Let $A = \mathbb{N}$ and let xRy if and only if $x \mid y$.
- Let $A = \mathbb{N}$ and let xRy if and only if $x \leq y$.

The last examples clarify the reason for using xRy or $x \sim_R y$ to indicate that the relation R holds between x and y: many common relations ($<$, $\leq$, $=$, $|$, $\equiv$) are expressed with the relational symbol in the middle.

7.3.1 Representation as a Digraph

Every relation on a single set A can be modeled as a directed graph (albeit one that may contain loops). For example, the graph in Figure 7.5 describes the "likes" relation for a particular set of 3 people.

In this case, we see that:

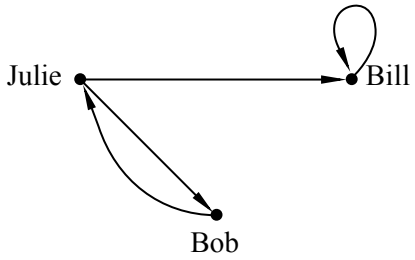


Figure 7.5 The directed graph for the "likes" relation on the set {Bill, Bob, Julie}.

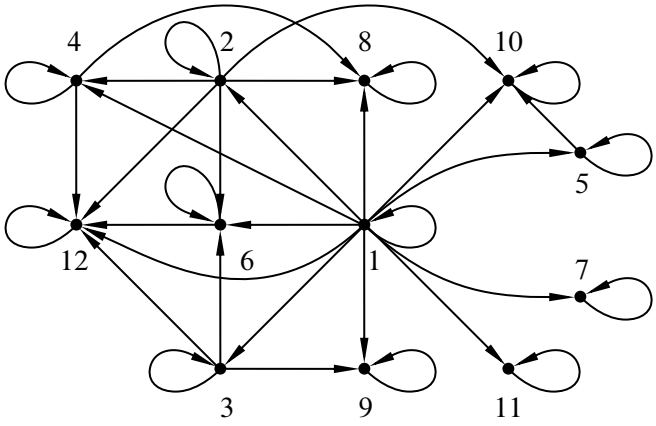


Figure 7.6 The digraph for divisibility on $\{1, 2, \ldots, 12\}$.

- Julie likes Bill and Bob, but not herself.
- Bill likes only himself.
- Bob likes Julie, but not Bill nor himself.

Everything about the relationship is conveyed by the directed graph and nothing more. This is no coincidence; a set A together with a relation R is precisely the same thing as directed graph $G = (V, E)$ with vertex set $V = A$ and edge set $E = R$ (where E may have loops).

As another example, we have illustrated the directed graph for the divisibility relationship on the set $\{1, 2, \ldots, 12\}$ in Figure 7.6. In this graph, every node has a loop (since every positive number divides itself) and the composite numbers are the nodes with indegree more than 1 (not counting the loop).

Relations on a single set can also be represented as a 0, 1-matrix. In this case, the matrix is identical to the adjacency matrix for the corresponding digraph. For

example, the matrix for the relation shown in Figure 7.5 is simply

$$\begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

where $v_1 =$ Julie, $v_2 =$ Bill, and $v_3 =$ Bob.

7.3.2 Symmetry, Transitivity, and Other Special Properties

Many relations on a single set that arise in practice possess one or more noteworthy properties. These properties are summarized in the box on the following page. In each case, we provide the formal of the definition of the property, explain what the property looks like in a digraph G for the relation, and give an example of what the property means for the "likes" relation.

For example, the congruence relation modulo 5 on $\mathbb{Z}$ is reflexive symmetric, and transitive, but not irreflexive, antisymmetric, or asymmetric. The same is true for the "connected" relation $R : V \to V$ on an undirected graph $G = (V, E)$ where uRv if u and v are in the same connected component of graph G. In fact, relations that have these three properties are so common that we give them a special name: *equivalence relations*. We will discuss them in greater detail in just a moment.

As another example, the "divides" relation on $\mathbb{Z}^+$ is reflexive, antisymmetric, and transitive, but not irreflexive, symmetric, or asymmetric. The same is true for the "$\leq$" relation on $\mathbb{R}$. Relations that have these three properties are also very common and they fall into a special case of relations called a *partial order*. We will discuss partial orders at length in Sections 7.5–7.9.

As a final example, consider the "likes" relation on the set {Julie, Bill, Bob} illustrated in Figure 7.5. This relation has *none* of the six properties described in the box.

7.4 Equivalence Relations

A relation is an *equivalence relation* if it is reflexive, symmetric, and transitive. Congruence modulo n is an excellent example of an equivalence relation:

- It is reflexive because $x \equiv x \pmod{n}$.
- It is symmetric because $x \equiv y \pmod{n}$ implies $y \equiv x \pmod{n}$.
- It is transitive because $x \equiv y \pmod{n}$ and $y \equiv z \pmod{n}$ imply that $x \equiv z \pmod{n}$.

Properties of a Relation $R : A \to A$

Reflexivity R is *reflexive* if

$\forall x \in A.\ xRX.$

"Everyone likes themselves."

Every node in G has a loop.

Irreflexivity R is *irreflexive* if

$\neg\exists x \in A.\ xRx.$

"No one likes themselves."

There are no loops in G.

Symmetry R is *symmetric* if

$\forall x, y \in A.\ xRy$ IMPLIES $yRx.$

"If x likes y, then y likes x."

If there is an edge from x to y in G, then there is an edge from y to x in G as well.

Antisymmetry R is *antisymmetric* if

$\forall x, y \in A\ (xRy$ AND $yRx)$ IMPLIES $x = y.$

"No pair of distinct people like each other."

There is at most one directed edge between any pair of distinct nodes.

Asymmetry R is *asymmetric* if

$\neg\exists x, y \in A.\ xRy$ AND $yRx.$

"No one likes themselves and no pair of people like each other."

There are no loops and there is at most one directed edge between any pair of nodes.

Transitivity R is *transitive* if

$\forall x, y, z \in A.\ (xRy$ AND $yRz)$ IMPLIES $xRz.$

"If x likes y and y likes z, then x likes z too."

For any walk $v_0, v_1, \ldots, v_k$ in G where $k \geq 2,\ v_0 \to v_k$ is in G (and, hence, $v_i \to v_j$ is also in G for all $i < j$.

There is an even more well-known example of an equivalence relation: equality itself. Thus, an equivalence relation is a relation that shares some key properties with "=".

7.4.1 Partitions

There is another way to think about equivalence relations, but we'll need a couple of definitions to understand this alternative perspective.

Definition 7.4.1. Given an equivalence relation $R : A \to A$, the *equivalence class* of an element $x \in A$ is the set of all elements of A related to x by R. The equivalence class of x is denoted $[x]$. Thus, in symbols:

$$[x] = \{\, y \mid xRy \,\}.$$

For example, suppose that $A = \mathbb{Z}$ and xRy means that $x \equiv y \pmod 5$. Then

$$[7] = \{\ldots, -3, 2, 7, 12, 22, \ldots\}.$$

Notice that 7, 12, 17, etc., all have the same equivalence class; that is, $[7] = [12] = [17] = \cdots$.

Definition 7.4.2. A *partition* of a finite set A is a collection of disjoint, nonempty subsets $A_1, A_2, \ldots, A_n$ whose union is all of A. The subsets are usually called the *blocks* of the partition.[3] For example, one possible partition of $A = \{a, b, c, d, e\}$ is

$$A_1 = \{a, c\} \qquad A_2 = \{b, e\} \qquad A_3 = \{d\}.$$

Here's the connection between all this stuff: there is an exact correspondence between *equivalence relations on* A and *partitions of* A. We can state this as a theorem:

Theorem 7.4.3. *The equivalence classes of an equivalence relation on a set A form a partition of A.*

We won't prove this theorem (too dull even for us!), but let's look at an example.

[3] We think they should be called the *parts* of the partition. Don't you think that makes a lot more sense?

The congruent-mod-5 relation partitions the integers into five equivalence classes:

$$\{\ldots, -5, 0, 5, 10, 15, 20, \ldots\}$$
$$\{\ldots, -4, 1, 6, 11, 16, 21, \ldots\}$$
$$\{\ldots, -3, 2, 7, 12, 17, 22, \ldots\}$$
$$\{\ldots, -2, 3, 8, 13, 18, 23, \ldots\}$$
$$\{\ldots, -1, 4, 9, 14, 19, 24, \ldots\}$$

In these terms, $x \equiv y \pmod 5$ is equivalent to the assertion that x and y are both in the same block of this partition. For example, $6 \equiv 16 \pmod 5$, because they're both in the second block, but $2 \not\equiv 9 \pmod 5$ because 2 is in the third block while 9 is in the last block.

In social terms, if "likes" were an equivalence relation, then everyone would be partitioned into cliques of friends who all like each other and no one else.

7.5 Partial Orders

7.5.1 Strong and Weak Partial Orders

Definition 7.5.1. A relation R on a set A is a *weak partial order* if it is transitive, antisymmetric, and reflexive. The relation is said to be a *strong partial order* if it is transitive, antisymmetric, and irreflexive.[4]

Some authors defined partial orders to be what we call weak partial orders, but we'll use the phrase *partial order* to mean either a weak or a strong partial order. The difference between a weak partial order and a strong one has to do with the reflexivity property: in a weak partial order, *every* element is related to itself, but in a strong partial order, *no* element is related to itself. Otherwise, they are the same in that they are both transitive and antisymmetric.

Examples of weak partial orders include "$\leq$" on $\mathbb{R}$, "$\subseteq$" on the set of subsets of (say) $\mathbb{Z}$, and the "divides" relation on $\mathbb{N}^+$. Examples of strict partial orders include "$<$" on $\mathbb{R}$, and "$\subset$" on the set of subsets of $\mathbb{Z}$.[5]

[4] Equivalently, the relation is transitive and asymmetric, but stating it this way might have obscured the irreflexivity property.

[5] If you are not feeling comfortable with all the definitions that we've been throwing at you, it's probably a good idea to verify that each of these relations are indeed partial orders by checking that they have the transitivity and antisymmetry properties.

We often denote a weak partial order with a symbol such as $\preceq$ or $\sqsubseteq$ instead of a letter such as R. This makes sense from one perspective since the symbols call to mind $\leq$ and $\subseteq$, which define common partial orders. On the other hand, a partial order is really a set of related pairs of items, and so a letter like R would be more normal.

Likewise, we will often use a symbol like $\prec$ or $\sqsubset$ to denote a strong partial order.

7.5.2 Total Orders

A partial order is "partial" because there can be two elements with no relation between them. For example, in the "divides" partial order on $\{1, 2, \ldots, 12\}$, there is no relation between 3 and 5 (since neither divides the other).

In general, we say that two elements a and b are *incomparable* if neither $a \preceq b$ nor $b \preceq a$. Otherwise, if $a \preceq b$ or $b \preceq a$, then we say that a and b are *comparable*.

Definition 7.5.2. A *total order* is a partial order in which every pair of distinct elements is comparable.

For example, the "$\leq$" partial order on $\mathbb{R}$ is a total order because for any pair of real numbers x and y, either $x \leq y$ or $y \leq x$. The "divides" partial order on $\{1, 2, \ldots, 12\}$ is not a total order because $3 \nmid 5$ and $5 \nmid 3$.

7.6 Posets and DAGs

7.6.1 Partially Ordered Sets

Definition 7.6.1. Given a partial order $\preceq$ on a set A, the pair $(A, \preceq)$ is called a *partially ordered set* or *poset*.

In terms of graph theory, a poset is simply the directed graph $G = (A, \preceq)$ with vertex set A and edge set $\preceq$. For example, Figure 7.6 shows the graph form of the poset for the "divides" relation on $\{1, 2, \ldots, 12\}$. We have shown the graph form of the poset for the "$<$"-relation on $\{1, 2, 3, 4\}$ in Figure 7.7.

7.6.2 Posets Are Acyclic

Did you notice anything that is common to Figures 7.6 and 7.7? Of course, they both exhibit the transitivity and antisymmetry properties. And, except for the loops in Figure 7.6, they both do not contain any cycles. This is not a coincidence. In fact, the combination of the transitivity and asymmetry properties imply that the digraph

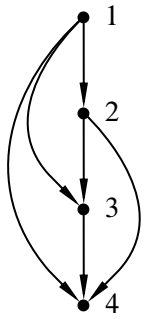


Figure 7.7 Representing the poset for the "<"-relation on $\{1, 2, 3, 4\}$ as a digraph.

for any poset is an acyclic graph (that is, a DAG), at least if you don't count loops as cycles. We prove this fact in the following theorem.

Theorem 7.6.2. *A poset has no directed cycles other than self-loops.*

Proof. We use proof by contradiction. Let $(A, \preceq)$ be a poset. Suppose that there exist $n \geq 2$ distinct elements $a_1, a_2, \ldots, a_n$ such that

$$a_1 \preceq a_2 \preceq a_3 \preceq \cdots \preceq a_{n-1} \preceq a_n \preceq a_1.$$

Since $a_1 \preceq a_2$ and $a_2 \preceq a_3$, transitivity implies $a_1 \preceq a_3$. Another application of transitivity shows that $a_1 \preceq a_4$ and a routine induction argument establishes that $a_1 \preceq a_n$. Since we know that $a_n \preceq a_1$, antisymmetry implies $a_1 = a_n$, contradicting the supposition that $a_1, \ldots, a_n$ are distinct and $n \geq 2$. Thus, there is no such directed cycle. ■

Thus, deleting the self-loops from a poset leaves a directed graph without cycles, which makes it a *directed acyclic graph* or *DAG*.

7.6.3 Transitive Closure

Theorem 7.6.2 tells us that every poset corresponds to a DAG. Is the reverse true? That is, does every DAG correspond to a poset? The answer is "Yes," but we need to modify the DAG to make sure that it satisfies the transitivity property. For example, consider the DAG shown in Figure 7.8. As any DAG must, this graph satisfies the antisymmetry property[6] but it does not satisfy the transitivity property because $v_1 \to v_2$ and $v_2 \to v_3$ are in the graph but $v_1 \to v_3$ is not in the graph.

[6] If $u \to v$ and $v \to u$ are in a digraph G, then G would have a cycle of length 2 and it could not be a DAG.

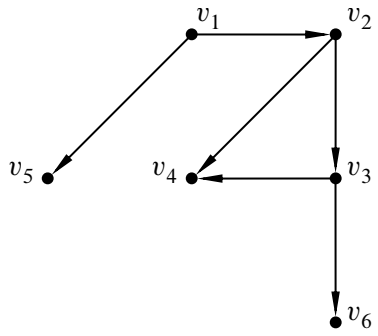


Figure 7.8 A 6-node digraph that does not satisfy the transitivity property.

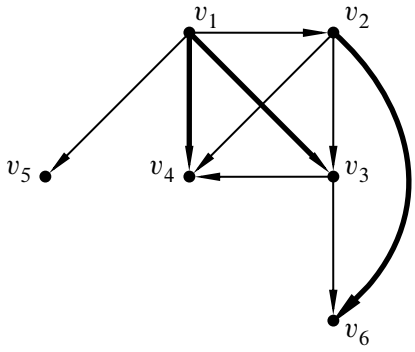


Figure 7.9 The transitive closure for the digraph in Figure 7.8. The edges that were added to form the transitive closure are shown in bold.

Definition 7.6.3. Given a digraph $G = (V, E)$, the *transitive closure* of G is the digraph $G^+ = (V, E^+)$ where

$$E^+ = \{ u \to v \mid \text{there is a directed path of positive length from } u \text{ to } v \text{ in } G \}.$$

Similarly, if R is the relation corresponding to G, the *transitive closure* of R (denoted R^+) is the relation corresponding to G^+.

For example, the transitive closure for the graph in Figure 7.8 is shown in Figure 7.9.

If G is a DAG, then the transitive closure of G is a strong partial order. The proof of this fact is left as an exercise in the problem section.

7.6.4 The Hasse Diagram

One problem with viewing a poset as a digraph is that there tend to be lots of edges due to the transitivity property. Fortunately, we do not necessarily have to draw

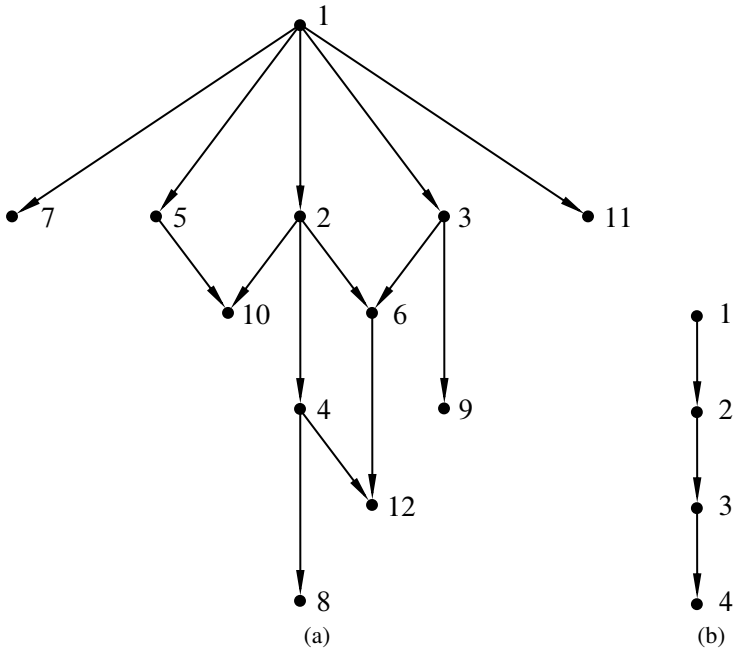


Figure 7.10 The Hasse diagrams for the posets in Figure 7.6 and 7.7.

all the edges if we know that the digraph corresponds to a poset. For example, we could choose not to draw any edge which would be implied by the transitivity property, knowing that it is really there by implication. In general, a *Hasse diagram* for a poset $(A, \preceq)$ is a digraph with vertex set A and edge set $\preceq$ minus all self-loops and edges implied by transitivity. For example, the Hasse diagrams of the posets shown in Figures 7.6 and 7.7 are shown in Figure 7.10.

7.7 Topological Sort

A total order that is consistent with a partial order is called a topological sort. More precisely,

Definition 7.7.1. A *topological sort* of a poset $(A, \preceq)$ is a total order $(A, \preceq_T)$ such that

$$x \preceq y \text{ IMPLIES } x \preceq_T y.$$

For example, consider the poset that describes how a guy might get dressed for a formal occasion. The Hasse diagram for such a poset is shown in Figure 7.11.

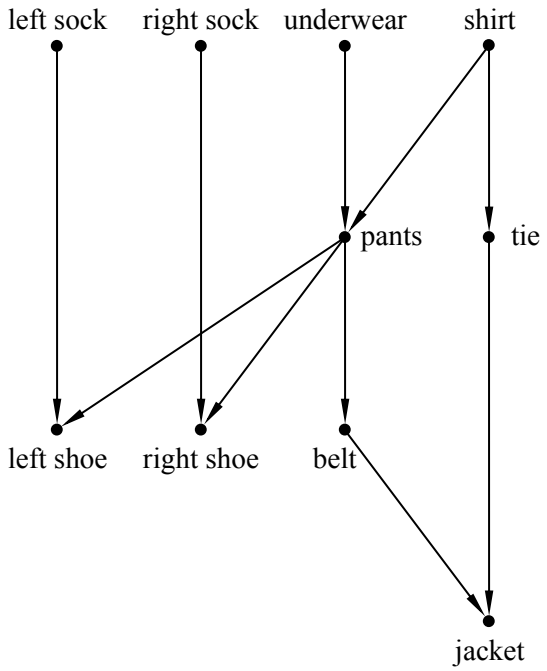


Figure 7.11 The Hasse diagram for a poset that describes which items much precede others when getting dressed.

In this poset, the *set* is all the garments and the *partial order* specifies which items much precede others when getting dressed.

There are several total orders that are consistent with the partial order shown in Figure 7.11. We have shown two of them in list form in Figure 7.12. Each such list is a topological sort for the partial order in Figure 7.11. In what follows, we will prove that every *finite* poset has a topological sort. You can think of this as a mathematical proof that you *can* get dressed in the morning (and then show up for math lecture).

Theorem 7.7.2. *Every finite poset has a topological sort.*

We'll prove the theorem constructively. The basic idea is to pull the "smallest" element a out of the poset, find a topological sort of the remainder recursively, and then add a back into the topological sort as an element smaller than all the others.

The first hurdle is that "smallest" is not such a simple concept in a set that is only partially ordered. In a poset $(A, \preceq)$, an element $x \in A$ is *minimal* if there is no other element $y \in A$ such that $y \preceq x$. For example, there are *four* minimal elements in the getting-dressed poset: left sock, right sock, underwear, and shirt. (It may seem

underwear	left sock
pants	shirt
belt	tie
shirt	underwear
tie	right sock
jacket	pants
left sock	right shoe
right sock	belt
left shoe	jacket
right shoe	left shoe
(a)	(b)

Figure 7.12 Two possible topological sorts of the poset shown in Figure 7.11. In each case, the elements are listed so that $x \preceq y$ iff x is above y in the list.

odd that the minimal elements are at the top of the Hasse diagram rather than the bottom. Some people adopt the opposite convention. If you're not sure whether minimal elements are on the top or bottom in a particular context, ask.) Similarly, an element $x \in A$ is *maximal* if there is no other element $y \in A$ such that $x \preceq y$.

Proving that every poset *has* a minimal element is extremely difficult, because it is not true. For example, the poset $(\mathbb{Z}, \leq)$ has no minimal element. However, there is at least one minimal element in every *finite* poset.

Lemma 7.7.3. *Every finite poset has a minimal element.*

Proof. Let $(A, \preceq)$ be an arbitrary poset. Let $a_1, a_2, \ldots, a_n$ be a maximum-length sequence of distinct elements in A such that

$$a_1 \preceq a_2 \preceq \cdots \preceq a_n.$$

The existence of such a maximum-length sequence follows from the Well Ordering Principle and the fact that A is finite. Now $a_0 \preceq a_1$ cannot hold for any element $a_0 \in A$ not in the chain, since the chain already has maximum length. And $a_i \preceq a_1$ cannot hold for any $i \geq 2$, since that would imply a cycle

$$a_i \preceq a_1 \preceq a_2 \preceq \cdots \preceq a_i$$

and no cycles exist in a poset by Theorem 7.6.2. Therefore a_1 is a minimal element. ■

Now we're ready to prove Theorem 7.7.2, which says that every finite poset has a topological sort. The proof is rather intricate; understanding the argument requires a clear grasp of all the mathematical machinery related to posets and relations!

Proof of Theorem 7.7.2. We use induction. Let $P(n)$ be the proposition that every n-element poset has a topological sort.

Base case: Every 1-element poset is already a total order and thus is its own topological sort. So $P(1)$ is true.

Inductive step: Now we assume $P(n)$ in order to prove $P(n+1)$ where $n \geq 1$. Let $(A, \preceq)$ be an $(n+1)$-element poset. By Lemma 7.7.3, there exists a minimal element in $a \in A$. Remove a and all pairs in $\preceq$ involving a to obtain an n-element poset $(A', \preceq')$. This has a topological sort $(A', \preceq'_T)$ by the assumption $P(n)$. Now we construct a total order $(A, \preceq_T)$ by adding a back as an element smaller than all the others. Formally, let

$$\preceq_T = \preceq'_T \cup \{(a, z) \mid z \in A\}.$$

All that remains is to check that this total order is consistent with the original partial order $(A, \preceq)$; that is, we must show that

$$x \preceq y \text{ IMPLIES } x \preceq_T y.$$

We assume that the left side is true and show that the right side follows. There are two cases.

Case 1 If $x = a$, then $a \preceq_T y$ holds, because $a \preceq_T z$ for all $z \in A$.

Case 2 if $x \neq a$, then y can not equal a either, since a is a minimal element in the partial order $\preceq$. Thus, both x and y are in A' and so $x \preceq' y$. This means $x \preceq'_T y$, since $\preceq'_T$ is a topological sort of the partial order $\preceq'$. And this implies $x \preceq_T y$ since $\preceq_T$ contains $\preceq'_T$.

Thus, $(A, \preceq_T)$ is a topological sort of $(A, \preceq)$. This shows that $P(n)$ implies $P(n+1)$ for all $n \geq 1$. Therefore $P(n)$ is true for all $n \geq 1$ by the principle of induction, which proves the theorem. ■

7.8 Parallel Task Scheduling

When items of a poset are tasks that need to be done and the partial order is a precedence constraint, topological sorting provides us with a way to execute the tasks sequentially without violating the precedence constraints.

But what if we have the ability to execute more than one task at the same time? For example, suppose that the tasks are programs, the partial order indicates data

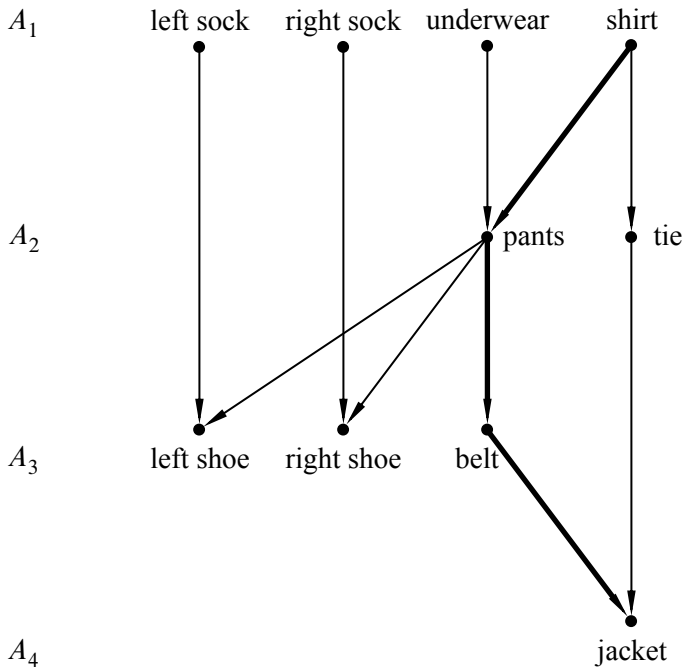


Figure 7.13 A parallel schedule for the tasks-in-getting-dressed poset in Figure 7.11. The tasks in A_i can be performed in step i for $1 \leq i \leq 4$. A chain of length 4 (the critical path in this example) is shown with bold edges.

dependence, and we have a parallel machine with lots of processors instead of a sequential machine with only one processor. How should we schedule the tasks so as to minimize the total time used?

For simplicity, assume all tasks take 1 unit of time and we have an unlimited number of identical processors. For example, in the clothes example in Figure 7.11, how long would it take to handle all the garments?

In the first unit of time, we should do all minimal items, so we would put on our left sock, our right sock, our underwear, and our shirt.[7] In the second unit of time, we should put on our pants and our tie. Note that we cannot put on our left or right shoe yet, since we have not yet put on our pants. In the third unit of time, we should put on our left shoe, our right shoe, and our belt. Finally, in the last unit of time, we can put on our jacket. This schedule is illustrated in Figure 7.13.

The total time to do these tasks is 4 units. We cannot do better than 4 units of

[7] Yes, we know that you can't actually put on both socks at once, but imagine you are being dressed by a bunch of robot processors and you are in a big hurry. Still not working for you? Ok, forget about the clothes and imagine they are programs with the precedence constraints shown in Figure 7.11.

time because there is a sequence of 4 tasks, each needing to be done before the next, of length 4. For example, we must put on our shirt before our pants, our pants before our belt, and our belt before our jacket. Such a sequence of items is known as a *chain*

Definition 7.8.1. A *chain* is a sequence $a_1 \preceq a_2 \preceq \cdots \preceq a_t$, where $a_i \neq a_j$ for all $i \neq j$, such that each item is comparable to the next in the chain, and it is smaller with respect to $\preceq$. The *length* of the chain is t, the number of elements in the chain.

Thus, the time it takes to schedule tasks, even with an unlimited number of processors, is at least the length of the longest chain. Indeed, if we used less time, then two items from a longest chain would have to be done at the same time, which contradicts the precedence constraints. For this reason, a longest chain is also known as a *critical path*. For example, Figure 7.13 shows the critical path for the getting-dressed poset.

In this example, we were in fact able to schedule all the tasks in t steps, where t is the length of the longest chain. The really nice thing about posets is that this is always possible! In other words, for any poset, there is a legal parallel schedule that runs in t steps, where t is the length of the longest chain.

There are lots of ways to prove this fact. Our proof will also give us the corresponding schedule in t time steps, and allow us to obtain some nice corollaries.

Theorem 7.8.2. *Given any finite poset $(A, \preceq)$ for which the longest chain has length t, it is possible to partition A into t subsets $A_1, A_2, \ldots, A_t$ such that for all $i \in \{1, 2, \ldots, t\}$ and for all $a \in A_i$, we have that all $b \preceq a$ appear in the set $A_1 \cup \ldots \cup A_{i-1}$.*

Before proving this theorem, first note that for each i, all items in A_i can be scheduled in time step i. This is because all preceding tasks are scheduled in preceding time steps, and thus are already completed. So the theorem implies that

Corollary 7.8.3. *The total amount of parallel time needed to complete the tasks is the same as the length of the longest chain.*

Proof of Theorem 7.8.2. For all $a \in A_i$, put a in A_i, where i is the length of the longest chain ending at a. For example, the A_i for the getting-dressed poset are shown in Figure 7.13. In what follows, we show that for all i, for all $a \in A_i$ and for all $b \preceq a$ with $b \neq a$, we have $b \in A_1 \cup A_2 \cup \ldots \cup A_{i-1}$.

We prove this by contradiction. Assume there is some i, $a \in A_i$, and $b \preceq a$ with $b \neq a$ and $b \notin A_1 \cup A_2 \cup \ldots \cup A_{i-1}$. By the way we defined A_i, this implies there is a chain of length at least i ending at b. Since $b \preceq a$ and $b \neq a$, we can extend this chain to a chain of length at least $i + 1$, ending at a. But then a could not be in A_i. This is a contradiction. ■

If we have an unlimited number of processors, then the time to complete all tasks is equal to the length of the longest chain of dependent tasks. The case where there are only a limited number of processors is very useful in practice and it is covered in the Problems section.

7.9 Dilworth's Lemma

Definition 7.9.1. An *antichain* in a poset is a set of elements such that any two elements in the set are incomparable.

For example, each A_i in the proof of Theorem 7.8.2 and in Figure 7.13 is an antichain since its elements have no dependencies between them (which is why they could be executed at the same time).

Our conclusions about scheduling also tell us something about antichains.

Corollary 7.9.2. *If the largest chain in a partial order on a set A is of size t, then A can be partitioned into t antichains.*

Proof. Let the antichains be the sets $A_1, A_2, \ldots, A_t$ defined in Theorem 7.8.2. ■

Corollary 7.9.2 implies a famous result[8] about partially ordered sets:

Lemma 7.9.3 (Dilworth)**.** *For all $t > 0$, every partially ordered set with n elements must have either a chain of size greater than t or an antichain of size at least n/t.*

Proof. By contradiction. Assume that the longest chain has length at most t and the longest antichain has size less than n/t. Then by Corollary 7.9.2, the n elements can be partitioned into at most t antichains. Hence, there are fewer than $t \cdot n/t = n$ elements in the poset, which is a contradiction. Hence there must be a chain longer than t or an antichain with at least n/t elements. ■

Corollary 7.9.4. *Every partially ordered set with n elements has a chain of size greater than $\sqrt{n}$ or an antichain of size at least $\sqrt{n}$.*

Proof. Set $t = \sqrt{n}$ in Lemma 7.9.3. ■

As an application, consider a permutation of the numbers from 1 to n arranged as a sequence from left to right on a line. Corollary 7.9.4 can be used to show that there must be a $\sqrt{n}$-length subsequence of these numbers that is completely

[8] Lemma 7.9.3 also follows from a more general result known as Dilworth's Theorem that we will not discuss.

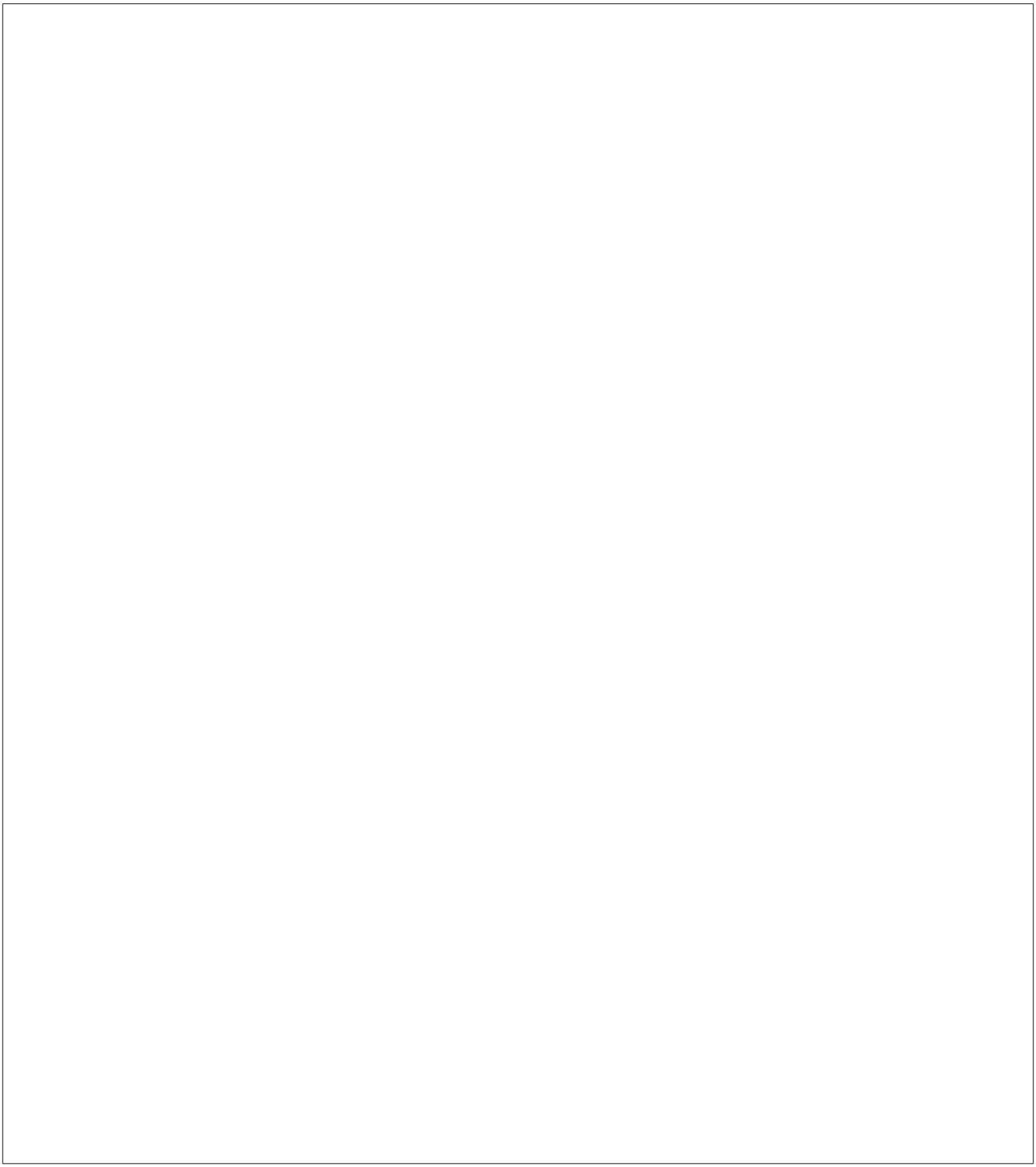
increasing or completely decreasing as you move from left to right. For example, the sequence

$$7, 8, 9, 4, 5, 6, 1, 2, 3$$

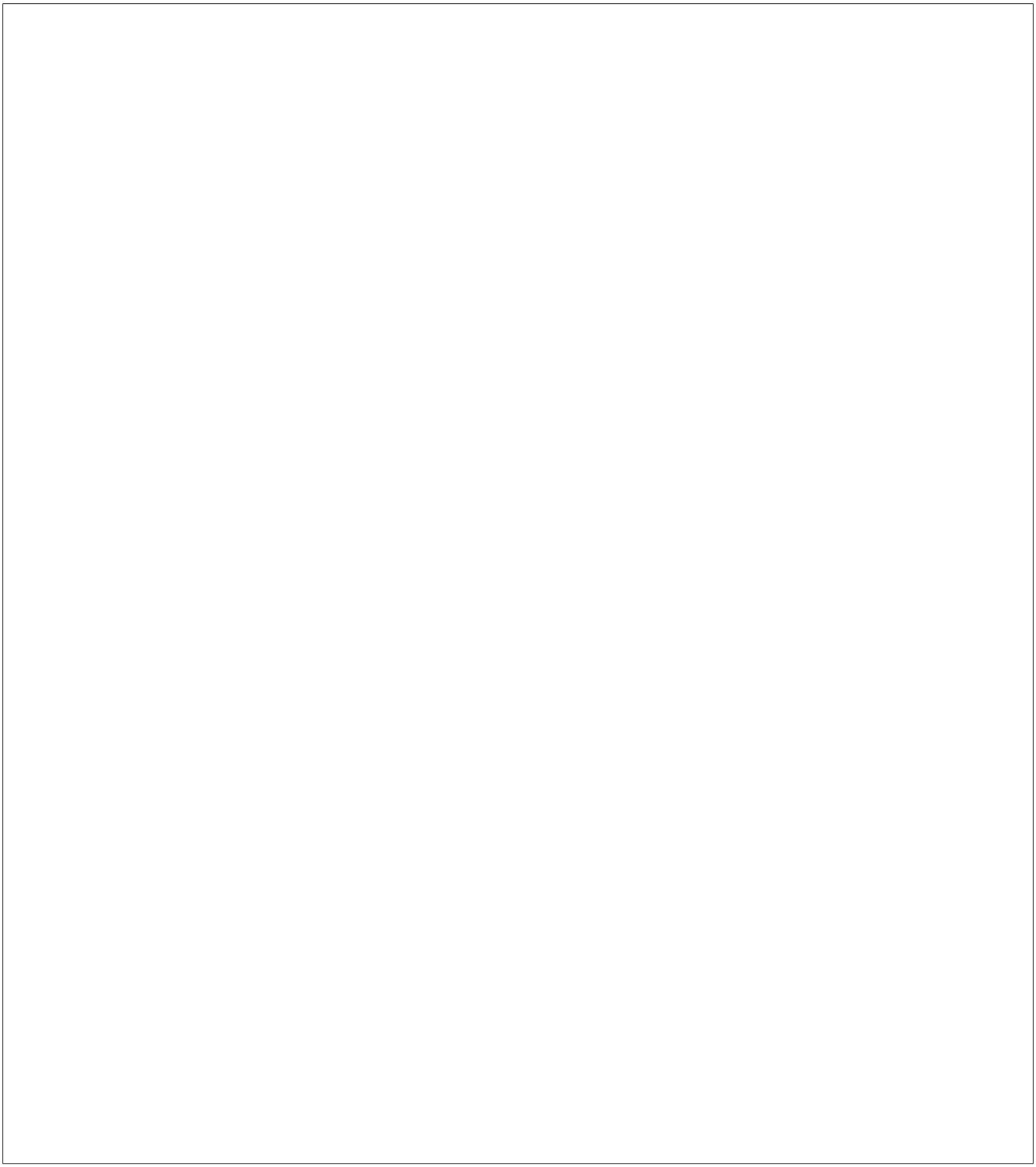
has an increasing subsequence of length 3 (for example, 7, 8, 9) and a decreasing subsequence of length 3 (for example, 9, 6, 3). The proof of this result is left as an exercise that will test your ability to find the right partial order on the numbers in the sequence.

8 State Machines

This chapter needs to be reworked.



III Counting



Introduction

Counting seems easy enough: 1, 2, 3, 4, etc. This direct approach works well for counting simple things—like your toes—and may be the only approach for extremely complicated things with no identifiable structure. However, subtler methods can help you count many things in the vast middle ground, such as:

- The number of different ways to select a dozen doughnuts when there are five varieties available.
- The number of 16-bit numbers with exactly 4 ones.

Perhaps surprisingly, but certainly not coincidentally, the number in each of these two situations is the same: 1820.

Counting is useful in computer science for several reasons:

- Determining the time and storage required to solve a computational problem—a central objective in computer science—often comes down to solving a counting problem.
- Counting is the basis of probability theory, which plays a central role in all sciences, including computer science.
- Two remarkable proof techniques, the "pigeonhole principle" and "combinatorial proof," rely on counting. These lead to a variety of interesting and useful insights.

In the next several chapters, we're going to present a lot of rules for counting. These rules are actually theorems, and we will prove some of them, but our focus won't be on the proofs *per se*—our objective is to teach you simple counting as a practical skill, like integration.

We begin our study of counting in Chapter 9 with a collection of rules and methods for finding closed-form expressions for commonly-occurring sums and products such as $\sum_{i=1}^{n} x^i$ and $n! = \prod_{i=1}^{n} i$. We also introduce asymptotic notations such as $\sim$, O, and Θ that are commonly used in computer science to express the how a quantity such as the running time of a program grows with the size of the input.

In Chapter 10, we show how to solve a variety of recurrences that arise in computational problems. These methods are especially useful when you need to design or analyze recursive programs.

In Chapters 11 and 12, we describe the most basic rules for determining the cardinality of a set. This material is simple yet powerful, and it provides a great tool set for use in your future career.

We conclude in Chapter 13 with a brief digression into the final frontier of counting—infinity. We'll define what it means for a set to be countable and show you some examples of sets that are really big—bigger even than the set of real numbers.

9 Sums and Asymptotics

Sums and products arise regularly in the analysis of algorithms, financial applications, physical problems, and probabilistic systems. For example, we have already encountered the sum $1 + 2 + 4 + \cdots + N$ when counting the number of nodes in a complete binary tree with N inputs. Although such a sum can be represented compactly using the sigma notation

$$\sum_{i=0}^{\log N} 2^i, \tag{9.1}$$

it is a lot easier and more helpful to express the sum by its *closed form* value

$$2N - 1.$$

By *closed form*, we mean an expression that does not make use of summation or product symbols or otherwise need those handy (but sometimes troublesome) dots.... Expressions in closed form are usually easier to evaluate (it doesn't get much simpler than $2N - 1$, for example) and it is usually easier to get a feel for their magnitude than expressions involving large sums and products.

But how do you find a closed form for a sum or product? Well, it's part math and part art. And it is the subject of this chapter.

We will start the chapter with a motivating example involving annuities. Figuring out the value of the annuity will involve a large and nasty-looking sum. We will then describe several methods for finding closed forms for all sorts of sums, including the annuity sums. In some cases, a closed form for a sum may not exist and so we will provide a general method for finding good upper and lower bounds on the sum (which are closed form, of course).

The methods we develop for sums will also work for products since you can convert any product into a sum by taking a logarithm of the product. As an example, we will use this approach to find a good closed-form approximation to

$$n! ::= 1 \cdot 2 \cdot 3 \cdots n.$$

We conclude the chapter with a discussion of asymptotic notation. Asymptotic notation is often used to bound the error terms when there is no exact closed form expression for a sum or product. It also provides a convenient way to express the growth rate or order of magnitude of a sum or product.

9.1 The Value of an Annuity

Would you prefer a million dollars today or $50,000 a year for the rest of your life? On the one hand, instant gratification is nice. On the other hand, the *total dollars* received at $50K per year is much larger if you live long enough.

Formally, this is a question about the value of an annuity. An *annuity* is a financial instrument that pays out a fixed amount of money at the beginning of every year for some specified number of years. In particular, an n-year, m-payment annuity pays m dollars at the start of each year for n years. In some cases, n is finite, but not always. Examples include lottery payouts, student loans, and home mortgages. There are even Wall Street people who specialize in trading annuities.[1]

A key question is, "What is an annuity worth?" For example, lotteries often pay out jackpots over many years. Intuitively, $50,000 a year for 20 years ought to be worth less than a million dollars right now. If you had all the cash right away, you could invest it and begin collecting interest. But what if the choice were between $50,000 a year for 20 years and a *half* million dollars today? Now it is not clear which option is better.

9.1.1 The Future Value of Money

In order to answer such questions, we need to know what a dollar paid out in the future is worth today. To model this, let's assume that money can be invested at a fixed annual interest rate p. We'll assume an 8% rate[2] for the rest of the discussion.

Here is why the interest rate p matters. Ten dollars invested today at interest rate p will become $(1+p)\cdot 10 = 10.80$ dollars in a year, $(1+p)^2\cdot 10 \approx 11.66$ dollars in two years, and so forth. Looked at another way, ten dollars paid out a year from now is only really worth $1/(1+p)\cdot 10 \approx 9.26$ dollars today. The reason is that if we had the $9.26 today, we could invest it and would have $10.00 in a year anyway. Therefore, p determines the value of money paid out in the future.

So for an n-year, m-payment annuity, the first payment of m dollars is truly worth m dollars. But the second payment a year later is worth only $m/(1+p)$ dollars. Similarly, the third payment is worth $m/(1+p)^2$, and the n-th payment is worth only $m/(1+p)^{n-1}$. The total value, V, of the annuity is equal to the sum of the

[1] Such trading ultimately led to the subprime mortgage disaster in 2008–2009. We'll talk more about that in Section 19.5.3.

[2] U.S. interest rates have dropped steadily for several years, and ordinary bank deposits now earn around 1.5%. But just a few years ago the rate was 8%; this rate makes some of our examples a little more dramatic. The rate has been as high as 17% in the past thirty years.

payment values. This gives:

$$\begin{aligned} V &= \sum_{i=1}^{n} \frac{m}{(1+p)^{i-1}} \\ &= m \cdot \sum_{j=0}^{n-1} \left(\frac{1}{1+p}\right)^{j} && \text{(substitute } j = i-1\text{)} \\ &= m \cdot \sum_{j=0}^{n-1} x^{j} && \text{(substitute } x = 1/(1+p)\text{).} \end{aligned} \tag{9.2}$$

The goal of the preceding substitutions was to get the summation into a simple special form so that we can solve it with a general formula. In particular, the terms of the sum

$$\sum_{j=0}^{n-1} x^{j} = 1 + x + x^2 + x^3 + \cdots + x^{n-1}$$

form a *geometric series*, which means that the ratio of consecutive terms is always the same and it is a positive value less than one. In this case, the ratio is always x, and $0 < x < 1$ since we assumed that $p > 0$. It turns out that there is a nice closed-form expression for any geometric series; namely

$$\sum_{i=0}^{n-1} x^{i} = \frac{1-x^{n}}{1-x}. \tag{9.3}$$

Equation 9.3 can be verified by induction, but, as is often the case, the proof by induction gives no hint about how the formula was found in the first place. So we'll take this opportunity to describe a method that you could use to figure it out for yourself. It is called the *Perturbation Method*.

9.1.2 The Perturbation Method

Given a sum that has a nice structure, it is often useful to "perturb" the sum so that we can somehow combine the sum with the perturbation to get something much simpler. For example, suppose

$$S = 1 + x + x^2 + \cdots + x^{n-1}.$$

An example of a perturbation would be

$$xS = x + x^2 + \cdots + x^{n}.$$

The difference between S and xS is not so great, and so if we were to subtract xS from S, there would be massive cancellation:

$$\begin{aligned} S &= 1 + x + x^2 + x^3 + \cdots + x^{n-1} \\ -xS &= \quad - x - x^2 - x^3 - \cdots - x^{n-1} - x^n. \end{aligned}$$

The result of the subtraction is

$$S - xS = 1 - x^n.$$

Solving for S gives the desired closed-form expression in Equation 9.3:

$$S = \frac{1 - x^n}{1 - x}.$$

We'll see more examples of this method when we introduce *generating functions* in Chapter 12.

9.1.3 A Closed Form for the Annuity Value

Using Equation 9.3, we can derive a simple formula for V, the value of an annuity that pays m dollars at the start of each year for n years.

$$V = m\left(\frac{1 - x^n}{1 - x}\right) \qquad \text{(by Equations 9.2 and 9.3)} \tag{9.4}$$

$$= m\left(\frac{1 + p - (1/(1+p))^{n-1}}{p}\right) \qquad \text{(substituting } x = 1/(1+p)\text{)}. \tag{9.5}$$

Equation 9.5 is much easier to use than a summation with dozens of terms. For example, what is the real value of a winning lottery ticket that pays \$50,000 per year for 20 years? Plugging in $m = \$50{,}000$, $n = 20$, and $p = 0.08$ gives $V \approx \$530{,}180$. So because payments are deferred, the million dollar lottery is really only worth about a half million dollars! This is a good trick for the lottery advertisers.

9.1.4 Infinite Geometric Series

The question we began with was whether you would prefer a million dollars today or \$50,000 a year for the rest of your life. Of course, this depends on how long you live, so optimistically assume that the second option is to receive \$50,000 a year *forever*. This sounds like infinite money! But we can compute the value of an annuity with an infinite number of payments by taking the limit of our geometric sum in Equation 9.3 as n tends to infinity.

Theorem 9.1.1. *If* $|x| < 1$, *then*

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}.$$

Proof.

$$\begin{aligned}
\sum_{i=0}^{\infty} x^i &::= \lim_{n\to\infty} \sum_{i=0}^{n-1} x^i \\
&= \lim_{n\to\infty} \frac{1-x^n}{1-x} && \text{(by Equation 9.3)} \\
&= \frac{1}{1-x}.
\end{aligned}$$

The final line follows from that fact that $\lim_{n\to\infty} x^n = 0$ when $|x| < 1$. ■

In our annuity problem, $x = 1/(1+p) < 1$, so Theorem 9.1.1 applies, and we get

$$\begin{aligned}
V &= m \cdot \sum_{j=0}^{\infty} x^j && \text{(by Equation 9.2)} \\
&= m \cdot \frac{1}{1-x} && \text{(by Theorem 9.1.1)} \\
&= m \cdot \frac{1+p}{p} && (x = 1/(1+p)).
\end{aligned}$$

Plugging in $m = \$50{,}000$ and $p = 0.08$, we see that the value V is only \$675,000. Amazingly, a million dollars today is worth much more than \$50,000 paid every year forever! Then again, if we had a million dollars today in the bank earning 8% interest, we could take out and spend \$80,000 a year forever. So on second thought, this answer really isn't so amazing.

9.1.5 Examples

Equation 9.3 and Theorem 9.1.1 are incredibly useful in computer science. In fact, we already used Equation 9.3 implicitly when we claimed in Chapter 6 than an N-input complete binary tree has

$$1 + 2 + 4 + \cdots + N = 2N - 1$$

nodes. Here are some other common sums that can be put into closed form using Equation 9.3 and Theorem 9.1.1:

$$1 + 1/2 + 1/4 + \cdots = \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = \frac{1}{1-(1/2)} = 2 \tag{9.6}$$

$$0.99999\cdots = 0.9\sum_{i=0}^{\infty} \left(\frac{1}{10}\right)^i = 0.9\left(\frac{1}{1-1/10}\right) = 0.9\left(\frac{10}{9}\right) = 1 \tag{9.7}$$

$$1 - 1/2 + 1/4 - \cdots = \sum_{i=0}^{\infty} \left(\frac{-1}{2}\right)^i = \frac{1}{1-(-1/2)} = \frac{2}{3} \tag{9.8}$$

$$1 + 2 + 4 + \cdots + 2^{n-1} = \sum_{i=0}^{n-1} 2^i = \frac{1-2^n}{1-2} = 2^n - 1 \tag{9.9}$$

$$1 + 3 + 9 + \cdots + 3^{n-1} = \sum_{i=0}^{n-1} 3^i = \frac{1-3^n}{1-3} = \frac{3^n-1}{2} \tag{9.10}$$

If the terms in a geometric sum grow smaller, as in Equation 9.6, then the sum is said to be *geometrically decreasing*. If the terms in a geometric sum grow progressively larger, as in Equations 9.9 and 9.10, then the sum is said to be *geometrically increasing*. In either case, the sum is usually approximately equal to the term in the sum with the greatest absolute value. For example, in Equations 9.6 and 9.8, the largest term is equal to 1 and the sums are 2 and 2/3, both relatively close to 1. In Equation 9.9, the sum is about twice the largest term. In Equation 9.10, the largest term is 3^{n-1} and the sum is $(3^n - 1)/2$, which is only about a factor of 1.5 greater. You can see why this rule of thumb works by looking carefully at Equation 9.3 and Theorem 9.1.1.

9.1.6 Variations of Geometric Sums

We now know all about geometric sums—if you have one, life is easy. But in practice one often encounters sums that cannot be transformed by simple variable substitutions to the form $\sum x^i$.

A non-obvious, but useful way to obtain new summation formulas from old is by differentiating or integrating with respect to x. As an example, consider the following sum:

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

This is not a geometric sum, since the ratio between successive terms is not fixed, and so our formula for the sum of a geometric sum cannot be directly applied. But suppose that we differentiate Equation 9.3:

$$\frac{d}{dx}\left(\sum_{i=0}^{n-1} x^i\right) = \frac{d}{dx}\left(\frac{1-x^n}{1-x}\right). \tag{9.11}$$

The left-hand side of Equation 9.11 is simply

$$\sum_{i=0}^{n-1} \frac{d}{dx}(x^i) = \sum_{i=0}^{n-1} i x^{i-1}.$$

The right-hand side of Equation 9.11 is

$$\begin{aligned}\frac{-nx^{n-1}(1-x)-(-1)(1-x^n)}{(1-x)^2} &= \frac{-nx^{n-1}+nx^n+1-x^n}{(1-x)^2}\\ &= \frac{1-nx^{n-1}+(n-1)x^n}{(1-x)^2}.\end{aligned}$$

Hence, Equation 9.11 means that

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

Often, differentiating or integrating messes up the exponent of x in every term. In this case, we now have a formula for a sum of the form $\sum i x^{i-1}$, but we want a formula for the series $\sum i x^i$. The solution is simple: multiply by x. This gives:

$$\sum_{i=1}^{n-1} i x^i = \frac{x-nx^n+(n-1)x^{n+1}}{(1-x)^2} \tag{9.12}$$

and we have the desired closed-form expression for our sum[3]. It's a little complicated looking, but it's easier to work with than the sum.

Notice that if $|x| < 1$, then this series converges to a finite value even if there are infinitely many terms. Taking the limit of Equation 9.12 as n tends infinity gives the following theorem:

[3]Since we could easily have made a mistake in the calculation, it is always a good idea to go back and validate a formula obtained this way with a proof by induction.

Theorem 9.1.2. *If* $|x| < 1$, *then*

$$\sum_{i=1}^{\infty} i x^i = \frac{x}{(1-x)^2}.$$

As a consequence, suppose that there is an annuity that pays im dollars at the *end* of each year i forever. For example, if $m = \$50{,}000$, then the payouts are \$50,000 and then \$100,000 and then \$150,000 and so on. It is hard to believe that the value of this annuity is finite! But we can use Theorem 9.1.2 to compute the value:

$$\begin{aligned} V &= \sum_{i=1}^{\infty} \frac{im}{(1+p)^i} \\ &= m \cdot \frac{1/(1+p)}{(1-\frac{1}{1+p})^2} \\ &= m \cdot \frac{1+p}{p^2}. \end{aligned}$$

The second line follows by an application of Theorem 9.1.2. The third line is obtained by multiplying the numerator and denominator by $(1+p)^2$.

For example, if $m = \$50{,}000$, and $p = 0.08$ as usual, then the value of the annuity is $V = \$8{,}437{,}500$. Even though the payments increase every year, the increase is only additive with time; by contrast, dollars paid out in the future decrease in value exponentially with time. The geometric decrease swamps out the additive increase. Payments in the distant future are almost worthless, so the value of the annuity is finite.

The important thing to remember is the trick of taking the derivative (or integral) of a summation formula. Of course, this technique requires one to compute nasty derivatives correctly, but this is at least theoretically possible!

9.2 Power Sums

In Chapter 3, we verified the formula

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \tag{9.13}$$

But the source of this formula is still a mystery. Sure, we can prove it is true using well ordering or induction, but where did the expression on the right come from in

the first place? Even more inexplicable is the closed form expression for the sum of consecutive squares:

$$\sum_{i=1}^{n} i^2 = \frac{(2n+1)(n+1)n}{6}. \tag{9.14}$$

It turns out that there is a way to derive these expressions, but before we explain it, we thought it would be fun[4] to show you how Gauss proved Equation 9.13 when he was a young boy.[5]

Gauss's idea is related to the perturbation method we used in Section 9.1.2. Let

$$S = \sum_{i=1}^{n} i.$$

Then we can write the sum in two orders:

$$\begin{aligned} S &= 1 + \quad 2 \quad + \ldots + (n-1) + n, \\ S &= n + (n-1) + \ldots + \quad 2 \quad + 1. \end{aligned}$$

Adding these two equations gives

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

Hence,

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

Not bad for a young child. Looks like Gauss had some potential....

Unfortunately, the same trick does not work for summing consecutive squares. However, we can observe that the result might be a third-degree polynomial in n, since the sum contains n terms that average out to a value that grows quadratically in n. So we might guess that

$$\sum_{i=1}^{n} i^2 = an^3 + bn^2 + cn + d.$$

If the guess is correct, then we can determine the parameters a, b, c, and d by plugging in a few values for n. Each such value gives a linear equation in a, b,

[4] Remember that we are mathematicians, so our definition of "fun" may be different than yours.

[5] We suspect that Gauss was probably not an ordinary boy.

c, and d. If we plug in enough values, we may get a linear system with a unique solution. Applying this method to our example gives:

$$\begin{aligned} n = 0 \quad &\text{implies} \quad 0 = d \\ n = 1 \quad &\text{implies} \quad 1 = a + b + c + d \\ n = 2 \quad &\text{implies} \quad 5 = 8a + 4b + 2c + d \\ n = 3 \quad &\text{implies} \quad 14 = 27a + 9b + 3c + d. \end{aligned}$$

Solving this system gives the solution $a = 1/3$, $b = 1/2$, $c = 1/6$, $d = 0$. Therefore, *if* our initial guess at the form of the solution was correct, then the summation is equal to $n^3/3 + n^2/2 + n/6$, which matches Equation 9.14.

The point is that if the desired formula turns out to be a polynomial, then once you get an estimate of the *degree* of the polynomial, all the coefficients of the polynomial can be found automatically.

Be careful! This method let's you discover formulas, but it doesn't guarantee they are right! After obtaining a formula by this method, it's important to go back and *prove* it using induction or some other method, because if the initial guess at the solution was not of the right form, then the resulting formula will be completely wrong![6]

9.3 Approximating Sums

Unfortunately, it is not always possible to find a closed-form expression for a sum. For example, consider the sum

$$S = \sum_{i=1}^{n} \sqrt{i}.$$

No closed form expression is known for S.

In such cases, we need to resort to approximations for S if we want to have a closed form. The good news is that there is a general method to find closed-form upper and lower bounds that work for most any sum. Even better, the method is simple and easy to remember. It works by replacing the sum by an integral and then adding either the first or last term in the sum.

[6] Alternatively, you can use the method based on generating functions described in Chapter 12, which does not require any guessing at all.

Theorem 9.3.1. *Let $f : \mathbb{R}^+ \to \mathbb{R}^+$ be a nondecreasing[7] continuous function and let*

$$S = \sum_{i=1}^{n} f(i)$$

and

$$I = \int_1^n f(x)\,dx.$$

Then

$$I + f(1) \leq S \leq I + f(n).$$

Similarly, if f is nonincreasing, then

$$I + f(n) \leq S \leq I + f(1).$$

Proof. Let $f : \mathbb{R}^+ \to \mathbb{R}^+$ be a nondecreasing function. For example, $f(x) = \sqrt{x}$ is such a function.

Consider the graph shown in Figure 9.1. The value of

$$S = \sum_{i=1}^{n} f(i)$$

is represented by the shaded area in this figure. This is because the ith rectangle in the figure (counting from left to right) has width 1 and height $f(i)$.

The value of

$$I = \int_1^n f(x)\,dx$$

is the shaded area under the curve of $f(x)$ from 1 to n shown in Figure 9.2.

Comparing the shaded regions in Figures 9.1 and 9.2, we see that S is at least I plus the area of the leftmost rectangle. Hence,

$$S \geq I + f(1) \tag{9.15}$$

This is the lower bound for S. We next derive the upper bound.

Figure 9.3 shows the curve of $f(x)$ from 1 to n shifted left by 1. This is the same as the curve $f(x+1)$ from 0 to $n-1$ and it has the same area I.

Comparing the shaded regions in Figures 9.1 and 9.3, we see that S is at most I plus the area of the rightmost rectangle. Hence,

$$S \leq I + f(n). \tag{9.16}$$

[7] A function f is *nondecreasing* if $f(x) \geq f(y)$ whenever $x \geq y$. It is *nonincreasing* if $f(x) \leq f(y)$ whenever $x \geq y$.

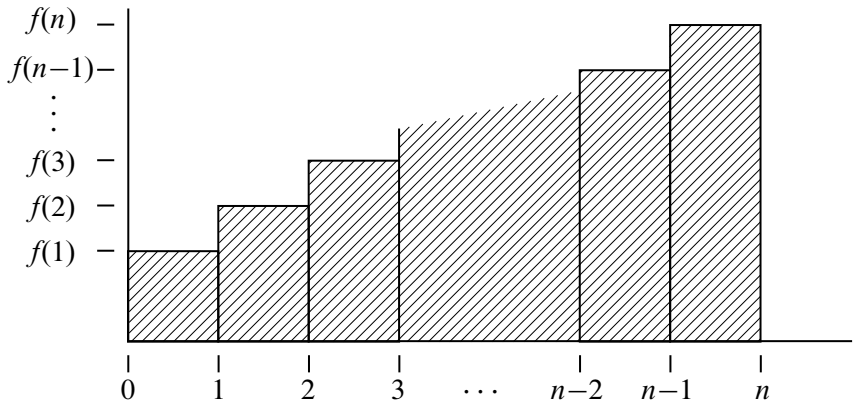


Figure 9.1 The area of the ith rectangle is $f(i)$. The shaded region has area $\sum_{i=1}^{n} f(i)$.

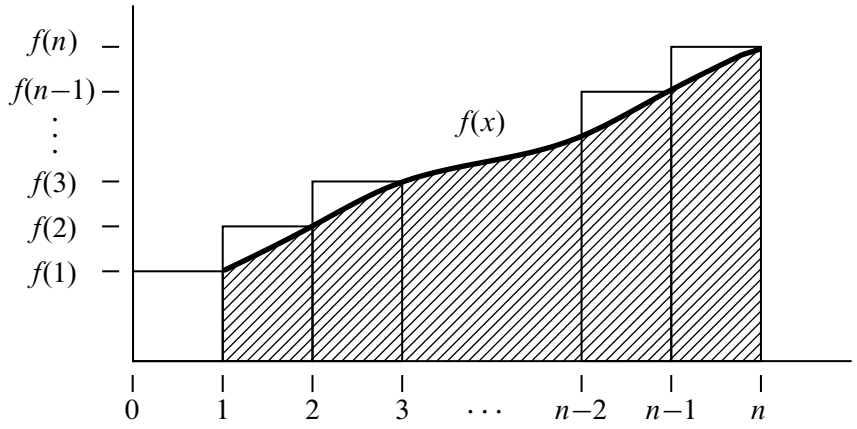


Figure 9.2 The shaded area under the curve of $f(x)$ from 1 to n (shown in bold) is $I = \int_1^n f(x)\,dx$.

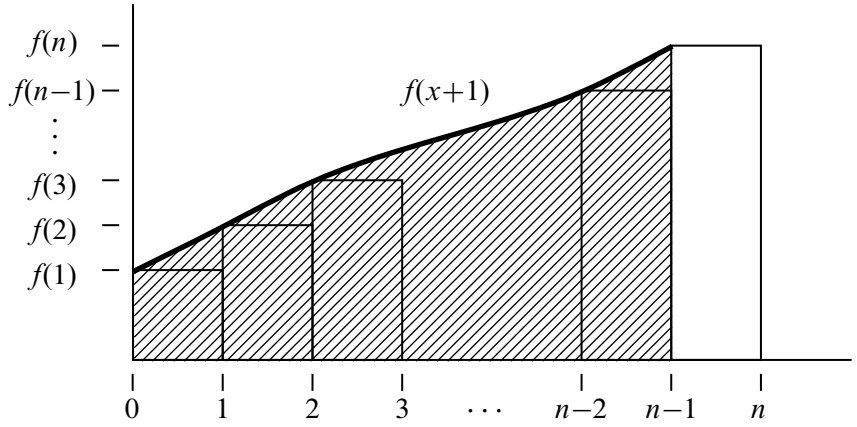


Figure 9.3 The shaded area under the curve of $f(x+1)$ from 0 to $n-1$ is I, the same as the area under the curve of $f(x)$ from 1 to n. This curve is the same as the curve in Figure 9.2 except that has been shifted left by 1.

Combining Equations 9.15 and 9.16, we find that

$$I + f(1) \leq S \leq I + f(n),$$

for any nondecreasing function f, as claimed

The argument for the case when f is nonincreasing is very similar. The analogous graphs to those shown in Figures 9.1–9.3 are provided in Figure 9.4. As you can see by comparing the shaded regions in Figures 9.4(a) and 9.4(b),

$$S \leq I + f(1).$$

Similarly, comparing the shaded regions in Figures 9.4(a) and 9.4(c) reveals that

$$S \geq I + f(n).$$

Hence, if f is nonincreasing,

$$I + f(n) \leq S \leq I + f(1).$$

as claimed. ■

Theorem 9.3.1 provides good bounds for most sums. At worst, the bounds will be off by the largest term in the sum. For example, we can use Theorem 9.3.1 to bound the sum

$$S = \sum_{i=1}^{n} \sqrt{i}$$

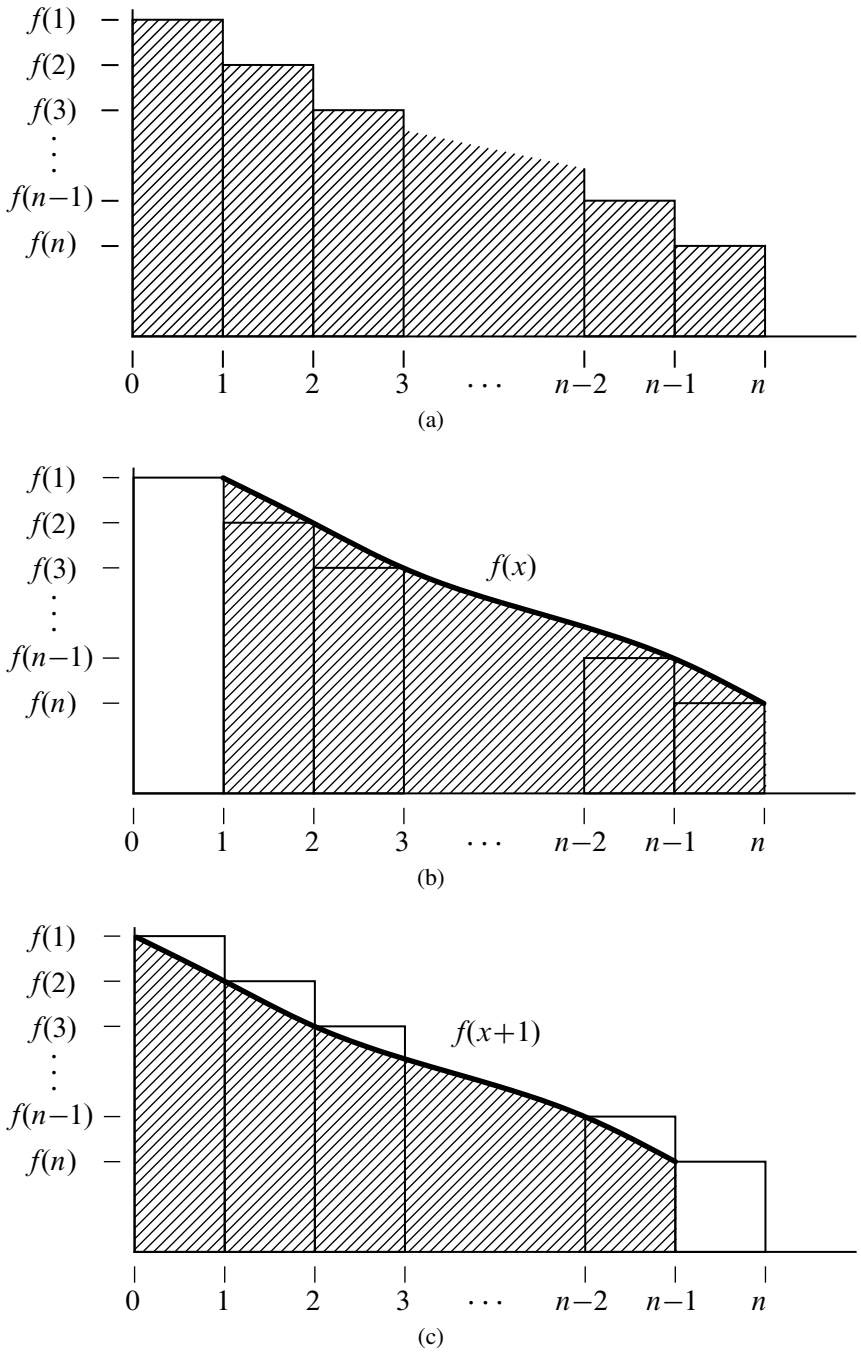


Figure 9.4 The area of the shaded region in (a) is $S = \sum_{i=1}^{n} f(i)$. The area in the shaded regions in (b) and (c) is $I = \int_1^n f(x)\,dx$.

as follows.

We begin by computing

$$\begin{aligned} I &= \int_1^n \sqrt{x}\, dx \\ &= \frac{x^{3/2}}{3/2}\Bigg|_1^n \\ &= \frac{2}{3}(n^{3/2} - 1). \end{aligned}$$

We then apply Theorem 9.3.1 to conclude that

$$\frac{2}{3}(n^{3/2} - 1) + 1 \;\le\; S \;\le\; \frac{2}{3}(n^{3/2} - 1) + \sqrt{n}$$

and thus that

$$\frac{2}{3}n^{3/2} + \frac{1}{3} \;\le\; S \;\le\; \frac{2}{3}n^{3/2} + \sqrt{n} - \frac{2}{3}.$$

In other words, the sum is very close to $\frac{2}{3}n^{3/2}$.

We'll be using Theorem 9.3.1 extensively going forward. At the end of this chapter, we will also introduce some notation that expresses phrases like "the sum is very close to" in a more precise mathematical manner. But first, we'll see how Theorem 9.3.1 can be used to resolve a classic paradox in structural engineering.

9.4 Hanging Out Over the Edge

Suppose that you have n identical blocks[8] and that you stack them one on top of the next on a table as shown in Figure 9.5. Is there some value of n for which it is possible to arrange the stack so that one of the blocks hangs out completely over the edge of the table without having the stack fall over? (You are not allowed to use glue or otherwise hold the stack in position.)

Most people's first response to this question—sometimes also their second and third responses—is "No. No block will ever get completely past the edge of the table." But in fact, if n is large enough, you can get the top block to stick out as far as you want: one block-length, two block-lengths, any number of block-lengths!

[8] We will assume that the blocks are rectangular, uniformly weighted and of length 1.

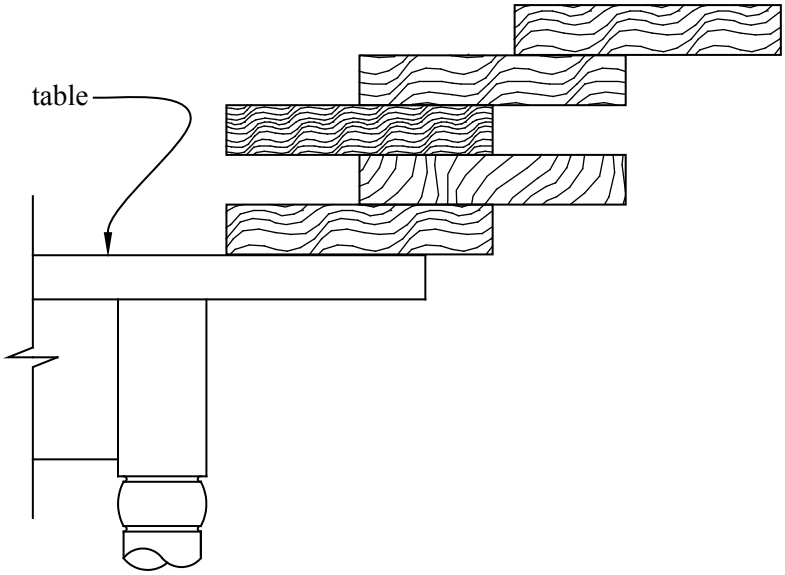


Figure 9.5 A stack of 5 identical blocks on a table. The top block is hanging out over the edge of the table, but if you try stacking the blocks this way, the stack will fall over.

9.4.1 Stability

A stack of blocks is said to be *stable* if it will not fall over of its own accord. For example, the stack illustrated in Figure 9.5 is not stable because the top block is sure to fall over. This is because the center or mass of the top block is hanging out over air.

In general, a stack of n blocks will be stable if and only if the center of mass of the top i blocks sits over the $(i+1)$st block for $i = 1, 2, \ldots, n-1$, and over the table for $i = n$.

We define the *overhang* of a stable stack to be the distance between the edge of the table and the rightmost end of the rightmost block in the stack. Our goal is thus to maximize the overhang of a stable stack.

For example, the maximum possible overhang for a single block is $1/2$. That is because the center of mass of a single block is in the middle of the block (which is distance $1/2$ from the right edge of the block). If we were to place the block so that its right edge is more than $1/2$ from the edge of the table, the center of mass would be over air and the block would tip over. But we can place the block so the center of mass is at the edge of the table, thereby achieving overhang $1/2$. This position is illustrated in Figure 9.6.

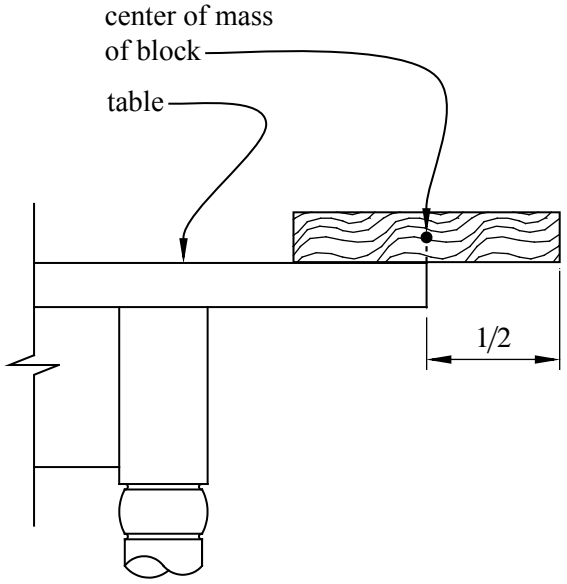


Figure 9.6 One block can overhang half a block length.

In general, the overhang of a stack of blocks is maximized by sliding the entire stack rightward until its center of mass is at the edge of the table. The overhang will then be equal to the distance between the center of mass of the stack and the rightmost edge of the rightmost block. We call this distance the *spread* of the stack. Note that the spread does not depend on the location of the stack on the table—it is purely a property of the blocks in the stack. Of course, as we just observed, the maximum possible overhang is equal to the maximum possible spread. This relationship is illustrated in Figure 9.7.

9.4.2 A Recursive Solution

Our goal is to find a formula for the maximum possible spread S_n that is achievable with a stable stack of n blocks.

We already know that $S_1 = 1/2$ since the right edge of a single block with length 1 is always distance $1/2$ from its center of mass. Let's see if we can use a recursive approach to determine S_n for all n. This means that we need to find a formula for S_n in terms of S_i where $i < n$.

Suppose we have a stable stack $\mathcal{S}$ of n blocks with maximum possible spread S_n. There are two cases to consider depending on where the rightmost block is in the stack.

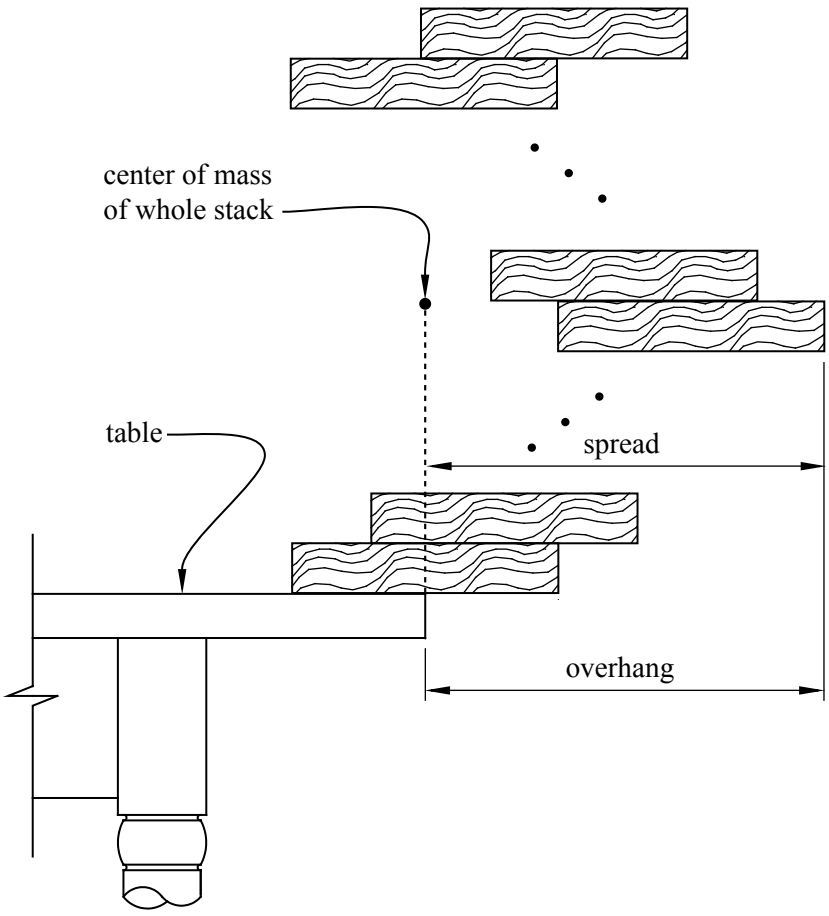


Figure 9.7 The overhang is maximized by maximizing the spread and then placing the stack so that the center of mass is at the edge of the table.

Case 1: *The rightmost block in $\mathcal{S}$ is the bottom block.* Since the center of mass of the top $n-1$ blocks must be over the bottom block for stability, the spread is maximized by having the center of mass of the top $n-1$ blocks be directly over the *left* edge of the bottom block. In this case the center of mass of $\mathcal{S}$ is[9]

$$\frac{(n-1)\cdot 1 + (1)\cdot \frac{1}{2}}{n} = 1 - \frac{1}{2n}$$

to the left of the right edge of the bottom block and so the spread for $\mathcal{S}$ is

$$1 - \frac{1}{2n}. \tag{9.17}$$

For example, see Figure 9.8.

In fact, the scenario just described is easily achieved by arranging the blocks as shown in Figure 9.9, in which case we have the spread given by Equation 9.17. For example, the spread is $3/4$ for 2 blocks, $5/6$ for 3 blocks, $7/8$ for 4 blocks, etc.

Can we do any better? The best spread in Case 1 is always less than 1, which means that we cannot get a block fully out over the edge of the table in this scenario. Maybe our intuition was right that we can't do better. Before we jump to any false conclusions, however, let's see what happens in the other case.

Case 2: *The rightmost block in $\mathcal{S}$ is among the top $n-1$ blocks.* In this case, the spread is maximized by placing the top $n-1$ blocks so that their center of mass is directly over the *right* end of the bottom block. This means that the center of mass for $\mathcal{S}$ is at location

$$\frac{(n-1)\cdot C + 1\cdot\left(C - \frac{1}{2}\right)}{n} = C - \frac{1}{2n}$$

where C is the location of the center of mass of the top $n-1$ blocks. In other words, the center of mass of $\mathcal{S}$ is $1/2n$ to the left of the center of mass of the top $n-1$ blocks. (The difference is due to the effect of the bottom block, whose center of mass is $1/2$ unit to the left of C.) This means that the spread of $\mathcal{S}$ is $1/2n$ greater than the spread of the top $n-1$ blocks (because we are in the case where the rightmost block is among the top $n-1$ blocks.)

Since the rightmost block is among the top $n-1$ blocks, the spread for $\mathcal{S}$ is maximized by maximizing the spread for the top $n-1$ blocks. Hence the maximum spread for $\mathcal{S}$ in this case is

$$S_{n-1} + \frac{1}{2n} \tag{9.18}$$

[9]The center of mass of a stack of blocks is the average of the centers of mass of the individual blocks.

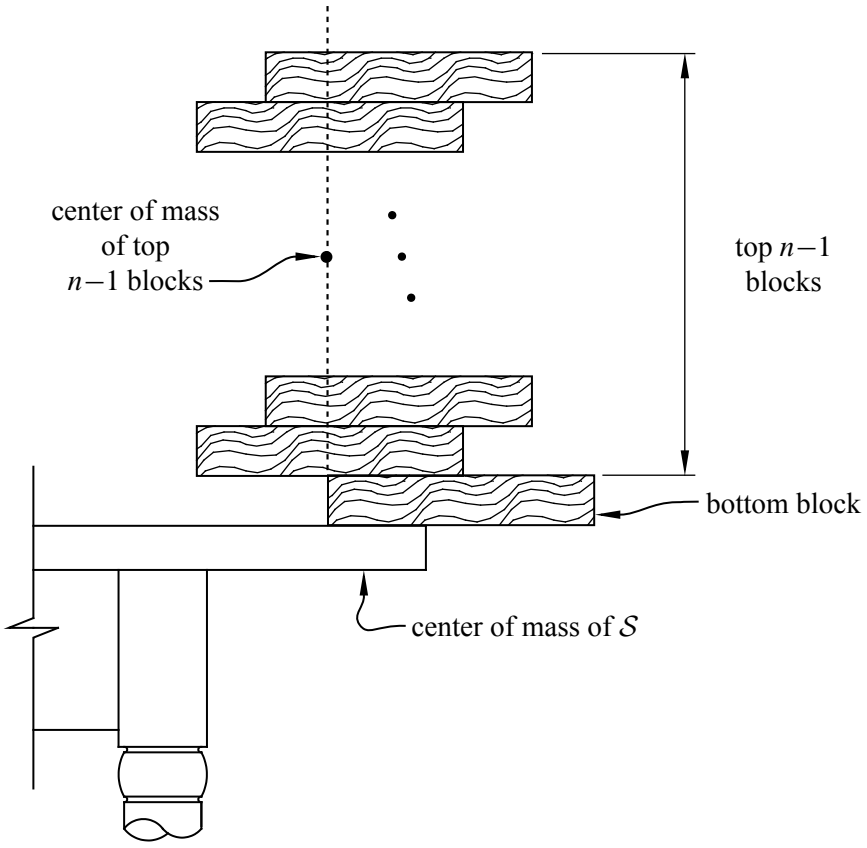


Figure 9.8 The scenario where the bottom block is the rightmost block. In this case, the spread is maximized by having the center of mass of the top $n-1$ blocks be directly over the left edge of the bottom block.

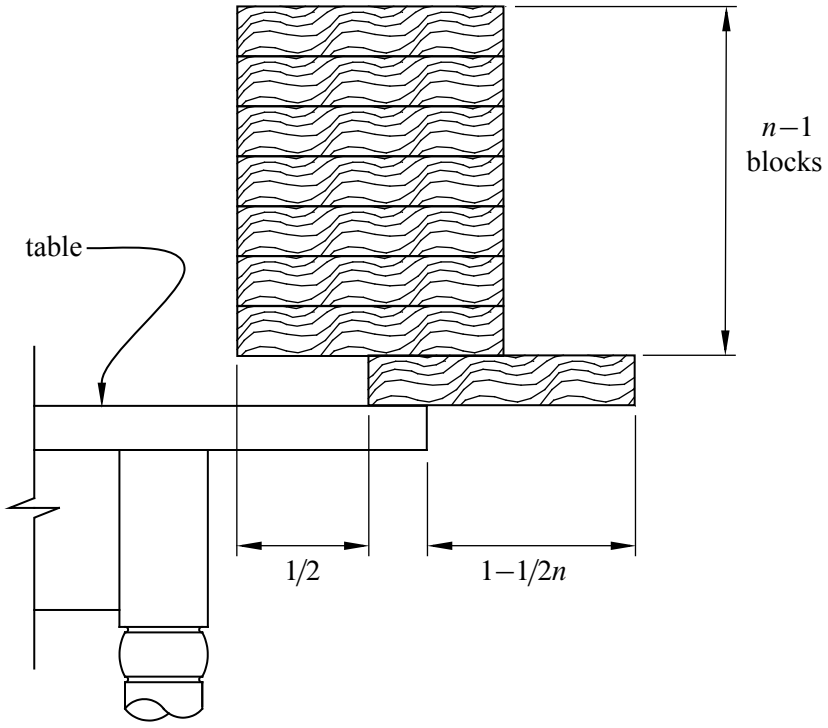


Figure 9.9 A method for achieving spread (and hence overhang) $1 - 1/2n$ with n blocks, where the bottom block is the rightmost block.

where S_{n-1} is the maximum possible spread for $n-1$ blocks (using any strategy).

We are now almost done. There are only two cases to consider when designing a stack with maximum spread and we have analyzed both of them. This means that we can combine Equation 9.17 from Case 1 with Equation 9.18 from Case 2 to conclude that

$$S_n = \max\left\{1 - \frac{1}{2n},\ S_{n-1} + \frac{1}{2n}\right\} \tag{9.19}$$

for any $n > 1$.

Uh-oh. This looks complicated. Maybe we are not almost done after all!

Equation 9.19 is an example of a *recurrence*. We will describe numerous techniques for solving recurrences in Chapter 10, but, fortunately, Equation 9.19 is simple enough that we can solve it without waiting for all the hardware in Chapter 10.

One of the first things to do when you have a recurrence is to get a feel for it by computing the first few terms. This often gives clues about a way to solve the recurrence, as it will in this case.

We already know that $S_1 = 1/2$. What about S_2? From Equation 9.19, we find that

$$\begin{aligned} S_2 &= \max\left\{1 - \frac{1}{4},\ \frac{1}{2} + \frac{1}{4}\right\} \\ &= 3/4. \end{aligned}$$

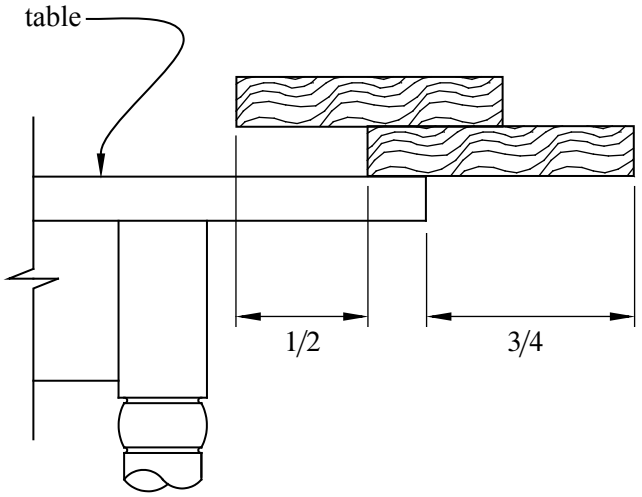
Both cases give the same spread, albeit by different approaches. For example, see Figure 9.10.

That was easy enough. What about S_3?

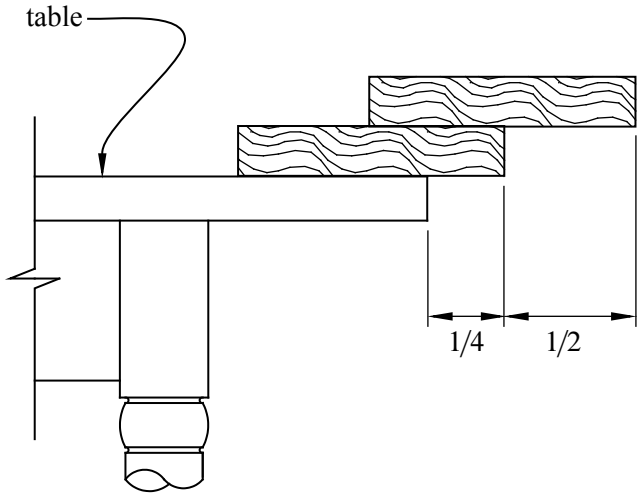
$$\begin{aligned} S_3 &= \max\left\{1 - \frac{1}{6},\ \frac{3}{4} + \frac{1}{6}\right\} \\ &= \max\left\{\frac{5}{6},\ \frac{11}{12}\right\} \\ &= \frac{11}{12}. \end{aligned}$$

As we can see, the method provided by Case 2 is the best. Let's check $n = 4$.

$$\begin{aligned} S_4 &= \max\left\{1 - \frac{1}{8},\ \frac{11}{12} + \frac{1}{8}\right\} \\ &= \frac{25}{24}. \end{aligned} \tag{9.20}$$



(a)



(b)

Figure 9.10 Two ways to achieve spread (and hence overhang) $3/4$ with $n = 2$ blocks. The first way (a) is from Case 1 and the second (b) is from Case 2.

Wow! This is a breakthrough—for two reasons. First, Equation 9.20 tells us that by using only 4 blocks, we can make a stack so that one of the blocks is hanging out completely over the edge of the table. The two ways to do this are shown in Figure 9.11.

The second reason that Equation 9.20 is important is that we now know that $S_4 > 1$, which means that we no longer have to worry about Case 1 for $n > 4$ since Case 1 never achieves spread greater than 1. Moreover, even for $n \leq 4$, we have now seen that the spread achieved by Case 1 never exceeds the spread achieved by Case 2, and they can be equal only for $n = 1$ and $n = 2$. This means that

$$S_n = S_{n-1} + \frac{1}{2n} \tag{9.21}$$

for all $n > 1$ since we have shown that the best spread can always be achieved using Case 2.

The recurrence in Equation 9.21 is much easier to solve than the one we started with in Equation 9.19. We can solve it by expanding the equation as follows:

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

and so on. This suggests that

$$S_n = \sum_{i=1}^{n} \frac{1}{2i}, \tag{9.22}$$

which is, indeed, the case.

Equation 9.22 can be verified by induction. The base case when $n = 1$ is true since we know that $S_1 = 1/2$. The inductive step follows from Equation 9.21.

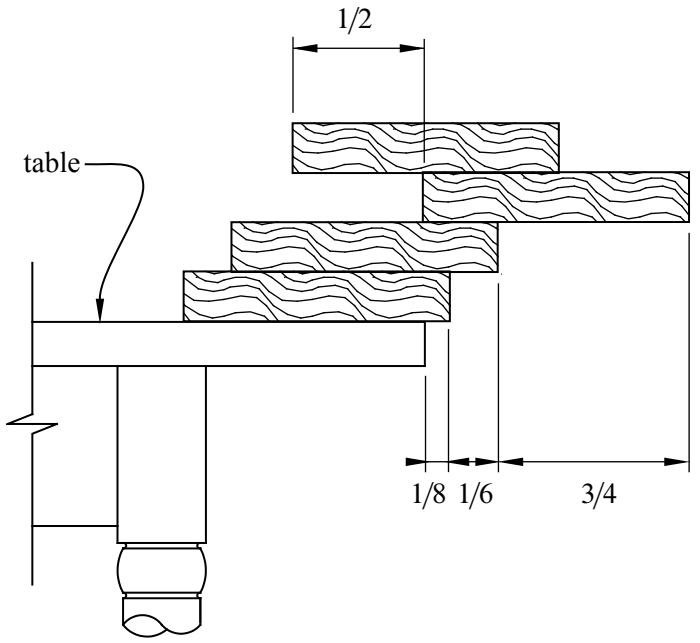
So we now know the maximum possible spread and hence the maximum possible overhang for any stable stack of books. Are we done? Not quite. Although we know that $S_4 > 1$, we still don't know how big the sum $\sum_{i=1}^{n} \frac{1}{2i}$ can get.

It turns out that S_n is very close to a famous sum known as the nth Harmonic number H_n.

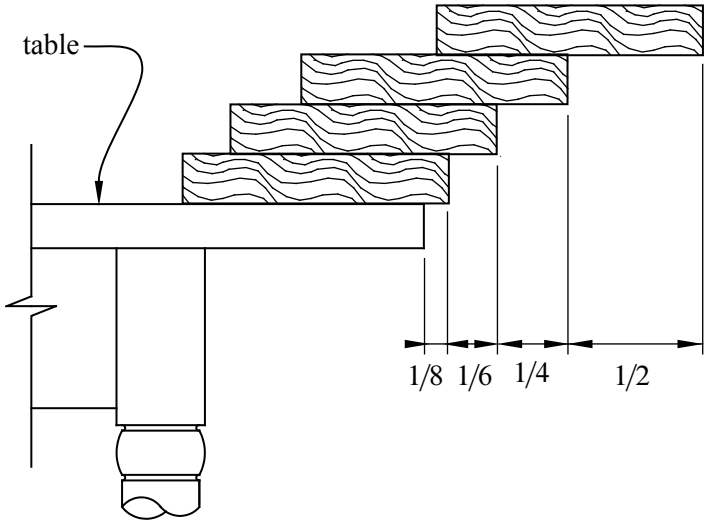
9.4.3 Harmonic Numbers

Definition 9.4.1. The *nth Harmonic number* is

$$H_n ::= \sum_{i=1}^{n} \frac{1}{i}.$$



(a)



(b)

Figure 9.11 The two ways to achieve spread (and overhang) $25/24$. The method in (a) uses Case 1 for the top 2 blocks and Case 2 for the others. The method in (b) uses Case 2 for every block that is added to the stack.

So Equation 9.22 means that

$$S_n = \frac{H_n}{2}. \tag{9.23}$$

The first few Harmonic numbers are easy to compute. For example,

$$H_4 = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} = \frac{25}{12}.$$

There is good news and bad news about Harmonic numbers. The bad news is that there is no closed-form expression known for the Harmonic numbers. The good news is that we can use Theorem 9.3.1 to get close upper and lower bounds on H_n. In particular, since

$$\int_1^n \frac{1}{x}\,dx = \ln(x)\Big|_1^n = \ln(n),$$

Theorem 9.3.1 means that

$$\ln(n) + \frac{1}{n} \leq H_n \leq \ln(n) + 1. \tag{9.24}$$

In other words, the nth Harmonic number is very close to $\ln(n)$.

Because the Harmonic numbers frequently arise in practice, mathematicians have worked hard to get even better approximations for them. In fact, it is now known that

$$H_n = \ln(n) + \gamma + \frac{1}{2n} + \frac{1}{12n^2} + \frac{\epsilon(n)}{120n^4} \tag{9.25}$$

Here γ is a value $0.577215664\ldots$ called *Euler's constant*, and $\epsilon(n)$ is between 0 and 1 for all n. We will not prove this formula.

We are now finally done with our analysis of the block stacking problem. Plugging the value of H_n into Equation 9.23, we find that the maximum overhang for n blocks is very close to $\frac{1}{2}\ln(n)$. Since $\ln(n)$ grows to infinity as n increases, this means that if we are given enough blocks (in theory anyway), we can get a block to hang out arbitrarily far over the edge of the table. Of course, the number of blocks we need will grow as an exponential function of the overhang, so it will probably take you a long time to achieve an overhang of 2 or 3, never mind an overhang of 100.

9.4.4 Asymptotic Equality

For cases like Equation 9.25 where we understand the growth of a function like H_n up to some (unimportant) error terms, we use a special notation, $\sim$, to denote the leading term of the function. For example, we say that $H_n \sim \ln(n)$ to indicate that the leading term of H_n is $\ln(n)$. More precisely:

Definition 9.4.2. For functions $f, g : \mathbb{R} \to \mathbb{R}$, we say f is *asymptotically equal* to g, in symbols,

$$f(x) \sim g(x)$$

iff

$$\lim_{x\to\infty} f(x)/g(x) = 1.$$

Although it is tempting to write $H_n \sim \ln(n) + \gamma$ to indicate the two leading terms, this is not really right. According to Definition 9.4.2, $H_n \sim \ln(n) + c$ where c is *any constant*. The correct way to indicate that γ is the second-largest term is $H_n - \ln(n) \sim \gamma$.

The reason that the $\sim$ notation is useful is that often we do not care about lower order terms. For example, if $n = 100$, then we can compute $H(n)$ to great precision using only the two leading terms:

$$|H_n - \ln(n) - \gamma| \le \left|\frac{1}{200} - \frac{1}{120000} + \frac{1}{120 \cdot 100^4}\right| < \frac{1}{200}.$$

We will spend a lot more time talking about asymptotic notation at the end of the chapter. But for now, let's get back to sums.

9.5 Double Trouble

Sometimes we have to evaluate sums of sums, otherwise known as *double summations*. This sounds hairy, and sometimes it is. But usually, it is straightforward—you just evaluate the inner sum, replace it with a closed form, and then evaluate the

outer sum (which no longer has a summation inside it). For example,[10]

$$\begin{aligned}
\sum_{n=0}^{\infty}\left(y^n\sum_{i=0}^{n}x^i\right) &= \sum_{n=0}^{\infty}\left(y^n\frac{1-x^{n+1}}{1-x}\right) && \text{Equation 9.3}\\
&= \left(\frac{1}{1-x}\right)\sum_{n=0}^{\infty}y^n - \left(\frac{1}{1-x}\right)\sum_{n=0}^{\infty}y^n x^{n+1}\\
&= \frac{1}{(1-x)(1-y)} - \left(\frac{x}{1-x}\right)\sum_{n=0}^{\infty}(xy)^n && \text{Theorem 9.1.1}\\
&= \frac{1}{(1-x)(1-y)} - \frac{x}{(1-x)(1-xy)} && \text{Theorem 9.1.1}\\
&= \frac{(1-xy)-x(1-y)}{(1-x)(1-y)(1-xy)}\\
&= \frac{1-x}{(1-x)(1-y)(1-xy)}\\
&= \frac{1}{(1-y)(1-xy)}.
\end{aligned}$$

When there's no obvious closed form for the inner sum, a special trick that is often useful is to try *exchanging the order of summation.* For example, suppose we want to compute the sum of the first n Harmonic numbers

$$\sum_{k=1}^{n}H_k = \sum_{k=1}^{n}\sum_{j=1}^{k}\frac{1}{j} \tag{9.26}$$

For intuition about this sum, we can apply Theorem 9.3.1 to Equation 9.24 to conclude that the sum is close to

$$\int_1^n \ln(x)\,dx = x\ln(x) - x\Big|_1^n = n\ln(n) - n + 1.$$

Now let's look for an exact answer. If we think about the pairs (k, j) over which

[10] Ok, so maybe this one is a little hairy, but it is also fairly straightforward. Wait till you see the next one!

we are summing, they form a triangle:

		j						
		1	2	3	4	5	...	n
k	1	1						
	2	1	1/2					
	3	1	1/2	1/3				
	4	1	1/2	1/3	1/4			
		...						
	n	1	1/2		...			$1/n$

The summation in Equation 9.26 is summing each row and then adding the row sums. Instead, we can sum the columns and then add the column sums. Inspecting the table we see that this double sum can be written as

$$\begin{aligned}
\sum_{k=1}^{n} H_k &= \sum_{k=1}^{n}\sum_{j=1}^{k}\frac{1}{j} \\
&= \sum_{j=1}^{n}\sum_{k=j}^{n}\frac{1}{j} \\
&= \sum_{j=1}^{n}\frac{1}{j}\sum_{k=j}^{n}1 \\
&= \sum_{j=1}^{n}\frac{1}{j}(n-j+1) \\
&= \sum_{j=1}^{n}\frac{n+1}{j}-\sum_{j=1}^{n}\frac{j}{j} \\
&= (n+1)\sum_{j=1}^{n}\frac{1}{j}-\sum_{j=1}^{n}1 \\
&= (n+1)H_n - n. \qquad (9.27)
\end{aligned}$$

9.6 Products

We've covered several techniques for finding closed forms for sums but no methods for dealing with products. Fortunately, we do not need to develop an entirely new set of tools when we encounter a product such as

$$n! ::= \prod_{i=1}^{n} i. \tag{9.28}$$

That's because we can convert any product into a sum by taking a logarithm. For example, if

$$P = \prod_{i=1}^{n} f(i),$$

then

$$\ln(P) = \sum_{i=1}^{n} \ln(f(i)).$$

We can then apply our summing tools to find a closed form (or approximate closed form) for $\ln(P)$ and then exponentiate at the end to undo the logarithm.

For example, let's see how this works for the factorial function $n!$ We start by taking the logarithm:

$$\begin{aligned}
\ln(n!) &= \ln(1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n) \\
&= \ln(1) + \ln(2) + \ln(3) + \cdots + \ln(n-1) + \ln(n) \\
&= \sum_{i=1}^{n} \ln(i).
\end{aligned}$$

Unfortunately, no closed form for this sum is known. However, we can apply Theorem 9.3.1 to find good closed-form bounds on the sum. To do this, we first compute

$$\begin{aligned}
\int_1^n \ln(x)\, dx &= x \ln(x) - x \Big|_1^n \\
&= n \ln(n) - n + 1.
\end{aligned}$$

Plugging into Theorem 9.3.1, this means that

$$n \ln(n) - n + 1 \;\le\; \sum_{i=1}^{n} \ln(i) \;\le\; n \ln(n) - n + 1 + \ln(n).$$

Exponentiating then gives

$$\frac{n^n}{e^{n-1}} \leq n! \leq \frac{n^{n+1}}{e^{n-1}}. \tag{9.29}$$

This means that $n!$ is within a factor of n of n^n/e^{n-1}.

9.6.1 Stirling's Formula

$n!$ is probably the most commonly used product in discrete mathematics, and so mathematicians have put in the effort to find much better closed-form bounds on its value. The most useful bounds are given in Theorem 9.6.1.

Theorem 9.6.1 (Stirling's Formula)**.** *For all $n \geq 1$,*

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\epsilon(n)}$$

where

$$\frac{1}{12n+1} \leq \epsilon(n) \leq \frac{1}{12n}.$$

Theorem 9.6.1 can be proved by induction on n, but the details are a bit painful (even for us) and so we will not go through them here.

There are several important things to notice about Stirling's Formula. First, $\epsilon(n)$ is always positive. This means that

$$n! > \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \tag{9.30}$$

for all $n \in \mathbb{N}^+$.

Second, $\epsilon(n)$ tends to zero as n gets large. This means that[11]

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n, \tag{9.31}$$

which is rather surprising. After all, who would expect both π and e to show up in a closed-form expression that is asymptotically equal to $n!$?

Third, $\epsilon(n)$ is small even for small values of n. This means that Stirling's Formula provides good approximations for $n!$ for most all values of n. For example, if we use

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

[11] The $\sim$ notation was defined in Section 9.4.4.

Approximation	$n \geq 1$	$n \geq 10$	$n \geq 100$	$n \geq 1000$
$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$	$< 10\%$	$< 1\%$	$< 0.1\%$	$< 0.01\%$
$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{1/12n}$	$< 1\%$	$< 0.01\%$	$< 0.0001\%$	$< 0.000001\%$

Table 9.1 Error bounds on common approximations for $n!$ from Theorem 9.6.1. For example, if $n \geq 100$, then $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ approximates $n!$ to within 0.1%.

as the approximation for $n!$, as many people do, we are guaranteed to be within a factor of

$$e^{\epsilon(n)} \leq e^{\frac{1}{12n}}$$

of the correct value. For $n \geq 10$, this means we will be within 1% of the correct value. For $n \geq 100$, the error will be less than 0.1%.

If we need an even closer approximation for $n!$, then we could use either

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{1/12n}$$

or

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{1/(12n+1)}$$

depending on whether we want an upper bound or a lower bound, respectively. By Theorem 9.6.1, we know that both bounds will be within a factor of

$$e^{\frac{1}{12n}-\frac{1}{12n+1}} = e^{\frac{1}{144n^2+12n}}$$

of the correct value. For $n \geq 10$, this means that either bound will be within 0.01% of the correct value. For $n \geq 100$, the error will be less than 0.0001%.

For quick future reference, these facts are summarized in Corollary 9.6.2 and Table 9.1.

Corollary 9.6.2. *For $n \geq 1$,*

$$n! < 1.09\sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

For $n \geq 10$,

$$n! < 1.009\sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

For $n \geq 100$,

$$n! < 1.0009\sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

9.7 Asymptotic Notation

Asymptotic notation is a shorthand used to give a quick measure of the behavior of a function $f(n)$ as n grows large. For example, the asymptotic notation $\sim$ of Definition 9.4.2 is a binary relation indicating that two functions grow at the *same* rate. There is also a binary relation indicating that one function grows at a significantly *slower* rate than another.

9.7.1 Little Oh

Definition 9.7.1. For functions $f, g : \mathbb{R} \to \mathbb{R}$, with g nonnegative, we say f is *asymptotically smaller* than g, in symbols,

$$f(x) = o(g(x)),$$

iff

$$\lim_{x\to\infty} f(x)/g(x) = 0.$$

For example, $1000x^{1.9} = o(x^2)$, because $1000x^{1.9}/x^2 = 1000/x^{0.1}$ and since $x^{0.1}$ goes to infinity with x and 1000 is constant, we have $\lim_{x\to\infty} 1000x^{1.9}/x^2 = 0$. This argument generalizes directly to yield

Lemma 9.7.2. *$x^a = o(x^b)$ for all nonnegative constants $a < b$.*

Using the familiar fact that $\log x < x$ for all $x > 1$, we can prove

Lemma 9.7.3. *$\log x = o(x^\epsilon)$ for all $\epsilon > 0$.*

Proof. Choose $\epsilon > \delta > 0$ and let $x = z^\delta$ in the inequality $\log x < x$. This implies

$$\log z < z^\delta/\delta = o(z^\epsilon) \qquad \text{by Lemma 9.7.2.} \tag{9.32}$$

■

Corollary 9.7.4. *$x^b = o(a^x)$ for any $a, b \in \mathbb{R}$ with $a > 1$.*

Lemma 9.7.3 and Corollary 9.7.4 can also be proved using l'Hôpital's Rule or the McLaurin Series for $\log x$ and e^x. Proofs can be found in most calculus texts.

9.7.2 Big Oh

Big Oh is the most frequently used asymptotic notation. It is used to give an upper bound on the growth of a function, such as the running time of an algorithm.

Definition 9.7.5. Given nonnegative functions $f, g : \mathbb{R} \to \mathbb{R}$, we say that

$$f = O(g)$$

iff

$$\limsup_{x\to\infty} f(x)/g(x) < \infty.$$

This definition[12] makes it clear that

Lemma 9.7.6. *If $f = o(g)$ or $f \sim g$, then $f = O(g)$.*

Proof. $\lim f/g = 0$ or $\lim f/g = 1$ implies $\lim f/g < \infty$. ■

It is easy to see that the converse of Lemma 9.7.6 is not true. For example, $2x = O(x)$, but $2x \not\sim x$ and $2x \neq o(x)$.

The usual formulation of Big Oh spells out the definition of lim sup without mentioning it. Namely, here is an equivalent definition:

Definition 9.7.7. Given functions $f, g : \mathbb{R} \to \mathbb{R}$, we say that

$$f = O(g)$$

iff there exists a constant $c \geq 0$ and an x_0 such that for all $x \geq x_0$, $|f(x)| \leq cg(x)$.

This definition is rather complicated, but the idea is simple: $f(x) = O(g(x))$ means $f(x)$ is less than or equal to $g(x)$, except that we're willing to ignore a constant factor, namely, c, and to allow exceptions for small x, namely, $x < x_0$.

We observe,

Lemma 9.7.8. *If $f = o(g)$, then it is* not *true that $g = O(f)$.*

[12] We can't simply use the limit as $x \to \infty$ in the definition of $O()$, because if $f(x)/g(x)$ oscillates between, say, 3 and 5 as x grows, then $f = O(g)$ because $f \leq 5g$, but $\lim_{x\to\infty} f(x)/g(x)$ does not exist. So instead of limit, we use the technical notion of lim sup. In this oscillating case, $\limsup_{x\to\infty} f(x)/g(x) = 5$.

The precise definition of lim sup is

$$\limsup_{x\to\infty} h(x) ::= \lim_{x\to\infty} \text{lub}_{y\geq x} h(y),$$

where "lub" abbreviates "least upper bound."

Proof.

$$\lim_{x\to\infty} \frac{g(x)}{f(x)} = \frac{1}{\lim_{x\to\infty} f(x)/g(x)} = \frac{1}{0} = \infty,$$

so $g \neq O(f)$.

■

Proposition 9.7.9. $100x^2 = O(x^2)$.

Proof. Choose $c = 100$ and $x_0 = 1$. Then the proposition holds, since for all $x \geq 1$, $\left|100x^2\right| \leq 100x^2$. ■

Proposition 9.7.10. $x^2 + 100x + 10 = O(x^2)$.

Proof. $(x^2+100x+10)/x^2 = 1+100/x+10/x^2$ and so its limit as x approaches infinity is $1+0+0 = 1$. So in fact, $x^2+100x+10 \sim x^2$, and therefore $x^2+100x+10 = O(x^2)$. Indeed, it's conversely true that $x^2 = O(x^2 + 100x + 10)$. ■

Proposition 9.7.10 generalizes to an arbitrary polynomial:

Proposition 9.7.11. $a_k x^k + a_{k-1}x^{k-1} + \cdots + a_1 x + a_0 = O(x^k)$.

We'll omit the routine proof.

Big Oh notation is especially useful when describing the running time of an algorithm. For example, the usual algorithm for multiplying $n \times n$ matrices uses a number of operations proportional to n^3 in the worst case. This fact can be expressed concisely by saying that the running time is $O(n^3)$. So this asymptotic notation allows the speed of the algorithm to be discussed without reference to constant factors or lower-order terms that might be machine specific. It turns out that there is another, ingenious matrix multiplication procedure that uses $O(n^{2.55})$ operations. This procedure will therefore be much more efficient on large enough matrices. Unfortunately, the $O(n^{2.55})$-operation multiplication procedure is almost never used in practice because it happens to be less efficient than the usual $O(n^3)$ procedure on matrices of practical size.[13]

9.7.3 Omega

Suppose you want to make a statement of the form "the running time of the algorithm is a least…". Can you say it is "at least $O(n^2)$"? No! This statement is meaningless since big-oh can only be used for *upper* bounds. For lower bounds, we use a different symbol, called "big-Omega."

[13] It is even conceivable that there is an $O(n^2)$ matrix multiplication procedure, but none is known.

Definition 9.7.12. Given functions $f, g : \mathbb{R} \to \mathbb{R}$, we say that

$$f = \Omega(g)$$

iff there exists a constant $c > 0$ and an x_0 such that for all $x \geq x_0$, we have $f(x) \geq c|g(x)|$.

In other words, $f(x) = \Omega(g(x))$ means that $f(x)$ is greater than or equal to $g(x)$, except that we are willing to ignore a constant factor and to allow exceptions for small x.

If all this sounds a lot like big-Oh, only in reverse, that's because big-Omega is the opposite of big-Oh. More precisely,

Theorem 9.7.13. *$f(x) = O(g(x))$ if and only if $g(x) = \Omega(f(x))$.*

Proof.

$$\begin{aligned}
&f(x) = O(g(x)) \\
&\quad \text{iff} \quad \exists c > 0, x_0.\ \forall x \geq x_0.\ |f(x)| \leq c g(x) && \text{(Definition 9.7.7)} \\
&\quad \text{iff} \quad \exists c > 0, x_0.\ \forall x \geq x_0.\ g(x) \geq \frac{1}{c}|f(x)| \\
&\quad \text{iff} \quad \exists c' > 0, x_0.\ \forall x \geq x_0.\ g(x) \geq c'|f(x)| && \text{(set } c' = 1/c\text{)} \\
&\quad \text{iff} \quad g(x) = \Omega(f(x)) && \text{(Definition 9.7.12)}
\end{aligned}$$

■

For example, $x^2 = \Omega(x)$, $2^x = \Omega(x^2)$, and $x/100 = \Omega(100x + \sqrt{x})$.

So if the running time of your algorithm on inputs of size n is $T(n)$, and you want to say it is at least quadratic, say

$$T(n) = \Omega(n^2).$$

Little Omega

There is also a symbol called little-omega, analogous to little-oh, to denote that one function grows strictly faster than another function.

Definition 9.7.14. For functions $f, g : \mathbb{R} \to \mathbb{R}$ with f nonnegative, we say that

$$f(x) = \omega(g(x))$$

iff

$$\lim_{x \to \infty} \frac{g(x)}{f(x)} = 0.$$

In other words,

$$f(x) = \omega(g(x))$$

iff

$$g(x) = o(f(x)).$$

For example, $x^{1.5} = \omega(x)$ and $\sqrt{x} = \omega(\ln^2(x))$.

The little-omega symbol is not as widely used as the other asymptotic symbols we have been discussing.

9.7.4 Theta

Sometimes we want to specify that a running time $T(n)$ is precisely quadratic up to constant factors (both upper bound *and* lower bound). We could do this by saying that $T(n) = O(n^2)$ and $T(n) = \Omega(n^2)$, but rather than say both, mathematicians have devised yet another symbol, Θ, to do the job.

Definition 9.7.15.

$$f = \Theta(g) \quad \text{iff} \quad f = O(g) \text{ and } g = O(f).$$

The statement $f = \Theta(g)$ can be paraphrased intuitively as "f and g are equal to within a constant factor." Indeed, by Theorem 9.7.13, we know that

$$f = \Theta(g) \quad \text{iff} \quad f = O(g) \text{ and } f = \Omega(g).$$

The Theta notation allows us to highlight growth rates and allow suppression of distracting factors and low-order terms. For example, if the running time of an algorithm is

$$T(n) = 10n^3 - 20n^2 + 1,$$

then we can more simply write

$$T(n) = \Theta(n^3).$$

In this case, we would say that T *is of order* n^3 or that $T(n)$ *grows cubically*, which is probably what we really want to know. Another such example is

$$\pi^2 3^{x-7} + \frac{(2.7x^{113} + x^9 - 86)^4}{\sqrt{x}} - 1.08^{3x} = \Theta(3^x).$$

Just knowing that the running time of an algorithm is $\Theta(n^3)$, for example, is useful, because if n doubles we can predict that the running time will *by and large*[14] increase by a factor of at most 8 for large n. In this way, Theta notation preserves information about the scalability of an algorithm or system. Scalability is, of course, a big issue in the design of algorithms and systems.

[14]Since $\Theta(n^3)$ only implies that the running time, $T(n)$, is between cn^3 and dn^3 for constants $0 < c < d$, the time $T(2n)$ could regularly exceed $T(n)$ by a factor as large as $8d/c$. The factor is sure to be close to 8 for all large n only if $T(n) \sim n^3$.

9.7.5 Pitfalls with Asymptotic Notation

There is a long list of ways to make mistakes with asymptotic notation. This section presents some of the ways that Big Oh notation can lead to ruin and despair. With minimal effort, you can cause just as much chaos with the other symbols.

The Exponential Fiasco

Sometimes relationships involving Big Oh are not so obvious. For example, one might guess that $4^x = O(2^x)$ since 4 is only a constant factor larger than 2. This reasoning is incorrect, however; 4^x actually grows as the square of 2^x.

Constant Confusion

Every constant is $O(1)$. For example, $17 = O(1)$. This is true because if we let $f(x) = 17$ and $g(x) = 1$, then there exists a $c > 0$ and an x_0 such that $|f(x)| \leq cg(x)$. In particular, we could choose $c = 17$ and $x_0 = 1$, since $|17| \leq 17 \cdot 1$ for all $x \geq 1$. We can construct a false theorem that exploits this fact.

False Theorem 9.7.16.

$$\sum_{i=1}^{n} i = O(n)$$

Bogus proof. Define $f(n) = \sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n$. Since we have shown that every constant i is $O(1)$, $f(n) = O(1) + O(1) + \cdots + O(1) = O(n)$. ■

Of course in reality $\sum_{i=1}^{n} i = n(n+1)/2 \neq O(n)$.

The error stems from confusion over what is meant in the statement $i = O(1)$. For any *constant* $i \in \mathbb{N}$ it is true that $i = O(1)$. More precisely, if f is any constant function, then $f = O(1)$. But in this False Theorem, i is not constant—it ranges over a set of values 0, 1,..., n that depends on n.

And anyway, we should not be adding $O(1)$'s as though they were numbers. We never even defined what $O(g)$ means by itself; it should only be used in the context "$f = O(g)$" to describe a relation between functions f and g.

Lower Bound Blunder

Sometimes people incorrectly use Big Oh in the context of a lower bound. For example, they might say, "The running time, $T(n)$, is at least $O(n^2)$," when they probably mean[15] "$T(n) = \Omega(n^2)$."

[15] This can also be correctly expressed as $n^2 = O(T(n))$, but such notation is rare.

Equality Blunder

The notation $f = O(g)$ is too firmly entrenched to avoid, but the use of "=" is really regrettable. For example, if $f = O(g)$, it seems quite reasonable to write $O(g) = f$. But doing so might tempt us to the following blunder: because $2n = O(n)$, we can say $O(n) = 2n$. But $n = O(n)$, so we conclude that $n = O(n) = 2n$, and therefore $n = 2n$. To avoid such nonsense, we will never write "$O(f) = g$."

Similarly, you will often see statements like

$$H_n = \ln(n) + \gamma + O\left(\frac{1}{n}\right)$$

or

$$n! = (1 + o(1))\sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

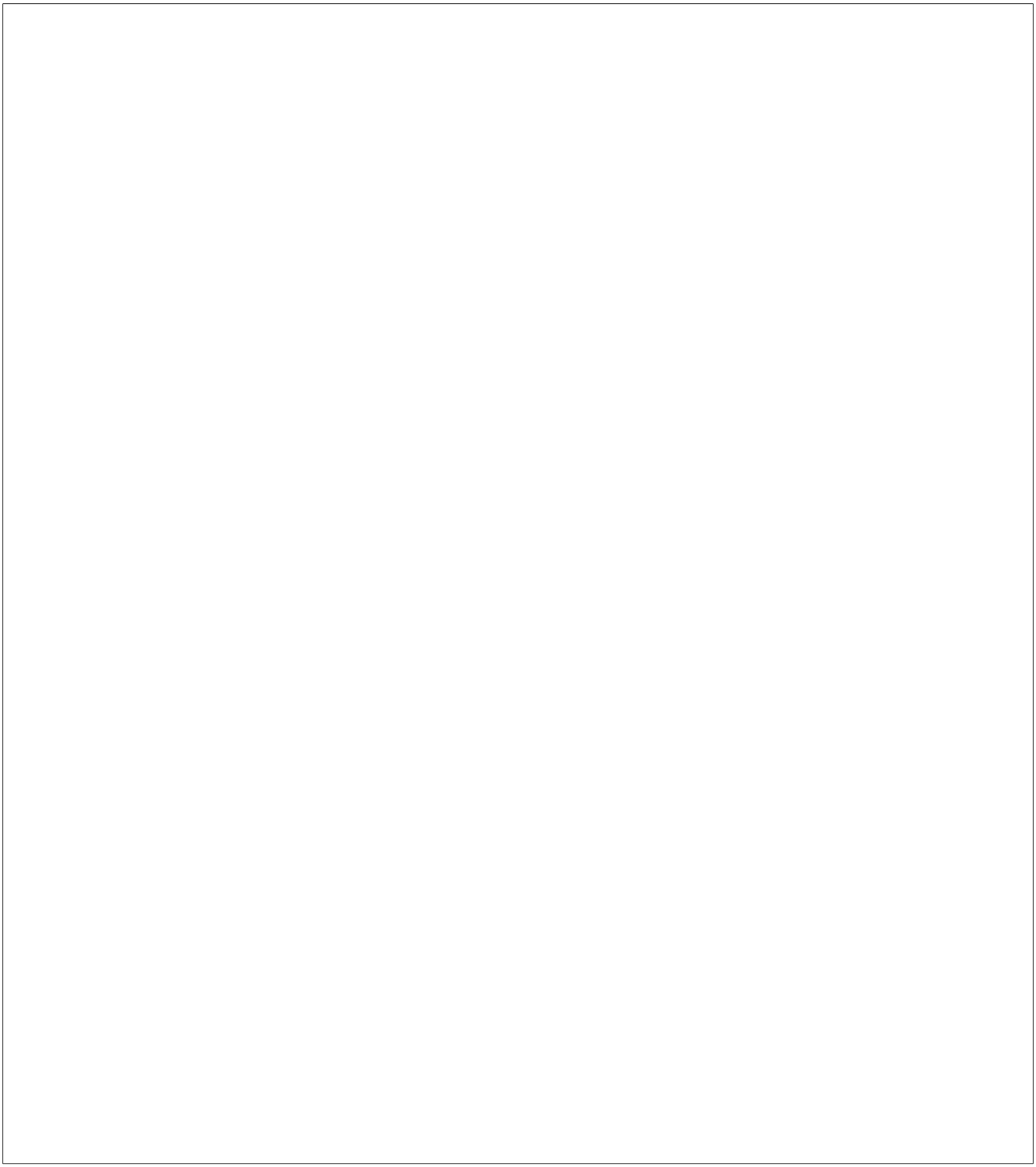
In such cases, the true meaning is

$$H_n = \ln(n) + \gamma + f(n)$$

for some $f(n)$ where $f(n) = O(1/n)$, and

$$n! = (1 + g(n))\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

where $g(n) = o(1)$. These transgressions are OK as long as you (and you reader) know what you mean.



10 Recurrences

A recurrence describes a sequence of numbers. Early terms are specified explicitly and later terms are expressed as a function of their predecessors. As a trivial example, this recurrence describes the sequence 1, 2, 3, etc.:

$$\begin{aligned} T_1 &= 1 \\ T_n &= T_{n-1} + 1 \qquad (\text{for } n \geq 2). \end{aligned}$$

Here, the first term is defined to be 1 and each subsequent term is one more than its predecessor.

Recurrences turn out to be a powerful tool. In this chapter, we'll emphasize using recurrences to analyze the performance of recursive algorithms. However, recurrences have other applications in computer science as well, such as enumeration of structures and analysis of random processes. And, as we saw in Section 9.4, they also arise in the analysis of problems in the physical sciences.

A recurrence in isolation is not a very useful description of a sequence. One can not easily answer simple questions such as, "What is the hundredth term?" or "What is the asymptotic growth rate?" So one typically wants to *solve* a recurrence; that is, to find a closed-form expression for the nth term.

We'll first introduce two general solving techniques: guess-and-verify and plug-and-chug. These methods are applicable to every recurrence, but their success requires a flash of insight—sometimes an unrealistically brilliant flash. So we'll also introduce two big classes of recurrences, linear and divide-and-conquer, that often come up in computer science. Essentially all recurrences in these two classes are solvable using cookbook techniques; you follow the recipe and get the answer. A drawback is that calculation replaces insight. The "Aha!" moment that is essential in the guess-and-verify and plug-and-chug methods is replaced by a "Huh" at the end of a cookbook procedure.

At the end of the chapter, we'll develop rules of thumb to help you assess many recurrences without any calculation. These rules can help you distinguish promising approaches from bad ideas early in the process of designing an algorithm.

Recurrences are one aspect of a broad theme in computer science: reducing a big problem to progressively smaller problems until easy base cases are reached. This same idea underlies both induction proofs and recursive algorithms. As we'll see, all three ideas snap together nicely. For example, one might describe the running time of a recursive algorithm with a recurrence and use induction to verify the solution.

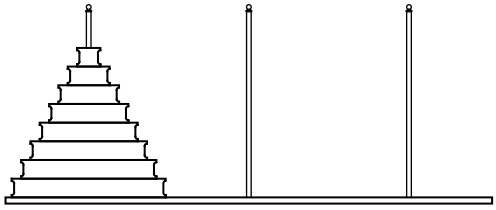


Figure 10.1 The initial configuration of the disks in the Towers of Hanoi problem.

10.1 The Towers of Hanoi

According to legend, there is a temple in Hanoi with three posts and 64 gold disks of different sizes. Each disk has a hole through the center so that it fits on a post. In the misty past, all the disks were on the first post, with the largest on the bottom and the smallest on top, as shown in Figure 10.1.

Monks in the temple have labored through the years since to move all the disks to one of the other two posts according to the following rules:

- The only permitted action is removing the top disk from one post and dropping it onto another post.
- A larger disk can never lie above a smaller disk on any post.

So, for example, picking up the whole stack of disks at once and dropping them on another post is illegal. That's good, because the legend says that when the monks complete the puzzle, the world will end!

To clarify the problem, suppose there were only 3 gold disks instead of 64. Then the puzzle could be solved in 7 steps as shown in Figure 10.2.

The questions we must answer are, "Given sufficient time, can the monks succeed?" If so, "How long until the world ends?" And, most importantly, "Will this happen before the final exam?"

10.1.1 A Recursive Solution

The Towers of Hanoi problem can be solved recursively. As we describe the procedure, we'll also analyze the running time. To that end, let T_n be the minimum number of steps required to solve the n-disk problem. For example, some experimentation shows that $T_1 = 1$ and $T_2 = 3$. The procedure illustrated above shows that T_3 is at most 7, though there might be a solution with fewer steps.

The recursive solution has three stages, which are described below and illustrated in Figure 10.3. For clarity, the largest disk is shaded in the figures.

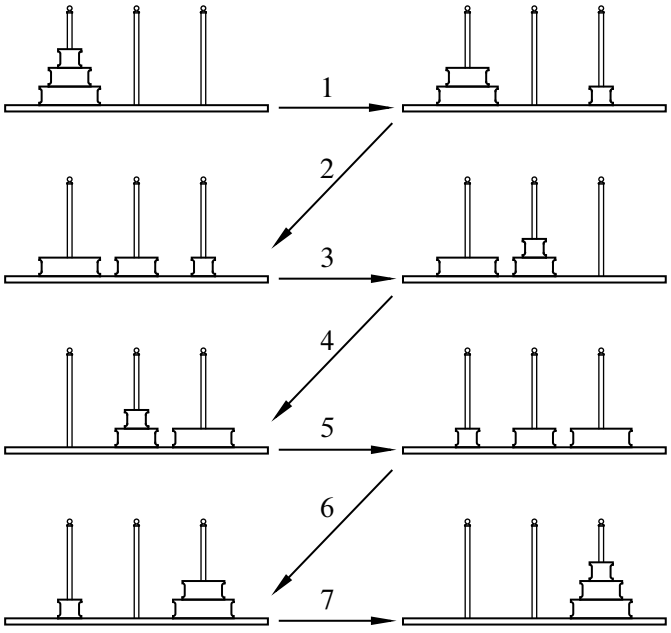


Figure 10.2 The 7-step solution to the Towers of Hanoi problem when there are $n = 3$ disks.

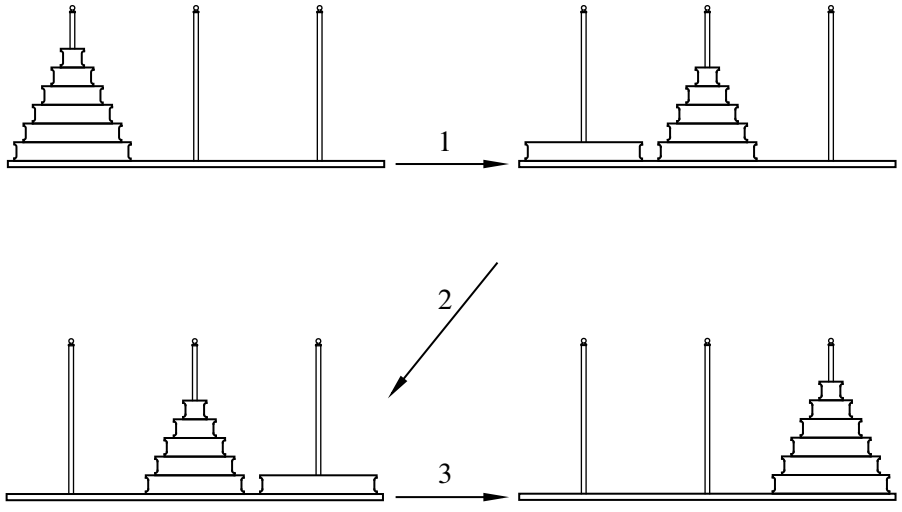


Figure 10.3 A recursive solution to the Towers of Hanoi problem.

Stage 1. Move the top $n-1$ disks from the first post to the second using the solution for $n-1$ disks. This can be done in T_{n-1} steps.

Stage 2. Move the largest disk from the first post to the third post. This takes just 1 step.

Stage 3. Move the $n-1$ disks from the second post to the third post, again using the solution for $n-1$ disks. This can also be done in T_{n-1} steps.

This algorithm shows that T_n, the minimum number of steps required to move n disks to a different post, is at most $T_{n-1}+1+T_{n-1} = 2T_{n-1}+1$. We can use this fact to upper bound the number of operations required to move towers of various heights:

$$T_3 \le 2 \cdot T_2 + 1 = 7$$
$$T_4 \le 2 \cdot T_3 + 1 \le 15$$

Continuing in this way, we could eventually compute an upper bound on T_{64}, the number of steps required to move 64 disks. So this algorithm answers our first question: given sufficient time, the monks can finish their task and end the world. This is a shame. After all that effort, they'd probably want to smack a few high-fives and go out for burgers and ice cream, but nope—world's over.

10.1.2 Finding a Recurrence

We can not yet compute the exact number of steps that the monks need to move the 64 disks, only an upper bound. Perhaps, having pondered the problem since the beginning of time, the monks have devised a better algorithm.

In fact, there is no better algorithm, and here is why. At some step, the monks must move the largest disk from the first post to a different post. For this to happen, the $n-1$ smaller disks must all be stacked out of the way on the only remaining post. Arranging the $n-1$ smaller disks this way requires at least T_{n-1} moves. After the largest disk is moved, at least another T_{n-1} moves are required to pile the $n-1$ smaller disks on top.

This argument shows that the number of steps required is at least $2T_{n-1}+1$. Since we gave an algorithm using exactly that number of steps, we can now write an expression for T_n, the number of moves required to complete the Towers of Hanoi problem with n disks:

$$T_1 = 1$$
$$T_n = 2T_{n-1}+1 \qquad (\text{for } n \ge 2).$$

This is a typical recurrence. These two lines define a sequence of values, $T_1, T_2, T_3, \ldots$. The first line says that the first number in the sequence, T_1, is equal to 1. The second line defines every other number in the sequence in terms of its predecessor. So we can use this recurrence to compute any number of terms in the sequence:

$$\begin{aligned} T_1 &= 1 \\ T_2 &= 2 \cdot T_1 + 1 = 3 \\ T_3 &= 2 \cdot T_2 + 1 = 7 \\ T_4 &= 2 \cdot T_3 + 1 = 15 \\ T_5 &= 2 \cdot T_4 + 1 = 31 \\ T_6 &= 2 \cdot T_5 + 1 = 63. \end{aligned}$$

10.1.3 Solving the Recurrence

We could determine the number of steps to move a 64-disk tower by computing T_7, T_8, and so on up to T_{64}. But that would take a lot of work. It would be nice to have a closed-form expression for T_n, so that we could quickly find the number of steps required for any given number of disks. (For example, we might want to know how much sooner the world would end if the monks melted down one disk to purchase burgers and ice cream *before* the end of the world.)

There are several methods for solving recurrence equations. The simplest is to *guess* the solution and then *verify* that the guess is correct with an induction proof. As a basis for a good guess, let's look for a pattern in the values of T_n computed above: 1, 3, 7, 15, 31, 63. A natural guess is $T_n = 2^n - 1$. But whenever you guess a solution to a recurrence, you should always verify it with a proof, typically by induction. After all, your guess might be wrong. (But why bother to verify in this case? After all, if we're wrong, its not the end of the... no, let's check.)

Claim 10.1.1. *$T_n = 2^n - 1$ satisfies the recurrence:*

$$\begin{aligned} T_1 &= 1 \\ T_n &= 2T_{n-1} + 1 \qquad \textit{(for } n \geq 2\textit{)}. \end{aligned}$$

Proof. The proof is by induction on n. The induction hypothesis is that $T_n = 2^n - 1$. This is true for $n = 1$ because $T_1 = 1 = 2^1 - 1$. Now assume that $T_{n-1} = 2^{n-1} - 1$ in order to prove that $T_n = 2^n - 1$, where $n \geq 2$:

$$\begin{aligned} T_n &= 2T_{n-1} + 1 \\ &= 2(2^{n-1} - 1) + 1 \\ &= 2^n - 1. \end{aligned}$$

The first equality is the recurrence equation, the second follows from the induction assumption, and the last step is simplification. ■

Such verification proofs are especially tidy because recurrence equations and induction proofs have analogous structures. In particular, the base case relies on the first line of the recurrence, which defines T_1. And the inductive step uses the second line of the recurrence, which defines T_n as a function of preceding terms.

Our guess is verified. So we can now resolve our remaining questions about the 64-disk puzzle. Since $T_{64} = 2^{64} - 1$, the monks must complete more than 18 billion billion steps before the world ends. Better study for the final.

10.1.4 The Upper Bound Trap

When the solution to a recurrence is complicated, one might try to prove that some simpler expression is an upper bound on the solution. For example, the exact solution to the Towers of Hanoi recurrence is $T_n = 2^n - 1$. Let's try to prove the "nicer" upper bound $T_n \leq 2^n$, proceeding exactly as before.

Proof. (Failed attempt.) The proof is by induction on n. The induction hypothesis is that $T_n \leq 2^n$. This is true for $n = 1$ because $T_1 = 1 \leq 2^1$. Now assume that $T_{n-1} \leq 2^{n-1}$ in order to prove that $T_n \leq 2^n$, where $n \geq 2$:

$$\begin{aligned} T_n &= 2T_{n-1} + 1 \\ &\leq 2(2^{n-1}) + 1 \\ &\not\leq 2^n \qquad \leftarrow \text{Uh-oh!} \end{aligned}$$

The first equality is the recurrence relation, the second follows from the induction hypothesis, and the third step is a flaming train wreck. ■

The proof doesn't work! As is so often the case with induction proofs, the argument only goes through with a *stronger* hypothesis. This isn't to say that upper bounding the solution to a recurrence is hopeless, but this is a situation where induction and recurrences do not mix well.

10.1.5 Plug and Chug

Guess-and-verify is a simple and general way to solve recurrence equations. But there is one big drawback: you have to *guess right*. That was not hard for the Towers of Hanoi example. But sometimes the solution to a recurrence has a strange form that is quite difficult to guess. Practice helps, of course, but so can some other methods.

Plug-and-chug is another way to solve recurrences. This is also sometimes called "expansion" or "iteration". As in guess-and-verify, the key step is identifying a pattern. But instead of looking at a sequence of *numbers*, you have to spot a pattern in a sequence of *expressions*, which is sometimes easier. The method consists of three steps, which are described below and illustrated with the Towers of Hanoi example.

Step 1: Plug and Chug Until a Pattern Appears

The first step is to expand the recurrence equation by alternately "plugging" (applying the recurrence) and "chugging" (simplifying the result) until a pattern appears. Be careful: too much simplification can make a pattern harder to spot. The rule to remember—indeed, a rule applicable to the whole of college life—is *chug in moderation*.

$$\begin{aligned}
T_n &= 2T_{n-1} + 1 \\
&= 2(2T_{n-2} + 1) + 1 && \text{plug} \\
&= 4T_{n-2} + 2 + 1 && \text{chug} \\
&= 4(2T_{n-3} + 1) + 2 + 1 && \text{plug} \\
&= 8T_{n-3} + 4 + 2 + 1 && \text{chug} \\
&= 8(2T_{n-4} + 1) + 4 + 2 + 1 && \text{plug} \\
&= 16T_{n-4} + 8 + 4 + 2 + 1 && \text{chug}
\end{aligned}$$

Above, we started with the recurrence equation. Then we replaced T_{n-1} with $2T_{n-2} + 1$, since the recurrence says the two are equivalent. In the third step, we simplified a little—but not too much! After several similar rounds of plugging and chugging, a pattern is apparent. The following formula seems to hold:

$$\begin{aligned}
T_n &= 2^k T_{n-k} + 2^{k-1} + 2^{k-2} + \ldots + 2^2 + 2^1 + 2^0 \\
&= 2^k T_{n-k} + 2^k - 1
\end{aligned}$$

Once the pattern is clear, simplifying is safe and convenient. In particular, we've collapsed the geometric sum to a closed form on the second line.

Step 2: Verify the Pattern

The next step is to verify the general formula with one more round of plug-and-chug.

$$\begin{aligned} T_n &= 2^k T_{n-k} + 2^k - 1 \\ &= 2^k (2T_{n-(k+1)} + 1) + 2^k - 1 && \text{plug} \\ &= 2^{k+1} T_{n-(k+1)} + 2^{k+1} - 1 && \text{chug} \end{aligned}$$

The final expression on the right is the same as the expression on the first line, except that k is replaced by $k + 1$. Surprisingly, this effectively *proves* that the formula is correct for all k. Here is why: we know the formula holds for $k = 1$, because that's the original recurrence equation. And we've just shown that if the formula holds for some $k \geq 1$, then it also holds for $k + 1$. So the formula holds for all $k \geq 1$ by induction.

Step 3: Write T_n Using Early Terms with Known Values

The last step is to express T_n as a function of early terms whose values are known. Here, choosing $k = n - 1$ expresses T_n in terms of T_1, which is equal to 1. Simplifying gives a closed-form expression for T_n:

$$\begin{aligned} T_n &= 2^{n-1} T_1 + 2^{n-1} - 1 \\ &= 2^{n-1} \cdot 1 + 2^{n-1} - 1 \\ &= 2^n - 1. \end{aligned}$$

We're done! This is the same answer we got from guess-and-verify.

Let's compare guess-and-verify with plug-and-chug. In the guess-and-verify method, we computed several terms at the beginning of the sequence, T_1, T_2, T_3, etc., until a pattern appeared. We generalized to a formula for the nth term, T_n. In contrast, plug-and-chug works backward from the nth term. Specifically, we started with an expression for T_n involving the preceding term, T_{n-1}, and rewrote this using progressively earlier terms, T_{n-2}, T_{n-3}, etc. Eventually, we noticed a pattern, which allowed us to express T_n using the very first term, T_1, whose value we knew. Substituting this value gave a closed-form expression for T_n. So guess-and-verify and plug-and-chug tackle the problem from opposite directions.

10.2 Merge Sort

Algorithms textbooks traditionally claim that sorting is an important, fundamental problem in computer science. Then they smack you with sorting algorithms until life as a disk-stacking monk in Hanoi sounds delightful. Here, we'll cover just *one* well-known sorting algorithm, Merge Sort. The analysis introduces another kind of recurrence.

Here is how Merge Sort works. The input is a list of n numbers, and the output is those same numbers in nondecreasing order. There are two cases:

- If the input is a single number, then the algorithm does nothing, because the list is already sorted.
- Otherwise, the list contains two or more numbers. The first half and the second half of the list are each sorted recursively. Then the two halves are merged to form a sorted list with all n numbers.

Let's work through an example. Suppose we want to sort this list:

$$10, 7, 23, 5, 2, 8, 6, 9.$$

Since there is more than one number, the first half (10, 7, 23, 5) and the second half (2, 8, 6, 9) are sorted recursively. The results are 5, 7, 10, 23 and 2, 6, 8, 9. All that remains is to merge these two lists. This is done by repeatedly emitting the smaller of the two leading terms. When one list is empty, the whole other list is emitted. The example is worked out below. In this table, underlined numbers are about to be emitted.

First Half	Second Half	Output
5, 7, 10, 23	<u>2</u>, 6, 8, 9	
<u>5</u>, 7, 10, 23	6, 8, 9	2
7, 10, 23	<u>6</u>, 8, 9	2, 5
<u>7</u>, 10, 23	8, 9	2, 5, 6
10, 23	<u>8</u>, 9	2, 5, 6, 7
10, 23	<u>9</u>	2, 5, 6, 7, 8
<u>10</u>, <u>23</u>		2, 5, 6, 7, 8, 9
		2, 5, 6, 7, 8, 9, 10, 23

The leading terms are initially 5 and 2. So we output 2. Then the leading terms are 5 and 6, so we output 5. Eventually, the second list becomes empty. At that point, we output the whole first list, which consists of 10 and 23. The complete output consists of all the numbers in sorted order.

10.2.1 Finding a Recurrence

A traditional question about sorting algorithms is, "What is the maximum number of comparisons used in sorting n items?" This is taken as an estimate of the running time. In the case of Merge Sort, we can express this quantity with a recurrence. Let T_n be the maximum number of comparisons used while Merge Sorting a list of n numbers. For now, assume that n is a power of 2. This ensures that the input can be divided in half at every stage of the recursion.

- If there is only one number in the list, then no comparisons are required, so $T_1 = 0$.
- Otherwise, T_n includes comparisons used in sorting the first half (at most $T_{n/2}$), in sorting the second half (also at most $T_{n/2}$), and in merging the two halves. The number of comparisons in the merging step is at most $n - 1$. This is because at least one number is emitted after each comparison and one more number is emitted at the end when one list becomes empty. Since n items are emitted in all, there can be at most $n - 1$ comparisons.

Therefore, the maximum number of comparisons needed to Merge Sort n items is given by this recurrence:

$$\begin{aligned} T_1 &= 0 \\ T_n &= 2T_{n/2} + n - 1 \qquad (\text{for } n \geq 2 \text{ and a power of 2}). \end{aligned}$$

This fully describes the number of comparisons, but not in a very useful way; a closed-form expression would be much more helpful. To get that, we have to solve the recurrence.

10.2.2 Solving the Recurrence

Let's first try to solve the Merge Sort recurrence with the guess-and-verify technique. Here are the first few values:

$$\begin{aligned} T_1 &= 0 \\ T_2 &= 2T_1 + 2 - 1 = 1 \\ T_4 &= 2T_2 + 4 - 1 = 5 \\ T_8 &= 2T_4 + 8 - 1 = 17 \\ T_{16} &= 2T_8 + 16 - 1 = 49. \end{aligned}$$

We're in trouble! Guessing the solution to this recurrence is hard because there is no obvious pattern. So let's try the plug-and-chug method instead.

Step 1: Plug and Chug Until a Pattern Appears

First, we expand the recurrence equation by alternately plugging and chugging until a pattern appears.

$$\begin{aligned}
T_n &= 2T_{n/2} + n - 1 \\
&= 2(2T_{n/4} + n/2 - 1) + (n-1) && \text{plug} \\
&= 4T_{n/4} + (n-2) + (n-1) && \text{chug} \\
&= 4(2T_{n/8} + n/4 - 1) + (n-2) + (n-1) && \text{plug} \\
&= 8T_{n/8} + (n-4) + (n-2) + (n-1) && \text{chug} \\
&= 8(2T_{n/16} + n/8 - 1) + (n-4) + (n-2) + (n-1) && \text{plug} \\
&= 16T_{n/16} + (n-8) + (n-4) + (n-2) + (n-1) && \text{chug}
\end{aligned}$$

A pattern is emerging. In particular, this formula seems holds:

$$\begin{aligned}
T_n &= 2^k T_{n/2^k} + (n - 2^{k-1}) + (n - 2^{k-2}) + \ldots + (n - 2^0) \\
&= 2^k T_{n/2^k} + kn - 2^{k-1} - 2^{k-2} \ldots - 2^0 \\
&= 2^k T_{n/2^k} + kn - 2^k + 1.
\end{aligned}$$

On the second line, we grouped the n terms and powers of 2. On the third, we collapsed the geometric sum.

Step 2: Verify the Pattern

Next, we verify the pattern with one additional round of plug-and-chug. If we guessed the wrong pattern, then this is where we'll discover the mistake.

$$\begin{aligned}
T_n &= 2^k T_{n/2^k} + kn - 2^k + 1 \\
&= 2^k (2T_{n/2^{k+1}} + n/2^k - 1) + kn - 2^k + 1 && \text{plug} \\
&= 2^{k+1} T_{n/2^{k+1}} + (k+1)n - 2^{k+1} + 1 && \text{chug}
\end{aligned}$$

The formula is unchanged except that k is replaced by $k+1$. This amounts to the induction step in a proof that the formula holds for all $k \geq 1$.

Step 3: Write T_n Using Early Terms with Known Values

Finally, we express T_n using early terms whose values are known. Specifically, if we let $k = \log n$, then $T_{n/2^k} = T_1$, which we know is 0:

$$\begin{aligned} T_n &= 2^k T_{n/2^k} + kn - 2^k + 1 \\ &= 2^{\log n} T_{n/2^{\log n}} + n \log n - 2^{\log n} + 1 \\ &= nT_1 + n \log n - n + 1 \\ &= n \log n - n + 1. \end{aligned}$$

We're done! We have a closed-form expression for the maximum number of comparisons used in Merge Sorting a list of n numbers. In retrospect, it is easy to see why guess-and-verify failed: this formula is fairly complicated.

As a check, we can confirm that this formula gives the same values that we computed earlier:

n	T_n	$n \log n - n + 1$
1	0	$1 \log 1 - 1 + 1 = 0$
2	1	$2 \log 2 - 2 + 1 = 1$
4	5	$4 \log 4 - 4 + 1 = 5$
8	17	$8 \log 8 - 8 + 1 = 17$
16	49	$16 \log 16 - 16 + 1 = 49$

As a double-check, we could write out an explicit induction proof. This would be straightforward, because we already worked out the guts of the proof in step 2 of the plug-and-chug procedure.

10.3 Linear Recurrences

So far we've solved recurrences with two techniques: guess-and-verify and plug-and-chug. These methods require spotting a pattern in a sequence of numbers or expressions. In this section and the next, we'll give cookbook solutions for two large classes of recurrences. These methods require no flash of insight; you just follow the recipe and get the answer.

10.3.1 Climbing Stairs

How many different ways are there to climb n stairs, if you can either step up one stair or hop up two? For example, there are five different ways to climb four stairs:

1. step, step, step, step