Building Blocks for Theoretical Computer Science (Version 1.3)

Margaret M. Fleck

January 1, 2013

Contents

Preface

This book teaches two different sorts of things, woven together. It teaches you how to read and write mathematical proofs. It provides a survey of basic mathematical objects, notation, and techniques which will be useful in later computer science courses. These include propositional and predicate logic, sets, functions, relations, modular arithmetic, counting, graphs, and trees. And, finally, it gives a brief introduction to some key topics in theoretical computer science: algorithm analysis and complexity, automata theory, and computability.

Why learn formal mathematics?

Formal mathematics is relevant to computer science in several ways. First, it is used to create theoretical designs for algorithms and prove that they work correctly. This is especially important for methods that are used frequently and/or in applications where we don't want failures (aircraft design, Pentagon security, ecommerce). Only some people do these designs, but many people use them. The users need to be able to read and understand how the designs work.

Second, the skills you learn in doing formal mathematics correspond closely to those required to design and debug programs. Both require keeping track of what types your variables are. Both use inductive and/or recursive design. And both require careful proofreading skills. So what you learn in this class will also help you succeed in practical programming courses.

Third, when you come to design complex real software, you'll have to document what you've done and how it works. This is hard for many peo-

ple to do well, but it's critical for the folks using your software. Learning how to describe mathematical objects clearly will translate into better skills describing computational objects.

Everyone can do proofs

You probably think about proofs as something created by brilliant young theoreticians. Some of you **are** brilliant young theoreticians and you'll think this stuff is fun because it's what you naturally like to do. However, many of you are future software and hardware engineers. Some of you may never think of formal mathematics as "fun." That's ok. We understand. We're hoping to give you a sense of why it's pretty and useful, and enough fluency with it to communicate with the theoretical side of the field.

Many people think that proofs involve being incredibly clever. That is true for some steps in some proofs. That's where it helps to actually be a brilliant young theoretician. But many proofs are very routine. And many steps in clever proofs are routine. So there's quite a lot of proofs that all of us can do. And we can all read, understand, and appreciate the clever bits that we didn't think up ourselves.

In other words, don't be afraid. You can do proofs at the level required for this course, and future courses in theoretical computer science.

Is this book right for you?

This book is designed for students who have taken an introductory programming class of the sort intended for scientists or engineers. Algorithms will be presented in "pseudo-code," so it does not matter which programming language you have used. But you should have written programs that manipulate the contents of arrays e.g. sort an array of numbers. You should also have written programs that are recursive, i.e. in which a function (aka procedure aka method) calls itself.

We'll also assume that you have taken one semester of calculus. We will make very little direct reference to calculus: only a brief and relatively infor-

mal use of limits in Chapter 14. However, we will assume a level of fluency with precalculus (algebra, geometry, logarithms, etc) that is typically developed during while taking calculus. If you already have significant experience writing proofs, including inductive proofs, this book may be too easy for you. You may wish to read some of the books listed as supplementary readings in Appendix B.

For instructors

This text is designed for a broad range of computer science majors, ranging from budding young theoreticians to capable programmers with very little interest in theory. It assumes only limited mathematical maturity, so that it can be used very early in the major. Therefore, a central goal is to explain the process of proof construction clearly to students who can't just pick it up by osmosis.

This book is designed to be succinct, so that students will read and absorb its contents. Therefore, it includes only core concepts and a selection of illustrative examples, with the expectation that the instructor will provide supplementary materials as needed (see Appendix B for useful follow-on books) and that students can look up a wider range of facts, definitions, and pictures, e.g. on the internet.

Although the core topics in this book are old and established, terminology and notation have changed over the years and vary somewhat between authors. To avoid overloading students, I have chosen one clean, modern version of notation, definitions, and terminology to use consistently in the main text. Common variations are documented at the end of each chapter. If students understand the underlying concepts, they should have no difficulty adapting when they encounter different conventions in other books.

Many traditional textbooks do a careful and exhaustive treatment of each topic from first principles, including foundational constructions and theorems which prove that the conceptual system is well-formed. However, the most abstract parts of many topics are hard for beginners to absorb and the importance of the foundational development is lost on most students at this level. The early parts of this textbook remove most of this clutter, to focus more

clearly on the key concepts and constructs. At the end, we revisit certain topics to look at more abstract examples and selected foundational issues. See Appendix C for a quick guide to topic rearrangement.

Chapter 1

Math review

This book assumes that you understood precalculus when you took it. So you used to know how to do things like factoring polynomials, solving high school geometry problems, using trigonometric identities. However, you probably can't remember it all cold. Many useful facts can be looked up (e.g. on the internet) as you need them. This chapter reviews concepts and notation that will be used a lot in this book, as well as a few concepts that may be new but are fairly easy.

1.1 Some sets

You've all seen sets, though probably a bit informally. We'll get back to some of the formalism in a couple weeks. Meanwhile, here are the names for a few commonly used sets of numbers:

- $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$ is the integers.
- $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ is the non-negative integers, also known as the natural numbers.
- $\mathbb{Z}^+ = \{1, 2, 3, \ldots\}$ is the positive integers.
- $\mathbb{R}$ is the real numbers

- $\mathbb{Q}$ is the rational numbers
- $\mathbb{C}$ is the complex numbers

Notice that a number is positive if it is greater than zero, so zero is not positive. If you wish to exclude negative values, but include zero, you must use the term "non-negative." In this book, natural numbers are defined to include zero. Notice also that the real numbers contain the integers, so a "real" number is not required to have a decimal part. And, finally, infinity is not a number in standard mathematics.

Remember that the rational numbers are the set of fractions $\frac{p}{q}$ where q can't be zero and we consider two fractions to be the same number if they are the same when you reduce them to lowest terms.

The real numbers contain all the rationals, plus irrational numbers such as $\sqrt{2}$ and π. (Also e, but we don't use e much in this end of computer science.) We'll assume that $\sqrt{x}$ returns (only) the positive square root of x.

The complex numbers are numbers of the form $a + bi$, where a and b are real numbers and i is the square root of -1. Some of you from ECE will be aware that there are many strange and interesting things that can be done with the complex numbers. We won't be doing that here. We'll just use them for a few examples and expect you to do very basic things, mostly easy consequences of the fact that $i = \sqrt{-1}$.

If we want to say that the variable x is a real, we write $x \in \mathbb{R}$. Similarly $y \notin \mathbb{Z}$ means that y is not an integer. In calculus, you don't have to think much about the types of your variables, because they are largely reals. In this class, we handle variables of several different types, so it's important to state the types explicitly.

It is sometimes useful to select the real numbers in some limited range, called an *interval* of the real line. We use the following notation:

- closed interval: $[a, b]$ is the set of real numbers from a to b, including a and b
- open interval: (a, b) is the set of real numbers from a to b, not including a and b

- half-open intervals: $[a, b)$ and $(a, b]$ include only one of the two endpoints.

1.2 Pairs of reals

The set of all pairs of reals is written $\mathbb{R}^2$. So it contains pairs like $(-2.3, 4.7)$. Similarly, the set $\mathbb{R}^3$ contains triples of reals such as $8, 7.3, -9$. In a computer program, we implement a pair of reals using an object or struct with two fields.

Suppose someone presents you with a pair of numbers (x, y) or one of the corresponding data structures. Your first question should be: what is the pair intended to represent? It might be

- a point in 2D space
- a complex number
- a rational number (if the second coordinate isn't zero)
- an interval of the real line

The intended meaning affects what operations we can do on the pairs. For example, if (x, y) is an interval of the real line, it's a set. So we can write things like $z \in (x, y)$ meaning z is in the interval (x, y). That notation would be meaningless if (x, y) is a 2D point or a number.

If the pairs are numbers, we can add them, but the result depends on what they are representing. So $(x, y) + (a, b)$ is $(x + a, y + b)$ for 2D points and complex numbers. But it is $(xb + ya, yb)$ for rationals.

Suppose we want to multiply $(x, y) \times (a, b)$ and get another pair as output. There's no obvious way to do this for 2D points. For rationals, the formula would be (xa, yb), but for complex numbers it's $(xa - yb, ya + xb)$.

Stop back and work out that last one for the non-ECE students, using more familiar notation. $(x + yi)(a + bi)$ is $xa + yai + xbi + byi^2$. But $i^2 = -1$. So this reduces to $(xa - yb) + (ya + xb)i$.

Oddly enough, you can also multiply two intervals of the real line. This carves out a rectangular region of the 2D plane, with sides determined by the two intervals.

1.3 Exponentials and logs

Suppose that b is any real number. We all know how to take integer powers of b, i,e. b^n is b multiplied by itself n times. It's not so clear how to precisely define b^x, but we've all got faith that it works (e.g. our calculators produce values for it) and it's a smooth function that grows really fast as the input gets bigger and agrees with the integer definition on integer inputs.

Here are some special cases to know

- b^0 is one for any b.
- $b^{0.5}$ is $\sqrt{b}$
- b^{-1} is $\frac{1}{b}$

And some handy rules for manipulating exponents:

$$\begin{aligned} b^x b^y &= b^{x+y} \\ a^x b^x &= (ab)^x \\ (b^x)^y &= b^{xy} \\ b^{(x^y)} &\neq (b^x)^y \end{aligned}$$

Suppose that $b > 1$. Then we can invert the function $y = b^x$, to get the function $x = \log_b y$ ("logarithm of y to the base b"). Logarithms appear in computer science as the running times of particularly fast algorithms. They are also used to manipulate numbers that have very wide ranges, e.g. probabilities. Notice that the log function takes only positive numbers as inputs. In this class, $\log x$ with no explicit base always means $\log_2 x$ because analysis of computer algorithms makes such heavy use of base-2 numbers and powers of 2.

Useful facts about logarithms include:

$$\begin{aligned} b^{\log_b(x)} &= x \\ \log_b(xy) &= \log_b x + \log_b y \\ \log_b(x^y) &= y \log_b x \\ \log_b x &= \log_a x \log_b a \end{aligned}$$

In the change of base formula, it's easy to forget whether the last term should be $\log_b a$ or $\log_a b$. To figure this out on the fly, first decide which of $\log_b x$ and $\log_a x$ is larger. You then know whether the last term should be larger or smaller than one. One of $\log_b a$ and $\log_a b$ is larger than one and the other is smaller: figure out which is which.

More importantly, notice that the multiplier to change bases is a constant, i.e doesn't depend on x. So it just shifts the curve up and down without really changing its shape. That's an extremely important fact that you should remember, even if you can't reconstruct the precise formula. In many computer science analyses, we don't care about constant multipliers. So the fact that base changes simply multiply by a constant means that we frequently don't have to care what the base actually is. Thus, authors often write $\log x$ and don't specify the base.

1.4 Some handy functions

The factorial function may or may not be familiar to you, but is easy to explain. Suppose that k is any positive integer. Then k factorial, written $k!$, is the product of all the positive integers up to and including k. That is

$$k! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (k-1) \cdot k$$

For example, $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. $0!$ is defined to be 1.

The max function returns the largest of its inputs. For example, suppose we define $f(x) = \max(x^2, 7)$. Then $f(3) = 9$ but $f(2) = 7$. The min function is similar.

The floor and ceiling functions are heavily used in computer science, though not in many areas other of science and engineering. Both functions take a real number x as input and return an integer near x. The floor function returns the largest integer no bigger than x. In other words, it converts x to an integer, rounding down. This is written $\lfloor x \rfloor$. If the input to floor is already an integer, it is returned unchanged. Notice that floor rounds downward even for negative numbers. So:

$\lfloor 3.75 \rfloor = 3$

$\lfloor 3 \rfloor = 3$

$\lfloor -3.75 \rfloor = -4$

The ceiling function, written $\lceil x \rceil$, is similar, but rounds upwards. For example:

$\lceil 3.75 \rceil = 4$

$\lceil 3 \rceil = 3$

$\lceil -3.75 \rceil = -3$

Most programming languages have these two functions, plus a function that rounds to the nearest integer and one that "truncates" i.e. rounds towards zero. Round is often used in statistical programs. Truncate isn't used much in theoretical analyses.

1.5 Summations

If a_i is some formula that depends on i, then

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + a_3 + \ldots + a_n$$

For example

$$\sum_{i=1}^{n} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots + \frac{1}{2^n}$$

Products can be written with a similar notation, e.g.

$$\prod_{k=1}^{n} \frac{1}{k} = \frac{1}{1} \cdot \frac{1}{2} \cdot \frac{1}{3} \cdot \ldots \cdot \frac{1}{n}$$

Certain sums can be re-expressed "in closed form" i.e. without the summation notation. For example:

$$\sum_{i=1}^{n} \frac{1}{2^i} = 1 - \frac{1}{2^n}$$

In Calculus, you may have seen the infinite version of this sum, which converges to 1. In this class, we're always dealing with finite sums, not infinite ones.

If you modify the start value, so we start with the zeroth term, we get the following variation on this summation. Always be careful to check where your summation starts.

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

Many reference books have tables of useful formulas for summations that have simple closed forms. We'll see them again when we cover mathematical induction and see how to formally prove that they are correct.

Only a very small number of closed forms need to be memorized. One example is

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

Here's one way to convince yourself it's right. On graph paper, draw a box n units high and $n+1$ units wide. Its area is $n(n+1)$. Fill in the leftmost part of each row to represent one term of the summation: the left box in the first row, the left two boxes in the second row, and so on. This makes a little triangular pattern which fills exactly half the box.

1.6 Strings

You've probably seen strings in introductory programming, e.g. when processing input from the user or a disk file. In computer science, a **string** is a finite-length sequence of characters and its length is the number of characters in it. E.g. `pineapple` is a string of length 9.

The special symbol ϵ is used for the string containing no characters, which has length 0. Zero-length strings may seem odd, but they appear frequently in computer programs that manipulate text data. For example, a program that reads a line of input from the keyboard will typically receive a zero-length string if the user types ENTER twice.

We specify concatenation of strings, or concatenation of strings and individual characters, by writing them next to each other. For example, suppose that $\alpha =$ `blue` and $\beta =$ `cat`. Then $\alpha\beta$ would be the string `bluecat` and β`s` be the string `cats`.

A *bit string* is a string consisting of the characters 0 and 1. If A is a set of characters, then A^* is the set of all (finite-length) strings containing only characters from A. For example, if A contains all lower-case alphabetic characters, then A^* contains strings like `e`, `onion`, and `kkkkmmmmmmbb`. It also contains the empty string ϵ.

Sometimes we want to specify a pattern for a set of similar strings. We often use a shorthand notation called *regular expressions*. These look much like normal strings, but we can use two operations:

- `a` | `b` means either one of the characters `a` and `b`.
- `a`* means zero or more copies of the character `a`.

Parentheses are used to show grouping.

So, for example, `ab`* specifies all strings consisting of an `a` followed by zero or more `b`'s: `a`, `ab`, `abb`, and so forth. `c`(`b` | `a`)*`c` specifies all strings consisting of one `c`, followed by zero or more characters that are either `a`'s or `b`'s, followed one `c`. E.g. `cc`, `cac`, `cbbac`, and so forth.

1.7 Variation in notation

Mathematical notation is not entirely standardized and has changed over the years, so you will find that different subfields and even different individual authors use slightly different notation. These "variation in notation" sections tell you about synonyms and changes in notation that you are very likely to encounter elsewhere. Always check carefully which convention an author is using. When doing problems on homework or exams for a class, follow the house style used in that particular class.

In particular, authors differ as to whether zero is in the natural numbers. Moreover, $\sqrt{-1}$ is named j over in ECE and physics, because i is used for current. Outside of computer science, the default base for logarithms is usually e, because this makes a number of continuous mathematical formulas work nicely (just as base 2 makes many computer science formulas work nicely).

In standard definitions of the real numbers and in this book, infinity is not a number. People are sometimes confused because ∞ is sometimes used in notation where numbers also occur, e.g. in defining limits in calculus. There are non-standard number systems that include infinite numbers. And points at infinity are sometimes introduced in discussions of 2D and 3D geometry. We won't make use of such extensions.

Regular expressions are quite widely used in both theory and practical applications. As a result there are many variations on the notation. There are also other operations that make it easy to specify a wider range of patterns.

Chapter 2

Logic

This chapter covers propositional logic and predicate logic at a basic level. Some deeper issues will be covered later.

2.1 A bit about style

Writing mathematics requires two things. You need to get the logical flow of ideas correct. And you also need to express yourself in standard style, in a way that is easy for humans (not computers) to read. Mathematical style is best taught by example and is similar to what happens in English classes.

Mathematical writing uses a combination of equations and also parts that look superficially like English. Mathematical English is almost like normal English, but differs in some crucial ways. You are probably familiar with the fact that physicists use terms like "force" differently from everyone else. Or the fact that people from England think that "paraffin" is a liquid whereas that word refers to a solid substance in the US. We will try to highlight the places where mathematical English isn't like normal English.

You will also learn how to make the right choice between an equation and an equivalent piece of mathematical English. For example, $\wedge$ is a shorthand symbol for "and." The shorthand equations are used when we want to look at a complex structure all at once, e.g. discuss the logical structure of a proof. When writing the proof itself, it's usually better to use the longer English

equivalents, because the result is easier to read. There is no hard-and-fast line here, but we'll help make sure you don't go too far in either direction.

2.2 Propositions

Two systems of logic are commonly used in mathematics: propositional logic and predicate logic. We'll start by covering propositional logic.

A *proposition* is a statement which is true or false (but never both!). For example, "Urbana is in Illinois" or $2 \leq 15$. It can't be a question. It also can't contain variables, e.g. $x \leq 9$ isn't a proposition. Sentence fragments without verbs (e.g. "bright blue flowers") or arithmetic expressions (e.g. $5 + 17$), aren't propositions because they don't state a claim.

The lack of variables prevents propositional logic from being useful for very much, though it has some applications in circuit analysis, databases, and artificial intelligence. Predicate logic is an upgrade that adds variables. We will mostly be using predicate logic in this course. We just use propositional logic to get started.

2.3 Complex propositions

Statements can be joined together to make more complex statements. For example, "Urbana is in Illinois and Margaret was born in Wisconsin." To talk about complex sequences of statements without making everything too long, we represent each simple statement by a variable. E.g. if p is "Urbana is in Illinois" and q is "Margaret was born in Wisconsin", then the whole long statement would be "p and q". Or, using shorthand notation $p \wedge q$.

The statement $p \wedge q$ is true when both p and q are true. We can express this with a "truth table":

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

Similarly, $\neg p$ is the shorthand for "not p." In our example, $\neg p$ would be "Urbana is not in Illinois." $\neg p$ is true exactly when p is false.

$p \vee q$ is the shorthand for "p or q", which is true when either p or q is true. Notice that it is also true when both p and q are true, i.e. its true table is:

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

When mathematicians use "or", they always intend it to be understood with this "inclusive" reading.

Normal English sometimes follows the same convention, e.g. "you need to wear a green hat or a blue tie" allows you to do both. But normal English is different from mathematical English, in that "or" sometimes excludes the possibility that both statements are true. For example, "Clean up your room or you won't get desert" strongly suggests that if you do clean up your room, you will get desert. So normal English "or" sometimes matches mathematical "or" and sometimes another operation called **exclusive or** defined by

p	q	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

Exclusive or has some important applications in computer science, especially in encoding strings of letters for security reasons. However, we won't see it much in this class.

2.4 Implication

Two propositions p and q can also be joined into the **conditional** statement. "if p, then q." The proposition after the "if" (p in this case) is called the "hypothesis" and the proposition after "then" (q in this example) is called

the "conclusion." As in normal English, there are a number of alternative ways to phrase the statement "if p, then q", e.g. "p implies q" or "q follows from p".

The shorthand for this conditional is $p \to q$ and its truth table is

p	q	$p \to q$
T	T	T
T	F	F
F	T	T
F	F	T

For example, "If Obama is president, then Obama lives in the White House" is true (at least in 2010). But "If Obama is president, then $2 > 4$" is false. All the examples tend to be a bit artificial, because we don't have variables yet.

In normal English, we tend not to use conditionals in which the "if" part is false. E.g. "If Bush is president, then Urbana is in Illinois." In mathematical English, such statements occur more often. And, worse, they are always considered true, no matter whether the "then" part is true or false. For example, this statement is true: "If Bush is president, then $2 > 4$."

The easiest way to remember the right output values for this operation is to remember that the value is false in exactly one case: when p is true and q is false.

Normal English requires that conditional sentences have some sort of causal connection between the two propositions, i.e. one proposition is true because the other is true. E.g. "If Helen learns to write C++, she will get a good job." It would seem odd if we said "If Urbana is in Illinois, then Margaret was born in Wisconsin." because there's no reason why one follows from the other. In mathematical English, this statement is just fine: there doesn't have to be any causal connection.

In normal English if/then statements, there is frequently a flow of time involved. Unless we make a special effort to build a model of time, propositional logic is timeless. This makes the English motivating examples slightly awkward. It's not a big problem in mathematics, because mathematical proofs normally discuss a world that is static. It has a cast of characters (e.g. variables, sets, functions) with a fixed set of properties, and we are just

reasoning about what those properties are. Only very occasionally do we talk about taking an object and modifying it.

In computer programming, we often see things that look like conditional statements, e.g. "if $x > 0$, then increment y". But these are commands for the computer to do something, changing its little world. whereas the similar-looking mathematical statements are timeless. Formalizing what it means for a computer program to "do what it's supposed to" requires modelling how the world changes over time. You'll see this in later CS classes.

2.5 Converse, contrapositive, biconditional

The converse of $p \to q$ is $q \to p$. The two statements are not equivalent. To see this, compare the previous truth table with this one:

p	q	$q \to p$
T	T	T
T	F	T
F	T	F
F	F	T

The converse mostly occurs in two contexts. First, getting the direction of implication backwards is a common bug in writing proofs. That is, using the converse rather than the original statement. This is a bug because implications frequently hold in only one direction.

Second, the phrase "p implies q, and conversely" means that p and q are true under exactly the same conditions. The shorthand for this is the biconditional operator $p \leftrightarrow q$.

p	q	$q \leftrightarrow p$
T	T	T
T	F	F
F	T	F
F	F	T

Another common way to phrase the biconditional is "p if and only if q."

The contrapositive of $p \to q$ is formed by swapping the roles of p and q and negating both of them to get $\neg q \to \neg p$. Unlike the converse, the

contrapositive **is** equivalent to the original statement. Here's a truth table showing why:

p	q	$\neg q$	$\neg p$	$\neg q \to \neg p$
T	T	F	F	T
T	F	T	F	F
F	T	F	T	T
F	F	T	T	T

To figure out the last column of the table, recall that $\neg q \to \neg p$ will be false in only one case: when the hypothesis ($\neg q$) is true and the conclusion ($\neg p$) is false.

Let's consider what these variations look like in an English example:

- If it's below zero, my car won't start.
- converse: If my car won't start, it's below zero
- contrapositive: If my car will start, then it's not below zero.

2.6 Complex statements

Very complex statements can be made using combinations of connectives. E.g. "If it's below zero or my car does not have gas, then my car won't start and I can't go get groceries." This example has the form

$$(p \vee \neg q) \to (\neg r \wedge \neg s)$$

The shorthand notation is particularly useful for manipulating complicated statements, e.g. figuring out the negative of a statement.

When you try to read a complex set of propositions all glued together with connectives, there is sometimes a question about which parts to group together first. English is a bit vague about the rules. So, for example, in the previous example, you need to use common sense to figure out that "I can't go get groceries" is intended to be part of the conclusion of the if/then statement.

In mathematical shorthand, there are conventions about which parts to group together first. In particular, you apply the "not" operators first, then the "and" and "or". Then you take the results and do the implication operations. This is basically similar to the rules in (say) high-school algebra. Use parentheses if you intend the reader to group things differently, or if you aren't sure that your reader will group your statement as you intended.

You can build truth tables for complex statements, e.g.

p	q	r	$q \wedge r$	$(q \wedge r) \rightarrow p$
T	T	T	T	T
T	F	T	F	T
F	T	T	T	F
F	F	T	F	T
T	T	F	F	T
T	F	F	F	T
F	T	F	F	T
F	F	F	F	T

Truth tables are a nice way to show equivalence for compound propositions which use only 2-3 variables. However, if there are k variables, your table needs 2^k lines to cover all possible combinations of input truth values. This is cumbersome for larger numbers of variables.

2.7 Logical Equivalence

Two (simple or compound) propositions p and q are *logically equivalent* if they are true for exactly the same input values. The shorthand notation for this is $p \equiv q$. One way to establish logical equivalence is with a truth table.

For example, we saw that implication has the truth table:

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

A frequently useful fact is that $p \rightarrow q$ is logically equivalent to $\neg p \vee q$.

To show this, we build the truth table for $\neg p \vee q$ and notice that the output values exactly match those for $p \rightarrow q$.

p	q	$\neg p$	$\neg p \vee q$
T	T	F	T
T	F	F	F
F	T	T	T
F	F	T	T

Two very well-known equivalences are *De Morgan's Laws*. These state that $\neg(p \wedge q)$ is equivalent to $\neg p \vee \neg q$. and that $\neg(p \vee q)$ is equivalent to $\neg p \wedge \neg q$. Similar rules in other domains (e.g. set theory) are also called De Morgan's Laws. They are especially helpful, because they tell you how to simplify the negation of a complex statement involving "and" and "or".

We can show this easily with another truth table:

p	q	$\neg p$	$\neg q$	$p \vee q$	$\neg(p \vee q)$	$\neg p \wedge \neg q$
T	T	F	F	T	F	F
T	F	F	T	T	F	F
F	T	T	F	T	F	F
F	F	T	T	F	T	T

T and F are special constant propositions with no variables that are, respectively, always true and always false. So, since $p \wedge \neg p$ is always false, we have the following equivalence:

$$p \wedge \neg p \equiv F$$

Notice that, in mathematics, the equal operator $=$ can only be applied to objects such as numbers. When comparing logical expressions that return true/false values, you must use $\equiv$. If use $\equiv$ to create complex logical equations, use indenting and whitespace to make sure the result is easy to read.

2.8 Some useful logical equivalences

It is easy to find (e.g. on the internet) long tables of useful logical equivalences. Most of them are commonsense and/or similar to rules from algebra. For example, $\wedge$ and $\vee$ are commutative, e.g. $p \wedge q \equiv q \wedge p$.

The distributive laws, however, work slightly differently from those in algebra. In algebra we have one rule:

$$a(b + c) = ab + ac$$

where as in logic we have two rules:

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$
$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$$

So, in logic, you can distribute either operator over the other. Also, arithmetic has a clear rule that multiplication is done first, so the righthand side doesn't require parentheses. The order of operations is less clear for the logic, so more parentheses are required.

2.9 Negating propositions

An important use of logical equivalences is to help you correctly state the negation of a complex proposition, i.e. what it means for the complex proposition not to be true. This is important when you are trying to prove a claim false or convert a statement to its contrapositive. Also, looking at the negation of a definition or claim is often helpful for understanding precisely what the definition or claim means. Having clear mechanical rules for negation is important when working with concepts that are new to you, when you have only limited intuitions about what is correct.

For example, suppose we have a claim like "If M is regular, then M is paracompact or M is not Lindelöf." I'm sure you have no idea whether this is even true, because it comes from a math class you are almost certain not to have taken. However, you can figure out its negation.

First, let's convert the claim into shorthand so we can see its structure. Let r be "M is regular", p be "M is paracompact", and l be "M is Lindelöf." Then the claim would be $r \to (p \vee \neg l)$.

The negation of $r \to (p \vee \neg l)$ is $\neg(r \to (p \vee \neg l))$. However, to do anything useful with this negated expression, we normally need to manipulate it into an equivalent expression that has the negations on the individual propositions.

The key equivalences used in doing this manipulation are:

- $\neg(\neg p) \equiv p$
- $\neg(p \wedge q) \equiv \neg p \vee \neg q$
- $\neg(p \vee q) \equiv \neg p \wedge \neg q$
- $\neg(p \to q) \equiv p \wedge \neg q$.

The last of these follows from an equivalence we saw above: $p \to q \equiv \neg p \vee q$ plus one of DeMorgan's laws.

So we have

$$\neg(r \to (p \vee \neg l)) \equiv r \wedge \neg(p \vee \neg l) \equiv r \wedge \neg p \wedge \neg\neg l \equiv r \wedge \neg p \wedge l$$

So the negation of our original claim is "M is regular and M is not paracompact and M is Lindelöf." Knowing the mechanical rules helps you handle situations where your logical intuitions aren't fully up to the task of just seeing instinctively what the negation should look like. Mechanical rules are also very helpful when you are tired or stressed (e.g. during an exam).

Notice that we've also derived a new logical equivalence $\neg(r \to (p \vee \neg l)) \equiv r \wedge \neg p \wedge l$ by applying a sequence of known equivalences. This is how we establish new equivalences when truth tables get unwieldy.

2.10 Predicates and Variables

Propositions are a helpful beginning but too rigid to represent most of the interesting bits of mathematics. To do this, we need predicate logic, which

allows variables and predicates that take variables as input. We'll get started with predicate logic now, but delay covering some of the details until they become relevant to the proofs we're looking at.

A predicate is a statement that becomes true or false if you substitute in values for its variables. For example, "$x^2 \geq 10$" or "y is winter hardy." Suppose we call these $P(x)$ and $Q(y)$. Then $Q(y)$ is true if y is "mint" but not if y is "tomato".[1]

If we substitute concrete values for all the variables in a predicate, we're back to having a proposition. That wasn't much use, was it?

The main use of predicates is to make general statements about what happens when you substitute a variety of values for the variables. For example:

P(x) is true for every x

For example, "For every integer x, $x^2 \geq 10$" (false).

Consider "For all x, $2x \geq x$." Is this true or false? This depends on what values x can have. Is x any integer? In that case the claim is false. But if x is supposed to be a natural number, then the claim is true.

In order to decide whether a statement involving quantifiers is true, you need to know what types of values are allowed for each variable. Good style requires that you state the type of the variable explicitly when you introduce it, e.g. "For all natural numbers x, $2x \geq x$." Exceptions involve cases where the type is very, very clear from the context, e.g. when a whole long discussion is all about (say) the integers. If you aren't sure, a redundant type statement is a minor problem whereas a missing type statement is sometimes a big problem.

2.11 Other quantifiers

The general idea of a quantifier is that it expresses how many of the values in the domain make the claim true. Normal English has a wide range of

[1] A winter hardy plant is a plant that can survive the winter in a cold climate, e.g. here in central Illinois.

quantifiers which tell you roughly how many of the values work, e.g. "some", "a couple", "a few", "many", "most." For example, "most students in this class have taken a programming class."

By contrast, mathematics uses only three quantifiers, one of which is used rarely. We've seen the *universal quantifier* "for all." The other common one is the *existential quantifier* "there exists," as in

There is an integer x such that $x^2 = 0$.

In normal English, when we say that there is an object with some properties, this tends to imply that there's only one or perhaps only a couple. If there were many objects with this property, we normally expect the speaker to say so. So it would seem odd to say

There is an integer x such that $x^2 > 0$.

Or

There exists an integer x such that $5 < x < 100$.

Mathematicians, however, are happy to say things like that. When they say "there exists an x," with certain properties, they mean that there exists at least one x with those properties. They are making no claims about how many such x's there are.

However, it is sometimes important to point out when one and only one x has some set of properties. The mathematical jargon for this uses the *unique existence* quantifier, as in:

There is a unique integer x such that $x^2 = 0$.

Mathematicians use the adjective "unique" to mean that there's only one such object (similar to the normal usage but not quite the same).

2.12 Notation

The universal quantifier has the shorthand notation $\forall$. For example,

$$\forall x \in \mathbb{R}, x^2 + 3 \geq 0$$

In this sentence, $\forall$ is the quantifier. $x \in \mathbb{R}$ declares the variable and the set ($\mathbb{R}$) from which its values can be taken, called its *domain* or its *replacement set*. As computer scientists, we can also think of this as declaring the type of x, just as in a computer program. Finally, $x^2 + 3 \geq 0$ is the predicate.

The existential quantifier is written $\exists$, e.g. $\exists y \in \mathbb{R}, y = \sqrt{2}$. Notice that we don't write "such that" when the quantifier is in shorthand. The unique existence quantifier is written $\exists!$ as in $\exists! x \in \mathbb{R}, x^2 = 0$. When existential quantifiers are written in English, rather than shorthand, we need to add the phrase "such that" to make the English sound right, e.g.

> There exists a real number y such that $y = \sqrt{2}$.

There's no deep reason for adding "such that." It's just a quirk about how mathematical English is written, which you should copy so that your written mathematics looks professional. "Such that" is sometimes abbreviated "s.t."

2.13 Useful notation

If you want to state a claim about two numbers, you can use two quantifiers as in:

$$\forall x \in \mathbb{R}, \forall y \in \mathbb{R}, x + y \geq x$$

This is usually abbreviated to

$$\forall x, y \in \mathbb{R}, x + y \geq x$$

This means "for all real numbers x and y, $x + y \geq x$" (which isn't true).

In such a claim, the two variables x and y might contain different values, but it's important to realize that they might also be equal. For example, the following sentence is true:

$$\exists x, y \in \mathbb{Z}, x - y = 0$$

We saw above that the statement "if p, then q" has the contrapositive "if $\neg q$, then $\neg p$." This transformation can be extended to statements with a quantifier (typically universal). For example, the statement

$$\forall x, \text{ if } p(x), \text{ then } q(x)$$

would have a contrapositive

$$\forall x, \text{ if } \neg q(x), \text{ then } \neg p(x)$$

Notice that the quantifier stays the same: we only transform the if/then statement inside it.

2.14 Notation for 2D points

When writing mathematics that involves 2D points and quantifiers, you have several notational options. You can write something like $\forall x, y \in \mathbb{Z}$ ("for any integers x and y") and then later refer to the pair (x, y). Or you can treat the pair (x, y) as a single variable, whose replacement set is all 2D points. For example, the following says that the real plane ($\mathbb{R}^2$) contains a point on the unit circle:

$$\exists (x, y) \in \mathbb{R}^2, x^2 + y^2 = 1$$

Another approach is to write something like

$$\exists p \in \mathbb{R}^2, p \text{ is on the unit circle}$$

When you later need to make precise what it means to be "on the unit circle," you will have to break up p into its two coordinates. At that point, you say

that that since p is a point on the plane, it must have the form (x, y), where x and y are real numbers. This defines the component variables you need to expand the definition of "on the unit circle" into the equation $x^2 + y^2 = 1$.

2.15 Negating statements with quantifiers

Suppose we have a universal claim like $\forall x \in \mathbb{R}, x^2 \geq 0$. This claim will be false if there is at least one real number x such that $x^2 < 0$. In general, a statement of the form "for all x in A, $P(x)$" is false exactly when there is some value x in A such that $P(x)$ is false. In other words, when "there exists x in A such that $P(x)$ is not true". In shorthand notation:

$$\neg(\forall x, P(x)) \equiv \exists x, \neg P(x)$$

Similarly,

$$\neg(\exists x, P(x)) \equiv \forall x, \neg P(x)$$

So this is a bit like the de Morgan's laws: when you move the negation across the operator, you change it to the other similar operator.

We saw above how to move negation operators from the outside to the inside of expressions involving $\wedge$, $\vee$, and the other propositional operators. Together with these two new rules to handle quantifiers, we now have a mechanical procedure for working out the negation of any random statement in predicate logic.

So if we have something like

$$\forall x, P(x) \rightarrow (Q(x) \wedge R(x))$$

Its negation is

$$\begin{aligned}\neg(\forall x, P(x) \to (Q(x) \wedge R(x))) &\equiv \exists x, \neg(P(x) \to (Q(x) \wedge R(x))) \\ &\equiv \exists x, P(x) \wedge \neg(Q(x) \wedge R(x))) \\ &\equiv \exists x, P(x) \wedge (\neg Q(x) \vee \neg R(x))\end{aligned}$$

2.16 Binding and scope

A quantifier is said to "bind" the variable it defines. For example, in the statement

$$\forall x \in \mathbb{R}, x^2 + 3 \geq 0$$

the quantifier $\forall$ binds the variable x.

The "bound" variable in a quantification is only defined for a limited time, called the "scope" of the binding. This is usually the end of the quantified statement or the end of the line, but you sometimes have to use common sense about what the author intended. Parentheses are often used to indicate that the author intends a shorter scope.

If a variable hasn't been bound by a quantifier, or otherwise given a value or a set of replacement values, it is called "free." Statements containing free variables don't have a defined truth value, so they cannot be (for example) a step in a proof.

Variables bound by quantifiers are like the dummy variables used in summations and integrals. For example, the i in $\sum_{i=0}^{n} \frac{1}{i}$ is only defined while you are still inside the summation. Variables in computer programs also have a "scope" over which their declaration is valid, e.g. the entire program, a single code file, or local to an individual function/procedure/method. If you try to use a variable outside the scope of its definition, you'll get a compiler error.

When writing mathematics, variables have to be defined, just like variables in computer programs. It's not polite to start using a variable without telling the reader where this variable comes from: is it bound by a quantifier? is it being set to a specific value (e.g. let $x = 3.1415$)? The reader also needs to know what type of value this variable might contain.

2.17 Variations in Notation

Although the core concepts of predicate logic are very standard, a few details vary from author to author. Please stick to the conventions used above, because it's less confusing if we all use the same notation. However, don't be surprised if another class or book does things differently. For example:

- There are several conventions about inserting commas after the quantifier and/or parentheses around the following predicate. We won't be picky about this.
- Some subfields (but not this class) have a convention that "and" is applied before "or," so that parentheses around "and" operations can be omitted. We'll keep the parentheses in such cases.
- Some authors use certain variations of "or" (e.g. "either ... or") with an exclusive meaning, when writing mathematical English. In this class, always read "or" as inclusive unless there is a clear explicit indiciation that it's meant to be exclusive. For example, "Take some bread or some cereal" should be read as inclusive in a mathematical context. If we wanted to indicate an exclusive reading, we would write something like "Take bread or some cereal, but not both."
- In circuits and certain programming languages, "true" and "false" are represented by 1 and 0. There are a number of different shorthand notations for logical connectives such as "and". Finally, programmers often use the term "boolean" to refer to true/false values and to expressions that return true/false values.

Chapter 3

Proofs

Many mathematical proofs use a small range of standard outlines: direct proof, examples/counter-examples, and proof by contrapositive. These notes explain these basic proof methods, as well as how to use definitions of new concepts in proofs. More advanced methods (e.g. proof by induction, proof by contradiction) will be covered later.

3.1 Proving a universal statement

Now, let's consider how to prove a claim like

> For every rational number q, $2q$ is rational.

First, we need to define what we mean by "rational".

> A real number r is *rational* if there are integers m and n, $n \neq 0$, such that $r = \frac{m}{n}$.

In this definition, notice that the fraction $\frac{m}{n}$ does not need to satisfy conditions like being proper or in lowest terms So, for example, zero is rational because it can be written as $\frac{0}{1}$. However, it's critical that the two numbers

in the fraction be integers, since even irrational numbers can be written as fractions with non-integers on the top and/or bottom. E.g. $\pi = \frac{\pi}{1}$.

The simplest technique for proving a claim of the form $\forall x \in A, P(x)$ is to pick some representative value for x.[1]. Think about sticking your hand into the set A with your eyes closed and pulling out some random element. You use the fact that x is an element of A to show that $P(x)$ is true. Here's what it looks like for our example:

> Proof: Let q be any rational number. From the definition of "rational," we know that $q = \frac{m}{n}$ where m and n are integers and n is not zero. So $2q = 2\frac{m}{n} = \frac{2m}{n}$. Since m is an integer, so is $2m$. So $2q$ is also the ratio of two integers and, therefore, $2q$ is rational.

At the start of the proof, notice that we expanded the word "rational" into what its definition said. At the end of the proof, we went the other way: noticed that something had the form required by the definition and then asserted that it must be a rational.

WARNING!! Abuse of notation. Notice that the above definition of "rational" used the word "if". If you take this literally, it would mean that the definition could be applied only in one direction. This isn't what's meant. Definitions are always intended to work in both directions. Technically, I should have written "if and only if" (frequently shortened to "iff"). This little misuse of "if" in definitions is very, very common.

Notice also that we spelled out the definition of "rational" but we just freely used facts from high school algebra as if they were obvious. In general, when writing proofs, you and your reader come to some agreement about what parts of math will be considered familiar and obvious, and which require explicit discussion. In practical terms, this means that, when writing solutions to homework problems, you should try to mimic the level of detail in examples presented in lecture and in model solutions to previous homeworks.

[1]The formal name for this is "universal instantiation."

3.2 Another example of direct proof involving odd and even

Here's another claim that can be proved by direct proof.

Claim 1 *For any integer k, if k is odd then k^2 is odd.*

This has a slightly different form from the previous claim: $\forall x \in \mathbb{Z}$, if $P(x)$, then $Q(x)$

Before doing the actual proof, we first need to be precise about what we mean by "odd". And, while we are on the topic, what we mean by "even."

Definition 1 *An integer n is even if there is an integer m such that $n = 2m$.*

Definition 2 *An integer n is odd if there is an integer m such that $n = 2m + 1$.*

Such definitions are sometimes written using the jargon "has the form," as in "An integer n is even if it has the form $2m$, where m is an integer."

We'll assume that it's obvious (from our high school algebra) that every integer is even or odd, and that no integer is both even and odd. You probably also feel confident that you know which numbers are odd or even. An exception might be zero: notice that the above definition makes it definitely even. This is the standard convention in math and computer science.

Using these definitions, we can prove our claim as follows:

> Proof of Claim 1: Let k be any integer and suppose that k is odd. We need to show that k^2 is odd.
>
> Since k is odd, there is an integer j such that $k = 2j + 1$. Then we have
>
> $$k^2 = (2j + 1)^2 = 4j^2 + 4j + 1 = 2(2j^2 + 2j) + 1$$
>
> Since j is an integer, $2j^2 + 2j$ is also an integer. Let's call it p. Then $k^2 = 2p + 1$. So, by the definition of odd, k^2 is odd.

As in the previous proof, we used our key definition twice in the proof: once at the start to expand a technical term ("odd") into its meaning, then again at the end to summarize our findings into the appropriate technical terms.

At the start of the proof, notice that we chose a random (or "arbitrary" in math jargon) integer k, like last time. However, we also "supposed" that the hypothesis of the if/then statement was true. It's helpful to collect up all your given information right at the start of the proof, so you know what you have to work with.

The comment about what we need to show is not necessary to the proof. It's sometimes included because it's helpful to the reader. You may also want to include it because it's helpful *to you* to remind you of where you need to get to at the end of the proof.

Similarly, introducing the variable p isn't really necessary with a claim this simple. However, using new variables to create an exact match to a definition may help you keep yourself organized.

3.3 Direct proof outline

In both of these proofs, we started from the known information (anything in the variable declarations and the hypothesis of the if/then statement) and moved gradually towards the information that needed to be proved (the conclusion of the if/then statement). This is the standard "logical" order for a direct proof. It's the easiest order for a reader to understand.

When working out your proof, you may sometimes need to reason backwards from your desired conclusion on your scratch paper. However, when you write out the final version, reorder everything so it's in logical order.

You will sometimes see proofs that are written partly in backwards order. This is harder to do well and requires a lot more words of explanation to help the reader follow what the proof is trying to do. When you are first starting out, especially if you don't like writing a lot of comments, it's better to stick to a straightforward logical order.

3.4 Proving existential statements

Here's an existential claim:

Claim 2 *There is an integer k such that $k^2 = 0$.*

An existential claim such as the following asserts the existence of an object with some set of properties. So it's enough to exhibit some specific concrete object, of our choosing, with the required properties. So our proof can be very simple:

Proof: Zero is such an integer. So the statement is true.

We could spell out a bit more detail, but it's really not necessary. Proofs of existential claims are often very short, though there are exceptions.

Notice one difference from our previous proofs. When we pick a value to instantiate a universally quantified variable, we have no control over exactly what the value is. We have to base our reasoning just on what set it belongs to. But when we are proving an existential claim, we get to pick our own favorite choice of concrete value, in this case zero.

Don't prove an existential claim using a general argument about why there must exist numbers with these properties.[2] This is not only overkill, but very hard to do correctly and harder on the reader. Use a specific, concrete example.

3.5 Disproving a universal statement

Here's a universal claim that is false:

Claim 3 *Every rational number q has a multiplicative inverse.*

[2]In higher mathematics, you occuasionally must make an abstract argument about existence because it's not feasible to produce a concrete example. We will see one example towards the end of this course. But this is very rare.

Definition 3 *If q and r are real numbers, r is a multiplicative inverse for q if $qr = 1$.*

In general, a statement of the form "for all x in A, $P(x)$" is false exactly when there is some value y in A for which $P(y)$ is false.[3] So, to disprove a universal claim, we need to prove an existential statement. So it's enough to exhibit one concrete value (a "counter-example") for which the claim fails. In this case, our disproof is very simple:

> Disproof of Claim 3: This claim isn't true, because we know from high school algebra that zero has no inverse.

Don't try to construct a general argument when a single specific counterexample would be sufficient.

3.6 Disproving an existential statement

There's a general pattern here: the negation of $\forall x, P(x)$ is $\exists x, \neg P(x)$. So the negation of a universal claim is an existential claim. Similarly the negation of $\exists x, P(x)$ is $\forall x, \neg P(x)$. So the negation of an existential claim is a universal one.

Suppose we want to disprove an existential claim like:

Claim 4 *There is an integer k such that $k^2 + 2k + 1 < 0$.*

We need to make a general argument that, no matter what value of k we pick, the equation won't hold. So we need to prove the claim

Claim 5 *For every integer k, it's not the case that $k^2 + 2k + 1 < 0$.*

Or, said another way,

[3]Notice that "for which" is occasionally used as a variant of "such that." In this case, it makes the English sound very slightly better.

Claim 6 *For every integer k, $k^2 + 2k + 1 \geq 0$.*

The proof of this is fairly easy:

> Proof: Let k be an integer. Then $(k+1)^2 \geq 0$ because the square of any real number is non-negative. But $(k+1)^2 = k^2+2k+1$. So, by combining these two equations, we find that $k^2 + 2k + 1 \geq 0$.

3.7 Recap of proof methods

So, our general pattern for selecting the proof type is:

	prove	disprove
universal	general argument	specific counter-example
existential	specific example	general argument

Both types of proof start off by picking an element x from the domain of the quantification. However, for the general arguments, x is a random element whose identity you don't know. For the proofs requiring specific examples, you can pick x to be your favorite specific concrete value.

3.8 Direct proof: example with two variables

Let's do another example of direct proof. First, let's define

Definition 4 *An integer n is a perfect square if $n = k^2$ for some integer k.*

And now consider the claim:

Claim 7 *For any integers m and n, if m and n are perfect squares, then so is mn.*

> Proof: Let m and n be integers and suppose that m and n are perfect squares.

> By the definition of "perfect square", we know that $m = k^2$ and $n = j^2$, for some integers k and j. So then mn is k^2j^2, which is equal to $(kj)^2$. Since k and j are integers, so is kj. Since mn is the square of the integer kj, mn is a perfect square, which is what we needed to show.

Notice that we used a different variable name in the two uses of the definition of perfect square: k the first time and j the second time. It's important to use a fresh variable name each time you expand a definition like this. Otherwise, you could end up forcing two variables (m and n in this case) to be equal when that isn't (or might not be) true.

Notice that the phrase "which is what we needed to show" helps tell the reader that we're done with the proof. It's polite to indicate the end in one way or another. In typed notes, it may be clear from the indentation. Sometimes, especially in handwritten proofs, we put a box or triangle of dots or Q.E.D. at the end. Q.E.D. is short for Latin "Quod erat demonstrandum," which is just a translation of "what we needed to show."

3.9 Another example with two variables

Here's another example of a claim involving two variables:

Claim 8 *For all integers j and k, if j and k are odd, then jk is odd.*

A direct proof would look like:

> Proof: Let j and k be integers and suppose they are both odd. Because j is odd, there is an integer p such that $j = 2p + 1$. Similarly, there is an integer q such that $k = 2q + 1$.
>
> So then $jk = (2p+1)(2q+1) = 4pq+2p+2q+1 = 2(2pq+p+q)+1$. Since p and q are both integers, so is $2pq + p + q$. Let's call it m. Then $jk = 2m + 1$ and therefore jk is odd, which is what we needed to show.

3.10 Proof by cases

When constructing a proof, sometimes you'll find that part of your given information has the form "p or q." Perhaps your claim was stated using "or," or perhaps you had an equation like $|x| > 6$ which translates into an or ($x > 6$ or $x < -6$) when you try to manipulate it. When the given information allows two or more separate possibilities, it is often best to use a technique called "proof by cases."

In proof by cases, you do part of the proof two or more times, once for each of the possibilities in the "or." For example, suppose that our claim is:

Claim 9 *For all integers j and k, if j is even or k is even, then jk is even.*

We can prove this as follows:

> Proof: Let j and k be integers and suppose that j is even or k is even. There are two cases:
>
> Case 1: j is even. Then $j = 2m$, where m is an integer. So the $jk = (2m)k = 2(mk)$. Since m and k are integers, so is mk. So jk must be even.
>
> Case 2: k is even. Then $k = 2n$, where n is an integer. So the $jk = j(2n) = 2(nj)$. Since n and j are integers, so is nj. So jk must be even.
>
> So jk is even in both cases, which is what we needed to show.

It is ok to have more than two cases. It's also ok if the cases overlap, e.g. one case might assume that $x \leq 0$ and another case might assume that $x \geq 0$. However, you must be sure that all your cases, taken together, cover all the possibilities.

In this example, each case involved expanding the definition of "even." We expanded the definition twice but, unlike our earlier examples, only one expansion is active at a given time. So we could have re-used the variable m when we expanded "even" in Case 2. I chose to use a fresh variable name (n) because this is a safer strategy if you aren't absolutely sure when re-use is ok.

3.11 Rephrasing claims

Sometimes you'll be asked to prove a claim that's not in a good form for a direct proof. For example:

Claim 10 *There is no integer k such that k is odd and k^2 is even.*

It's not clear how to start a proof for a claim like this. What is our given information and what do we need to show?

In such cases, it is often useful to rephrase your claim using logical equivalences. For example, the above claim is equivalent to

Claim 11 *For every integer k, it is not the case that k is odd and k^2 is even.*

By DeMorgan's laws, this is equivalent to

Claim 12 *For every integer k, k is not odd or k^2 is not even.*

Since we're assuming we all know that even and odd are opposites, this is the same as

Claim 13 *For every integer k, k is not odd or k^2 is odd.*

And we can restate this as an implication using the fact that $\neg p \vee q$ is equivalent to $p \rightarrow q$:

Claim 14 *For every integer k, if k is odd then k^2 is odd.*

Our claim is now in a convenient form: a universal if/then statement whose hypothesis contains positive (not negated) facts. And, in fact, we proved this claim earlier in these notes.

3.12 Proof by contrapositive

A particularly common sort of rephrasing is to replace a claim by its contrapositive. If the original claim was $\forall x, P(x) \to Q(x)$ then its contrapositive is $\forall x, \neg Q(x) \to \neg P(x)$. Remember that any if/then statement is logically equivalent to its contrapositive.

Remember that constructing the hypothesis requires swapping the hypothesis with the conclusion AND negating both of them. If you do only half of this transformation, you get a statement that isn't equivalent to the original. For example, the converse $\forall x, Q(x) \to P(x)$ is not equivalent to the original claim.

For example, suppose that we want to prove

Claim 15 *For any integers a and b, if $a + b \geq 15$, then $a \geq 8$ or $b \geq 8$.*

This is hard to prove in its original form, because we're trying to use information about a derived quantity to prove something about more basic quantities. If we rephrase as the contrapositive, we get

Claim 16 *For any integers a and b, if it's not the case that $a \geq 8$ or $b \geq 8$, then it's not the case that $a + b \geq 15$.*

And this is equivalent to:

Claim 17 *For any integers a and b, if $a < 8$ and $b < 8$, then $a + b < 15$.*

Notice that when we negated the conclusion of the original statement, we needed to change the "or" into an "and" (DeMorgan's Law).

When you do this kind of rephrasing, your proof should start by explaining to the reader how you rephrased the claim. It's technically enough to say that you're proving the contrapositive. But, for a beginning proof writer, it's better to actually write out the contrapositive of the claim. This gives you a chance to make sure you have constructed the contrapositive correctly. And, while you are writing the rest of the proof, it helps remind you of exactly what is given and what you need to show.

So a proof our our original claim might look like:

Proof: We'll prove the contrapositive of this statement. That is, for any integers a and b, if $a < 8$ and $b < 8$, then $a + b < 15$.

So, suppose that a and b are integers such that $a < 8$ and $b < 8$. Since they are integers (not e.g. real numbers), this implies that $a \leq 7$ and $b \leq 7$. Adding these two equations together, we find that $a + b \leq 14$. But this implies that $a + b < 15$. □

There is no hard-and-fast rule about when to switch to the contrapositive of a claim. If you are stuck trying to write a direct proof, write out the contrapositive of the claim and see whether that version seems easier to prove.

3.13 Another example of proof by contrapositive

Here's another example where it works well to convert to the contrapositive:

Claim 18 *For any integer k, if $3k + 1$ is even, then k is odd.*

If we rephrase as the contrapositive, we get:

Claim 19 *For any integer k, if k is even, $3k + 1$ is odd.*

So our complete proof would look like:

Proof: We will prove the contrapositive of this claim, i.e. that for any integer k, if k is even, $3k + 1$ is odd.

So, suppose that k is an integer and k is even. Then, $k = 2m$ for some integer m. Then $3k + 1 = 3(2m) + 1 = 2(3m) + 1$. Since m is an integer, so is $3m$. So $3k + 1$ must be odd, which is what we needed to show.

Chapter 4

Number Theory

We've now covered most of the basic techniques for writing proofs. So we're going to start applying them to specific topics in mathematics, starting with number theory.

Number theory is a branch of mathematics concerned with the behavior of integers. It has very important applications in cryptography and in the design of randomized algorithms. Randomization has become an increasingly important technique for creating very fast algorithms for storing and retrieving objects (e.g. hash tables), testing whether two objects are the same (e.g. MP3's), and the like. Much of the underlying theory depends on facts about which integers evenly divide one another and which integers are prime.

4.1 Factors and multiples

You've undoubtedly seen some of the basic ideas (e.g. divisibility) somewhat informally in earlier math classes. However, you may not be fully clear on what happens with special cases, e.g. zero, negative numbers. We also need clear formal definitions in order to write formal proofs. So, let's start with

> Definition: Suppose that a and b are integers. Then a divides b if $b = an$ for some integer n. a is called a factor or divisor of b. b is called a multiple of a.

The shorthand for a divides b is $a \mid b$. Be careful about the order. The divisor is on the left and the multiple is on the right.

Some examples:

- $7 \mid 77$
- $77 \nmid 7$
- $7 \mid 7$ because $7 = 7 \cdot 1$
- $7 \mid 0$ because $0 = 7 \cdot 0$, zero is divisible by any integer.
- $0 \nmid 7$ because $0 \cdot n$ will always give you zero, never 7. Zero is a factor of only one number: zero.
- $(-3) \mid 12$ because $12 = 3 \cdot -4$
- $3 \mid (-12)$ because $-12 = 3 \cdot -4$

An integer p is even exactly when $2 \mid p$. The fact that zero is even is just a special case of the fact that zero is divisible by any integer.

4.2 Direct proof with divisibility

We can prove basic facts about divisibility in much the same way we proved basic facts about even and odd.

Claim 20 *For any integers a,b,and c, if $a \mid b$ and $a \mid c$ then $a \mid (b + c)$.*

> Proof: Let a,b,and c and suppose that $a \mid b$ and $a \mid c$.
>
> Since $a \mid b$, there is an integer k such that $b = ak$ (definition of divides). Similarly, since $a \mid c$, there is an integer j such that $c = aj$. Adding these two equations, we find that $(b + c) = ak + aj = a(k + j)$. Since k and j are integers, so is $k + j$. Therefore, by the definition of divides, $a \mid (b + c)$. $\square$

When we expanded the definition of divides for the second time, we used a fresh variable name. If we had re-used k, then we would have wrongly forced b and c to be equal.

The following two claims can be proved in a similar way:

Claim 21 *For any integers a,b,and c, if $a \mid b$ and $b \mid c$ then $a \mid c$. (Transitivity of divides.)*

Claim 22 *For any integers a,b, and c, if $a \mid b$, then $a \mid bc$.*

You've probably seen "transitivity" before in the context of inequalities. E.g. if $a < b$ and $b < c$, then $a < c$. We'll get back to the general notion of transitivity later in the term.

4.3 Stay in the Set

Students are sometimes tempted to rephrase the definition of $a \mid b$ as "$\frac{b}{a}$ is an integer." This is not a good idea because it introduces a non-integer rational number into a problem that only needs to involve integers. Stepping outside the set of interest in this way is occasionally useful, but more often it leads to inelegant and/or buggy solutions. This happens for three reasons:

- The purely integer proofs are typically simpler.
- When constructing math from the ground up, the integers are typically constructed first and the rationals built from them. So using rationals to prove facts about integers can lead to circular proofs.
- On computers, integer operations yield exact answers but floating point operations are only approximate. So implementing an integer calculation using real numbers often introduces errors.

4.4 Prime numbers

We're all familiar with prime numbers from high school. Firming up the details:

> Definition: an integer $q \geq 2$ is prime if the only positive factors of q are q and 1. An integer $q \geq 2$ is composite if it is not prime.

For example, among the integers no bigger than 20, the primes are 2, 3, 5, 7, 11, 13, 17, and 19. Numbers smaller than 2 are neither prime nor composite.

A key fact about prime numbers is

> Fundanmental Theorem of Arithmetic: Every integer ≥ 2 can be written as the product of one or more prime factors. Except for the order in which you write the factors, this prime factorization is unique.

The word "unique" here means that there is only one way to factor each integer.

For example, $260 = 2 \cdot 2 \cdot 5 \cdot 13$ and $180 = 2 \cdot 2 \cdot 3 \cdot 3 \cdot 5$.

We won't prove this theorem right now, because it requires a proof technique called "induction," which we haven't seen yet.

There are quite fast algorithms for testing whether a large integer is prime. However, even once you know a number is composite, algorithms for factoring the number are all fairly slow. The difficulty of factoring large composite numbers is the basis for a number of well-known cryptographic algorithms (e.g. the RSA algorithm).

4.5 GCD and LCM

If c divides both a and b, then c is called a **common divisor** of a and b. The largest such number is the **greatest common divisor** of a and b. Shorthand for this is $\gcd(a, b)$.

You can find the GCD of two numbers by inspecting their prime factorizations and extracting the shared factors. For example, $140 = 2^2 \cdot 5 \cdot 7$ and $650 = 2 \cdot 5^2 \cdot 13$. So $\gcd(140, 6500)$ is $2 \cdot 5 = 10$.

Similarly, a common multiple of a and b is a number c such that $a|c$ and $b|c$. The least common multiple (lcm) is the smallest positive number for which this is true. The lcm can be computed using the formula:

$$\text{lcm}(a, b) = \frac{ab}{\gcd(a, b)}$$

For example, $\text{lcm}(140, 650) = \frac{140 \cdot 650}{10} = 9100$.

If two integers a and b share no common factors, then $\gcd(a, b) = 1$. Such a pair of integers are called **relatively prime**.

If k is a non-zero integer, then k divides zero. the largest common divisor of k and zero is k. So $\gcd(k, 0) = \gcd(0, k) = k$. However, $\gcd(0, 0)$ isn't defined. All integers are common divisors of 0 and 0, so there is no greatest one.

4.6 The division algorithm

The obvious way to compute the gcd of two integers is to factor both into primes and extract the shared factors. This is easy for small integers. However, it quickly becomes very difficult for larger integers, both for humans and computers. Fortunately, there's a fast method, called the Euclidean algorithm. Before presenting this algorithm, we need some background about integer division.

In general, when we divide one integer by another, we get a quotient and a remainder:

Theorem 1 (Division Algorithm) *The Division Algorithm: For any integers a and b, where b is positive, there are unique integers q (the quotient) and r (the remainder) such that $a = bq + r$ and $0 \leq r < b$.*

For example, if 13 is divided by 4, the quotient is 3 and the remainder is

1. Notice that the remainder is required to be non-negative. So -10 divided by 7 has the remainder 4, because $(-10) = 7 \cdot (-2) + 4$. This is the standard convention in mathematics. The word "unique" in this theorem simply means that only one pair of number q and r satisfy the pair of equations.

Now, notice the following non-obvious fact about the gcd:

Claim 23 *For any integers a, b, q and r, where b is positive, if $a = bq + r$, then* $\gcd(a, b) = \gcd(b, r)$.

> Proof: Suppose that n is some integer which divides both a and b. Then n divides bq and so n divides $a - bq$. (E.g. use various lemmas about divides from last week.) But $a - bq$ is just r. So n divides r.
>
> By an almost identical line of reasoning, if n divides both b and r, then n divides a.
>
> So, the set of common divisors of a and b is exactly the same as the set of common divisors of b and r. But $\gcd(a, b)$ and $\gcd(b, r)$ are just the largest numbers in these two sets, so if the sets contain the same things, the two gcd's must be equal.

If r is the remainder when a is divided by b, then $a = bq + r$, for some integer q. So we can immediately conclude that:

> Corollary: Suppose that a and b are integers and b is positive. Let r be the remainder when a is divided by b. Then $\gcd(a, b) = \gcd(b, r)$.

The term "corollary" means that this fact is a really easy consequence of the preceding claim.

4.7 Euclidean algorithm

We can now give a fast algorithm for computing gcd, which dates back to Euclid. Suppose that $remainder(a, b)$ returns the remainder when a is divided by b. Then we can compute the gcd as follows:

```
gcd(a,b: positive integers)
   x := a
   y := b
   while (y > 0)
      begin
      r := remainder(x,y)
      x := y
      y := r
      end
   return x
```

Let's trace this algorithm on inputs $a = 105$ and $b = 252$. Traces should summarize the values of the most important variables.

x	y	$r = \text{remainder}(x, y)$
105	252	105
252	105	42
105	42	21
42	21	0
21	0	

Since x is smaller than y, the first iteration of the loop swaps x and y. After that, each iteration reduces the sizes of a and b, because $a \bmod b$ is smaller than b. In the last iteration, y has gone to zero, so we output the value of x which is 21.

To verify that this algorithm is correct, we need to convince ourselves of two things. First, it must halt, because each iteration reduces the magnitude of y. Second, by our corollary above, the value of $\gcd(x, y)$ does not change from iteration to iteration. Moreover, $\gcd(x, 0)$ is x, for any non-zero integer x. So the final output will be the gcd of the two inputs a and b.

This is a genuinely very nice algorithm. Not only is it fast, but it involves very simple calculations that can be done by hand (without a calculator). It's much easier than factoring both numbers into primes, especially as the individual prime factors get larger. Most of us can't quickly see whether a large number is divisible by, say, 17.

4.8 Pseudocode

Notice that this algorithm is written in *pseudocode*. Pseudocode is an abstracted type of programming language, used to highlight the important structure of an algorithm and communicate between researchers who may not use the same programming language. It borrows many control constructs (e.g. the while loop) from imperative languages such as C. But details required only for a mechanical compiler (e.g. type declarations for all variables) are omitted and equations or words are used to hide details that are easy to figure out.

If you have taken a programming course, pseudocode is typically easy to read. Many small details are not standardized, e.g. is the test for equality written = or ==? However, it's usually easy for a human (though not a computer) to figure out what the author must have intended.

A common question is how much detail to use. Try to use about the same amount as in the examples shown in the notes. And think about how easily your pseudocode could be read by a classmate. Actual C or Java code is almost never acceptable pseudocode, because it is way too detailed.

4.9 A recursive version of gcd

We can also write gcd as a recursive algorithm

```
procedure gcd(a,b: positive integers)
r := remainder(a,b)
if (r = 0) return b
else return gcd(b,r)
```

This code is very simple, because this algorithm has a natural recursive structure. Our corollary allows us to express the gcd of two numbers in terms of the gcd of a smaller pair of numbers. That is to say, it allows us to reduce a larger version of the task to a smaller version of the same task.

4.10 Congruence mod k

Many applications of number theory, particularly in computer science, use modular arithmetic. In modular arithmetic, there are only a finite set of numbers and addition "wraps around" from the highest number to the lowest one. This is true, for example, for the 12 hours on a standard US clock: 3 hours after 11 o'clock is 2 o'clock, not 14 o'clock.

The formal mathematical definitions of modular arithmetic are based on the notion of congruence. Specifically, two integers are "congruent mod k" if they differ by a multiple of k. Formally:

> Definition: If k is any positive integer, two integers a and b are congruent mod k (written $a \equiv b \pmod{k}$) if $k \mid (a - b)$.

Notice that $k \mid (a - b)$ if and only if $k \mid (b - a)$. So it doesn't matter which number is subtracted from the other.

For example:

- $17 \equiv 5 \pmod{12}$ (Familiar to those of us who've had to convert between US 12-hour clocks and European/military 24-hour clocks.)
- $3 \equiv 10 \pmod{7}$
- $3 \equiv 38 \pmod{7}$ (Since $38 - 3 = 35$.)
- $38 \equiv 3 \pmod{7}$
- $-3 \equiv 4 \pmod{7}$ (Since $(-3) + 7 = 4$.)
- $-3 \not\equiv 3 \pmod{7}$
- $-3 \equiv 3 \pmod{6}$
- $-29 \equiv -13 \pmod{8}$ (Since $(-13) - (-29) = 16$.)

Congruence mod k is a relaxation of our normal rules for equality of numbers, in which we agree that certain pairs of numbers will be considered interchangeable.

4.11 Proofs with congruence mod k

Let's try using our definition to prove a simple fact about modular arithmetic:

Claim 24 *For any integers a, b, c, d, and k, k positive, if $a \equiv b \pmod{k}$ and $c \equiv d \pmod{k}$, then $a + c \equiv b + d \pmod{k}$.*

> Proof: Let a, b, c, d, and k be integers with k positive. Suppose that $a \equiv b \pmod{k}$ and $c \equiv d \pmod{k}$.
>
> Since $a \equiv b \pmod{k}$, $k \mid (a - b)$, by the definition of congruence mod k. Similarly, $c \equiv d \pmod{k}$, $k \mid (c - d)$.
>
> Since $k \mid (a - b)$ and $k \mid (c - d)$, we know by a lemma about divides (above) that $k \mid (a - b) + (c - d)$. So $k \mid (a + c) - (b + d)$
>
> But then the definition of congruence mod k tells us that $a + c \equiv b + d \pmod{k}$. □

This proof can easily be modified to show that

Claim 25 *For any integers a, b, c, d, and k, k positive, if $a \equiv b \pmod{k}$ and $c \equiv d \pmod{k}$, then $ac \equiv bd \pmod{k}$.*

So standard arithmetic operations interact well with our relaxed notion of equality.

4.12 Equivalence classes

The true power of modular conguence comes when we gather up a group of conguent integers and treat them all as a unit. Such a group is known as a **congruence class** or an **equivalence class**. Specifically, suppose that we fix a particular value for k. Then, if x is an integer, the equivalence class of x (written $[x]$) is the set of all integers congruent to x mod k. Or, equivalently, the set of integers that have remainder x when divided by k.

For example, suppose that we fix k to be 7. Then

$$[3] = \{3, 10, -4, 17, -11, \ldots\}$$
$$[1] = \{1, 8, -6, 15, -13, \ldots\}$$
$$[0] = \{0, 7, -7, 14, -14, \ldots\}$$

Notice that $[-4]$, and $[10]$ are exactly the same set as $[3]$. That is $[-4] = [10] = [3]$. So we have one object (the set) with many different names (one per integer in it). This is like a student apartment shared by Fred, Emily, Ali, and Michelle. The superficially different phrases "Emily's apartment" and "Ali's apartment" actually refer to one and the same apartment.

Having many names for the same object can become confusing, so people tend to choose a special preferred name for each object. For the k equivalence classes of integers mod k, mathematicians tend to prefer the names $[0], [1], \ldots, [k-1]$. Other names (e.g. $[30]$ when $k = 7$) tend to occur only as intermediate results in calculations.

Because standard arithmetic operations interact well with modular congruence, we can set up a system of arithmetic on these equivalence classes. Specifically, we define addition and multiplication on equivalence classes by:

$$[x] + [y] = [x + y]$$
$$[x] * [y] = [x * y]$$

So, (still setting $k = 7$) we can do computations such as

$$[4] + [10] = [4 + 10] = [14] = [0]$$
$$[-4] * [10] = [-4 * 10] = [-40] = [2]$$

This new set of numbers ($[0], [1], \ldots, [k-1]$), with these modular rules of arithmetic and equality, is known as the "integers mod k" or $\mathbb{Z}_k$ for short. For example, the addition and multiplication tables for $\mathbb{Z}_4$ are:

$+_4$	[0]	[1]	[2]	[3]
[0]	[0]	[1]	[2]	[3]
[1]	[1]	[2]	[3]	[0]
[2]	[2]	[3]	[0]	[1]
[3]	[3]	[0]	[1]	[2]

$\times_4$	[0]	[1]	[2]	[3]
[0]	[0]	[0]	[0]	[0]
[1]	[0]	[1]	[2]	[3]
[2]	[0]	[2]	[0]	[2]
[3]	[0]	[3]	[2]	[1]

People making extensive use of modular arithmetic frequently drop the square brackets. We're keeping the brackets for the moment to help you understand more clearly how the modular integers are created from the normal integers.

4.13 Wider perspective on equivalence

Modular arithmetic is often useful in describing periodic phenomena. For example, hours of the day form a circular space that is $\mathbb{Z}_{12}$. However, the tradition in time-keeping is to prefer the names $[1], [2], \ldots, [12]$ rather than the $[0], [2], \ldots, [11]$ traditional in math.

Low-precision integer storage on computers frequently uses modular arithmetic. For example, values in digitized pictures are often stored as 8-bit unsigned numbers. These numbers range from 0 to 255, i.e. the structure is $\mathbb{Z}_{256}$. In $\mathbb{Z}_{256}$, if you add [17] to [242], the result will be [3].

Equivalence classes of a more general type are used to construct rational numbers. That is, rational numbers are basically fractions, except that two fractions $\frac{x}{y}$ and $\frac{p}{q}$ are considered equivalent if $xq = py$. So, for example, the set $[\frac{2}{3}]$ would contain values such as $\frac{2}{3}$, $\frac{6}{9}$, and $\frac{-4}{-6}$. Most people prefer to name a rational number using a fraction in lowest terms.

Further afield, notice that musicians have more than one name for each note. There are seven note names, each of which can be sharped or flatted. However, there are only 12 different notes in a standard Western scale: many of these 21 names actually refer to the same note.[1] For example, A# is the same note as Bb. So, using more mathematical notation than the musicians would find comfortable, we could say that $[A\#]$ contains both A# and Bb.

[1] More precisely, this is true on instruments like pianos. More subtle note distinctions are possible on some other instruments.

4.14 Variation in Terminology

In these notes, a divides b is defined to be true if $b = an$ for some integer n. There is some variation among authors as to what happens when a is zero. Clearly, a non-zero number can't be a multiple of zero. But is zero a multiple of itself? According to our definition, it is, but some authors explicitly exclude this special case. Fortunately, this is a special case that one rarely sees in practice. The greatest common divisor is also known as the highest common factor (HCF).

In the shorthand notation $a \equiv b \pmod{k}$, the notation $\pmod{k}$ is logically a modifier on our notion of equality ($\equiv$). In retrospect, it might have made more sense to write something like $a \equiv_k b$. However, $a \equiv b \pmod{k}$ has become the standard notation and we have to live with it.

There are many variant notations for the quotient and remainder created by integer division, particularly if you include the functions built in to most programming languages. Popular names for the remainder include mod, modulo, rem, remainder, and %. The behavior on negative inputs differs from language to language and, in some cases, from implementation to implementation.[2] This lack of standardization often results in hard-to-find program bugs.

Some authors use $\mathbb{Z}/n\mathbb{Z}$ instead of $\mathbb{Z}_n$ as the shorthand name for the integers mod n. The notation $\mathbb{Z}_n$ may then refer to a set which is structurally the same as the integers mod n, but in which multiplication and exponentiation are used as the basic operations, rather than addition and multiplication. The equivalence class of n is sometimes written $\overline{n}$.

[2]The "modulo operation" entry on wikipedia has a nice table of what happens in different languages.

Chapter 5

Sets

So far, we've been assuming only a basic understanding of sets. It's time to discuss sets systematically, including a useful range of constructions, operations, notation, and special cases. We'll also see how to compute the sizes of sets and prove claims involving sets.

5.1 Sets

Sets are an extremely general concept, defined as follows:

> Definition: A set is an unordered collection of objects.

For example, the natural numbers are a set. So are the integers between 3 and 7 (inclusive). So are all the planets in this solar system or all the programs written by students in CS 225 in the last three years. The objects in a set can be anything you want.

The items in the set are called its elements or members. We've already seen the notation for this: $x \in A$ means that x is a member of the set A.

There's three basic ways to define a set:

- describe its contents in mathematical English, e.g. "the integers between 3 and 7, inclusive."

- list all its members, e.g. $\{3, 4, 5, 6, 7\}$
- use so-called set builder notation, e.g. $\{x \in \mathbb{Z} \mid 3 \leq x \leq 7\}$

Set builder notation has two parts separated with a vertical bar or a colon. The first part names a variable (in this case x) that ranges over all objects in the set. The second part gives one or more constraints that these objects must satisfy, e.g. $3 \leq x \leq 7$. The type of the variable (integer in our example) can be specified either before or after the vertical bar. The separator (| or :) is often read "such that."

Here's an example of a set containing an infinite number of objects

- "multiples of 7"
- $\{\ldots -14, -7, 0, 7, 14, 21, 28, \ldots\}$
- $\{x \in \mathbb{Z} \mid x \text{ is a multiple of } 7\}$

We couldn't list all the elements, so we had to use "...". This is only a good idea if the pattern will be clear to your reader. If you aren't sure, use one of the other methods.

If we wanted to use shorthand for "multiple of", it might be confusing to have | used for two different purposes. So it would probably be best to use the colon variant of set builder notation:

$\{x \in \mathbb{Z} : 7 \mid x\}$

The notation can be used on sets containing non-numerical objects. For example, suppose that $A = \{\texttt{dog}, \texttt{cat}, \texttt{tree}\}$. Then the set $\{\alpha\texttt{s} : \alpha \in A\}$ contains the strings `dogs`, `cats`, and `trees`.

5.2 Things to be careful about

A set is an unordered collection. So $\{1, 2, 3\}$ and $\{2, 3, 1\}$ are two names for the same set. Each element occurs only once in a set. Or, alternatively, it doesn't matter how many times you write it. So $\{1, 2, 3\}$ and $\{1, 2, 3, 2\}$ also name the same set.

We've seen ordered pairs and triples of numbers, such as $(3, 4)$ and $(4, 5, 2)$. The general term for an ordered sequence of k numbers is a k-tuple.[1] Tuples are very different from sets, in that the order of values in a tuple matters and duplicate elements don't magically collapse. So $(1, 2, 2, 3) \neq (1, 2, 3)$ and $(1, 2, 2, 3) \neq (2, 2, 1, 3)$. Therefore, make sure to enclose the elements of a set in curly brackets and carefully distinguish curly brackets (set) from parentheses (ordered pair).

A more subtle feature of tuples is that a tuple must contain at least two elements. In formal mathematics, a 1-dimensional value x is just written as x, not as (x). And there's no such thing in mathematics as a 0-tuple. So a tuple is simply a way of grouping two or more values into a single object.

By contrast, a set is like a cardboard box, into which you can put objects. A kitty is different from a box containing a kitty. Similarly, a set containing a single object is different from the object by itself. For example, $\{57\}$ is not the same as 57. A set can also contain nothing at all, like an empty box. The set containing nothing is called the empty set or the null set, and has the shorthand symbol $\emptyset$.[2]

The empty set may seem like a pain in the neck. However, computer science applications are full of empty lists, strings of zero length, and the like. It's the kind of special case that all of you (even the non-theoreticians) will spend your life having to watch out for.

Both sets and tuples can contain objects of more than one type, e.g. $(\text{cat}, \text{Fluffy}, 1983)$ or $\{a, b, 3, 7\}$. A set can also contain complex objects, e.g. $\{(a, b), (1, 2, 3), 6\}$ is a set containing three objects: an ordered pair, an ordered triple, and a single number.

5.3 Cardinality, inclusion

If A is a finite set (a set containing only a finite number of objects), then $|A|$ is the number of (different) objects in A. This is also called the **cardinality**

[1] There are latinate terms for longer sequences of numbers, e.g. quadruple, but they aren't used much.

[2] Don't write the emptyset as {}. Like a spelling mistake, this will make readers think less of your mathematical skills.

of A. For example, $|\{a, b, 3\}| = 3$. And $|\{a, b, a, 3\}|$ is also 3, because we count a group of identical objects only once. The notation of cardinality also extends to sets with infinitely many members ("infinite sets") such as the integers, but we won't get into the details of that right now.

Notice that the notation $|A|$ might mean set cardinality or it might be the more familiar absolute value. To tell which, figure out what type of object A is. If it's a set, the author meant cardinality. If it's a number, the author meant absolute value.

If A and B are sets, then A is a subset of B (written $A \subseteq B$) if every element of A is also in B. Or, if you want it formally: $\forall x, x \in A \rightarrow x \in B$. For example, $\mathbb{Q} \subseteq \mathbb{R}$, because every member of the rationals is also a member of the reals.

The notion of subset allows the two sets to be equal. So $A \subseteq A$ is true for any set A. So $\subseteq$ is like $\leq$. If you want to force the two sets to be different (i.e. like $<$), you must say that A is a **proper** subset of B, written $A \subset B$. You'll occasionally see reversed versions of these symbols to indicate the opposite relation, e.g. $B \supseteq A$ means the same as $A \subseteq B$.

5.4 Vacuous truth

If we have a set A, an interesting question is whether the empty set should be considered a subset of A. To answer this, let's first back up and look at one subtlety of mathematical logic.

Consider the following claim:

Claim 26 *For all natural numbers n, if $14 + n < 10$, then n wood elves will attack Siebel Center tomorrow.*

I claim this is true, a fact which most students find counter-intuitive. In fact, it wouldn't be true if n was declared to be an integer.

Notice that this statement has the form $\forall n, P(n) \rightarrow Q(n)$, where $P(n)$ is the predicate $14+n < 10$. Because n is declared to be a natural number, n is never negative, so $n + 14$ will always be at least 14. So $P(n)$ is always false.

Therefore, our conventions about the truth values for conditional statements imply that $P(n) \to Q(n)$ is true. This argument works for any choice of n. So $\forall n, P(n) \to Q(n)$ is true.

Because even mathematicians find such statements a bit wierd, they typically say that such a claim is *vacuously* true, to emphasize to the reader that it is only true because of this strange convention about the meaning of conditionals. Vacuously true statements typically occur when you are trying to apply a definition or theorem to a special case involving an abnormally small or simple object, such as the empty set or zero or a graph with no arrows at all.

In particular, this means that the empty set is a subset of any set A. For $\emptyset$ to be a subset of A, the definition of "subset" requires that for every object x, if x is an element of the empty set, then x is an element of A. But this if/then statement is considered true because its hypothesis is always false.

5.5 Set operations

Given two sets A and B, the intersection of A and B ($A \cap B$) is the set containing all objects that are in both A and B. In set builder notation:

$$A \cap B = \{S \mid S \in A \text{ and } S \in B\}$$

Let's set up some sample sets:

- $M = \{\text{egg}, \text{bread}, \text{milk}\}$
- $P = \{\text{milk}, \text{egg}, \text{flour}\}$

Then $M \cap P$ is $\{\text{milk}, \text{egg}\}$.

If the intersection of two sets A and B is the empty set, i.e. the two sets have no elements in common, then A and B are said to be **disjoint**.

The union of sets A and B ($A \cup B$) is the set containing all objects that are in one (or both) of A and B. So $M \cup P$ is $\{\text{milk}, \text{egg}, \text{bread}, \text{flour}\}$.

The set difference of A and B ($A - B$) contains all the objects that are in A but not in B. In this case,

$$M - P = \{\text{bread}\}$$

The complement of a set A ($\overline{A}$) contains all objects that aren't in A. For this to make sense, you need to define your "universal set" (often written U). U contains all the objects of the sort(s) you are discussing. For example, in some discussions, U might be all real numbers. In other discussions, U might contain all complex numbers. We cannot define a single, all-purpose universal set containing everything you might imagine putting in a set, because constructing a set that extensive leads to logical paradoxes. So your definition of U must be specific to your situation and, therefore, you and your reader need to come to an understanding about what's in it.

So, if our universe U is all integers, and A contains all the multiples of 3, then $\overline{A}$ is all the integers whose remainder mod 3 is either 1 or 2. $\overline{\mathbb{Q}}$ would be the irrational numbers if our universe is all real numbers. If we had been working with complex numbers, it might be the set of all irrational real numbers plus all the numbers with an imaginary component.

If A and B are two sets, their Cartesian product ($A \times B$) contains all ordered pairs (x, y) where x is in A and y is in B. That is

$$A \times B = \{(x, y) \mid x \in A \text{ and } y \in B\}$$

For example, if $A = \{a, b\}$ and $B = \{1, 2\}$, then

$$A \times B = \{(a, 1), (a, 2), (b, 1), (b, 2)\}$$
$$B \times A = \{(1, a), (2, a), (1, b), (2, b)\}$$

Notice that these two sets aren't equal: order matters for Cartesian products.

We can generalize this definition to create the Cartesian product of three or more sets. So, for example, if $C = \{p, q\}$, then

$A \times B \times C == \{(a,1,p),(a,1,q),(a,2,p),(a,2,q),(b,1,p),(b,1,q),(b,2,p),(b,2,q)\}$

5.6 Set identities

It's easy to find (e.g. on the net), long lists of identities showing when two sequences of set operations yield the same output set. For example:

DeMorgan's Law: $\overline{A \cup B} = \overline{A} \cap \overline{B}$

I won't go through these in detail because they are largely identical to the identities you saw for logical operations, if you make the following correspondences:

- $\cup$ is like $\vee$
- $\cap$ is like $\wedge$
- $\overline{A}$ is like $\neg P$
- $\emptyset$ (the empty set) is like F
- U (the universal set) is like T

The two systems aren't exactly the same. E.g. set theory doesn't use a close analog of the $\rightarrow$ operator. But they are very similar.

5.7 Size of set union

Many applications require that we calculate[3] the size of the set created by applying set operations. These sets are often sets of options for some task.

[3]Or sometimes just estimate.

If two sets A and B don't intersect, then the size of their union is just the sum of their sizes. That is: $|A \cup B| = |A| + |B|$. For example, suppose that it's late evening and you want to watch a movie. You have 37 movies on cable, 57 DVD's on the shelf, and 12 movies stored in I-tunes. If these three sets of movies don't intersect, you have a total of $37 + 57 + 12 = 106$ movies.

If your input sets do overlap, then adding up their sizes will double-count some of the objects. So, to get the right number for the union, we need to correct for the double-counting. For our movie example, suppose that the only overlap is that 2 movies are on I-tunes and also on DVD. Then you would have $(37 + 57 + 12) - 2 = 104$ movies to choose from.

The formal name for this correction is the "Inclusion-Exclusion Principle". Formally, suppose you have two sets A and B. Then

$$\text{Inclusion-Exclusion Principle: } |A \cup B| = |A| + |B| - |A \cap B|$$

We can use this basic 2-set formula to derive the corresponding formula for three sets A, B, and C:

$$\begin{aligned}
|A \cup B \cup C| &= |A| + |B \cup C| - |A \cap (B \cup C)| \\
&= |A| + |B| + |C| - |B \cap C| - |A \cap (B \cup C)| \\
&= |A| + |B| + |C| - |B \cap C| - |(A \cap B) \cup (A \cap C)| \\
&= |A| + |B| + |C| - |B \cap C| - (|A \cap B| + |A \cap C| - |(A \cap B) \cap (A \cap C)|) \\
&= |A| + |B| + |C| - |B \cap C| - |A \cap B| - |A \cap C| + |A \cap B \cap C|
\end{aligned}$$

In addition to the inclusion-exclusion principle, this derivation uses the distributive law (third step) and the fact that intersection is commutative and associative (last step).

5.8 Product rule

Now, suppose that we form the Cartesian product of two sets A and B, where $|A| = n$ and $|B| = q$. To form an element (x, y) in the product, we have n

choices for x and q choices for y. So we have nq ways to create an element of the product. So $|A \times B| = nq$.

In general:

> **The product rule**: if you have p choices for one part of a task, then q choices for a second part, and your options for the second part don't depend on what you chose for the first part, then you have pq options for the whole task.

This rule generalizes in the obvious way to sequences of several choices: you multiply the number of options for each choice. For example, suppose that T-shirts come in 4 colors, 5 sizes, 2 sleeve lengths, and 3 styles of neckline, there are $4 \cdot 5 \cdot 2 \cdot 3 = 120$ total types of shirts.

We could represent a specific T-shirt type as a 4-tuple (c, s, l, n): c is its color, s is its size, l is its sleeve length, and n is its neckline. E.g. one T-shirt type is $(\text{red}, \text{small}, \text{long}, \text{vee})$ The set of all possible T-shirt types would then be a 4-way Cartesian product of the set of all colors, the set of all sizes, the set of all sleeve lengths, and the set of all necklines.

5.9 Combining these basic rules

These two basic counting rules can be combined to solve more complex practical counting problems. For example, suppose we have a set S which contains all 5-digit decimal numbers that start with 2 one's or end in 2 zeros, where we don't allow leading zeros. How large is S, i.e. how many numbers have this form?

Let T be the set of 5-digit numbers starting in 2 one's. We know the first two digits and we have three independent choices (10 options each) for the last three. So there are 1000 numbers in T.

Let R be the set of 5-digit numbers ending in 2 zeros. We have 9 options for the first digit, since it can't be zero. We have 10 options each for the second and third digits, and the last two are fixed. So we have 900 numbers in R.

What's the size of $T \cap R$? Numbers in this set start with 2 one's and end with 2 zeros, so the only choice is the middle digit. So it contains 10 numbers. So

$$|S| = |T| + |R| - |T \cap R| = 1000 + 900 - 10 = 1890$$

5.10 Proving facts about set inclusion

So far in school, most of your proofs or derivations have involved reasoning about equality. Inequalities (e.g. involving numbers) have been much less common. With sets, the situation is reversed. Proofs typically involve reasoning about subset relations, even when proving two sets to be equal. Proofs that rely primarily on a chain of set equalities do occur, but they are much less common. Even when both approaches are possible, the approach based on subset relations is often easier to write and debug.

As a first example of a typical set proof, let's suppose that we the following two sets and we'd like to prove that $A \subseteq B$

$$A = \{\lambda(2,3) + (1-\lambda)(7,4) \mid \lambda \in [0,1]\}$$
$$B = \{(x,y) \mid x, y \in \mathbb{R}, x \geq 0, \text{ and } y \geq 0\}$$

When presented with a claim like this, you should first take a minute to verify that the claim really is true. Set B is the upper right quadrant of the plane. To understand the definition of set A, remember how to multiply a real number by a vector: $a(x,y) = (ax, ay)$. This definition generates all points of the form $\lambda(2,3) + (1-\lambda)(7,4)$ where λ is a real number between 0 and 1. Try putting some sample values of λ into this equation and plotting them in 2D: what geometrical object is this?[4] Make sure you believe that this object does live in the upper right quadrant.

Now, remember our definition of $\subseteq$: a set A is a subset of a set B if and only if, for any object x, $x \in A$ implies that $x \in B$. So, to prove our claim, we need to pick a random object x from A and show that it lives in B. So a starting sketch of our proof might look like:

[4]Hint: use a pencil and paper. Your plot doesn't need to be neat to see the pattern.

> Proof: Let sets A and B be defined as above. Let x be an element of A.
>
> [missing details]
>
> So x is an element of B.
>
> Since x was arbitrarily chosen, we've shown that any element of A is also an element of B. So A is a subset of B.

We can now use the definition of A to extend forward from the hypotheses into the missing part. In particular, the definition of A implies that x is a 2D point, so it's probably helpful to give names to the two coordinates:

> Proof: Let sets A and B be defined as above. Let x be an element of A. Then $x = \mu(2,3) + (1-\mu)(7,4)$ for some $\mu \in [0,1]$. So $x = (p,q)$ where $p = 2\mu + 7(1-\mu)$ and $q = 3\mu + 4(1-\mu)$
>
> [missing details]
>
> So x is an element of B.
>
> Since x was arbitrarily chosen, we've shown that any element of A is also an element of B. So A is a subset of B.

Notice that the variable λ in the definition of A is local to the definition of A. So when we use this definition to spell out the properties of our element x, we need to introduce a new variable. I've used a fresh variable name μ to emphasize that this is a new variable.

At this point, it's worth looking at the end part of the proof. If $x = (p,q)$ and we're trying to show that x is in B, then this translates into showing that p and q are both non-negative. So we can expand the end of the proof backwards, narrowing the missing part even further:

> Proof: Let sets A and B be defined as above. Let x be an element of A. Then $x = \mu(2,3) + (1-\mu)(7,4)$ for some $\mu \in [0,1]$. So $x = (p,q)$ where $p = 2\mu + 7(1-\mu)$ and $q = 3\mu + 4(1-\mu)$
>
> [missing details]
>
> So $p \geq 0$ and $q \geq 0$. This means that $x = (p,q)$ is an element of B.

> Since x was arbitrarily chosen, we've shown that any element of A is also an element of B. So A is a subset of B.

Now we have a straightforward algebra problem: get from the complex equations defining p and q to the fact that both are non-negative. Working out the details of that algebra gives us the final proof:

> Proof: Let sets A and B be defined as above. Let x be an element of A. Then $x = \mu(2,3) + (1-\mu)(7,4)$ for some $\mu \in [0,1]$. So $x = (p,q)$ where $p = 2\mu + 7(1-\mu)$ and $q = 3\mu + 4(1-\mu)$
>
> Simplifying these equations, we get that $p = 2\mu + 7 - 7\mu = 7 - 5\mu$ and $q = 3\mu + 4 - 4\mu = 4 - \mu$ Since μ is in the interval $[0,1]$, $\mu \leq 1$. So $7 - 5\mu$ and $4 - \mu$ are both ≥ 0.
>
> So $p \geq 0$ and $q \geq 0$. This means that $x = (p,q)$ is an element of B.
>
> Since x was arbitrarily chosen, we've shown that any element of A is also an element of B. So A is a subset of B.

The last paragraph in this proof is actually is optional. When you first start, it's a useful recap because you might be a bit fuzzy about what you needed to prove. Experienced folks often omit it, depending on the reader to understand the basic outline of a subset inclusion proof. But you will still see it occasionally at the end of a very long (e.g. multi-page) proof, where the reader might have been so buried in the details that they've lost track of the overall goals of the proof.

5.11 An abstract example

Now, let's try to do a similar proof, but for a claim involving generic sets rather than specific sets.

Claim 27 *For any sets A, B, and C, if $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$.*

This property is called "transitivity," just like similar properties for (say) $\leq$ on the real numbers. Both $\subseteq$ and $\leq$ are examples of a general type of object called a *partial order*, for which transitivity is a key defining property.

Let's start our proof by gathering up all the given information in the hypothesis of the claim:

> Proof: Let A, B, and C be sets and suppose that $A \subseteq B$ and $B \subseteq C$.

Our ultimate goal is to show that $A \subseteq C$. This is an if/then statement: for any x, if $x \in A$, then $x \in C$. So we need to pick a representative x and assume the hypothesis is true, then show the conclusion. So our proof continues:

> Let x be an element of A. Since $A \subseteq B$ and $x \in A$, then $x \in B$ (definition of subset). Similarly, since $x \in B$ and $B \subseteq C$, $x \in C$. So for any x, if $x \in A$, then $x \in C$. So $A \subseteq C$ (definition of subset again). $\square$

5.12 An example with products

Here's another claim, involving Cartesian products:

Claim 28 *For any sets A, B, and C, if $A \times B \subseteq A \times C$ and $A \neq \emptyset$, then $B \subseteq C$.*

Notice that this claim fails if $A = \emptyset$. For example, $\emptyset \times \{1, 2, 3\}$ is a subset of $\emptyset \times \{a, b\}$, because both of these sets are empty. However $\{1, 2, 3\}$ is not a subset of $\{a, b\}$.

This is like dividing both sides of an algebraic equation by a non-zero number: if $xy \leq xz$ and $x \neq 0$ then $y \leq z$. This doesn't work we allow x to be zero. Set operations don't always work exactly like operations on real numbers, but the parallelism is strong enough to suggest special cases that ought to be investigated.

A general property of proofs is that the proof should use all the information in the hypothesis of the claim. If that's not the case, either the proof has a bug (e.g. on a homework assignment) or the claim could be revised to make it more interesting (e.g. when doing a research problem, or a buggy homework problem). Either way, there's an important issue to deal with. So, in this case, we need to make sure that our proof does use the fact that $A \neq \emptyset$.

Here's a draft proof:

> Proof draft: Suppose that A, B, and C are sets and suppose that $A \times B \subseteq A \times C$ and $A \neq \emptyset$. We need to show that $B \subseteq C$.
>
> So let's choose some $x \in B$. ...

The main fact we've been given is that $A \times B \subseteq A \times C$. To use it, we need an element of $A \times B$. Right now, we only have an element of B. We need to find an element of A to pair it with. To do this, we reach blindly into A, pull out some random element, and give it a name. But we have to be careful here: what if A doesn't contain any elements? So we have to use the assumption that $A \neq \emptyset$.

> Proof: Suppose that A, B, and C are sets and suppose that $A \times B \subseteq A \times C$ and $A \neq \emptyset$. We need to show that $B \subseteq C$.
>
> So let's choose some $x \in B$. Since $A \neq \emptyset$, we can choose an element t from A. Then $(t, x) \in A \times B$ by the definition of Cartesian product.
>
> Since $(t, x) \in A \times B$ and $A \times B \subseteq A \times C$, we must have that $(t, x) \in A \times C$ (by the definition of subset). But then (again by the definition of Cartesian product) $x \in C$.
>
> So we've shown that if $x \in B$, then $x \in C$. So $B \subseteq C$, which is what we needed to show.

5.13 A proof using sets and contrapositive

Here's a claim about sets that's less than obvious:

Claim 29 *For any sets A and B, if $(A - B) \cup (B - A) = A \cup B$ then $A \cap B = \emptyset$.*

Notice that the conclusion $A \cap B = \emptyset$ claims that something does not exist (i.e. an object that's in both A and B). So this is a good place to apply proof by contrapositive.

> Proof: Let's prove the contrapositive. That is, we'll prove that if $A \cap B \neq \emptyset$, then $(A - B) \cup (B - A) \neq A \cup B$.
>
> So, let A and B be sets and suppose that $A \cap B \neq \emptyset$. Since $A \cap B \neq \emptyset$, we can choose an element from $A \cap B$. Let's call it x.
>
> Since x is in $A \cap B$, x is in both A and B. So x is in $A \cup B$.
>
> However, since x is in B, x is not in $A - B$. Similarly, since x is in A, x is not in $B - A$. So x is not a member of $(A - B) \cup (B - A)$. This means that $(A - B) \cup (B - A)$ and $A \cup B$ cannot be equal, because x is in the second set but not in the first. $\square$.

5.14 Variation in notation

In mathematical writing, it is quite rare to create tuples that contain other tuples. In principle, tuples containing tuples are formally different from longer tuples. E.g. $((a, b), c)$ is formally a different object from (a, b, c), and $(a, (b, c))$ is different from both. However, most mathematical writers treat this difference as a petty nuisance and view the two objects as interchangeable.

Linked lists used in computer science behave very differently from mathematical tuples. The linked list $((a, b), c)$ is completely different from the list (a, b, c) and a linked list can contain a single element or no elements. It is a bad idea to confuse linked lists and mathatical tuples, even though they are written with the same notation.

Chapter 6

Relations

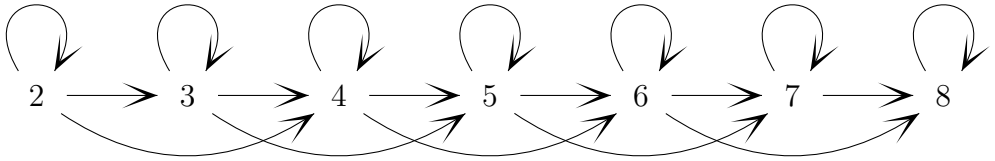
Mathematical relations are an extremely general framework for specifying relationships between pairs of objects. This chapter surveys the types of relations that can be constructed on a single set A and the properties used to characterize different types of relations.

6.1 Relations

A *relation R on a set A* is a subset of $A \times A$, i.e. R is a set of ordered pairs of elements from A. For simplicity, we will assume that the base set A is always non-empty. If R contains the pair (x, y), we say that x is related to y, or xRy in shorthand. We'll write $x \not R y$ to mean that x is not related to y.

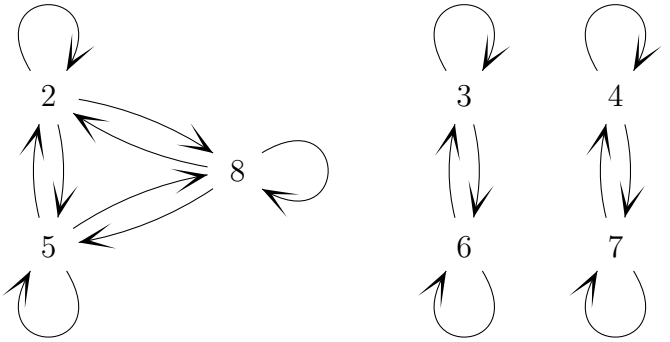
For example, suppose we let $A = \{2, 3, 4, 5, 6, 7, 8\}$. We can define a relation W on A by xWy if and only if $x \leq y \leq x + 2$. Then W contains pairs like $(3, 4)$ and $(4, 6)$, but not the pairs $(6, 4)$ and $(3, 6)$. Under this relation, each element of A is related to itself. So W also contains pairs like $(5, 5)$.

We can draw pictures of relations using directed graphs. We draw a graph node for each element of A and we draw an arrow joining each pair of elements that are related. So W looks like:

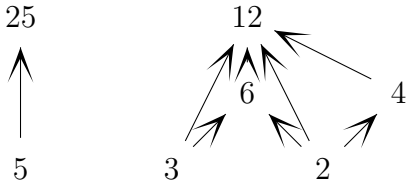


In fact, there's very little formal difference between a relation on a set A and a directed graph, because graph edges can be represented as ordered pairs of endpoints. They are two ways of describing the same situation.

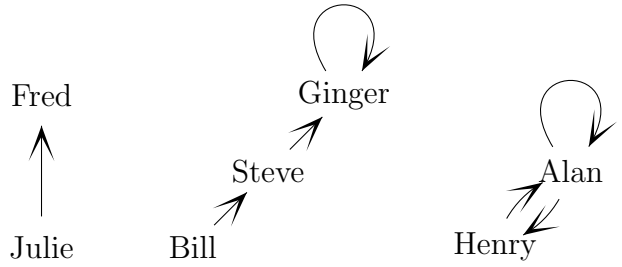
We can define another relation S on A by saying that xSy is in S if $x \equiv y \pmod 3$. Then S would look like:



Or, suppose that $B = \{2, 3, 4, 5, 6, 12, 25\}$. Let's set up a relation T on B such that xTy if $x|y$ and $x \neq y$. Then our picture would look like



Mathematical relations can also be used to represent real-world relationships, in which case they often have a less regular structure. For example, suppose that we have a set of students and student x is related to student y if x nominated y for ACM president. The graph of this relation (call it Q) might look like:



Relations can also be defined on infinite sets or multi-dimensional objects. For example, we can define a relation Z on the real plane $\mathbb{R}^2$ in which (x, y) is related to (p, q) if and only if $x^2 + y^2 = p^2 + q^2$. In other words, two points are related if they are the same distance from the origin.

For complex relations, the full directed graph picture can get a bit messy. So there are simplified types of diagrams for certain specific special types of relations, e.g. the so-called Hasse diagram for partial orders.

6.2 Properties of relations: reflexive

Relations are classified by several key properties. The first is whether an element of the set is related to itself or not. There are three cases:

- Reflexive: every element is related to itself.
- Irreflexive: no element is related to itself.
- Neither reflexive nor irreflexive: some elements are related to themselves but some aren't.

In our pictures above, elements related to themselves have self-loops. So it's easy to see that W and S are reflexive, T is irreflexive, and Q is neither. The familiar relations $\leq$ and $=$ on the real numbers are reflexive, but $<$ is irreflexive. Suppose we define a relation M on the integers by xMy if and only if $x + y = 0$. Then 2 isn't related to itself, but 0 is. So M is neither reflexive nor irreflexive.

The formal definition states that if R is a relation on a set A then

- R is reflexive if xRx for all $x \in A$.
- R is irreflexive if $x \not R x$ for all $x \in A$.

Notice that irreflexive is not the negation of reflexive. The negation of reflexive would be:

- not reflexive: there is an $x \in A, x \not R x$

6.3 Symmetric and antisymmetric

Another important property of a relation is whether the order matters within each pair. That is, if xRy is in R, is it always the case that yRx? If this is true, then the relation is called *symmetric*.

In a graph picture of a symmetric relation, a pair of elements is either joined by a pair of arrows going in opposite directions, or no arrows. In our examples with pictures above, only S is symmetric.

Relations that resemble equality are normally symmetric. For example, the relation X on the integers defined by xXy iff $|x| = |y|$ is symmetric. So is the relation N on the real plane defined by $(x, y)N(p, q))$ iff $(x-p)^2+(y-q)^2 \leq 25$ (i.e. the two points are no more than 5 units apart).

Relations that put elements into an order, like $\leq$ or divides, have a different property called *antisymmetry*. A relation is *antisymmetric* if two distinct[1] elements are never related in both directions. In a graph picture of an antisymmetric relation, a pair of points may be joined by a single arrow, or not joined at all. They are never joined by arrows in both directions. In our pictures above, W and T are antisymmetric.

As with reflexivity, there are mixed relations that have neither property. So the relation Q above is neither symmetric nor antisymmetric.

If R is a relation on a set A, here's the formal definition of what it means for R to be symmetric (which doesn't contain anything particularly difficult):

[1] "Distinct" means not equal.

symmetric: for all $x, y \in A, xRy$ implies yRx

There's two ways to define antisymmetric. They are logically equivalent and you can pick whichever is more convenient for your purposes:

antisymmetric: for all x and y in A with $x \neq y$, if xRy, then $y \not R x$

antisymmetric: for all x and y in A, if xRy and yRx, then $x = y$

To interpret the second definition, remember that when mathematicians pick two values x and y, they leave open the possibility that the two values are actually the same. In normal conversational English, if we mention two objects with different names, we normally mean to imply that they are different objects. This isn't the case in mathematics. I find that the first definition of antisymmetry is better for understanding the idea, but the second is more useful for writing proofs.

6.4 Transitive

The final important property of relations is transitivity. A relation R on a set A is *transitive* if

transitive: for all $a, b, c \in A$, if aRb and bRc, then aRc

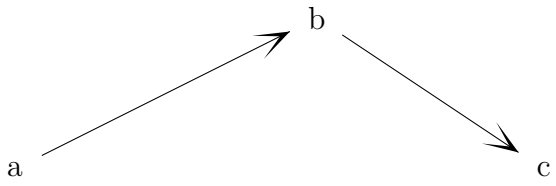
As we've seen earlier, transitivity holds for a broad range of familiar numerical relations such as $<$, $=$, divides, and set inclusion. For example, for real numbers, if $x < y$ and $y < z$, then $x < z$. Similarly, if $x|y$ and $y|z$, then $x|z$. For sets, $X \subseteq Y$ and $Y \subseteq Z$ implies that $X \subseteq Z$.

If we look at graph pictures, transitivity means that whenever there is an indirect path from x to y then there must also be a direct arrow from x to y. This is true for S and B above, but not for W or Q.

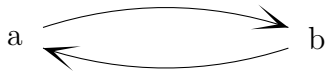
We can also understand this by spelling out what it means for a relation R on a set A not to be transitive:

not transitive: there are $a, b, c \in A, aRb$ and bRc and $a \not R c$

So, to show that a relation is not transitive, we need to find one counter-example, i.e. specific elements a, b, and c such that aRb and bRc but not aRc. In the graph of a non-transitive relation, you can find a subsection that looks like:

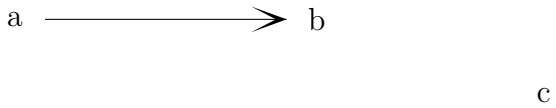


It could be that a and c are actually the same element, in which case the offending subgraph might look like:



The problem here is that if aRb and bRa, then transitivity would imply that aRa and bRb.

One subtle point about transitive is that it's an if/then statement. So it's ok if some sets of elements just aren't connected at all. For example, this subgraph is consistent with the relation being transitive.



A disgustingly counter-intuitive special case is the relation in which absolutely no elements are related, i.e. no arrows at all in the graph picture. This relation is transitive. it's never possible to satisfy the hypothesis of the

definition of transitive. It's also symmetric, for the same reason. And, oddly enough, antisymmetric. All of these properties hold via vacuous truth.

Vacuous truth doesn't apply to reflexive and irreflexive, because they are unconditional requirements, not if/then statements. So this empty relation is irreflexive and not reflexive.

6.5 Types of relations

Now that we have these basic properties defined, we can define some important types of relations. Three of these are ordering relations:

- A **partial order** is a relation that is reflexive, antisymmetric, and transitive.
- A **linear order** (also called a total order) is a partial order R in which every pair of elements are **comparable**. That is, for any two elements x and y, either xRy or yRx.
- A **strict partial order** is a relation that is irreflexive, antisymmetric, and transitive.

Linear orders are all very similar to the normal $\leq$ ordering on the real numbers or integers. Partial orders differ from linear orders, in that there are some pairs of elements which aren't ordered with respect to each other. For example, the divides relation on the integers is a partial order but not a linear order, because it doesn't relate some pairs of integers in either direction. For example, 5 doesn't divide 7, but neither does 7 divide 5. A strict partial order is just like a partial order, except that objects are not related to themselves. For example, the relation T in section 6.1 is a strict partial order.

The fourth type of relation is an equivalence relation:

> Definition: An **equivalence relation** is a relation that is reflexive, symmetric, and transitive.

These three properties create a well-behaved notation of equivalence or congruence for a set of objects, like congruence mod k on the set of integers. In particular, if R is an equivalence relation on a set A and x is an element of A, we can define the *equivalence class of* x to be the set of all elements related to x. That is

$$[x]_R = \{y \in A \mid xRy\}$$

When the relation is clear from the context (as when we discussed congruence mod k), we frequently omit the subscript on the square brackets.

For example, we saw the relation Z on the real plane $\mathbb{R}^2$, where $(x, y)Z(p, q)$ if and only if $x^2 + y^2 = p^2 + q^2$. Then $[(0, 1)]_Z$ contains all the points related to $(0, 1)$, i.e. the unit circle.

6.6 Proving that a relation is an equivalence relation

Let's see how to prove that a new relation is an equivalence relation. These proofs are usually very mechanical. For example, let F be the set of all fractions, i.e.

$$F = \{\frac{p}{q} \mid p, q \in \mathbb{Z} \text{ and } q \neq 0\}$$

Fractions aren't the same thing as rational numbers, because each rational number is represented by many fractions. We consider two fractions to be equivalent, i.e. represent the same rational number, if $xq = yp$. So, we have an equivalence relation $\sim$ defined by: $\frac{x}{y} \sim \frac{p}{q}$ if and only if $xq = yp$.

Let's show that $\sim$ is an equivalence relation.

> Proof: Reflexive: For any x and y, $xy = xy$. So the definition of $\sim$ implies that $\frac{x}{y} \sim \frac{x}{y}$.
> Symmetric: if $\frac{x}{y} \sim \frac{p}{q}$ then $xq = yp$, so $yp = xq$, so $py = qx$, which implies that $\frac{p}{q} \sim \frac{x}{y}$.

> Transitive: Suppose that $\frac{x}{y} \sim \frac{p}{q}$ and $\frac{p}{q} \sim \frac{s}{t}$. By the definition of $\sim$, $xq = yp$ and $pt = qs$. So $xqt = ypt$ and $pty = qsy$. Since $ypt = pty$, this means that $xqt = qsy$. Cancelling out the q's, we get $xt = sy$. By the definition of $\sim$, this means that $\frac{x}{y} \sim \frac{s}{t}$.
>
> Since $\sim$ is reflexive, symmetric, and transitive, it is an equivalence relation.

Notice that the proof has three sub-parts, each showing one of the key properties. Each part involves using the definition of the relation, plus a small amount of basic math. The reflexive case is very short. The symmetric case is often short as well. Most of the work is in the transitive case.

6.7 Proving antisymmetry

Here's another example of a relation, this time an order (not an equivalence) relation. Consider the set of intervals on the real line $J = \{(a, b) \mid a, b \in \mathbb{R} \text{ and } a < b\}$. Define the containment relation C as follows:

> $(a, b)\ C\ (c, d)$ if and only if $a \leq c$ and $d \leq b$

To show that C is a partial order, we'd need to show that it's reflexive, antisymmetric, and transitive. We've seen how to prove two of these properties. Let's see how to do a proof of antisymmetry.

For proving antisymmetry, it's typically easiest to use this form of the definition of antisymmetry: if xRy and yRx, then $x = y$. Notice that C is a relation on intervals, i.e. pairs of numbers, not single numbers. Substituting the definition of C into the definition of antisymmetry, we need to show that

> For any intervals (a, b) and (c, d), if $(a, b)\ C\ (c, d)$ and $(c, d)\ C\ (a, b)$, then $(a, b) = (c, d)$.

So, suppose that we have two intervals (a, b) and (c, d) such that $(a, b)\ C\ (c, d)$ and $(c, d)\ C\ (a, b)$. By the definition of C, $(a, b)\ C\ (c, d)$ implies that $a \leq c$ and $d \leq b$. Similarly, $(c, d)\ C\ (a, b)$ implies that $c \leq a$ and $b \leq d$.

Since $a \leq c$ and $c \leq a$, $a = c$. Since $d \leq b$ and $b \leq d$, $b = d$. So $(a, b) = (c, d)$.

Chapter 7

Functions and onto

This chapter covers functions, including function composition and what it means for a function to be *onto*. In the process, we'll see what happens when two dissimilar quantifiers are nested.

7.1 Functions

We're all familiar with functions from high school and calculus. However, these prior math courses concentrate on functions whose inputs and outputs are numbers, defined by an algebraic formula such as $f(x) = 2x + 3$. We'll be using a broader range of functions, whose input and/or output values may be integers, strings, characters, and the like.

Suppose that A and B are sets, then a function f from A to B is an assignment of exactly one element of B (i.e. the output value) to each element of A (i.e. the input value). A is called the *domain* of f and B is called the *co-domain*. All of this information can be captured in the shorthand *type signature*: $f : A \to B$. If x is an element of A, then the value $f(x)$ is also known as the *image* of x.

For example, suppose P is a set of five people:

$$P = \{\text{Margaret}, \text{Tom}, \text{Chen}, \text{LaSonya}, \text{Emma}\}$$

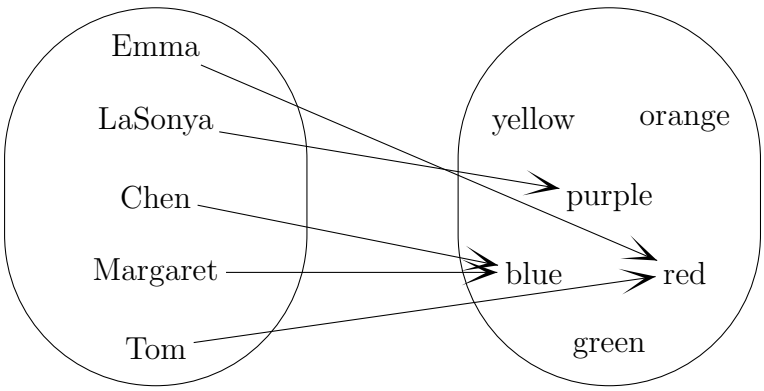
And suppose C is a set of colors:

$$C = \{\text{red}, \text{blue}, \text{green}, \text{purple}, \text{yellow}, \text{orange}\}$$

We can define a function $f : P \to C$ which maps each person to their favorite color. For example, we might have the following input/output pairs:

$$\begin{aligned} f(\text{Margaret}) &= \text{Blue} \\ f(\text{Tom}) &= \text{Red} \\ f(\text{LaSonya}) &= \text{Purple} \\ f(\text{Emma}) &= \text{Red} \\ f(\text{Chen}) &= \text{Blue} \end{aligned}$$

We also use a bubble diagram to show how f assigns output values to input values.



Even if A and B are finite sets, there are a very large number of possible functions from A to B. Suppose that $|A| = n$, $|B| = p$. We can write out the elements of A as $x_1, x_2, \ldots, x_n$. When constructing a function $f : A \to B$, we have p ways to choose the output value for x_1. The choice of $f(x_1)$ doesn't affect our possible choices for $f(x_2)$: we also have p choices for that value. So we have p^2 choices for the first two output values. If we continue this process for the rest of the elements of A, we have p^n possible ways to construct our function f.

For any set A, the *identity function* id_A maps each value in A to itself. That is, $\mathrm{id}_A : A \to A$ and $\mathrm{id}_A(x) = x$.

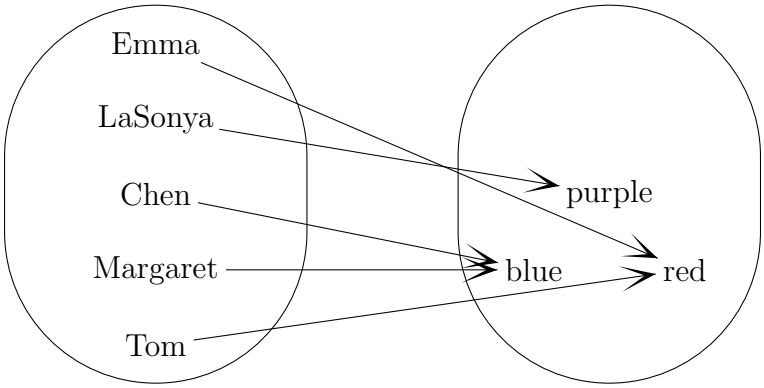
7.2 When are functions equal?

Notice that the domain and co-domain are an integral part of the definition of the function. To be equal, two functions must (obviously) assign the same output value to each input value. But, in addition, they must have the same *type signature*.

For example, suppose D is a slightly smaller set of colors:

$$D = \{\text{red}, \text{blue}, \text{purple}\}$$

Then the function $g : P \to D$ shown below is not equal to the function f from section 7.1, even though $g(x) = f(x)$ for every x in P.



Similarly, the following definitions describe quite different functions, even though they are based on the same equation.

- $f : \mathbb{N} \to \mathbb{N}$ such that $f(x) = 2x$.
- $f : \mathbb{R} \to \mathbb{R}$ such that $f(x) = 2x$.

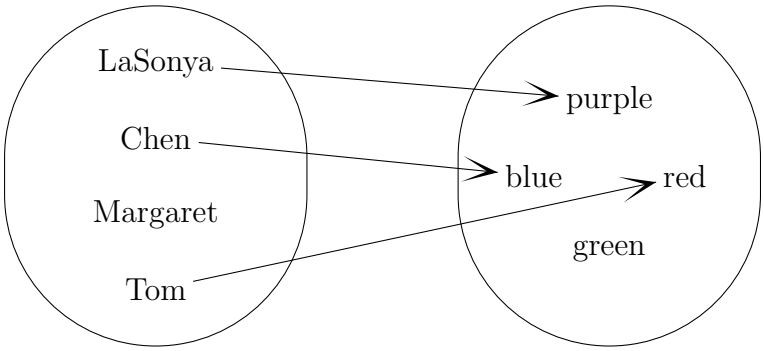
However, the following are all definitions of the same function, because the three variations have the same type signature and assign the same output value to each input value.

- $f : \mathbb{Z} \to \mathbb{Z}$ such that $f(x) = |x|$.
- $f : \mathbb{Z} \to \mathbb{Z}$ such that $f(x) = \max(x, -x)$.
- $f : \mathbb{Z} \to \mathbb{Z}$ such that $f(x) = x$ if $x \geq 0$ and $f(x) = -x$ if $x \leq 0$.

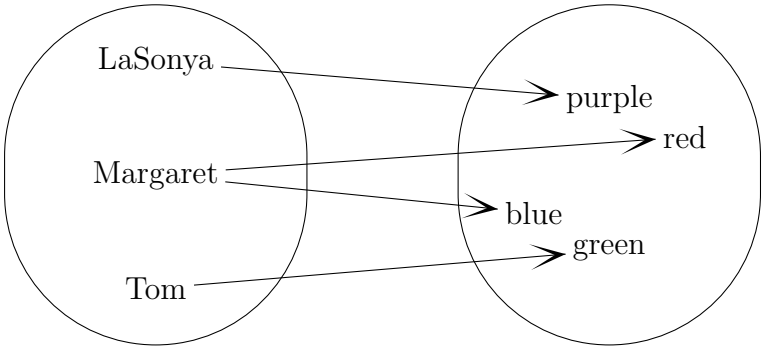
Notice that the last definition uses two different cases to cover different parts of the domain. This is fine, as long as all the cases, taken together, provide exactly one output value for each input value.

7.3 What isn't a function?

For each input value, a function must provide one and only one output value. So the following isn't a function, because one input has no output:



The following isn't a function, because one input is paired with two outputs:

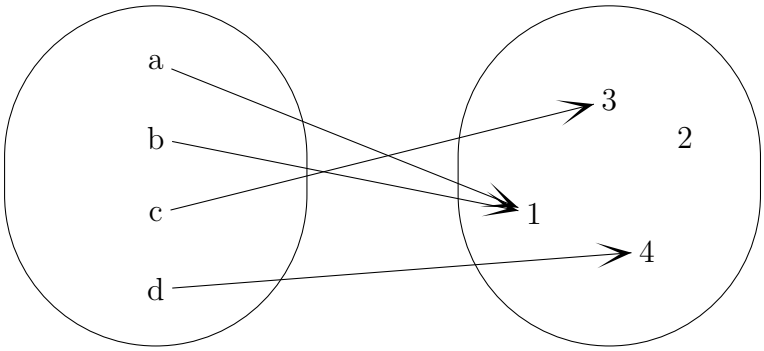


7.4 Images and Onto

The image of the function $f : A \to B$ is the set of values produced when f is applied to all elements of A. That is, the image is

$$f(A) = \{f(x) : x \in A\}$$

For example, suppose $M = \{a, b, c, d\}$, $N = \{1, 2, 3, 4\}$, and our function $g : M \to N$ is as in the following diagram. Then $g(A) = \{1, 3, 4\}$.



A function $f : A \to B$ is *onto* if its image is its whole co-domain. Or, equivalently,

$$\forall y \in B, \exists x \in A, f(x) = y$$

The function g that we just saw isn't onto, because no input value is mapped

onto 2.

Whether a function is onto critically depends on its type signature. Suppose we define $p : \mathbb{Z} \to \mathbb{Z}$ by $p(x) = x + 2$. If we pick an output value y, then the input value $y - 2$ maps onto y. So the image of p is all of $\mathbb{Z}$. So this function is onto.

However, suppose we define $q : \mathbb{N} \to \mathbb{N}$ using the same formula $q(x) = x + 2$. q isn't onto, because none of the input values map onto 0 or 1.

7.5 Why are some functions not onto?

You may think many examples of non-onto functions look like they could have been onto if the author had set up the co-domain more precisely. Sometimes the co-domain is excessively large simply because the image set is awkward to specify succinctly. But, also, in some applications, we specifically want certain functions not to be onto.

For example, in graphics or certain engineering applications, we may wish to map out or draw a curve in 2D space. The whole point of the curve is that it occupies only part of 2D and it is surrounded by whitespace. These curves are often specified "parametrically," using functions that map into, but not onto, 2D.

For example, we can specify a (unit) circle as the image of a function $f : [0, 1] \to \mathbb{R}^2$ defined by $f(x) = (\cos 2\pi x, \sin 2\pi x)$. If you think of the input values as time, then f shows the track of a pen or robot as it goes around the circle. The cosine and sine are the x and y coordinates of its position. The 2π multiplier simply puts the input into the right range so that we'll sweep exactly once around the circle (assuming that sine and cosine take their inputs in radians).

7.6 Negating onto

To understand the concept of onto, it may help to think about what it means for a function **not** to be onto. This is our first example of negating a

statement involving two nested (different) quantifiers. Our definition of onto is

$$\forall y \in B, \exists x \in A, f(x) = y$$

So a function f is not onto if

$$\neg\forall y \in B, \exists x \in A, f(x) = y$$

To negate this, we proceed step-by-step, moving the negation inwards. You've seen all the identities involved, so this is largely a matter of being careful.

$$\neg\forall y \in B, \exists x \in A, f(x) = y$$
$$\equiv \exists y \in B, \neg\exists x \in A, f(x) = y$$
$$\equiv \exists y \in B, \forall x \in A, \neg(f(x) = y)$$
$$\equiv \exists y \in B, \forall x \in A, f(x) \neq y$$

So, if we want to show that f is not onto, we need to find some value y in B, such that no matter which element x you pick from A, $f(x)$ isn't equal to y.

7.7 Nested quantifiers

Notice that the definition of onto combines a universal and an existential quantifier, with the scope of one including the scope of the other. These are called *nested quantifiers*. When quantifiers are nested, the meaning of the statement depends on the order of the two quantifiers.

For example,

> For every person p in the Fleck family, there is a toothbrush t such that p brushes their teeth with t.

This sentence asks you to consider some random Fleck. Then, given that choice, it asserts that they have a toothbrush. The toothbrush is chosen after we've picked the person, so the choice of toothbrush can depend on the choice of person. This doesn't absolutely force everyone to pick their own toothbrush. (For a brief period, two of my sons were using the same one because they got confused.) However, at least this statement is consistent with each person having their own toothbrush.

Suppose now that we swap the order of the quantifiers, to get

> There is a toothbrush t, such that for every person p in the Fleck family, p brushes their teeth with t.

In this case, we're asked to choose a toothbrush t first. Then we're asserting that every Fleck uses this one fixed toothbrush t. Eeeuw! That wasn't what we wanted to say!

We do want the existential quantifier first when there's a single object that's shared among the various people, as in:

> There is a stove s, such that for every person p in the Fleck family, p cooks his food on s.

Notice that this order issue only appears when a statement a mixture of existential and universal quantifiers. If all the quantifiers are existential, or if all the quantifiers are universal, the order doesn't matter.

To take a more mathematical example, let's look back at modular arithmetic. Two numbers x and y are multiplicative inverses if $xy = yx = 1$. In the integers $\mathbb{Z}$, only 1 has a multiplicative inverse. However, in $\mathbb{Z}_k$, many other integers have inverses. For example, if $k = 7$, then $[3][5] = [1]$. So $[3]$ and $[5]$ are inverses.

For certain values of k every non-zero element of $\mathbb{Z}_k$ has an inverse.[1] You can verify that this is true for $\mathbb{Z}_7$: $[3]$ and $[5]$ are inverses, $[2]$ and $[4]$ are inverses, $[1]$ is its own inverse, and $[6]$ is its own inverse. So we can say that

[1] Can you figure out which values of k have this property?

$$\forall \text{ non-zero } x \in \mathbb{Z}_7,\ \exists y \in \mathbb{Z}_7,\ xy = yx = 1$$

Notice that we've put the universal quantifier outside the existential one, so that each number gets to pick its own inverse. Reversing the order of the quantifers would give us the following statement:

$$\exists y \in \mathbb{Z}_7,\ \forall \text{ non-zero } x \in \mathbb{Z}_7,\ xy = yx = 1$$

This version isn't true, because you can't pick one single number that works as an inverse for all the rest of the non-zero numbers, even in modular arithmetic.

However, we do want the existential quantifier first in the following claim, because $0y = y0 = 0$ for every $y \in \mathbb{Z}_7$.

$$\exists x \in \mathbb{Z}_7,\ \forall y \in \mathbb{Z}_7,\ xy = yx = x$$

7.8 Proving that a function is onto

Now, consider this claim:

Claim 30 *Define the function g from the integers to the integers by the formula $g(x) = x - 8$. g is onto.*

> Proof: We need to show that for every integer y, there is an integer x such that $g(x) = y$.
>
> So, let y be some arbitrary integer. Choose x to be $(y + 8)$. x is an integer, since it's the sum of two integers. But then $g(x) = (y + 8) - 8 = y$, so we've found the required pre-image for y and our proof is done.

For some functions, several input values map onto a single output value. In that case, we can choose any input value in our proof, typically whichever

is easiest for the proof-writer. For example, suppose we had $g : \mathbb{Z} \to \mathbb{Z}$ such that $g(x) = \lfloor \frac{x}{2} \rfloor$. To show that g is onto, we're given an output value x and need to find the corresponding input value. The simplest choice would be $2x$ itself. But you could also pick $2x + 1$.

Suppose we try to build such a proof for a function that isn't onto, e.g. $f : \mathbb{Z} \to \mathbb{Z}$ such that $f(x) = 3x + 2$.

> Proof: We need to show that for every integer y, there is an integer x such that $f(x) = y$.
>
> So, let y be some arbitrary integer. Choose x to be $\frac{(y-2)}{3}$. ...

If f was a function from the reals to the reals, we'd be ok at this point, because x would be a good pre-image for y. However, f's inputs are declared to be integers. For many values of y, $\frac{(y-2)}{3}$ isn't an integer. So it won't work as an input value for f.

7.9 A 2D example

Here's a sample function whose domain is 2D. Let $f : \mathbb{Z}^2 \to \mathbb{Z}$ be defined by $f(x, y) = x + y$. I claim that f is onto.

First, let's make sure we know how to read this definition. $f : \mathbb{Z}^2$ is shorthand for $\mathbb{Z} \times \mathbb{Z}$, which is the set of pairs of integers. So f maps a pair of integers to a single integer, which is the sum of the two coordinates.

To prove that f is onto, we need to pick some arbitrary element y in the co-domain. That is to say, y is an integer. Then we need to find a sample value in the domain that maps onto y, i.e. a "preimage" of y. At this point, it helps to fiddle around on our scratch paper, to come up with a suitable preimage. In this case, $(0, y)$ will work nicely. So our proof looks like:

> Proof: Let y be an element of $\mathbb{Z}$. Then $(0, y)$ is an element of $f : \mathbb{Z}^2$ and $f(0, y) = 0 + y = y$. Since this construction will work for any choice of y, we've shown that f is onto.

Notice that this function maps many input values onto each output value. So, in our proof, we could have used a different formula for finding the input value, e.g. $(1, y-1)$ or $(y, 0)$. A proof writer with a sense of humor might use $(342, y-342)$.

7.10 Composing two functions

Suppose that $f : A \to B$ and $g : B \to C$ are functions. Then $g \circ f$ is the function from A to C defined by $(g \circ f)(x) = g(f(x))$. Depending on the author, this is either called the composition of f and g or the composition of g and f. The idea is that you take input values from A, run them through f, and then run the result of that through g to get the final output value.

Take-home message: when using function composition, look at the author's shorthand notation rather than their mathematical English, to be clear on which function gets applied first.

In this definition, notice that g came first in $(g \circ f)(x)$ and g also comes first in $g(f(x))$. I.e. unlike $f(g(x))$ where f comes first. The trick for remembering this definition is to remember that f and g are in the same order on the two sides of the defining equation.

For example, suppose we define two functions f and g from the integers to the integers by:

$$f(x) = 3x + 7$$
$$g(x) = x - 8$$

Since the domains and co-domains for both functions are the integers, we can compose the two functions in both orders. But two composition orders give us different functions:

$$(f \circ g)(x) = f(g(x)) = 3g(x) + 7 = 3(x-8) + 7 = 3x - 24 + 7 = 3x - 17$$

$$(g \circ f)(x) = g(f(x)) = f(x) - 8 = (3x + 7) - 8 = 3x - 1$$

Frequently, the declared domains and co-domains of the two functions aren't all the same, so often you can only compose in one order. For example, consider the function $h : \{strings\} \to \mathbb{Z}$ which maps a string x onto its length in characters. (E.g. $h(\texttt{Margaret}) = 8$.) Then $f \circ h$ exists but $(h \circ f)$ doesn't exist because f produces numbers and the inputs to h are supposed to be strings.

7.11 A proof involving composition

Let's show that onto-ness works well with function composition. Specifically:

Claim 31 *For any sets A, B, and C and for any functions $f : A \to B$ and $g : B \to C$, if f and g are onto, then $g \circ f$ is also onto.*

> Proof: Let A, B, and C be sets. Let $f : A \to B$ and $g : B \to C$ be functions. Suppose that f and g are onto.
>
> We need to show that $g \circ f$ is onto. That is, we need to show that for any element x in C, there is an element y in A such that $(g \circ f)(y) = x$.
>
> So, pick some element x in C. Since g is onto, there is an element z in B such that $g(z) = x$. Since f is onto, there is an element y in A such that $f(y) = z$.
>
> Substituting the value $f(y) = z$ into the equation $g(z) = x$, we get $g(f(y)) = x$. That is, $(g \circ f)(y) = x$. So y is the element of A we needed to find.

7.12 Variation in terminology

A function is often known as a "map" and "surjective" is often used as a synonym for "onto." The image of a function is sometimes written $Im(f)$. The useful term "type signature" is not traditional in pure mathematics, though it's in wide use in computer science.

The term "range" is a term that has become out-dated in mathematics. Depending on the author, it may refer to either the image or the co-domain of a function, which creates confusion. Avoid using it.

Some authors write gf rather than $g \circ f$ for the composition of two functions. Also, some authors define $g \circ f$ such that $g \circ f(x) = f(g(x))$, i.e. the opposite convention from what we're using.

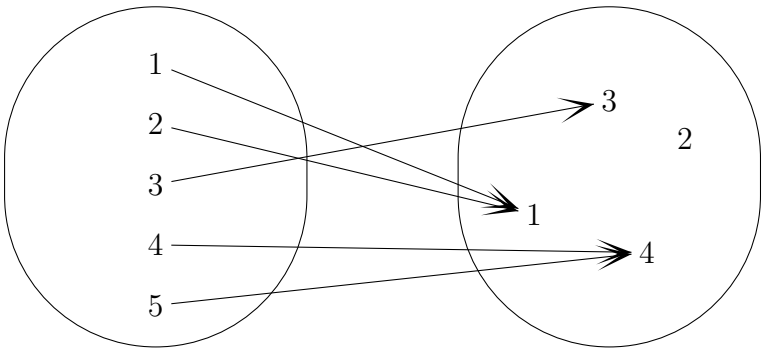
Chapter 8

Functions and one-to-one

In this chapter, we'll see what it means for a function to be one-to-one and bijective. This general topic includes counting permutations and comparing sizes of finite sets (e.g. the pigeonhole principle). We'll also see the method of adding stipulations to a proof "without loss of generality" as well as the technique of proving an equality via upper and lower bounds.

8.1 One-to-one

Suppose that $f : A \to B$ is a function from A to B. If we pick a value $y \in B$, then $x \in A$ is a *pre-image* of y if $f(x) = y$. Notice that I said **a** pre-image of y, not **the** pre-image of y, because y might have more than one preimage. For example, in the following function, 1 and 2 are pre-images of 1, 4 and 5 are pre-images of 4, 3 is a pre-image of 3, and 2 has no preimages.



A function is *one-to-one* if it never assigns two input values to the same output value. Or, said another way, no output value has more than one pre-image. So the above function isn't one-to-one, because (for example) 4 has more than one pre-image. If we define $g : \mathbb{Z} \to \mathbb{Z}$ such that $g(x) = 2x$. Then g is one-to-one.

As with onto, whether a function is one-to-one frequently depends on its type signature. For example, the absolute value function $|x|$ is not one-to-one as a function from the reals to the reals. However, it is one-to-one as a function from the natural numbers to the natural numbers.

One formal definition of one-to-one is:

$$\forall x, y \in A, x \neq y \to f(x) \neq f(y)$$

For most proofs, it's more convenient to use the contrapositive:

$$\forall x, y \in A, f(x) = f(y) \to x = y$$

When reading this definition, notice that when you set up two variables x and y, they don't have to have different (math jargon: "distinct") values. In normal English, if you give different names to two objects, the listener is expected to understand that they are different. By contrast, mathematicians always mean you to understand that they might be different but there's also the possibility that they might be the same object.

8.2 Bijections

If a function f is both one-to-one and onto, then each output value has exactly one pre-image. So we can invert f, to get an inverse function f^{-1}. A function that is both one-to-one and onto is called *bijective* or *a bijection*. If f maps from A to B, then f^{-1} maps from B to A.

Suppose that A and B are finite sets. Constructing an onto function from A to B is only possible when A has at least as many elements as B. Constructing a one-to-one function from A to B requires that B have at least as many values as A. So if there is a bijection between A and B, then the two sets must contain the same number of elements. As we'll see later, bijections are also used to define what it means for two infinite sets to be the "same size."

8.3 Pigeonhole Principle

Suppose that A contains more elements than B. Then it's impossible to construct a function A to B that is one-to-one, because two elements of A must be mapped to the same element of B. If we rephrase this in terms of putting labels on objects, we get the following Pigeonhole Principle:

> **Pigeonhole principle**: Suppose you have n objects and assign k labels to these objects. If $n > k$, then two objects must get the same label.

For example, if you have 8 playing cards and the cards come in five colors, then at least two of the cards share the same color. These elementary examples look more interesting when they involve larger numbers. For example, it would work poorly to assign three letter, case insensitive, usernames for the undergraduates at the University of Illinois. The Urbana-Champaign campus alone has over 30,000 undergraduates. But there are only $26^3 = 17,576$ 3-letter usernames. So the pigeonhole principle implies that there would be a pair of undergraduates assigned the same username.

The pigeonhole principle (like many superheros) has a dual identity. When you're told to apply it to some specific objects and labels, it's ob-

vious how to do so. However, it is often pulled out of nowhere as a clever trick in proofs, where you would have never suspected that it might be useful. Such proofs are easy to read, but sometimes hard to come up with.

For example, here's a fact that we can prove with a less obvious application of the pigeonhole principle.

Claim 32 *Among the first 100 powers of 17, there are two (distinct) powers which differ by a multiple of 57.*

The trick here is to notice that two numbers differ by a multiple of 57 exactly when they have the same remainder mod 57. But we have 100 distinct powers of 17 and only 57 different possible values for the remaider mod 57. So our proof might look like:

> Proof: The first 100 powers of 17 are $17^1, 17^1, \ldots, 17^{100}$. Consider their remainders mod 57: $r_1, r_2, \ldots, r_{100}$. Since there are only 57 possible remainders mod 57, two of the numbers $r_1, r_2, \ldots, r_{100}$ must be equal, by the pigeonhole principle. Let's suppose that r_j and r_k are equal. Then 17^j and 17_k have the same remainder mod 57, and so 17^j and 17_k differ by a multiple of 57.

The actual use of the pigeonhole principle almost seems like an afterthought, after we've come up with the right trick to structuring our analysis of the problem. This is typical of harder pigeonhole principle proofs.

8.4 Permutations

Now, suppose that $|A| = n = |B|$. We can construct a number of one-to-one functions from A to B. How many? Suppose that $A = \{x_1, x_2, \ldots, x_n\}$. We have n ways to choose the output value for x_1. But that choice uses up one output value, so we have only $n - 1$ choices for the output value of x_2. Continuing this pattern, we have $n(n-1)(n-2)\ldots 2 \cdot 1$ ($n!$ for short) ways to construct our function.

Similarly, suppose we have a group of n dragon figurines that we'd like to arrange on the mantlepiece. We need to construct a map from positions on

the mantlepiece to dragons. There are $n!$ ways to do this. An arrangement of n objects in order is called a permutation of the n objects.

More frequently, we need to select an ordered list of k objects from a larger set of n objects. For example, we have 30 dragon figurines but space for only 10 on our mantlepiece. Then we have 30 choices for the first figurine, 29 for the second, and so forth down to 21 choices for the last one. Thus we have $30 \cdot 29 \cdot \ldots 21$ ways to decorate the mantlepiece.

In general, an ordered choice of k objects from a set of n objects is known as a k-permutation of the n objects. There are $n(n-1)\ldots(n-k+1) = \frac{n!}{(n-k)!}$ different k-permutations of n objects. This number is called $P(n,k)$. $P(n,k)$ is also the number of one-to-one functions from a set of k objects to a set of n objects.

8.5 Further applications of permutations

Many real-world counting problems can be solved with the permutations formulas, but some problems require a bit of adaptation. For example, suppose that 7 adults and 3 kids need to get in line at the airport. Also suppose that we can't put two children next to each other, because they will fight.

The trick for this problem is to place the 7 adults in line, with gaps between them. Each gap might be left empty or filled with one kid. There are 8 gaps, into which we have to put the 3 kids. So, we have 7! ways to assign adults to positions. Then we have 8 gaps in which we can put kid A, 7 for kid B, and 6 for kid C. That is $7! \cdot 8 \cdot 7 \cdot 6$ ways to line them all up.

Now, let's suppose that we have a set of 7 scrabble tiles and we would like to figure out how many different letter strings we can make out of them. We could almost use the permutations formula, except that some of the tiles might contain the same letter. For example, suppose that the tiles are: C, O, L, L, E, G, E.

First, let's smear some dirt on the duplicate tiles, so we can tell the two copies apart: $C, O, L_1, L_2, E_1, G, E_2$. Then we would calculate 7! permutations of this list. However, in terms of our original problem, this is double-counting some possibilities, because we don't care about the differ-

ence between duplicates like L_1 and L_2. So we need to divide out by the number of ways we can permute the duplicates. In this case, we have 2! ways to permute the L's and 2! ways to permute the E's. So the true number of orderings of L is $\frac{7!}{2!2!}$.

Similarly, the number of reorderings of $J = (a, p, p, l, e, t, r, e, e, s)$ is $\frac{10!}{2!3!}$.

In general, suppose we have n objects, where n_1 are of type 1, n_2 are of type 2, and so forth through n_k are of type k. Then the number of ways to order our list of objects is $\frac{n!}{n_1!n_2!...n_k!}$.

8.6 Proving that a function is one-to-one

Now, let's move on to examples of how to prove that a specific function is one-to-one.

Claim 33 *Let $f : \mathbb{Z} \to \mathbb{Z}$ be defined by $f(x) = 3x + 7$. f is one-to-one.*

Let's prove this using our definition of one-to-one.

> Proof: We need to show that for every integers x and y, $f(x) = f(y) \to x = y$.
>
> So, let x and y be integers and suppose that $f(x) = f(y)$. We need to show that $x = y$.
>
> We know that $f(x) = f(y)$. So, substituting in our formula for f, $3x + 7 = 3y + 7$. So $3x = 3y$ and therefore $x = y$, by high school algebra. This is what we needed to show.

When we pick x and y at the start of the proof, notice that we haven't specified whether they are the same number or not. Mathematical convention leaves this vague, unlike normal English where the same statement would strongly suggest that they were different.

8.7 Composition and one-to-one

Like onto, one-to-one works well with function composition. Specifically:

Claim 34 *For any sets A, B, and C and for any functions $f : A \to B$ and $g : B \to C$, if f and g are one-to-one, then $g \circ f$ is also one-to-one.*

We can prove this with a direct proof, by being systematic about using our definitions and standard proof outlines. First, let's pick some representative objects of the right types and assume everything in our hypothesis.

> Proof: Let A, B, and C be sets. Let $f : A \to B$ and $g : B \to C$ be functions. Suppose that f and g are one-to-one.
>
> We need to show that $g \circ f$ is one-to-one.

To show that $g \circ f$ is one-to-one, we need to pick two elements x and y in its domain, assume that their output values are equal, and then show that x and y must themselves be equal. Let's splice this into our draft proof. Remember that the domain of $g \circ f$ is A and its co-domain is C.

> Proof: Let A, B, and C be sets. Let $f : A \to B$ and $g : B \to C$ be functions. Suppose that f and g are one-to-one.
>
> We need to show that $g \circ f$ is one-to-one. So, choose x and y in A and suppose that $(g \circ f)(x) = (g \circ f)(y)$
>
> We need to show that $x = y$.

Now, we need to apply the definition of function composition and the fact that f and g are each one-to-one:

> Proof: Let A, B, and C be sets. Let $f : A \to B$ and $g : B \to C$ be functions. Suppose that f and g are one-to-one.
>
> We need to show that $g \circ f$ is one-to-one. So, choose x and y in A and suppose that $(g \circ f)(x) = (g \circ f)(y)$
>
> Using the definition of function composition, we can rewrite this as $g(f(x)) = g(f(y))$. Combining this with the fact that g is one-to-one, we find that $f(x) = f(y)$. But, since f is one-to-one, this implies that $x = y$, which is what we needed to show.

8.8 Strictly increasing functions are one-to-one

Now, let's do a slightly trickier proof. First, a definition. Suppose that A and B are sets of real numbers (e.g. the reals, the rationals, the integers). A function $f : A \to B$ is *increasing* if, for every x and y in A, $x \leq y$ implies that $f(x) \leq f(y)$. f is called *strictly increasing* if, for every x and y in A, $x < y$ implies that $f(x) < f(y)$.[1] An increasing function can have plateaus where the output value stays constant, whereas a strictly increasing function must always increase.

Claim 35 *For any sets of real numbers A and B, if f is any strictly increasing function from A to B, then f is one-to-one.*

A similar fact applies to strictly decreasing functions.

To prove this, we will restate one-to-one using the alternative, contrapositive version of its definition.

$$\forall x, y \in A, x \neq y \to f(x) \neq f(y)$$

Normally, this wouldn't be a helpful move. The hypothesis involves a negative fact, whereas the other version of the definition has a positive hypothesis, normally a better place to start. But this is an atypical example and, in this case, the negative information turns out to be a good starting point.

> Proof: Let A and B be sets of numbers and let $f : A \to B$ be a strictly increasing function. Let x and y be distinct elements of A. We need to show that $f(x) \neq f(y)$.
>
> Since $x \neq y$, there's two possibilities.
>
> Case 1: $x < y$. Since f is strictly increasing, this implies that $f(x) < f(y)$. So $f(x) \neq f(y)$

[1] In math, "strictly" is often used to exclude the possibility of equality.

> Case 2: $y < x$. Since f is strictly increasing, this implies that $f(y) < f(x)$. So $f(x) \neq f(y)$.
>
> In either case, we have that $f(x) \neq f(y)$, which is what we needed to show.

The phrase "distinct elements of A" is math jargon for $x \neq y$.

When we got partway into the proof, we had the fact $x \neq y$ which isn't easy to work with. But the trichotomy axiom for real number states that for any x and y, we have exactly three possibilities: $x = y$, $x < y$, or $y < x$. The constraint that $x \neq y$ eliminates one of these possibilities.

8.9 Making this proof more succinct

In this example, the proofs for the two cases are very, very similar. So we can fold the two cases together. Here's one approach, which I don't recommend doing in the early part of this course but which will serve you well later on:

> Proof: Let A and B be sets of numbers and let $f : A \to B$ be a strictly increasing function. Let x and y be distinct elements of A. We need to show that $f(x) \neq f(y)$.
>
> Since $x \neq y$, there's two possibilities.
>
> Case 1: $x < y$. Since f is strictly increasing, this implies that $f(x) < f(y)$. So $f(x) \neq f(y)$
>
> Case 2: $y < x$. Similar to case 1.
>
> In either case, we have that $f(x) \neq f(y)$, which is what we needed to show.

This method only works if you, and your reader, both agree that it's obvious that the two cases are very similar and the proof will really be similar. Dangerous assumption right now. And we've only saved a small amount of writing, which isn't worth the risk of losing points if the grader doesn't think it was obvious.

But this simplification can be very useful in more complicated situations where you have may have lots of cases, the proof for each case is long, and the proofs for different cases really are very similar.

Here's another way to simplify our proof:

> Proof: Let A and B be sets of numbers and let $f : A \to B$ be a strictly increasing function. Let x and y be distinct elements of A. We need to show that $f(x) \neq f(y)$.
>
> We know that $x \neq y$, so either $x < y$ or $y < x$. Without loss of generality, assume that $x < y$.
>
> Since f is strictly increasing, $x < y$ implies that $f(x) < f(y)$. So $f(x) \neq f(y)$, which is what we needed to show.

The phrase "without loss of generality" means that we are adding an additional assumption to our proof but we claim it isn't really adding any more information. In this case, s and t are both arbitrary indices introduced at the same time. It doesn't really matter which of them is called s and which is called t. So if we happen to have chosen our names in an inconvenient order, we could simply swap the two names.

Simplifying a problem "without loss of generality" is a powerful but potentially dangerous proof technique. You'll probably probably want to see a few times before you use it yourself. It's actually used fairly often to simplify proofs, so it has an abbreviation: WOLOG or WLOG.

8.10 Variation in terminology

The term "injective" is a synonym for one-to-one and one-to-one correspondence is a synonym for bijective. The phrase "by symmetry" is often used in place of "without loss of generality."

"Monotonically increasing" is a synonym for "increasing." The terms "non-decreasing" and "weakly increasing" are also synonyms for "increasing," which are often used when the author worries that the reader might forget that "increasing" allows plateaus.

Chapter 9

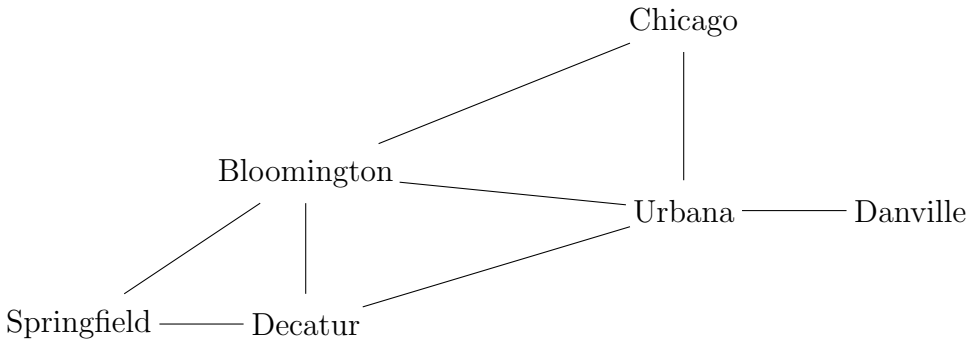
Graphs

Graphs are a very general class of object, used to formalize a wide variety of practical problems in computer science. In this chapter, we'll see the basics of (finite) undirected graphs, including graph isomorphism and connectivity.

9.1 Graphs

A graph consists of a set of nodes V and a set of edges E. We'll sometimes refer to the graph as a pair of sets (V, E). Each edge in E joins two nodes in V. Two nodes connected by an edge are called **neighbors** or **adjacent**

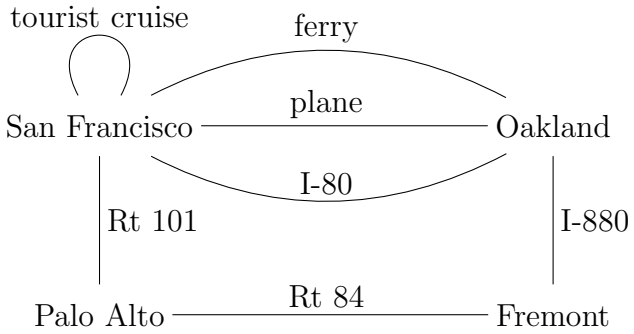
For example, here is a graph in which the nodes are Illinois cities and the edges are roads joining them:



A graph edge can be traversed in both directions, as in this street example, i.e. the edges are **undirected**. When discussing relations earlier, we used **directed graphs**, in which each edge had a specific direction. Unless we explicitly state otherwise, a "graph" will always be undirected. Concepts for undirected graphs extend in straightforward ways to directed graphs.

When there is only one edge connecting two nodes x and y, we can name the edge using the pair of nodes. We could call the edge xy or (since order doesn't matter) yx or $\{x, y\}$. So, in the graph above, the Urbana-Danville edge connects the node Urbana and the node Danville.

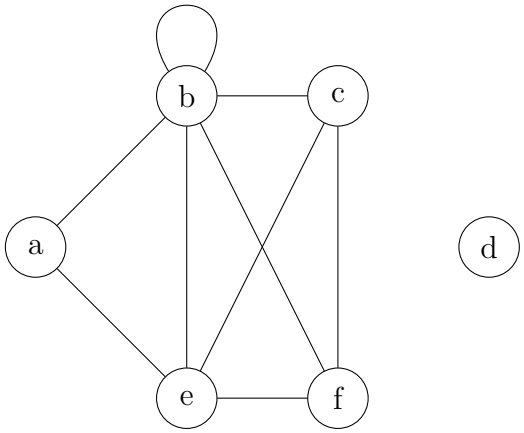
In some applications, we need graphs in which two nodes are connected by **multiple edges**, i.e. parallel edges with the same endpoints. For example, the following graph shows ways to travel among four cities in the San Francisco Bay Area. It has three edges from San Francisco to Oakland, representing different modes of transportation. When multiple edges are present, we typically label the edges rather than trying to name edges by their endpoints. This diagram also illustrates a **loop** edge which connects a node to itself.



A graph is called a **simple graph** if it has neither multiple edges nor loop edges. Unless we explicitly state otherwise, a "graph" will always be a simple graph. Also, we'll assume that it has at least one node and that it has only a finite number of edges and nodes. Again, most concepts extend in a reasonable way to infinite and non-simple graphs.

9.2 Degrees

The degree of a node v, written $deg(v)$ is the number of edges which have v as an endpoint. Self-loops, if you are allowing them, count twice. For example, in the following graph, a has degree 2, b has degree 6, d has degree 0, and so forth.



Each edge contributes to two node degrees. So the sum of the degrees of all the nodes is twice the number of edges. This is called the Handshaking Theorem and can be written as

$$\sum_{v \in V} deg(v) = 2|E|$$

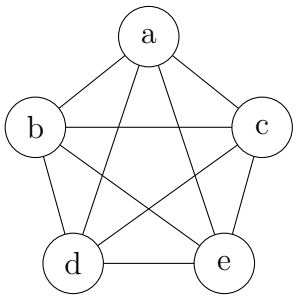
This is a slightly different version of summation notation. We pick each node v in the set V, get its degree, and add its value into the sum. Since V is finite, we could also have given names to the nodes $v_1, \ldots, v_n$ and then written

$$\sum_{k=1}^{n} v_k \in V deg(v) = 2|E|$$

The advantage to the first, set-based, style is that it generalizes well to situations involving infinite sets.

9.3 Complete graphs

Several special types of graphs are useful as examples. First, the complete graph on n nodes (shorthand name K_n), is a graph with n nodes in which every node is connected to every other node. K_5 is shown below.



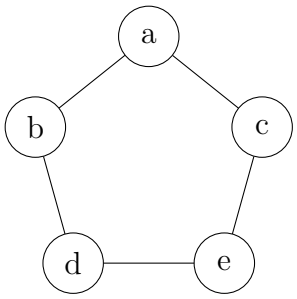
To calculate the number of edges in K_n, think about the situation from the perspective of the first node. It is connected to $n-1$ other nodes. If we look at the second node, it adds $n-2$ more connections. And so forth. So we have $\sum_{k=1}^{n}(n-k) = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$ edges.

9.4 Cycle graphs and wheels

Suppose that we have n nodes named $v_1, \ldots, v_n$, where $n \geq 3$. Then the cycle graph C_n is the graph with these nodes and edges connecting v_i to v_{i+1}, plus an additional edge from v_n to v_1. That is, the set of edges is:

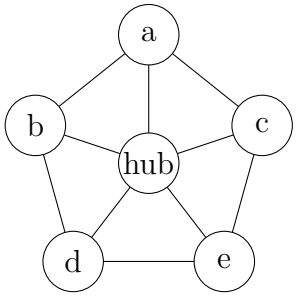
$$E = \{v_1v_2,\ v_2v_3,\ \ldots,\ v_{n-1}v_n,\ v_nv_1\}$$

So C_5 looks like



C_n has n nodes and also n edges. Cycle graphs often occur in networking applications. They could also be used to model games like "telephone" where people sit in a circle and communicate only with their neighbors.

The wheel W_n is just like the cycle graph C_n except that it has an additional central "hub" node which is connected to all the others. Notice that W_n has $n + 1$ nodes (not n nodes). It has $2n$ edges. For example, W_5 looks like



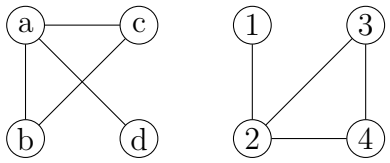
9.5 Isomorphism

In graph theory, we only care about how nodes and edges are connected together. We don't care about how they are arranged on the page or in space, how the nodes and edges are named, and whether the edges are drawn as straight or curvy. We would like to treat graphs as interchangeable if they have the same abstract connectivity structure.

Specifically, suppose that $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are graphs. An **isomorphism** from G_1 to G_2 is a bijection $f : V_1 \to V_2$ such that nodes a and b are joined by an edge if and only if $f(a)$ and $f(b)$ are joined by an

edge. The graphs G_1 and G_2 are **isomorphic** if there is an isomorphism from G_1 to G_2.

For example, the following two graphs are isomorphic. We can prove this by defining the function f so that it maps 1 to d, 2 to a, 3 to c, and 4 to b. The reader can then verify that edges exist in the left graph if and only if the corresponding edges exist in the right graph.

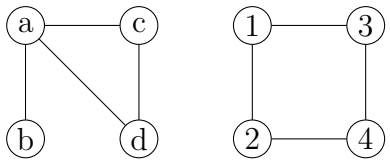


Graph isomorphism is another example of an equivalence relation. Each equivalence class contains a group of graphs which are superficially different (e.g. different names for the nodes, drawn differently on the page) but all represent the same underlying abstract graph.

To prove that two graphs are not isomorphic, we could walk through all possible functions mapping the nodes of one to the nodes of the other. However, that's a huge number of functions for graphs of any interesting size. An exponential number, in fact. Instead, a better technique for many examples is to notice that a number of graph properties are "invariant," i.e. preserved by isomorphism.

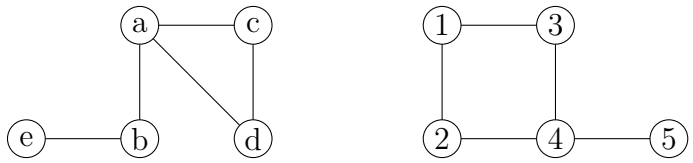
- The two graphs must have the same number of nodes and the same number of edges.
- For any node degree k, the two graphs must have the same number of nodes of degree k. For example, they must have the same number of nodes with degree 3.

We can prove that two graphs are not isomorphic by giving one example of a property that is supposed to be invariant but, in fact, differs between the two graphs. For example, in the following picture, the lefthand graph has a node of degree 3, but the righthand graph has no nodes of degree 3, so they can't be isomorphic.



9.6 Subgraphs

It's not hard to find a pair of graphs that aren't isomorphic but where the most obvious properties (e.g. node degrees) match. To prove that such a pair isn't isomorphic, it's often helpful to focus on certain specific local features of one graph that aren't present in the other graph. For example, the following two graphs have the same node degrees: one node of degree 1, three of degree 2, one of degree 3. However, a little experimentation suggests they aren't isomorphic.



To make a convincing argument that these graphs aren't isomorphic, we need to define the notion of a **subgraph**. If G and G' are graphs, then G' is a subgraph of G if and only if the nodes of G' are a subset of the nodes of G and the edges of G' are a subset of the edges of G. If two graphs G and F are isomorphic, then any subgraph of G must have a matching subgraph somewhere in F.

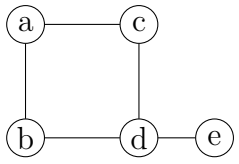
A graph has a huge number of subgraphs. However, we can usually find evidence of non-isomorphism by looking at small subgraphs. For example, in the graphs above, the lefthand graph has C_3 as a subgraph, but the righthand graph does not. So they can't be isomorphic.

9.7 Walks, paths, and cycles

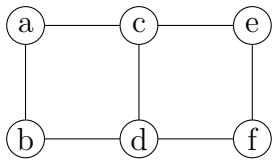
In a graph G, a walk of length k from node a to node b is a finite sequence of nodes $a = v_1, v_2, \ldots, v_n = b$ and a finite sequence of edges $e_1, e_2, \ldots, e_{n-1}$ in which e_i connects v_i and v_{i+1}, for all i. Under most circumstances, it isn't necessary to give both the sequence of nodes and the sequence of edges: one of the two is usually sufficient. The length of a walk is the number of edges in it. The shortest walks consist of just a single node and have length zero.

A walk is **closed** if its starting and ending nodes are the same. Otherwise it is **open**. A **path** is a walk in which no node is used more than once. A **cycle** is a closed walk with at least three nodes in which no node is used more than once except that the starting and ending nodes are the same. The apparently ad-hoc requirement that a cycle contain at least three nodes is important, because it forces a cycle to form a ring that can (for example) enclose a region of 2D.

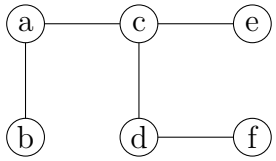
For example, in the following graph, there is a length-3 walk from a to e: ac, cd, de. Another walk of length 3 would have edges: ab, bd, de. These two walks are also paths. There are also longer walks from a to e, which aren't paths because they re-use nodes, e.g. the walk with a node sequence a, c, d, b, d, e. If you have a walk between two nodes, you can always creat a path between the nodes by pruning unnecessary loops from the walk.



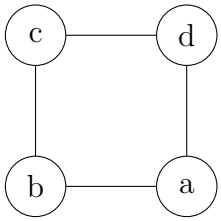
In the following graph, one cycle of length 4 has edges: ab, bd, dc, ca. Other closely-related cycles go through the same nodes but with a different starting point or in the opposite direction, e.g. dc, bd, ab, ca. (Remember that cd and dc are two names for the same edge.) Unlike cycles, closed walks can re-use nodes, e.g. ab, ba, ac, ce, ec, ca is a closed walk but not a cycle.



The following graph is **acyclic**, i.e. it doesn't contain any cycles.

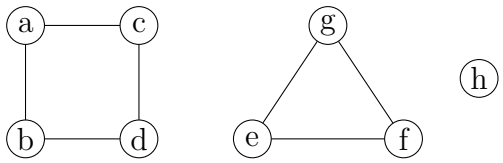


Notice that the cycle graph C_n contains $2n$ different cycles. For example, if the vertices of C_4 are labelled as shown below, then one cycle is ab, bc, cd, da, another is cd, bc, ab, da, and so forth.



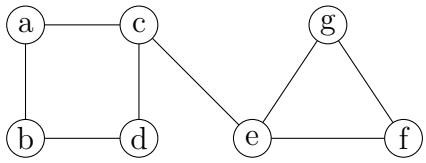
9.8 Connectivity

A graph G is **connected** if there is a walk between every pair of nodes in G. Our previous examples of graphs were connected. The following graph is not connected, because there is no walk from (for example), a to g.



If we have a graph G that might or might not be connected, we can divide G into **connected components.** Each connected component contains a maximal (i.e. biggest possible) set of nodes that are all connected to one another, plus all their edges. So, the above graph has three connected components: one containing nodes a,b,c, and d, a second containing nodes e, f, and g, and a third that contains only the node h.

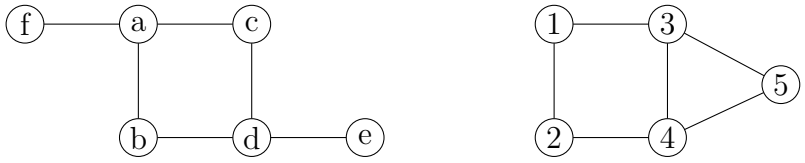
Sometimes two parts of a graph are connected by only a single edge, so that the graph would become disconnected if that edge were removed. This is called a **cut edge.** For example, in the following graph, the edge *ce* is a cut edge. In some applications, cut edges are a problem. E.g. in networking, they are places where the network is vulnerable. In other applications (e.g. compilers), they represent opportunities to divide a larger problem into several simpler ones.



9.9 Distances

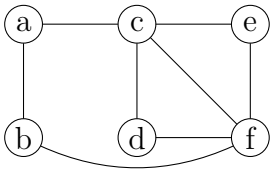
In graphs, distances are based on the lengths of paths connecting pairs of nodes. Specifically, the distance $d(a, b)$ between two nodes a and b is the

length of the shortest path from a to b. The diameter of a graph is the maximum distance between any pair of nodes in the graph. For example, the lefthand graph below has diameter 4, because $d(f, e) = 4$ and no other pair of nodes is further apart. The righthand graph has diameter 2, because $d(1, 5) = 2$ and no other pair of nodes is further apart.



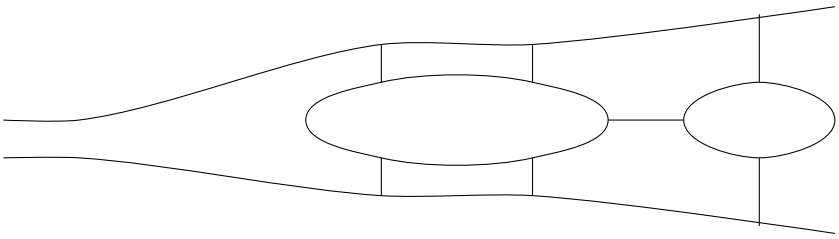
9.10 Euler circuits

An Euler circuit of a graph G is a closed walk that uses each edge of the graph exactly once. Notice that "G has an Euler circuit" means that every edge of G is in the circuit; it's not enough for some subgraph of G to have a suitable circuit. For example, one Euler circuit of the following graph would be ac, cd, df, fe, ec, cf, fb, ba.



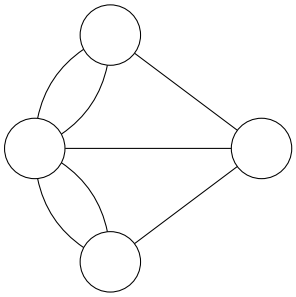
An Euler circuit is possible exactly when the graph is connected and each node has even degree. Each node has to have even degree because, in order to complete the circuit, you have to leave each node that you enter. If the node has odd degree, you will eventually enter a node but have no unused edge to go out on.

Fascination with Euler circuits dates back to the 18th century. At that time, the city of Königberg, in Prussia, had a set of bridges that looked roughly as follows:



Folks in the town wondered whether it was possible to take a walk in which you crossed each bridge exactly once, coming back to the same place you started. This is the kind of thing that starts long debates late at night in pubs, or keeps people amused during boring church services. Leonard Euler was the one who explained clearly why this isn't possible.

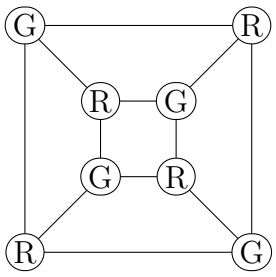
For our specific example, the corresponding graph looks as follows. Since all of the nodes have odd degree, there's no possibility of an Euler circuit.



9.11 Bipartite graphs

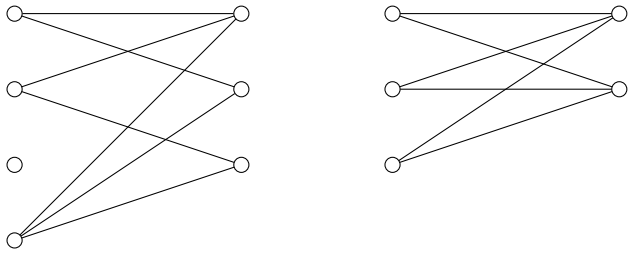
Another special type of graph is a bipartite graph. A graph $G = (V, E)$ is **bipartite** if we can split V into two non-overlapping subsets V_1 and V_2 such that every edge in G connects an element of V_1 with an element of V_2. That is, no edge connects two nodes from the same part of the division.

For example, the cube graph is bipartite. If we assign nodes to the R and G groups as shown below, then there are no edges connecting an R node to an R node, or a G node to a G node.



Bipartite graphs often appear in matching problems, where the two subsets represent different types of objects. For example, one group of nodes might be students, the other group of nodes might be workstudy jobs, and the edges might indicate which jobs each student is interested in.

The complete bipartite graph $K_{m,n}$ is a bipartite graph with m nodes in V_1, n nodes in V_2, and which contains all possible edges that are consistent with the definition of bipartite. The diagram below shows a partial bipartite graph on a set of 7 nodes, as well as the complete bipartite graph $K_{3,2}$.



The complete bipartite graph $K_{m,n}$ has $m + n$ nodes and mn edges.

9.12 Variation in terminology

Although the core ideas of graph theory are quite stable, terminology varies a lot and there is a huge range of specialized terminology for specific types of graphs. In particular, nodes are often called "vertices" Notice that the singular of this term is "vertex" (not "vertice"). A complete graph on some set of vertices is also known as a clique.

What we're calling a "walk" used to be widely called a "path." Authors who still use this convention would then use the term "simple path" to exclude repetition of vertices. Terms for closely-related concepts, e.g. cycle, often change as well.

Authors vary as to whether the wheel graph W_n has n or $n+1$ nodes.

Chapter 10

2-way Bounding

In high school and early college mathematics, we are often proving equalities and we often prove them by manipulating a sequence of equalities. For example

$$\frac{3x^2 - 5x - 2}{x - 2} = \frac{(x - 2)(3x + 1)}{x - 2} = 3x + 1$$

On more complex problems, it's often necessary to adopt a more flexible approach: bound the quantity of interest from both directions. When our best upper and lower bounds are equal, we've established an equality. Otherwise, we've limited the quantity to a known (hopefully small) range.

Because we need to establish a lower bound and also an upper bound, a 2-way bounding proof contains two sub-proofs. Sometimes the two sub-proofs are similar. However, the true power of the technique comes from the fact that the two sub-proofs can use entirely different techniques. Therefore, the method is widely used when proving more difficult results, e.g. in upper-level computer science and mathematics courses (e.g. real analysis, algorithms).

The most common error building these proofs is to omit one half entirely. For example, someone intending to show that $f(x) = k$ might prove that $f(x) \leq k$ but forget to also prove that $f(x) \geq k$. Do not do this. If one of the bounds is obvious, say that explicitly. In this chapter, we will see how this method works on a variety of examples.

10.1 Marker Making

Suppose that we have a 12" by 15" sheet of cookie dough, from which we'd like to cut maple-leaf cookies. Scraps of dough that have been reclaimed and re-rolled never produce results as nice as those from the original sheet, so we'd like to pack as many cookies into our original sheet as possible. How many cookies can we cut from this sheet?

We could experiment with cutting several sheets of dough, placing our cookie cutters in various different ways. Suppose we managed to fit in 25 cookies on our best attempt. We now know that 25 is a lower bound on the maximum number of cookies we can cut out. But it might be very hard to prove we've found the best layout.

Now, suppose that we put our maple-leaf cutters onto graph paper and work out that the area of each cookie is at least 6 square inches. Since our sheet of dough has area 180 square inches, we know that it's impossible to cut out more than 30 cookies. So we now have an upper bound of 30 on the number of cookies in the optimal layout. In this example, we would normally expect our lower bound and our upper bound to be some distance apart, because it's so difficult to search through all the ways to arrange our cutters.

Home kitchens do not need to worry heavily about efficiency, but industrial plants do. In particular, clothing manufacturers need to create **markers** for cutting clothing parts out of rectangles of cloth. Because the appearance and material properties of cloth are tied to its warp direction, parts must be lined up in specific orientations. Scrap cloth, though recyclable for other purposes, cannot simply be reformed and re-used. Efficient markers are important, because any waste will be replicated many times, but difficult for humans to create. So considerable effort has gone into the development of efficient marker making algorithms.

In both engineering and in theoretical computer science, it is common to encounter quantities that cannot be calculated directly but, rather, must be bounded from above and below. Sometimes we can make the upper and lower bounds meet, giving us a so-called **tight** bound. Often, we can't.

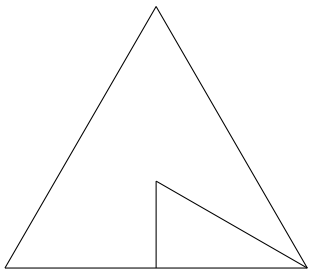
10.2 Pigeonhole point placement

Here's a problem of a very different sort, which uses 2-way bounding toegether with the pigeonhole principle. Like so many pigeonhole proofs, the proof depends on a non-obvious trick. In this case, it's a trick about dividing up a triangle. Once you've seen the trick, however, the proof is a classic example of bounding a quantity from above and below.

Claim 36 *Suppose that T is an equilateral triangle with sides of length 2 units We can place a maximum of four points in the triangle such that every pair of points are more than 1 unit apart.*

To show that 4 is the maximum number of points we can place with the required separations, we need to show that it's possible to place 4 points, but it is not possible to place 5 points. It's easiest to tackle these two subproblems separately, using different techniques.

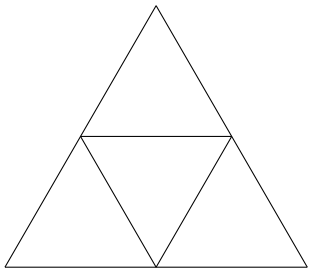
> Proof: To show that the maximum is at least four, notice that we can place three points at the corners of the triangle and one point in the center. The points at the corners are two units apart. To see that the point in the center is more than one unit from any corner, notice that the center, the corner, and the midpoint of the side form a right triangle.



1

> The hypotenuse of this triangle connects the center point to the corner point. Since one leg of the triangle has length 1, the hypotenuse must have length greater than 1.

To show that the maximum number of points cannot be greater than four, divide up the triangle into four smaller equilateral triangles as follows:

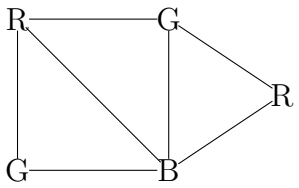


Suppose we tried to place five or more points into the big triangle. Since there are only four small triangles, by the pigeonhole principle, some small triangle would have to contain at least two points. But since the small triangle has side length only 1, these points can't be separated by more than one unit.

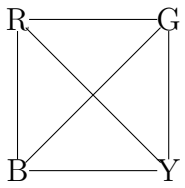
10.3 Graph coloring

A *coloring* of a graph G assigns a color to each node of G, with the restriction that two adjacent nodes never have the same color. If G can be colored with k colors, we say that G is k-colorable. The *chromatic number* of G, written $\chi(G)$, is the smallest number of colors needed to color G.

For example, only three colors are required for this graph:



But the complete graph K_n requires n colors, because each node is adjacent to all the other nodes. E.g. K_4 can be colored as follows:



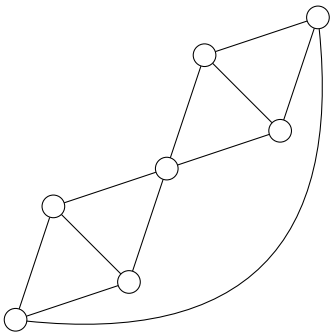
To establish that n is the chromatic number for a graph G, we need to establish two facts:

- $\chi(G) \leq n$: G can be colored with n colors.
- $\chi(G) \geq n$: G cannot be colored with less than n colors

For small finite graphs, the simplest way to show that $\chi(G) \leq n$ is to show a coloring of G that uses n colors. For a larger class of graphs, we could describe an algorithm for doing the coloring. For example, we can color a cyclic graph with an even number of nodes by alternating colors around the circle. A bipartite graph never requires more than two colors.

Showing that $\chi(G) \geq n$ can sometimes be equally straightforward. For example, if the graph has any edges at all, its chromatic number must be at least 2. If G contains a copy of K_n, the chromatic number of G must be at least n because K_n can't be colored with less than n colors.

However, the following example shows why this process can get tricky. It's relatively easy to color with 4 colors. But the largest complete graph in it is K_3, which gives us a lower bound of only 3. Showing that the chromatic number is actually 4 requires carefully stepping through all possible ways to assign three colors to the nodes and explaining why none can end with a complete coloring.



10.4 Why care about graph coloring?

Graph coloring is required for solving a wide range of practical problems. For example, there is a coloring algorithm embedded in most compilers. Because the general problem can't be solved efficiently, the implemented algorithms use limitations or approximations of various sorts so that they can run in a reasonable amount of time.

For example, catalog and on-line retailers frequently organize their clothes offerings into collections, e.g. "Golfing Geezer" or "Tough but Toucheable" or "Uniforms 4 U." All items in a collection are guaranteed to be compatible. This helps retailers market clothes to folks with no dress sense, as well as cope with the fact that people are sensitive to fine color distinctions that aren't reproduced accurately in catalogs and on computer screens. How many collections are required to organize the retailer's inventory of clothes?

We can model this problem as graph coloring. Each graph node is an item of clothing. Edges join pairs of items that aren't compatible, e.g. the bright green pants don't go with the grey-orange shirt, the business formal blouse doesn't go with the golfing pants. The "color" on each node is the name of a collection. If we find that too many collections are required, we might want to remove selected items of clothing (e.g. the bottle green bell bottoms that match nothing) from our inventory.

We can model a sudoku puzzle by setting up one node for each square.

The colors are the 9 numbers, and some are pre-assigned to certain nodes. Two nodes are connected if their squares are in the same block or row or column. The puzzle is solvable if we can 9-color this graph, respecting the pre-assigned colors.

We can model exam scheduling as a coloring problem. The exams for two courses should not be put at the same time if there is a student who is in both courses. So we can model this as a graph, in which each course is a node and courses are connected by edges if they share students. The question is then whether we can color the graph with k colors, where k is the number of exam times in our schedule.

In the exam scheduling problem, we actually expect the answer to be "no," because eliminating conflicts would require an excessive number of exam times. So the real practical problem is: how few students do we have to take out of the picture (i.e. give special conflict exams to) in order to be able to solve the coloring problem with a reasonable value for k. We also have the option of splitting a course (i.e. offering a scheduled conflict exam) to simplify the graph.

A particularly important use of coloring in computer science is register allocation. A large java or C program contains many named variables. But a computer has a smallish number (e.g. 32) of fast registers which can feed basic operations such as addition. So variables must be allocated to specific registers.

The nodes in this coloring problem are variables. The colors are registers. Two variables are connected by an edge if they are in use at the same time and, therefore, cannot share a register. As with the exam scheduling problem, we actually expect the raw coloring problem to fail. The compiler then uses so-called "spill" operations to break up the dependencies and create a graph we can color with our limited number of registers. The goal is to use as few spills as possible.

10.5 Proving set equality

Finally, 2-way bounding proofs are often used to prove that two sets A and B are equal. That is, we show that $A \subseteq B$ and $B \subseteq A$, using separate

subproofs. We can then conclude that $A = B$. This looks superficially different from the previous examples, because the relation is $\subseteq$ rather than $\leq$. But the main idea is the same: we first show that A is no larger than B, and then show that A is no smaller than B.

As an example, let's look at

Claim 37 *Let* $A = \{15p + 9q \mid p, q \in \mathbb{Z}\}$ *Then* $A = \{$*multiples of 3*$\}$.

Before you try to prove this, first put some sample integer values into the formula $15p + 9q$. See if you can convince yourself informally that this formula really generates all and only multiples of 3.

Using the bounding method, we would write the proof as follows:

> Proof:
>
> (1) Show that $A \subseteq \{$multiples of 3$\}$.
>
> Let x be an element of A. By the definition of A, $x = 15s + 9t$, for some integers s and t. But then $x = 3(5s + 3t)$. $4s + 3t$ is an integer, since s and t are integers. So x is a multiple of 3.
>
> (2) Show that $\{$multiples of 3$\} \subseteq A$.
>
> Notice that (*) $15 \cdot (-1) + 9 \cdot 2 = 3$.
>
> Let x be a multiple of 3. Then $x = 3n$ for some integer n. Substituting (*) into this equation, we get $x = (15 \cdot (-1) + 9 \cdot 2)n$. So $x = 15 \cdot (-n) + 9 \cdot (2n)$. So x is an element of A.
>
> Since we've shown that $A \subseteq \{$multiples of 3$\}$ and $\{$multiples of 3$\} \subseteq A$, we can conclude that $A = \{$multiples of 3$\}$.

10.6 Variation in terminology

When coloring graphs, we have placed colors on vertices. A closely-related set of problems involve placing colors on edges. The terms "vertex coloring" and "edge coloring" are used when it's necessary to distinguish the two.

Chapter 11

Induction

This chapter covers mathematical induction.

11.1 Introduction to induction

At the start of the term, we saw the following formula for computing the sum of the first n integers:

Claim 38 *For any positive integer n, $\Sigma_{i=1}^{n}\ i = \frac{n(n+1)}{2}$.*

At that point, we didn't prove this formula correct, because this is most easily done using a new proof technique: induction.

Mathematical induction is a technique for showing that a statement $P(n)$ is true for all natural numbers n, or for some infinite subset of the natural numbers (e.g. all positive even integers). It's a nice way to produce quick, easy-to-read proofs for a variety of fact that would be awkward to prove with the techniques you've seen so far. It is particularly well suited to analyzing the performance of recursive algorithms. Most of you have seen a few of these in previous programming classes; you'll see many more in later classes.

Induction is very handy, but it may strike you as a bit weird. It may take you some time to get used to it. In fact, you have two tasks which are somewhat independent:

- Learn how to write an inductive proof.
- Understand why inductive proofs are legitimate.

You can learn to write correct inductive proofs even if you remain somewhat unsure of why the method is legitimate. Over the next few classes, you'll gain confidence in the validity of induction and its friend recursion.

11.2 An Example

A proof by induction has the following outline:

> Claim: $P(n)$ is true for all positive integers n.
>
> Proof: We'll use induction on n.
>
> **Base**: We need to show that $P(1)$ is true.
>
> **Induction**: Suppose that $P(n)$ is true for $n = 1, 2, \ldots, k-1$. We need to show that $P(k)$ is true.

The part of the proof labelled "induction" is a conditional statement. We assume that $P(n)$ is true for values of n no larger than $k-1$. This assumption is called the *inductive hypothesis*. We use this assumption to show that $P(k)$ is true.

For our formula example, our proposition $P(n)$ is $\Sigma_{i=1}^{n}\ i = \frac{n(n+1)}{2}$. Substituting this definition of P into the outline, we get the following outline for our specific claim:

> Proof: We will show that $\Sigma_{i=1}^{n}\ i = \frac{n(n+1)}{2}$ for any positive integer n, using induction on n.
>
> **Base**: We need to show that the formula holds for $n = 1$, i.e. $\Sigma_{i=1}^{1}\ i = \frac{1 \cdot 2}{2}$.
>
> **Induction**: Suppose that $\Sigma_{i=1}^{n}\ i = \frac{n(n+1)}{2}$ for $n = 1, 2, \ldots, k-1$. We need to show that $\Sigma_{i=1}^{k}\ i = \frac{k(k+1)}{2}$.

The full proof might then look like:

> Proof: We will show that $\Sigma_{i=1}^{n}\ i = \frac{n(n+1)}{2}$ for any positive integer n, using induction on n.
>
> **Base**: We need to show that the formula holds for $n = 1$. $\Sigma_{i=1}^{1}\ i = 1$. And also $\frac{1 \cdot 2}{2} = 1$. So the two are equal for $n = 1$.
>
> **Induction**: Suppose that $\Sigma_{i=1}^{n}\ i = \frac{n(n+1)}{2}$ for $n = 1, 2, \ldots, k-1$. We need to show that $\Sigma_{i=1}^{k}\ i = \frac{k(k+1)}{2}$.
>
> By the definition of summation notation, $\Sigma_{i=1}^{k}\ i = (\Sigma_{i=1}^{k-1}\ i) + k$
>
> Our inductive hypothesis states that at $n = k-1$, $\Sigma_{i=1}^{k-1}\ i = (\frac{(k-1)k}{2})$.
>
> Combining these two formulas, we get that $\Sigma_{i=1}^{k}\ i = (\frac{(k-1)k}{2}) + k$.
>
> But $(\frac{(k-1)k}{2}) + k = (\frac{(k-1)k}{2}) + \frac{2k}{2} = (\frac{(k-1+2)k}{2}) = \frac{k(k+1)}{2}$.
>
> So, combining these equations, we get that $\Sigma_{i=1}^{k}\ i = \frac{k(k+1)}{2}$ which is what we needed to show.

One way to think of a proof by induction is that it's a template for building direct proofs. If I give you the specific value $n = 47$, you could write a direct proof by starting with the base case and using the inductive step 46 times to work your way from the $n = 1$ case up to the $n = 47$ case.

11.3 Why is this legit?

There are several ways to think about mathematical induction, and understand why it's a legitimate proof technique. Different people prefer different motivations at this point, so I'll offer several.

> **Domino Theory:** Imagine an infinite line of dominoes. The base step pushes the first one over. The inductive step claims that one domino falling down will push over the next domino in the line. So dominos will start to fall from the beginning all the way down the line. This process continues forever, because

the line is infinitely long. However, if you focus on any specific domino, it falls after some specific finite delay.

Recursion fairy: The recursion fairy is the mathematician's version of a programming assistant. Suppose you tell her how to do the proof for $P(1)$ and also why $P(1)$ up through $P(k)$ implies $P(k+1)$. Then suppose you pick any integer (e.g. 1034) then she can take this recipe and use it to fill in all the details of a normal direct proof that P holds for this particular integer. That is, she takes $P(1)$, then uses the inductive step to get from $P(1)$ to $P(2)$, and so on up to $P(1034)$.

Defining property of the integers: The integers are set up mathematically so that induction will work. Some formal sets of axioms defining the integers include a rule saying that induction works. Other axiom sets include the "well-ordering" property: any subset that has a lower bound also has a smallest element. This is equivalent to an axiom that explicitly states that induction works. Both of these axioms prevent the integers from having extra very large elements that can't be reached by repeatedly adding one to some starting integer. So, for example, ∞ is not an integer.

These arguments don't depend on whether our starting point is 1 or some other integer, e.g. 0 or 2 or -47. All you need to do is ensure that your base case covers the first integer for which the claim is supposed to be true.

11.4 Building an inductive proof

In constructing an inductive proof, you've got two tasks. First, you need to set up this outline for your problem. This includes identifying a suitable proposition P and a suitable integer variable n.

Notice that $P(n)$ must be a statement, i.e. something that is either true or false. For example, it is **never** just a formula whose value is a number. Also, notice that $P(n)$ must depend on an integer n. This integer n is known

as our *induction variable*. The assumption at the start of the inductive step ("$P(k)$ is true") is called the inductive hypothesis.

Your second task is to fill in the middle part of the induction step. That is, you must figure out how to relate a solution for a larger problem $P(k)$ to a solution for one or more of the smaller problems $P(1) \ldots, P(k-1)$. Most students want to do this by starting with a small problem, e.g. $P(k-1)$, and adding something to it. For more complex situations, however, it's usually better to start with the larger problem and try to find an instance of the smaller problem inside it.

11.5 Another example

The previous example applied induction to an algebraic formula. We can also apply induction to other sorts of statements, as long as they involve a suitable integer n.

Claim 39 *For any natural number n, $n^3 - n$ is divisible by 3.*

In this case, $P(n)$ is "$n^3 - n$ is divisible by 3."

So the outline of our proof looks like

> Proof: By induction on n.
>
> Base: Let $n = 0$. [prove that $n^3 - n$ is divisible by 3 when $n = 0$]
>
> Induction: Suppose that $n^3 - n$ is divisible by 3, for $n = 0, 1, \ldots, k$. We need to show that $(k+1)^3 - (k+1)$ is divisible by 3.

Fleshing out the details of the algebra, we get the following full proof:

> Proof: By induction on n.
>
> Base: Let $n = 0$. Then $n^3 - n = 0^3 - 0 = 0$ which is divisible by 3.
>
> Induction: Suppose that $n^3 - n$ is divisible by 3, for $n = 0, 1, \ldots, k$. We need to show that $(k+1)^3 - (k+1)$ is divisible by 3.

$(k+1)^3-(k+1) = (k^3+3k^2+3k+1)-(k+1) = (k^3-k)+3(k^2+k)$

From the inductive hypothesis, $(k^3 - k)$ is divisible by 3. And $3(k^2 + k)$ is divisible by 3 since $(k^2 + k)$ is an integer. So their sum is divisible by 3. That is $(k + 1)^3 - (k + 1)$ is divisible by 3. □

Notice that we've also used a variation on our induction outline, where the induction hypothesis covers values up through k (instead of $k - 1$) and we prove the claim at $n = k + 1$ (instead of at $n = k$). It doesn't matter whether your hypothesis goes through $n = k - 1$ or $n = k$, as long as you prove the claim for the next larger integer.

Also notice that our hypothesis says that the claim holds for $n = 0, 1, \ldots, k$. We are using the convention that this notation implies that $k \geq 0$, not that k is necessarily any larger (e.g. $k \geq 1$). That is, the 1 in this expresion is used only to show the pattern of increase, and it does not imply anything about the size of k.

The zero base case is technically enough to make the proof solid, but sometimes a zero base case doesn't provide good intuition or confidence. So you'll sometimes see an extra base case written out, e.g. $n = 1$ in this example, to help the author or reader see why the claim is plausible.

11.6 Some comments about style

Notice that the start of the proof tells you which variable in your formula (n in this case) is the induction variable. In this formula, the choice of induction variable is fairly obvious. But sometimes there's more than one integer floating around that might make a plausible choice for the induction variable. It's good style to always mention that you are doing a proof by induction, say what your induction variable is, and label your base and inductive steps.

Notice that the proof of the base case is very short. In fact, I've written about about twice as long as you'd normally see it. Almost all the time, the base case is trivial to prove and fairly obvious to both you and your reader. Often this step contains only some worked algebra and a check mark at the end. However, it's critical that you do check the base case. And, if your base

case involves an equation, compute the results for both sides (not just one side) so you can verify they are equal.

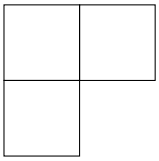
The important part of the inductive step is ensuring that you assume $P(1), \ldots, P(k-1)$ and use these facts to show $P(k)$. At the start, you must spell out your inductive hypothesis, i.e. what $P(n)$ is for your claim. It's usually helpful to explicitly substitute in some key values for n, e.g. work out what $P(k-1)$ is. Make sure that you use the information from the inductive hypothesis in your argument that $P(k+1)$ holds. If you don't, it's not an inductive proof and it's very likely that your proof is buggy.

At the start of the inductive step, it's also a good idea to say what you need to show, i.e. quote what $P(k)$ is.

These "style" issues are optional in theory, but actually critical for beginners writing inductive proofs. You will lose points if your proof isn't clear and easy to read. Following these style points (e.g. labelling your base and inductive steps) is a good way to ensure that it is, and that the logic of your proof is correct.

11.7 A geometrical example

Let's see another example of the basic induction outline, this time on a geometrical application. *Tiling* some area of space with a certain type of puzzle piece means that you fit the puzzle pieces onto that area of space exactly, with no overlaps or missing areas. A right triomino is a 2-by-2 square minus one of the four squares.



I claim that

Claim 40 *For any positive integer n, a $2^n \times 2^n$ checkerboard with any one square removed can be tiled using right triominoes.*

Proof: by induction on n.

Base: Suppose $n = 1$. Then our $2^n \times 2^n$ checkerboard with one square removed is exactly one right triomino.

Induction: Suppose that the claim is true for $n = 1, \ldots, k$. That is a $2^n \times 2^n$ checkerboard with any one square removed can be tiled using right triominoes as long as $n \leq k$.

Suppose we have a $2^{k+1} \times 2^{k+1}$ checkerboard C with any one square removed. We can divide C into four $2^k \times 2^k$ sub-checkerboards P, Q, R, and S. One of these sub-checkerboards is already missing a square. Suppose without loss of generality that this one is S. Place a single right triomino in the middle of C so it covers one square on each of P, Q, and R.

Now look at the areas remaining to be covered. In each of the sub-checkerboards, exactly one square is missing (S) or already covered (P, Q, and R). So, by our inductive hypothesis, each of these sub-checkerboards minus one square can be tiled with right triominoes. Combining these four tilings with the triomino we put in the middle, we get a tiling for the whole of the larger checkerboard C. This is what we needed to construct.

11.8 Graph coloring

We can also use induction to prove a useful general fact about graph colorability:

Claim 41 *For any positive integer D, if all nodes in a graph G have degree $\leq D$, then G can be colored with $D + 1$ colors.*

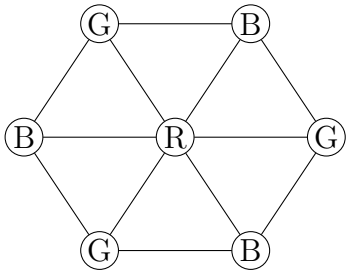
The objects involved in this claim are graphs. To apply induction to objects like graphs, we organize our objects by their size. Each step in the induction process will show that the claim holds for all objects of a particular

(integer) size. For graphs, the "size" would typically be either the number of nodes or the number of edges. For this proof, it's most convenient to use the number of nodes.

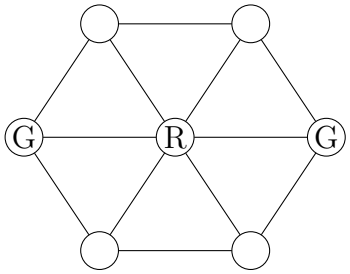
> Proof: Let's pick a positive integer D and prove the claim by induction on the number of nodes in G.
>
> Base: Since $D \geq 1$, the graph with just one node can obviously be colored with $D + 1$ colors.
>
> Induction: Suppose that any graph with at most $k - 1$ nodes and maximum node degree $\leq D$ can be colored with $D + 1$ colors.
>
> Let G be a graph with k nodes and maximum node degree $\leq D$. Remove some node v (and its edges) from G to create a smaller graph G'.
>
> G' has $k - 1$ nodes. Also, the maximum node degree of G' is no larger than D, because removing a node can't increase the degree. So, by the inductive hypothesis, G' can be colored with $D + 1$ colors.
>
> Because v has at most D neighbors, its neighbors are only using D of the available colors, leaving a spare color that we can assign to v. The coloring of G' can be extended to a coloring of G with $D + 1$ colors.

We can use this idea to design an algorithm (called the "greedy" algorithm) for coloring a graph. This algorithm walks through the nodes one-by-one, giving each node a color without revising any of the previously-assigned colors. When we get to each node, we see what colors have been assigned to its neighbors. If there is a previously used color not assigned to a neighbor, we re-use that color. Otherwise, we deploy a new color. The above theorem shows that the greedy algorithm will never use more than $D + 1$ colors.

Notice, however, that $D + 1$ is only an upper bound on the chromatic number of the graph. The actual chromatic number of the graph might be a lot smaller. For example, $D + 1$ would be 7 for the wheel graph W_6 but this graph actually has chromatic number only three:



The performance of the greedy algorithm is very sensitive to the order in which the nodes are considered. For example, suppose we start coloring W_6 by coloring the center hub, then a node v on the outer ring, and then the node opposite v. Then our partial coloring might look as shown below. Completing this coloring will require using four colors.



Notice that whether we need to deploy a new color to handle a node isn't actually determined by the degree of the node but, rather, by how many of its neighbors are already colored. So a useful heuristic is to order nodes by their degrees and color higher-degree nodes earlier in the process. This tends to mean that, when we reach a high-degree node, some of its neighbors will not yet be colored. So we will be able to handle the high-degree nodes with fewer colors and then extend this partial coloring to all the low-degree nodes.

11.9 Postage example

In the inductive proofs we've seen so far, we didn't actually need the full information in our inductive hypothesis. Our inductive step assumed that $P(n)$ was true for all values of n from the base up through $k-1$, a so-

called "strong" inductive hypothesis. However, the rest of the inductive step actually depended only on the information that $P(k-1)$ was true. We could, in fact, have used a simpler inductive hypothesis, known as a "weak" inductive hypothesis, in which we just assumed that $P(k-1)$ was true.

We'll now see some examples where a strong inductive hypothesis is essential, beucase the result for $n = k$ depends on the result for some smaller value of n, but it's not the immediately previous value $k-1$. Here's a classic example:

Claim 42 *Every amount of postage that is at least 12 cents can be made from 4-cent and 5-cent stamps.*

Before trying to prove a claim of this sort, you should try small examples and verify that the stated base case is correct. That is, in this case, that you can make postage for some representative integers, ≥ 12, but you can't do it for 11.

For example, 12 cents uses three 4-cent stamps. 13 cents of postage uses two 4-cent stamps plus a 5-cent stamp. 14 uses one 4-cent stamp plus two 5-cent stamps. If you experiment with small values, you quickly realize that the formula for making k cents of postage depends on the one for making $k-4$ cents of postage. That is, you take the stamps for $k-4$ cents and add another 4-cent stamp. We can make this into an inductive proof as follows:

> Proof: by induction on the amount of postage.
>
> Base: If the postage is 12 cents, we can make it with three 4-cent stamps. If the postage is 13 cents, we can make it with two 4-cent stamps. plus a 5-cent stamp. If it is 14, we use one 4-cent stamp plus two 5-cent stamps. If it is 15, we use three 5-cent stamps.
>
> Induction: Suppose that we have show how to construct postage for every value from 12 up through $k-1$. We need to show how to construct k cents of postage. Since we've already proved base cases up through 15 cents, we'll assume that $k \geq 16$.
>
> Since $k \geq 16$, $k-4 \geq 12$. So by the inductive hypothesis, we can construct postage for $k-4$ cents using m 4-cent stamps and

n 5-cent stamps, for some natural numbers m and n. In other words $k - 4 = 4m + 5n$.

But then $k = 4(m + 1) + 5n$. So we can construct k cents of postage using $m + 1$ 4-cent stamps and n 5-cent stamps, which is what we needed to show.

Notice that we needed to directly prove four base cases, since we needed to reach back four integers in our inductive step. It's not always obvious how many base cases are needed until you work out the details of your inductive step. The first base case was $n = 12$ because our claim only starts working consistently for integers ≥ 12.

11.10 Nim

In the parlour game Nim, there are two players and two piles of matches. At each turn, a player removes some (non-zero) number of matches from one of the piles. The player who removes the last match wins.[1]

Claim 43 *If the two piles contain the same number of matches at the start of the game, then the second player can always win.*

Here's a winning strategy for the second player. Suppose your opponent removes m matches from one pile. In your next move, you remove m matches from the other pile, thus evening up the piles. Let's prove that this strategy works.

Proof by induction on the number of matches (n) in each pile.

Base: If both piles contain 1 match, the first player has only one possible move: remove the last match from one pile. The second player can then remove the last match from the other pile and thereby win.

[1] Or, in some variations, loses. There seem to be several variations of this game.

> Induction: Suppose that the second player can win any game that starts with two piles of n matches, where n is any value from 1 through $k-1$. We need to show that this is true if $n = k$.
>
> So, suppose that both piles contain k matches. A legal move by the first player involves removing j matches from one pile, where $1 \leq j \leq k$. The piles then contain k matches and $k-j$ matches.
>
> The second player can now remove j matches from the other pile. This leaves us with two piles of $k-j$ matches. If $j = k$, then the second player wins. If $j < k$, then we're now effectively at the start of a game with $k-j$ matches in each pile. Since $j \geq 1$, $k-j \leq k-1$. So, by the induction hypothesis, we know that the second player can finish the rest of the game with a win.

The induction step in this proof uses the fact that our claim $P(n)$ is true for a smaller value of n. But since we can't control how many matches the first player removes, we don't know how far back we have look in the sequence of earlier results $P(1) \ldots P(k)$. Our previous proof about postage can be rewritten so as to avoid strong induction. It's less clear how to rewrite proofs like this Nim example.

11.11 Prime factorization

Early in this course, we saw the "Fundamental Theorem of Arithmetic," which states that every positive integer n, $n \geq 2$, can be expressed as the product of one or more prime numbers. Let's prove that this is true.

Recall that a number n is prime if its only positive factors are one and n. n is composite if it's not prime. Since a factor of a number must be no larger than the number itself, this means that a composite number n always has a factor larger than 1 but smaller than n. This, in turn, means that we can write n as ab, where a and b are both larger than 1 but smaller than n.[2]

Proof by induction on n.

[2] We'll leave the details of proving this as an exercise for the reader.

Base: 2 can be written as the product of a single prime number, 2.

Induction: Suppose that every integer between 2 and k can be written as the product of one or more primes. We need to show that $k+1$ can be written as a product of primes. There are two cases:

Case 1: $k+1$ is prime. Then it is the product of one prime, i.e. itself.

Case 2: $k+1$ is composite. Then $k+1$ can be written as ab, where a and b are integers such that a and b lie in the range $[2, k]$. By the induction hypothesis, a can be written as a product of primes $p_1 p_2 \ldots p_i$ and b can be written as a product of primes $q_1 q_2 \ldots q_j$. So then $k+1$ can be written as the product of primes $p_1 p_2 \ldots p_i q_1 q_2 \ldots q_j$.

In both cases $k+1$ can be written as a product of primes, which is what we needed to show.

Again, the inductive step needed to reach back some number of steps in our sequence of results, but we couldn't control how far back we needed to go.

11.12 Variation in notation

Certain details of the induction outline vary, depending on the individual preferences of the author and the specific claim being proved. Some folks prefer to assume the statement is true for k and prove it's true for $k+1$. Other assume it's true for $k-1$ and prove it's true for k. For a specific problems, sometimes one or the other choice yields a slightly simpler proofs.

Folks differ as to whether the notation $n = 0, 1, \ldots, k$ implies that k is necessarily at least 0, at least 1, or at least 2.

Some authors prefer to write strong induction hypotheses all the time, even when a weak hypothesis would be sufficient. This saves mental effort, because you don't have to figure out in advance whether a strong hypothesis

was really required. However, for some problems, a strong hypothesis may be more complicated to state than a weak one.

Authors writing for more experienced audiences may abbreviate the outline somewhat, e.g. packing an entirely short proof into one paragraph without labelling the base and inductive steps separately. However, being careful about the outline is important when you are still getting used to the technique.

Chapter 12

Recursive Definition

This chapter covers recursive definition, including finding closed forms.

12.1 Recursive definitions

Thus far, we have defined objects of variable length using semi-formal definitions involving .... For example, we defined the summation $\sum_{i=1}^{n} i$ by

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

This method is only ok when the reader can easily see what regular pattern the ... is trying to express. When precision is essential, e.g. when the pattern is less obvious, we need to switch to "recursive definitions."

Recursive function definitions in mathematics are basically similar to recursive procedures in programming languages. A recursive definition defines an object in terms of smaller objects of the same type. Because this process has to end at some point, we need to include explicit definitions for the smallest objects. So a recursive definition always has two parts:

- Base case or cases

- Recursive formula

For example, the summation $\sum_{i=1}^{n} i$ can be defined as:

- $g(1) = 1$
- $g(n) = g(n-1) + n$, for all $n \geq 2$

Both the base case and the recursive formula must be present to have a complete definition. However, it is traditional not to explicitly label these two pieces. You're just expected to figure out for yourself which parts are base case(s) and which is the recursive formula. The input values are normally assumed to be integers.

The true power of recursive definition is revealed when the result for n depends on the results for more than one smaller value, as in the strong induction examples. For example, the famous Fibonacci numbers are defined:

- $F_0 = 0$
- $F_1 = 1$
- $F_i = F_{i-1} + F_{i-2}, \quad \forall i \geq 2$

So $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13$, $F_8 = 21$, $F_9 = 34$. It isn't at all obvious how to express this pattern non-recursively.

12.2 Finding closed forms

Many recursive numerical formulas have a closed form, i.e. an equivalent expression that doesn't involve recursion (or summation or the like). Sometimes you can find the closed form by working out the first few values of the function and then guessing the pattern. More often, you need to use an organized technique. The simplest technique for finding closed forms is called "unrolling."

For example, suppose we have a function $T : \mathbb{N} \to \mathbb{Z}$ defined by

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n-1)+3, \quad \forall n \geq 2 \end{aligned}$$

The values of this function are $T(1) = 1$, $T(2) = 5$, $T(3) = 13$, $T(4) = 29$, $T(5) = 61$. It isn't so obvious what the pattern is.

The idea behind unrolling is to substitute a recursive definition into itself, so as to re-express $T(n)$ in terms of $T(n-2)$ rather than $T(n-1)$. We keep doing this, expressing $T(n)$ in terms of the value of T for smaller and smaller inputs, until we can see the pattern required to express $T(n)$ in terms of n and $T(0)$. So, for our example function, we would compute:

$$\begin{aligned} T(n) &= 2T(n-1)+3 \\ &= 2(2T(n-2)+3)+3 \\ &= 2(2(2T(n-3)+3)+3)+3 \\ &= 2^3T(n-3)+2^2\cdot 3+2\cdot 3+3 \\ &= 2^4T(n-4)+2^3\cdot 3+2^2\cdot 3+2\cdot 3+3 \\ &\ldots \\ &= 2^kT(n-k)+2^{k-1}\cdot 3+\ldots+2^2\cdot 3+2\cdot 3+3 \end{aligned}$$

The first few lines of this are mechanical substitution. To get to the last line, you have to imagine what the pattern looks like after k substitutions.

We can use summation notation to compactly represent the result of the kth unrolling step:

$$\begin{aligned} T(n) &= 2^kT(n-k)+2^{k-1}\cdot 3+\ldots+2^2\cdot 3+2\cdot 3+3 \\ &= 2^kT(n-k)+3(2^{k-1}+\ldots+2^2+2+1) \\ &= 2^kT(n-k)+3\sum_{i=0}^{k-1}(2^i) \end{aligned}$$

Now, we need to determine when the input to T will hit the base case. In our example, the input value is $n-k$ and the base case is for an input of 1. So we hit the base case when $n-k=1$. i.e. when $k=n-1$. Substituting this value for k back into our equation, and using the fact that $T(1)=1$, we get

$$\begin{aligned} T(n) &= 2^k T(n-k) + 3\sum_{i=0}^{k-1}(2^i) \\ &= 2^{n-1}T(1) + 3\sum_{i=0}^{n-2}(2^i) \\ &= 2^{n-1} + 3\sum_{k=0}^{n-2}(2^k) \\ &= 2^{n-1} + 3(2^{n-1}-1) = 4(2^{n-1}) - 3 = 2^{n+1} - 3 \end{aligned}$$

So the closed form for this function is $T(n) = 2^{n+1} - 3$. The unrolling process isn't a formal proof that our closed form is correct. However, we'll see below how to write a formal proof using induction.

12.3 Divide and conquer

Many important algorithms in computer science involve dividing a big problem of (integer) size n into a sub-problems, each of size n/b. This general method is called "divide and conquer." Analyzing such algorithms involves recursive definitions that look like:

$$\begin{aligned} S(1) &= c \\ S(n) &= aS(\lceil n/b \rceil) + f(n), \quad \forall n \geq 2 \end{aligned}$$

The base case takes some constant amount of work c. The term $f(n)$ is the work involved in dividing up the big problem and/or merging together

the solutions for the smaller problems. The call to the ceiling function is required to ensure that the input to S is always an integer.

Handling such definitions in full generality is beyond the scope of this class.[1] So let's consider a particularly important special case: dividing our problem into two half-size problems, where the dividing/merging takes time proportional to the size of the problem. And let's also restrict our input n to be a power of two, so that we don't need to use the ceiling function. We then get a recursive definition that looks like:

$$\begin{aligned} S(1) &= c \\ S(n) &= 2S(n/2) + n, \quad \forall n \geq 2 \ \ (n \text{ a power of } 2) \end{aligned}$$

Unrolling this, we get

$$\begin{aligned} S(n) &= 2S(n/2) + n \\ &= 2(2S(n/4) + n/2) + n \\ &= 4S(n/4) + n + n \\ &= 8S(n/8) + n + n + n \\ &\cdots \\ &= 2^i S(\frac{n}{2^i}) + in \end{aligned}$$

We hit the base case when $\frac{n}{2^i} = 1$ i.e. when $i = \log n$ (i.e. log base 2, which is the normal convention for algorithms applications). Substituting in this value for i and the base case value $S(1) = c$, we get

$$S(n) = 2^i S(\frac{n}{2^i}) + in = 2^{\log n} c + n \log n = cn + n \log n$$

So the closed form for $S(n)$ is $cn + n \log n$.

In real applications, our input n might not be a power of 2, so our actual recurrence might look like:

[1] See any algorithms text for more details.

$$\begin{aligned} S(1) &= c \\ S(n) &= 2S(\lceil n/2 \rceil) + n, \quad \forall n \geq 2 \end{aligned}$$

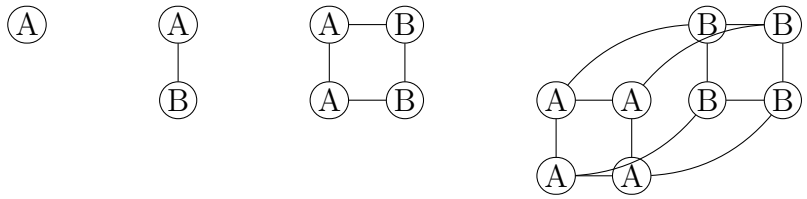
We could extend the details of our analysis to handle the input values that aren't powers of 2. In many practical contexts, however, we are only interested in the overall shape of the function, e.g. is it roughly linear? cubic? exponential? So it is often sufficient to note that S is increasing, so values of S for inputs that aren't powers of 2 will lie between the values of S at the adjacent powers of 2.

12.4 Hypercubes

Non-numerical objects can also be defined recursively. For example, the hypercube Q_n is the graph of the corners and edges of an n-dimensional cube. It is defined recursively as follows (for any $n \in \mathbb{N}$):

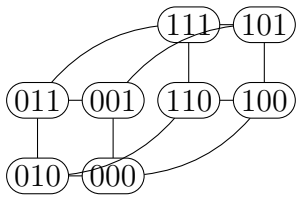
> Q_0 is a single node with no edges
>
> Q_n consists of two copies of Q_{n-1} with edges joining corresponding nodes, for any $n \geq 1$.

That is, each node v_i in one copy of Q_{n-1} is joined by an edge to its clone copy v_i' in the second copy of Q_{n-1}. Q_0, Q_1, Q_2, and Q_3 look as follows. The node labels distinguish the two copies of $Qn - 1$



The hypercube defines a binary coordinate system. To build this coordinate system, we label nodes with binary numbers, where each binary digit

corresponds to the value of one coordinate. The edges connect nodes that differ in exactly one coordinate.



Q^n has 2^n nodes. To compute the number of edges, we set up the following recursive definition for the number of edges $E(n)$ in the Q_n:

$E(0) = 0$

$E(n) = 2E(n-1) + 2^{n-1}$, for all $n \geq 1$

The 2^{n-1} term is the number of nodes in each copy of Q^{n-1}, i.e. the number of edges required to join corresponding nodes. We'll leave it as an exercise to find a closed form for this recursive definition.

12.5 Proofs with recursive definitions

Recursive definitions are ideally suited to inductive proofs. The main outline of the proof often mirrors the structure of the recursive definition. For example, let's prove the following claim about the Fibonacci numbers:

Claim 44 *For any $n \geq 0$, F_{3n} is even.*

Let's check some concrete values: $F_0 = 0$, $F_3 = 2$, $F_6 = 8$, $F_9 = 34$. All are even. Claim looks good. So, let's build an inductive proof:

Proof: by induction on n.

Base: $F_0 = 0$, which is even.

Induction: Suppose that F_{3n} is even for $n = 0, 1, \ldots, k$. We need to show that F_{3k} is even. We need to show that that $F_{3(k+1)}$ is even.

$F_{3(k+1)} = F_{3k+3} = F_{3k+2} + F_{3k+1}$

But $F_{3k+2} = F_{3k+1} + F_{3k}$. So, substituting into the above equation, we get:

$F_{3(k+1)} = (F_{3k+1} + F_{3k}) + F_{3k+1} = 2F_{3k+1} + F_{3k}$

By the inductive hypothesis F_{3k} is even. $2F_{3k+1}$ is even because it's 2 times an integer. So their sum must be even. So $F_{3(k+1)}$ is even, which is what we needed to show.

Some people feel a bit uncertain if the base case is a special case like zero. It's ok to also include a second base case. For this proof, you would check the case for $n = 1$ i.e. verify that F_3 is even. The extra base case isn't necessary for a complete proof, but it doesn't cause any harm and may help the reader.

12.6 Inductive definition and strong induction

Claims involving recursive definitions often require proofs using a strong inductive hypothesis. For example, suppose that the function $f : \mathbb{N} \to \mathbb{Z}$ is defined by

$f(0) = 2$

$f(1) = 3$

$\forall n \geq 1,\ f(n+1) = 3f(n) - 2f(n-1)$

I claim that:

Claim 45 $\forall n \in \mathbb{N}, f(n) = 2^n + 1$

We can prove this claim as follows:

Proof: by induction on n.

Base: $f(0)$ is defined to be 2. $2^0+1 = 1+1 = 2$. So $f(n) = 2^n+1$ when $n = 0$.

$f(1)$ is defined to be 3. $2^1 + 1 = 2 + 1 = 3$. So $f(n) = 2^n + 1$ when $n = 1$.

Induction: Suppose that $f(n) = 2^n + 1$ for $n = 0, 1, \ldots, k$.

$f(k + 1) = 3f(k) - 2f(k - 1)$

By the induction hypothesis, $f(k) = 2^k+1$ and $f(k-1) = 2^{k-1}+1$. Substituting these formulas into the previous equation, we get:

$f(k+1) = 3(2^k+1)-2(2^{k-1}+1) = 3\cdot 2^k+3-2^k-2 = 2\cdot 2^k+1 = 2^{k+1} + 1$

So $f(k + 1) = 2^{k+1} + 1$, which is what we needed to show.

We need to use a strong induction hypothesis, as well as two base cases, because the inductive step uses the fact that the formula holds for two previous values of n (k and $k - 1$).

12.7 Variation in notation

Recursive definitions are sometimes called inductive definitions or (especially for numerical functions) recurrence relations. Folks who call them "recurrence relations" typically use the term "initial condition" to refer to the base case.

Chapter 13

Trees

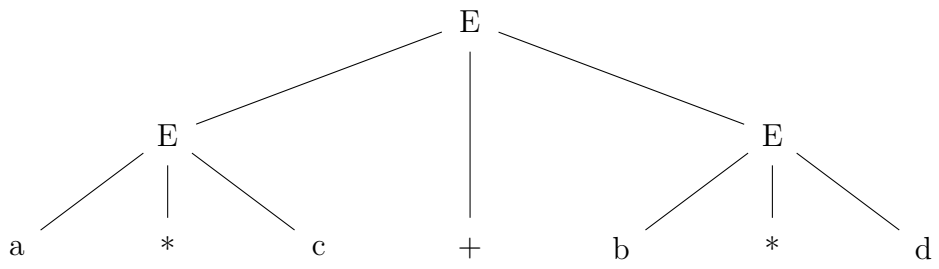
This chapter covers trees and induction on trees.

13.1 Why trees?

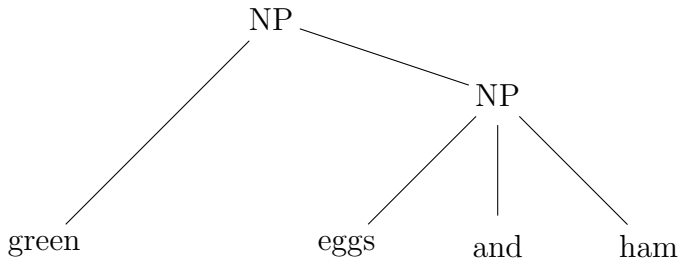
Trees are the central structure for storing and organizing data in computer science. Examples of trees include

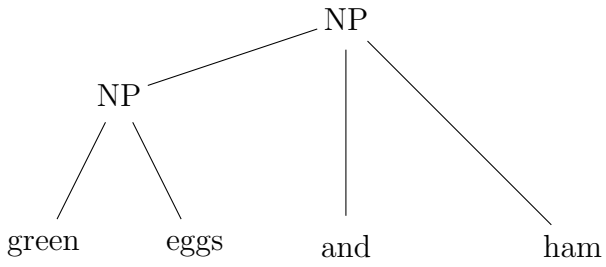
- Trees which show the organization of real-world data: family/geneaology trees, taxonomies (e.g. animal subspecies, species, genera, families)
- Data structures for efficiently storing and retrieving data. The basic idea is the same one we saw for binary search within an array: sort the data, so that you can repeatedly cut your search area in half.
- Parse trees, which show the structure of a piece of (for example) computer program, so that the compiler can correctly produce the corresponding machine code.
- Decision trees, which classify data by asking a series of questions. Each tree node contains a question, whose answer directs you to one of the node's children.

For example, here's a parse tree for the arithmetic expression $a * c + b * d$.

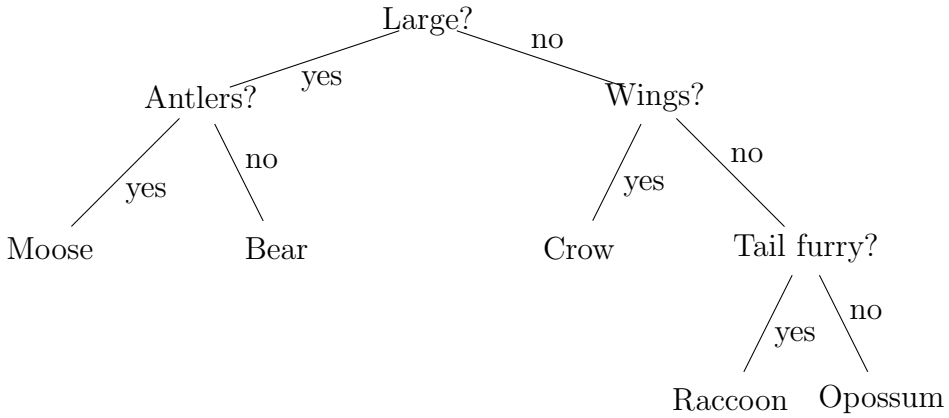


Computer programs that try to understand natural language use parse trees to represent the structure of sentences. For example, here are two possible structures for the phrase "green eggs and ham." In the first, but not the second, the ham would be green.

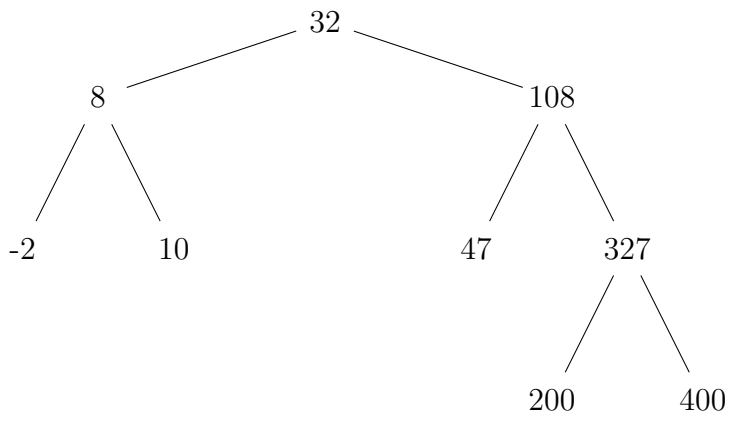




Here's a decision tree for figuring out what kind of animal is raiding your backyard trashcan. Decision trees are often used for engineering classification problems for which precise numerical models do not exist, such as transcribing speech waveforms into the basic sounds of a natural language.



And here is a tree storing the set of numbers $\{-2, 8, 10, 32, 47, 108, 200, 327, 400\}$



13.2 Defining trees

Formally, a tree is a undirected graph with a special node called the *root*, in which every node is connected to the root by exactly one path. When a pair of nodes are neighbors in the graph, the node nearest the root is called the *parent* and the other node is its *child*. By convention, trees are drawn with the root at the top. Two children of the same parent are known as *siblings*.

To keep things simple, we will assume that the set of nodes in a tree is finite. We will also assume that each set of siblings is ordered from left to right, because this is common in computer science applications.

A **leaf node** is a node that has no children. A node that does have children is known as an **internal node**. The root is an internal node, except in the special case of a tree that consists of just one node (and no edges).

The nodes of a tree can be organized into **levels**, based on how many edges away from the root they are. The root is defined to be level 0. Its children are level 1. Their children are level 2, and so forth. The **height** of a tree is the maximum level of any of its nodes or, equivalently, the maximum level of any of its leaves or, equivalently, the maximum length of a path from the root to a leaf.

If you can get from x to g by following zero or more parent links, then g is an **ancestor** of x and x is a **descendent** of g. So x is an ancestor/descendent of itself. The ancestors/descendents of x other than x itself are its **proper** ancestors/descendents. If you pick some random node a in a tree T, the **subtree rooted at** a consists of a (its root), all of a's descendents, and all the edges linking these nodes.

13.3 m-ary trees

Many applications restrict how many children each node can have. A binary tree (very common!) allows each node to have at most two children. An m-ary tree allows each node to have up to m children. Trees with "fat" nodes with a large bound on the number of children (e.g. 16) occur in some storage applications.

Important special cases involve trees that are nicely filled out in some sense. In a **full** m-ary tree, each node has either zero or m children. Never an intermediate number. So in a full 3-ary tree, nodes can have zero or three children, but not one child or two children.

In a **complete** m-ary tree, all leaves are at the same height. Normally, we'd be interested only in **full and complete** m-ary trees, where this means that the whole bottom level is fully populated with leaves.

For restricted types of trees like this, there are strong relationships between the numbers of different types of notes. for example:

Claim 46 *A full m-ary tree with i internal nodes has $mi + 1$ nodes total.*

To see why this is true, notice that there are two types of nodes: nodes with a parent and nodes without a parent. A tree has exactly one node with no parent. We can count the nodes with a parent by taking the number of parents in the tree (i) and multiplying by the branching factor m.

Therefore, the number of leaves in a full m-ary tree with i internal nodes is $(mi + 1) - i = (m - 1)i + 1$.

13.4 Height vs number of nodes

Suppose that we have a binary tree of height h. How many nodes and how many leaves does it contain? This clearly can't be an exact formula, since some trees are more bushy than others. But we can give useful upper and lower bounds.

To minimize the node counts, consider a tree of height h that has just one leaf. It contains $h + 1$ nodes connected into a straight line by h edges. So the minimum number of leaves is 1 (regardless of h) and the minimum number of nodes is $h + 1$.

The node counts are maximized by a tree which is full and complete. For these trees, the number of leaves is 2^h. More generally, the number of nodes at level L is 2^L. So the total number of nodes n is $\sum_{L=0}^{h} 2^L$. The closed form for this summation is $2^{h+1} - 1$. So, for full and complete binary trees, the height is proportional to $\log_2 n$.

Balanced binary trees are binary trees in which all leaves are at approximately the same height. The exact definition of "approximately" depends on the specific algorithms used to keep the tree balanced.[1] Balanced trees are more flexible than full and complete binary trees, but they also have height proportional to $\log_2 n$, where n is the number of nodes. This means that data stored in a balanced binary tree can be accessed and modified in $\log_2 n$ time.

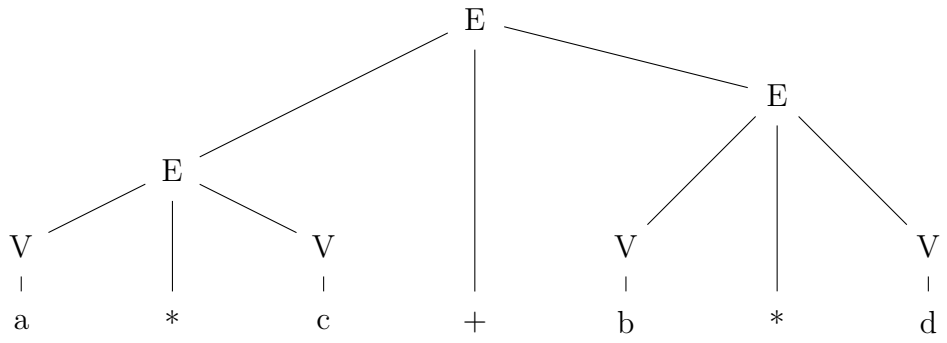
13.5 Context-free grammars

Applications involving languages, both human languages and computer languages, frequently involve **parse trees** that show the structure of a sequence of **terminal symbols**. A terminal symbol might be a word or a character, depending on the application. The terminal symbols are stored in the leaf nodes of the parse tree. The non-leaf nodes contains symbols that help indicate the structure of the sentence.

For example, the following tree shows one structure for the sentence:

[1] See textbooks on data structures and algorithms for further information.

$a * c + b * d$. This structure assumes you intend the multiplication to be done first, before the addition, i.e. $(a * c) + (b * d)$. The tree uses eight symbols: E, V, a, b, c, d, $+$, $*$.



This sort of labelled tree can be conveniently specified by a **context-free grammar**. A context-free grammar is a set of rules which specify what sorts of children are possible for a parent node with each type of label. The lefthand side of each rule gives the symbol on the parent node and the righthand side shows one possible pattern for the symbols on its children. If all the nodes of a tree T have children matching the rules of some grammar G, we say that the tree T and the **terminal sequence** stored in its leaves are **generated** by G.

For example, the following grammar contains four rules for an E node. The first rule allows an E node to have three children, the leftmost one labelled E, the middle one labelled $+$ and the right one labelled V. According to these rules, a node labelled V can only have one child, labelled either a, b, c, or d. The tree shown above follows these grammar rules.

$$
\begin{aligned}
E &\rightarrow E + V \\
E &\rightarrow E * V \\
E &\rightarrow V + V \\
E &\rightarrow V * V \\
V &\rightarrow a \\
V &\rightarrow b \\
V &\rightarrow c \\
V &\rightarrow d
\end{aligned}
$$

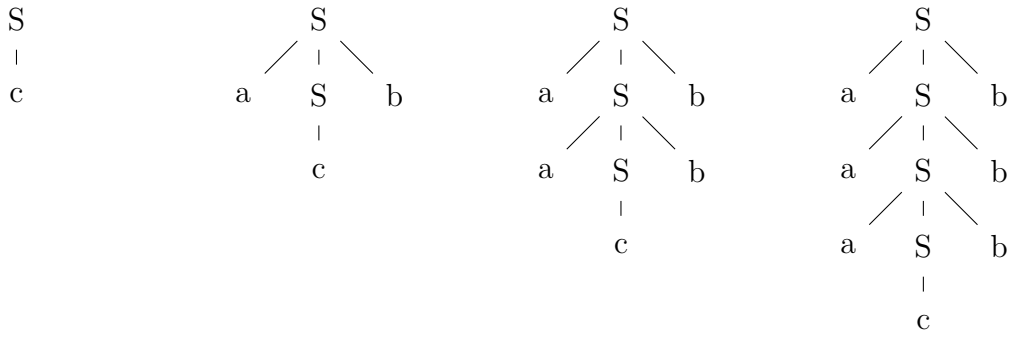
If a grammar contains several patterns for the children of a certain node label, we can pack them onto one line using a vertical bar to separate the options. E.g. we can write this grammar more compactly as

$$
\begin{aligned}
E &\rightarrow E + V \mid E * V \mid V + V \mid V * V \\
V &\rightarrow a \mid b \mid c \mid d
\end{aligned}
$$

To be completely precise about which trees are allowed by a grammar, we must specify two details beyond the set of grammar rules. First, we must give the set of **terminals**, i.e. symbols that are allowed to appear on the leaf nodes. Second, we must state which symbols, called **start symbols**, are allowed to appear on the root node. For example, for the above grammar, we might stipulate that the terminals are a, b, c, d, $+$, and $*$ and that the start symbols are E and V. Symbols that can appear on the lefthand side of rules are often written in capital letters.

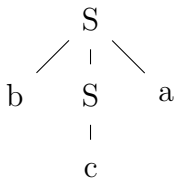
Consider the following grammar, with start symbol S and terminals a, b, and c.

$$
\begin{aligned}
S &\rightarrow aSb \\
S &\rightarrow c
\end{aligned}
$$

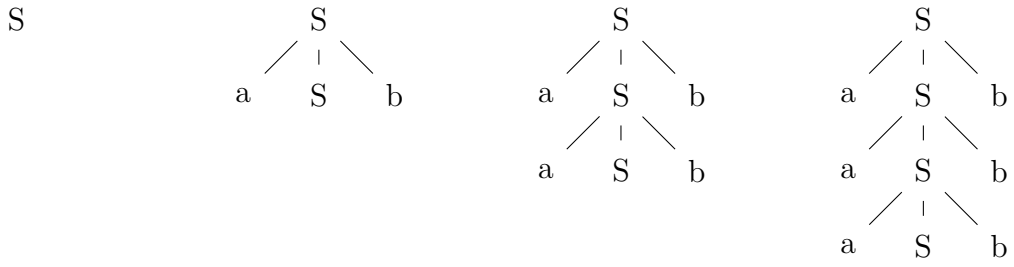


The sequences of terminals for the above trees are (left to right): c, acb, aacbb, aaacbbb. The sequences from this grammar always have a c in the middle, with some number of a's before it and the same number of b's after it.

Notice that the left-to-right order of nodes in the tree must match the order in the grammar rules. So, for example, the following tree doesn't match the above grammar.



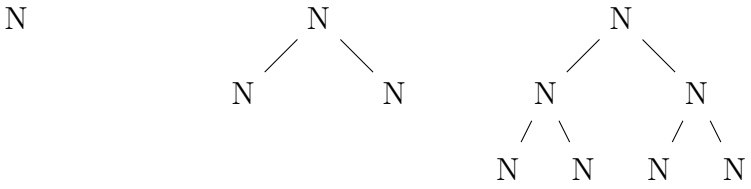
Notice also that we didn't allow S to be a terminal. If we added S to the set of terminals, our grammar would also allow the following trees:

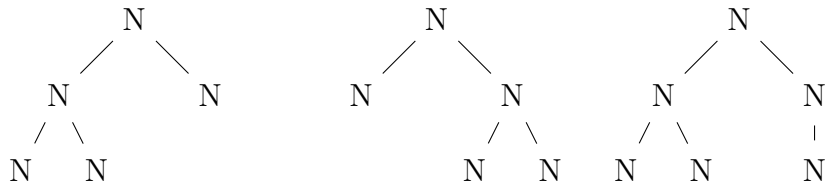


Sometimes it is helpful to use relatively few symbols, so as to concentrate on the possible shapes for the parse tree. Consider, for example, noun compounds such as "dump truck driver" or "wood salad tongs". Rather than getting into the specifics of the words involved, we might represent each noun with the symbol N. A noun compound structure can look like any binary tree, so it would have the following grammar, where N is both a start and a terminal symbol

$$N \;\rightarrow\; N \mid NN$$

Here are a number of trees generated by these rules:



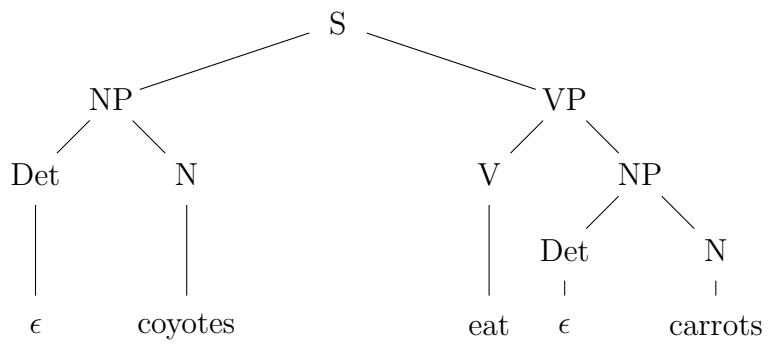


The bottom left tree shows the structure for "dump truck driver" and the bottom middle the structure of "wood salad tongs".

In some applications, it's convenient to have a branch of the parse tree which generates nothing in the terminal sequence. This is done by having a rule whose righthand side is ϵ, the standard symbol for an empty string. For example, here is a grossly-oversimplified grammar of an English sentence, with start symbol S and terminals: coyotes, rabbits, carrots, eat, kill, wash, the, all, some, and ϵ.

$$
\begin{aligned}
S &\rightarrow NP\ \ VP \\
VP &\rightarrow V\ \ NP \mid V\ \ NP\ \ NP \\
NP &\rightarrow Det\ \ N \\
N &\rightarrow coyotes \mid rabbits \mid carrots \\
V &\rightarrow eat \mid kill \mid wash \\
Det &\rightarrow \epsilon \mid the \mid all \mid some
\end{aligned}
$$

This will generate terminal sequences such as "All coyotes eat some rabbits." Every noun phrase (NP) is required to have a determiner (Det), but this determiner can expand into a word that's invisible in the terminal sequence. So we can also generate sequences like "Coyotes eat carrots."



It is possible[2] to revise this grammar so that it generates this sentence without using ϵ. However, some theories of the meaning of natural language sentences make a distinction between items which are completely missing and items which are present in the underlying structure but missing in the superficial structure. In this case, one might claim that the above sentence has a determiner which is understood to be "all."

13.6 Recursion trees

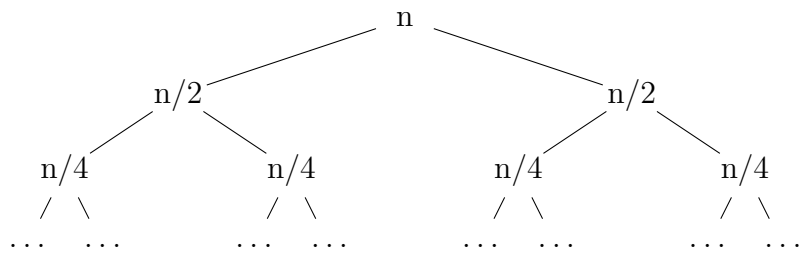
One nice application for trees is visualizing the behavior of certain recursive definitions, especially those used to describe algorithm behavior. For example, consider the following definition, where c is some constant.

$$\begin{aligned} S(1) &= c \\ S(n) &= 2S(n/2) + n, \quad \forall n \geq 2 \ \ (n \text{ a power of } 2) \end{aligned}$$

We can draw a picture of this definition using a "recursion tree". The top node in the tree represents $S(n)$ and contains everything in the formula for $S(n)$ **except the recursive calls to** S. The two nodes below it represent two copies of the computation of $S(n/2)$. Again, the value in each node

[2]Except when the grammar can generate a terminal sequence that contains only invisible ϵ symbols.

contains the non-recursive part of the formula for computing $S(n/2)$. The value of $S(n)$ is then the sum of the values in all the nodes in the recursion tree.



To sum everything in the tree, we need to ask:

- How high is this tree, i.e. how many levels do we need to expand before we hit the base case $n = 1$?
- For each level of the tree, what is the sum of the values in all nodes at that level?
- How many leaf nodes are there?

In this example, the tree has height $\log n$, i.e. there are $\log n$ non-leaf levels. At each level of the tree, the node values sum to n. So the sum for all non-leaf nodes is $n \log n$. There are n leaf nodes, each of which contributes c to the sum. So the sum of everything in the tree is $n \log n + cn$, which is the same closed form we found for this recursive definition in Section 12.3.

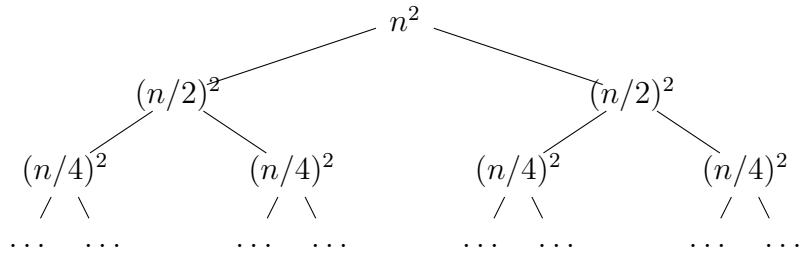
Recursion trees are particularly handy when you only need an approximate description of the closed form, e.g. is its leading term quadratic or cubic?

13.7 Another recursion tree example

Now, let's suppose that we have the following definition, where c is some constant.

$$\begin{aligned} P(1) &= c \\ P(n) &= 2P(n/2) + n^2, \qquad \forall n \geq 2 \ \ (n \text{ a power of } 2) \end{aligned}$$

Its recursion tree is



The height of the tree is again $\log n$. The sums of all nodes at the top level is n^2. The next level down sums to $n^2/2$. And then we have sums: $n^2/4$, $n^2/8$, $n^2/16$, and so forth. So the sum of all nodes at level k is $n^2\frac{1}{2^k}$.

The lowest non-leaf nodes are at level $\log n - 1$. So the sum of all the non-leaf nodes in the tree is

$$\begin{aligned} P(n) &= \sum_{k=0}^{\log n - 1} n^2 \frac{1}{2^k} = n^2 \sum_{k=0}^{\log n - 1} \frac{1}{2^k} \\ &= n^2(2 - \frac{1}{2^{\log n - 1}}) = n^2(2 - \frac{2}{2^{\log n}}) = n^2(2 - \frac{2}{n}) = 2n^2 - 2n \end{aligned}$$

Adding cn to cover the leaf nodes, our final closed form is $2n^2 + (c-2)n$.

13.8 Tree induction

When doing induction on trees, we divide the tree up at the top. That is, we view a tree as consisting of a root node plus some number of subtrees rooted at its children. The induction variable is typically the height of the tree. The child subtrees have height less than that of the parent tree. So, in the inductive step, we'll be assuming that the claim is true for these shorter child subtrees and showing that it's true for the taller tree that contains them.

For example, we claimed above that

Claim 47 *Let T be a binary tree, with height h and n nodes. Then $n \leq 2^{h+1} - 1$.*

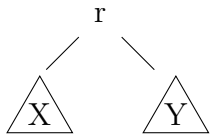
Proof by induction on h, where h is the height of the tree.

Base: The base case is a tree consisting of a single node with no edges. It has $h = 0$ and $n = 1$. Then we work out that $2^{h+1} - 1 = 2^1 - 1 = 1 = n$.

Induction: Suppose that the claim is true for all binary trees of height $< h$. Let T be a binary tree of height h ($h > 0$).

Case 1: T consists of a root plus one subtree X. X has height $h - 1$. So X contains at most $2^h - 1$ nodes. T only contains one more node (its root), so this means T contains at most 2^h nodes, which is less than $2^{h+1} - 1$.

Case 2: T consists of a root plus two subtrees X and Y. X and Y have heights p and q, both of which have to be less than h, i.e. $\leq h - 1$. X contains at most $2^{p+1} - 1$ nodes and Y contains at most $2^{q+1} - 1$ nodes, by the inductive hypothesis. But, since p and q are less than h, this means that X and Y each contain $\leq 2^h - 1$ nodes.



So the total number of nodes in T is the number of nodes in X plus the number of nodes in Y plus one (the new root node). This is $\leq 1+(2^p-1)+(2^q-1) \leq 1+2(2^h-1) = 1+2^{h+1}-2 = 2^{h+1}-1$

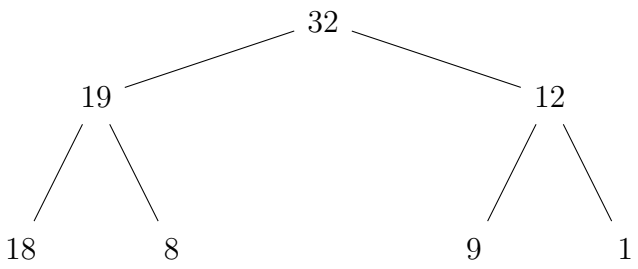
So the total number of nodes in T is $\leq 2^{h+1}-1$, which is what we needed to show. $\square$

In writing such a proof, it's tempting to think that if the full tree has height h, the child subtrees must have height $h-1$. This is only true if the tree is complete. For a tree that isn't necessarily complete, one of the subtrees must have height $h-1$ but the other subtree(s) might be shorter than $h-1$. So, for induction on trees, it is almost always necessary to use a strong inductive hypothesis.

In the inductive step, notice that we split up the big tree (T) at its root, producing two smaller subtrees (X) and (Y). Some students try to do induction on trees by grafting stuff onto the bottom of the tree. **Do not do this.** There are many claims, especially in later classes, for which this grafting approach will not work and it is essential to divide at the root.

13.9 Heap example

In practical applications, the nodes of a tree are often used to store data. Algorithm designers often need to prove claims about how this data is arranged in the tree. For example, suppose we store numbers in the nodes of a full binary tree. The numbers obey the *heap property* if, for every node X in the tree, the value in X is at least as big as the value in each of X's children. For example:



Notice that the values at one level aren't uniformly bigger than the values at the next lower level. For example, 18 in the bottom level is larger than 12 on the middle level. But values never decrease as you move along a path from a leaf up to the root.

Trees with the heap property are convenient for applications where you have to maintain a list of people or tasks with associated priorities. It's easy to retrieve the person or task with top priority: it lives in the root. And it's easy to restore the heap property if you add or remove a person or task.

I claim that:

Claim 48 *If a tree has the heap property, then the value in the root of the tree is at least as large as the value in any node of the tree.*

To keep the proof simple, let's restrict our attention to full binary trees:

Claim 49 *If a full binary tree has the heap property, then the value in the root of the tree is at least as large as the value in any node of the tree.*

Let's let $v(a)$ be the value at node a and let's use the recursive structure of trees to do our proof.

Proof by induction on the tree height h.

Base: $h = 0$. A tree of height zero contains only one node, so obviously the largest value in the tree lives in the root!

Induction: Suppose that the claim is true for all full binary trees of height $< h$. Let T be a tree of height h ($h > 0$) which has the heap property. Since T is a full binary tree, its root r has two children p and q. Suppose that X is the subtree rooted at p and Y is the subtree rooted at q.

Both X and Y have height $< h$. Moreover, notice that X and Y must have the heap property, because they are subtrees of T and the heap property is a purely local constraint on node values.

Suppose that x is any node of T. We need to show that $v(r) \geq v(x)$. There are three cases:

Case 1: $x = r$. This is obvious.

Case 2: x is any node in the subtree X. Since X has the heap property and height $\leq h$, $v(p) \geq v(x)$ by the inductive hypothesis. But we know that $v(r) \geq v(p)$ because T has the heap property. So $v(r) \geq v(x)$.

Case 3: x is any node in the subtree Y. Similar to case 2.

So, for any node x in T, $v(r) \geq v(x)$.□

13.10 Proof using grammar trees

Consider the following grammar G, with start symbol S and terminals a and b. I claim that all trees generated by G have the same number of nodes with label a as with label b.

$$\begin{aligned} S &\rightarrow ab \\ S &\rightarrow SS \\ S &\rightarrow aSb \end{aligned}$$

We can prove this by induction as follows:

Proof by induction on the tree height h.

Base: Notice that trees from this grammar always have height at least 1. The only way to produce a tree of height 1 is from the first rule, which generates exactly one a node and one b node.

Induction: Suppose that the claim is true for all trees of height $< k$, where $k \geq 1$. Consider a tree T of height k. The root must be labelled S and the grammar rules give us three possibilities for what the root's children look like:

Case 1: The root's children are labelled a and b. This is just the base case.

Case 2: The root's children are both labelled S. The subtrees rooted at these children have height $< k$. So, by the inductive hypothesis, each subtree has equal numbers of a and b nodes. Say that the left tree has m of each type and the right tree has n of each type. Then the whole tree has $m + n$ nodes of each type.

Case 3: The root has three children, labelled a, S, and b. Since the subtree rooted at the middle child has height $< k$, it has equal numbers of a and b nodes by the inductive hypothesis. Suppose it has m nodes of each type. Then the whole tree has $m+1$ nodes of each type.

In all three cases, the whole tree T has equal numbers of a and b nodes.

13.11 Variation in terminology

There are actually two sorts of trees in the mathematical world. This chapter describes the kind of trees most commonly used in computer science, which are formally "rooted trees" with a left-to-right order. In graph theory, the term "tree" typically refers to "free trees," which are connected acyclic graphs with no distinguished root node and no clear up/down or left-right directions. We will see free trees later, when analyzing planar graphs. Variations on these two types of trees also occur. For example, some data structures applications use trees that are almost like ours, except that a single child node must be designated as a "left" or "right" child.

Infinite trees occur occasionally in computer science applications. They are an obvious generalization of the finite trees discussed here.

Tree terminology varies a lot, apparently because trees are used for a wide range of applications with very diverse needs. In particular, some authors use the term "complete binary tree" to refer to a tree in which only the leftmost part of the bottom level is filled in. The term "perfect" is then used for a tree whose bottom level is entirely filled. Some authors don't allow a node to be an ancestor or descendent of itself. A few authors don't consider the root node to be a leaf.

Many superficial details of context-free grammar notation change with the application. For example, the BNF notation used in programming language specifications looks different at first glance because, for example, the grammar symbols are whole words surrounded by angle brackets.

Conventional presentations of context-free grammars are restricted so that terminals can't occur on the lefthand side of rules and there is a single start symbol. The broader definitions here allow us to capture the full range of examples floating around practical applications, without changing what sets of terminal sequences can be generated by these grammars.

Chapter 14

Big-O

This chapter covers asymptotic analysis of function growth and big-O notation.

14.1 Running times of programs

An important aspect of designing a computer programs is figuring out how well it runs, in a range of likely situations. Designers need to estimate how fast it will run, how much memory it will require, how reliable it will be, and so forth. In this class, we'll concentrate on speed issues.

Designers for certain small platforms sometimes develop very detailed models of running time, because this is critical for making complex applications work with limited resources. E.g. making God of War run on your Iphone. However, such programming design is increasingly rare, because computers are getting fast enough to run most programs without hand optimization.

More typically, the designer has to analyze the behavior of a large C or Java program. It's not feasible to figure out exactly how long such a program will take. The transformation from standard programming languages to machine code is way too complicated. Only rare programmers have a clear grasp of what happens within the C or Java compiler. Moreover, a very detailed analysis for one computer system won't translate to another pro-

gramming language, another hardware platform, or a computer purchased a couple years in the future. It's more useful to develop an analysis that abstracts away from unimportant details, so that it will be portable and durable.

This abstraction process has two key components:

- Ignore behavior on small inputs, concentrating on how well programs handle large inputs. (This is often called asymptotic analysis.)
- Ignore multiplicative constants

Multiplicative constants are ignored because they are extremely sensitive to details of the implementation, hardware platform, etc. Behavior on small inputs is ignored, because programs typically run fast enough on small test cases. Difficult practical problems typically arise when a program's use expands to larger examples. For example, a small database program developed for a community college might have trouble coping if deployed to handle (say) all registration records for U. Illinois.

14.2 Asymptotic relationships

So, suppose that you model the running time of a program as a function $f(n)$, where n is some measure of the size of the input problem. E.g. n might be the number of entries in a database application. Your competitor is offering a program that takes $g(n)$ on an input of size n. Is $f(n)$ faster than $g(n)$, slower than $g(n)$, or comparable to $g(n)$? To answer this question, we need formal tools to compare the growth rates of two functions.

Suppose that $f(n) = n$ and $g(n) = n^2$. For small positive inputs, n^2 is smaller. For the input 1, they have the same value, and then g gets bigger and rapidly diverges to become much larger than f. We'd like to say that g is "bigger," because it has bigger outputs for large inputs.

When a function is the sum of faster and slower-growing terms, we'll only be interested in the faster-growing term. For example, $n^2 + 7n + 105$ will be treated as equivalent to n^2. As the input n gets large, the behavior of the

function is dominated by the term with the fastest growth (the first term in this case).

Formally, we'll compare two functions $f(n)$ and $g(n)$ whose inputs and outputs are real numbers by looking at their ratio $\frac{f(n)}{g(n)}$. Because we are only interested in the running times of algorithms, most of our functions produce positive outputs. The exceptions, e.g. the log function, produce positive outputs once the inputs are large enough. So this ratio exists as long as we agree to ignore a few smaller input values.

We then look at what happens to this ratio as the input goes to infinity. Specifically, we'll define

(asymptotically equivalent) $f(n) \sim g(n)$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$

(asymptotically smaller) $f(n) \ll g(n)$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Recall from calculus that $\lim_{n\to\infty} h(n)$ is the value that output values $h(n)$ approach more and more closely as the input n gets larger and larger.[1] So two asymptotically equivalent functions become more and more similar as the input values get larger. If one function is asymptotically smaller, there is a gap between the output values that widens as the input values get larger.

So, for example $\lim_{n\to\infty} \frac{n^2+n}{n^2} = 1$ so $n^2 + n \sim n^2$. $\lim_{n\to\infty} \frac{n^2+17n}{n^3} = 0$ so $n^2 + 17n \ll n^3$. And $\lim_{n\to\infty} \frac{3n^2+17n}{n^2} = 3$ so $3n^2 + 17n$ and n^2 aren't related in either direction by $\sim$ or $\ll$.

If we pick two random functions f and g, $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ does not always exist because $\frac{f(n)}{g(n)}$ might oscillate rather than converging towards some particular number. In practice, we'll apply the above definitions only to selected well-behaved primitive functions and use a second, looser definition to handle both multiplicative constants and functions that might be badly behaved.

[1] Don't panic if you can't recall, or never saw, the formal definition of a limit. An informal understanding should be sufficient.

14.3 Ordering primitive functions

Let's look at some primitive functions and try to put them into growth order. It should be obvious that higher-order polynomials grow faster than lower-order ones. For example $n^2 \ll n^5$ because

$\lim_{n\to\infty} \frac{n^2}{n^5} = \lim_{n\to\infty} \frac{1}{n^3} = 0$

You're probably familiar with how fast exponentials grow. There's a famous story about a judge imposing a doubling-fine on a borough of New York, for ignoring the judge's orders. Since the borough officials didn't have much engineering training, it took them a few days to realize that this was serious bad news and that they needed to negotiate a settlement. So $n^2 \ll 2^n$. And, in general, for any exponent k, you can show that $n^k \ll 2^n$.

The base of the exponent matters. Consider 2^n and 3^n. $\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} (\frac{2}{3})^n = 0$ So $2^n \ll 3^n$.

Less obviously, $2^n \ll n!$. This is true because 2^n and $n!$ are each the product of n terms. For 2^n, they are all 2. For $n!$ they are the first n integers, and all but the first two of these are bigger than 2. So as n grows larger, the difference between 2^n and $n!$ widens. We'll do a formal proof of this result in Section 14.7 below.

We can summarize these facts as:

$$n \ll n^2 \ll n^3 \ldots \ll 2^n \ll 3^n \ll n!$$

For the purpose of designing computer programs, only the first three of these running times are actually good news. Third-order polynomials already grow too fast for most applications, if you expect inputs of non-trivial size. Exponential algorithms are only worth running on extremely tiny inputs, and are frequently replaced by faster algorithms (e.g. using statistical sampling) that return approximate results.

Now, let's look at slow-growing functions, i.e. functions that might be the running times of efficient programs. We'll see that algorithms for finding entries in large datasets often have running times proportional to $\log n$. If you draw the log function and ignore its strange values for inputs smaller than 1, you'll see that it grows, but much more slowly than n. And the

constant function 1 grows even more slowly: it doesn't grow at all!

Algorithms for sorting a list of numbers have running times that grow like $n \log n$. We know from the above that $1 \ll \log n \ll n$. We can multiply this equation by n because factors common to f and g cancel out in our definition of $\ll$. So we have $n \ll n \log n \ll n^2$.

We can summarize these primitive function relationships as:

$$1 \ll \log n \ll n \ll n \log n \ll n^2$$

It's well worth memorizing the relative orderings of these basic functions, since you'll see them again and again in this and future CS classes.

14.4 The dominant term method

The beauty of these asymptotic relationships is that it is rarely necessary to go back to the original limit definition. Having established the relationships among a useful set of primitive functions, we can usually manipulate expressions at a high level. And we can typically look only at the terms that dominant each expression, ignoring other terms that grow more slowly.

The first point to notice is that $\sim$ and $\ll$ relations for a sum are determined entirely by the dominant term, i.e. the one that grows fastest. If $f(n) \ll g(n)$, then $g(n) \sim g(n) \pm f(n)$. For example, suppose we are confronted with the question of whether

$$47n + 2n! + 17 \ll 3^n + 102n^3$$

The dominant term on the lefthand side is $2n!$. On the righthand side, it is 3^n. So our question reduces to whether

$$2n! \ll 3^n$$

We know from the previous section that this is false, because

$$3^n \ll n!$$

.

The asymptotic relationships also interact well with normal rules of algebra. For example, if $f(n) \ll g(n)$ and $h(n)$ is any non-zero function, then $f(n)h(n) \ll g(n)h(n)$.

14.5 Big-O

Asymptotic equivalence is a great way to compare well-behaved reference functions, but it is less good for comparing program running times to these reference functions. There are two issues. We'd like to ignore constant multipliers, even on the dominant term, because they are typically unknown and prone to change. Moreover, running times of programs may be somewhat messy functions, which may involve operations like floor or ceiling. Some programs[2] may run dramatically faster on "good" input sizes than on "bad" sizes, creating oscillation that may cause our limit definition not to work.

Therefore, we typically use a more relaxed relationship, called **big-O** to compare messier functions to one another or two reference functions. Suppose that f and g are functions whose domain and co-domain are subsets of the real numbers. Then $f(n)$ is $O(g(n))$ (read "big-O of g) if and only if

> There are positive real numbers c and k such that $0 \leq f(n) \leq cg(n)$ for every $n \geq k$.

The factor c in the equation models the fact that we don't care about multiplicative constants, even on the dominant term. The function $f(n)$ is allowed to wiggle around as much as it likes, compared to $cg(n)$, as long as it remains smaller. And we're entirely ignoring what f does on inputs smaller than k.

If $f(n)$ is asymptotically smaller than $g(n)$, then $f(n)$ is $O(g(n))$ but $g(n)$ is not $O(f(n))$ but So, for example, $3n^2 + 17n$ is $O(2^n)$. But $3n^2$ is not $O(n + 132)$. The big-O relationship also holds when the two functions have the same dominant term, with or without a difference in constant multiplier. E.g. $3n^2$ is $O(n^2 - 17n)$ because the dominant term for both functions has

[2]Naive methods of testing for primality, for example.

the form cn^2. So the big-O relationship is a non-strict partial order like $\leq$ on real numbers, whereas $\ll$ is a strict partial order like $<$.

When $g(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$, then $f(n)$ and $g(n)$ are forced to remain close together as n goes to infinity. In this case, we say that $f(n)$ is $\Theta(g(n))$ (and also $g(n)$ is $\Theta(f(n))$). The Θ relationship is an equivalence relation on this same set of functions. So, for example, the equivalence class $[n^2]$ contains functions such as n^2, $57n^2 - 301$, $2n^2 + n + 2$, and so forth.

Notice that logs with different bases differ only by a constant multiplier, i.e. a multiplier that doesn't depend on the input n. E.g. $\log_2(n) = \log_2(3)\log_3(n)$. This means that $\log_p(n)$ is $\Theta(\log_q(n))$ for any choice of p and q. Because the base of the log doesn't change the final big-O answer, computer scientists often omit the base of the log function, assuming you'll understand that it does not matter.

14.6 Applying the definition of big-O

To show that a big-O relationship holds, we need to produce suitable values for c and k. For any particular big-O relationship, there are a wide range of possible choices. First, how you pick the multiplier c affects where the functions will cross each other and, therefore, what your lower bound k can be. Second, there is no need to minimize c and k. Since you are just demonstrating existence of suitable c and k, it's entirely appropriate to use overkill values.

For example, to show that $3n$ is $O(n^2)$, we can pick $c = 3$ and $k = 1$. Then $3n \leq cn^2$ for every $n \geq k$ translates into $3n \leq 3n^2$ for every $n \geq 1$, which is clearly true. But we could have also picked $c = 100$ and $k = 100$.

Overkill seems less elegant, but it's easier to confirm that your chosen values work properly, especially in situations like exams. Moreover, slightly overlarge values are often more convincing to the reader, because the reader can more easily see that they do work.

To take a more complex example, let's show that $3n^2 + 7n + 2$ is $O(n^2)$. If we pick $c = 3$, then our equation would look like $3n^2 + 7n + 2 \leq 3n^2$. This clearly won't work for large n.

So let's try $c = 4$. Then we need to find a lower bound on n that makes $3n^2 + 7n + 2 \leq 4n^2$ true. To do this, we need to force $7n + 2 \leq n^2$. This will be true if n is big, e.g. ≥ 100. So we can choose $k = 100$.

14.7 Proving a primitive function relationship

Let's see how to pin down the formal details of one ordering between primitive functions, using induction. I claimed above that $2^n \ll n!$. To show this relationship, notice that increasing n to $n+1$ multiplies the lefthand side by 2 and the righthand side by $n + 1$. So the ratio changes by $\frac{2}{n+1}$. Once n is large enough $\frac{2}{n+1}$ is at least $\frac{1}{2}$. Firming up the details, we find that

Claim 50 *For every positive integer $n \geq 4$, $\frac{2^n}{n!} < (\frac{1}{2})^{n-4}$.*

If we can prove this relationship, then since it's well-known from calculus that $(\frac{1}{2})^n$ goes to zero as n increases, $\frac{2^n}{n!}$ must do so as well.

To prove our claim by induction, we outline the proof as follow:

> Proof: Suppose that n is an integer and $n \geq 4$. We'll prove that $\frac{2^n}{n!} < (\frac{1}{2})^{n-4}$ using induction on n.
>
> Base: $n = 4$. [show that the formula works for $n = 4$]
>
> Induction: Suppose that $\frac{2^n}{n!} < (\frac{1}{2})^{n-4}$ holds for $n = 4, 5, \ldots, k$. And, in particular, $\frac{2^k}{k!} < (\frac{1}{2})^{k-4}$
>
> We need to show that the claim holds for $n = k + 1$, i.e. that $\frac{2^{k+1}}{(k+1)!} < (\frac{1}{2})^{k-3}$

When working with inequalities, the required algebra is often far from obvious. So, it's especially important to write down your inductive hypothesis (perhaps explicitly writing out the claim for the last one or two values covered by the hypothesis) and the conclusion of your inductive step. You can then work from both ends to fill in the gap in the middle of the proof.

Proof: Suppose that n is an integer and $n \geq 4$. We'll prove that $\frac{2^n}{n!} < (\frac{1}{2})^{n-4}$ using induction on n.

Base: $n = 4$. Then $\frac{2^n}{n!} = \frac{16}{24} < 1 = (\frac{1}{2})^0 = (\frac{1}{2})^{n-4}$.

Induction: Suppose that $\frac{2^n}{n!} < (\frac{1}{2})^{n-4}$ holds for $n = 4, 5, \ldots, k$.

Since $k \geq 4$, $\frac{2}{k+1} < \frac{1}{2}$. So $\frac{2^{k+1}}{(k+1)!} < \frac{1}{2} \cdot \frac{2^k}{k!}$.

By our inductive hypothesis $\frac{2^k}{k!} < (\frac{1}{2})^{k-4}$. So $\frac{1}{2} \cdot \frac{2^k}{k!} < (\frac{1}{2})^{k-3}$.

So then we have $\frac{2^{k+1}}{(k+1)!} < \frac{1}{2} \cdot \frac{2^k}{k!} < (\frac{1}{2})^{k-3}$ which is what we needed to show.

Induction can be used to establish similar relationships between other pairs of primitive functions, e.g. n^2 and 2^n.

14.8 Variation in notation

Although the concepts in this area are reasonable constant across authors, the use of shorthand symbols is not. The symbol $\sim$ is used for many diverse purposes in mathematics. Authors frequently write "f(n) is o(g(n))" (with a small o rather than a big one) to mean $f(n) \ll g(n)$. And $\ll$ may be used to mean big-O. Fortunately, the usage of big-O seems to be standard.

In computer science, we typically look at the behavior of functions as input values get large. Areas (e.g. perturbation theory) use similar definitions and notation, but with limits that tend towards some specific finite value (e.g. zero).

In the definition of big-O, some authors replace $0 \leq f(n) \leq cg(n)$ with $|f(n)| \leq c|g(n)|$. This version of the definition can compare functions with negative output values but is correspondingly harder for beginners to work with. Some authors state the definition only for functions f and g with positive output values. This is awkward because the logarithm function produces negative output values for very small inputs.

Outside theory classes, computer scientists often say that $f(n)$ is $O(g(n))$ when they actually mean the stronger statement that $f(n)$ is $\Theta(g(n))$. When you do know that the bound is tight, i.e. that the functions definitely grow

at the same rate, it's more helpful to your readers to make this clear by using Θ.

The notation $O(f(n))$ is often embedded in equations, meaning "some random function that is $O(f(n))$. For enample, authors frequently write things like $n^3 + O(n^2)$, in which $O(n^2)$ is some unspecified function that grows no faster than n^2.

Very, very annoyingly, for historical reasons, the statement $f(n)$ is $O(g(n))$ is often written as $f(n) = O(g(n))$. This looks like a sort of equality, but it isn't. It is actually expressing an inequality.

Chapter 15

Algorithms

This chapter covers how to analyze the running time of algorithms.

15.1 Introduction

The techniques we've developed earlier in this course can be applied to analyze how much time a computer algorithm requires, as a function of the size of its input(s). We will see a range of simple algorithms illustrating a variety of running times. Three methods will be used to analyze the running times: nested loops, resource consumption, and recursive definitions.

We will figure out only the big-O running time for each algorithm, i.e. ignoring multiplicative constants and behavior on small inputs. This will allow us to examine the overall design of the algorithms without excessive complexity. Being able to cut corners so as to get a quick overview is a critical skill when you encounter more complex algorithms in later computer science classes.

15.2 Basic data structures

Many simple algorithms need to store a sequence of objects $a_1, \ldots, a_n$. In pseudocode, the starting index for a sequence is sometimes 0 and sometimes

1, and you often need to examine the last subscript to find out the length. Sequences can be stored using either an array or a linked list. The choice sometimes affects the algorithm analysis, because these two implementation methods have slightly different features.

An array provides constant-time access to any element. So you can quickly access elements in any order you choose and the access time does not depend on the length of the list. However, the length of an array is fixed when the array is built. Changing the array length takes time proportional to the length of the array, i.e. $O(n)$. Adding or deleting objects in the middle of the array requires pushing other objects sideways. This can also take $O(n)$ time. Two-dimensional arrays are similar, except that you need to supply two subscripts e.g. $a_{x,y}$.

In a linked list, each object points to the next object in the list. An algorithm has direct access only to the elements at the ends of the list.[1] Objects in the middle of the list can only be accessed by walking element-by-element from one end, which can take $O(n)$ time. However, the length of the list is flexible and objects can added to, or removed from, the ends of the list in constant time. Once you are at a position in the middle of a list, objects can be added or deleted at that position in constant time.

A linked list starts with its *head* and ends with its *tail*. For example, suppose our list is $L = (1, 7, 3, 4, 7, 19)$. Then $head(L)$ is 1 and $tail(L)$ is 19. The function *pop* removes and returns the value at the head of a list. I.e. pop(L) will return 1 leave the list L containing $(7, 3, 4, 7, 19)$.

For some algorithms, the big-O performance does not depend on whether arrays or linked lists are used. This happens when the number of objects is fixed and the objects are accessed in sequential order. However, remember that a big-O analysis ignores multiplicative constants. All other things being equal, array-based implementations tend to have smaller constants and therefore run faster.

[1] Strictly speaking, this works only for certain types of linked lists. See a data structures text for all the gory details.

15.3 Nested loops

Algorithms based on nested loops are the easiest to analyze. Suppose that we have a set of 2D points and we would like to find the pair that are closest together. Our code might look as in Figure 15.1, assuming that the function dist computes the distance between two 2D points.

To analyze this code in big-O terms, first notice that the start-up code in lines 1-4 and the ending code in line 12 takes the same amount of time regardless of the input size n. So we'll say that it takes "constant time" or $O(1)$ time. The block of code inside both loops (lines 7-11) takes a constant time to execute once. So the big-O running time of this algorithm is entirely determined by how many times the loops run.

The outer loop runs n times. The inner loop runs n times during each iteration of the outer loop. So the block of code inside both loops executes $O(n^2)$ times.

```
01  closestpair(p_1, ..., p_n) : array of 2D points)
02          best1 = p_1
03          best2 = p_2
04          bestdist = dist(p_1,p_2)
05          for i = 1 to n
06                  for j = 1 to n
07                          newdist = dist(p_i,p_j)
08                          if (i ≠ j and newdist < bestdist)
09                                  best1 = p_i
10                                  best2 = p_j
11                                  bestdist = newdist
12          return (best1, best2)
```

Figure 15.1: Finding the closest pair using nested loops.

This code examines each pair of 2D points twice, once in each order. We could avoid this extra work by having the inner loop (j) run only from $i+1$ to n. In this case, the code inside both loops will execute $\sum_{i=1}^{n}(n-i)$ times. This is equal to $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$. This is still $O(n^2)$: our optimization improved the constants but not the big-O running time. Improving the big-

O running time—$O(n \log n)$ is possible–requires restructuring the algorithm and involves some geometrical detail beyond the scope of this chapter.

15.4 Merging two lists

When code contains a while loop, rather than a for loop, it can be less obvious how many times the loop will run. For example, suppose we have two sorted lists, $a_1, \ldots, a_p$ and $b_1, \ldots, b_q$. We can merge them very efficiently into a combined sorted list. To do this, we make a new third empty list to contain our merged output. Then we examine the first elements of the two input lists and move the smaller value onto our output list. We keep looking at the first elements of both lists until one list is empty. We then copy over the rest of the non-empty list. Figure 15.2 shows pseudocode for this algorithm.

```
01  merge(L1,L2: sorted lists of real numbers)
02        O = emptylist
03        while (L1 is not empty or L2 is not empty)
04              if (L1 is empty)
05                    move head(L2) to the tail of O
06              else if (L2 is empty)
07                    move head(L1) to the tail of O
08              else if (head(L1) <= head(L2))
09                    move head(L1) to the tail of O
10              else move head(L2) to the tail of O
11        return O
```

Figure 15.2: Merging two lists

For merge, a good measure of the size of the input is the length of the output array n, which is equal to the sum of the lengths of the two input arrays $(p + q)$. The merge function has one big while loop. Since the operations within the loop (lines 4-10) all take constant time, we just need to figure out how many times the loop runs, as a function of n.

A good way to analyze while loops is to track a resource that has a known size and is consumed as the loop runs. In this case, each time the loop runs, we move one number from L_1 or L_2 onto the output list O. We have n

numbers to move, after while the loop halts. So the loop must run n times. So merge takes $O(n)$ (aka linear) time. This is called a *resource consumption* analysis.

15.5 A reachability algorithm

The function in Figure 15.3 determines whether one node t in a graph is reachable from another node s, i.e. whether there is a path connecting s and t. To do this, we start from s and explore outwards by following the edges of the graph. When we visit a node, we place a marker on it so that we can avoid examining it again (and thereby getting the code into an infinite loop). The temporary list M contains nodes that we have reached, but whose neighbors have not yet been explored.[2]

```
01  reachable(G: a graph; s,t: nodes in G)
02          if s = t return true
03          Unmark all nodes of G.
04          M = emptylist
05          Mark node s and add it to M
06          while (M is not empty)
07                  p = pop(M)
08                  for every node q that is a neighbor of p in G
09                          if q = t return true
10                          else if q is not marked, mark q and add it to M
11          return false
```

Figure 15.3: Can node s be reached from node t?

We can use a resource consumption argument to analyze the while loop in this code. Suppose that G has n nodes and m edges. No node is put onto the list M more than once. So the loop in lines 6-10 runs no more than n times.

[2]Line 5 is deliberately vague about which end of M the nodes are added to and, therefore, vague about the order in which nodes are explored.

Line 8 starts at a node q and find all its neighbors p.[3] So it traces all the edges involving q. During the whole run of the code, a graph edge might get traced twice, once in each direction. There are m edges in the graph. So lines 9-10 cannot run more than $2m$ times.

In total, this algorithm needs $O(n+m)$ time. This is an interesting case because neither of the two terms n or m dominates the other. It is true that the number of edges m is no $O(n^2)$ and thus the connected component algorithm is $O(n^2)$. However, in most applications, relatively few of these potential edges are actually present. So the $O(n+m)$ bound is more helpful.

Notice that there is a wide variety of graphs with n nodes and m edges. Our analysis was based on the kind of graph that wold causes the algorithm to run for the longest time, i.e. a graph in which the algorithm reaches every node and traverses every edge, reaching t last. This is called a *worst-case* analysis. On some input graphs, our code might run much more quickly, e.g. if we encounter t early in the search or if much of the graph is not connected to s. Unless the author explicitly indicates otherwise, big-O algorithm analyses are normally understood to be worst-case.

15.6 Binary search

We will now look at a strategy for algorithm design called "divide and conquer," in which a larger problem is solved by dividing it into several (usually two) smaller problems. For example, the code in Figure 15.4 uses a technique called binary search to calculate the square root of its input. For simplicity, we'll assume that the input n is quite large, so that we only need the answer to the nearest integer.

This code operates by defining a range of integers in which the answer must live, initially between 1 and n. Each recursive call tests whether the midpoint of the range is higher or lower than the desired answer, and selects the appropriate half of the range for further exploration. The helper function squarerootrec returns when it has found $\lfloor\sqrt(n)\rfloor$. The function then checks whether this value, or the next higher integer, is the better approximation.

[3] We assume that the internal computer representation of the graph stores a list of neighbors for each node.

```
01  squareroot(n: positive integer)
02          p = squarerootrec(n, 1, n)
03          if (n − p^2 ≤ (p + 1)^2 − n)
05                  return p
05          else return p + 1


11  squarerootrec(n, bottom, top: positive integers)
12          if (bottom = top) return bottom
13          middle = floor((bottom+top)/2)
14          if (middle^2 == n)
15                  return middle
16          else if (middle^2 ≤ n)
17                  return squarerootrec(n,middle,top)
18          else
19                  return squarerootrec(n,bottom,middle)
```

Figure 15.4: Binary search for $\sqrt{n}$

This isn't the fastest way to find a square root,[4] but this simple method generalizes well to situations in which we have only a weak model of the function we are trying to optimize. This method requires only that you can test whether a candidate value is too low or too high. Suppose, for example, that you are tuning a guitar string. Many amateur players can tell if the current tuning is too low or too high, but have a poor model of how far to turn the tuning knob. Binary search would be a good strategy for optimizing the tuning.

To analyze how long binary search takes, first notice that the start-up and clean-up work in the main squareroot function takes only constant time. So we can basically ignore its contribution. The function squarerootrec makes one recursive call to itself and otherwise does a constant amount of work. The base case requires only a constant amount of work. So if the running time of squarerootrec is $T(n)$, we can write the following recursive definition for $T(n)$, where c and d are constants.

[4] A standard faster approach is Newton's method.

- $T(1) = c$
- $T(n) = T(n/2) + d$

If we unroll this definition k times, we get $T(n) = T(\frac{n}{2^k}) + kd$. Assuming that n is a power of 2, we hit the base case when $k = \log n$. So $T(n) = c + d \log n$, which is $O(\log n)$.

15.7 Mergesort

Mergesort takes an input linked list of numbers and returns a new linked list containing the sorted numbers. This is somewhat different from bubble and insertion sort, which rearrange the values within a single array (and don't return anything).

Mergesort divides its big input list (length n) into two smaller lists of length $n/2$. Lists are divided up repeatedly until we have a large number of very short lists, of length 1 or 2 (depending on the preferences of the code writer). A length-1 list is necessarily sorted. A length 2 list can be sorted in constant time. Then, we take all these small sorted lists and merge them together in pairs, gradually building up longer and longer sorted lists until we have one sorted list containing all of our original input numbers. Figure 15.5 shows the resulting pseudocode.

```
01  mergesort(L = a_1, a_2, ..., a_n: list of real numbers)
02        if (n = 1) then return L
03        else
04              m = ⌊n/2⌋
05              L_1 = (a_1, a_2, ..., a_m)
06              L_2 = (a_{m+1}, a_{m+2}, ..., a_n)
07              return merge(mergesort(L_1),mergesort(L_2))
```

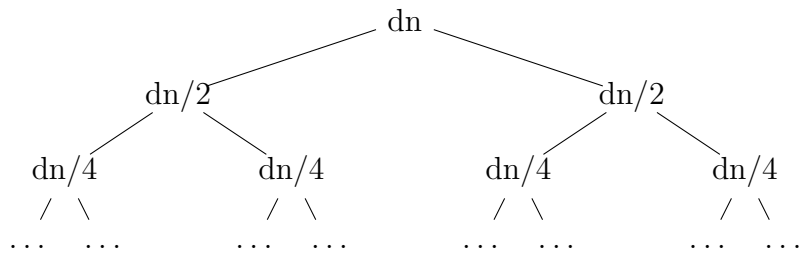
Figure 15.5: Sorting a list using mergesort

On an input of length n, mergesort makes two recursive calls to itself. It also does $O(n)$ work dividing the list in half, because it must walk, element by element, from the head of the list down to the middle position. And it

does $O(n)$ work merging the two results. So if the running time of mergesort is $T(n)$, we can write the following recursive definition for $T(n)$, where c and d are constants.

- $T(1) = c$
- $T(n) = 2T(n/2) + dn$

This recursive definition has the following recursion tree:



The tree has $O(\log n)$ non-leaf levels and the work at each level sums up to dn. So the work from the non-leaf nodes sums up to $O(n \log n)$. In addition, there are n leaf nodes (aka base cases for the recursive function), each of which involves c work. So the total running time is $O(n \log n) + cn$ which is just $O(n \log n)$.

15.8 Tower of Hanoi

The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. It consists of three pegs and a set of k disks of graduated size that fit on them. The disks start out in order on one peg. You are allowed to move only a single disk at a time. The goal is to rebuild the ordered tower on another peg without ever placing a disk on top of a smaller disk.

The best way to understand the solution is recursively. Suppose that we know how to move k disks from one peg to another peg, using a third temporary-storage peg. To move $k+1$ disks from peg A to peg B using a third peg C, we first move the top k disks from A to C using B as temporary storage. Then we move the biggest disk from A to B. Then we move the other k disks from C to B, using A as temporary storage. So our recursive solver would have pseudocode as in Figure 15.6.

```
01  hanoi(A,B,C: pegs, d_1, d_2 ... d_n: disks)
02          if (n = 1) move d_1 = d_n from A to B.
03          else
04                  hanoi(A,C,B,d_1, d_2, ... d_{n-1})
05                  move d_n from A to B.
06                  hanoi(C,B,A,d_1, d_2, ... d_{n-1})
```

Figure 15.6: Solving the Towers of Hanoi problem

The function hanoi breaks up a problem of size n into two problems of size $n-1$. Alert! Warning bells! This can't be good: the sub-problems aren't much smaller than the original problem!

Anyway, hanoi breaks up a problem of size n into two problems of size $n-1$. Other than the two recursive calls, it does only a constant amount of work. So the running time $T(n)$ for the function hanoi would be given by the recursive definition (where c and d are constants):

- $T(1) = c$
- $T(n) = 2T(n-1) + d$

If we unroll this definition, we get

$$\begin{aligned} T(n) &= 2T(n-1)+d \\ &= 2\cdot 2(T(n-2)+d)+d \\ &= 2\cdot 2(2(T(n-3)+d)+d)+d \\ &= 2^3T(n-3)+2^2d+2d+d \\ &= 2^kT(n-k)+d\sum_{i=0}^{k-1}2^i \end{aligned}$$

We'll hit the base case when $k = n-1$. So

$$\begin{aligned} T(n) &= 2^kT(n-k)+d\sum_{i=0}^{k-1}2^i \\ &= 2^{n-1}c+d\sum_{i=0}^{n-2}2^i \\ &= 2^{n-1}c+d(2^{n-1}-1) \\ &= 2^{n-1}c+2^{n-1}d-d \\ &= O(2^n) \end{aligned}$$

15.9 Multiplying big integers

Suppose we want to multiply two integers. Back in the Bad Old Days of slow computers, we would need to multiply moderate-sized integers digit-by-digit. These days, we typically have a CPU that can multiply two moderate-sized numbers (e.g. 16-bit, 32-bit) as a single operation. But some applications (e.g. in cryptography) involve multiplying very, very large integers. Each very long integer must then be broken up into a sequence of 16-bit or 32-bit integers.

Let's suppose that we are breaking up each integer into individual digits, and that we're working in base 2. The recursive multiplication algorithm divides each n-digit input number into two $n/2$-digit halves. Specifically,

suppose that our input numbers are x and y and they each have $2m$ digits. We can then divide them up as

$$x = x_1 2^m + x_0$$
$$y = y_1 2^m + y_0$$

If we multiply x by y in the obvious way, we get

$$xy = A2^{2m} + B2^m + C$$

where $A = x_1 y_1$, $B = x_0 y_1 + x_1 y_0$, and $C = x_0 y_0$. Set up this way, computing xy requires multiplying four numbers with half the number of digits.

In this computation, the operations other than multiplication are fast, i.e. $O(m) = O(n)$. Adding two numbers can be done in one sweep through the digits, right to left. Multiplying by 2^m is also fast, because it just requires left-shifting the bits in the numbers. Or, if you aren't very familiar with binary yet, notice that multiplying by 10^m just requires adding m zeros to the end of the number. Left-shifting in binary is similar.

So, the running time of this naive method has the recursive definition:

- $T(1) = c$
- $T(n) = 4T(n/2) + O(n)$

The closed form for $T(n)$ is $O(n^2)$ (e.g. use unrolling).

The trick to speeding up this algorithm is to rewrite our algebra for computing B as follows

$$B = (x_1 + x_0)(y_1 + y_0) - A - C$$

This means we can compute B with only one multiplication rather that two. So, if we use this formula for B, the running time of multiplication has the recursive definition

- $P(1) = c$
- $P(n) = 3P(n/2) + O(n)$

It's not obvious that we've gained anything substantial, but we have. If we build a recursion tree for P, we discover that the kth level of the tree contains 3^k problems, each involving $n\frac{1}{2^k}$ work. So each non-leaf level requires $n(\frac{3}{2})^k$ work. The sum of the non-leaf work is dominated by the bottom non-leaf level.

The tree height is $\log_2(n)$, so the bottom non-leaf level is at $\log_2(n) - 1$. This level requires $n(\frac{3}{2})^{\log_2 n}$ work. If you mess with this expression a bit, using facts about logarithms, you find that it's $O(n^{\log_2 3})$ which is approximately $O(n^{1.585})$.

The number of leaves is $3^{log_2 n}$ and constant work is done at each leaf. Using log identities, we can show that this expression is also $O(n^{\log_2 3})$.

So this trick, due to Anatolii Karatsuba, has improved our algorithm's speed from $O(n^2)$ to $O(n^{1.585})$ with essentially no change in the constants. If $n = 2^{10} = 1024$, then the naive algorithm requires $(2^{10})^2 = 1,048,576$ multiplications, whereas Katatsuba's method requires $3^{10} = 59,049$ multiplications. So this is a noticable improvement and the difference will widen as n increases.

There are actually other integer multiplication algorithms with even faster running times, e.g. Schoöhage-Strassen's method takes $O(n \log n \log \log n)$ time. But these methods are more involved.

Chapter 16

NP

Some tasks definitely require exponential time. That is, we can not only display an exponential-time algorithm, but we can also prove that the problem cannot be solved in anything less than exponential time. For example, we've seen how to solve the Towers of Hanoi problem using $O(2^n)$ steps. Because the largest disk has to be put first onto the goal peg, and it can't be moved until everything above it is gone, any algorithm has to involve a recursive decomposition similar to what we used in Chapter 15. So it's impossible to improve on the exponential time. The group of problems requiring exponential time is called **EXP**.

However, there is another large class of tasks where the best known algorithm is exponential, but no one has proved that it is impossible to construct a polynomial-time algorithm. This group of problems is known as **NP**.[1] It is an important unsolved question whether the problems in NP really require exponential time.

16.1 Finding parse trees

Suppose we are given an input sentence, such as

[1] NP is short for "non-deterministic polynomial," but that full name makes sense only when you have additional theory background.

> Scores of famished zombies are roaming the north end of campus.

If the sentence contains n words and we have a context-free grammar G, it takes $O(n^3)$ time for a properly-designed parsing algorithm to find a parse tree for the sentence or determine that none exists. However, generating **all** parse trees for such a sentence takes exponential time.

To see this, consider sentences that end in a large number of prepositional phrases, e.g.

> Prof. Rutenbar killed a zombie with a red hat near the Academic Office on the ground floor by the statue with a large sword.

When building the parse tree for such a sentence, each prepositional phrase needs to be associated with either the main verb or with some preceding noun phrase. For example, "with a large sword" might be the instrument used to kill the zombie, in which case it should be linked with "killed" in the parse tree. But it's also possible that the sword is part of the description of the statue, in which case "with a large sword" would be attached to "statue" in the tree. No matter what you do with earlier bits of the sentence, there are at least two possible ways to add each new phrase to the parse tree. So if there are n prepositional phrases, there are at least 2^n possible parse trees.

This example may seem silly. However, similar ambiguities occur with other sorts of modifiers. E.g. in "wild garlic cheese," which is wild, the garlic or the cheese? In normal conversation, sentences tend to be short and we normally have enough context to infer the correct interpretation, and thus build the correct parse tree. However, explosion in the number of possible parse trees causes real issues when processing news stories, which tend to have long sentences with many modifiers.

16.2 What is NP?

Now, let's look at a problem that is in NP: **graph colorability.** Specifically, given a graph G and an integer k, determine whether G can be colored with k colors. For small graphs, it seems like this problem is quite simple. Moreover,

heuristics such as the greedy algorithm from Chapter 11 work well for many larger graphs found in practical applications. However, if $k \geq 3$, for unlucky choices of input graph, all known algorithms take time that is exponential in the size of the graph. However, we don't know whether exponential time is really required or whether there is some clever trick that would let us build a faster algorithm.

To make the theoretical definitions easier to understand, let's upgrade our colorability algorithm slightly. As stated above, our colorability algorithm needs to deliver a yes/no answer. Let's ask our algorithm to, in addition, deliver a justification that its answer is correct.[2] For example, for graph colorability, a "yes" answer would also come with an assignment of the k colors to the nodes of G. It's easy to verify that such an assignment is a legal coloring using the specified number of colors and, thus, that the "yes" answer is correct.

The big difference between graph colorability and parse tree generation is this ability to provide succinct, easily checked solutions with justifications. Specifically, a computational problem is in the set **NP** when an algorithm can provide justifications of "yes" answers, where each justification can be checked in polynomial time. That is, the justification checking time is polynomial in the size of the input to the algorithm (e.g. the graph G in our colorability problem). This implies that the justification itself must be succinct, because the checker must read through it. Graph colorability is in NP. Parse tree generation is in EXP but not in NP, because even the raw answer without justification has exponential length.

For graph colorability, it only seems to be possible to provide succinct justifications for "yes" answers. For "no" answers, we can sometimes give a short and convincing explanation. But, for difficult graphs, we might have to walk through all exponentially-many possible assignments of colors to nodes, showing why none of them is a legal coloring. So there doesn't seem to be a general-purpose way to provide succinct justifications for negative answers.

Obviously, the choice of which answer is "yes" vs. "no" depends on how we phrase the problem. Consider the **non-colorability problem**: given a graph G and an integer k, is it impossible to color G with k colors? For this problem, we can give succinct justifications for "no" answers but apparently

[2]Like we force you folks to do, when answering questions on theory exams!

not for "yes" answers. Problems for which we can give polynomial-time checkable justifications for negative answers are in the set **co-NP**.

There are some difficult problems where we can provide justifications for both negative and positive answers. For example, there are efficient ways to verify whether an integer is prime or not, but factoring integers is much harder. For example, we do not know of any polynomial-time way to determine whether an integer m has a factor in the range $[2, n]$.[3] However, having gone to the considerable pain of factoring m, we can provide succinct, easily-verified justifications for both "yes" and "no" answers using the prime factorization of m. So this problem is in both NP and co-NP.

Algorithms that can be solved in polynomial time are in the set P. P is a subset of both NP and co-NP. Theoreticians strongly suspect that these three sets—P, NP, and co-NP—are all different. However, no one has found a way to prove this. So it is possible that all three sets are equal, i.e. that we've missed some big trick that would let us build more efficient algorithms for the problems in NP and co-NP.

16.3 Circuit SAT

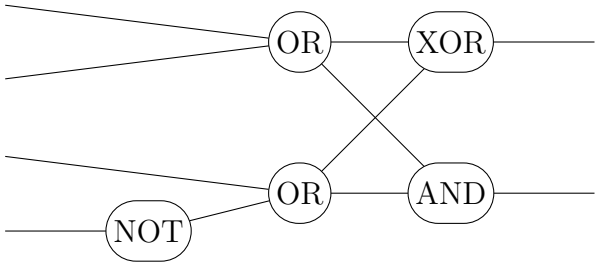
Another interesting problem in NP is satisfiability of boolean circuits. A **boolean circuit** is a type of graph consisting of a set of **gates** connected together by wires. Each wire carries one of the two values 0 (false) and 1 (true). The label on each gate describes how the value on its output wires (all output wires have the same value) is determined by the values on its input wires. For example, the output wires of an AND gate have value 1 if both input wires have value 1, and the output wires have value 0 otherwise.

Like a logical operator, the behavior of a gate can be described by a truth table. For simplicity, we'll restrict ourselves to gates with one or two inputs. A boolean circuit with any possible behavior can be built using the basic three operators (AND, OR, NOT). However, in practical applications, it's common to also use other gates, e.g. XOR (exclusive OR), NAND (negation of AND).

For example, here is a very simple boolean circuit. In this picture, inputs

[3] That is, polynomial in the number of digits in the input numbers.

are on the left and outputs on the right. For ease of understanding by software and theory folks, the type of each gate is indicated with the name of the operator. For serious circuit design, you'd need to memorize the special node shapes commonly used for different gate types.



When designing circuits for practical applications, it's important to ensure that the circuit really does what it is supposed to do. Or, at least, that it doesn't make certain critical errors that would lead to tragedy. For example, the circuit controlling a 4-way intersection must ensure that two crossing lanes of traffic cannot both have a green light at the same time. A slow-moving 6-legged robot should never raise more than three of its legs at a time. A heating system should not open the gas jet if the pilot light is off. Significant portions of such controllers are well-modelled as boolean circuits.

To find significant design flaws in a boolean circuit, we need to determine whether there is a set of n input values could create one of the forbidden patterns of output values. This problem is called *circuit satisfiability*. There is an obvious exponential algorithm for circuit satisfiability: if we have n input wires, generate all 2^n possible input patterns and see what happens on the output wires. That's too slow, so circuit testers must be content with partial testing. However, if we determine that the bad output is possible, we can provide a succinct easily-verified justification: show one pattern of input values that gives rise to the bad output. So this problem is in NP.

Sadly, what we'd really like for practical applications would be easily-

verified justifications of negative answers. Or, equivalently, positive answers to the the complementary problem of circuit safety: is the circuit safe from bad outputs? This seems to be much harder than justifications for circuit satisfiability, because proving a negative answer seems to require walking through all exponentially many input patterns and showing why none of them produces the bad output pattern. So circuit safety is in co-NP but apparently not in NP.

16.4 What is NP complete?

An interesting, and very non-obvious, fact about the harder problems in NP is that they share a common fate. It can be proven that finding a polynomial-time algorithm for either circuit satisfiability or graph colorability would imply that *all* problems in NP have polynomial-time algorithms. Such problems are called *NP-complete*.

Other examples of NP complete problems include

- Propositional logic satisfiability: given an expression in propositional logic, is there an assignment of true/false values to the variables which would make the logic expression true?
- Clique: given an integer n and a graph G, does G contain a copy of K_n? (Brute force search is polynomial in the size of G but exponential in n.)
- The Marker Making problem from Section 10.1, when rotation of parts is not allowed.
- Vertex cover: given a graph G, find a set S of nodes in G such that each edge of G has at least one of its endpoints in S.
- The Travelling Salesman Problem: find the shortest path through a set of cities, visiting each city exactly once.

These problems typically remain NP-complete even when stripped down to their bare essentials. For example, we can restrict our circuits to have

only a single output. We can restrict our logic expression to be the AND of some number of 3-variable OR expressions. Or we can use only the primitive operators/gates AND, OR, and NOT. We can restrict the shapes in Marker Making to be rectangular. Only the most grossly simplified versions of these problems fail to be NP-complete.

There are two techniques for showing that a problem is NP-complete. First, we can show directly that the problem is so general that it can simulate any other problem in NP. The details are beyond the scope of the book. But notice that circuits are the backbone of computer construction and logic is the backbone for constructing mathematics, so it makes sense that both mechanisms are general and powerful. Once we have proved that some key problems are NP-complete, we can then show that other problems are NP-complete by showing that the new problem can simulate the basic primitives of a problem already known to be NP-complete. For example, we might simulate logic expressions using colored graph machinery.

16.5 Variation in notation

A justification for a positive answer can also be called a "proof," "certificate," or "witness."

Chapter 17

Proof by Contradiction

This chapter covers proof by contradiction. This is a powerful proof technique that can be extremely useful in the right circumstances. We'll need this method in Chapter 20, when we cover the topic of uncountability. However, contradiction proofs tend to be less convincing and harder to write than direct proofs or proofs by contrapositive. So this is a valuable technique which you should use sparingly.

17.1 The method

In proof by contradiction, we show that a claim P is true by showing that its negation $\neg P$ leads to a contradiction. If $\neg P$ leads to a contradiction, then $\neg P$ can't be true, and therefore P must be true. A contradiction can be any statement that is well-known to be false or a set of statements that are obviously inconsistent with one another, e.g. n is odd and n is even, or $x < 2$ and $x > 7$.

Proof by contradiction is typically used to prove claims that a certain type of object cannot exist. The negation of the claim then says that an object of this sort **does** exist. The existence of an object with specified properties is often a good starting point for a proof. For example:

Claim 51 *There is no largest even integer.*

> Proof: Suppose not. That is, suppose that there were a largest even integer. Let's call it k.
>
> Since k is even, it has the form $2n$, where n is an integer. Consider $k+2$. $k+2 = (2n)+2 = 2(n+1)$. So $k+2$ is even. But $k+2$ is larger than k. This contradicts our assumption that k was the largest even integer. So our original claim must have been true. □

The proof starts by informing the reader that you're about to use proof by contradiction. The phrase "suppose not" is one traditional way of doing this. Next, you should spell out exactly what the negation of the claim is. Then use mathematical reasoning (e.g. algebra) to work forwards until you deduce some type of contradiction.

17.2 $\sqrt{2}$ is irrational

One of the best known examples of proof by contradiction is the proof that $\sqrt{2}$ is irrational. This proof, and consequently knowledge of the existence of irrational numbers, apparently dates back to the Greek philosopher Hippasus in the 5th century BC.

We defined a rational number to be a real number that can be written as a fraction $\frac{a}{b}$, where a and b are integers and b is not zero. If a number can be written as such a fraction, it can be written as a fraction in lowest terms, i.e. where a and b have no common factors. If a and b have common factors, it's easy to remove them.

Also, we proved (above) that, for any integer k, if k is odd then k^2 is odd. So the contrapositive of this statement must also be true: (*) if k^2 is even then k is even.

Now, we can prove our claim:

> Suppose not. That is, suppose that $\sqrt{2}$ were rational.
>
> Then we can write $\sqrt{2}$ as a fraction $\frac{a}{b}$ where a and b are integers with no common factors.

Since $\sqrt{2} = \frac{a}{b}$, $2 = \frac{a^2}{b^2}$. So $2b^2 = a^2$.

By the definition of even, this means a^2 is even. But then a must be even, by (*) above. So $a = 2n$ for some integer n.

If $a = 2n$ and $2b^2 = a^2$, then $2b^2 = 4n^2$. So $b^2 = 2n^2$. This means that b^2 is even, so b must be even.

We now have a contradiction. a and b were chosen not to have any common factors. But they are both even, i.e. they are both divisible by 2.

Because assuming that $\sqrt{2}$ was rational led to a contradiction, it must be the case that $\sqrt{2}$ is irrational. □

17.3 There are infinitely many prime numbers

Contradiction also provides provides a nice proof of a classic theorem about prime numbers, dating back to Euclid, who lived around 300 B.C.

Euclid's Theorem: There are infinitely many prime numbers.

This is a lightly disguised type of non-existence claim. The theorem could be restated as "there is no largest prime" or "there is no finite list of all primes." So this is a good situation for applying proof by contradiction.

Proof: Suppose not. That is, suppose there were only finitely many prime numbers. Let's call them p_1, p_2, up through p_n.

Consider $Q = p_1 p_2 \cdots p_n + 1$.

If you divide Q by one of the primes on our list, you get a remainder of 1. So Q isn't divisible by any of the primes p_1, p_2, up through p_n. However, by the Fundamental Theorem of Arithmetic, Q must have a prime factor (which might be either itself or some smaller number). This contradicts our assumption that p_1, p_2,$\ldots p_n$ was a list of all the prime numbers. □

Notice one subtlety. We're not claiming that Q must be prime. Rather, we're making the much weaker claim that Q isn't divisible by any of the first n primes. It's possible that Q might be divisible by another prime larger than p_n.

17.4 Lossless compression

A final example concerns file compression. A file compression algorithm attempts to reduce the size of files, by transforming each input file to an output file with fewer bits. A **lossless** algorithm allows you to reconstruct the original file exactly from its compressed version, whereas a **lossy** algorithm only allows you to reconstruct an approximation to the original file. Or, said another way, a lossless algorithm must convert input files to output files in a one-to-one-manner, so that two distinct input files are never compressed to the same output file.

Claim 52 *A lossless compression algorithm that makes some files smaller must make some (other) files larger.*

> Proof: Suppose not. That is, suppose that we had a lossless compression algorithm A that makes some files smaller and does not make any files larger.
>
> Let x be the shortest file whose compressed size is smaller than its original size. (If there are two such files of the same length, pick either at random.) Suppose that the input size of x is m characters.
>
> Suppose that S is the set of distinct files with fewer than m characters. Because x shrinks, A compresses x to a file in S. Because no files smaller than x shrink, each file in S compresses to a file (perhaps the same, perhaps different) in S.
>
> Now we have a problem. A is supposed to be lossless, therefore one-to-one. But A maps a set containing at least $|S| + 1$ files to a set containing $|S|$ files, so the Pigeonhole Principle states that two input files must be mapped to the same output file. This is a contradiction.

So, on the face of it, lossless file compression algorithms can't win. How do they work so well in practice? One secret is that compression algorithms can ensure that file sizes never increase much. If a file would increase in size, the algorithm stores the original version unchanged, preceded with a one-bit marker. This bounds the potential damage if we encounter a "bad" input file.

The second secret is that commonly-occurring files are not created at random but have definite patterns. Text files contain natural language text. Digitized images contain values that tend to change gradually. Compression algorithms are tuned so that common types of files shrink. The fact that some files might get bigger isn't a serious practical problem if those files are unlikely to occur on your disk.

17.5 Philosophy

Proof by contradiction strikes many people as mysterious, because the argument starts with an assumption known to be false. The whole proof consists of building up a fantasy world and then knocking it down. Although the method is accepted as valid by the vast majority of theoreticians, these proofs are less satisfying than direct proofs which construct the world as we believe it to be. The best mathematical style avoids using proof by contradiction except when it will definitely result in a much simpler argument.

There is, in fact, a minority but long-standing thread within theoretical mathematics, called "constructive mathematics," which does not accept this proof method. They have shown that most of standard mathematics can be re-built without it. For example, the irrationality of $\sqrt{2}$ can be proved constructively, by showing that there is an error separating $\sqrt{2}$ from any chosen fraction $\frac{a}{b}$.

Chapter 18

Collections of Sets

So far, most of our sets have contained atomic elements (such as numbers or strings) or tuples (e.g. pairs of numbers). Sets can also contain other sets. For example, $\{\mathbb{Z}, \mathbb{Q}\}$ is a set containing two infinite sets. $\{\{a, b\}, \{c\}\}$ is a set containing two finite sets. In this chapter, we'll see a variety of examples involving sets that contain other sets. To avoid getting confused, we'll use the term **collection** to refer to a set that contains other sets, and use a script letter for its variable name.

18.1 Sets containing sets

Sets containing sets arise naturally when an application needs to consider some or all of the subsets of a base set $\mathscr{A}$. For example, suppose that we have a set of 6 students:

$$\mathscr{A} = \{\text{Ian}, \text{Chen}, \text{Michelle}, \text{Emily}, \text{Jose}, \text{Anne}\}$$

We might divide $\mathscr{A}$ up into non-overlapping groups based on what dorm they live in, to get the collection:

$$\mathscr{B} = \{\{\text{Ian}, \text{Chen}, \text{Jose}\}, \{\text{Anne}\}, \{\text{Michelle}, \text{Emily}\}\}$$

We could also construct a set of overlapping groups, each containing students who play a common musical instrument (e.g. perhaps Michelle and Chen both play the oboe). The collection of these groups might look like:

$$\mathscr{D} = \{\{\text{Ian}, \text{Emily}, \text{Jose}\}, \{\text{Anne}, \text{Chen}, \text{Ian}\}, \{\text{Michelle}, \text{Chen}\}, \{\text{Ian}\}\}$$

Or we could try to list all ways that we could choose a 3-person committee from this set of students, which would be a rather large collection containing elements such as $\{\text{Ian}, \text{Emily}, \text{Jose}\}$ and $\{\text{Ian}, \text{Emily}, \text{Michelle}\}$.

When a collection like $\mathscr{B}$ is the domain of a function, the function maps an entire subset to an output value. For example, suppose we have a function $f : \mathscr{B} \to \{\text{dorms}\}$. Then f would map each set of students to a dorm. E.g. $f(\{\text{Michelle}, \text{Emily}\}) = \text{Babcock}$.

The value of a function on a subset can depend in various ways on whatever is in the subset. For example, suppose that we have a collection

$$\mathscr{D} = \{\{-12, 7, 9, 2\}, \{2, 3, 7\}, \{-10, -3, 10, 4\}, \{1, 2, 3, 6, 8\}\}$$

We might have a function $g : \mathscr{D} \to \mathbb{R}$ which maps each subset to some descriptive statistic. For example, g might map each subset to its mean value. And then we would have $g(\{-12, 7, 9, 2\}) = 1.5$ and $g(\{1, 2, 3, 6, 9\}) = 4.2$.

When manipulating sets of sets, it's easy to get confused and "lose" a layer of structure. To avoid this, imagine each set as a box. Then the collection $\mathscr{F} = \{\{a, b\}, \{c\}, \{a, p, q\}\}$ is a box containing three boxes. One of the inside boxes contains a and b, the other contains c, and the third contains a, p, and q. So the cardinality of $\mathscr{F}$ is three.

The empty set, like any other set, can be put into another set. So $\{\emptyset\}$ is a set containing the empty set. Think of it as a box containing an empty box. The set $\{\emptyset, \{3, 4\}\}$ has two elements: the empty set and the set $\{3, 4\}$.

18.2 Powersets and set-valued functions

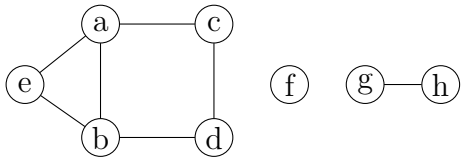
If A is a set, the powerset of A (written $\mathbb{P}(A)$ is the collection containing all subsets of A. For example, suppose that $A = \{1, 2, 3\}$. Then

$$\mathbb{P}(A) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

Suppose A is finite and contains n elements. When forming a subset, we have two choices for each element x: include x in the subset or don't include it. The choice for each element is independent of the choice we make for the other elements. So we have 2^n ways to form a subset and, thus, the powerset $\mathbb{P}(A)$ contains 2^n elements.

Notice that the powerset of A always contains the empty set, regardless of what's in A. As a consequence, $\mathbb{P}(\emptyset) = \{\emptyset\}$.

Powersets often appear as the co-domain of functions which need to return a set of values rather than just a single value. For example, suppose that we have the following graph whose set of nodes is $V = \{a, b, c, d, e, f, g, h\}$.

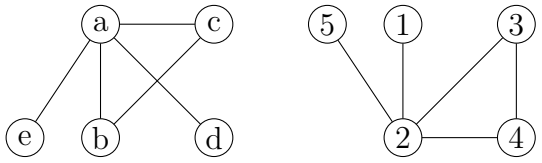


Now, let's define the function n so that it takes a node as input and returns the neighbors of that node. A node might have one neighbor, but it could have several, and it might have no neighbors. So the outputs of n can't be individual nodes. They must be sets of nodes. For example, $n(a) = \{b, c, e\}$ and $N(f) = \emptyset$. It's important to be consistent about the output type of n: it always returns a set. So $n(g) = \{h\}$, **not** $n(g) = h$.

Formally, the domain of n is V and the co-domain is $\mathbb{P}(V)$. So the type signature of n would be $n : V \to \mathbb{P}(V)$.

Suppose we have the two graphs shown below, with sets of nodes $X =$

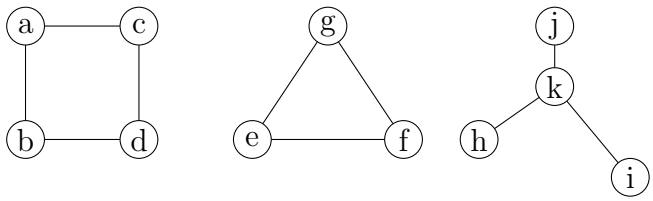
$\{a, b, c, d, e\}$ and $Y = \{1, 2, 3, 4, 5\}$. And suppose that we're trying to find all the possible isomorphisms between the two graphs. We might want a function f that retrieves likely corresponding nodes. For example, if p is a node in X, then $f(p)$ might be the set of nodes in Y with the same degree as p.



f can't return a single node, because there might be more than one node in Y with the same degree. Or, if the two graphs aren't isomorphic, no nodes in Y with the same degree. So we'll have f return a set of nodes. For example, $f(e) = \{1, 5\}$ and $f(a) = \{2\}$. The co-domain of f will need to be $\mathbb{P}(Y)$. So, to declare f, we'd write $f : X \to \mathbb{P}(Y)$.

18.3 Partitions

When we divide a base set A into non-overlapping subsets which include every element of A, the result is called a *partition* of A. For example, suppose that A is the set of nodes in the following graph. The partition $\{\{a, b, c, d\}, \{e, f, g\}, \{h, i, j, k\}\}$ groups nodes into the same subset if they belong to the same connected component.



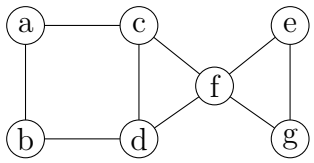
Notice that being in the same connected component is an equivalence relation on the nodes of a graph. In general, each equivalence relation corre-

sponds to a partition of its base set, and vice versa. Each set in the partition is exactly one of the equivalence classes of the relation. For example, congruence mod 4 corresponds to the following partition of the integers:

$$\{\{0, 4, -4, 8, -8, \ldots\}, \{1, 5, -3, 9, -7, \ldots\},$$
$$\{2, 6, -2, 10, -6, \ldots\}, \{3, 7, -1, 11, -5, \ldots\}\}$$

We could also write this partition as $\{[0], [1], [2], [3]\}$ since each equivalence class is a set of numbers.

Collections of subsets don't always form partitions. For example, consider the following graph G.



Suppose we collect sets of nodes in G that form a cycle. We'll get the following set of subsets. This isn't a partition because some of the subsets overlap.

$$\{\{f, c, d\}, \{a, b, c, d\}, \{a, b, c, d, f\}, \{f, e, g\}\}$$

Formally, a partition of a set A is a set of non-empty subsets of A which cover all the elements of A and which don't overlap. So, if the subsets in the partition are $A_1, A_2, \ldots A_n$, then they must satisfy three conditions:

1. covers all of A: $A_1 \cup A_2 \cup \ldots \cup A_n = A$
2. non-empty: $A_i \neq \emptyset$ for all i
3. no overlap: $A_i \cap A_j = \emptyset$ for all $i \neq j$.

It's possible for a partition of an infinite set A to contain infinitely many subsets. For example, we can partition the integers into subsets each of which contains integers with the same magnitude:

$$\{\{0\}, \{1, -1\}, \{2, -2\}, \{3, -3\}, \ldots\}$$

We need more general notation to cover the possibility of an infinite partition. Suppose that $\mathscr{C}$ is a partition of A. Then $\mathscr{C}$ must satisfy the following conditions:

1. covers all of A: $\bigcup_{X \in \mathscr{C}} X = A$
2. non-empty: $X \neq \emptyset$ for all $X \in \mathscr{C}$
3. no overlap: $X \cap Y = \emptyset$ for all $X, Y \in \mathscr{C}$, $X \neq Y$

The three defining conditions of an equivalence relation (reflexive, symmetric, and transitive) were chosen so as to force the equivalence classes to be a partition. Relations without one of these properties would generate "equivalence classes" that might be empty, have partial overlaps, and so forth.

18.4 Combinations

In many applications, we have an n-element set and need to count all subsets of a particular size k. A subset of size k is called a k-combination. Notice the difference between a permutation and a combination: we care about the order of elements in a permutation but not in a combination.

For example, how many ways can I select a 7-card hand from a 60-card deck of Magic cards (assuming no two cards are identical)?[1]

One way to analyze this problem is to figure out how many ways we can select an ordered list of 7 cards, which is $P(60, 7)$. This over-counts the

[1] Ok, ok, for those of you who actually play Magic, decks do tend to contain identical land cards. But maybe we are using lots of special lands or perhaps we'll treat cards with different artwork as different.

number of possibilities, so we have to divide by the number of different orders in which the same 7-cards might appear. That's just 7!. So our total number of hands is $\frac{P(60,7)}{7!}$ This is $\frac{60 \cdot 59 \cdot 58 \cdot 57 \cdot 56 \cdot 55 \cdot 54}{7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2}$. Probably not worth simplifying or multiplying this out unless you really have to. (Get a computer to do it.)

In general, suppose that we have a set S with n elements and we want to choose an unordered subset of k elements. We have $\frac{n!}{(n-k)!}$ ways to choose k elements in some particular order. Since there are $k!$ ways to put each subset into an order, we need to divide by $k!$ so that we will only count each subset once. So the general formula for the number of possible subsets is $\frac{n!}{k!(n-k)!}$.

The expression $\frac{n!}{k!(n-k)!}$ is often written $C(n,k)$ or $\binom{n}{k}$. This is pronounced "n choose k." It is also sometimes called a "binomial coefficient," for reasons that will become obvious shortly. So the shorthand answer to our question about magic cards would be $\binom{60}{7}$.

Notice that $\binom{n}{r}$ is only defined when $n \geq r \geq 0$. What is $\binom{0}{0}$? This is $\frac{0!}{0!0!} = \frac{1}{1 \cdot 1} = 1$.

18.5 Applying the combinations formula

The combinations formula is often used when we want to select a set of locations or positions to contain a specific value. For example, recall that a bit string is a string of 0's and 1's. Suppose we want to figure out how many 16-digit bit strings contain exactly 5 zeros. Let's think of the string as having 16 positions. We need to choose 5 of these to be the positions containing the zeros. We can how apply the combinations formula: we have $\binom{16}{5}$ ways to select these 5 positions.

To take a slightly harder example, let's figure out how many 10-character strings from the 26-letter ASCII alphabet contain no more than 3 A's. Such strings have to contain 0, 1, 2, or 3 A's. To find the number of strings containing exactly three A's, we first pick three of the 10 positions to contain the A's. There are $\binom{10}{3}$ ways to do this. Then, we have seven positions to fill with our choice of any character except A. We have 25^7 ways to do that. So our total number of strings with 3 A's is $\binom{10}{3}25^7$.

To get the total number of strings, we do a similar analysis to count

the strings with 0, 1, and 2 A's. We then add up the counts for the four possibilities to get a somewhat messy final answer for the number of strings with 3 or fewer A's:

$\binom{10}{3}25^7 + \binom{10}{2}25^8 + \binom{10}{1}25^9 + 25^{10}$

18.6 Combinations with repetition

Suppose I have a set S and I want to select a group of objects of the types listed in S, but I'm allowed to pick more than one of each type of object. For example, suppose I want to pick 6 plants for my garden and the set of available plants is $S = \{\text{thyme}, \text{oregano}, \text{mint}\}$. The garden store can supply as many as I want of any type of plant. I could pick 3 thyme and 3 mint. Or I could pick 2 thyme, 1 oregano, and 3 mint.

There's a clever way to count the possibilities here. Let's draw a picture of a selection as follows. We'll group all our thymes together, then our oreganos, then our mints. Between each pair of groups, we'll put a cardboard separator #. So 2 thyme, 1 oregano, and 3 mint looks like

T T # O # M M M

And 3 thyme and 3 mint looks like

T T T ## M M M

But this picture is redundant, since the items before the first separator are always thymes, the ones between the separators are oreganos, and the last group are mints. So we can simplify the diagram by using a star for each object and remembering their types implicitly. Then 2 thyme, 1 oregano, and 3 mint looks like

** # * # ***

And 3 thyme and 3 mint looks like

*** ## ***

To count these pictures, we need to count the number of ways to arrange 6 stars and two #'s. That is, we have 8 positions and need to choose 2 to fill with #'s. In other words, $\left(\begin{array}{c} 8 \\ 2 \end{array} \right)$.

In general, suppose we are picking a group of k objects (with possible duplicates) from a list of n types. Then our picture will contain k stars and $n-1$ #'s. So we have $k+n-1$ positions in the picture and need to choose $n-1$ positions to contain the #'s. So the number of possible pictures is $\left(\begin{array}{c} k+n-1 \\ n-1 \end{array} \right)$.

Notice that this is equal to $\left(\begin{array}{c} k+n-1 \\ k \end{array} \right)$ because we have an identity that says so (see above). We could have done our counting by picking a subset of k positions in the diagram that we would fill with stars (and then the rest of the positions will get the #'s).

If wanted to pick 20 plants and there were five types available, I would have $\left(\begin{array}{c} 24 \\ 4 \end{array} \right) = \left(\begin{array}{c} 24 \\ 20 \end{array} \right)$ options for how to make my selection. $\left(\begin{array}{c} 24 \\ 4 \end{array} \right) = \frac{24 \cdot 23 \cdot 22 \cdot 21}{4 \cdot 3 \cdot 2} = 23 \cdot 22 \cdot 21$.

18.7 Identities for binomial coefficients

There are a large number of useful identities involving binomial coefficients, most of which you can look up as you need them. Two really basic ones are worth memorizing. First, a simple consequence of the definition of $\binom{n}{k}$ is that

$$\binom{n}{k} = \binom{n}{n-k}$$

Pascal's identity also shows up frequently. It states that

(Pascal's identity) $\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}$

This is not hard to prove from the definition of $\binom{n}{k}$. To remember it, suppose that S is a set with $n+1$ elements. The lefthand side of the equation is the number of k-element subsets of S.

Now, fix some element a in S. There are two kinds of k-element subsets: (1) those that don't contain a and (2) those that do contain a. The first term on the righthand side counts the subsets in group (1): all k-element subsets of $S - \{a\}$. The second term on the righthand side counts the $k-1$-element subsets of $S - \{a\}$. We then add a to each of these to get the subsets in group (2).

If we have Pascal's identity, we can give a recursive definition for the binomial coefficients, for all natural numbers n and k with $k \leq n$.

Base: For any natural number k, $\binom{n}{0} = \binom{n}{n} = 1$.

Induction: $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$, whenever $k < n$

18.8 Binomial Theorem

Remember that a *binomial* is a sum of two terms, e.g. $(x + y)$. Binomial coefficients get their name from the following useful theorem about raising a binomial to an integer power:

Claim 53 *(Binomial Theorem) Let x and y be variables and let n be any natural number. Then*

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

When we expand the product $(x + y)^n$, each term is the product of n variables, some x's and the rest y's. For example, if $n = 5$, one term is $yxyyx$. So each term is an ordered list of x's and y's.

We can think of our large set of terms as partitioned into subsets, each containing terms with the same number of x's. For example, the set of terms with two x's would be

$$\begin{aligned}[xxyyy] &= \{xxyyy, xyxyy, xyyxy, xyyyx, yxxyy,\\ &\quad yxyxy, yxyyx, yyxxy, yyxyx, yyyxx\}\end{aligned}$$

When we collect terms, the coefficient for each term will be the size of this set of equivalent terms. E.g. the coefficient for x^2y^3 is 10, because $[xxyyy]$ contains 10 elements. To find the coefficient for $x^{n-k}y^k$, we need to count how many ways we can make a sequence of n variable names that contains k y's and $n-k$ x's. This amounts to picking a subset of k elements from a set of n positions in the sequence. In other words, there are $\binom{n}{k}$ such terms.

18.9 Variation in notation

We've used the notation $\mathbb{P}(A)$ for the powerset of A. Another common notation is 2^A.

We've used the term "collection" to refer to sets containing other sets. A collection is just a special type of set, so it's also ok to just call them sets.

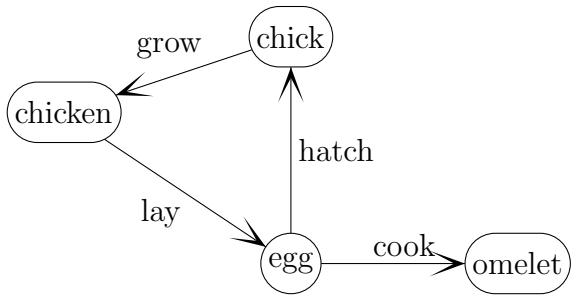
Chapter 19

State Diagrams

In this chapter, we'll see state diagrams, an example of a different way to use directed graphs.

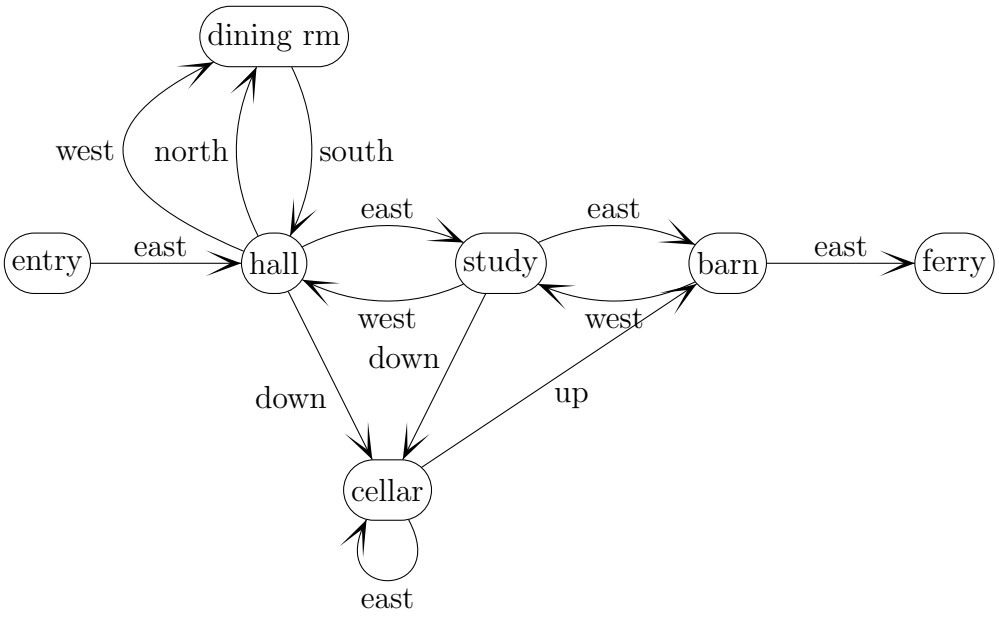
19.1 Introduction

State diagrams are a type of directed graph, in which the graph nodes represent states and labels on the graph edges represent actions. For example, here is a state diagram representing the life cycle of a chicken:



The label on the edge from state A to state B indicates what action happens as the system moves from state A to state B. In many applications, all the transitions involve one basic type of action, such as reading a character

or going though a doorway. In that case, the diagram might simply indicate the details of this action. For example, the following diagram for a multi-room computer game shows only the direction of motion on each edge.



Walks (and therefore paths and cycles) in state diagrams must follow the arrow directions. So, for example, there is no path from the ferry to the study. Second, an action can result in no change of state, e.g. attempting to go east from the cellar. Finally, two different actions may get you to the same new state, e.g. going either west or north from the hall gets you to the dining room.

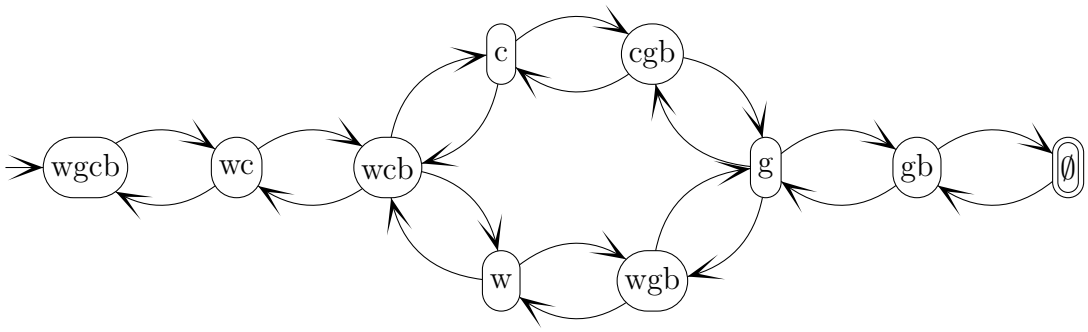
Remember that the full specification of a walk in a graph contains both a sequence of nodes and a sequence of edges. For state diagrams, these correspond to a sequence of states and a sequence of actions. It's often important to include both sequences, both to avoid ambiguity and because the states and actions may be important to the end user. For example, for one walk from the hall to the barn, the full state and action sequences look like:

states:	hall	dining room	hall	cellar	barn
actions:		west	south	down	up

19.2 Wolf-goat-cabbage puzzle

State diagrams are often used to model puzzles or games. For example, one famous puzzle involves a farmer taking a wolf, a goat, and a cabbage to market. To do this, he must cross from the east to the west side of a river using a boat that can only carry him plus one of his three possessions. He cannot leave the wolf and goat together unsupervised, nor the goat and the cabbage, because one will eat the other.

We can represent each state of this system by listing the objects that are on the east bank: w is the wolf, g is the goat, c is the cabbage, and b is the boat. States like wc and wgb are legal, but wg would not be a legal state. The diagram of legal states then looks as follows:

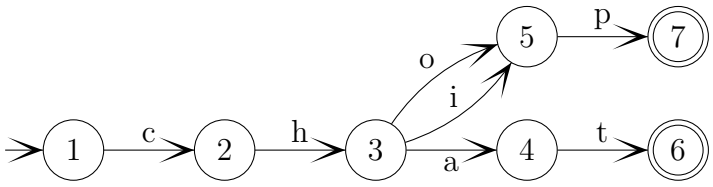


In this diagram, actions aren't marked on the edges: you're left to infer the action from the change in state. The start state (wgcb) where the system begins is marked by showing an arrow leading into it. The end state (∅) where nothing is left on the east bank is marked with a double ring.

In this diagram, it is possible for a (directed) walk to loop back on itself, repeating states. So there are two shortest solutions to this puzzle, but also an infinite number of other solutions that involve undoing and then redoing some piece of work.

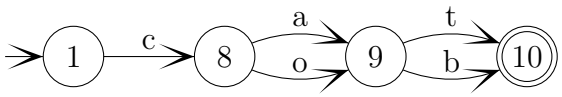
19.3 Phone lattices

Another standard application for state diagrams is in modelling pronunciations of words for speech recognition. In these diagrams, known as *phone lattices*, each edge represents the action of reading a single sound (a *phone*) from the input speech stream. To make our examples easy to read, we'll pretend that English is spelled phonetically, i.e. each letter of English represents exactly one sound/phone.[1] For example, we could model the set of words {`chat`, `chop`, `chip`} using the diagram:



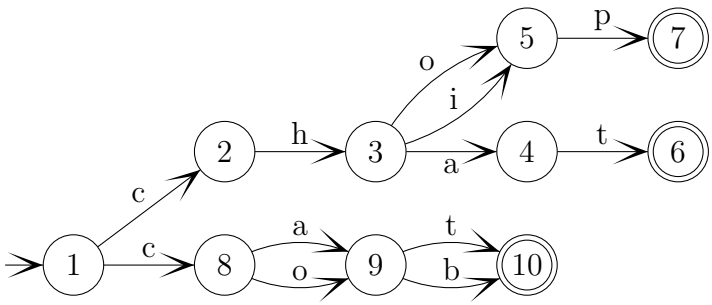
As in the wolf-goat-cabbage puzzle, we have a clearly defined start state (state 1, marked with an incoming arrow). This time there are two end states (6 and 7) marked with double rings. It's conventional that there is only one start state but there may be several end states.

Another phone lattice might represent the set of words {`cat`, `cot`, `cab`, `cob`}.



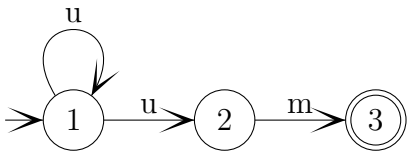
We can combine these two phone lattices into one large diagram, representing the union of these two sets of words:

[1] This is, of course, totally wrong. But the essential ideas stay the same when you switch to a phonetic spelling of English.



Notice that there are two edges leading from state 1, both marked with the phone c. This indicates that the user (e.g. a speech understanding program) has two options for how to handle a c on the input stream. If this was inconvenient for our application, it could be eliminated by merging states 2 and 8.

Occasionally, it's useful for a phone lattice to contain a loop. E.g. people often drag out the pronunciation of the word "um." So the following represents all words of the form `uu*m`, i.e. `um`, `uum`, `uuuuuum`, etc.



Many state diagrams are passive representations of a set of possibilities. For example, a room layout for a computer game merely shows what walks are possible; the player makes the decisions about which walk to take. Phone lattices are often used in a more active manner, as a very simple sort of computer called a *finite automaton*. The automaton reads an input sequence of characters, following the edges in the phone lattice. At the end of the input, it reports whether it has or hasn't successfully reached an end state.

19.4 Representing functions

To understand how state diagrams might be represented mathematically and/or stored in a computer, let's first look in more detail at how functions are represented. A function $f : A \to B$ associates each input value from A with an output value from B. So we can represent f mathematically as a set of input/output pairs (x, y), where x comes from A and y comes from B. For example, one particular function from $\{a, b, c\}$ to $\{1, 2, 3, 4\}$ might be represented as

$$\{(a, 4), (b, 1), (c, 4)\}$$

In a computer, this information might be stored as a linked list of pairs. Each pair might be a short list, or an object or structure. However, finding a pair in such a list requires time proportional to the length of the list. This can be very slow if the list is long i.e. if the domain of the function is large.

Another option is to convert input values to small integers. For example, in C, lowercase integer values (as in the set A above) can be converted to small integers by subtracting 97 (the ASCII code for a). We can then store the function's input/output pairs using an array. Each array position represents an input value. Each array cell contains the corresponding output value.

19.5 Transition functions

Formally, we can define a state diagram to consist of a set of states S, a set of actions A, and a *transition function* δ that maps out the edges of the graph, i.e. shows how actions result in state changes. Each edge of the state diagram shows what happens when you are in a particular state s and execute some action a, i.e. which new state(s) could you be in.

So an input to δ is a pair (s, a) of a state and an action. Each output of δ is a set of states. So the type signature of δ looks like $\delta : S \times A \to \mathbb{P}(S)$.

For many input pairs, δ produces a set containing a single state. So why does δ produce a set of states rather than a single state? This is for two reasons. First, in some state diagrams, it might not be possible to execute

certain actions from certain states, e.g. you can't go up from the study in our room layout example. δ returns $\emptyset$ in such cases. Second, as we saw in the phone lattice example, it is sometimes convenient to allow the user or the computer system several choices. In this case, δ returns a set of possibilities for the new state.

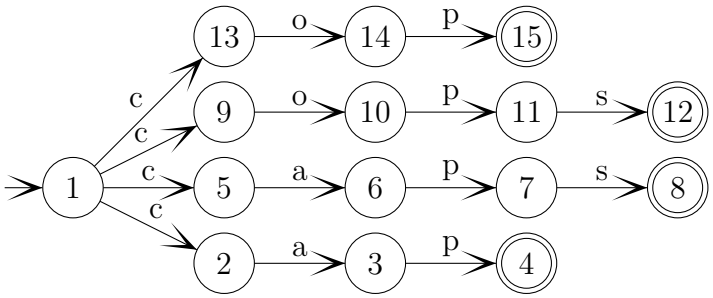
The hard part of implementing state diagrams in a computer program is storing the transition function. We could build a 2D array, whose cells represent all the pairs (s, a). Each cell would then contain a list of output states. This wouldn't be very efficient, however, because state diagrams tend to be *sparse*: most state/action pairs don't produce any new state.

One better approach is to build a 1D array of states. The cell for each state contains a list of actions possible from that state, together with the new states for each action. For example, in our final phone lattice, the entry for state 1 would be $((c, (2, 8)))$ and the entry for state 3 would be $((o, (5)), (i, (5)), (a, (4)))$. This *adjacency list* style of storage is much more compact, because we are no longer wasting space representing the large number of impossible actions.

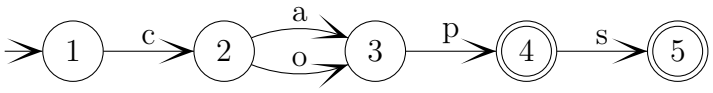
Another approach is to build a function, called a *hash function* that maps each state/action pair to a small integer. We then allocate a 1D array with one position for each state/action pair. Each array cell then contains a list of new states for this state/action pair. The details of hash functions are beyond the scope of this class. However, modern programming languages often include built-in *hash table* or *dictionary* objects that handle the details for you.

19.6 Shared states

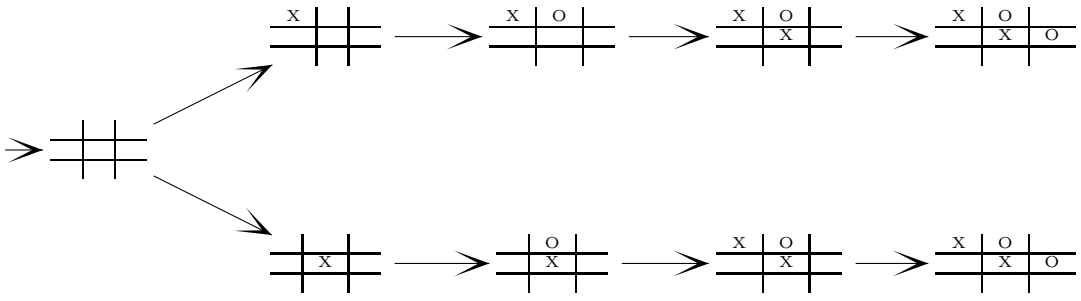
Suppose that each word in our dictionary had its own phone lattice, and we then merge these individual lattices to form lattices representing sets of words. We might get a lattice like the following one, which represents the set of words {`cop`, `cap`, `cops`, `caps`}.



Although this lattice encodes the right set of words, it uses a lot more states than necessary. We can represent the same information using the following, much more compact, phone lattice.

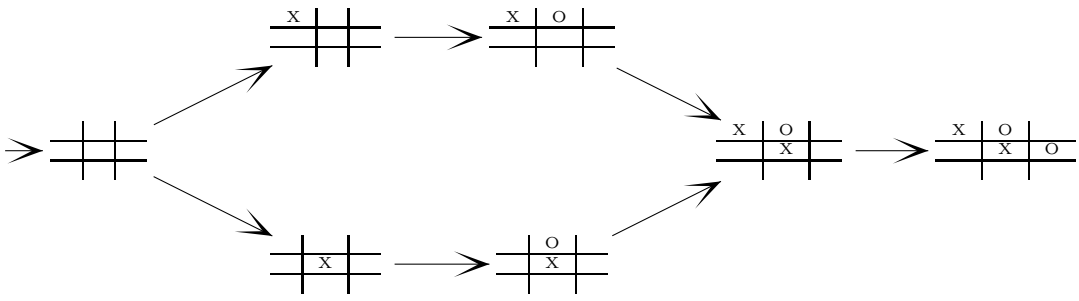


State merger is even more important when states are generated dynamically. For example, suppose that we are trying to find an optimal strategy for playing a game like tic-tac-toe. Blindly enumerating sequences of moves might create state diagrams such as:



Searching through the same sequence of moves twice wastes time as well as space. If we build an index of states we've already generated, we can detect when we get to the same state a second time. We can then use the results of our previous work, rather than repeating it. This trick, called *dynamic*

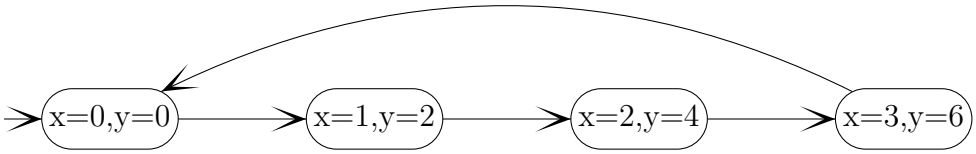
programming, can significantly improve the running time of algorithms.



We can also use state diagrams to model what happens when computer programs run. Consider the following piece of code

```
cyclic()
    y = 0
    x = 0
    while (y < 100)
         x = remainder(x+1,4)
         y = 2x
```

We can represent the state of this machine by giving the values of x and y. We can then draw its state diagram as shown below. By recognizing that we return to the same state after four iterations, we not only keep the diagram compact but also capture the fact that the code goes into an infinite loop.



19.7 Counting states

The size of state diagrams varies dramatically with the application. For example, the wolf-goat-cabbage problem has only $2^4 = 16$ possible states, because we have two choices for the position of each of the four key objects (wolf, goat, cabbage, boat). Of these, six aren't legal because they leave a hungry animal next to its food with the boat on the other side of the river. For this application, we have no trouble constructing the full state diagram.

Many applications, however, generate quite big state diagrams. For example, a current speech understanding system might need to store tens of thousands of words and millions of short phrases. This amount of data can be packed into computer memory only by using high-powered storage and compression tricks.

Eventually, the number of states becomes large enough that it simply isn't possible to store them explicitly. For example, the board game Go is played on a 19 by 19 grid of positions. Each position can be empty, filled with a black stone, or filled with a white stone. So, to a first approximation, we have 3^{361} possible game positions. Some of these are forbidden by the rules of the game, but not enough to make the size of the state space tractable.

Some games and theoretical models of computation use an unbounded amount of memory, so that their state space becomes infinite. For example, Conway's "Game of Life" runs on an 2D grid of cells, in which each cell has 8 neighbors. The game starts with an initial configuration, in which some set of cells is marked as live, and the rest are marked as dead. At each time step, the set of live cells is updated using the rules:

- A live cell stays live if it 2 or 3 neighbors. Otherwise it becomes dead.
- A dead cell with exactly 3 neighbors becomes live.

For some initial configurations, the system goes into a stable state or oscillates between several configurations. For some configurations (e.g. so-called "gliders") the set of live cells moves across the plane. If our viewing window shifts along with the glider, we can still represent this situation with only a finite number of states. However, more interestingly, some finite configurations (so-called "glider guns") grow without bound as time moves

forwards, so that the board includes more and more live cells. For these initial configurations, a representation of the system definitely requires infinitely many states.

When a system has an intractably large number of states, whether finite or infinite, we obviously aren't going to build its state space explicitly. Analysis of such systems requires techniques like computing high-level properties of states, generating states in a smart way, and using heuristics to decide whether a state is likely to lead to a final solution.

19.8 Variation in notation

State diagrams of various sorts, and constructs very similar to state diagrams, are used in a wide range of applications. Therefore, there are many different sets of terminology for equivalent and/or subtly different variations on this idea. In particular, when a state diagram is viewed as an active device, i.e. as a type of machine or computer, it is often called a *state transition* or an *automaton*.

Start states are also known as *initial* states. End states can be called *final* states or *goal* states.

There are two choices for setting up the transition function δ. State diagrams where the transition function returns a set of states are called *nondeterministic*. Those where δ returns a single state are called *deterministic*. Deterministic state diagrams are less flexible but easier to implement efficiently.

Chapter 20

Countability

This chapter covers infinite sets and countability.

20.1 The rationals and the reals

You're familiar with three basic sets of numbers: the integers, the rationals, and the reals. The integers are obviously discrete, in that there's a big gap between successive pairs of integers.

To a first approximation, the rational numbers and the real numbers seem pretty similar. The rationals are dense in the reals: if I pick any real number x and a distance δ, there is always a rational number within distance δ of x. Between any two real numbers, there is always a rational number.

We know that the reals and the rationals are different sets, because we've shown that a few special numbers are not rational, e.g. π and $\sqrt{2}$. However, these irrational numbers seem like isolated cases. In fact, this intuition is entirely wrong: the vast majority of real numbers are irrational and the rationals are quite a small subset of the reals.

20.2 Completeness

One big difference between the two sets is that the reals have a so-called "completeness" property. It states that any subset of the reals with an upper bound has a smallest upper bound. (And similarly for lower bounds.) So if I have a sequence of reals that converges, the limit it converges to is also a real number. This isn't true for the rationals. We can make a series of rational numbers that converge π (for example) such as

$$3, 3.1, 3.14, 3.141, 3.1415, 3.14159, 3.141592, 3.1415926, 3.14159265$$

But there is no rational number equal to π.

In fact, the reals are set up precisely to make completeness work. One way to construct the reals is to construct all convergent sequences of rationals and add new points to represent the limits of these sequences. Most of the machinery of calculus depends on the existence of these extra points.

20.3 Cardinality

We know how to calculate and compare the sizes of finite sets. To extend this idea to infinite sets, we use bijections functions to compare the sizes of sets:

> Definition: Two sets A and B have the same cardinality ($|A| = |B|$) if and only if there is a bijection from A to B.

We've seen that there is a bijection between two finite sets exactly when the sets have the same number of elements. So this definition of cardinality matches our normal notion of how big a finite set is. But, since we don't have any numbers of infinite size, working with bijections extends better to infinite sets.

The integers and the natural numbers have the same cardinality, because we can construct a bijection between them. Consider the function $f : \mathbb{N} \to \mathbb{Z}$ where $f(n) = \frac{n}{2}$ when n is even and $f(n) = \frac{-(n+1)}{2}$ when n is odd. f maps

the even natural numbers bijectively onto the non-negative integers. It maps the odd natural numbers bijectively onto the negative integers.

Similarly, we can easily construct bijections between $\mathbb{Z}$ or $\mathbb{N}$ and various of their infinite subsets. For example, the formula $f(n) = 3^n$ creates a bijection from the integers to the powers of 3. If S is any infinite subset of the natural numbers, we can number the elements of S in order of increasing size: $s_0, s_1, s_2, \ldots$. This creates a bijection between S and $\mathbb{N}$.

Because the integers are so important, there's a special name for sets that have the same cardinality as the integers:

> An infinite set A is *countably infinite* if there is a bijection from $\mathbb{N}$ (or equivalently $\mathbb{Z}$) onto A.

The term *countable* is used to cover both finite sets and sets that are countably infinite. All subsets of the integers are countable.

20.4 Cantor Schroeder Bernstein Theorem

For certain countably infinite sets, it's awkward to directly build a bijection to the integers or natural numbers. Fortunately, a more general technique is available. Remember that for finite sets, we could build a one-to-one function from A to B if and only if $|A| \leq |B|$. Using this idea, we can define a partial order on sets, finite or infinite:

> Definition: $|A| \leq |B|$ if and only if there is a one-to-one function from A to B.

It is the case that if $|A| \leq |B|$ and $|B| \leq |A|$, then $|A| = |B|$. That is, if you can build one-to-one functions in both directions, a bijection does exist. This result is called the Cantor Schroeder Bernstein Theorem.[1] It allows us to do very slick 2-way bounding proofs that a wide range of sets are countably infinite.

[1] See the Fendel and Resek book in the bibliography for a proof, and the Liebeck book for a much simpler proof for the case where one of the sets is $\mathbb{N}$.

For example, consider $\mathbb{N}^2$, the set of pairs of natural numbers. It's possible to directly construct a bijection $f : \mathbb{N}^2 \to \mathbb{N}$, but the details are a bit messy. Instead, let's build one-to-one functions in both directions. Easy direction first: define $f_1 : \mathbb{N} \to \mathbb{N}^2$ by $f_1(n) = (n, 0)$. This is one-to-one, so $|\mathbb{N}| \leq |\mathbb{N}^2|$ (which you were probably prepared to consider obvious).

In the opposite direction, consider the following function: $f_2 : \mathbb{N}^2 \to \mathbb{N}$ such that $f_2(n, m) = 2^n 3^m$. This is one-to-one because prime factorizations are unique. So $|\mathbb{N}^2| \leq |\mathbb{N}|$. Since we have one-to-one functions in both directions, Cantor Schroeder Bernstein implies that $|\mathbb{N}^2| = |\mathbb{N}|$. Therefore $\mathbb{N}^2$ is countably infinite.

This construction can be extended to show the countability of any finite Cartesian product of integers or natural numbers. E.g. the set of 7-tuples of integers is countable. This also implies that a countable union of countable sets is countable, because we can use pairs of natural numbers to index the members of such a union. That is, the kth element of the jth set in the union would be associated with the element (j, k) in $\mathbb{N}^2$.

20.5 More countably infinite sets

Suppose that we have a finite set M of characters. For example, M might be the set of 26 upper-case alphabetic characters. Then the set M^* contains all character strings of various finite lengths, such as `I`, `THIS`, `FINITE`, and `RUMBLESEAT`. It also contains the string ϵ of zero length. I claim that M^* is countably infinite.

To prove this, we'll build one-to-one functions in both directions. First, we can create a one-to-one function f from $\mathbb{N}$ to M^* by mapping each natural number n to the string consisting of n A's. For example, $f(0) = \epsilon$, $f(2) =$ `AA`, and $f(5) =$ `AAAAA`.

In the other direction, notice that each letter in M has a 2-digit ASCII code: A has the code 65, B is 66, and so on up to 90 for Z. We can translate each string into a sequence of digits by replacing each letter with its ASCII code. E.g. `RUBY` becomes 82856689. This doesn't work for the string ϵ, so we'll translate it specially to the number 0. We've now created a one-to-one mapping from strings in A^* to natural numbers.

We can also show that the non-negative rational numbers are countably infinite. It's easy to make a one-to-one function from the natural numbers to the non-negative rational numbers: just map each natural number n to itself. So $|\mathbb{N}| \leq |\mathbb{Q}^{\geq 0}|$. So now we just need a one-to-one function from the non-negative rationals to the integers, to show that $|\mathbb{Q}^{\geq 0}| \leq |\mathbb{N}|$.

To map the non-negative rational numbers to the natural numbers, first map each rational number to one representative fraction, e.g. the one in lowest terms. This isn't a bijection, but it is one-to-one. Then use the method we saw above to map the fractions, which are just pairs of non-negative integers, to the natural numbers. We now have the required one-to-one function from the non-negative rationals to the natural numbers.

This construction can be adapted to also handle negative rational numbers. So the set of rational numbers is countably infinite. And, more generally, any subset of the rationals is countable.

20.6 $\mathbb{P}(\mathbb{N})$ isn't countable

Before looking at the real numbers, let's first prove a closely-related result that's less messy: $\mathbb{P}(\mathbb{N})$ isn't countable. Recall that $\mathbb{P}(\mathbb{N})$ is the power set of the natural numbers i.e. the set containing all subsets of the natural numbers.

Suppose that A is a finite set $\{a_0, a_1, a_2, \ldots, a_n\}$. We can represent a subset X of A as a bit vector $\{b_0, b_1, b_2, \ldots, b_n\}$ where b_i is 1 if and only if a_i is in X. For example, if $A = \{7, 8, 9, 10, 11\}$, then the bit-vector 01100 would represent the subset $\{8, 9\}$ and the bit-vector 10101 would represent the subset $\{7, 9, 11\}$. Similarly, we can represent a subset of the natural numbers as an infinite-length bit vector.

We'll use a procedure called **diagonalization** (due to Georg Cantor) to show that it's impossible to build a bijection from the natural numbers to these infinite bit vectors representing the subsets of the natural numbers Suppose that there were such a bijection. Then we could put all the bit vectors into a list, e.g. v_0 would be the first bit vector, v_1 the second, and so forth. Our list of bit vectors might look like this, where the kth column contains the value of the kth digit (b_k) for all the bit vectors:

	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	...
v_0	1	1	0	1	1	0	1	1	1	1	...
v_1	1	1	0	0	1	0	1	1	0	0	...
v_2	0	0	0	0	1	0	0	1	0	0	...
v_3	0	1	1	1	1	0	1	0	0	0	...
v_4	0	0	0	0	1	1	1	0	1	1	...
v_5	1	1	1	0	1	0	1	0	0	1	...
...	...										

This is supposed to be a complete list of all the bit vectors. But we can construct a bit vector x that's not on the list. The value of x_k, i.e. the kth bit in our new vector, will be 0 if the k digit of v_k is 1, and 1 if the k digit of v_k is 0. Notice that x is different from v_3 because the two vectors differ in the third position. It can't be v_{20} because the two vectors differ in the twentieth position. And, in general, x can't equal v_k because the two vectors differ in the kth position. For the example above, the new vector not in the list would start out: 0 0 1 0 0 1 ....

So, it's not possible to put these infinite bit vectors into a list indexed by the natural numbers, because we can always construct a new bit vector that's not on the list. That is, there can't be a one-to-one function from the infinite bit vectors to the natural numbers. So there can't be a one-to-one function from the subsets of the natural numbers to the natural numbers. So $\mathbb{P}(\mathbb{N})$ isn't countable. That is, the subsets of the natural numbers are more numerous than the natural numbers themselves.

20.7 More uncountability results

This same diagonalization trick can be used to show that various other sets aren't countable. For example, we can show that the real numbers aren't countable. To show this, we show that even the numbers in the interval $[0, 1]$ aren't countable. Suppose that the elements of $[0, 1]$ were countable. Then we could put these real numbers into a list a_1, a_2, and so forth. Let's write out a table of the decimal expansions of the numbers on this list. Now, examine the digits along the diagonal of this table: a_{11}, a_{22}, etc. Suppose we construct a new number b whose kth digit b_k is 4 when a_{kk} is 5, and 5

otherwise. Then b won't match any of the numbers in our table, so our table wasn't a complete list of all the numbers in $[0, 1]$. So, $[0, 1]$ is not countable, and therefore the reals can't be countable.

Next, notice that an infinite bit vector is a function from the natural numbers to the set $\{0, 1\}$. So we've shown that there are uncountably many functions from the natural numbers to $\{0, 1\}$. So there must be uncountably many functions from the natural numbers to the natural numbers, or from the integers to the integers.

Another generalization involves noticing that our diagonalization proof doesn't really depend on any special properties of the natural numbers. So it can be adapted to show that, if A is any set, $\mathbb{P}(A)$ has a (strictly) larger cardinality than A. So, not only is $\mathbb{P}(\mathbb{N})$ larger than $\mathbb{N}$, but $\mathbb{P}(\mathbb{P}(\mathbb{N}))$ is even larger. So there is a whole sequence of larger and larger infinite cardinalities.

So, in particular, $\mathbb{P}(\mathbb{R})$ is larger than $\mathbb{R}$. However, notice that $\mathbb{R}^2$ has the same cardinality as $\mathbb{R}$. Let's consider a simpler version of this problem: $[0, 1)^2$ has the same cardinality as $[0, 1)$. Any element of $[0, 1)^2$ can be represented as two infinite sequences of decimal digits: $0.a_1a_2a_3a_4\ldots$ and $0.b_1b_2b_3b_4\ldots$. We can map this to a single real number by interleaving the digits of the two numbers: $0.a_1b_1a_2b_2a_3b_3\ldots$. This defines a bijection between the two sets. This method can be adapted to create a bijection between all of $\mathbb{R}^2$ and $\mathbb{R}$.

20.8 Uncomputability

We can pull some of these facts together into some interesting consequences for computer science. Notice that a formula for a function is just a finite string of characters. So the set of formulas is countable. But the set of functions, even from the integers to the integers, is uncountable. So there are more functions than formulas, i.e. some functions which have no finite formula.

Similarly, notice that a computer program is simply a finite string of ASCII characters. So there are only countably many computer programs. But there are uncountably many functions. So there are more functions than programs, i.e. there are functions which cannot be computed by any program.

Specifically, it can be shown that it is not possible to build a program that reads the code of other programs and decides whether they eventually halt or run forever. At least, not a fully-general algorithm that will work on any possible input program. This famous result is called the **Halting Problem** and its proof uses a variation of diagonalization.

The reason why it's so hard to tell if a program halts is that, even though the code for a computer program is finite in length, the program may go through infinite many different states as it runs. Specifically, program behavior falls into three categories:

(1) The program eventually halts, so the trace is finite.

(2) The program loops, in the sense of returning back to a previous state.

(3) The program keeps going forever, consuming more and more storage space rather than returning to a previous state.

The second type of behavior is like the decimal expansion of a rational number: repeating. The third type of behavior is similar to a real number's decimal expansion: an infinite sequence which doesn't repeat. The potential for running forever without repeating states is what makes the Halting Problem so difficult.

The Halting problem is closely connected to results from other areas involving non-periodic behavior, because other types of systems can be used to simulate the essential features of computer programs. A simple, but effective computer can be simulated in Conway's Game of Life. Configurations such as glider guns that generate infinitely many different states correspond directly to programs that run forever without looping.

A final example involves tilings of the plane. Periodic tilings, which are made up of many copies of a single repeated pattern, are like rational numbers or looping programs. Aperiodic tilings, whose existence was only proved in 1966, are like real numbers or the third type of program: they go on forever, but not by repeating a single pattern over and over. The original proof of their existence involved simulating a simple computer using a set of 20,426 boring square tiles. By the 1970's, the critical simulations had been made simple and Roger Penrose had developed aperiodic sets of tiles that were small (e.g. 2 tiles) and made pretty patterns. (See the internet for pictures.)

Aside from looking cool, aperiodic tilings can have types of symmetries that periodic tilings can't, e.g. 10-fold rotational symmetry. In 1982, a material science professor named Dan Shechtman observed such symmetries in electron diffraction patterns. His discovery of these "quasicrystals" was initially greeted with skepticism but eventually won him the Nobel prize in 2011.

20.9 Variation in notation

Authors differ as to whether the term *countable* includes finite sets or only countably infinite sets.

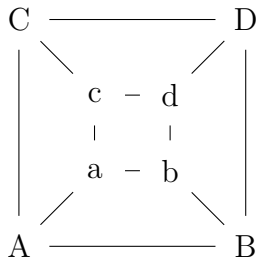
Chapter 21

Planar Graphs

This chapter covers special properties of planar graphs.

21.1 Planar graphs

A planar graph is a graph which can be drawn in the plane without any edges crossing. Some pictures of a planar graph might have crossing edges, but it's possible to redraw the picture to eliminate the crossings. For example, although the usual pictures of K_4 and Q_3 have crossing edges, it's easy to redraw them so that no edges cross. For example, a planar picture of Q_3 is shown below. However, if you fiddle around with drawings of $K_{3,3}$ or K_5, there doesn't seem to be any way to eliminate the crossings. We'll see how to prove that these two graphs aren't planar.



Why should we care? Planar graphs have some interesting mathematical properties, e.g. they can be colored with only 4 colors. Also, as we'll see later, we can use facts about planar graphs to show that there are only 5 Platonic solids.

There are also many practical applications with a graph structure in which crossing edges are a nuisance, including design problems for circuits, subways, utility lines. Two crossing connections normally means that the edges must be run at different heights. This isn't a big issue for electrical wires, but it creates extra expense for some types of lines e.g. burying one subway tunnel under another (and therefore deeper than you would ordinarily need). Circuits, in particular, are easier to manufacture if their connections live in fewer layers.

21.2 Faces

When a planar graph is drawn with no crossing edges, it divides the plane into a set of regions, called *faces*. By convention, we also count the unbounded area outside the whole graph as one face. The *boundary* of a face is the subgraph containing all the edges adjacent to that face and a *boundary walk* is a closed walk containing all of those edges. The *degree* of the face is the minimum length of a boundary walk. For example, in the figure below, the lefthand graph has three faces. The boundary of face 2 has edges df, fe, ec, cd, so this face has degree 4. The boundary of face 3 (the unbounded face) has edges bd, df, fe, ec, ca, ab, so face 3 has degree 6.

The righthand graph above has a spike edge sticking into the middle of face 1. The boundary of face 1 has edges bf, fe, ec, cd, ca, ab. However, any boundary walk must traverse the spike twice, e.g. one possible boundary walk is $bf, fe, ec, cd, cd, ca, ab$, in which cd is used twice. So the degree of face 1 in the righthand graph is 7. In such cases, it may help to think of walking along inside the face just next to the boundary, rather walking along the boundaries themselves. Notice that the boundary walk for such a face is not a cycle.

Suppose that we have a graph with e edges, v nodes, and f faces. We know that the Handshaking theorem holds, i.e. the sum of node degrees is $2e$. For planar graphs, we also have a *Handshaking theorem for faces*: the sum of the face degrees is $2e$. To see this, notice that a typical edge forms part of the boundary of two faces, one to each side of it. The exceptions are edges, such as those involved in a spike, that appear twice on the boundary of a single face.

Finally, for connected planar graphs, we have Euler's formula: $v - e + f = 2$. We'll prove that this formula works.[1]

21.3 Trees

Before we try to prove Euler's formula, let's look at one special type of planar graph: free trees. In graph theory, a **free tree** is any connected graph with no cycles. Free trees are somewhat like normal trees, but they don't have a designated root node and, therefore, they don't have a clear ancestor-descendent ordering to their notes.

A free tree doesn't divide the plane into multiple faces, because it doesn't

[1] You can easily generalize Euler's formula to handle graphs with more than one connected components.

contain any cycles. A free tree has only one face: the entire plane surrounding it. So Euler's theorem reduces to $v - e = 1$, i.e. $e = v - 1$. Let's prove that this is true, by induction.

> Proof by induction on the number of nodes in the graph.
>
> Base: If the graph contains no edges and only a single node, the formula is clearly true.
>
> Induction: Suppose the formula works for all free trees with up to n nodes. Let T be a free tree with $n + 1$ nodes. We need to show that T has n edges.
>
> Now, we find a node with degree 1 (only one edge going into it). To do this start at any node r and follow a walk in any direction, without repeating edges. Because T has no cycles, this walk can't return to any node it has already visited. So it must eventually hit a dead end: the node at the end must have degree 1. Call it p.
>
> Remove p and the edge coming into it, making a new free tree T' with n nodes. By the inductive hypothesis, T' has $n - 1$ edges. Since T has one more edge than T', T has n edges. Therefore our formula holds for T.

21.4 Proof of Euler's formula

We can now prove Euler's formula ($v - e + f = 2$) works in general, for any connected planar graph.

> Proof: by induction on the number of edges in the graph.
>
> Base: If $e = 0$, the graph consists of a single node with a single face surrounding it. So we have $1 - 0 + 1 = 2$ which is clearly right.
>
> Induction: Suppose the formula works for all graphs with no more than n edges. Let G be a graph with $n + 1$ edges.
>
> Case 1: G doesn't contain a cycle. So G is a free tree and we already know the formula works for free trees.

Case 2: G contains at least one cycle. Pick an edge p that's on a cycle. Remove p to create a new graph G'.

Since the cycle separates the plane into two faces, the faces to either side of p must be distinct. When we remove the edge p, we merge these two faces. So G' has one fewer faces than G.

Since G' has n edges, the formula works for G' by the induction hypothesis. That is $v' - e' + f' = 2$. But $v' = v$, $e' = e - 1$, and $f' = f - 1$. Substituting, we find that

$$v - (e - 1) + (f - 1) = 2$$

So

$$v - e + f = 2$$

21.5 Some corollaries of Euler's formula

Corollary 1 *Suppose G is a connected planar graph, with v nodes, e edges, and f faces, where $v \geq 3$. Then $e \leq 3v - 6$.*

Proof: The sum of the degrees of the faces is equal to twice the number of edges. But each face must have degree ≥ 3. So we have $3f \leq 2e$.

Euler's formula says that $v - e + f = 2$, so $f = e - v + 2$ and thus $3f = 3e - 3v + 6$. Combining this with $3f \leq 2e$, we get $3e - 3v + 6 \leq 2e$. So $e \leq 3v - 6$.

We can also use this formula to show that the graph K_5 isn't planar. K_5 has five nodes and 10 edges. This isn't consistent with the formula $e \leq 3v-6$. Unfortunately, this method won't help us with $K_{3,3}$, which isn't planar but does satisfy this equation.

We can also use this Corollary 1 to derive a useful fact about planar graphs:

Corollary 2 *If G is a connected planar graph, G has a node of degree less than six.*

> Proof: This is clearly true if G has one or two nodes.
>
> If G has at least three nodes, then suppose that the degree of each node was at least 6. By the handshaking theorem, $2e$ equals the sum of the degrees of the nodes, so we would have $2e \geq 6v$. But corollary 1 says that $e \leq 3v - 6$, so $2e \leq 6v - 12$. We can't have both $2e \geq 6v$ and $2e \leq 6v - 12$. So there must have been a node with degree less than six.

If our graph G isn't connected, the result still holds, because we can apply our proof to each connected component individually. So we have:

Corollary 3 *If G is a planar graph, G has a node of degree less than six.*

21.6 $K_{3,3}$ is not planar

When all the cycles in our graph have at least four nodes, we can get a tighter relationship between the numbers of nodes and edges.

Corollary 4 *Suppose G is a connected planar graph, with v nodes, e edges, and f faces, where $v \geq 3$. and if all cycles in G have length ≥ 4, then $e \leq 2v - 4$.*

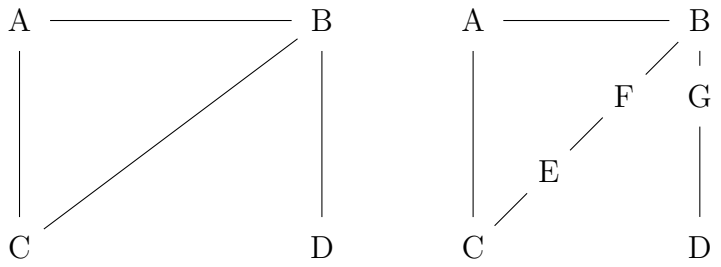
> Proof: The sum of the degrees of the faces is equal to twice the number of edges. But each face must have degree ≥ 4 because all cycles have length ≥ 4. So we have $4f \leq 2e$, so $2f \leq e$.
>
> Euler's formula says that $v - e + f = 2$, so $e - v + 2 = f$, so $2e - 2v + 4 = 2f$. Combining this with $2f \leq e$, we get that $2e - 2v + 4 \leq e$. So $e \leq 2v - 4$.

This result lets us show that $K_{3,3}$ isn't planar. All the cycles in $K_{3,3}$ have at least four nodes. But $K_{3,3}$ has 9 edges and 6 nodes, which isn't consistent with this formula. So $K_{3,3}$ can't be planar.

21.7 Kuratowski's Theorem

The two example non-planar graphs $K_{3,3}$ and K_5 weren't picked randomly. It turns out that any non-planar graph must contain a subgraph closely related to one of these two graphs. Specifically, we'll say that a graph G is a *subdivision* of another graph F if the two graphs are isomorphic or if the only difference between the graphs is that G divides up some of F's edges by adding extra degree 2 nodes in the middle of the edges.

For example, in the following picture, the righthand graph is a subdivision of the lefthand graph.

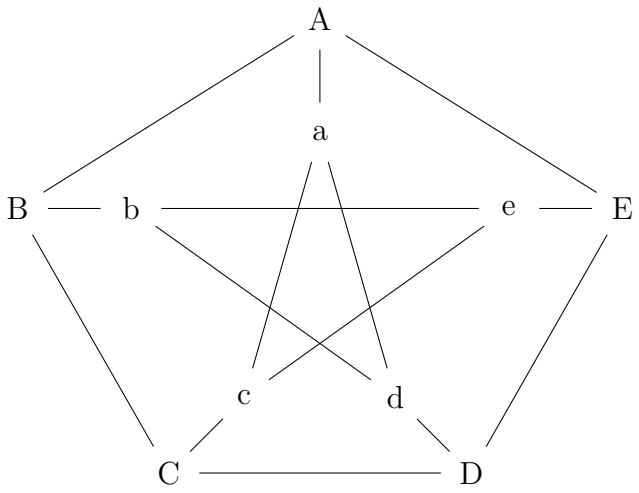


We can now state our theorem precisely.

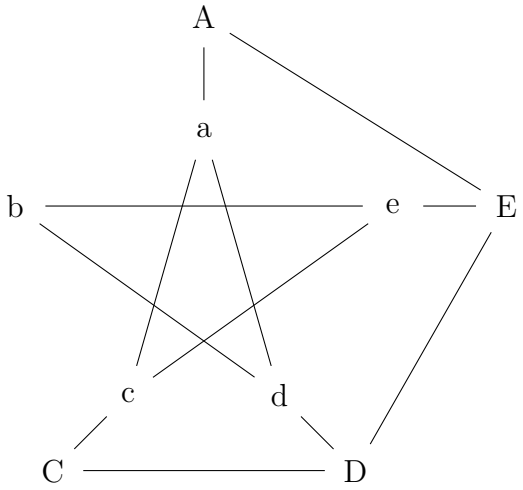
Claim 54 *Kuratowski's Theorem: A graph is nonplanar if and only if it contains a subgraph that is a subdivision of $K_{3,3}$ or K_5.*

This was proved in 1930 by Kazimierz Kuratowski, and the proof is apparently somewhat difficult. So we'll just see how to apply it.

For example, here's a graph known as the Petersen graph (after a Danish mathematician named Julius Petersen).

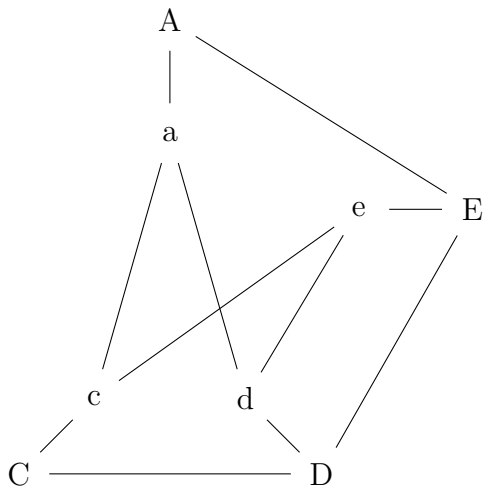


This isn't planar. The offending subgraph is the whole graph, except for the node B (and the edges that connect to B):

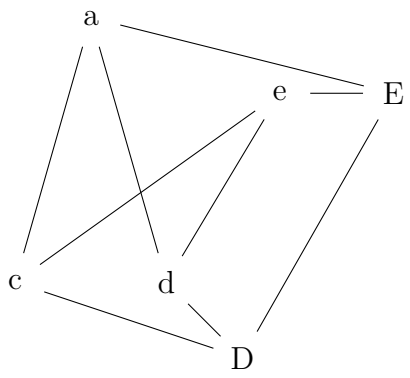


This subgraph is a subdivision of $K_{3,3}$. To see why, first notice that the node b is just subdividing the edge from d to e, so we can delete it. Or,

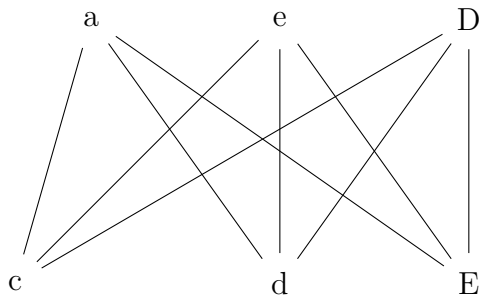
formally, the previous graph is a subdivision of this graph:



In the same way, we can remove the nodes A and C, to eliminate unnecessary subdivisions:



Now deform the picture a bit and we see that we have $K_{3,3}$.



21.8 Coloring planar graphs

One application of planar graphs involves coloring maps of countries. Two countries sharing a border[2] must be given different colors. We can turn this into a graph coloring problem by assigning a graph node to each country. We then connect two nodes with an edge exactly when their regions share a border. This graph is called the *dual* of our original map. Because the maps are planar, these dual graphs are always planar.

Planar graphs can be colored much more easily than other graphs. For example, we can prove that they never require more than 6 colors:

> Proof: by induction on the number of nodes in G.
>
> Base: The planar graph with just one node has maximum degree 0 and can be colored with one color.
>
> Induction: Suppose that any planar graph with $< k$ nodes can be colored with 6 colors. Let G be a planar graph with k nodes.

[2]Two regions touching at a point are not considered to share a border.

> By Corollary 3, G has a node of degree less than 6. Let's pick such a node and call it v.
>
> Remove some node v (and its edges) from G to create a smaller graph G'. G' is a planar graph with $k-1$ nodes. So, by the inductive hypothesis, G' can be colored with 6 colors.
>
> Because v has less than 6 neighbors, its neighbors are only using 5 of the available colors. So there is a spare color to assign to v, finishing the coloring of G.

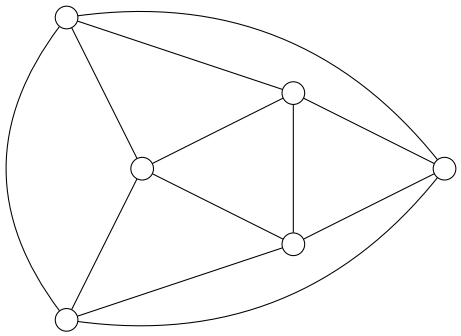
It's not hard, but a bit messy, to upgrade this proof to show that planar graphs require only five colors. Four colors is much harder. Way back in 1852, Francis Guthrie hypothesized that any planar graph could be colored with only four colors, but it took 124 years to prove that he was right. Alfred Kempe thought he had proved it in 1879 and it took 11 years for another mathematician to find an error in his proof.

The *Four Color Theorem* was finally proved by Kenneth Appel and Wolfgang Haken at UIUC in 1976. They reduced the problem mathematically, but were left with 1936 specific graphs that needed to be checked exhaustively, using a computer program. Not everyone was happy to have a computer involved in a mathematical proof, but the proof has come to be accepted as legitimate.

21.9 Application: Platonic solids

A fact dating back to the Greeks is that there are only five *Platonic solids*: cube, dodecahedron, tetrahedron, icosahedron, octahedron. These are convex polyhedra whose faces all have the same number of sides (k) and whose nodes all have the same number of edges going into them (d).

To turn a Platonic solid into a graph, imagine that it's made of a stretchy material. Make a small hole in one face. Put your fingers into that face and pull sideways, stretching that face really big and making the whole thing flat. For example, an octahedron (8 triangular sides) turns into the following graph. Notice that it still has eight faces, one for each face of the original solid, each with three sides.



Graphs of polyhedra are slightly special planar graphs. Polyhedra aren't allowed to have extra nodes partway along edges, so each node in the graph must have degree at least three. Also, since the faces must be flat and the edges straight, each face needs to be bounded by at least three edges. So, if G is the graph of a Platonic solid, all the nodes of G must have the same degree $d \geq 3$ and all faces must have the same degree $k \geq 3$.

Now, let's do some algebra to see why there are so few possibilities for the structure of such a graph.

By the handshaking theorem, the sum of the node degrees is twice the number of edges. So, since the degrees are equal to d, we have $dv = 2e$ and therefore

$$v = \frac{2e}{d}$$

By the handshaking theorem for faces, the sum of the face degrees is also twice the number of edges. That is $kf = 2e$. So

$$f = \frac{2e}{k}$$

Euler's formula says that $v - e + f = 2$, so $v + f = 2 + e > e$. Substituting the above equations into this one, we get:

$$\frac{2e}{d} + \frac{2e}{k} > e$$

Dividing both sides by $2e$:

$$\frac{1}{d} + \frac{1}{k} > \frac{1}{2}$$

If we analyze this equation, we discover that d and k can't both be larger than 3. If they were both 4 or above, the left side of the equation would be $\leq \frac{1}{2}$. Since we know that d and k are both ≥ 3, this implies that one of the two is actually equal to three and the other is some integer that is at least 3.

Suppose we set d to be 3. Then the equation becomes $\frac{1}{3} + \frac{1}{k} > \frac{1}{2}$. So $\frac{1}{k} > \frac{1}{6}$, which means that k can't be any larger than 5. Similarly, if k is 3, then d can't be any larger than 5.

This leaves us only five possibilities for the degrees d and k: $(3, 3)$, $(3, 4)$, $(3, 5)$, $(4, 3)$, and $(5, 3)$. Each of these corresponds to one of the Platonic solids.

Appendix A

Jargon

Mathematicians write in a dialect of English that differs from everyday English and from formal scientific English. To read and write mathematics fluently, you need to be aware of the differences.

A.1 Strange technical terms

Many technical terms, abbreviations, and shorthand symbols are easy to track down. Perhaps they belong to an obvious topic area. Perhaps they are used so heavily that no one forgets what they mean. Perhaps they are easily found on Wikipedia. But others seem to appear mysteriously out of nowhere.

abuse of notation The author is using a notation that doesn't technically comply with the rules of mathematical writing, but is very convenient or conventional. For example, using "if" in a a definition where we really mean "if and only if." Or writing $f(2,3)$ for the result of applying f to the pair $(2,3)$. Strictly we should have written $f((2,3))$, but no one except a computer program would ever do that.

$\exists!$ This is the unique existence quantifier. $\exists! x, P(x)$ means that there is exactly one x for which $P(x)$ holds.

By symmetry A variant wording for "without loss of generality."

IH Inductive hypothesis, as in a proof by induction.

QED Short for "quod erat demonstrandum." This is just the Latin translation of "which is what we needed to show." The Latin and the English versions are both polite ways to tell your reader that the proof is finished.

tuple A generalization of "pair," "triple," etc to larger numbers of items. A k-tuple is an ordered sequence of k items. One occasionally hears "2-tuple" in place of "pair."

NTS Need to show, as in "we need to show that all horses have four legs."

WLOG, WOLOG, without loss of generality We're stipulating an extra fact or relationship, but claiming that it doesn't actually add any new information to the problem. For example, if we have two real variables x and y, with identical properties so far, it's ok to stipulate that $x \leq y$ because we're simply fixing which of the two names refers to the smaller number.

A.2 Odd uses of normal words

In addition to the obvious technical terms (e.g. "rational number"), some words are used rather differently in mathematics than in everyday English.

Clearly See "obviously."

Consider The author is about to pull an example or constant or the like out of thin air. Typically, this part of the proof was constructed backwards, and the reasons for this step will not become apparent until later in the proof.

Distinct Not equal. For example, if x and y are chosen to be "distinct elements" of a set A, then x and y aren't allowed to have the same value.

General Shorhand for "general purpose," i.e. works for all inputs or in all circumstances.

Graph Might be a picture showing the relationship between two variables, or it might be a diagram with nodes and links between them.

Obviously Perhaps it really is obvious. Or perhaps the author didn't feel like working out the details (or forgot them in the heat of lecture). Or, infamously, perhaps he just wants to intimidate the audience.

Or The connective "or" and its variations (e.g. "either...or") leave open the possibility that both statements are true. If a mathematician means to exclude the possibility that both are true ("exclusive or"), they say so explicitly. In everyday English, you need to use context to determine whether an "or" was meant to be inclusive or exclusive.

Has the form E.g. "if x is rational, then x has the form $\frac{p}{q}$ where $p, q \in \mathbb{Z}$ and $q \neq 0$. Means that the definition of this type of object forces the object to have this specific internal structure.

Plainly See "obviously."

Proper For example, "proper subset" or "proper subgraph." Used to exclude the possibility of equality. Very similar to "strict."

Recall The author is about to tell you something basic that you probably ought to know, but he realizes that some of you have gaps in your background or have forgotten things. He's trying to avoid offending some of the audience by suggesting they don't know something so basic, while not embarrassing everyone else by making them confess that they don't.

Similarly You can easily fill in these details by adapting a previous part of the proof, and you'll just get bored if I spell them out. Occasionally misused in a manner similar to "obviously."

Strict A "strict" relationship is one that excludes the possibility of equality. So "strictly less than" means that the two items can't be equal. A "strictly increasing" function never stays on the same value. Compare "proper."

Suppose not A stereotypical way to start a proof by contradiction. It's shorthand for "Suppose that the claim is not true."

Unique There is only one item with the specified properties. "There is a unique real number whose square is zero."

Vacuously true The claim is true, but only because it's impossible to satisfy the hypothesis. Recall that in math, an if/then statement is considered true if its hypothesis is true. Vacuously true statements often occur when definitions are applied to examples that are very small (e.g. the empty set) or lacking some important feature (e.g. a graph with no edges, a function whose domain is the empty set).

Well-defined A mathematical object is well-defined if its definition isn't buggy. The term usually appears in a context where the definition might look ok at first glance but could have a subtle bug.

A.3 Constructions

Mathematicians also use certain syntactic constructions in ways that aren't quite the same as in normal English.

If/then statements An if/then statement with a false hypothesis is considered true in mathematics. In normal English, such statements are rare and it's not clear if they are true, false, or perhaps neither.

Imperatives A mathematical construction is often written as a recipe that the author and reader are carrying out together. E.g. "We can find a real number x such that $x^2 < 47$. The picture behind this is that the author is helping you do the mathematics, not doing it by himself while you watch. Verbs in the imperative form are sometimes actions that the reader should do by himself (e.g. "prove XX" on a problem set) but sometimes commands that the reader should do by following explicit instructions from the author. E.g. "define $f : \mathbb{N} \to \mathbb{N}$ by $f(n) = 2n$" tells you to define a function, but you are being told exactly how to do it. On a problem set, this would be background information, not a question you need to answer.

Two variables If a mathematician sets up two variables with different names, e.g. x and y, he leaves open the possibility that they might be equal. If

they are intended to have different values, they will be explicitly specified as "distinct." In normal English, giving two things different names suggests very strongly that they are different objects.

No resetting Do not change the value of a variable in the middle of a proof. Instead, use a fresh variable name. For example, suppose you know that $p = 3jk + 1$ in a proof about divisibility. To simplify the form of the equation, it's tempting to reset j to be three times its original value, so that you have $p = jk + 1$. This isn't done in standard mathematical writing. Instead, use a fresh variable name, e.g. $s = 3j$ and then $p = sk + 1$.

Variable names can be re-used when the context changes (e.g. you've started a new proof). When working through a series of examples, e.g. a bunch of example functions to be discussed, rotate variable names, so that the new definition appears after the reader has had time to forget the old definition.

Variable names Variable names are single-letter, e.g. f but not foo. The letter can be adorned with a wide variety of accents, subscripts, and superscripts, however.

A.4 Unexpectedly normal

Mathematicians are, of course, underlyingly speakers of normal English. So, sometimes, they apply the normal rules of conversational English even when these aren't part of the formal mathematical system.

names of variables In principle, any letter can be used as the name for any variable you need in a proof, with or without accent marks of various sorts. However, there are strong conventions favoring certain names. E.g. x is a real number, n is an integer, f is a function, and T is a set. Observe what names authors use in your specific topic area and don't stray too far from their conventions.

topics of equations Proofs are conversations about particular objects (e.g. polynomials, summations). Therefore, equations and other statements

are often about some specific topic object, whose properties are being described. When this is the case, the topic belongs on the lefthand side, just like the subject of a normal sentence.

Appendix B

Acknowledgements and Supplementary Readings

Most of the basic ideas and examples in this book date back many years and their original source is almost impossible to trace. I've consulted so many books and worked with so many helpful teachers, students, and course staff over the years that the details have blended into a blur. Nevertheless, certain books have had a particularly strong influence on my presentation. Most of them would make excellent supplementary materials, both for instructors and for students.

It's impossible to even think about discrete mathematics without citing the classic encyclopedic text by Rosen [Rosen 2007]. It helped me understand what topics would be most relevant to a computer science, as opposed to a mathematics, audience. Biggs [Biggs 2002], Matousek and Nesetril [Matousek and Nesetril 1998] and West [West 2001] have been invaluable references for discrete mathematics and graph theory.

From Liebeck [Liebeck 2006], Sipser [Sipser 2006], and Biggs [Biggs 2002], I learned how to extract core concepts and present them concisely. From Fendel and Resek [Fendel and Resek 1990], I learned how to explain proof mechanics and talk about the process of constructing proofs. From my numerous students and course staff, both at Iowa and Illinois, I have learned to understand why some of these concepts seem so hard to beginners. My early co-instructors Eric Shaffer and Viraj Kumar helped mould the syllabus

to the needs of our students at Illinois. Sariel Har-Peled and David Forsyth have provided many helpful pointers.

A special individual citation goes to Jeff Erickson for the recursion fairy (though my fairy has a slightly different job than his) and for extended arguments about the proper way to present induction (though we still disagree).

Finally, this book touches briefly on a couple topics which are not widely known. Details of the mathematics underlying aperiodic tilings and their connection to Turing machines can be found in [Grünbaum and Shephard 1986] and [Robinson 1971]. For more information on the Marker Making problem, see the work of Victor Milenkovic and Karen Daniels, starting with [Milenovic et al 1991, Milenovic et al 1992].

Bibliography

[Biggs 2002] Norman L. Biggs (2002) *Discrete Mathematics*, second edition, Oxford University Press, Oxford, UK.

[Fendel and Resek 1990] Daniel Fendel and Diane Resek (1990) *Foundations of Higher Mathematics: Exploration and Proof*, Addison-Wesley, Reading MA.

[Grünbaum and Shephard 1986] Branko Grünbaum and G. C. Shephard (1986) *Tilings and Patterns*, W. H. Freeman, 1987

[Liebeck 2006] Martin Liebeck (2006) *A Concise Introduction to Pure Mathematics*, Chapman Hall/CRC, Boca Raton, FL.

[Matousek and Nesetril 1998] Jiri Matousek and Jaroslav Nesetril (1998) *Invitation to Discrete Mathematics* Oxford University Press.

[Milenovic et al 1991] Victor Milenkovic, Karen Daniels, and Zhenyu Li (1991) "Automatic Marker Making," Proc. Third Canadian Conference on Computational Geometry, Vancouver, 1991, pp. 243-246.

[Milenovic et al 1992] Victor Milenkovic, Karen Daniels, and Zhenyu Li (1992) "Placement and Compaction of Non-Convex Polygons for Clothing Manufacture," Proc. Fourth Canadian Conference on Computational Geometry, Newfoundland, pp. 236-243.

[Robinson 1971] Raphael M. Robinson (1971) "Undecidability and nonperiodicity for tilings of the plane", *Inventiones Mathematicae* 12/3, pp. 177-209.

[Rosen 2007] Kenneth H. Rosen (2007) *Discrete Mathematics and Its Applications*, sixth edition, McGraw Hill, New York, NY.

[Sipser 2006] Michael Sipser, *Introduction to the Theory of Computation*, second edition, Thomson, Boston MA.

[West 2001] Douglas B. West (2001) *Introduction to Graph Theory*, second edition, Prentice-Hall, Upper Saddle River, NJ.

Appendix C

Where did it go?

When teaching the first few chapters of this book, you may be disconcerted to find some important topics apparently missing or breezed through absurdly fast. In many cases, what's happening is that the "missing" topics are covered later. This guide will help you get oriented.

It is traditional to cover most of logic and proof techniques right at the start of the course and, similarly, completely cover each topic (e.g. relations) when it first appears. This may seem intellectually tidy and it does work for the small minority who are future theoreticians. However, this method overwhelms most of the class and they get only a superficial understanding of key methods. In this book, the most difficult concepts and techniques are staged in gradually.

Counting material has been integrated with three other topics: Chapter 5 (Sets), Chapter 8 (Functions and One-to-One), and Chapter 18 (Collections of Sets).

Chapters 2-3 (Proofs and Logic). Vacuous truth is in Chapter 5 (Sets). Nested quantifiers are in Chapter 7 (Functions and Onto). Connections to circuits are in Chapter 16 (NP). Proof by contradiction is in Chapter 17 (Proof by Contradiction). The non-traditional topic of 2-way bounding is in Chapter 10.

Chapter 4 (Number theory). Equivalence Classes are introduced, but only pre-formally, in this chapter. Formal details are gradually added in Chapter

6 (Relations) and Chapter 18 (Collections of Sets).

Chapter 5 (Sets). This chapter asks them to prove subset relationships. However, proving a set equality by proving containment into both directions is delayed until Chapter 10 (2-way Bounding). Sets nested inside other sets are covered in Chapter 18 (Collections of sets).

Chapter 6 (Relations). Formal details of equivalence relations are covered in Chapter 18 (Collections of Sets). Notice that equivalence classes were introduced pre-formally in Chapter 4.

Chapters 7-8 (Functions). Set-valued functions are in Chapter 18 (Collections of Sets). The formal definition of a function as a set of pairs is delayed until Chapter 19 (State Diagrams).

Chapter 9 (Graphs). Graph in this book means undirected. Graphs are introduced early and used as examples in many later chapters. It returns as a main topic in Chapter 21 (Planar Graphs). Directed graphs are covered in Chapter 6 (Relations) and Chapter 19 (State Diagrams). The treatment of graphs is deliberately representative, not exhaustive.

Chapter 11 (Induction). This chapter introduces the basic idea, which is further developed in Chapter 12 (Recursive Definition) and Chapter 13 (Trees). Induction involving inequalities is delayed until Chapter 14 (Big-O). This organization lets us stage induction practice over several problem sets, easier examples first.