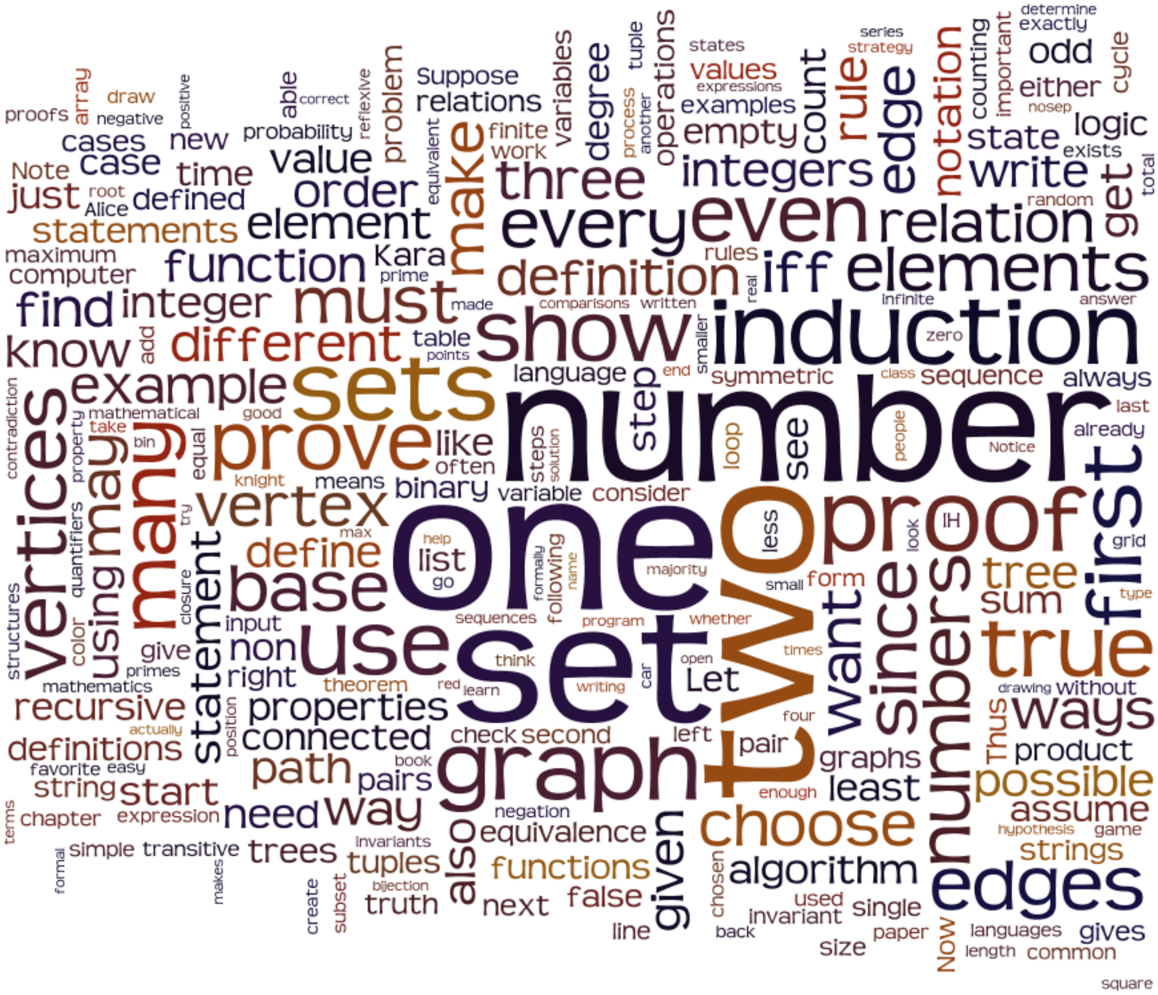
Reasoning and Writing in the Mathematics Underlying Computation



Jack Snoeyink

August 17, 2025

Contents

'Reeling and Writhing, of course, to begin with,' the Mock Turtle replied; 'and then the different branches of Arithmetic – Ambition, Distraction, Uglification, and Derision.'

—Lewis Carroll, *Alice's Adventures in Wonderland*

Glossary of Notation

One of the biggest problems of mathematics is to explain to everyone else what it is all about. The technical trappings of the subject, its symbolism and formality, its baffling terminology, its apparent delight in lengthy calculations: these tend to obscure its real nature. A musician would be horrified if his art were to be summed up as "a lot of tadpoles drawn on a row of lines," but that's all that the untrained eye can see in a page of sheet music. . . In the same way, the symbolism of mathematics is merely its coded form, not its substance.

—Ian Stewart, in *From Here to Infinity*

Quick index: Click symbol to jump to page with definition, or click the section name to jump to the glossary section with a brief definition, LaTeX and html code, and link to page with definition. Here symbols are roughly organized by number and direction of strokes, below by order introduced or importance.

Logic: $\neg \to \Rightarrow \leftrightarrow \Leftrightarrow \equiv \wedge \vee \forall \exists \nexists \therefore \oplus$
Set symbols $|| \setminus \times \cup \cap \in \notin \subseteq \oplus A^k L^*$
Set letters: $\emptyset \Lambda \mathbb{N} \mathbb{Q} \mathbb{R} \mathbb{Z} 2^A O() \Omega() \mathcal{P}() \mathbb{R}^+ \mathbb{R}^{\geq 0} \Theta() \mathbb{Z}^+$
Parens: $() \binom{n}{r} \langle\rangle \lfloor\rfloor \lceil\rceil [\,] \{\} \{||\} (,) (,] [,) [,] [..] \{\ldots\}$
Relations: $| \nmid = \neq \triangleq \approx \simeq_{i,b} \equiv_K \equiv_L \precnsim \prec \leq < > \geq \succ \succnsim aRb \not R \hat{R}$
Functions: $! \circ \sqrt{\ } f() f: \to n^{\underline{r}} \binom{n}{r} {}_nC_r \ln \lg \bmod gcd(a,b)$
Graphs $\bigcirc \square d(v) G = (V,E) K_n K_{n,m} N(v)$
Math: $x_{ij}^{12} \cdot \times \pm \infty E() F_i \varphi \hat{\varphi} Pr\{\} \mathfrak{QED}$
Greek: a–$\Omega \Lambda \varphi \hat{\varphi} O() \Omega() \Theta() \sum \prod$

Logic Basic notation and meanings; click page number or reference for detail. (Below 'col' refers to columns of a truth table.)

$\overline{p}$	`\overline p`		negation of p.	table 2.1
$\neg$	`\neg`	`¬`	logical negation	table 2.1
$\wedge$	`\land`	`∧`	logical 'and'	table 2.1
$\vee$	`\lor`	`∨`	logical inclusive 'or'	table 2.1
$\oplus$	`\oplus`	`⊕`	logical exclusive 'or'	table 2.1
$\to$	`\to`	`→`	implies, conditional	table 2.3
$\leftrightarrow$	`\leftrightarrow`	`↔`	iff, biconditional	table 2.3
$\equiv$	`\equiv`	`≡`	logically equiv.; same TT cols	15
$\Leftrightarrow$	`\Leftrightarrow`	`⇔`	synonym for $\equiv$	156
$\Rightarrow$	`\Rightarrow`	`⇒`	logical implication	23
$\therefore$	`\therefore`	`∴`	therefore, in proofs	table 2.5
$\forall$	`\forall`	`∀`	Universal quantifier	62
$\exists$	`\exists`	`∃`	Existential quantifier	62
$\nexists$	`\nexists`	`∄`	there does not exist	64

Sayings about something hard to understand:
English: "It's Greek to me."
Greek: "It's Chinese to me."
Chinese: "It's heavenly to me."

Sets General conventions, tables for definitions, special sets, and operations:
lower case letters: individual elements
upper case letters: sets or other multipart structures
caligraphic letters: families of sets; sets of multipart structures
Greek letters: strings, including logic expressions

$\in$	`\in`	`∈`	set inclusion; is an element of	39, T5.1
$\notin$	`\not \in`	`∉`	is not an element of	39
$\|A\|$	`|A|`		set cardinality; # of elts in A	table 5.1
$\subseteq$	`\subseteq`	`⊂`	subset; is contained in	39, T5.1
$\cup$	`\cup`	`∪`	union; set of elts in either	table 5.1
$\cap$	`\cap`	`∩`	intersection; set of elts in both	table 5.1
$\setminus$	`\setminus`	`∖`	set difference; in 1st, not 2nd	table 5.1
$\oplus$	`\oplus`	`⊕`	symmetric diff; in one, not both	table 5.1
$\times$	`\times`	`×`	Cartesian product; set of all pairs	40, T5.1
A^k	`A^k`		repeated product $A \times A \times \cdots A$	40
L^*	`L^*`		Kleene star; all finite strings from L	42,137

$\emptyset$	`\emptyset`	`∅`	Empty set {}	38
Λ	`\Lambda`	`Λ`	Empty string	
$\mathbb{N}$	`\mathbb {N}`	`ℕ`	Natural numbers $\{0, 1, 2, \ldots\}$	38
$\boldsymbol{N}$	`I\!\!N`		fake $\mathbb{N}$ w/o mathbb font	
$O(g(n))$	`O(g(n))`	`O( )`	set of fcns upper bdd by $cg(n)$	104
$\Omega()$	`\Omega ( )`	`Ω()`	set of fcns lower bounded	104
$\mathcal{P}(A)$	`{\cal P}(A)`	`℘(A)`	Power set, 2^A; all subsets	T 5.1
$\mathbb{Q}$	`\mathbb {Q}`	`ℚ`	Rationals $\{p/q \mid p \in \mathbb{Z}, q \in \mathbb{Z}^+\}$	39
$\mathbb{R}$	`\mathbb {R}`	`ℝ`	Real numbers	39
$\mathbb{R}^+$	`\R ^+`		Positive reals	39
$\mathbb{R}^{\geq 0}$	`\R ^{\ge 0}`		Non-negative reals	39
$\boldsymbol{R}$	`I\!\!R`		fake $\mathbb{R}$ w/o mathbb font	
$\Theta()$	`\Theta ( )`	`Θ()`	fcns upper & lower bounded	104
$\mathbb{Z}$	`\mathbb {Z}`	`ℤ`	Integers	38
$\mathbb{Z}^+$	`\Z ^+`		Positive integers	39
$\boldsymbol{Z}$	`Z\!\!\!Z`		fake $\mathbb{Z}$ w/o mathbb font	
2^A	`2^A`		synonym for power set	T 5.1

Parentheses, brackets and braces In LaTeX, writing `\left( \right)` lets parentheses or other grouping characters grow to fit their contents, like $\left(\frac{a+b}{2}\right)$.

$( )$	`( )`	parentheses: tuples, sequences, lists	40
$\binom{n}{r}$	`\binom {n}{r}`	binomial coeff; choose r from n	47,133
$\{\}$	`\{ \}`	braces define set from list of elts	38

$[a, b]$	`[a,b]`	Closed interval of reals $\{x \mid a \le x \le b\}$	39
(a, b)	`(a,b)`	Open interval of reals $\{x \mid a < x < b\}$	39
$(a, b]$	`(a,b]`	Half-open interval $\{x \mid a < x \le b\}$	39
$[a, b)$	`[a,b)`	Half-open interval $\{x \mid a \le x < b\}$	39
$[a..b]$	`[a..b]`	Set of integers $\{a, a+1, \ldots, b-1, b\}$	39
$[bool]$		Iverson bracket $[T] = 1; [F] = 0$	97
$[a]$	`[a]`	Equivalence class $[a] = \{b \mid bRa\}$	224

$\{\ldots\}$	`\{\ldots\}`	`{…}`	define set by pattern	38
$\{exp \mid rule\}$	`\{ \mid \}`		define set by rule	39
$\langle\rangle$	`\langle \rangle`	`⟨ ⟩`	angle brackets; for lists	135
$\lfloor\rfloor$	`\lfloor \rfloor`	`⌊ ⌋`	floor, max integer $\le$	112
$\lceil\rceil$	`\lceil \rceil`	`⌈ ⌉`	ceiling, min integer $\ge$	112

Relations

aRb	`aRb`		Abbrev. in relation: $(a, b) \in R$	91
$\not R$	`\not \!R`		Not in relation: $(a, b) \notin R$	91
$=$			known or assigned equal	19,91
$\neq$	`\neq`	`≠`	not equal	70,91
$\stackrel{?}{=}$	`\overset ?{=}`		I want to show equal	155
$\approx$	`\approx`	`≈`	approximately equal	113, 143
$m\|n$	`m|n`		Divides: $\exists i \in \mathbb{Z}, mi = n$	115
$m \nmid n$	`m\not |n`		m does not divide n	115
$\hat{R}$	`\hat R`		Transitive closure of R	214
$\simeq_{i,b}$	`\simeq _{i,b}`		Indistinguishable sets	226
$\equiv_K$	`\equiv _K`		Equiv strings for machine K	220
$\equiv_L$	`\equiv _L`		Equivalent strings for lang L	223
$<$		`<`	less than	91
$>$		`>`	greater than	91
$\le$	`\leq`	`≤`	less than or equal	91
$\ge$	`\geq`	`≥`	greater than or equal	91
$\prec$	`\prec`	`≺`	predecessor	
$\succ$	`\succ`	`≻`	successor; aboveness	217
$\prec\!\!\!\prec$	`\prec \!\!\!\prec`		trans closure of predecessor	
$\succ\!\!\!\succ$	`\succ \!\!\! \succ`		trans closure of successor	217

Functions

$!$	`!`	factorial	46,133
$P(n, r)$		permutations; $n!/(n-r)!$	47
$\binom{n}{r}$	`\binom {n}{r}`	binomial coeff; choose	47,133
${}_nC_r$	`_nC_r`	synonym for choose	47
$C(n, r)$		synonym for choose	47
$f(a)$	`f(a)`	Result of fcn f on input a	95

$f: A \to B$	`f\colon A\to B`	`f:A&to;B`	Fcn name:domain→range	95
$\circ$	`\circ`	`∘`	function composition	96
$\sqrt{x^2}$	`\sqrt {x^2}`	`√`	square root	97
$[bool]$			Iverson bracket $[T] = 1; [F] = 0$	97
lg	`\lg`		log base 2	113
ln	`\ln`		natural (base e) log	113
$\lfloor\ \rfloor$	`\lfloor \rfloor`	`l&floor;⌋`	floor, max integer ≤	112
$\lceil\ \rceil$	`\lceil \rceil`	`l⌈ ⌉`	ceiling, min integer ≥	112
mod	`\bmod`		mod; remainder	114
$gcd(a, b)$			greatest common divisor	123,

Graphs

○	`\fullmoon`	`◯`	graph node	249
□	`\Box`	`□`	empty binary tree	138
$d(v)$			vertex degree; $d(v) = \|N(v)\|$	238
$G = (V, E)$			graph (vertex & edge sets)	92,231
K_n			complete graph	238
$K_{n,m}$			complete bipartite graph	238
$N(v)$			set of neighbor edges	238

Misc. Math In LaTeX, math is surrounded by `$ $` or `\[ \]`. Curly braces, {}, group strings of more than one character. Sub/superscripts: x_{ij}^{12} `x_{i_j}^{12}` `x<sub>i<sub>j</sub></sub> <sup>12</sup>`

·	`\cdot`	`·`	multiplication	43
±	`\pm`	`±`	plus or minus	58
∞	`\infty`	`∞`	infinity	101, T 5.1
$E()$			expectation of random variable	270
F_i	`F_i`		ith Fibonacci number	133
φ	`\varphi`	`φ`	Golden ratio, $\varphi = (1 + \sqrt{5})/2$	144
$\hat{\varphi}$	`\hat \varphi`		$\hat{\varphi} = (1 - \varphi)$	144
$Pr\{\}$	`{\it Pr}\{\}`		probability of event	268
$\sum_i$	`\sum _{i}`	`∑`	Series; sum of terms in sequence	43,136
$\prod_i$	`\prod _{i}`	`∏`	Product of terms in sequence	43
$\bigcup_i$	`\bigcup _{i}`		Union	136
$\bigcap_i$	`\bigcap _{i}`		Intersection	136
∎			End of proof	48

Greek letters with LaTeX codes; HTML codes are `α` through `Ω`, omitting the `\var` variations.

a \alpha	∂ \theta	o o	τ \tau
β \beta	ϑ \vartheta	π \pi	υ \upsilon
γ \gamma	γ \gamma	ϖ \varpi	ϕ \phi
δ \delta	κ \kappa	ρ \rho	φ \varphi
ϵ \epsilon	λ \lambda	ϱ \varrho	χ \chi
ε \varepsilon	μ \mu	σ \sigma	ψ \psi
ζ \zeta	ν \nu	ς \varsigma	ω \omega
η \eta	ξ \xi		

Γ \Gamma	Λ \Lambda	Σ \Sigma	Ψ \Psi
Δ \Delta	Ξ \Xi	Υ \Upsilon	Ω \Omega
Θ \Theta	Π \Pi	Φ \Phi	

- http://detexify.kirelabs.org Look up LaTeX symbols that you draw onscreen.
- http://www.mathjax.org, jsMath: Common tools that support LaTeX in markdown for notebooks, blogs, and wikis. You will need to surround math equations with tags: `$$ $$`, `\( \)`, `{math}`, or ` ```latex ` are common.
- https://math.meta.stackexchange.com/questions/5020/mathjax-basic-tutorial-and-quick- has many examples.
- https://www.overleaf.com?r=634aa898&rm=e&rs=b Overleaf full-featured, collaborative online LaTeX editor.
- http://miktex.org, http://www.tug.org/texlive TeX/LaTeX for your computer.

Preface

For me, the first challenge for computing science is to discover how to maintain order in a finite, but very large, discrete universe that is intricately intertwined. And a second, but not less important challenge is how to mould what you have achieved in solving the first problem, into a teachable discipline: it does not suffice to hone your own intellect (that will join you in your grave), you must teach others how to hone theirs. The more you concentrate on these two challenges, the clearer you will see that they are only two sides of the same coin: teaching yourself is discovering what is teachable.

—Edsger Dijkstra [1]

Why discrete? Calculus deals with continuous functions, derivatives, and integrals over the uncountably infinite real numbers. Discrete mathematics deals with functions, relations, sets, graphs, and other discrete structures defined over finite and countable sets. While serious mathematicians have been known to look down on discrete math as "recreational mathematics," computer scientists find it an ideal fit for computers, which operate on bits.

This book, with its exercises, puzzles, problems, and explorations, aims to help you develop your skills in precise reasoning and precise communication about discrete structures that are important to computer science. You will learn terminology and notation that let you communicate both with computer programming languages and with potential clients. (Not that we'll deal with programming or clients in this class, but I hope that what you learn in this class will reduce debugging time and avoid costly miscommunication in your subsequent classes and career.) You will gain skills in counting discrete structures, both to check your understanding and to determine the computer resources needed to process them. You will learn how to guarantee properties of structures and programs. You will polish your writing skills, which are important even for CS majors. And I hope you will have as much fun with the material as I do, because then you'll spend the time to learn it well.

This book is being written for this course, which means that it is incomplete and likely to contain errors—don't trust everything that you read.* I wecome your corrections, comments, suggestions, elaborations, and rewrites. There is a sakai forum for textbook corrections with instructions. You can also send me email, ideally with an annotated pdf or fdf, or with enough detail that I can find where to apply your comment or correction. For example, I need not only the page number, but also the form factor (e.g. dsipad or dsphone) and document date from the title page.

*Important advice with any written material.

This book includes dynamic content via pdf annotations – hypertext, html links, pop-ups, and answers hidden under rectangles that can be selected

and dragged away or deleted. You will not see dynamic content in your web browser or tablet pdf viewer, so I strongly suggest that you download the pdf and a viewer that understands the Adobe specification for annotations. Examples include Adobe Reader or Acrobat, Foxit Reader, and Apple Preview. The annotations will not appear on printed copy.

This book has three distinguishing characteristics. First, it uses advanced applications and puzzles to encourage practice with the basic definitions of discrete structures and proofs of properties. I have taken to heart what Ralph P. Boas [12] said more than half a century ago:

> When I was teaching mathematics to future naval officers during the war, I was told that the Navy had found that men who had studied calculus made better line officers than men who had not studied calculus. Nothing is clearer (it was clear even to the Navy) than that a line officer never has the slightest use for calculus.. . . What is the explanation of the paradox?
>
> I think that the answer is supplied by a phenomenon that everybody who teaches mathematics has observed: the students always have to be taught what they should have learned in the preceding course.. . . [s]He does not learn algebra in the algebra course; he learns it in calculus, when he is forced to use it. He does not learn calculus in the calculus course, either; but if he goes on to differential equations he may have a pretty good grasp of elementary calculus when he gets through. And so on through the hierarchy of courses; the most advanced course, naturally, is learned only by teaching it.

Thus, this book presents quite advanced material that exercises the basics—a Turing machine game in chapter 1 to motivate logic, sets, tuples, and functions in Chapters 2–8; public key cryptography in chapter 7 to review the mathematics of base conversion, logarithms, exponents, floor, and mod; and finite state machine minimization in chapter 12 to demonstrate equivalence relations. Chapters 11 and after break advanced material into many small challenges that exercise the basics; you are encouraged to try to answer these challenges on your own before looking at answers that are hidden in the pdf or in the back of the book. Quizzes and tests for this class examine the basics, and not the advanced material, since you can expect to see it again if you continue on in computer science. (So go ahead and learn it; especially by teaching others in your study group, or in the on-line community for this class.)

Second, there is an emphasis on writing, with the aid of technology, to communicate the truths that we derive to others and to ourselves.* Writing, even mathematical writing, is a creative activity.

*This is not "discreet" mathematics, which is the kind you don't tell others about.

Third, it tries to identify basic, intermediate, and advanced material for each of the topics it covers. (Text in black is basic, regular-size blue is intermediate, and smaller blue is advanced.)

Mastery of the material in the basic, intermediate, or advanced sections corresponds roughly to C, B, or A grades, which my institution (UNC Chapel Hill) officially defines as follows:

C A totally acceptable performance demonstrating an adequate level of attainment for a student at a given stage of development.

B Strong performance demonstrating a high level of attainment.

A Mastery of course content at the highest level of attainment.

Of course, most students do not think of 'C' as *totally acceptable*, and neither should you. But to reach the highest level of attainment, one must build on a solid foundation. So, if a chapter topic is new to you, start by mastering the basics, which usually involves a bit of memorization and trying out the basic exercises. Then reread the basics and work on the intermediate sections, which apply these basics, explain choices made, and develop vocabulary and style for communication. The advanced sections show how the topic branches out to connect to new applications or deeper areas of mathematics. Unless your class already knows the basics well, expect your instructor to pick and choose from the advanced sections because there is more content in this book than fits comfortably in a semester.

Some other books are "proof first," starting with proofs in number theory, algorithms, or graphs. This book tends to be "count first," using counting questions to ensure that a new definition has been understood. There are more encyclopedic books,* but they don't always motivate the basics by connections to advanced topics or applications, they serve the instructor's needs more than the students' needs, and can be costly.

*I like Epp's "Discrete Math with Appl," 4th ed. [2] best. Rosen's "Discrete Math and Its Appl," 4th–7th ed. [22] are good, too.

Wikipedia is an excellent resource if you want a clear exposition of a certain topic, thanks to some selfless mathematicians and computer scientists on the web. What a book needs to do in the internet age is to introduce the topics, encourage you to practice to become familiar with them and their vocabulary, and open new vistas for further exploration, whether in subsequent classes or on your own.

Thanks to many:

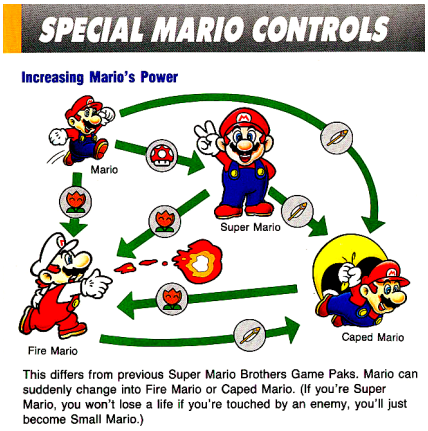
Authors and collectors of puzzle problems: Martin Gardner, Peter Winkler

Latex package writers: D. P. Story,

Software developers: Kai Leith

Proofreaders: Shawn Brown, Clinton Freeman, Aram Han, Steven Love, David Millman, Matt O'Meara, Nada Rahmouni, William Snoeyink, Vishal Verma

Chapter 1
Introduction to Discrete Structures & Problem Solving



Already in 1989, the Super Mario World instructions included this finite state machine diagram and expected eight-year-olds to understand it.

—Nick Pippenger

The mathematics that underlies computation also underlies recreational mathematics, so what better way to begin than with a programmable puzzle game, Kara.

Objectives: By studying this chapter and playing with Kara, you will be able to see that a simple model of computation depends upon logic, sets, functions, tuples, and graphs, all of which we will learn to define formally in the next few chapters. You will also explore problem-solving steps that will become very familiar through the rest of this book.

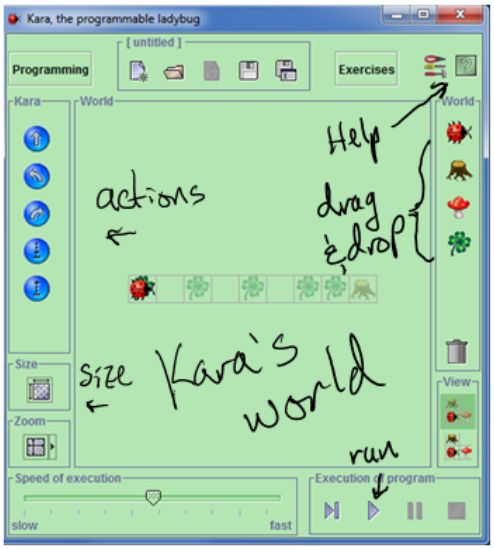


Figure 1.1: Kara's world editor: Click ? for help, Exercises for puzzles or Programming to create solutions.

1.1 Defining Kara's world

Kara is a ladybug that wanders a two-dimensional grid world, picking up or dropping clovers. You can give Kara a set of internal states, including a start and a stop state, and a list of rules that, depending on her current state and what her sensors tell her about the clovers, tree stumps and mushrooms in her immediate neighborhood, dictate her next actions and movements.

Kara is part of a suite of educational software from Raimond Reichert's doctoral dissertation [20], supervised by Jürg Nievergelt at ETH Zürich. I encourage you to download this to any computer on which you have or can install java, either from swisseduc or from `www.cs.unc.edu/~snoeyink/`

`kara`. Double-clicking kara-en.jar should show Kara's world, as in figure 1.1. Try some exercises; I'll wait.

Kara is a variant of a *Turing machine* (TM for short), a simple model of computation named after Alan Turing that incorporates most of the discrete structures that we will meet formally in the chapters that follow.

logic Kara's sensors report whether she has a clover beneath her, a tree to either side, or a tree or mushroom ahead. Each sensor provides a *logical predicate*, a variable whose value is either true or false. Chapters 2 and 4 define concise notation for predicate logic and first-order logic, which we use to make precise, unambiguous definitions and statements in this book.

sets The foundational discrete structure is the *set*, which is an unordered collection of distinct *elements*, defined in subsection 3.1.1 and expanded on in chapter 5. Kara has her finite set of states, the sets of sensors used by each state, and the set of symbols, or alphabet, that can be placed in her world.

tuples, strings A *tuple* is an ordered list with a fixed number of elements, as defined in subsection 3.1.2. Kara herself would be described as a tuple consisting of a set of states, a function encoding the rules, and special start and stop states. Kara's tape is a *matrix*, which is just a specific two-dimensional arrangement of a rather large tuple. Lists, sequences, and strings may have varying numbers of elements; e.g., each of Kara's rules contains a *string* of actions, possibly empty. We need *recursion* from chapter 8 to formally define operations on structures of varying size.

functions, relations Kara's rules are *functions* that map an input state and sensor readings to a new state and a string of actions. You are familiar with many mathematical functions, though the formal definition of a function $f : A \to B$ as a special case of a relation in chapter 6 may be new to you.

graphs Kara is programmed by drawing a *graph* in which *vertices* representing states are connected by *edges* representing rules. Graphs give us useful visual encodings of relations, and are good models for many puzzles and problems, as we shall see in chapter 13.

Other discrete structures important to computer science, including languages, grammars, finite state machines, and Turing machines, will be defined primarily to exercise these basic structures. By defining a common vocabulary for these discrete structures, we will be able to communicate our ideas precisely to the computer, to other people, and, because we think in language and symbols, to ourselves.

1.2 On problem solving

This book is full of mathematical puzzles and problems because practice is the best way to become a good problem solver. Whether you think of yourself as "good at math" or "not so good at math," you can improve by learning some tricks and then spending a lot of time in practice and a little time in introspection, observing what works for you.

Two very readable books on problem solving are M. Gardner's "Aha! Insight," [6] which has simply stated problems that require insight for their solution, and discussion on how to develop this insight, and G. Polya's "How To Solve It" [19] which begins with general advice and heuristics for solving mathematical problems then gives specific advice under headings from "Analogy" to "Working Backwards." Mixed in with serious topics like "Induction" are less serious topics, such as his characterization of "The traditional mathematics professor... He writes a, he says b, he means c; but it should be d."

A math professor talks in someone else's sleep.

Let me include some problem-solving advice, gleaned from these and other sources, and illustrate it on an example problem. Let's teach Kara (or your favorite TM) how to count in binary (base 2).*

*If you haven't yet tried a Kara exercise, take a 15 minute break to do so.

- Allocate time. Read a problem early, even if you don't plan to write the answer until later.
- Express the rules of the game that you are playing in precise language.
- Work out a few examples on scratch paper. So get some scratch paper; look to reuse printer paper that has one clean side.
- Consider changing the problem: If you discover by doing examples that the problem specification is incomplete, what do you think should be added? If the problem is too hard initially, can you start with a simplification?
- Notice the assumptions and decisions that you make; you will want to justify these when you write down your solution, so start making notes now.
- Draw diagrams. Visual representations of a problem or model can often bring new insight.
- Break the task into smaller tasks, then solve these smaller tasks. This is where the real work is done.
- Ask yourself at each step, "What can go wrong to keep me from completing this step?" Create new worked examples that demonstrate the right thing to do in each case.
- Test your formula or program using your worked examples.
- Review your solution. Check that it solves the problem as originally stated. Next, can you simplify it? Can you generalize it so that it solves more than the problem originally stated?
- Document the key insights in and from your solution, both for yourself

and others.

Allocate time. Read the problem right away, and decide whether it is a straightforward problem in which you put into practice things that you already know, a problem in which you first need to read up on some new system or concepts, or a problem in which you will need an "Aha!" insight. For the problem of counting in binary, we will need to know a little more about Kara and do some counting by hand on scratch paper to gain insight into how counting works.

When reading a problem, try to observe features that may become important. Allow time for these features to percolate in your brain while you do other tasks, whether related, like working examples on scratch paper, or unrelated, like taking a shower. For example, we can observe that Kara decides her next action on a limited amount of information—the symbols at or adjacent to her current cell, and her current state. This means that Kara cannot remember large count values without somehow writing them down on the tape.

We humans also need to write things down; allow time for writing, testing, and rewriting a solution. Writing as you go can help ensure that you spend time working on a solution, and not just "thinking about the problem," *aka* "spinning your wheels," without making forward (or even backward) progress.

Express the rules in precise language. Other limitations on Kara can be discovered in the documentation or by experimenting with programs: Kara is not allowed to walk into trees, to pick up a clover that isn't there, or to put a clover down if one is already there. Kara "crashes" if her sensor pattern fails to match any of her rules or matches more than one rule (in the default "deterministic mode"); these are deemed programming failures.

Kara's world, her two-dimensional tape, is finite (not infinite as in a standard TM), but has wrap-around—if Kara goes off one side, she reappears on the parallel side. Most of the examples have a ring of trees around Kara's world to prevent this. Kara can start at any place in this world, but by default starts facing to the right.

Computer systems have rules that they will enforce by crashing or doing the wrong thing. Type-checking is designed to help us by catching programs that would break rules before they do. Mathematical systems also have rules, whether on operations (don't divide by zero, or take $\log x$ for $x \leq 0$) or notational conventions (use the equal sign (=) only for things that are really equal, not "equivalent" or "hopefully equal.") As we introduce such rules,* and define the notation to express them precisely, develop your own personal

*It is rules and limitations, not freedoms and capabilities, that stimulate creativity.

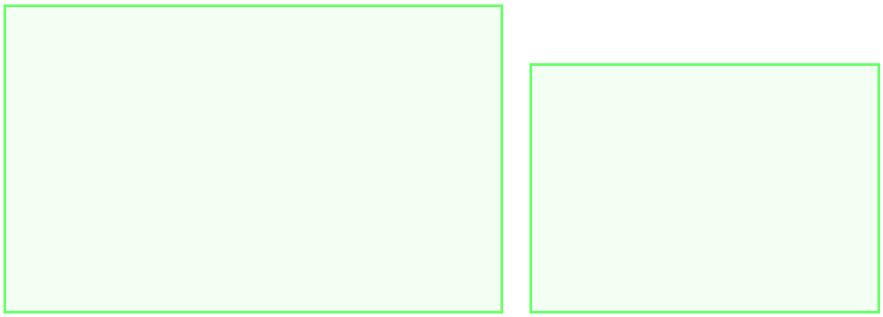


Figure 1.2: After you get your own scratch paper and make your own diagrams for counting in binary and Kara's sequence of actions, you can move or delete the green rectangles to reveal mine. (If my scratch paper is not hidden, then download the pdf for this book and use a viewer that understands annotations, like Adobe or Foxit Reader. Your web browser or tablet likely won't show you the dynamic content.

type-checks that you can use to catch the mistakes that would be made by Polya's "traditional mathematics professor."

Work out examples on scratch paper. Draw diagrams. If we want Kara to count in binary, we should start by figuring out how we can do it ourselves. Pull out your scratch paper and write in a column: 0, 1, 10, 11, 100, . . . , as I've done in figure 1.2, or use blanks for 0 and clovers for 1 as Kara does. Write enough numbers that you can begin to see patterns. Graph paper, or lined paper turned sideways, may help to keep columns straight. Perhaps underline the *bits* (*bi*nary dig*its*) that change from one line to the next.

Change the problem. Remember your decisions. Next, sketch two rows of Kara's world, one with a number and one blank. Then draw the path that Kara would take to fill in the blanks with the next number.

While doing this, you may realize that the initial problem doesn't specify completely how binary numbers should be written in Kara's world; should we write them left-to-right or right-to left? I've chosen left-to-right for my scratch paper diagram. I have also chosen to ring the world with trees like in the examples. Kara can use these to determine where the rows and columns end.

Even for assigned problems, you may find that the statement of the problem is incomplete without additional assumptions. Recognizing this is a good skill, and if you start early enough, you can query the instructor or teaching assistant for their assumptions, just like you can query a client in the real world. If you don't have time for that, make what you think

are reasonable assumptions and clearly document them, and, if graders or clients agree that those could be reasonable assumptions, then they cannot fault you for your solution. It is when you make undocumented assumptions that you may have trouble.

Break the task into smaller tasks. In my diagram of Kara actions, I imagine Kara going through three states:

plus1 Starting at the lowest order bit (rightmost), Kara adds one, which involves putting a 0 in the next row for each 1 above. When Kara hits the first 0 above, she puts a 1 in the next row, then goes to a 'copy' state.

copy Kara copies the bits above to the next row until the left end, then goes to 'rewind.'

rewind Kara moves to the right end of the newly completed row, then goes to 'plus1.'

Before we can write rules for each of these three states, we need to choose which direction we want Kara facing at the start and end of each state. Let's have Kara face right in 'rewind,' and down in 'plus1' and 'copy.' This allows us to start Kara anywhere on the first row in 'rewind', where she can go to the right until she senses a tree ahead, and then turn down before going to 'plus1.' In 'copy', she'll start facing down, but turn to face right just before 'rewind.'

We've broken the larger task into the smaller tasks of deciding on Kara's sensors, rules, and actions for each of these three states. Before reading further, try to complete these tasks yourself: first on paper, then in Kara.

Figuring out how to break a complex problem or system into manageable parts is an essential skill for computer scientists, so we will return to this over and over again. Much of the notation (logic, quantifiers) and many of the concepts (sum and product rules for counting, proof and induction) are to help us precisely break things into simpler parts.

Ask yourself what can go wrong. We haven't told Kara how to stop! Since rewind gets us ready to do the next row, she can check if there is a next row, and stop if not.

What if Kara fills a row with 1s (clovers)? As described above, the plus1 state would never find a 0, and Kara would crash. We could decide to stop, or we could continue so that the next row becomes all blank, like an odometer resets to 0000000. We should make a narrow world to test this; figure 1.3 shows a 6×20.

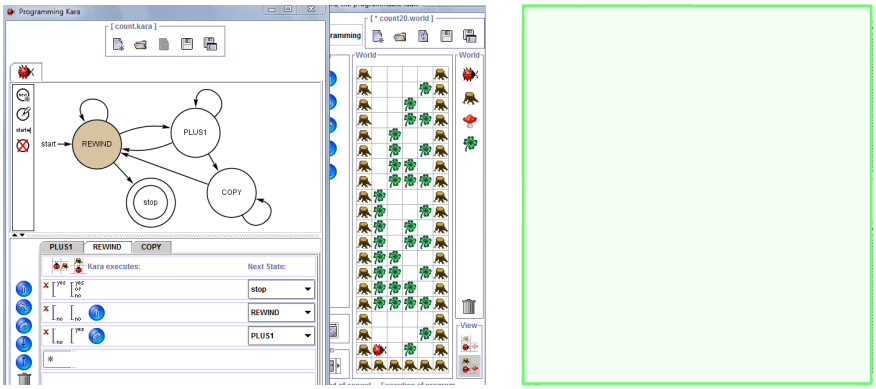


Figure 1.3: A four-state Kara program for counting in binary, and the results

Test your solution. Students are often too credulous—too ready to believe that work intended as a solution must be a solution. (One cause may be that most math and computing problems in school are carefully stated to have simple, or simple to grade, answers.) With powerful tools of math and computing, we can also easily find a solution of the wrong problem.

Cultivate your inner skeptic—your work is an *attempt at a solution* until it has been tested and your belief that you have a solution is justified. Your scratch paper work should give you a collection of worked examples on which to begin testing. Asking, "What can go wrong," should produce more.

Review your solution. We've solved this for Kara's two-dimensional world. In a more traditional TM, the one-dimensional tape extends to infinity on the right, and starts with the head at an end-of-tape mark on the left. Here I would choose to write numbers in reverse, with the low order bits on the left. Counting could then just overwrite bits in the plus1 state and rewind, never needing to copy the unchanged bits because they would already be present on the tape.

Mathematicians think solutions are answers, but chemists know solutions are all mixed up.

Document your solution. Keep notes on definitions, examples that helped you understand them, and insights you gain in working out practice problems. Collect and organize your diagrams and notes as you edit the final version of your results.

1.3 The purpose of puzzles

Studies by K. Anders Ericsson indicate that the main difference between people with adequate and exceptional performance in several areas is the

amount of practice directed toward the performance. Spend the time to develop habits of precise thinking and healthy skepticism. These are important for

computer science students: Many students begin by programming "experimentally"; combining commands almost randomly until they get something that seems to work. If instead you refine your understanding of the task in precise language, you can far more quickly refine towards a correct program, including test sets to verify its correctness. chapter 11 demonstrates deriving the steps of important algorithms.

puzzle and problem solvers: You can develop and improve your abilities to solve both puzzles and serious problems by using precise language to record what you've done so far, which helps you stay out of ruts, as you explore what the problem looks like from different points of view. Often the right perspective, model, or diagram can make a puzzle or problem suddenly reveal a path to a solution. section 1.2 collects problem solving hints that are also scattered throughout the book.

those working under contract: When you negotiate a contract with a client, both of you must share common vocabulary to agree on the specifications of the task to be done. This remains important at the end, when you demonstrate that you have done what is specified, and that if there is a specification change then you are owed more money.

developers of critical systems: Computers control aircraft, medical devices, and Mars landers are systems in which glitches have already cost human lives and/or staggering amounts of money. Formal analysis methods are used to prove that a system cannot become stuck in an infinite loop, or overdose a patient—one cannot simply hope that all bugs have been found.

documenting for code maintenance: Code specifies what is done, but to maintain code you want to know why and what are the limitations. Formal development methods identify and document those, making code easier to maintain.

anyone listening to politicians: Many policies and actions are justified by the intended consequences, when the actual consequences may be quite different. There can be a gap between "I wrote a program that will measure similarity between faces" and "I wrote a program to compare pixels that I hope will measure similarity between faces;" there is almost certainly a gap between, "I will vote for policies that will end unemployment" and "I will vote for policies that I hope will increase employment in some sectors."

1.4 Summary

Kara, a computational puzzle game, depends on many of the discrete structures that we will define formally in this book.

Puzzle solving skills can be developed if you consciously work at them. Finding an answer is just one step; there are steps you can take before and after you find an answer that can deepen your understanding of a problem and its solution.

Mathematics is a creative activity. You need to learn its language and idioms to be able to express yourself precisely, and that takes effort and practice, much like any other language. But new languages give us new ways to understand the world around and within us, to communicate, and to create – all of which are fundamental to being human.

1.5 Exercises and Explorations

Quiz Prep 1.1. Be able to explain the goal of and give an example of each of these problem solving steps.

1. Allocate your time.
2. Learn the rules of the game.
3. Work out examples on scratch paper.
4. Consider changing the problem.
5. Note the assumptions and decisions.
6. Draw diagrams.
7. Break the task into smaller tasks.
8. Ask "What can go wrong?" and create new examples.
9. Test using your worked examples.
10. Review your solution.
11. Document the key insight, both for yourself and others.

You could think of examples in one of these four contexts, or make up your own: selecting when to take what courses to satisfy requirements, going to a movie by borrowing a car or convincing a friend to drive, winning a level of your favorite video game, or writing a Kara program for an exercise.

Exercise 1.2. Download Kara and do one or more exercises. Create the Kara program for counting in binary by working through section 1.2.

Extension 1.3. Gray code. You want to design a *position coder* that can report the angle of a rotating shaft. You could encode 2^b positions by striping the shaft with the b-bit binary count pattern from figure 1.2 in b concentric circles and positioning b brushes or photodiodes to sense the bit pattern. Unfortunately, adding or subtracting 1 may change many bits of a binary number, and in a mechanical device they never all change simultaneously. Thus, we get spurious readings as the shaft rotates from one count to the next.

Frank Gray observed that the same 2^b b-bit binary numbers could be put into a circular order so that from one number to the next only a single bit changes: toggling from 0→1 or 1→0. His *reflected binary Gray code* can be constructed as follows. For one bit, use order 0,1. For two bits, use order 00,01,11,10, which places 0s in front of the one-bit code, then 1s in front of the reverse of the 1-bit code. For k bits do the same: write 0s in front of the $(k-1)$-bit code, then 1s in front of the reverse of the $(k-1)$-bit code. On scratch paper, try writing the orders for the 3-,4-, and 5-bit codes, then check figure 1.4.

Program Kara (or another TM simulator) to list binary numbers in Gray code order. You may do this on a 1-d tape by toggling one bit for each new

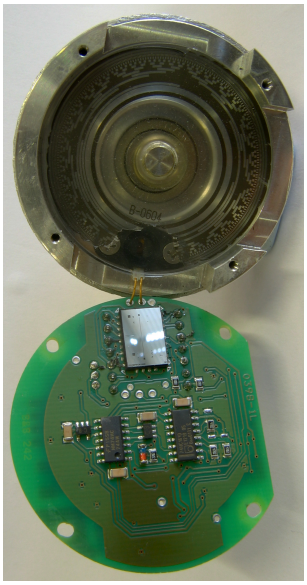


Figure 1.4: A rotary position encoder; zooming in shows a 13-bit Gray code. Initially hidden in pdf is a 5-bit Gray code.

number, or by copying rows on Kara's 2-d tape. What is the pattern for which bit should be toggled next? ▶

Exploration 1.4. *Busy Beaver Turing machines* fill an initially blank tape with a string of as many clovers as possible, and then halt. (Halting is the challenge – it is trivial to make Kara run forever filling the tape with clovers.) Kara is a little different from the standard Turing machine, but if we use an $n \times 1$ world and limit the number of move-ahead actions, we can create a Kara version of the Busy Beaver problem:

- Begin with at least an 18×1 tape with a mushroom at the left end and tree at the right and Kara just to the left of the tree. (Kara will face the tree in this configuration, so she can sense the tree is there—she wouldn't be able to do so if we started with Kara on the left end.)
- Create a Kara program with only one new state (plus the stop state). Your program should write as many clovers as possible, but still halt with Kara in the stop state (not by crashing.)
- Since Kara can do several actions for one rule, let's say that no string of actions can contain more than 3 ⇑ (move ahead) actions.
- Do not use the mushroom sensor. If you crash into the mushroom, make the tape longer, since the tape is supposedly infinite in that direction. If you continue to crash, then your Kara program probably has an infinite

loop.

1. Use a single new state. With one rule it is easy to write 4 clovers and stop; with two rules it should be easy to write 7. I know 3 different ways to write 10 clovers and stop. Can you beat my best single-state machine, which writes 14 clovers and stops? (Remember, the "and stop" is what makes this a challenge.)

2. What is the best you can do with two new states? (Mine is 58, which I doubt is the best possible.) Start with an $n \times 1$ world with large n so Kara does not crash into the mushroom at the left end.

Chapter 2
Propositional Logic

'Contrariwise,' continued Tweedledee, 'if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic.'

—Lewis Carroll, *Through the Looking Glass*

Precise reasoning is based on logic, so we begin by introducing propositional logic, which allows us to determine the truth or falsehood of statements such as, "If it does not rain in the morning, I commute by bike."

Objectives: After studying this chapter and related exercises, you will be able to demonstrate your ability to express statements in symbolic form, using the logic operations of negation, and, inclusive or, implies, if and only if (iff), and exclusive or (and their symbols: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ and $\oplus$) to express statements without ambiguity. You will be able to distinguish between inclusive and exclusive 'or.' You will be able to create truth tables in order to recognize tautologies and logically equivalent expressions, including de Morgan's laws and conditional expressions.

You will also meet properties of logic operations and rules of logical inference, both of which can help us rewrite expressions while preserving their truth values. You will encounter Boolean algebra and circuit notations for the same logical expressions. Finally, you will be able to demonstrate how to use at least one of these methods to solve a logic puzzle.

The goal in this chapter is to reduce reasoning in propositional logic to be purely mechanical—something that a computer can do. For now, we are willing to sacrifice creativity for precision; we want to agree on the truth or falsity of statements with no ambiguity.

2.1 Logic operations

We start with *propositions*, which are statements that are either true or false. We can abbreviate example statements with lowercase letters (*logic variables*), e.g., let p = "It rained this morning," and q = "I commuted by bike." We can write down all possibilities for whether p and q are true or false in a *truth table*. A truth table for n variables has 2^n rows, which are all the distinct ways to assign these variables true or false. We can generate the rows systematically by counting in binary, as we had Kara do in section 1.2.

To make columns quickly, first determine the number of trues.

The truth table in table 2.1 defines the operations *not*, *and*, *or*, *if*, and their symbols. The statement, "If it does not rain in the morning, I commute by bike," becomes, "if not p then q", or "(not p) implies q," and is written in symbols as $\overline{p} \rightarrow q$.

Mnemonic for $\wedge$ $\vee$:
'and':'a$\wedge$d': both true,
'or':'o$\vee$': at least one true.

Table 2.1: Truth table for logic operations

		not	and	or	implies	xor	tautology	contradiction	De Morgan	
p	q	$\overline{p}, \neg p$	$p \wedge q$	$p \vee q$	$p \to q$	$p \oplus q$	$p \vee \overline{p}$	$q \wedge \overline{q}$	$\overline{\overline{p} \vee \overline{q}}$	$\overline{\overline{p} \wedge \overline{q}}$
T	T	F	T	T	T	F	T	F	T	T
T	F	F	F	T	F	T	T	F	F	T
F	T	T	F	T	T	T	T	F	F	T
F	F	T	F	F	T	F	T	F	F	F

The notation for these operations varies in some books and most programming languages. E.g., for 'not,' I tend to write overline, but use $\neg$ with parentheses online.* Programming languages often use '!' or '~' for 'not.' *Boolean algebra* and circuit design often use '0' and '1' for *False* and *True*, with multiplication ($\cdot$) and addition (+) for 'and' and 'or.'

*Overline avoids parentheses, $\overline{p \wedge q} \equiv \neg(p \wedge q)$, but some browsers can omit thin lines.

The interpretation of these operations, however, is quite standard. In mathematics, the statement $p \wedge q$ is true only if p and q are both true; it is false otherwise. (One way to be true; three ways to be false.) The statement $p \vee q$ is an *inclusive or*, which is true if p, q, or both are true; it is false only if both p and q are false. (Three ways to be true; one to be false.) If we want exactly one of p or q to be true, we must use *exclusive or*, $p \oplus q$. (Two ways to be true, and two ways to be false.)

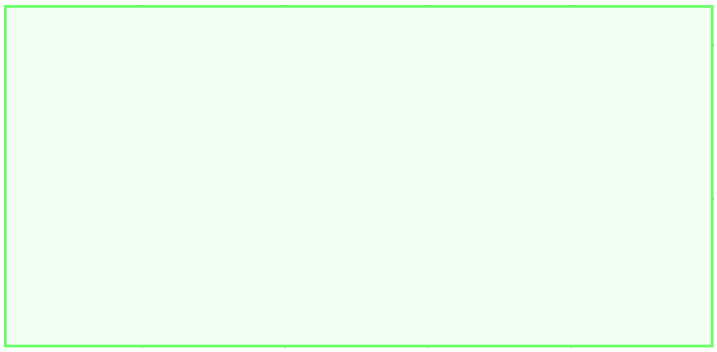
Parentheses are used to specify the order of operations; if omitted, then by convention we first evaluate 'not,' then 'and,' then 'or,' then 'implies,' and last 'iff,' just as we evaluate multiplication before addition. (XOR's may be lumped with 'or' or 'iff,' depending on your programming language.) I parenthesize to avoid memorizing order of precidence for logic operations.

English language usage for "or" is often ambiguous, forcing us to guess based on meaning. If I say, "I left my keys in the car; I hope the driver's or passenger's door is unlocked," my listener assumes I'd be pleased to find both doors unlocked (inclusive or). But if I say, "Either it rains, or I commute by bike," my listener is likely to assume a causal connection, and consider it an exception if they discover that I did bike on a rainy day. And if I habitually bike on rainy days, they might even accuse me of lying. Translating English into notation forces us to remove the ambiguity.

If I say, "I like my pizza with pepperoni and mushrooms or olives," do I like a pizza with all three toppings? How about with just olives? How many different pizzas do I like? The answers depend on whether "or" is inclusive or exclusive, and whether it is evaluated first, or the "and" first, since that has precidence. Make a truth table with variables p, m, o for pizzas with pepperoni, mushrooms, olives, and the four possibilities: $p \wedge (m \oplus o)$, $p \wedge (m \vee o)$, $(p \wedge m) \oplus o$, and $(p \wedge m) \vee o$, to see that there can be 2, 3, 4, or 5

different pizzas, depending on interpretation. (To make 3 the more natural interpretation, I'd actually say, "I like my pizza with pepperoni and either mushrooms or olives.")

Table 2.2: A truth table showing, for the 8 different pizzas with 3 toppings, there can be 2, 3, 4, or 5 pizzas that satisfy, "I like my pizza with pepperoni and mushrooms or olives." Each row represents a different pizza. In each column, the small T/F is the output of the two-variable expression in parentheses; the large T/Fs combine this with the third variable. This is a compact way to do one operation at a time.



In this book, "or" should always be interpreted as inclusive or, unless the phrase "and not both" is added to indicate exclusive or. In mathematics, neither inclusive nor exclusive 'or' implies any causal connection: either it rains or I ride my bike says one or the other or both happen – it doesn't mean that lack of rain causes me to bike, any more than not biking causes it to rain. (There are temporal and modal logics that capture the subtleties of possibilities, necessities, and exceptions. These are beyond the scope of this book, even though they are crucial for showing that complex distributed systems do not get stuck waiting for resources, and give rise to nice puzzle problems like the Dining Philosophers or Byzantine Generals.)

An expression whose truth table column is all *true* is a *tautology*; $p \vee \neg p$ is the simplest tautology. A expression with all *false* is a *contradiction*; $q \wedge \neg q$ is the simplest contradiction. (The goal in mathematics is to find tautologies; statements that are always true. Sometimes we use proof by contradiction, since the negation of a contradiction is a tautology.)

Two expressions are *logically equivalent*, indicated by $\equiv$, if and only if their corresponding columns in the truth table are identical. The simplest equivalence would be double negation $\overline{\overline{p}} \equiv p$.

We may replace any expression by a logically equivalent expression. We need not think about whether the statements themselves are true or false – replacing logically equivalent expressions ensures that true statements

Be alert for notations and concepts that save thinking!

remain true and false statements remain false. By writing statements in logic notation, we can often recognize equivalents and simplify expressions purely by symbol manipulation.

For example, in the last two columns of table 2.1, you can observe two equivalences known as de Morgan's laws for *and*($\wedge$) and *or*($\vee$):*

$$p \wedge q \equiv \overline{\overline{p} \vee \overline{q}}$$
$$p \vee q \equiv \overline{\overline{p} \wedge \overline{q}}.$$

*The first says, "p and q are both true iff neither p is false nor q is false."

Negating both sides gives the usual way to write them:

$$\overline{p \wedge q} \equiv \overline{p} \vee \overline{q}$$
$$\overline{p \vee q} \equiv \overline{p} \wedge \overline{q}.$$

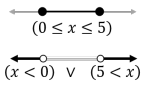
So, mathematically, saying "p and q aren't both true" is equivalent to saying "p is false or q is false."

de Morgan's laws are worth memorizing; I remember them as a purely symbolic manipulation: "To negate an AND or OR statement, flip the $\wedge/\vee$ and negate both sides." They allow me to rewrite expressions without thinking about the meaning, even when p or q may be more complex. For example, to test if a pizza is not one of my favorites (which have pepperoni and either mushrooms or olives), we can apply de Morgan twice: first to $\overline{p \wedge (\ )}$, and second to the $\overline{m \vee o}$ inside the parentheses.

$$\overline{p \wedge (m \vee o)} \;\equiv\; \overline{p} \vee \overline{m \vee o} \;\equiv\; \overline{p} \vee (\overline{m} \wedge \overline{o}).$$

Thus, if a pizza lacks pepperoni, or lacks both mushrooms and olives, it is not my favorite.

Logic operators may even be hidden in mathematical abbreviations: the condition $0 \le x \le 5$ means $0 \le x$ *and* $x \le 5$. The negation of this entire condition is not ~~$0 > x > 5$~~; no values x satisfy that! Instead, the hidden "and" becomes "or" by de Morgan:



$$\overline{(0 \le x) \wedge (x \le 5)} \equiv (\overline{0 \le x}) \vee (\overline{x \le 5})$$
$$\equiv (0 > x) \vee (x > 5).$$

de Morgan also explains why the logic negation $\overline{0 \le x} \equiv 0 > x$; We negate "0 is less than or equal to x,' to get "0 is not less than AND not equal to x," which means "0 is greater than x."†

†Don't confuse logic negation of $x \le 5$ with (arithmetic) negation of both sides: $-x \le -5$ true for exactly the same xs as $x \le 5$, but the logic negation swaps T/F for all reals x.

By drawing the number line, we see that our negation reveals the two cases in which x lies outside the intarval $[0, 5]$. But even for negating $a \le x \le b$, we don't have to think about the meaning if we recognize the hidden "and", then apply de Morgan's law.

2.2 Conditionals

Let's further explore the *conditionals*. "If p then q," denoted $p \to q$ and read as "*p implies q*," is false whenever p is true and q is false, and true otherwise. (Three ways to be true, and one way to be false.) In fact, $p \to q$ is logically equivalent to $\overline{p} \vee q$, as seen in table 2.3. By de Morgan, $\overline{p \to q} \equiv p \wedge \overline{q}$, which agrees with the above: the only way $p \to q$ is false is if p is true and q is false.

Memorize this equivalence.

Table 2.3: Truth table for conditionals including iff = if and only if

		if		contra-pos	converse	inverse	iff	
p	q	$p \to q$	$\overline{p} \vee q$	$\overline{q} \to \overline{p}$	$q \to p$	$\overline{p} \to \overline{q}$	$p \leftrightarrow q$	$p = q$
T	T	T	T	T	T	T	T	T
T	F	F	F	F	T	T	F	F
F	T	T	T	T	F	F	F	F
F	F	T	T	T	T	T	T	T

For statements p = "it is raining" and q = "I commute by bus:"

Conditional:	$p \to q$	If it is raining, then I commute by bus.
Contrapositive:	$\overline{q} \to \overline{p}$	If I don't commute by bus, then it isn't raining.
Converse:	$q \to p$	If I commute by bus, then it is raining.
Inverse:	$\overline{p} \to \overline{q}$	If it isn't raining, then I don't commute by bus.

Suppose that I promise, "If you loan me your car, I'll drive you to the airport." What can happen? You could loan me your car: if I then drive you, I have fulfilled my promise – my statement is true. If I don't drive you, then I broke my promise and my statement is false. On the other hand, if you do not loan me your car (because it is in the shop for repair), then I claim I have already kept my promise, whether I get my car to drive you, or you take the bus. When the premise (you loan me your car) is false, then the statement is considered *trivially* or *vacuously true*. So, "either I drove you to the airport or you didn't loan me your car (and possibly both)." The two English sentences in quotes may seem different, but in mathematics they are logically equivalent and can be substituted for each other.

Let me say it again: whenever p is false, we declare $p \to q$ to be (vacuously) true. If you haven't seen this idea before, it will take some time to get used to.[1] A puzzle in the margin demonstrates this. The Wason selection task lays out 4 cards that each have a number on one side and a letter on the other. It asks which card(s) you must flip to test the condition that, "if a number is even, then the letter on the opposite side is a vowel." ? Only 10%

[1] See what I did there? I'm saying that if you have seen the idea of being vacuously true, then you may or may not need time to get used to it (again or still), but if you have not seen it before, then you should expect to take time to get used to it.

of people tested identify the correct cards, though afterward most will agree with the explanation of which cards are correct.

People have a much higher success rate when the statement involves a social privilege with a condition. For the four cards in the margin with a beverage on one side and age on the other, which card(s) must you flip to test the condition that, "if someone is drinking beer or wine at a bar, they must be at least 21 years old." ?. So think about these as "if you are enjoying a privilege legally, then you must have fulfilled the condition," which is equivalent to "if you have not fulfilled the condition, then you must not be enjoying the privilege legally."

The logician sent to the store was instructed, "Buy a box of butter, and if they have eggs, buy a dozen." He returned with 12 boxes of butter and said, "They had eggs."

In addition to "if p then q" and "p implies q," several other English expressions translate to $p \to q$: "whenever p, q," "q if p," and "p only if q." That last expression says that the only way that p can be true is for q to also be true, so it might be better to translate it, "if q is false then p is false," $\overline{q} \to \overline{p}$. This is the *contrapositive* of $p \to q$; in table 2.3, you can see that the conditional and the contrapositive are logically equivalent. You can also see this by replacing the "implies" by its equivalent "or" statement: $\overline{q} \to \overline{p} \equiv \overline{\overline{q}} \vee \overline{p} \equiv \overline{p} \vee q$.

Negative poets write inverse; Retro poets wear Converse.

The expression 'p if q' means $q \to p$ and is the *converse* of $p \to q$. The expression "if p is false then q is false" could translate to $\overline{p} \to \overline{q}$, which is the *inverse* of $p \to q$. The names are not so important, but recognizing which pairs are logically equivalent is. Memorize this: to form an equivalent to a conditional statement, negate both sides and swap the order (contrapositive). If you just swap, or just negate, you get converse and inverse, which are equivalent to each other, but not to the original conditional, since $q \to p \equiv \overline{q} \vee p$.

The *biconditional* expression 'p if and only if q' is common. It combines the conditional (p only if q) and converse (p if q), and is logically equivalent to $(q \to p) \wedge (p \to q)$, to $(\overline{q} \vee p) \wedge (\overline{p} \vee q)$, to $(p \wedge q) \vee (\overline{p} \wedge \overline{q})$, to $\overline{p \oplus q}$, and even to $p = q$, though we avoid using that for potential confusion with *assignment*. The biconditional is abbreviated 'p iff q' and written in propositional logic as $p \leftrightarrow q$. (Two ways to be true; two ways to be false.) It has the lowest precidence, but I typically use parentheses so you need not remember that.

Note from the truth tables that the negation of "iff" is the exclusive or: $p \leftrightarrow q$ is true iff the Boolean (T/F) values are the same, and $p \oplus q$ is true iff the Boolean values are different.* It can be helpful to think of $\oplus$ as computing parity: any expression using just $\oplus$ outputs true iff the number of trues in the input is odd. Likewise any expression using just $\leftrightarrow$ outputs true iff the number of falses in the input is even.

*The position of $\oplus$ in the operator precidence list ($\neg$, $\wedge$, $\vee$, $\to$, and $\leftrightarrow$) is not stardardized. It is always after $\wedge$, but use parentheses to disambiguate with other operators.

Continuing to illustrate with social privilege, at the time I am writing, to apply for a North Carolina driver's license (without first getting a learner's permit), you should be at least 18, have proof of identity, and, if you were born outside the US, you need proof of residence in North Carolina. Let a

be an aspiring applicant's age, let i be true iff they have proof of identity, b be true iff they were born in the US, and r be true iff they have proof of NC residence. Finally, A is true iff they may apply. Assuming I have been complete in the conditions, $A \leftrightarrow ((a \geq 18) \wedge i \wedge (\overline{b} \rightarrow r))$ will always be true.

- My son is over 18, has proof of identity, was born in Canada, but has proof of NC residence. He can apply.
- I am over 18, have proof of identity, was born in the US, and have proof of NC residence (which I won't need). I can apply.
- My daughter is over 18, has proof of identity, was born in the US, and does not have proof of NC residence. She can apply.
- The only people at least 18 with proof of identity that cannot apply are those not born in the US that do not have proof of NC residence.

Note that the conditions for my daughter and I can be expressed concisely because "if you were born outside the US, then ..." is considered vacuously true for those born in the US. The alternative is the verbose $(b \wedge (a \geq 18) \wedge i) \vee (\overline{b} \wedge r \wedge (a \geq 18) \wedge i)$. The expression length would grow exponentially if we would add more "if/then" rules. This is why we agree to interpret $F \rightarrow q$ as *vacuously T*; it is a convenience that lets us state conditions more simply.

Three symbols that students often confuse are $\leftrightarrow$, $\equiv$, and $=$.This is not a surprise because they represent the same concept (equality) in different contexts. Consider plugging each of the three symbols in place of the squares in three contexts: between logic variables p,q, between logic expressions, and between numbers or structures A,B.

	$p \square q$	$(\overline{p \rightarrow \overline{q}}) \square (p \wedge q)$	$A \square B$
$\leftrightarrow$	✓	∘	×××
$\equiv$	××	✓	×××
$=$	×	××	✓

$\leftrightarrow$ is a logic operator that will be T or F depending on its inputs, so it makes perfect sense with logic variables (✓), and no sense with numbers (×××). It combines two logical expressions into a single one (∘).

$\equiv$ is shorthand for "I claim that, in a truth table, the columns for these two statements are identical." (✓). The $\equiv$ symbol has other uses in mathematics, but I use it only for logical equivalence; $\Leftrightarrow$ is a synonym.

$=$ is either a claim of equality for the structures on the right and left, or an assignment of the value on the right to the variable named on the left (✓). Words in the context should disambiguate: "If $x = y$" versus "Let $x = y$." The equals sign can be used for logic variables (×) but, unless you are working in a spreadsheet, the other symbols are less ambiguous. It should not be used for logical equivalence of expressions (××).

2.3 Logic puzzles

Relevant solution steps: Start early. Use scratch paper. Break into tasks. Define notation.

Truth tables can be useful in many logic puzzles. Get your scratch paper

ready, and try these two. Understanding what is being asked is the first step.

Q1: On an island often visited by Raymond Smullyan, every inhabitant is a "knight" who always tells the truth, or a "knave" who always lies. You meet three inhabitants, *Alice*, *Bob*, and *Chris*, who are unfortunately not dressed in their characteristic gear. However, two of them make statements:

Alice says: Bob is a knave or Chris is a knight.
Bob says: Alice is a knight if, and only if, Chris is a knave.

Can you determine uniquely what each of Alice, Bob, and Chris are?

Hints:

We'll work through this in several ways in section 2.6, but you can already solve it using a truth table if you: 1) Define variable names for convenient propositions, which may be true or false. (Later, we hope to determine which). 2) Transform what Alice and Bob say into statements that must be true no matter whether they lie or not. 3) Make a truth table to check if there is a unique row that makes both statements true.

While you think about that one, here is another from "Test Your Logic," by George J. Summers [24], that we can solve by truth table in detail.

Q2. When Cora was killed by poison, Anna and Beth were questioned by the police about the manner of her death. They stated:

Anna: If it was murder, Beth did it.
Beth: If it was not suicide, it was murder.

The police made the following assumptions, which subsequent developments revealed were correct.

1: If neither Anna nor Beth lied, it was an accident.
2: If either Anna or Beth lied, it was not an accident.

What is the manner of Cora's death: accident, suicide, or murder?

Let's identify three propositions for the manner of Cora's death that we abbreviate as single letters: a =accident, s =suicide, and m =murder. Add one more, B =murder by Beth, and we can directly translate the statements of Anna and Beth and the police assumptions into symbols.

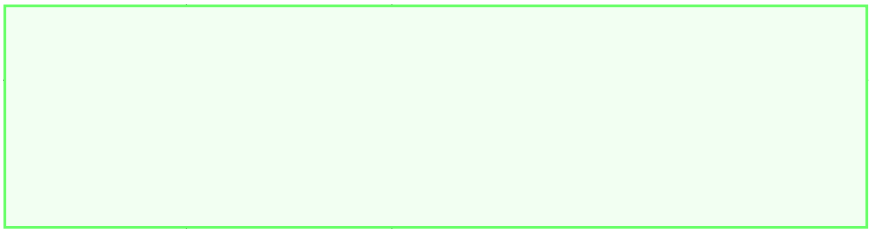
Anna:
Beth:
1:
2:

With four logic variables, it initially appears as if we need $2^4 = 16$ rows in our truth table, but note that most combinations are impossible—if Cora's death is an accident, then by definition it is not suicide or murder. Make a truth table to find the answer to this puzzle. Moving the rectangle in table 2.4 can reveal the truth table form, the rows, and the completed table, but try to make your own truth table first. A shorthand that I use to compactly yet

To see what I mean, drag the box covering table 2.4 down to expose the first row.

carefully evaluate an expression with multiple operators is to write truth values for each subexpression in small letters under the corresponding operator.

Table 2.4: A truth table for solving the puzzle of Cora's death



You should find only one way for both police assumptions to be true: Cora was murdered, but not by Beth. Is that what you conclude from your table? (In my table, I have to look for rows that have two large Ts for the rightmost two expressions; add a column that 'and's those expressions to make that search easier.)

Once we have determined the variables and propositions, a truth table is a purely mechanical way to determine which assignments, out of all the possibilities, are consistent with the statements that we know must be true. (With less than a dozen variables it is pretty easy to get a spreadsheet to calculate them, see Exercise 2.16.) An answer from a truth table may be unsatisfying, because the truth table does not help us explain "why?" For working problems by hand, or reducing problems with many variables, it helps to know some logic properties.

2.4 Properties of operations and inference

There are many properties that we can observe about logic operations. For a formal proof of any of these properties, simply create the appropriate truth table and check for equivalence.

Commutative: Order doesn't matter for 'and,' 'or,' 'iff,' and 'xor:'* $p \wedge q \equiv q \wedge p$, $p \vee q \equiv q \vee p$, $p \leftrightarrow q \equiv q \leftrightarrow p$, and $p \oplus q \equiv q \oplus p$.

Associative: Grouping doesn't matter for 'and,' 'or,' 'iff,' and 'xor:'† $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$, $(p \vee q) \vee r \equiv p \vee (q \vee r)$, $(p \leftrightarrow q) \leftrightarrow r \equiv p \leftrightarrow (q \leftrightarrow r)$, and $(p \oplus q) \oplus r \equiv p \oplus (q \oplus r)$.

Distributive: ‡ $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$, $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$, and $p \rightarrow (q \wedge r) \equiv (p \rightarrow q) \wedge (p \rightarrow r)$.

Complement: $p \vee \overline{p} \equiv T$, $p \wedge \overline{p} \equiv F$, $\overline{T} \equiv F$, $\overline{F} \equiv T$.

de Morgan: $\overline{p \wedge q} \equiv \overline{p} \vee \overline{q}$, $\overline{p \vee q} \equiv \overline{p} \wedge \overline{q}$.

* 'implies' is not commutative: check $p \rightarrow q \not\equiv q \rightarrow p$

† 'implies' is not associative: check $p \rightarrow (q \rightarrow r) \not\equiv (p \rightarrow q) \rightarrow r$.

‡ Memorize these only for and/or. Note $(p \wedge q) \rightarrow r \equiv (p \rightarrow r) \vee (q \rightarrow r)$, so is not distributive.

Identities: $p \wedge T \equiv p$, $p \vee F \equiv p$.
Dominance: $p \wedge F \equiv F$, $p \vee T \equiv T$.
Idempotence: $p \wedge p \equiv p$, $p \vee p \equiv p$.
Inverse: Negation is its own inverse: $\overline{\overline{p}} \equiv p$. For 'xor' and 'iff' each element is its own (additive or multiplicative) inverse: $p \oplus p \equiv F$, $p \leftrightarrow p \equiv T$,

I claim that none of these properties is initially inherently interesting, because anything that you can deduce from the properties, you can also deduce from the operations themselves by exhaustively examining all possible inputs. On the other hand, this examination is slow, tedious, and, if you are not a computer but an error-prone carbon-based unit, you will find that knowing and using properties will allow for easier communication. Effort spent learning these properties will be repaid not only in logic but also in other areas because the same properties arise in operations on sets, functions, and more. (And because the same properties arise in different systems, then they do become interesting; whole branches of modern mathematics are about what systems have given properties.)

Note that not all operations have all properties. For example, the implies operation is neither commutative nor associative: $p \to q \not\equiv q \to p$ and $(p \to q) \to r \not\equiv p \to (q \to r)$, and though it distributes over and/or from the right, it does not the left: $(p \wedge q) \to r \not\equiv (p \to r) \wedge (q \to r)$. You can prove these by finding a single row in a truth table where the left and right sides are not equal. I've memorized the properties for 'and,' 'or,' and 'not' ($\wedge$, $\vee$, $\neg$), and reduce other operations to those.

2.4.1 Inference with properties

We can use these properties to derive new facts from given facts. We will say much more about proof and proof format in chapter 9, but here are two brief examples. Suppose that you know that $(p \to q) \wedge p$, i.e., if a morning is clear, I commute by bike, and this morning was clear. We can prove the following equivalence.

$(p \to q) \wedge p$	given information
$\equiv (\overline{p} \vee q) \wedge p$	equivalence for $\to$
$\equiv (\overline{p} \wedge p) \vee (q \wedge p)$	distribute $\wedge$ over $\vee$
$\equiv F \vee (q \wedge p)$	contradiction $\equiv F$
$\equiv (q \wedge p)$	$\vee$ with F is identity

We conclude that knowing the conditional $p \to q$ and its premise p is equivalent to knowing both p and q: that I commuted by bike and that the morning was clear. We can either say that $(p \to q) \wedge p \equiv q \wedge p$, or that $((p \to q) \wedge p) \leftrightarrow (q \wedge p)$ is a tautology.

Does it seem odd to you that from knowing, "If a morning is clear, I commute by bike, and this morning was clear," we conclude, " I commuted by bike and the morning was clear." Why should we repeat that the morning was clear?

The reason is that we are claiming logical equivalence: that $((p \to q) \wedge p)$ and $(p \wedge q)$ have identical columns in the truth table, which is the same as saying that no matter what values we assign to the free logic variables, p and q, we find that $((p \to q) \wedge p) \leftrightarrow (q \wedge p)$ is true. Most of the time we care only about the cases in which the premises are true, and would be happy to observe that from the truth of "If it is clear, then I cycle, and today was clear," we could conclude that "I cycled." On the other hand, knowing that I cycled would not let us conclude that today was clear, which is why we cannot claim equivalence.

A theorem is typically written as an *implication*: "If a is true, then b is true," for some formulas a and b. We write $a \Rightarrow b$, which is also read as "a implies b," as an abbreviation for "whenever a is true, b is also true," which is equivalent to saying that "$a \to b$ is a tautology."

The biconditional and logical equivalence make claims in two directions, while implication makes a claim in only one direction. This makes proving biconditionals more difficult, and laziness suggests that we prove them only when both forward and backward claims are important. This explains why definitions use "iff," but most theorems prove the weaker "if-then."

For the second brief example of a proof, we establish an implication: If $p \wedge q$ is true, then p is true; $(p \wedge q) \to p$ is a tautology.

$(p \wedge q)$	given information
$\to p$	implied by $\wedge$

We have just developed our first two of nine *rules of inference* that are listed in table 2.5. *Modus ponens* says that if we know $p \to q$ and we know p, then we may conclude q; that $((p \to q) \wedge p) \to q$ is a tautology. *Simplification* says that if we know $p \wedge q$, then we know p; that $(p \wedge q) \to p$ is a tautology.

Translate each of these into your own words and you may decide that rules of inference are just common sense given complicated names. In fact, they are shortcuts, allowing us to rewrite expressions without thinking and without making truth tables. We will usually use them without naming them, but if you are ever unsure whether some inference is valid, check it with a truth table.

If you go further in the study of logic, you will encounter four distinct but similar concepts. Let me introduce them with the symbols that are commonly used. Although we focus on the second, you need to know that the other three exist to avoid conflating these.

Table 2.5: Nine rules of inference. If you know the expressions above the line are all true, you may therefore ($\therefore$) conclude the expression below the line is true. Create truth tables to establish these.

Modus Ponens	Modus Tollens	Hypothetical Syllogism	Constructive Dilemma
$p \to q$	$p \to q$	$p \to q$	$(p \to q) \wedge (r \to s)$
p	$\overline{q}$	$q \to r$	$p \vee r$
$\therefore q$	$\therefore \overline{p}$	$\therefore p \to r$	$\therefore q \vee s$

Simplification	Conjunction	Disjunctive Syllogism	Absorption	Addition
$p \wedge q$	p	$p \vee q$	$p \to q$	p
$\therefore p$	q	$\overline{p}$	$\therefore p \to (p \wedge q)$	$\therefore p \vee q$
	$\therefore p \wedge q$	$\therefore q$		

$p \to q$ Material implication: p and q are statements involving logical variables; the result is an expression that is true or false according to the logic operator 'IF.' We don't know whether it is true or false, however, until we know the values of the variables.

$a \Rightarrow \beta$ Logical implication: a and β are statements, and whenever a is true, β is true. Another way to say it is that $a \to \beta$ is a tautology. Material implication is a calculation of T or F within each specific row of a truth table; logical implication is a claim about all rows in a truth table. Although this claim could be true or false, we don't think of it as returning a T/F value, so it cannot be part of a logic expression itself.

$\Gamma \models \tau$ Tautological implication, aka Logically consistent: A statement τ is logically consistent with a set of statements Γ if and only if, in every possible world in which all statements in Γ are true, τ is also true.

$\Gamma \vdash \tau$ Provable from: a statement τ is a theorem of a set of statements Γ if and only if there is a formal proof of τ from Γ. Gödel proved that any system expressive enough to encode statements about itself (standard arithmetic on integers, for example) must have logically consistent statements that cannot be proved.

Russell to Whitehead: "My Godel is killing me!"

A proof system is *sound* if only true things can be proved, and is *complete* if every true thing can be proved. Propositional logic, with the right axioms of inference, is sound and complete, but you must go outside of propositional logic to prove it.

2.5 Boolean circuits

There are 10 types of people: those who understand binary and those who don't.

Computers are built on logic implemented as electronic circuits, which often use 0 and 1 for *false* and *true*. Either 'nand' or 'nor' gates, whose truth tables are in table 2.6, can be used to build any of the logic operations of table 2.1 or table 2.3.

Logic gates can also perform numerical computation: Consider a *half*

Table 2.6: Logic gates

		not	and	or	xor	nand	nor	half adder	
								carry	result
a	b	$\overline{a}$	$a \wedge b$	$a \vee b$	$a \oplus b$	$\overline{a \wedge b}$	$\overline{a \vee b}$	$a \wedge b$	$a \oplus b$
0	0	1	0	0	0	1	1	0	0
0	1	1	0	1	1	1	0	0	1
1	0	0	0	1	1	1	0	0	1
1	1	0	1	1	0	0	0	1	0

adder, which sums two bits, $a + b$, to produce a 2-bit number whose most-significant, or 2s bit, is the *carry* and whose least-significant, or 1s bit, is the *result*. As shown in the table, *carry* = $a \wedge b$ and *result* = $a \oplus b$. A *full adder* sums a, b, and a carry bit c to produce a 2-bit sum with carry and result bits. What logic expressions on these three variables produce the carry and result?

carry:

result: Here are three equivalent expressions that could be wired up as circuits: , and

An expression is in *disjunctive normal form* (DNF) if it can be written as an 'or' of 'and' clauses of basic variables or their negations. The final formulae for carry and result above are DNF. An expression is in *conjunctive normal form* (CNF) if it can be written as an 'and' of 'or's. The Latin root *dis*, meaning apart, puts 'or' at the top of a disjunction; *con*, meaning together/with, puts 'and' at the top of a conjunction. The result from the full adder, also known as *parity*, requires a much larger formula in DNF or CNF than using 'xor' ($\oplus$).

Any formula can be written in DNF. One easy way is for each True in the truth table column, write an 'and' for all variables or their negations to select the True for that row. Can you see how to use de Morgan's laws to make a CNF formula by starting with the Falses in the column?

To determine the smallest formula is a difficult problem, but a tool that can help when we have just a few variables is the *Karnaugh map*, which gives a different view of a truth table. Here is an example for four variables: we make a 4×4 array in which rows correspond to the 4 possibilities for inputs a, b and columns correspond to the 4 possibilities for inputs c, d. These input possibilities are listed in the order 00, 01, 11, 10, called Gray code order, so that moving to an adjacent cell in the array changes only one variable. In each cell in the array we write the desired output 0 or 1, or X if for some reason we don't care. Rectangles (including squares) with side lengths 1, 2, and 4 in this array, including those that wrap around the boundaries, have simple 'and' expressions. We aim to cover all the 1s and none of the 0s with a small number of possibly overlapping rectangles, then 'or' these 'and' expressions to form a DNF formula. For the 4-variable example in the margin, a minimum DNF

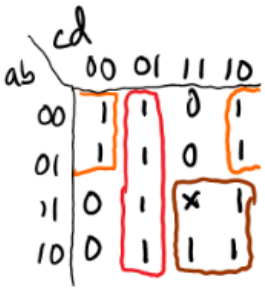


Figure 2.1: Karnaugh map

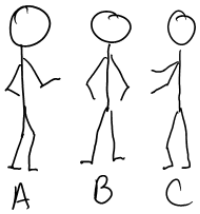
formula is $(\neg a \wedge \neg d) \vee (\neg c \wedge d) \vee (a \wedge c)$

2.6 Solving a puzzle with logic notation

Let's look at three different methods to use logic notation introduced in this chapter to solve the knight/knave puzzle of section 2.3. Notation helps us in four ways: it forces us to resolve ambiguities of the problem statement's English, it serves as a memory aid as we scribble ideas on scratch paper, it reveals patterns that help us ensure that all cases are covered, and it structures the presentation of our findings so they can be understood by others.

I must point out that the nice, neat presentation orders of my proposed solutions are different from the chaotic orders on my scratch paper. It is common to take detours and explore blind alleys; these are left out when I write down the steps of a short, clear path to a solution. This section will do you the most good if you first work through this question on your own scratch paper:

Relevant problem-solving steps: reviewing and documenting your solution.



A1. Recall that knights always tell the truth and knaves always lie. When you meet *Alice*, *Bob*, and *Chris*,

Alice says: Bob is a knave or Chris is a knight.
Bob says: Alice is a knight if, and only if, Chris is a knave.

Can you determine uniquely what each of Alice, Bob, and Chris are?

2.6.1 Defining and using notation

We would like to know if Alice is a knight or a knave. Let's take a moment to define notation so we can abbreviate both the reality and the claims about who is what.

Let A stand for the statement that 'Alice is a knight.' This is a proposition, because it is either true or false. Later we hope to find out which, but for now just think of A as an abbreviation. Because knights always tell the truth, we can also interpret this as A = 'what Alice says is true.'

*Check the negation to help clarify a proposition.

The negation $\overline{A}$ stands for 'it is not true that Alice is a knight.'* Since we are told that every person is a knight or a knave, we could instead say: $\overline{A}$ = 'Alice is a knave,' or equivalently $\overline{A}$ = 'what Alice says is false.'

Try before peeking.

Similarly define B = 'Bob is a knight,' and C = 'Chris is a knight.' Alice's statement can now be written in concise and precise notation: ____. Writing notation forces you to interpret 'or' as inclusive ($\vee$) or exclusive ($\oplus$), resolving an ambiguity of using English. The 'or' in Alice's statement would be considered inclusive—we allow the possibility that both halves of her statement are true.

We don't yet know whether Alice' statement is true or false, because we don't know if Alice is a knight or a knave.[†] What would be true if Alice's statement is false? We can negate to get [ ], and optionally apply de Morgan's law to get the equivalent [ ].

[†] "Alice is a knight *or* knave" is *exclusive or*, since the puzzle rules won't allow Alice to be both.

We can construct, from Alice's statement and its negation, a new statement that is true whether Alice lies or not: "If Alice tells the truth then (Bob is a knave or Chris is a knight) and if Alice lies then it is false that (Bob is a knave or Chris is a knight)." This becomes a little shorter, but remains cumbersome when we translate directly into notation. (Try before peeking.)

[ ]

Notice that this statement is a conditional and its inverse, which makes it equivalent to a concise 'if and only if' statement: [ ]. You should be able to read your statement in notation as, "Alice speaks the truth iff (Bob is a knave or Chris is a knight)."

Now, construct a similar statement that must be true from what Bob said. Any of [ ], [ ], or [ ] work here, since all can be read as, "Bob speaks the truth iff Alice and Chris are opposites."

2.6.2 Solution via truth table

We can use a truth table to explore all possible assignments of knight and knave to Alice, Bob, and Chris, or equivalently, all T/F assignments to A, B, and C. With 3 variables, our table will have $2^3 = 8$ lines.[*] On your scratch paper, write the column headers A, B, and C. Then fill the C column, alternating T/F; the B column, alternating TT/FF; and the A with TTTT/FFFF. (Or start with falses if you prefer; but use patterns to make sure you cover all possible assignments.)

[*] Corresponding to the numbers 0–7 in binary (see sections 1.2 or 7.1.4)

Now, add two more columns with the propositions from Alice and Bob—not the statements they say, which may be true or false, but the statements like $A \leftrightarrow$ (what Alice said) that are true whether Alice is a knight or a knave. Fill in the rest of the truth table. Go ahead and use the shorthand of writing truth values for each subexpression in small letters under the corresponding operator.

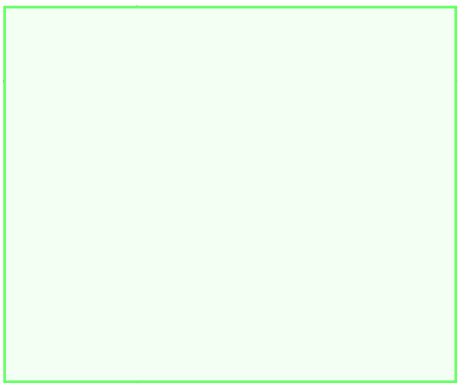
To see what I mean, drag the box covering table 2.7 down to expose the first row.

When you are done, compare your table to mine in table 2.7. You should find a unique row in the table for which the two statements are both true. This row tells us who are knights[?], and who are knaves[?].

2.6.3 Solution by inference from properties

We can use logic properties and logical inference to solve the same puzzle by starting with statements that we know to be true, then combining and simplifying to derive other true statements.

Table 2.7: A truth table for solving the 3 islanders puzzle



Here, start with the two 'iff' statements that we learn from what is said by Alice ($A \leftrightarrow$ "Bob is a knave or Chris is a knight") and Bob ($B \leftrightarrow$ "Alice is a knight iff Chris is a knave"). The rules of inference didn't say anything about biconditionals[†] convert each statement to an equivalent 'and' of 'or' clauses, known as *conjunctive normal form* (CNF). For the statement to be true, the rule of Simplification (table 2.5) implies that each 'or' clause must be true. We use other rules to further simplify 'and's of 'or' clauses.

[†] If they did, then Bob's $(B \leftrightarrow \overline{C}) \leftrightarrow A$, combines with Alice's to give a simpler start: $(B \leftrightarrow \overline{C}) \leftrightarrow (\overline{B} \vee C)$

First, convert 'D iff E' to CNF, and use that pattern to convert the statements we learned from Alice and Bob. We get a total of seven 'or' clauses that must each be true.

$$\begin{aligned}
D \leftrightarrow E &\equiv (D \rightarrow E) \wedge (\overline{D} \rightarrow \overline{E}) && \text{iff to if and inverse}\\
&\equiv (\overline{D} \vee E) \wedge (D \vee \overline{E}). && \text{if to or+negation}\\
A \leftrightarrow (\overline{B} \vee C) &\equiv (\overline{A} \vee \overline{B} \vee C) \wedge (A \vee \overline{(\overline{B} \vee C)}) && \text{iff to CNF}\\
&\equiv (\overline{A} \vee \overline{B} \vee C) \wedge (A \vee B) \wedge (A \vee \overline{C}). && \text{de Morgan and distrib.}\\
B \leftrightarrow (A \leftrightarrow \overline{C}) &\equiv (\overline{B} \vee (A \leftrightarrow \overline{C})) \wedge (B \vee (A \leftrightarrow C) && \text{outer iff to CNF}\\
&\equiv (\overline{B} \vee \overline{A} \vee \overline{C}) \wedge (\overline{B} \vee A \vee C) && \text{inner iffs to CNF}\\
&\quad \wedge (B \vee \overline{A} \vee C) \wedge (B \vee A \vee \overline{C}) && \text{and distrib.}
\end{aligned}$$

Now, we can bring together the first clause from Alice and third clause from Bob, distribute over 'and,' and simplify away a contradiction to get a new statement that must be true:

$$(\overline{A} \vee \overline{B} \vee C) \wedge (B \vee \overline{A} \vee C) \equiv (\overline{A} \vee C) \vee (\overline{B} \wedge B) \equiv (\overline{A} \vee C) \vee F \equiv (\overline{A} \vee C).$$

Combined with Alice's third clause, $(\overline{A} \vee C) \wedge (A \vee \overline{C})$ is equivalent to $A \leftrightarrow C$, so we know that Alice and Chris are the same, no matter what Bob is. That

means that $\overline{B}$: Bob lies. But then A: Alice speaks truth, and C: Chris does also.

The formal justification for the last two sentences can be given by rules of logical inference. Here is a two-column proof (more in section 9.2) in which each entry on the left is true for the reason on the right. You can ignore two-column proof structure for now, but when reasoning gets complicated, refining an argument to this level of detail can avoid errors or gaps.

1. $A \leftrightarrow C$	Given from above
2. $A \leftrightarrow (\overline{B} \vee C)$	From Alice's statement
3. $B \leftrightarrow \overline{A \leftrightarrow C}$	From Bob's statement
4. $(B \rightarrow \overline{A \leftrightarrow C}) \wedge (\overline{A \leftrightarrow C} \rightarrow B)$	Defn of first $\leftrightarrow$ in 3
5. $B \rightarrow \overline{A \leftrightarrow C}$	Simplif. rule on 4
6. $\overline{B}$	Modus tolens of 5 with 1
7. $\overline{B} \vee C$	Addition to 6
8. $(\overline{B} \vee C) \rightarrow A$	Defn $\leftrightarrow$ in 2, simplif.
9. A	Modus ponens of 8 with 7
10. C	Simpl. 1 & modus ponens w/ 9.
11. $\Omega\mathfrak{Q}$	

On my scratch paper, I had also combined $(\overline{A} \vee \overline{B} \vee C) \wedge (\overline{A} \vee \overline{B} \vee \overline{C}) \equiv (\overline{A} \vee \overline{B})$, which is true, but unhelpful since I had no direct statement about Alice and Bob.

There is a general form of the rule we have been applying, known as the *resolution rule*. If you have clauses $a_1 \vee a_2 \vee \ldots a_k \vee C$ and $b_1 \vee b_2 \vee \ldots b_j \vee \overline{C}$ both true, then $a_1 \vee a_2 \vee \ldots a_k \vee b_1 \vee b_2 \vee \ldots b_j$ must also be true. That is, $(A \vee C) \wedge (B \vee \overline{C}) \rightarrow (A \vee B)$ is a tautology. If you ever find both C and $\overline{C}$, then this contradiction would imply that there is no way to choose values to make all the clauses true. (Note that the resolution rule would give an empty clause.) Reasoning-based AI has long used resolution this way: if you want to prove that knowing a, b, and c you can prove d, then put $a \wedge b \wedge c \wedge \overline{d}$ into CNF and try to derive the empty clause. If you succeed, then you know $(a \wedge b \wedge c)$ implies d. Resolution is sound and complete for propositional logic, meaning that it never proves something false, and can be used to prove any true theorem.

2.6.4 Solution by cases and decision tree

In computer science, we often use a combination of logic properties with analysis of cases for selected variables—a combination of the two previous methods. It is important to use some structure or diagram to ensure that all cases are covered. For example, we might draw a decision tree that splits into cases depending on who lied. At each node of the tree we choose a person, and consider the cases where the person lied or told the truth by plugging T or F in for their variable in the two statements that we know must be true: $A \leftrightarrow (\overline{B} \vee C)$ and $B \leftrightarrow (A \oplus C)$.

If Alice lied, then we would know $F \leftrightarrow (\overline{B} \vee C) \equiv B \wedge \overline{C}$; that is, Bob is a knight and Chris is a knave. But since Alice and Chris are both knaves,

Bob's claim is false, contradicting that Bob is a knight. Thus, Alice must be telling the truth.

Plugging in T for A simplifies $T \leftrightarrow (\overline{B} \vee C) \equiv \overline{B} \vee C$. We consider two more cases, based on B.

Figure 2.2: A decision tree: At the first stage we plug in False or True for A, then simplify the expressions for what A and B said based on this assumption we want only on branch of the tree to be free of contradiction.

Suppose that B is true. Then $T \leftrightarrow (\overline{F} \vee C) \equiv C$, so Chris is a knight, along with Alice and Bob. But again, Bob's claim that Alice and Chris are opposite gives a contradiction. So B must be false. (This makes Alice's statement true, no matter whether Chris is knight or knave.)

Now, Bob is a lying knave, so we know Chris is the same as Alice, who is a knight. This makes both statements true. Furthermore, we have ruled out all other possibilities, so this is the unique solution.

This may seem the most straightforward of the three solutions in this section, because it is a direct attack that requires the least amount of new notation or concepts. It helps that I presented it in an order that closed blind alleys quickly. Notice, however, that it actually requires the most thought because it is not mechanical, like a truth table, and it speaks mainly about hypotheticals that are not true, as opposed to the properties and inferences in which every statement written down was true. Look back over the three solutions and explain to yourself where thought was required and where you (or a computer) could do calculations or manipulate symbols without thinking other than to recall the rules.

We will use all three methods of solution, often in combinations.

2.7 Summary

The definition and use of logical notation ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ and $\oplus$) forces us to make unambiguous, precise statements about how we intend to combine logic variables. Even before we know the specific values of the variables, we can simplify statements with logical equivalences, or use the rules of logical inference to preserve truth. Then we can write down a truth table listing all possible T/F assignments to variables and the outcomes—propositional logic becomes purely mechanical.

It is important to learn the language of this formal notation, because we will use it to write the definitions and to reason about the properties of all of the discrete structures in this book.

2.8 Exercises and Explorations

Quiz Prep 2.1. Be able to fill in truth tables like table 2.1 and table 2.3. Be able to demonstrate logical equivalence for the rules of inference in table 2.5. E.g., fill in this truth table demonstrating the logical implication (and not the logical equivalence) of Hypothetical Syllogism. (Here p, q, & r are logical variables and A, B, & C are abbreviations for implications.)

			A	B	C		
p	q	r	$p \to q$	$q \to r$	$p \to r$	$(A \wedge B) \to C$	$(A \wedge B) \leftrightarrow C$

Quiz Prep 2.2. Match each of the following statements, with logic variables p and q, to their logical equivalent(s), which use only the "and" ($\wedge$) and negation (overline) operations.

1. $p \to q$
2. $\overline{p \vee q}$
3. $p \leftrightarrow q$
4. $p \oplus \overline{q}$

a. $\overline{p \wedge \overline{q}} \wedge \overline{\overline{p} \wedge q}$
b. $p \wedge \overline{q}$
c. $\overline{p \wedge \overline{q}}$
d. $\overline{\overline{p} \wedge q}$
e. $\overline{p} \wedge \overline{q}$

Quiz Prep 2.3. Write logical expressions for:

1. At least one of p, q, and r is true.
2. At most one of p, q, r is true. , or
3. Exactly one of p, q, r is true. , or

Quiz Prep 2.4. Use a truth table to prove the *exportation rule*: that $(p \wedge q) \rightarrow r$ is logically equivalent to $p \rightarrow (q \rightarrow r)$.

Exercise 2.5. Operator precedence: In arithmetic expressions we have precedence rules that says that in an expression like $1 \cdot 5 - 8/4 + 2^3$ evaluate the exponential, then multiplication and division (left to right), and finally addition and subtraction (left to right). (Some learn these as PEMDAS.) In logic, the order is parentheses, negation ($\neg$), and ($\wedge$), or ($\vee$), and if,iff ($\rightarrow, \leftrightarrow$), with operations evaluated from *right to left*. Insert parentheses in these expressions so they they will evaluate correctly even if you follow only the parentheses rule.

1. $p \vee q \wedge r$ is _____.
2. $p \rightarrow \neg q \rightarrow r$ is _____.
3. $p \leftrightarrow q \rightarrow r$ is _____.
4. $p \vee \neg q \rightarrow r$ is _____.
5. $p \wedge q \rightarrow p \vee q$ is _____.
6. $p \vee \neg q \leftrightarrow \neg(\neg p \wedge q)$ is _____.

▶

*Warning: incorrect statements in this problem!

Exercise 2.6. Find the mistake(s) in each of the following. *

1. The negation of $0 < x < 5$ is $0 \geq x \geq 5$.
2. p only if q means $q \rightarrow p$.

▶

Exercise 2.7. Use truth tables to establish the properties of section 2.4. These are among the most useful:

1. Commutative: $p \vee q \equiv q \vee p$, $p \leftrightarrow q \equiv q \leftrightarrow p$, but $p \rightarrow q \not\equiv q \rightarrow p$.
2. Associative: $(p \vee q) \vee r \equiv p \vee (q \vee r)$, $(p \leftrightarrow q) \leftrightarrow r \equiv p \leftrightarrow (q \leftrightarrow r)$, but $(p \rightarrow q) \rightarrow r \not\equiv p \rightarrow (q \rightarrow r)$
3. Distributive: $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$ and $p \rightarrow (q \wedge r) \equiv (p \rightarrow q) \wedge (p \rightarrow r)$.

a
b
$\therefore c$

Exercise 2.8. Use truth tables to establish the nine rules of inference in table 2.5. To establish an inference rule that follows the pattern in the margin, you would demonstrate that $(a \wedge b) \rightarrow c$ is a tautology.

Exercise 2.9. Show that the 'nand' operator, which is *false* if and only if both inputs are *true*, can be used to make NOT, AND, OR, IF, and IFF. One symbol for 'nand' is the 'Sheffer stroke,' which is just an upward arrow, ↑.

Hint:

Puzzle 2.10. (From "Test your logic: 50 puzzles in inductive reasoning" [24]) When Adrian, Buford and Carter eat out, each orders either ham or pork.

1. If Adrian orders ham, Buford orders pork.
2. Either Adrian or Carter orders ham, but not both.
3. Buford and Carter do not both order pork.

Who could have ordered ham yesterday and pork today? Hint:

Puzzle 2.11. On an island often visited by Raymond Smullyan, every inhabitant is a "knight" who always tells the truth, or a "knave" who always lies. You meet five inhabitants, A–E, and four of them make statements:

A says: E is a knight if and only if C is a knight.
B says: If D is a knave, then A is a knight.*
C says: A is a knave if and only if B is a knight.
D says: If A is a knave, then E is a knight.

*Hint: What we learn from B is not $(\overline{D} \to A)$, but , which captures that B may lie.

Write down in logic notation what we learn from each statement, using logic variable A to stand for "A is a knight." Thus, if A says statement P, we would learn $A \leftrightarrow (P)$. Then, determine who is a knight and who is a knave so that what we learn from each speaker is true. As in section 2.6, you can use a truth table, rules of inference, or a mix of both. ?

Puzzle 2.12. On Nomanisan Island, the natives communicate only person to person. Each inhabitant belong to one of two tribes: T or F. When two people from the same tribe communicate, they tell the truth, but when two people from different tribes communicate, they lie.

You meet 5 inhabitants, denoted A–E, and you ask them to talk about who is from what tribe. You observe these statements:

A to B: C is from tribe T iff E is from tribe T.
B to C: A is from tribe F or E is from T.
C to D: D is from tribe F iff E is from tribe F.
D to E: E is from tribe F or B is from F.
E to A: B is from tribe T or D is from T.

Write in notation what you learn from each statement. For example, from A to B: P, you learn $(A \leftrightarrow B) \leftrightarrow P$. Then determine who is from tribe T and who is from F.

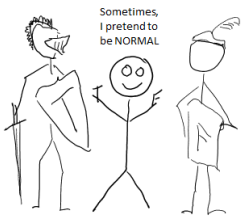
Puzzle 2.13. On Raymond Island, "knights" always tell the truth, "knaves" always lie, and "normals" sometimes tell the truth and sometimes lie.

1. You meet three islanders, *A*, *B*, and *C*, who say

   A: I am a knight.
   B: I am a normal.
   C: I am a knave.

   For which of the three can you uniquely determine the type? (For each one, say what type they could be and whether they are lying.)

2. Suppose you are a normal. What could you truthfully say that would clearly indicate your status?



Puzzle 2.14. On Raymond Island, "knights" always tell the truth, "knaves" always lie, and "normals" sometimes tell the truth and sometimes lie. You meet 4 natives, denoted *A*–*D*. At least one is a knight, and at least one is a knave. They state:

A: If *B* is a knave, then *C* is a knight.
B: *D* is a knight if and only if *C* is a knight.
C: If *B* is a knave, then *A* is a knave.
D: *C* is a knave if and only if *A* is a knave.

Define some notation that will let you write what you learn from each statement. Unlike the previous problems, you cannot equate true=knight and false=knave, because there is a third option. Also, you learn something if the speaker is a knight or knave, but nothing if the speaker is a normal. Who are knights, who are knaves, and who are normals?

Puzzle 2.15. (From "Test your logic: 50 puzzles in inductive reasoning" [24]) Aaron Green and his sisters, Betty and Clara, had a tragic dinner with Flora Brown and her brothers, Duane and Edwin. Their two separate families had pushed them into the same two professions: Aaron, Betty, and Duane were bankers; Clara, Edwin, and Flora were lawyers. At the dinner, one of the six was killed by one of the other five. Who was the killer?

1. If the killer and victim were related, the killer was a man.
2. If the killer and victim were not related, the killer was a banker.
3. If the killer and victim had the same occupation, the victim was a man.
4. If the killer and victim had different occupations, the victim was a woman.

5. If the killer and victim were the same gender, the killer was a lawyer.
6. If the killer and victim were different genders, the victim was a banker.

Discuss what types of notation you create or use to help you solve this without getting lost in the possibilities.

Exploration 2.16. Use your favorite spreadsheet program to create truth tables for logic operations or to solve logic puzzles. You can use TRUE/FALSE and logic operations (check your documentation for AND, NOT, OR, IF), or 0/1 and arithmetic operations (*, –, max, mod). First you'll want to set up a way to count through all possible T/F sequences for the variables, and then put in the logic expressions using those variables. Spreadsheets let you "fill down"* to replicate the formulas in a row to fill the entire table.

*The first spreadsheet program, Visicalc in 1979, already had replication commands.

Next, put the text of the formulas in row 1, and the spreadsheet math in row 2. Spreadsheets support logic (AND, OR, NOT) on TRUE/FALSE, or you can use arithmetic operations on numbers to achieve the same operations (*, MAX, 1–). The ampersand (&) concatenates strings in Excel, and is not the operator for 'AND.' Check the documentation on IF; in Excel, $A \to B$ is written as =IF(A,B,TRUE), or you can use the equivalent =OR(NOT(A),B). I use =(A=B) for the biconditional, $A \leftrightarrow B$.

Chapter 3
Sets, Tuples, & Counting

> Mathematics may be defined as the economy of counting. There is no problem in the whole of mathematics which cannot be solved by direct counting.
> —Ernst Mach

In this chapter we begin to define sets, which are unordered collections of elements with no repeats, and tuples, which are ordered lists of elements allowing repeats. Strings and arrays are generalizations of tuples. We also define the most basic operations on sets and tuples; chapter 5 defines additional operations and chapter 8 adds recursive definitions. To count the elements of a set, especially sets of tuples or permutations, we will apply the sum and product rules; we return to these rules more formally in section 6.3.

Counting can help check if we understand a definition. The last digit of a bank account or credit card number is typically a "check digit" that can be computed from the others—e.g. it might be the last digit of the sum of the other digits. It adds no new information to the account number, but can catch many errors, including any single digit mistyped. In a similar way, counting often gives a quick confirmation that we have correctly understood a new concept, structure, or algorithm.

Objectives: After studying this chapter, you will be able to define sets with braces, { }, tuples with parentheses, ( ), sets of tuples with Cartesian product, $\times$, and sets of strings with concatenation, and series with summation. You will be able to name common sets (including the natural numbers $\mathbb{N}$, integers $\mathbb{Z}$, and the empty set $\emptyset = \{\}$), and use notation for testing element inclusion in a set, $\in$ and its negation $\notin$, and for testing whether a set is a subset of another, $\subseteq$. You will be able to count elements in sets using the sum and product rules, and count permutations and combinations, using notation for summation ($\sum$), product ($\prod$), factorial (!), and choose (binomial coefficients $\binom{n}{k} = C(n,k)$).

	k-element set or bag	k-tuple
without repetition	$\binom{n}{k} = C(n,k) = \frac{n!}{(n-k)!\cdot k!}$	$P(n,k) = n^{\underline{k}} = \frac{n!}{(n-k)!}$
allowing repetition	$\binom{n-1+k}{k}$	n^k

3.1 Defining sets and tuples

Definitions are important to remember: how often have you been in an argument and realized that both parties were using the same words, but with different meanings? Formal definitions help ensure that, in mathematics and software development at least, we are talking about the same thing.

Good definitions are concise, but precise—even though the concepts they define may be abstract. When you encounter a definition, even if the concept is familiar, create small examples that fit and that do not fit the definition. Sometimes mathematical definitions don't seem to match our intuitive ideas: is a set that contains no elements still a set? We say 'yes' because that causes fewer special cases than saying 'no'—for example, the intersection of two sets is always a set. But it is always a good idea to check these 'boundary cases.'

In programming, a set (or tuple or list) is often represented by a *data structure*s that can change by adding or removing elements. In mathematics, it is better to think of sets as imutable ideal objectss – If I add elements to a set, I am creating a new set; it costs no resources to think of both the old and new sets as still existing. Two different data structures that contain the same elements are thought to represent the same mathematical set, even though it may take some work for the program to verify that fact.

3.1.1 Sets

A *set* is an unordered collection of distinct *elements* from some *universal set*, U. By common convention, we name sets with upper case letters and elements with lower case, and define notation $a \in S$ to be true if a is an element of S and false otherwise. (Sometimes we allow sets to be elements of other sets; we don't automatically unpack sets, as we'll illustrate below.)

We can specify the elements in a set using curly *braces* in three different ways.

$\emptyset$ is also the set of really funny jokes in these margins.

list: For a finite set, we can list all the elements surrounded by braces; three examples are $S = \{a, b\}$, the *empty set* $\emptyset = \{\}$, and $T = \{\emptyset, a, S\}$.

Neither the order of the elements nor repeated elements have any affect on which elements are in the set, so $\{b, a, a\}$ is the same set as S. As the set T shows, however, sets may become elements of other sets. Here, T has three elements, not two or four, as explained at the end of this section.

*Zero is natural to a computer scientist. Sec 8.1.1 defines $\mathbb{N}$ recursively.

†$\mathbb{Z}$ comes from *Zahlen*, the German word for numbers.

pattern: Informally, we can indicate a set by writing down some elements and using ellipses. Three examples are the alphabet $A = \{a, b, c, \ldots, z\}$, the set of *natural numbers** $\mathbb{N} = \{0, 1, 2, \ldots\}$, and the set of *integers*† $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, 3, \ldots\}$. Be sure that you write enough of the pattern that there is no ambiguity.

rule: Within braces, we can write a rule consisting of a function, a vertical bar, and a set to which the function is applied. Three examples are the state capitals, $C = \{x \mid \text{some state } S \text{ has capital city } x\}$, the squares $\{x^2 \mid x \in \mathbb{Z}\}$, and the *rational numbers* $\mathbb{Q} = \{p/q \mid p, q \in \mathbb{Z} \text{ and } q \neq 0\}$. Because sets have no duplicates, the value $1/2$ is considered to appear in $\mathbb{Q}$ once, even though it is added in many ways, e.g., $1/2 = 3/6 = 128/256$.

subsection 8.1.1 introduces recursive definition to formally define sets of arbitrary sizes, without using ellipses (. . .).

Several common sets have conventional names or symbols: $\emptyset$, $\mathbb{N}$, $\mathbb{Q}$, and $\mathbb{Z}$ are defined above. The reals $\mathbb{R}$ are used in calculus.

For integers and reals we have *ranges*, such as the positives $\mathbb{Z}^+$ and $\mathbb{R}^+$, negatives $\mathbb{Z}^-$ and $\mathbb{R}^-$, non-negatives $\mathbb{N}$ and $\mathbb{R}^{\geq 0}$, and the closed,* open, and half-open intervals: in this book, $[m..n] = \{m, m+1, \ldots, n-1, n\}$ is always a set of consecutive integers, and $[a, b] = \{x \mid a \leq x \leq b\}$, $(a, b) = \{x \mid a < x < b\}$, and $[a, b) = \{x \mid a \leq x < b\}$ are usually reals, although they will denote sets of integers in some contexts, like algorithm loop variables.

*Mnemonic: *c*losed *c*ontains the ends, and *o*pen *o*mits the ends

Please note that $a < x < b$ is a mathematical abbreviation for the logical statement $(a < x) \wedge (x < b)$. A few programming languages, including Python and Julia, support "inequality chaining" and therefore handle this abbreviation correctly. Most programming languages will warn you if you try to use an expression like $1 < x < 5$, because they recognize that the first inequality gives a Boolean (true/false) result, which the second tries to compare to the number 5. In languages that represent logic values as numbers (including C and FORTRAN, which represent true as 1 and false as 0) a compiler may not even warn you that the expression $1 < x < 5$ is always true, regardless of the value of x. As discussed at the end of section 2.1, we can use de Morgan's law to determine the set of values x that are not in (a, b), namely $\{x \mid (a \geq x) \vee (x \geq b)\}$.

The basic operation for a set S is *element inclusion*: testing whether a given element a from the universe U is in S. As mentioned above, the expression $a \in S$ is true if a is an element of S and false otherwise. For example, $2 \in \mathbb{Z}$, but $\overline{1/2 \in \mathbb{Z}}$, which we can write with less clutter as $1/2 \notin \mathbb{Z}$. Likewise, $x \notin (a, b)$ means either $x \leq a$ or $b \leq x$.

In chapter 5 we use element inclusion, $\in$, and logic to define other set operations (equality, subset, union, intersection). For example, set A is a *subset* of B, denoted $A \subseteq B$, iff every element in A is also in B, and $A = B$ iff every element of the universe is either in both A and B, or in neither A nor B. One consequence of defining all operations based on inclusion is that we have no way to determine the order elements are placed into a set, or

whether an element is "in a set more than once."[†] We thus say that *sets have no order and do not keep track of repeats*; an element is either in a set S or it is not.

[†] The *multi-set* or *bag* data structure tracks repeats, but not order. Tuples, defined next, track both.

The number of elements in a set S is called the *size* or *cardinality* of S, and is denoted $|S|$. (Read $|S| = n$ as, "The cardinality of S is n," or, "Set S has n distinct elements.") To count elements of a set, we simply assign them numbers 1, 2, 3, ... until every element has exactly one number. After subsection 6.2.2 we can say this more mathematically: $|S| = n$ if and only if there is a bijection (a one-to-one and onto function) $f : [1..n] \to S$. In most of this book we count *finite* sets only, but we'll see in section 6.3 that the bijection definition also lets us count and compare infinite sets.

The elements of sets may be sets themselves, as we saw in the last example $T = \{a, \emptyset, S\}$, where $S = \{a, b\}$. Set T contains three elements, two of which are sets: $\emptyset \in T$ and $S \in T$ are both true. (Since S is a set it is uppercase; by position in front of $\in$ it also being considered as an element.) Note that T does *not* contain b as an element: $b \notin T$, even though $\{a, b\} \in T$. Likewise, $\emptyset \notin S$.

Counting helps us distinguish between the empty set, denoted $\emptyset$ or $\{\}$, which has zero elements, and the set containing the empty set, denoted $\{\emptyset\}$ or $\{\{\}\}$, which has one element that happens to be a set itself.

Since a 'set of sets' is hard to say, we will instead say *collection of sets* or *family of sets*. For example, $\{\{a, b\}, \{b, c\}, \{a, c\}\}$ is the family of the three two-element sets that can be made from the elements of $\{a, b, c\}$. section 5.3 has other examples.

3.1.2 Tuples

As just defined, sets are unordered collections of elements. In many instances, however, order matters; the ordered pairs (x, y) of Cartesian coordinates are the most familiar mathematical example.

[*] 'Tuple,' in American English, is pronounced with a long 'u'.

A *k-tuple*[*] is an ordered sequence of k elements, which we write down in parentheses, $(a_1, a_2, \ldots, a_k)$. Two tuples are equal iff all of their corresponding elements are equal: $(a_1, a_2, \ldots, a_k) = (b_1, b_2, \ldots, b_k)$ iff for all $i \in [1..k]$ we have $a_i = b_i$.

2-tuples and 3-tuples are more commonly called *ordered pairs* and *ordered triples*. These names emphasize the two important characteristics of a k-tuple: the number of elements is fixed at k, and the order matters: (a, a, b) and (a, b, a) are different 3-tuples. There is also one 0-tuple, $()$, which may be called the *empty tuple*.

The basic operation to create tuples is the *Cartesian product of two sets*, which actually makes a set that contains all possible pairs:

$$A \times B = \{(a, b) \mid \text{ for all } a \in A \text{ and for all } b \in B\}.$$

Let $L = \{a, b, c\}$ and $D = \{0, 1\}$. Can you list the tuples in these sets?

In pdf, green boxes hide answers or hints. Once you have your own answers, select them and move or delete. They shouldn't appear in printouts, unless you choose to print with annotations.

$L \times D =$ [ ]

$D \times L =$ [ ]

$L \times L = L^2 =$ [ ].

$D \times L \times D =$ [ ]

[ ]

$L \times \emptyset =$ [ ]

$L \times \{( )\} =$ [ ]

Notice three things about the notation used here: First, we abbreviate $L \times L$ as L^2. In general, whenever you see a set S^k, it means take the Cartesian product of k copies of S to make the set of all possible k-tuples. Since order matters and repeats are allowed in a tuple, $L^2 = L \times L$ contains nine pairs. In contrast, the number of two-element sets from L is only three.

Second, we interpret $(0, (a, 1)) = ((0, a), 1) = (0, a, 1)$; unlike sets, tuples just become longer, rather than nesting inside each other. In general, if we join a j-tuple to a k-tuple, then we obtain a $(j + k)$-tuple and can suppress all parentheses but the outermost. This holds also for the 0-tuple; the pair $((a), ( ))$ reduces to the 1-tuple (a). Notice that this means that $D \times (L \times D) = (D \times L) \times D$; Cartesian product is *associative*; it does not matter which product we do first. (It is generally not commutative; note above that $D \times L \neq L \times D$.)

Finally, note the difference between the empty set $\emptyset$ and the set containing the empty tuple. The empty set has no elements, so the Cartesian product definition for $L \times \emptyset$ has no way to choose a $b \in \emptyset$, so the result is empty. The set containing the empty tuple has one element, and since this element does not affect tuples it is paired with, $L \times \{( )\} = L$.

We have already seen tuples applied: the rows of a truth table with k variables are the elements of $\{T, F\}^k$, which is the set of all possible k-tuples of T and F.

$\{T,F\}^3 = \{TTT, FFF, TTF, FFT, TFT, FTF, TFF, FTT\}$

Not all elements of a tuple need to have the same type. This makes tuples perfect for associating different types of data. A Kara program, for example, may be described as a 5-tuple (S, s_0, f, R, δ) containing a set of states S, a distinguished start state $s_0 \in S$, a final or stop state $f \in S$, a set of sensors R, and a transition function δ that maps the current state and the sensor readings to a new state and string of actions from the set $M = \{\curvearrowleft, \Uparrow, \curvearrowright, \underline{\uparrow}, \underline{\downarrow}\}$.

Relational databases gather data into tuples – e.g., a student record may be a tuple with a name, id number, address, and other associated data. The database of all students may be viewed as a set of these tuples. The

distinguishing feature of a set of tuples is that each tuple has the same finite number of elements and same types in each position.

3.1.3 Sequences, strings, and series

Several structures that are extensions of tuples will be defined formally in chapter 8. We use the terms *list* when we speak about tuples that may have arbitrary finite length, and *sequence* when that length may even be infinite. *Vectors* are tuples with additional operations, including addition (translation), scalar multiplication (scaling), rotation, and calculating lengths. *Matrices*, *arrays*, and *tables* are tuples, displayed with a two-dimensional layout, and perhaps given extra operations.

Strings are lists containing characters from a given *alphabet* set. We write them without parenthesis or braces, and often name them with lowercase Greek letters like α, β, γ, σ, or τ. The *empty string* is commonly denoted by capital Lambda, Λ. A set of strings is called a *language*. For example, using the alphabet $A = \{a, b, c\}$, we can define strings $\alpha = a$, $\beta = baa$, and $\gamma = cab$. We can define the language of all three-letter strings $A^3 = \{aaa, aab, aac, \ldots cca, ccb, ccc\}$, the language of all strings $A^* = \{\Lambda, a, b, aa, ab, bb, ba, aaa, aab, \ldots\}$, and the language D of all strings from A^* that are in the official Scrabble dictionary: $D = \{aa, aba, abaca, abba, ba, baa, baba, bacca, cab, caca\}$.*

Find with a regular expression search for $\wedge[abc]\$$.

A list, sequence, or string by itself is really just a tuple, sometimes given slightly different notation. The main difference comes when we make sets: a set of lists or strings may contain elements of many sizes, but a set of k-tuples contains only tuples of length k.

The main operation for lists and strings is *concatenation*, which joins two input lists or strings of length m and n into one of length $m + n$ by putting elements of the second list or string after those of the first. chapter 8 formally defines this and other operations.

We extend the definition of concatenation from strings (as lists) to languages L and M by concatenating all possible pairs of strings: $LM = \{\alpha\beta \mid \alpha \in L \text{ and } \beta \in M\}$, like Cartesian product except that if L or M has entries of different lengths then LM will have entries of different lengths. chapter 8 will formally define L^* to be all strings that can be made by concatenating a finite number of strings from L.

A *series* is a sum of a sequence of numbers, replacing commas by plus signs. Any finite sequence, such as $(x_1, x_2, \ldots, x_n)$, has a series $x_1 + x_2 + \cdots + x_n$ with a well-defined value. As you should recall from calculus, an infinite series, $x_1 + x_2 + x_3 + \cdots$, may converge to a limit, may diverge to infinity, or may be undefined. Do you recognize the behaviors of the following examples?

1. $x_i = i$: $\sum_{i \geq 1} i = 1 + 2 + 3 + \cdots$?

2. $x_i = 1/i$: $\sum_{i\geq 1} \frac{1}{i} = 1 + 1/2 + 1/3 + \cdots$?
3. $x_i = (-1)^i$: $\sum_{i\geq 1} (-1)^i = -1 + 1 - 1 + 1 - \cdots$?
4. $x_i = (1/2)^i$: $\sum_{i\geq 1} \frac{1}{2^i} = 1/2 + 1/4 + 1/8 + \cdots$?
5. $x_i = (-1/2)^i$: $\sum_{i\geq 1} \frac{1}{(-2)^i} = -1/2 + 1/4 - 1/8 + 1/16 - \cdots$?

In this book, nearly all of our series are finite; the main thing that I want to borrow from calculus is Euler's *Sigma summation notation* to write sums without ellipses. Beneath the sigma, $\sum$, name one or more index variables and a set that their values come from, and follow with an expression that depends upon the index variables. For example, the finite sum $\sum_{i\in[1..n]} x_i = x_1 + x_2 + \cdots + x_n$ uses index variable i and set $[1..n]$.

$\sqrt{-1}\ 2^3 \sum \pi$,
and it was delicious.
(The Greek capital sigma is read here as sum.)

Note that an index variable is just a placeholder; as long as you choose a variable that does not already have a value or otherwise appear in the expression, you can substitute freely: $\sum_{j\in[1..n]} x_j = \sum_{i\in[1..n]} x_i$.

We often omit parts of the notation that can be inferred from context. E.g., rather than writing a set of integers, we may just write the condition on the index variable:

$$\sum_{i\in\mathbb{Z}^+} x_i = \sum_{i\geq 1} x_i = x_1 + x_2 + x_3 + \cdots$$

Many write sums with upper and lower index limits, like $\sum_{i=1}^{n} x_i$, but I encourage you to sum over a set of indices instead. Specifying indices by a set or logical condition gives flexibility—for example, we can easily sum over evens or primes. We can also sum over pairs with a single summation. Compare two sums of products: We might sum $x_i \cdot y_j$ over all pairs of indices $i,j \in [1..n]$, allowing repeats and where order matters, as $\sum_{(i,j)\in[1..n]^2} x_i \cdot y_j$. summing On the other hand, we might sum $x_i \cdot x_j$ only over only non-decreasing pairs with $1 \leq i \leq j \leq n$ so as not to multiply pairs twice, or over increasing pairs with $1 \leq i < j \leq n$ if want to omit the squares x_i^2. If we decide to turn this last sum into nested summations, it is easier to get the ranges correct if we keep them all below the $\sum$:

$$\sum_{1\leq i<j\leq n} x_i \cdot x_j = \sum_{1\leq i<n} \left(\sum_{i<j\leq n} x_i \cdot x_j \right) = \sum_{1<j\leq n} \left(\sum_{1\leq i<j} x_i \cdot x_j \right).$$

In a similar fashion, the product of elements in a sequence is represented by Greek capital Pi: $\prod_{1\leq i\leq n} x_i = x_1 \cdot x_2 \cdots x_n$. The next chapter will use big-and and big-or to apply the corresponding logic operation to a sequence of true/false values.

3.2 Counting elements in sets

We have already been counting the number of True values for logic operations to check that we have a shared undersanding, or to distinguish between a

set of sets and a set of tuples. In the rest of this chapter, I introduce some rules, concepts, and notation that we can use to count elements in a set. We revisit these more formally in later chapters.

The basic idea will be to impose set or tuple structures on what we want to count to make counting easier or more generalizable. Have you ever needed to check if a standard pack of playing cards was complete? You might know there are 52 cards in a pack, so you could count to 52 (disregarding any Jokers or premium cards.) That is tedious and possibly error prone, so you might deal packets of 10 and check that you dealt 5 packets with 2 left over. That is, you impose structure that lets you do five smaller counts, and $5 \cdot 10 + 2 = 52$.

You likely added $10 + 3$, but using the distributive property saved me the mental effort of multiplying by 13.

Why 52? That's even nicer, once you recognize that the deck contains all unique ways to pair a suit with a value: $\{\heartsuit, \spadesuit, \clubsuit, \diamondsuit\} \times \{A, 2..10, J, Q, K\}$. Since the choice of suit does not affect the number of choices for value, the number of these pairs is $4 \cdot (10 + 3) = 40 + 12 = 52$.

To count cards we imposed structures that let us add and multiply, letting us count larger numbers with less tedium. We can spell those out as two simple rules that help us count by breaking more complicated questions into simpler ones that we can combine for the answer. The rest of this chapter is built on application of these two simple rules to count sets, tuples, permutations, and multisets.

Is it really that important to know the numbers of all the various things I ask you to count? Should you be trying to memorize a bunch of formulas? Is there one right way to answer any given counting question? No, no, and no. The key is to use the language of sets and tuples (and bijections, after subsection 6.2.2) to practice translating an initial question into precise notation, manipulating the notation to identify smaller subproblems that have easy solutions, then correctly combining subproblem solutions into a final answer. When the answer is a count, checking agreement between different people is especially convenient, even if they broke the question down in different ways.

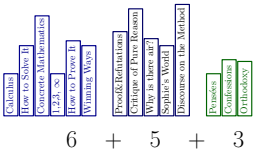
3.2.1 The sum and product rules

We will repeatedly use two simple rules for counting elements in certain sets: the sum rule and product rule. I introduce them informally here, and more formally in section 6.3.

Sum rule: Suppose that each element of a set has one assigned *type*. Then the total number of elements is the sum of the numbers of elements of each type.

Product rule: Suppose that elements of a set each have several features

that can be chosen independently. Then the number of elements is the product of the numbers of choices for each feature.



Here are simple examples of both rules. You have a set of six different math books, five different philosophy books, and three different religion books. How many books do you have? The sum rule says, "[redacted]"

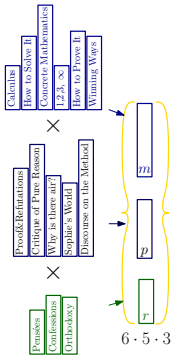
What are we counting? Books in a set, which means no repetition and order does not matter.

Why the sum rule? Because each book is one of the three types, math, philosophy, or religion, applying the sum rule counts each book in the set exactly once.

Now you want a set of three books, one of each type; how many sets of three are possible? You want to count all tuples of the form (m, p, r), where m is a math book, p is a philosophy book, and r is a religion book. You choose one of six math books, one of five philosophy books, and one of three religion books—the product rule says, "[redacted] sets of three are possible."

What are we counting? The possible sets of three books, one of each type. (We can instead think of this as counting triples (m, p, r), using the three types to impose an order; more on this in subsection 3.2.3.)

Why the product rule? Because we choose each type of book independently, we multiply the 6 math choices, 5 philosophy, and 3 religion to get the total number of different 3-tuples or 3-sets that have one book of each type.



If we name the sets of math, philosophy, and religion books M, P, and R, respectively, then the set of tuples is the Cartesian product $M \times P \times R$; the product rule says we count these tuples by multiplying the sizes of the sets: $|M \times P \times R| = |M| \cdot |P| \cdot |R|$. For another example, the reason a truth table with k variables has 2^k rows is because the possible assignments are $\{T, F\}^k$ and $|\{T, F\}^k| = |\{T, F\}|^k = 2^k$.

Note how the product rule is consistent with what we observed for concatenation with the empty set $\emptyset$ or with the set consisting of the empty tuple:

$$|L \times \emptyset| = |L| \cdot |\emptyset| = |L| \cdot 0 = 0 = |\emptyset|, \text{ and}$$
$$|L \times \{()\}| = |L| \cdot |\{()\}| = |L| \cdot 1 = |L|.$$

Consistency is one sign of a good definition; applicability is another. In the next two subsections, we'll apply the product rule to count *permutations*, which are all ways to put distinct items in order, and *combinations*, which are all ways to choose a set of distinct items. We introduce factorial and choose notation to count the numbers of permutation and combinations, respectively.

As you read, ask yourself, "What are we counting?" and, "How is the counting task broken into smaller pieces using the sum rule (for an either/or choice) or product rule (for simultaneous independent choices)?"

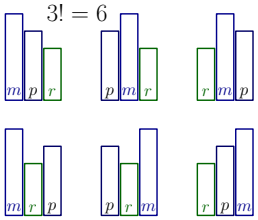
The skills of precise statement of problems and of breaking complex problems into simpler ones are essential in computer science. The set and tuple notation will help you as you become more familiar with it. There is always more than one way to state or solve a problem, so the goal is not for you to memorize formulas, but to understand the principles and practice with the notation. Remember the problem solving steps of section 1.2.

3.2.2 Permutations and factorial

The permutations of n distinct items are the set of all n-tuples that never repeat any item. This is the set of all ordered lists that contain each of the n items exactly once.

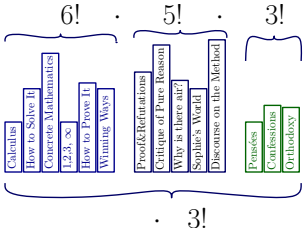
*$n!$ is read as 'n factorial,' or as 'N' very loudly.

The number of *permutations* (different orders, no repeats) of n items is n factorial*, written $n!$. You are probably already familiar with the fact that $n! = \prod_{1 \le i \le n} i = n \cdot (n-1) \cdots 2 \cdot 1$. This counts permutations by the product rule: for the first element we choose from n items, for the next element we choose from the $n-1$ remaining . . . , and for the last element only one item remains. Each choice produces a different permutation—knowing the permutation, you can work out the choices that produced it. The *number* of choices at each step does not depend on which choices were made before, so repeated application of the product rule says that the number of permutations of n items is $n!$. By convention, $0! = 1$; there is only one possible order for the 0 items in the empty tuple.



For example, if I have chosen math book m, philosophy book p, and religion book r, then I can choose one of $3! = 6$ orders to shelve them, {*mpr*, *mrp*, *pmr*, *prm*, *rmp*, *rpm*}, where I use string notation to avoid having to type parentheses and commas for these triples. The product rule, therefore, says that the number of ways to arrange three of the 14 books on the shelf and get one of each type is $6 \cdot 5 \cdot 3 \cdot (3!) = 540$, since choosing one of the 3! orders of book types is independent of the choice of which book of each type.

For 14 books we have $14! > 87$ billion permutations: $14! = 87,178,291,200$ orders in which 14 books can be placed on the shelf. In how many ways can we shelve the 14 books so that books of the same type are together? Here we combine permutations by the product rule. We can choose how we permute the math (6!), the philosophy (5!), and the religion books (3!) in their own groups, then choose one of the 3! orders to shelve the groups. Each way gives a different shelving order, with $6! \cdot 5! \cdot 3! \cdot 3! = 3,110,400$. When communicating to others, I like to first present the factorials unexpanded, so it is easier to see where the numbers come from, then the final value to show how large the count is. If you are going to omit one, omit the final value, since the factorials are more informative.



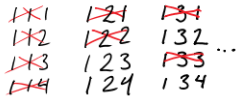
In how many ways can we shelve any three of the 14 books, of any type?

We just stop early: $14 \cdot 13 \cdot 12$ is the number of ways, which can be written as $14!/11!$. Since order on the shelf matters, we count tuples. In general, the *k-permutations* from n items are the k-tuples of those n elements that have no repeats. The product rule says their number, $P(n,k)$, is the first k terms of $n!$, namely

$$P(n,k) = \prod_{n \geq i > (n-k)} i = \underbrace{n \cdot (n-1) \cdots (n-k+1)}_{k} = n!/(n-k)!.$$

Some mathematics texts define *falling power* notation, $n^{\underline{k}} = n!/(n-k)!$; calculators typically use P^n_k or ${}_nP_k$ for the number of *k-permutations* of n items.

The number of k-tuples from n items, allowing repetition, is n^k by the product rule (we make k independent choices from n items.) We can calculate the number of these k-tuples that have at least one item repeated, $R = n^k - n!/(n-k)!$. Here we use the sum rule as subtraction: the set of all k-tuples from n items can be split into the k-tuples with no repeats (the k-permutations, counted by $n!/(n-k)!$), and the k-tuples with at least one repeat (counted by R). Thus, by the sum rule, $n!/(n-k)! + R = n^k$. This calculation works whenever $n > 0$ or $k > 0$. Since 0^0 is undefined, we should avoid trying to make 0-tuples out of no elements.



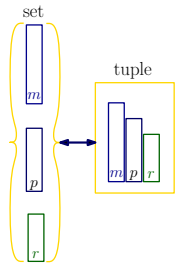
3.2.3 Combinations and choose

A *k-combination* is a set with k elements chosen from a set of n possible items. Since these are sets, no repeats are allowed. The number of possible k-combinations of n items is denoted $\binom{n}{k}$, which can be read as n *choose* k or as the *binomial coefficient* n, k. On a regular keyboard it can also be written as $C(n,k)$; calculators sometimes use ${}_nC_k$ or $COMB(n,k)$. As we shall see later in this section, it can be calculated using factorial as $\binom{n}{k} = n!/(k!(n-k)!)$, but I recommend getting used to it as "n choose k."

For example, if you want to choose 3 out of 14 books, you can do so in $\binom{14}{3} = 364$ ways.* A 3-combination does not put the books in order; it just gives a set of three books.

*This is not (14/3), so don't put in the division operation!

I can combine chosen sets by sum and product rules. Want to know how many sets of three books have only one type of book? For math $\binom{6}{3} = 20$, for philosophy $\binom{5}{3} = 10$, and for religion $\binom{3}{3} = 1$, so the sum rule says a total of 31 sets have a single type. How many ways can I choose two math books and one non-math? Take the product of the ways to choose the two math books with the number of non-math to choose from: $\binom{6}{2} \cdot 8 = 120$. The number of sets of three that don't have all three types of books? $\binom{14}{3} - 6 \cdot 5 \cdot 3 = 274$.

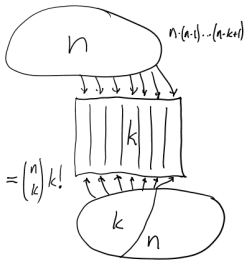


That last calculation looks a little strange. It is clear that $\binom{14}{3}$ counts an unordered set that chooses 3 out of 14 books, but isn't the calculation $6 \cdot 5 \cdot 3$

calculating in order, like the product rule counts tuples? In fact, what we are doing is transforming each set with one math, one philosophy, and one religion book, $\{m, p, r\}$, into a single corresponding tuple (m, p, r) by putting the books in order by type. (We can do this only for sets with one of each type!) We can turn each such tuple back into a single set by ignoring the order. This invertible function (a *bijection*, as defined in subsection 6.2.2) shows that we have the same number of sets with one of each book type as we have tuples from math×phil×relig. This set of tuples is easily counted by the product rule. Many counting problems are solved by transforming to some set that is easier to count; the last section of this chapter uses three transformations to count multisets.

Ok, so we have notation for the number of k-combinations of n items as $\binom{n}{k}$, but what is this number? Just like we can twist the sum rule to subtract, we can twist the product rule to divide. I state the result as a *lemma*[*] because its proof demonstrates the common mathematical trick known as *combinatorial proof*: observing that two formulae give equal values because they count the same set in two different ways.

[*] A "lemma," from a Greek word meaning "supposition," is a little theorem proved on the way to something bigger.



Lemma 3.2.1. *For $0 \le k \le n$, the number of k-combinations of n distinct items, $\binom{n}{k}$, equals $\frac{n!}{k!(n-k)!}$.*

Proof. We have already seen one way to count all k-permutations from n distinct items: We have n choices for the first element, $n-1$ for the second, ..., so the product rule says there are $P(n,k) = n!/(n-k)!$ k-permutations. Here is a different way to get the same count: choose a k-combination – a set of k of n items – then choose the order for those k elements. Those choices are independent, so the product rule says the number is $\binom{n}{k} \cdot k!$. More importantly, we get every k-permutation exactly once, because from a k permutation we can read off both the k-combination chosen and the permutation of those k elements. Therefore, $n!/(n-k)! = \binom{n}{k}k!$, and since $k! > 0$ we can divide to get the claimed result. QED

QED, the traditional end-of-proof mark, is Latin, *"Quod Erat Demonstrandum,"* ("which was to be proved"), or English, *"Quite Easily Done."*

Let's check the boundary cases. The formula correctly says that there is one way to choose a set of 0 out of $n \ge 0$ items, the empty set, and only one way to choose n, the entire set. (Recall that order of choosing doesn't matter for sets, just whether I choose something or not.) For any integer $k < 0$ or $k > n$, let's say that $\binom{n}{k} = 0$, since that is there are no ways to choose a k element set in these cases.

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
    1   6  15  20  15   6   1
```

The 'choose' numbers (binomial coefficients) can be tabulated using Pascal's triangle, where the numbers in each row after the first are the sum of the two numbers above. (We don't print the zeros so the triangle is more easily seen.) The defining formula $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ comes directly from the sum rule, because we can choose k of $[1..n]$ either by choosing element n and

then choosing $k-1$ elements from $[1..(n-1)]$, or by not choosing element n and instead choosing k elements from $[1..(n-1)]$. (These two cases are exclusive, and order doesn't matter, so it is fine for us to decide about the last element first.) Pascal's triangle has many wonderful properties.

Why are the 'choose' numbers called binomial coefficients? Because they are the coefficients of the two-variable polynomial in the binomial theorem:

Theorem 3.2.2. *For any non-negative integer $n \in \mathbb{N}$ and reals a, b, the nth power of the binomial $a+b$, is the sum over all integers:*

$$(a+b)^n = \sum_k \binom{n}{k} a^{n-k} b^k.$$

Why is this true? First, note that I can sum over all integers since $\binom{n}{k} = 0$ for any $k < 0$ and $k > n$. Next, consider fully expanding the product

$$(a+b)^n = \underbrace{(a+b)(a+b)\cdots(a+b)}_{n \text{ terms}}.$$

The campus tour leader said, "This is our binomial cafeteria." A bright high schooler asked, "Is that because there are so many combinations to choose from?" She replied, "No, you should buy no meal here."

There are a total of 2^n terms, if you don't commute "a"s and "b"s, because n times you choose to multiply either "a" or "b." The coefficient of $a^{n-k}b^k$ is the number of terms with $n-k$ "a"s and k "b"s, which is the number of ways to choose k positions for the "b"s from $[1..n]$ and fill in the rest with "a"s. This number is $\binom{n}{k}$.

Let's check the boundary cases. The only way to get a^n is to choose no "b"s, and $\binom{n}{0} = 1$. The only way to get b^n is to choose n "b"s, and $\binom{n}{n} = 1$. So these work, too.

Several nice observations about binomial coefficients and their sums come from plugging in values for a and b in Theorem 3.2.2, including:

$$\sum_{0 \le k \le n} \binom{n}{k} = 2^n. \qquad \text{pluging in } a, b = 1$$

$$\sum_{0 \le k \le n} (-1)^k \binom{n}{k} = 0. \qquad \text{plug in } a = 1,\ b = -1 \tag{3.1}$$

$$\sum_{0 \le k \le n} (-2)^k \binom{n}{k} = (-1)^n. \qquad \text{plug in } a = 1,\ b = -2$$

$$\binom{n}{k} = \binom{n}{n-k} \qquad \text{by swapping } a \text{ and } b.$$

3.3 A mixed example: distributing donuts

Three friends want to share an assortment of a dozen donuts (all different). Each one secretly has a favorite and a least favorite donut. Consider these questions about the number of ways they can choose favorites and least-favorites, and how they can distribute the donuts.

These questions are frivolous, but have a serious point: it is not easy to communicate precisely, or count carefully, but well-chosen notation can

help. This can be of vital importance when you and a client are trying to agree on the specification of code to be written to solve a problem.

Q1. In how many ways can they share the assortment, with each getting the same number of donuts and none left over? ?

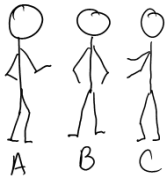
Q2. In how many ways can the favorite donuts be chosen so that it is possible for each to get their favorite? ?

Q3. In how many ways can the least-favorite donuts be chosen so that it is possible for each to avoid their least favorite? ?

Q4. In how many ways can both favorites and least-favorites be chosen so each can get their favorite and none get their least favorite? ?

Q5. If the favorites and least-favorites happen to be six different donuts, how many ways can they share the assortment so each gets their favorite and avoids their least favorite? ?

A1. How many ways are there to give four donuts each to A, B and C; denote donuts by [1..12] with no repetition.



Here we assume that the order the donuts arrive is not important; after all, I get to decide what order I eat my donuts. Since all that matters is who gets what donuts, A gets a set, B gets a set, and C gets a set. We might as well first chose the set for A, in $\binom{12}{4}$ ways, then the set for B in $\binom{8}{4}$ ways, then the set for C in $\binom{4}{4} = 1$ way, since C gets the donuts remaining. Because the number of choices are independent, the product rule applies. I'm happy to leave the answer as a product of binomial coefficients, because it makes it easier to see where the number came from, but for completeness $\binom{12}{4}\binom{8}{4} = 34,650$.

We can obtain the same answer from the number of permutations of donuts, 12!. Let K denote the number of different assignments of sets of donuts to A, B, and C. Each permutation of the 12 donuts can be made into an assignment by simply giving the first four donuts to A, the next four to B, and the rest to C.

Each assignment actually comes from many permutations: We can list donuts of A in 4! ways, then those of B in 4! ways, then those of C in 4! ways. In fact, the K assignments give $K(4!)^3$ permutations of the 12 donuts, and because we can recover the assignment from any permutation, we learn that $12! = K(4!)^3$. Solve for K to get the same count: $K = 12!/(4!)^3 = 34,650$.

There is always more than one way to count, if you know the basic rules.

A2. In how many ways can the favorites be chosen so that it is possible for each friend to get his or her favorite?

Note that the question is not how many ways can the donuts be distributed after the favorites are chosen, but how many ways can the favorites be chosen. Take a minute to understand the question and doodle a diagram for what is happening: A, B, and C are each going to choose from donuts [1..12]. Does the word "choose" mean we are looking for a combination? Does the order matter? Can a favorite be repeated?

If two people choose the same favorite, then not everyone can get his or her favorite. The inverse is also true: If no two people choose the same favorite, then everyone can get his or her favorite. So, we are looking for choices in which no favorite is repeated.

The order that A, B, and C choose favorites is unimportant, but who chooses what donut is important. If A and B swap favorites, then that is the same set of favorites chosen, but a different way to choose. So a choice is not a set but a tuple; not a combination, but a permutation. We count the ways for A, B, and C to choose with no repetition, which is $12 \cdot 11 \cdot 10 = 1320$.

Let's also answer three questions that weren't asked to see the contrast. Suppose that, before anyone takes a donut, A, B, and C each independently decide on their favorite type.

a. What fraction of choices do not have repeats?

   A, B, and C would have 12^3 ways to choose with repetition. This means that $12 \cdot 11 \cdot 10/12^3 = 110/144 > 75\%$ of the possible choices of favorites are conflict-free.

b. How many different sets of donuts could be the set of chosen favorites?

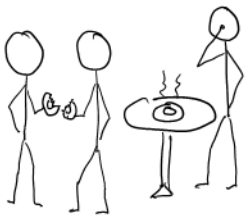
   The set of favorite donuts may contain 3, 2, or 1 donuts, since people choose with repetition. The number of possible sets of favorites is $\binom{12}{3} + \binom{12}{2} + \binom{12}{1} = 298$.

c. How many ways can donuts be distributed so everyone gets their favorite?

   There are two cases:

   c.i) if two friends have the same favorite: 0 ways.

   c.ii) if no two friends have the same favorite: 1,680, since we hand out the favorites, then choose 3 of 9 remaining for A, 3 of 6 for B, and 3 of 3 for C, for a total of $\binom{9}{3}\binom{6}{3}\binom{3}{3}$ ways.



A3. In how many ways can least-favorite donuts be chosen so that it is possible for each friend to avoid his or her least favorite?

Again, we are asking about choosing donuts, not distributing them yet. We fail only if everyone dislikes the same donut, which can happen in 12

ways out of the 12^3 choices allowing repetition. (So 143/144 if this was a random choice.)

We could also count how many ways to distribute the donuts so no-one gets their least favorite. Here we have three cases:

A3.i) if all three friends have the same least favorite, someone has to get it: 0 ways.

A3.ii) if two have the same least favorite: 8,400. Suppose A,B agree that they dislike donut 12, so give 12 to C. C dislikes something else, so choose whether to give that to A or B. Then C chooses 3 of 10, the one who got C's least-favorite chooses 3 of 7, and the other gets the remaining 4. All choices are independent, so the product rule applies. The total is $2\binom{10}{3}\binom{7}{3}\binom{4}{4}$.

A3.iii) if three different donuts are least favorites: 10,920. We use the sum rule to break this into two subcases, depending on whether 3 or 2 people eat the three least favorites. In each subcase the product rule will apply.
First subcase: we can give each friend another's least favorite in two ways, and then give each friend 3 more donuts, for a subtotal of $2 \cdot 1,680$.
Second subcase: one friend can take the other two's least favorites (3 ways) and another take theirs (2 ways). The first friend needs 2 more, the second 3 more, and the last takes the remaining 4, for a subtotal of $6\binom{9}{2}\binom{7}{3}\binom{4}{4} = 7,560$.

Note that we don't add the numbers from the three main cases, because only one of the cases occurs, depending on the friends' prior choices. In case A3.iii) we do add because there we are using the sum rule to count different ways of distributing donuts *after* the friends have chosen.

A4. In how many ways can both favorites and least-favorites be chosen so each friend can get their favorite and none get their least favorite?

Again, we are talking about friends' choices, not distributing donuts yet. Let everyone chose favorites first, in $12 \cdot 11 \cdot 10$ ways. A can't pick his or her own favorite as a least favorite, but could pick B's or C's, so let's split the cases by the number of people whose least favorite overlaps with someone else's favorite.

0: No-one's least favorite overlaps the favorites. Each chose from 9 non-favorites, so there are 9^3 possibilities minus the 9 ways they might all choose the same least favorite.
1: Choose who disliked one of the favorites (3 ways), which favorite they dislike (2 ways), and the non-overlappers have 9^2 choices.
2: Choose which two disliked favorites (3 ways), which favorites they dislike

(2^2 ways), and the non-overlapper has 9 choices.
3: Choose which favorites each of the three dislike (2^3 ways).

Working the cases was getting tedious until I wrote it as a summation and recognized that it simplifies by the *binomial theorem*: $\left(\sum_{i\in[0..3]}\binom{3}{i}2^i 9^{3-i}\right)-9 = (2+9)^3 - 9 = 11^3 - 9$.

Whenever a complicated count gives a short formula like this, check if there is a simpler way to count. Here, A, B, and C each choose their least favorite from 11 donuts (to each avoid their favorite). If they happened to all three choose the same donut (which must be one of the 9 non-favorites) then throw that choice out. Final answer: $12\cdot 11\cdot 10\cdot(11^3-9) = 1,745,040$, which is just over 58% of all 12^6 choices, or over 75% of the $(12\cdot 11)^3$ ways if each pick separate favorite and least-favorite (as you would expect them to do).

A5. If the favorites and least-favorites happen to be six different donuts of the dozen, how many ways can they share the assortment so each gets their favorite and avoids their least favorite?

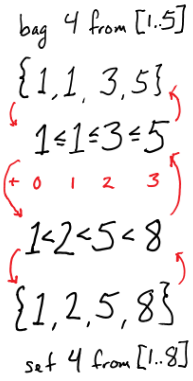
Here we combine cases c.ii) and A3.iii) by the sum rule to distribute donuts: We first give each of the friends their favorites. Next, we decide if each gets one of their friends' least favorite (2 ways), or if we give 2 to one, 1 to another, and 0 to the third (6 ways). Then we complete each set of four by making independent choices that multiply by the product rule. Total number of ways: $2\binom{6}{2}\binom{4}{2}\binom{2}{2} + 6\binom{6}{1}\binom{5}{2}\binom{3}{3} = 540$.

Notice how the sum and product rules have helped us break complex counting problems into simpler steps.

3.4 Counting multisets

For selecting k from n items, we have counted the numbers of possible sets (unordered, no repeats; $\binom{n}{k}$), tuples (ordered, repeats allowed; n^k), and k-permutations (ordered, no repeats; $n!/(n-k)!$). For completeness, we should count the *bags* or *multisets*, which are unordered but allow repeats. Here is how we can transform from multisets, to special tuples, to special k-permutations of more items, and back to k-combinations to show that the correct count is $\binom{n+k-1}{k}$. Get your scratch paper ready.

We better start with some examples of sets of all possible multisets from [1..3] for different k. Remember, a multiset can have repeats, but order doesn't matter.
$k = 1$ is boring: Only three $\{\{1\},\{2\},\{3\}\}$.
$k = 2$: Only three sets, but six multisets: $\{\{1,1\},\{1,2\},\{1,3\},\{2,2\},\{2,3\},\{3,3\}\}$
I'm quickly tiring of typing set braces, so allow me to type these as sets of strings:
$k = 2$ again: $\{11, 12, 13, 22, 23, 33\}$; six ways.
$k = 3$: $\{111, 112, 113, 122, 123, 133, 222, 223, 233, 333\}$; ten ways.



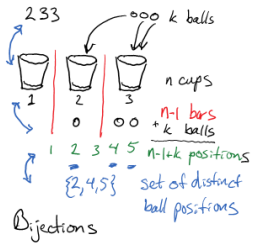


Figure 3.1: Counting multisets; the 'beer pong' view

*This *balls and bars trick* is worth remembering.

Recognize that strings are tuples, so I have essentially transformed each multiset into a unique tuple of numbers in non-decreasing order. From such a tuple, I can recover the unique multiset by simply ignoring the order of elements. Thus, multisets are paired one-to-one with k-tuples from n items that are in non-decreasing order. We can use this to make the fifteen multisets of $[1..3]$ with $k = 4$: extend each 3-letter string into one or more non-decreasing 4-letter strings. Note that 111 extends to 1111,1112, and 1113, but 113 extends only to 1133.

Now, consider a tuple $(t_0 \le t_1 \le \cdots \le t_{k-1})$ where each $t_i \in [1..n]$. For each $i \in [0..k-1]$, let $u_i = i + t_i$. I claim that this makes a strictly increasing k-tuple $(u_0 < u_1 < \cdots < u_{k-1})$ where each $u_i \in [1..n+k-1]$. Moreover, this transformation is reversible: starting with any increasing k-tuple $(u_0 < u_1 < \cdots < u_{k-1})$ where each $u_i \in [1..n+k-1]$, simply let $t_i = u_i - i$ for all $i \in [0..k-1]$. Thus, non-decreasing k-tuples from n items are paired one-to-one with increasing k-tuples from $n+k-1$ items.

But now we have eliminated the repetition, so these increasing k-tuples are clearly paired one-to-one with the k-combinations chosen from $[1..n+k-1]$.

So, rather than count the set M of all multisets of k from $[1..n]$, transform them to the set N of non-decreasing k-tuples from $[1..n]$. Transform that to the set I of increasing k-tuples from $[1..n+k-1]$, and that to the set S of k-combinations from $[1..n+k-1]$. Each set M, N, I, and S has the same number of elements, because we can pair up the elements in adjacent sets. But S is easy to count; it has $\binom{n+k-1}{k}$ elements.

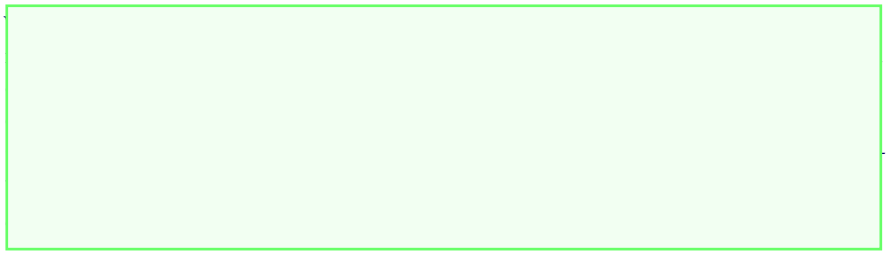
That was pretty dense, so let's do it again, using a different view of the transformations that involves throwing balls into cups or bins.

Think of the initial problem as throwing k identical balls into bins labeled $[1..n]$. Each choice of k elements with repetition corresponds to exactly one result of throwing balls into bins, and vice versa: 111 means three balls in the first bin, and 123 means one ball each in bins 1, 2, and 3. It is clear that this transformation can be inverted, so we have a bijection from the multisets of k elements from $[1..n]$ to the results of throwing k balls into n bins.

Transform again: imagine the bin boundaries as $n-1$ vertical bars, and consider all mixed sequences of k balls and $n-1$ bars. For example, the sequence 111 is $\circ\circ\circ\|$, 123 is $\circ|\circ|\circ$, and 233 is $|\circ|\circ\circ$. Again, this is clearly invertible, so we have another bijection.

But counting ball/bar sequences is easy by a third transformation—we choose the positions of the k balls from the set $[1..n-1+k]$ with no repeats, and fill in the other positions with bars.* This can be done in $\binom{n+k-1}{k}$ ways. Because this transformation is also invertible, the composition of the three is a bijection. Thus, the number of k-element multisets from $[1..n]$, allowing repetition, equals the number of k-element sets from $[1..n-1+k]$ without repetition.

So, suppose that you have 14 books and three shelves (each of which can hold all the books). How many ways can you put the books on shelves, where the number and order on each shelf matters??



3.5 Summary

This chapter gives basic definitions for sets and tuples, then explores what these definitions imply by checking ordinary examples, by checking extreme examples (like the empty set or empty tuple), and by counting how many things are defined.

Each of the structures and concepts defined in this chapter is extended in later chapters. After the next chapter formally defines quantifiers, chapter 5 defines more operations for sets, chapter 6 returns to counting, and chapter 8 introduces recursive definitions to formally define sets, tuples, lists, strings, series, and summations of arbitrary size.

Here is how the counting rules and concepts will be restated using notation and operations from upcoming chapters:

Sum rule For disjoint sets A and B, the size of the union is the sum of the sizes of each set, $|A \uplus B| = |A| + |B|$.

Product rule For counting pairs, $|A \times B| = |A| \cdot |B|$. This is why there are n^k k-tuples from n elements allowing repetition, and why a set S has $2^{|S|}$ subsets.

Factorial $n! = n(n-1)\cdots 2\cdot 1$ counts the orders of n items without repetition.

Permutations aka Falling powers $P(n,k) = n^{\underline{k}} = n!/(n-k)!$ counts the orders of k of n items without repetition.

Combinations aka Binomial coefficients $\binom{n}{k} = C(n,k) = n!/((n-k)!k!)$ counts the ways to choose an unordered set of k of n types of item without repetition.

A frivolous example of figuring out the number of ways three friends can select from a dozen donuts illustrates that careful counting can clarify questions stated in English. The next chapter enriches our logic with predicates and quantifiers to allow us to say more in unambiguous mathematical notation.

3.6 Exercises and Explorations

Quiz Prep 3.1. Answer these questions on sets and tuples:

1. Let $A = \{a, \emptyset\}$. Are these statements true (T) or false (F)?

   ? $a \in A$? $\{a\} \in A$? $a \subseteq A$? $\{a\} \subseteq A$
   ? $\emptyset \in A$? $\{\emptyset\} \in A$? $\emptyset \subseteq A$? $\{\emptyset\} \subseteq A$

2. What is the smallest set that contains, as an element and a subset, both the empty set $\emptyset$ and the set $\{a, b\}$? .

3. Which of these can be the Cartesian product of two sets $A \times B$? For each that can, give an example A and B.

   (a) $S = \{(a, 1), (b, 1), (a, 2), (b, 2), (a, 3), (b, 3)\}$. .
   (b) $T = \{(1, 1), (1, 2), (2, 2), (3, 2), (3, 1), (2, 1)\}$. .
   (c) $U = \{(a, 1), (b, 2), (c, 3)\}$.

Quiz Prep 3.2. What are the following counts?

1. How many elements are in the set $\{t, o, y, o, t, a\}$? ?
2. How many different sets can be made using up to n given elements? ?
3. How many different k-tuples can be made from n types of elements? (Repeats are allowed in a tuple.) ?
4. How many k-tuples made from n types of elements use at least two types? ?
5. How many k-tuples can be made from n elements with no repeats? ?
6. How many sets of size k can be made from n elements? (A set has no repeated elements.) ?
7. How many bags of size k can be made from n elements? (A *bag* or multi-set is an unordered collection that allows repeated elements.) ?
8. How many k-tuples from $[1..n]^k$ have their elements in increasing order? ?
9. How many k-tuples from $[1..n]^k$ have elements in non-decreasing order? ?

Quiz Prep 3.3. In a universe U with n elements,

1. In how many different ways can you make a set, $A \subseteq U$?

2. In how many different ways can you make a pair of sets (A, B), with $A \subseteq U$ and $B \subseteq U$?

3. How many ways can you choose a pair (A, B) so that A is a subset of B?

4. How many ways can you choose a set of two sets $\{A, B\}$? ?

5. How many ways can you choose a set of two sets $\{A, B\}$ so A is a subset of B? ?

Exercise 3.4. Assume that sets A and B do not contain tuples. Under what conditions does $A \times B = B \times A$? Be complete. ▶

Exercise 3.5. Combinatorial Pizza offers small, medium, and large pizzas with 14 possible toppings from 3 categories:

- Cheese (2): Mozzarella, Feta
- Veggie (7): Mushrooms, Peppers, Onions, Olives, Capers, Artichoke, Pineapple
- Meat (5): Salami, Pepperoni, Ham, Salmon, Anchovies

Combinatorial Pizza has five specials:

- Sampler: three different toppings from any categories.
- Balanced Diet: one topping from each of the three categories.
- Carnivore: You may choose one to four different kinds of meat.
- Vegan: Any size pizza with three Veggie toppings. (You can order more than one of a topping—my daughter likes triple pineapple.)
- Gut-buster: A large pizza with up to five toppings. (Ever had quintuple anchovies?)

The volume of a pizza of thickness a and radius z: pi $z z a$. -Wolfram Alpha

Let's assume that you don't care in what order the pizza chef puts the toppings on your pizzas. (E.g., They put the cheese first, even if you ask otherwise.) So, what you are ordering is a set of toppings (possibly a multi-set for the Vegan and Gut-buster).

In answering a question like this, the formula is more informative than the number, so be sure you show the formula. Each letter has a pop-up with the correct number so that you can check yourself.

1. How many different ways can you order a medium or large "Sampler" pizza? ?
2. How many different ways are there to order a large "Balanced Diet" pizza — one topping from each of the three categories? ?
3. How many different ways are there to order two small "Balanced Diet" pizzas? ?
4. How many different "Carnivore" pizzas can be made? ?
5. How many different small "Vegan" pizzas are there? ?
6. How many different "Gut-buster" pizzas are there? ?

▶

Exercise 3.6. On a common single dial padlock, with numbers 1-40, a combination* is a 3-tuple.

How many combinations does such a lock have? ?

Since you can test all third digits with a single slow turn, how many pairs of the first two digits are there? ?

On some locks, it is enough to dial the first and second digit to within ±2. In that case, how many "slow turns" suffice to try all combinations? ?

*Should be called a "3-tuple" lock, because order is important and repeats are allowed.

▶

Exercise 3.7. In activity games, board games, and card games, players are often arranged in a circle. Sometimes the capability or personalities of the players to your left or right (or both) make a difference in your chances to win the game. See if you can get the same answer as the pop-up before looking at the reasons.

1. How many different orders are there for n players? Two orders are considered the same if and only if every player has the same player to their right. ?

2. What if $n = 2m$ players come as m pairs that want to sit next to each other? Now how many orders? ?

3. You need to choose a set of k of the n people to help you; in how many ways can you do so if order does not matter? ?

4. You want to pick the k of $n = 2m$ people that came in pairs so you take at most one of any pair; now how many ways? ?

▶

Exercise 3.8. In how many ways can I choose k numbers from $[1..n]$, disregarding order, so that no two chosen numbers are consecutive (differ by 1)? ▶

Exercise 3.9. The number of possible subsets with r elements in a set S of n elements is denoted $\binom{n}{r}$. All the formulas below come from counting these sets in different ways.

1. Why is $\binom{n-1}{r-1} + \binom{n-1}{r} = \binom{n}{r}$?

2. Why is $\binom{n}{r} = \binom{n}{n-r}$?

3. Why is $n\binom{n-1}{r-1} = r\binom{n}{r}$?

4. Why is $\sum_{0 \le k < n} \binom{k}{r-1} = \binom{n}{r}$?

5. Why is $\binom{n}{r} = \frac{n!}{(n-r)!\cdot r!}$?

Puzzle 3.10. "On the first day of Christmas, my true love gave to me, a partridge in a pear tree." In the song, "The 12 days of Christmas," how many gifts does the singer receive on each day and how many total?

Hint:

Puzzle 3.11. How many positive integers have the property that their digits are strictly increasing as you read them from left to right? (Examples: 1, 128, 123,456,789.) How many positive integers have digits that are strictly decreasing from left to right? (Examples: 1, 42, 9630.) ▶

Puzzle 3.12. A good exercise is to count the number of 5-card poker hands of different values. Let's assume a standard 52-card deck, with 13 cards (A, 2–9, 10, J, Q, K, A) in each of 4 suits (♡, ♢, ♣, ♠) and no jokers or wild cards. Ace is listed twice as it can be either low or high in straights, but not both in the same hand. See if you get the same counts as I do. As a warm-up, the number of possible hands is $\binom{52}{5} = 2,598,960$.

royal straight flush 10, J, Q, K, A of the same suit: 4 ways
straight flush 5 consecutive cards of the same suit, minus the royal straight flush: 36 ways.
four of a kind 624 ways.
full house Three of one card, two of another: 3,744 ways.
flush All five cards of the same suit, minus all straights: 5,108 ways.
straight 5 consecutive numbered cards of any suit, minus all flushes: 10,200 ways.
triple 54,912 ways.
two pair 123,552 ways.
single pair 1,098,240 ways.
none of the above 1,302,540 ways.

Puzzle 3.13.

1. Paths in a grid: In how many ways can I go from the origin $(0,0)$ to (x,y) in a grid if each move increases x by one or y by one, but not both?
2. What if I may not step on (i,j), which is in $[0..x] \times [0..y]$ but not at the origin or (x,y)?

Extension 3.14. We often use an ordered pair, (x,y), to represent a point in Cartesian coordinates. Two ordered pairs are equal, $(a,b) = (x,y)$, if and only if $a = x$ and $b = y$. Suppose that we represent an ordered pair (x,y) by the set $\{x,\{x,y\}\}$. Argue that we have the property that $\{a,\{a,b\}\} = \{x,\{x,y\}\}$ iff $a = x$ and $b = y$. (Remember that a,b might be sets themselves.) Extend this to k-tuples.

Chapter 4
First Order Logic: Quantifiers

Civilization advances by extending the number of operations we can do without thinking about them. Operations of thought are like cavalry charges in a battle - they are strictly limited in number, they require fresh horses and must be made only at decisive moments.

—Alfred North Whitehead

In propositional logic, we started by translating into mathematical notation specific statements, such as, "If it did not rain this morning, I commuted by bike." First order logic adds quantifiers so that we can translate more general statements like, "For every weekday last month, if it did not rain in the morning, then I commuted by bike." The mathematical notation is precise, unambiguous, and can be manipulated with little thought, once you know the rules.

That is not to say this is easy – only the first section of this chapter is easy, because it introduces the vocabulary. The second section practices using the vocabulary – notice how translating English statements into notation forces us to recognize and resolve ambiguities. Saying exactly what you mean is hard work, but that is what we need to do to give instructions to computers, so it is good to have notation to help us.

Objectives: By working through this chapter and the exercises, you will be able to precisely interpret statements using predicates and the quantifiers for all, $\forall$, and there exists, $\exists$. You will be able to identify free and bound variables in statements, to express complex statements about sets or tuples using nested quantifiers, and to negate quantified statements, revealing connections between $\forall$ and $\exists$, and connections to de Morgan's laws for logic.

4.1 Quantified statements

Before defining quantifiers, we need one special type of function: A *predicate* is a function that maps each possible input to either T or F, *true* or *false*. As examples, here are two predicates each taking their input x from a set of days, D:

$p(x)$ = "It rained in the morning on day x," and

$q(x)$ = "I commuted by bike on day x."

Combining these, we can write the statement "If it did not rain in the morning on day x, then I commuted by bike on day x" as $\overline{p(x)} \to q(x)$. This statement has a *free variable* x; pick a day from D for x to get a statement

that is true or false for that day. Thus, predicates can be used as propositions in a logical statement if we supply an input $x \in D$.

Suppose that I want to claim this statement is true for many days, not just one day. Restrict the set of days $D = \{d_1, d_2, \ldots, d_k\}$ to those days in which I have a job to commute to, since the statement $\overline{p(x)} \to q(x)$ would be false, for example, on any sunny morning in the 1800s. Here are three equivalent ways to write the statement, "For every day in D, if it did not rain in the morning, then I commuted by bike." The first uses a big and ($\bigwedge_{x\in D}$) much like $\sum$ *summation notation*. The second expands this as many little ands ($\wedge$), using ellipses ($\cdots$) and trusting the reader to complete the pattern. The third introduces 'for all' ($\forall_{x\in D}$), the *universal quantifier*, and is the most common (although my use of a subscript for $x \in D$ to match the big-and notation is not common).

Three logicians walk into a bar, and the bartender asks, "Does everyone want beer?" The first says, "I don't know." The second says, "I don't know." The third says, "Yes."

$$\bigwedge_{x\in D} \overline{p(x)} \to q(x) \equiv (\overline{p(d_1)} \to q(d_1)) \wedge \cdots \wedge (\overline{p(d_k)} \to q(d_k))$$
$$\equiv \forall_{x\in D}(\overline{p(x)} \to q(x)).$$

In words, my claim is true if and only if the conditional is true for every x in D. The first and third notations apply even if D is an infinite set.

To say, "There was a day in D that it rained in the morning," we introduce 'there exists' ($\exists$), the *existential quantifier*:

$$\bigvee_{x\in D} p(x) \equiv p(d_1) \vee p(d_2) \vee \cdots \vee p(d_k) \equiv \exists_{x\in D}\,(p(x)).$$

In words, the claim is true if and only if it is true for at least one x in the quantification domain D.

4.1.1 Writing a quantified statement

Let's look at the four parts of a quantified statement:

$$\forall_{x\in D}\,(p(x))$$

First we have the quantifier symbol: the *universal quantifier* 'for all,' $\forall$, or the *existential quantifier* 'there exists,' $\exists$.

Second we have a *variable*, which is x in this example. We may choose any variable name that does not already have a value assigned – x must be a *free variable* that can be bound to different values inside the exprression. Without changing the meaning of the statement, we may replace x with any variable name that is neither already assigned a value nor already appearing inside p.

Third we have the *quantification domain*, D, which is the set of all values that the variable can be bound to. A quantified statement is incomplete

without a quantification domain, though we often omit the symbols when the set can be understood from context. For example, if we know we are speaking of the set of reals, the set of integers, or the set $\{1, \ldots, n\}$, we may simply write $\forall_x\ p(x)$, rather than $\forall_{x\in\mathbb{R}}\ p(x)$, $\forall_{x\in\mathbb{Z}}\ p(x)$, or $\forall_{x\in[1..n]}\ p(x)$. (And I may also write the last as $\forall_{1\le x\le n}\ p(x)$, trusting the reader to figure out that x is the variable being defined using an existing value of n.)

Fourth we have an expression, $(p(x))$ that returns T or F for any value assigned to x; it is a predicate in x. The outer parentheses are often omitted, but when you begin to work with quantifiers, include them to remind yourself that x has a defined value only within those parentheses. We say that this is the *scope* of the variable x, and that within its scope x is no longer free, but is *bound* by the quantifier. (Expression $p(x)$ may contain its own quantifiers, which must use variable names other than x, as we'll see below, but their domains and expressions may depend on the value assigned to x. Even complex expressions must return T or F for each possible value of $x \in D$.)

To establish that statement $\exists_{x\in D} p(x)$ is true, it is enough to find any $x \in D$ that *satisfies* $p(x)$ by making it true. To establish that $\forall_{x\in D} p(x)$ is true, we must check all $x \in D$, and report 'false' if there is any x that does not satisfy $p(x)$. One way to do this is to assume an adversary gives us a generic $x \in D$ and verify that $p(x)$ is true.

Let's check the boundary cases: What if the quantification domain is empty? For consistency, we say that $\forall_{x\in\emptyset}\ p(x)$ is *trivially true* and $\exists_{x\in\emptyset}\ p(x)$ is false, regardless of the predicate $p(x)$. What if the quantification domain has a single element, $D = \{d\}$? Then $\forall_{x\in D}\ (p(x)) \equiv \exists_{x\in D}\ (p(x)) \equiv p(d)$, but this is the only case where the quantifiers are logically equivalent.

As computer scientists, we can think of a quantified statement $\forall_{x\in D}\ (p(x))$ as a 'for' loop. The variable x is like a *loop variable* created in the loop statement, which is valid only within the body of the loop.* The set D contains the values that we loop over, and predicate $p(x)$ is the body of the loop. A 'for all' quantifier returns true unless it finds a $p(x)$ that is false, in which case it returns false. A 'there exists' returns false unless it finds a $p(x)$ that is true, in which case it returns true. The only difference from a 'for' loop is that we don't know the order of evaluation, since D is a set. But order doesn't matter since we are computing 'and' or 'or, ' which are both commutative and associative.

*Java and C++ have this scope limitation for variables defined inside a loop.

The notation for quantifiers has many variations and abbreviations. I've already mentioned suppressing the domain D and omitting parentheses around the expression. I should mention that my use of subscripts for the variable and domain is not standard; I do that to more clearly separate the variable and domain from the predicate and to match the notation for summation, big-and, and big-or. Some people put parentheses around the quantifier itself, writing $(\forall x)p(x)$ instead of $\forall_x(p(x))$, but I find that

silly because the quantifier by itself is incomplete, whereas $p(x)$ by itself makes sense as a statement about a free variable x. Others separate the quantifier and predicate with a comma, writing $\forall x, p(x)$, but this also gives the misleading impression that the parts are separate and equal, when really the quantifier is wrapped around a predicate that could stand on its own.

In English, note the difference between the three statements "x is positive," "Every x in the set S is positive," and "Some x in S is positive." The first, written mathematically as $(x > 0)$, claims that the number someone has given us is positive. This claim is true or false, depending on x, so this is a predicate with input x. The second, written $\forall_{x \in S}\ (x > 0)$, claims that that every number in S is positive. This claim will be true or false, if the set S is fixed. If the set S is considered a free variable, this is a predicate with input S. Similarly, the third, written $\exists_{x \in S}\ (x > 0)$, claims that there is a number in S that is positive. Again, this will be true or false, depending on S, so we have another predicate with input S. In the first statement, x is a free variable; in the second and third, x is bound by the quantifier and set S is the free variable.

For statements about a k-tuple, the quantification domain is often the set of element positions or *indices*, $[1..k]$. For example, suppose that we want to formally say (without ellipses) that the k-tuple $(a_1, a_2, \ldots, a_k)$ has its elements in increasing order, $a_1 < a_2 < \cdots < a_k$. We can quantify over indices and say $\forall_{i \in [2..k]}\ (a_{i-1} < a_i)$. We would not write $\forall_{x \in (a_1, a_2, \ldots, a_k)} \ldots$ – first, because a tuple is not a set, and second, because without indices we cannot refer to the element before or after x or identify a specific element in the case of a repeat. We also would not write $\forall_{a_i}\ (a_{i-1} < a_i)$, because the quantifier needs to be able to change the value of its variable, but each a_i is a value in a tuple that we should not modify. We'll consider an extended example with tuples in section 4.2.

Remembering that $\forall$ is a big 'and' and that $\exists$ is a big 'or' we can often use associative, commutative, and distributive properties of and/or to pull quantifiers out to the front. For example, in the statement $p(x) \vee (\forall_y\ q(y))$, if y is not free variable inside $p(x)$, then we can equivalently write $\forall_y\ (p(x) \vee q(y))$. Likewise, $p(x) \wedge (\exists_y\ q(y)) \equiv \exists_y\ (p(x) \wedge q(y))$. Statements with conditionals or biconditionals should be rewritten (at least temporarily) as equivalent and/or statements if you want to move quantifiers forward. A reason to move quantifiers forward is that subsection 4.1.4 gives rules of inference that strip away or add on quantifiers, allowing us to check simpler expressions.

4.1.2 Negation for quantifiers

Since 'for all' is a big 'and,' and 'exists' is a big 'or,' de Morgan's laws say that the negation of one is the other (with its predicate negated.) That

is, $\overline{\forall_x\, p(x)} \equiv \exists_x\, \overline{p(x)}$, and $\nexists_x\, p(x) \equiv \overline{\exists_x\, p(x)} \equiv \forall_x\, \overline{p(x)}$.* In words, the first equivalence says that to disprove $\forall_{x\in D}\, p(x)$, it is enough to show a single counterexample—an $x_0 \in D$ with $\overline{p(x_0)}$. To prove it, however, we must be able to demonstrate that $p(x)$ is true for any value of $x \in D$ that our malevolent opponent gives us. The second string of equivalences says that to disprove $\exists_{x\in D}\, p(x)$, we must be able to demonstrate, for any x given by our malevolent opponent, that either $x \notin D$ or $\overline{p(x)}$. To prove it, however, a single example $x_0 \in D$ with $p(x_0)$ is enough. This is consistent with the definitions from the previous section.

*Most math fonts include $\nexists$, which means $\neg\exists$, but have no symbol for $\neg\forall$. We seem to be more concerned with non-existence than non-universality.

The definitions for quantifying over an empty domain are also consistent with negation, since $\forall_{x\in\emptyset}\, p(x) \equiv \neg\exists_{x\in\emptyset}\, \overline{p(x)} \equiv T$, no matter what p is.

You must remember to swap quantifiers when you negate, just as you must remember to swap 'and's and 'or's in de Morgan's laws. Saying, "I don't always ride my bike $(\neg\forall_{x\in D}\, q(x))$," is quite different from "I always don't ride my bike $(\forall_{x\in D}\, \neg q(x))$." For me, the first is true and the second false. The first is equivalent to "There was a day I didn't ride my bike $(\exists_{x\in D}\, \neg q(x))$," and the second to "On no day did I ride my bike $(\neg\exists_{x\in D}\, q(x))$." Again, for me, the first it true and the second false.

Remembering that $\forall$ is a big 'and' and that $\exists$ is a big 'or' we can use associative, commutative, and distributive properties of and/or to pull quantifiers out to the front. For example, in the statement $p(x) \vee (\forall_y\, q(y))$, if y is not free variable inside $p(x)$, then we can equivalently write $\forall_y\, (p(x) \vee q(y))$. Likewise, $p(x) \wedge (\exists_y\, q(y)) \equiv \exists_y\, (p(x) \wedge q(y))$. If you want to move quantifiers that have statements with conditionals or biconditionals, first rewrite them using and/or/not and see if you need to apply de Morgan's rules before applying associative, commutative, or distributive properties. The advantage of moving quantifiers is that they give context by defining variables, and it is easier to apply the rules of inferrence of subsection 4.1.4, which allow us to strip away or add on quantifiers.

4.1.3 Scope and nested quantifiers

Recall that a quantifier $\forall_x$ or $\exists_x$ defines the variable x only for the statement immediately following. You should not use a variable name x that is already defined, or that you wish to keep free; you also should not use the value of x outside the *scope of the quantifier*—outside the immediately following statement.

To illustrate scope, we can translate "There was a day it rained and I rode my bicycle," as $\exists_{x\in D}\, (p(x) \wedge q(x))$ and see that both happened on the same day. We can translate "There was a day it rained, and there was a day I rode my bicycle," as $(\exists_{x\in D}\, p(x)) \wedge (\exists_{x\in D}\, q(x))$, which allows the two quantifiers to use different days for their different x variables. Because of the potential for

confusion, it is better to write $(\exists_{x\in D}\ p(x)) \wedge (\exists_{y\in D}\ q(y))$, but the mathematics is the same.

To explicitly name a value for use outside a quantified statement, we could say one of the following:

1. "Since $\exists_x\ p(x)$, we can let x be a day satisfying $p(x)$."
2. "Since $\exists_y\ p(y)$, we can let x be a day satisfying $p(x)$,"
3. "Since $\exists_x\ p(x)$, we can let x_0 be a day satisfying $p(x_0)$,"
4. "Choose x to satisfy $p(x)$,"

These are mathematically the same, so the choice depends on which, in context, is the least of four evils: 1's potential confusion due to re-use of x as a placeholder in the quantifier and as a defined variable, 2's switch from y to x, which some find disorienting at first, 3's introduction of an avoidable subscript on x_0, or 4's not explicitly saying that there exists an x that can be chosen.

In English, we sometimes omit the 'for all,' or put it at the end. In math, please develop the habit of including the quantifiers at the beginning of a statement for two reasons: First, quantifiers give much-needed context: naming the variable whose value will be chosen from a named, or understood, set, either by an adversary ($\forall$) or by us ($\exists$). Second, in statements with nested 'for all' and 'there exists' quantifiers, the order is significant. For a set of people, P, we can have a predicate loves(x, y) be true whenever person x loves person y. Now compare "Everyone loves somebody," $\forall_{e\in P}\ \exists_{s\in P}$ loves(e, s), to "There is somebody that everyone loves," $\exists_{s\in P}\ \forall_{e\in P}$ loves(e, s). In the first, your adversary gives you an arbitrary e and asks, "Who does this person love?" You may pick s depending on e. In the second, however, you pick s and *then* your adversary gets to challenge you with e—all es must love the same s for you to succeed. Nested quantifiers are not difficult if you take them one at a time.

In chess, Alice moves and announces, "Mate in two," because it's harder to say, "For all of your possible moves, there is a move for me (Alice) so that for all your moves, there is a move for me so that for all your moves, there is a move for me to take your king." $\forall\exists\forall\exists$ alternation makes the game.

For practice, consider the negations. For the first, if "everyone loves somebody" is false, then $\exists_{e\in P}\ \forall_{s\in P}\ \overline{\text{loves}(e, s)}$, or "There is someone that loves nobody." The negation of the second, $\forall_{s\in P}\ \exists_{e\in P}\ \overline{\text{loves}(e, s)}$, means "Everyone has somebody that doesn't love them."

For "Everybody loves everybody," $\forall_{e\in P}\ \forall_{s\in P}$ (loves(e, s)), it does not matter which quantifier comes first: $\forall_{s\in P}\ \forall_{e\in P}$ (loves(e, s)) is equivalent. (This is again because $\forall$ is a big 'and.') Similarly, for 'Someone loves someone," $\exists_{e\in P}\ \exists_{s\in P}$ (loves(e, s)), the quantifiers can occur in either order because $\exists$ is a big 'or.' Thus, we can abbreviate these statements as $\forall_{e,s\in P}$ (loves(e, s)) and $\exists_{e,s\in P}$ (loves(e, s)). But when 'for all' and 'there exists' quantifiers alternate, we change the meaning if we reorder the quantifiers.

4.1.4 Inference: instantiation and generalization

As you can see, rewriting English statements using quantifiers forces us to be precise about what we mean. Once we have precise statements, we can simplify using rules of inference that also come from the connection of $\forall$ and $\exists$ to 'and' and 'or.'

If we know that a quantified statement is true, we use "instantiation" rules to remove the quantifier and talk about a specific instance. If we are trying to show that a quantified statement is true, we use "generalization" rules to go from a proof for a specific or generic value to a quantified statement.

Universal instantiation says that if we know $\forall_{x\in X}\ p(x)$ then we can conclude $p(a)$ for any specific choice of $a \in X$; it is an application of simplification.

Existential instantiation lets us give a name to a value that we are told exists. If we know $\exists_{x\in X}\ p(x)$ we can conclude $p(y)$ for a variable y that is not currently in use, but whose value now becomes fixed so that $p(y)$ is true.

Universal generalization says that if we know $p(x)$ is true for whatever element x of X that our adversary may challenge us with then we may conclude $\forall_{x\in X}\ p(x)$; it is an application of conjunction from table 2.5.

Existential generalization says that if we can choose a specific element $a \in X$ for which $p(a)$ is true, then we may conclude $\exists_{x\in X}\ p(x)$; it is an application of absorption.

If we know a quantified statement is true, then we use instantiation to remove the quantifiers to make claims about specific values that we have (universal instantiation) or that we give names because we know that the values exist (existential instantiation). When we want to establish that a quantified statement is true, then we either pick a value for which it is true and by existential generalization conclude that there exists a value where it is true. Or we assume that someone gave us generic values, and since we can do the proof for those, we conclude by universal generalization that the claim is true for all values. We use these all the time in proofs; so often that we may neglect to mention them.

Let me apply these rules in an example, showing in excruciating detail that there is a rational number between any two rationals. Please skip to the next subsection on first reading, and use scratch paper when you do work through it.

I know: A rational number is a fraction $x = a/b$, with $a \in \mathbb{Z}$ and $b \in \mathbb{Z}^+$. Expressed with quantifiers, $x \in \mathbb{Q}$ iff $\exists_{a\in\mathbb{Z},b\in\mathbb{Z}^+}\ x = a/b$.

I want to show: Any two rationals $x < z$ have a rational y between them, $x < y < z$. That is, I want to build the expression: $\forall_{x\in\mathbb{Q}}\forall_{z\in\mathbb{Q}}\ \Big((x < z) \to (\exists_{y\in\mathbb{Q}}\ (x < y < z))\Big)$.

I'll want universal generalization twice to add those "forall" quantifiers. I'll assume that my adversary challenges me with rationals $x, z \in \mathbb{Q}$. The "if" means cases with $x \geq z$ are vacuously true, so I'll assume the challenge has $x < z$.

To show a rational y exists, I'll pick a value $y = (x + z)/2$, then show that y is rational and that $x < y < z$. Once I know those, adding quantifiers is easy: I know $\exists_{y\in\mathbb{Q}}\ (x < y < z)$ by existential generalization, then $\forall_{x,z\in\mathbb{Q}}\ \Big((x < z) \to (\exists_{y\in\mathbb{Q}}\ (x < y < z))\Big)$ by universal generalization twice. I'm now ready to begin.

I start with the given rationals $x < z$. For each, the definition says integers exist; applying existential instantiation twice each (choosing different variable names to avoid conflict), gives me values $a, c \in \mathbb{Z}$ and $b, d \in \mathbb{Z}^+$ with $x = a/b$ and $z = c/d$.

Next, I make y fit the definition of rational by doing the arithmetic to add fractions, $y = (x + z)/2 = (ad + cb)/2bd$. The numerator is an integer, $N = (ad + cb) \in \mathbb{Z}$, and denominator is a product of positive integers, $D = 2bd \in \mathbb{Z}^+$. Applying existential generalization twice gives $\exists_{N\in\mathbb{Z},D\in\mathbb{Z}^+}\ y = N/D$, so y is rational.

Now, I manipulate the given inequality, $x < z$, to get y between. Dividing by 2, $x/2 = a/2b < c/2d = z/2$. I add $a/2b$ to both sides, then $c/2d$ to both sides, and combine the two inequalities with a common denominator $2bd$ to see* $x = ad/2bd + ad/2bd < (ad + cb)/2bd < cb/2bd + cb/2bd = z$. We've already seen how to finish from here.

*Using the arithmetic property, "Inequality is preserved when adding the same value to both sides," which in formal math is unpacked with its quantifiers: if $x < z$, then for all $\forall_{x,z,c\in\mathbb{Q}}\ (x < z) \leftrightarrow (x + c < z + c)$.

Alternatively, I could have picked $y = (a + c)/(b + d)$, which is rational since $(a + c) \in \mathbb{Z}$ and $(b + d) \in \mathbb{Z}^+$. Multiplying both sides of $x = a/b < z = c/d$ by $bd > 0$, we learn $ad < cb$. Thus, $\frac{a}{b} = \frac{a(b+d)}{b(b+d)} = \frac{ab+ad}{b(b+d)} < \frac{ab+cb}{b(b+d)} = \frac{a+c}{b+d} = \frac{ad+cd}{(b+d)d} < \frac{cb+cd}{(b+d)d} = \frac{c}{d}$.

Finally, note that a similar claim for integers is false – it is not true that between any two integers $x < z$ there lies an integer. Use de Morgan to push the negations down and state the postive claim $\exists_{x,z\in\mathbb{Z}}\forall_{y\in Z}\ (x < z) \wedge \overline{(x < y < z)}$. As an "exists" claim, we may pick our pair $x < z \in \mathbb{Z}$, like $x = 0, z = 1$. Observe that any given $y \in \mathbb{Z}$ has either $y \le x$ or $y \ge z$ and use universal generalization.

So, to convince someone that $\exists_{x\in D}\ p(x)$, we most often choose a single example, x_0, and demonstrate that $p(x_0)$ is true. We conclude $\exists_{x\in D}\ p(x)$ by existential generalization. To convince someone that $\forall_{x\in D}\ p(x)$, we must demonstrate that $p(x)$ is true for all $x \in D$. Unless your examples cover all of D, you cannot prove 'for all' by doing a few examples; only by covering all of D.† The typical way to convincingly demonstrate 'for all' is to assume that our adversary gives us $x \in D$, and we demonstrate that $p(x)$ must be true for that x. If this demonstration is generic, and applies to all elements of D, then we may conclude $\forall_{x\in D}\ p(x)$ by universal generalization.

†Do start by writing examples on your scratch paper, however; they often suggest important ideas for a proof.

4.1.5 Idioms and abbreviations

It is important to remember that two variables quantified over the same set may take on the same value. Each quantifier just tries setting its variable to each element of the set, one by one, completely decoupled from what other quantifiers are doing. This is good, because it allows us to look at each piece of a quantified statement independently. When we need pieces to be coupled, we add the coupling to either the logic or the quantification domains.

For example, to write a statement about a pair of different days $x, y \in D$, we need to ensure that the days are different. Consider two claims: "Pick

any two different days, then it rained on at least one of them," and "There are at least two days when it rained."

The first would use universal quantifiers with a conditional to ignore the cases where $x = y$. We can equivalently write $\forall_{x\in D}\, \forall_{y\in D}\, ((x \neq y) \to (p(x) \vee p(y)))$ or the contrapositive, $\forall_{x,y\in D}\, (\overline{p(x)} \wedge \overline{p(y)}) \to (x = y)$. The contrapositive is a common idiom saying, "If you think you found two days where it did not rain, then you actually found the same day twice." In both statements, the cases we want to ignore are exactly where the conditional is vacuously true. (If we omit the conditional, then we must also include the $x = y$ cases, so can simplify $\forall_{x,y}\, (p(x) \vee p(y)) \equiv \forall_{x\in D}\, p(x)$, which claims that it rained every day in D.)

The second would use existential quantifiers with 'and': $\exists_{x\in D}\, \exists_{y\in D}\, ((x \neq y) \wedge (p(x) \wedge p(y)))$. Notice that a conditional no longer works here: $\exists_{x,y\in D}\, ((x \neq y) \to (p(x) \wedge p(y))) \equiv (D \neq \emptyset)$, because if we are able to choose any $x \in D$, we can also choose $y = x$, then the conditional becomes vacuously true, making the whole statement true without even checking the weather.

Generalize this: We may have a Boolean *condition* (like $x \neq y$) that identifies cases where we want to apply a Boolean *test* $p(x, y)$. Here is what to remember. In a $\forall$ quantifier, we use *condition* $\to$ *test* because cases with *condition* = False are ignored when the "implies" is vacuously true. In an $\exists$ quantifier, we use *condition* $\wedge$ *test* because cases with *condition* = False are ignored when the "and" returns false. (These patterns still apply to nested quantifiers by working from the outside in, with one quantifier type at a time and seeing what variables are in scope, but that is perhaps best seen inside section 4.2's long example of using quantifiers to make statements about execution traces.)

Another way to write a statement about a pair $x \neq y$ is to break the symmetry between x and y: first choose x, then quantify y over a domain that has x removed. The next chapter defines set subtraction, $D \setminus \{x\} = \{d \mid d \in D \wedge d \neq x\}$, which lets us restate "Pick any two different days, and one of them it rained" as $\forall_{x\in D}\, \forall_{y\in D\setminus\{x\}}\, (p(x) \vee p(y))$, and "There are at least two days when it rained" as $\exists_{x\in D}\, \exists_{y\in D\setminus\{x\}}\, (p(x) \wedge p(y))$. The same change applies to $\forall$. Changing the domain makes it harder to reorder or negate the quantified statements, so move the conditions into the logic before doing that.

We often abbreviate by putting a condition in the quantifier, writing $\exists_{x\neq y}\, (p(x) \wedge p(y))$. If we are talking about pairs of distinct integers, $i, j \in [1..n]$ with $i < j$, we may even write $\forall_{1\leq i<j\leq n}\, r(i, j) \equiv \forall_{i,j\in[1..n]}\, ((i < j) \to r(i, j))$. Here n must be known, so it can be understood that i and j are the new integer variables being defined.

The 'there exists' quantifier makes it easy to formally write, "There is at least one x in set S satisfying $p(x)$," as $\exists_{x\in S}\, p(x)$. To formally write "exactly one" or "at most one" is less easy, but these expressions are common enough that you should memorize how to do this.

The common way to write, "There is at most one x in set D satisfying $p(x)$," essentially says, "If you think you have two elements of D satisfying p, then they are really the same element." Formally, $\forall_{x,y\in D}\ (p(x) \wedge p(y)) \rightarrow (x = y)$. Notice that this allows zero or one xs—if no $x \in S$ satisfies $p(x)$, then each 'implies' is trivially true, so the 'for all' is true. For practice, check your understanding of the negation, $\exists_{x,y\in S}\ p(x) \wedge p(y) \wedge (x \neq y)$, which asserts that you can find two different values (and possibly more) that satisfy p.

If we wish to write "There is a unique $x\ldots$" or "There is exactly one x in set D satisfying $p(x)$," we are making two separate claims: that there is an x with $p(x)$, and, if you think you've found a second, then it's actually the same x again. Formally, $\exists_{x\in D}\ \forall_{y\in D}\ p(x) \wedge (p(y) \rightarrow (x = y))$. Here the negation says that there are either none or at least two. $\forall_{x\in D}\ \exists_{y\in D}\ \overline{p(x)} \vee (p(y) \wedge (x \neq y))$.

Some use $\exists!_x$ for "there is exactly one x," but it doesn't negate easily, so I avoid it.

Expressions for "There is a unique $x\ldots$" or "There are at least two$\ldots$" are examples of *mathematical idioms*—phrases so common that we read their intent rather than the literal words or symbols. So 'hit the books' and 'learn by heart' the idiom for "there is a unique."

4.2 Event-time logic using quantifiers

Logic and quantifiers help us precisely state complex properties or requirements without the ambiguity of English. Here we'll use event-time logic to illustrate how to translate English statements that involve nested quantifiers, negations, and conditionals. We'll also see notational abbreviations that are common for longer expressions. Later examples include partitions, subsection 5.3.1, and big-O notation, subsection 6.3.1.

Consider a file system that has a set of files F that are accessed by a set of processes P. The system records its stream of *events* as an n-tuple, called a *trace*, $t = (t_1, t_2, \ldots, t_n)$. Each trace element records one of three types of events:

$$t_i = \begin{cases} a(p,f) & \text{process } p \textit{ accesses} \text{ file } f, \\ l(p,f) & \text{process } p \textit{ locks} \text{ file } f, \text{ or} \\ u(p,f) & \text{process } p \textit{ unlocks} \text{ file } f \text{ (undoes all previous } l(p,f) \text{ ops).} \end{cases}$$

(It will be equivalent to say "file f is accessed/locked/unlocked by process p.") The stream of events is a discrete view of actions over time. The idea is that if a processor locks a file, then it should have sole access to the file; no other process should lock, unlock, or access it.

What should we look for in a trace to verify that the system is behaving as it should? Let's convert the following nine statements into mathematical notation, quantifying over processes P, files F, and event times $[1..n]$. The first six statements are information we might want to know about a particular trace; the last three are properties that we would want every trace to have.

Q1. Every process accesses file f at some time. (Here, assume that a specific file f has already been chosen.)
Q2. File f is accessed by at least two different processes.
Q3. Some process accesses every file.
Q4. Every file is accessed by some process (maybe a different one for each file.)
Q5. Some process never locks a file.
Q6. There are no lock or unlock operations in the trace; every operation is an access.
Q7. Every file that a process locks is later unlocked by that process.
Q8. Any process that unlocks a file must have previously locked it, and not unlocked it in-between.
Q9. Any file that is locked by a process must be unlocked by that process before a different process accesses it.

A trace with $F = \{X, Y, Z\}$ and $P = \{1, 2\}$ recording 7 events:
$t_1 : l(1, X)$
$t_2 : a(2, Z)$
$t_3 : a(1, X)$
$t_4 : a(2, Y)$
$t_5 : u(1, X)$
$t_6 : a(2, X)$
$t_7 : a(1, Z)$
All true except Q6, and Q1,Q2 for $f = Y$.

Try translating each of these into quantified statements before you look at the answers below. Use the problem-solving steps from section 1.2; on scratch paper make simple examples of lock, unlock, and access sequences like the one in the margin, and decide whether the statements should be true or false for your examples.

A1. Every process accesses file f at some time.

This example has quantifiers at both the beginning and the end, which can lead to ambiguity. Do we mean:

(a) "Every process, at some time, accesses file f:
$\forall_{p \in P}\ \exists_{i \in [1..n]}\ t_i = a(p, f)$."
(b) "There is a time at which every process accesses file f:
$\exists_{i \in [1..n]}\ \forall_{p \in P}\ t_i = a(p, f)$."

Mathematically, these are different. For (a), our adversary challenges us by choosing a process p, then we must find a time i at which the trace has recorded p accessing file f; that is, $t_i = a(p, f)$. For (b), we first choose time i, then all processes must access the file f at that time. Since we know that each trace event records the action of a single process, the only way (b) can be true is if P contains less than two processes. In fact, we may be tempted to read the English statement in (b) as $\forall_{p \in P}\ \exists_{i \in [1..n]}\ t_i = a(p, f)$, once we recognize that the notation in (b) does not make sense. The lesson here is, since you can talk yourself into different interpretations of the English, you should always translate into unambiguous notation before you check equivalence or do negation.

A statement with nested quantifiers is like an onion, as adding parenthesis to (a) can show: $\forall_{p \in P}\ \left(\exists_{i \in [1..n]}\ (t_i = a(p, f))\right)$. Breaking (a) into separate

predicates reveals where values are chosen for the variables:

$$\begin{aligned} \alpha(i,p,f) &:= (t_i = a(p,f)) \\ \beta(p,f) &:= \exists_{i\in[1..n]}\ \alpha(i,p,f) \\ \gamma(f) &:= \forall_{p\in P}\ \beta(p,f) \end{aligned}$$

Reading from the bottom, the whole statement $\gamma(f)$ has f as a *free variable* and will be true or false depending on the specific file name that replaces f. Statement $\gamma(f)$ is true iff no choice of processor p makes $\beta(p,f)$ false. We can think of looping over all $p \in P$ and checking that $\beta(p,f)$ is always true, or of having our worst adversary choose the value of p.

When $\beta(p,f)$ is evaluated, specific values have been chosen for both p and f; $\beta(p,f)$ is true if we can find at least one time i where $\alpha(i,p,f)$ is true—that is, if trace event t_i is $a(p,f)$. If we fail to find any i, then $\beta(p,f)$ is false. Notice that the processor p is already chosen when we look for the time i, so different processors can (and will) have different times.

Reversing the order of universal and existential quantifiers changes the order of who makes choices. To see that (a) and (b) are different statements, let's translate the negation of (b) back into English:

$$\begin{aligned} &\nexists_{i\in[1..n]}\ \forall_{p\in P}\ (t_i = a(p,f)) \\ &\qquad \equiv \forall_{i\in[1..n]}\ \exists_{p\in P}\ (t_i \neq a(p,f)), \end{aligned}$$

which says, "For any time, we can choose a process that was not accessing file f at that time." This must be true if there is more than one process in P.

A2. File f is accessed by at least two different processes.

$$\exists_{p,q\in P}\ \exists_{i,j\in[1..n]}\ ((p \neq q) \wedge (t_i = a(p,f)) \wedge (t_j = a(q,f))).$$

Here $\exists_{p,q\in P}$ is an abbreviation for $\exists_{p\in P}\ \exists_{q\in P}$ or $\exists_{q\in P}\ \exists_{p\in P}$. Since all quantifiers are existential, it doesn't matter what order we choose values for the process and time variables. We can further abbreviate as $\exists_{p,q,i,j}$ if the quantification domains can be understood from context.

The predicate $(p \neq q)$ ensures that picking the same value for p and q never makes the statement true; the quantifier $\exists_{p,q}$ would otherwise allow that. Sometimes we write $\exists_{p\neq q}$ to force a choice of two unequal values.

Since the processes must be different and the trace cannot have two different processes in one trace entry, neither the English nor the notation needs to mention that the accesses are at different times. On the other hand, to say that file f is accessed at least twice (possibly by the same

process), we would make the times different, but allow the processes to be the same:

$$\exists_{p,q,i,j}\ ((i \neq j) \wedge (t_i = a(p,f)) \wedge (t_j = a(q,f))).$$

A3. Some process accesses every file.

To me, this says a single process accesses every file: $\exists_p\ \forall_f\ \exists_i\ (t_i = a(p,f))$, where the quantifiers are over processes P, files F, and times $[1..n]$. It can be argued, however, that the English statement actually means that every file is accessed by some process, considered next (A4).

A4. Every file is accessed by some process.

Here a different process may access each file: $\forall_f\ \exists_p\ \exists_i\ (t_i = a(p,f))$. Note that this is not equivalent to my expression in A3. because there are situations where A4 is true but A3 is false. (A4 is a necessary, but not sufficient condition for my A3 to be true.)

A5. Some process never locks a file.

There is a process that, for all files and times, does not lock:
$\exists_{p\in P}\ \forall_{f\in F}\ \forall_{i\in[1..n]}\ (t_i \neq l(p,f))$.
Equivalently, there is a process for which there does not exist a time and file that it locks: $\exists_p\ \overline{\exists_f\ \exists_i\ (t_i = l(p,f))}$.

A6. There are no lock or unlock operations in the trace; every operation is an access.

In the context of this problem, these statements mean the same thing, but they are two different logical expressions:

$$\forall_i\ \forall_p\ \forall_f\ ((t_i \neq l(p,f)) \wedge (t_i \neq u(p,f)))$$

is not equivalent to

$$\forall_i\ \exists_p\ \exists_f\ (t_i = a(p,f))$$

unless you add the information that each t_i is exactly one of the three operations applied by one process to one file; see Exercise 9.6. Imagine adding a new operation, $c(p,f)$, in which processor p checks that file f is available and not locked. Then it becomes possible for a trace to have no lock or unlock operations, but not have every operation be an access.

A7. Every file that a process locks is later unlocked by that process.

Here we quantify over every process, file, and time, and use a conditional to restrict our attention to those times of the relevant lock operations.

$$\forall_{p,f,i}\ \Big((t_i = l(p,f)) \rightarrow (\exists_{j\in(i,n]}\ t_j = u(p,f))\Big).$$

I restricted the quantification domain for j so t_j is after t_i; I can drop that restriction, and move the quantifier earlier, by including $\wedge(i < j)$ in the innermost parentheses and replacing the conditional $a \rightarrow \beta$ by the equivalent $\overline{a} \vee \beta$:

$$\forall_{p,f,i}\ \exists_j \Big(\big(t_i \neq l(p,f)\big) \vee \big((i < j) \wedge (t_j = u(p,f))\big)\Big).$$

Recall that in this system a processor's unlock operation removes *all* previous locks that this processor placed on that file. This behavior is important for this and the following answers; if lock and unlock operations had to be paired, like balanced parentheses, the expressions become much more complex. The next expression checks for unlocking files that are already unlocked, which may indicate processes that don't understand this system behavior.

A8. Any process that unlocks a file must have previously locked it, and not unlocked it in-between.

$$\forall_{p,f,k}\ (t_k = u(p,f)) \rightarrow \Big(\exists_{i\in[1,k)}\ (t_i = l(p,f)) \wedge (\forall_{j\in(i,k)}\ (t_j \neq u(p,f)))\Big).$$

I can use the commutative, associative, and distributive properties of 'and,' 'or,' and 'implies' to move all quantifiers to the front, obtaining:

$$\forall_{p,f,k}\ \exists_{i\in[1,k)}\ \forall_{j\in(i,k)}\ (t_k = u(p,f)) \rightarrow ((t_i = l(p,f)) \wedge (t_j \neq u(p,f))).$$

A9. Any file that is locked by a process must be unlocked by that process before a different process accesses it.

$$\forall_{f,p,q}\ \forall_{i,k\in[1,n]}\ \Big(((t_i = l(p,f)) \wedge (t_k = a(q,f)) \wedge (i < k) \wedge (p \neq q)) \rightarrow (\exists_{j\in(i,k)}\ (t_j = u(p,f)))\Big).$$

I would shorten this by moving some of the variable restrictions from the predicates to the quantification domains and abbreviating:

$$\forall_{f,p\neq q,i<k}\ ((t_i = l(p,f)) \wedge (t_k = a(q,f)) \rightarrow (\exists_{i<j<k}\ (t_j = u(p,f))).$$

4.3 Summary

Working with quantifiers is an important skill. In mathematics, all important theorems involve quantifiers as they talk of existence, non-existence, or properties of mathematical constructs. Even the innocent-looking claim that "$\sqrt{2}$ is irrational" is saying that there do not exist integers $p, q \neq 0$ for

which $(p/q)^2 = 2$. In computer science we make many quantified claims: that no set of simultaneous update commands can corrupt our database, that every network client will receive a response within the timeout value, or that, for each possible input, our computer program computes the right result or stops with an appropriate error message. The rest of the book uses quantified logic statements to define new structures and their operations precisely.

Because there are many ways to state quantifiers in English, it takes practice to become proficient at translating statements into notation that correctly pins down their meanings. One way to practice is to negate quantified formulae and translate back to English to see what becomes true if the original statement is false.

4.4 Exercises and Explorations

Quiz Prep 4.1. Write the following conditions in notation using quantifiers.

1. The largest element in a set S is x.
2. All the elements in the tuple $(a_1, a_2, \ldots, a_n)$ are distinct.
3. The element x occurs exactly once in tuple $(a_1, a_2, \ldots, a_n)$.
4. The tuple $(a_1, a_2, \ldots, a_n)$ is in increasing order.
5. The tuple $(a_1, a_2, \ldots, a_n)$ is in non-decreasing order.
6. The maximum value in $(a_1, a_2, \ldots, a_n)$ first occurs at position j.

Quiz Prep 4.2. Convert between quantified expressions and English statements. Let P be the set of all people, and D the set of dorms on campus. Define *lives in* and *dated* predicates, $l\colon P \times D \to \{T, F\}$ and $\delta\colon P \times P \to \{T, F\}$, so that $l(p, d)$ is true iff $p \in P$ lives in dorm $d \in D$ and $\delta(p, q)$ is true iff $p \in P$ has dated $q \in P$.

1. What are the English meanings of the following? (Note: $\nexists$ means $\neg\exists$)
   (a) ? $\exists_{p\in P}\ \forall_{q\in P}\ \overline{\delta(p,q)}$.
   (b) ? $\forall_{p\in P}\ \exists_{q\in P}\ \overline{\delta(p,q)}$.
   (c) ? $\nexists_{p\in P}\ \forall_{q\in P}\ \delta(p,q)$.
   (d) ? $\exists_{p\in P}\ \forall_{d\in D}\ \exists_{q\in P}\ (\delta(p,q) \wedge l(q,d))$.
   (e) ? $\forall_{c,d\in D}\ \exists_{x,y\in P}\ (l(x,c) \wedge l(y,d) \wedge \delta(x,y))$.
   (f) ? $\forall_{c,d\in D}\ \forall_{x,y\in P}\ (((c \neq d) \wedge l(x,c) \wedge l(y,d)) \to \overline{\delta(x,y)})$
   (g) ? $\forall_{d\in D}\ \exists_{p\in P}\ \forall_{q\in P}\ (\delta(p,q) \to l(q,d))$
   (h) ? $\forall_{d\in D}\ \exists_{p\in P}\ \forall_{q\in P}\ (l(p,d) \wedge (l(q,d) \to \delta(p,q)))$
2. What quantified statements represent these English sentences?
   (a) No-one has dated themselves.
   (b) No-one has dated anyone.
   (c) Someone has dated every resident from some dorm.
   (d) Someone has dated some resident from every dorm.
   (e) There is a dorm in which every resident has dated someone.
   (f) There is a dorm whose residents have not dated anyone from another dorm.

(g) There are two people who, between the two of them, have dated someone from every dorm.

(h) There are two dorms where no resident of either has dated a resident of the other.

Quiz Prep 4.3. Match the each of the following quantified statements with a logically equivalent statement on the right that uses only the "and" ($\wedge$) and negation (overline) operations. Assume x and y are quantified over integers and $R(x, y)$ is a predicate that is either T or F.

1. $\forall_x \forall_y (\overline{R(x,y)} \vee \overline{R(y,x)})$
2. $\overline{\forall_x \forall_y (R(x,y) \wedge R(y,x) \to (x = y))}$
3. $\overline{\forall_{x,y,z} (R(x,y) \wedge R(y,z) \to R(x,z))}$
4. $\overline{\exists_x \forall_y ((x \neq y) \to R(x,y))}$
5. $\forall_x \forall_y (R(x,y) \vee R(y,x))$

a. $\exists_x \exists_y (R(x,y) \wedge R(y,x) \wedge (x \neq y))$

b. $\overline{\exists_x \exists_y (\overline{R(x,y)} \wedge \overline{R(y,x)})}$

c. $\forall_x \overline{\exists_y (R(x,y) \wedge R(y,x))}$

d. $\exists_x \exists_y \exists_z (R(x,y) \wedge R(y,z) \wedge \overline{R(x,z)})$

e. $\exists_x \exists_y \exists_z (\overline{R(x,y) \wedge R(y,z)} \wedge R(x,z))$

f. $\forall_x \overline{\exists_y ((x = y) \wedge (R(x,y)))}$

g. $\forall_x \exists_y ((x \neq y) \wedge \overline{R(x,y)})$

Exercise 4.4. Find the mistake(s) in each of the following.*

*Warning: incorrect statements in this problem!

1. Given sets A and B with elements from the universe U, to show that $A \subseteq B$, we must show that $\exists_{x \in U} (x \in A) \wedge (x \in B)$.
2. Given sets A and B, to show that $A \subseteq B$, we must show that for all x, both $x \in A$ and $x \in B$.
3. Since $a \cdot 1 = a$ for all integers a, if we know $b \cdot m = b$, where b and m are integers, then $m = 1$.

▶

Exercise 4.5. Negate $\exists_{x \in \emptyset} \overline{p(x)}$ to argue that $\forall_{x \in \emptyset} p(x)$ should be defined as true, no matter what the predicate p. Note that $\emptyset$ represents the empty set—the quantification domain with no elements.

Puzzle 4.6. Jack Palmer's 1924 jazz lyric says, "Everybody loves my baby, but my baby don't love nobody but me."

1. If we think the double negative is for emphasis, we might say, "Everybody loves my baby, but my baby doesn't love anybody but me." Translate this into an expression quantified over a set P of people with my baby $b \in P$ and me $m \in P$. Denote a loves b by the predicate $\ell(a, b)$.
2. Show that this leads to the narcissistic conclusion, "I am my baby."

3. What genre of music best fits the original lyric?

4. How might you write the lyric to convey the intended meaning?

Extension 4.7. There is no general "distributive rule for quantifiers." In these questions, assume that all quantifiers are over the integers, and p and q are predicates.

1. Show that $(\forall_x\ p(x)) \wedge (\forall_y\ q(y)) \equiv \forall_x\ (p(x) \wedge q(x))$.

2. Show that $(\exists_x\ p(x)) \vee (\exists_y\ q(y)) \equiv \exists_x\ (p(x) \vee q(x))$.

3. Show that $(\forall_x\ p(x)) \vee (\forall_y\ q(y)) \rightarrow \forall_x\ (p(x) \vee q(x))$.

4. Give an example of predicates p, q for which $\forall_x\ (p(x) \vee q(x))$ is true but $(\forall_x\ p(x)) \vee (\forall_y\ q(y))$ is false.

5. Given x, show that $p(x) \vee (\forall_y\ q(y)) \equiv \forall_y\ (p(x) \vee q(y))$.

6. Show that $(\exists_x\ (p(x) \wedge q(x)) \rightarrow (\exists_x\ p(x)) \wedge (\exists_y\ q(y))$.

7. Give an example of predicates p, q for which $(\exists_x\ p(x)) \wedge (\exists_y\ q(y))$ is true, but $\exists_x\ (p(x) \wedge q(x))$ is false.

8. Given x, show that $(\exists_y\ (p(x) \wedge q(y)) \equiv p(x) \wedge (\exists_y\ q(y))$.

Chapter 5
Set Operations and Properties

The student's task in learning set theory is to steep him[or her]self in unfamiliar but essentially shallow generalities till they become so familiar that they can be used with almost no conscious effort. In other words, general set theory is pretty trivial stuff. . . read it, absorb it, and forget it.

—Paul Halmos, preface to *Naive Set Theory* [10]

With the notation of logic, we can precisely define and specify operations and properties of discrete structures. In this chapter we do this for sets, which are the most basic. We also begin to show how proof, which reasons from definitions, can establish properties that are consequences of the definitions.

Objectives: Above, Paul Halmos states the overall goal of this chapter: the student will learn these definitions and properties as base vocabulary for talking about discrete structures.

You will be able to use logic and the basic definition of element inclusion, $\in$, to define operations of union, intersection, complement, difference, symmetric difference, disjoint union, cardinality, power set, and Cartesian product, and their accompanying notation (common $|A|$, $\cup$, $\cap$, $\times$, $\overline{A}$, and less common $\setminus$, $\oplus$, $\uplus$, $\mathcal{P}(A)$). You will be able to recall, or to check from the definitions, which operations have common properties (associative, commutative, distributive, identity, idempotent, and so forth), and use these properties to simplify expressions, as in logic. You will also continue to see how proof can be used to establish properties by using logic to reason from the definitions to the desired conclusions. You will be able to count sets by inclusion/exclusion. Finally, you will be able to define and recognize a partition of a set.

5.1 Set operations

Here are the definitions of several set operations. Although this coverage is brief, it is important, and you should either commit to memory the definitions or the pictures that illustrate them so that you can reproduce the definitions if asked.

After the definitions we note several properties that they imply. (You are encouraged to generate new pictures to make sure that you understand and agree with the properties.) We then look at some simple examples of proving properties from the definitions. More is said about proof in chapter 9, but notice how the proofs given here proceed from what is known to provide new

information.

5.1.1 Definitions for set operations

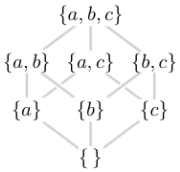
We've seen in subsection 3.1.1 the definition of a *set*—a

collection of elements from some universe U in no particular order—and the primitive test of whether an element is in a set. Logic lets us extend these to several set operations, which are defined in table 5.1 and illustrated with *Venn diagrams*. Each operation is a function (defined in the next chapter) that takes inputs and produces an output, so we write it's *type signature* using an arrow: inputs → outputs, not as a logic operation but consistent with how we will indicate domains and ranges for functions. We include counts as a check on understanding.

Some operations in the table, notably ⊆ and =, have both a definition and an equivalent statement. We can prove that these really are equivalent by applying the definitions and properties of logic.

Table 5.1: Summary of set operations, including diagrams, symbols, *names*, type signatures, and formal definitions.

	$a \in B$	*Set membership*: element, set → T/F. True iff a is in set B.
	$\lvert A \rvert$	*Cardinality*: set → $\{0, 1, 2, \ldots, \infty\}$. The number of elements in A, possibly ∞. E.g., $\lvert\emptyset\rvert = 0$, $\lvert\{a, b\}\rvert = 2$, and $\lvert\mathbb{Z}^+\rvert = \infty$.
	$A \subseteq B$	*Subset*: set,set → T/F. True iff $\forall_{x \in U}$ if $x \in A$, then $x \in B$. Equiv., $\forall_{x \in A}\ x \in B$; Every elt of A is in B. Note: if $A \subseteq B$, then $\lvert A \rvert \leq \lvert B \rvert$.
	$A = B$	*Set equality*: set,set → T/F. True iff $\forall_{x \in U}\ (x \in A) \leftrightarrow (x \in B)$. Equiv., $A \subseteq B$ and $B \subseteq A$.
	$A \cup B$	*Union*: set, set → set. The elements in either A or B: $A \cup B = \{x \mid (x \in A) \vee (x \in B)\}$.
	$A \cap B$	*Intersection*: set, set → set. The elements in both A and B: $A \cap B = \{x \mid (x \in A) \wedge (x \in B)\}$. Note: $\lvert A \rvert + \lvert B \rvert = \lvert A \cup B \rvert + \lvert A \cap B \rvert$.

	$A \setminus B$	*Set difference*: set, set → set. The elements in A that are not in B: $A \setminus B = \{x \mid (x \in A) \wedge (x \notin B)\}$. Sometimes written $A - B$ or $A \ominus B$. Note: $\lvert A \setminus B\rvert + \lvert A \cap B\rvert = \lvert A\rvert$.
	$\overline{A}$	*Complement*: set → set. All elements of U not in A: $\overline{A} = U \setminus A$. Note: $\lvert\overline{A}\rvert + \lvert A\rvert = \lvert U\rvert$.
	$A \oplus B$	*Symmetric difference*: set, set → set. The elements in A or B, but not both: $A \oplus B = \{x \mid (x \in A) \oplus (x \in B)\}$. Note: $\lvert A \oplus B\rvert + \lvert A \cap B\rvert = \lvert A \cup B\rvert$.
	$A \uplus B$	*Disjoint union*: set, set → set or error. if intersection $A \cap B$ is empty, return union $A \cup B$, otherwise 'error'. If no 'error', $\lvert A \uplus B\rvert = \lvert A\rvert + \lvert B\rvert$.
	$A \times B$	*Cartesian product*: set, set → set of pairs. The set of pairs of an element from A with one from B in all ways (see Sec. 3.1.2): $\{(a, b) \mid \forall a \in A\ b \in B\}$. Note $\lvert A \times B\rvert = \lvert A\rvert \cdot \lvert B\rvert$.
	$\mathcal{P}(A)$ or 2^A	*Power set*: set → family of sets. The collection of all subsets of A: $\mathcal{P}(A) = \{S \mid S \subseteq A\}$. E.g., $\mathcal{P}(\{a,b,c\}) = \{\emptyset,\{a\},\{b\},\{c\},\{b,c\},\{a,c\},\{a,b\},\{a,b,c\}\}$. Note: $\lvert\mathcal{P}(A)\rvert = 2^{\lvert A\rvert}$.
	$\binom{A}{k}$	*k-subsets*: set, integer → family of sets. The collection of all k-element subsets of A: $\binom{A}{k} = \{S \mid S \subseteq A \wedge \lvert S\rvert = k\}$. Note: $\lvert\binom{A}{k}\rvert = \binom{\lvert A\rvert}{k}$.

Since I am a computer scientist, I want to remind you to pay attention to the types in table 5.1. Many 'typo's (and 'thinko's) can be caught by realizing that an operation is being applied to a type that does not support it. Computer languages (like C++, Java, Python) use types* to help programmers catch many of their mistakes early. Physicists and engineers check units for the same purpose. When you are writing mathematics you should get in the habit of checking types: check that both operands for a union operation are sets, that the result of a subset operation is being treated as true or false, and so on.

*Strongly typed languages are checked mostly at compile time; others at run time. You should check your types at write-up time and rewrite time.

Be aware that sets can be elements themselves. Notice the difference between the types of the $\emptyset$ in $\emptyset \in A$ and in $\emptyset \subseteq A$; it is an element in the former, and a (sub)set in the latter. In fact, three of the four following expressions are

always true: for the one that is not, give an example set A for which it is true, and an example for which it is false. $\boxed{\emptyset \in A}$ $\boxed{\emptyset \subseteq A}$ $\boxed{\emptyset \in \mathcal{P}(A)}$ $\boxed{\emptyset \subseteq \mathcal{P}(A)}$

5.1.2 Properties for set operations

From the definitions of set operations we can derive many properties. As in logic, you don't need to memorize any of these properties because you can always go back to the definitions and re-derive them, but memorizing some can save work and allow for easier communication. The formal proofs for most properties are character-building (aka boring) exercises that simply recall corresponding properties of logic; we'll do some examples below.

Equality and subset: $A = B$ iff $(A \subseteq B)$ and $(B \subseteq A)$.
Commutativity: $A \cup B = B \cup A$ and $A \cap B = B \cap A$.
Associativity: $A \cup (B \cup C) = (A \cup B) \cup C$ and $A \cap (B \cap C) = (A \cap B) \cap C$.
Idempotence: $A = A \cup A = A \cap A$
Distribution: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ and $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.
Complement: $A = \overline{\overline{A}}$, $A \cup \overline{A} = U$, $A \cap \overline{A} = \emptyset$, $\overline{\emptyset} = U$, $\overline{U} = \emptyset$.
de Morgan's laws for sets: $\overline{A \cup B} = \overline{A} \cap \overline{B}$ and $\overline{A \cap B} = \overline{A} \cup \overline{B}$
Absorption and identity: $A \cup \emptyset = A$, $A \cap \emptyset = \emptyset$, $A \cup U = U$, and $A \cap U = A$.
$A \subseteq B$ iff $A \cap B = A$ iff $B \cup A = B$.

5.1.3 Proofs for set operations

As we mentioned in the introduction, the process of writing down the reasons a property is true is often of more value than the property itself—it can give deeper understanding of the underlying concepts and can make remembering the property easier.

When you are verifying or deriving properties, go back to the formal definition or to previously established properties to avoid circular reasoning. Don't rely on your intuitive understanding of the underlying concept; intuition can often guide you to the ideas to derive a property, but there is no substitute to going back to the definitions for a convincing demonstration.

In chapter 9 we go into more detail on proof formats and what is a proof, but let's already begin to look at informal demonstrations and formal proofs of some properties of set operations. I'll write the statements as *lemmas*[*]. Whenever you read a lemma, first try understand what it claims (what would be the opposite of the claim?), then whether you believe it (where do you become stuck if you try to build a counterexample?) Having done this, you may already have ideas that you could flesh out into a proof that would convince you. Then see if the given proofs are convincing. Does the proof supply too much detail or too little? The latter is common in written

[*]The Greek plural should be "lemmata."

proofs, partly due to lazy authors, but more due to the fact that different readers come with different knowledge and experience, and do not have direct interaction with the author to ask him or her to fill in gaps.

Lemma 5.1.1. *The empty set is a subset of every set:* $\emptyset \subseteq S$.

What is this claiming? That no matter what set S I am given, every element that is in the empty set can be found in S. That is what the definition of subset says.

What is the opposite? That someone can give me a set S such that some element b in the empty set is not in S.

But, of course, there are no elements of the empty set, so this is basically the proof. Let me write the proof in three ways: first, in a relatively formal *two-column* style, second, in paragraph style with definition details, and third, as a brief sketch of the key idea(s) that should convince a reader who knows the definitions well.

Two column, formal proof. The following are tautologies:

1. $\forall_{x \in U}\ x \notin \emptyset$	Defn of empty set
2. $\forall_{x \in U}\ ((x \in \emptyset) \rightarrow Q(x))$	Trivially true for any predicate $Q(x)$
3. $\forall_{S \subseteq U}\ \forall_{x \in U}\ ((x \in \emptyset) \rightarrow (x \in S))$	Substitute $Q(x) = (x \in S)$
4. $\forall_{S \subseteq U}\ \emptyset \subseteq S$	Defn $\subseteq$

This proves the lemma. ∎

1 introduced end-of-proof mark, QED.

Long narrative proof. The proof must work for all sets $S \subseteq U$, so assume that your worst adversary gives you S. We want to show that $\forall_{x \in U}\ ((x \in \emptyset) \rightarrow (x \in S))$, since this is the definition of subset.

But $\forall_x\ x \in \emptyset$ is false, because the empty set has no elements. So, $\forall_x$ the implication $((x \in \emptyset) \rightarrow (x \in S))$ is vacuously true, regardless of set S. ∎

Proof sketch. $\forall_x\ ((x \in \emptyset) \rightarrow (x \in S))$ is vacuously true. ∎

Ok, that one was too trivial to really need a proof once we understood the statement. (But notice how struggling to evaluate proofs clarified the statement!) Here is another:

Lemma 5.1.2. *Union preserves subsets: that is, for all sets A, B, C, if $A \subseteq B$ then $(A \cup C) \subseteq (B \cup C)$.*

What is the claim? If $A \subseteq B$, then adding the same elements to both sets doesn't destroy this property.

What would be the opposite? There is some $A \subseteq B$ and some C so that some element of $A \cup C$ is not in $B \cup C$.

Table 5.4: A truth table proving Lemma 5.1.2, that if $A \subseteq B$ then $(A \cup C) \subseteq (B \cup C)$.

$x\in A$	$x\in B$	$x\in C$	$((x\in A) \to (x\in B))$	$\to$	$\Big((x\in(A\cup C))\to(x\in(B\cup C))\Big)$		
T	T	T	T	T	T	T	T
F	T	T	T	T	T	T	T
T	F	T	F	T	T	T	T
F	F	T	T	T	T	T	T
T	T	F	T	T	T	T	T
F	T	F	T	T	F	T	T
T	F	F	F	T	T	F	F
F	F	F	T	T	F	T	F

Truth table proof. For a generic element of $x \in U$, the lemma is a tautology, as can be seen in table 5.4. ∎

Short proof. For a generic $x \in (A \cup C)$, we have $(x \in A) \vee (x \in C)$. But if $x \in A$ then $x \in B$, so we know that $(x \in B) \vee (x \in C)$, which is the definition of $x \in (B \cup C)$. ∎

If we attempt to prove the converse of the union property—for all sets A, B, C, if $(A \cup C) \subseteq (B \cup C)$ then $A \subseteq B$—then we better fail, since this is false. Stop and try to create an example of sets A, B, and C for which the converse is not true. What is an example in which each set has as few elements as possible?

The truth table for the converse is not a tautology; when $x \in A$, $x \notin B$, and $x \in C$, you find that $\Big((x\in(A\cup C)) \to (x\in(B\cup C))\Big) \to ((x \in A) \to (x \in B))$ is false. This attempt at a narrative proof finds the same type of *counterexample*:

Proof. Suppose we know that $(A \cup C) \subseteq (B \cup C)$. That is, for all $x \in U$, if $x \in (A \cup C)$ then $x \in (B \cup C)$, or equivalently, either $x \notin (A \cup C)$ or $x \in (B \cup C)$. Then every x is in one of three places: outside both A and C, inside B, or inside C.

We want to show that every x is either outside of A or inside B. Thus, a counterexample would have x in A and outside of B. This means that x would have to be inside C. And now it is easy to build such an example: a minimum one is $A = \{1\}$, $B = \emptyset$, and $C = \{1\}$. One counterexample is enough to show that a "for all" statement is false. ∎

Let's briefly prove one more important lemma: Two sets are equal iff each is a subset of the other.

Lemma 5.1.3. *$A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.*

Proof. We could do the two directions ('if' and 'only if') separately, but we don't have to since the definitions of $=$, $\leftrightarrow$, $\forall$ and $\subseteq$ give the following chain* of logically equivalent statements:

*Each "iff" connects to the **right** side of the line above, which is why we don't repeat $A = B$ on the left side

$$\begin{aligned} A = B \text{ iff } & \forall_x\ (x \in A \leftrightarrow x \in B) && \text{Defn } = \\ \text{iff } & \forall_x\ (x \in A \to x \in B) \wedge (x \in B \to x \in A) && \text{Defn } \leftrightarrow \\ \text{iff } & \forall_x\ (x \in A \to x \in B) \text{ and } \forall_y\ (y \in B \to y \in A) && \text{Defn } \forall \text{ as } \wedge \\ \text{iff } & A \subseteq B \text{ and } B \subseteq A. && \text{Defn } \subseteq \end{aligned}$$

∎

5.2 Inclusion/exclusion counting

The sum rule said that if finite sets A and B are *disjoint*, meaning $(A \cap B) = \emptyset$, then we can count the elements in the union by counting the sets separately: $|A \uplus B| = |A \cup B| = |A| + |B|$. But what if the sets are not disjoint? Then $|A| + |B|$ overcounts by counting every element of the intersection two times, but we can correct for that by subtracting off one of those counts:

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

We can extend this to three finite sets. To count the total in the union, we can add up the elements in each set, then subtract off the elements common to a pair of sets. Elements in all three sets will have been added three times, then subtracted three times, so we must put them back again.

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |B \cap C| - |A \cap C| + |A \cap B \cap C|.$$

*Proof can use Equation 3.1.

These are called inclusion/exclusion rules.*

Let's do a particular example where we can count the union and want to know the intersection. The rules for choosing computer account passwords at UNC Chapel Hill require at least 8 characters from a menu of 95 choices:

A. At least one is a letter (upper or lower case, so 52 choices per character).
B. At least one is a digit (10 choices per character).
C. At least one is a punctuation symbol from a list of 16, `!@#$%&*+={}?<>"'`. If double quote is used, it must be last.
D. May also contain any of 17 special symbols, `\/()[].,;:|^_-~` ` and space. Space may not be first or last, and \ may not be last.

Let's count 8 character passwords. First, ignore the "At least" rules in A-C, but obey all restrictions on special symbols. The first and last character have 93 choices (no quote for first, no \ for last, and no space for either), and the rest have 94 (no quote) so we start with set of

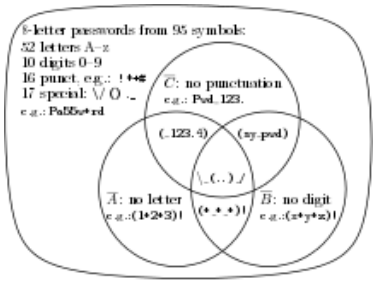


Figure 5.1: From all 8-letter passwords that obey the rules for special symbols, we must take out those that violate any of A-C. Inclusion/exclusion makes sure that

$93^2 94^6$ passwords, and want to take out those that violate A, B, or C. We can subtract from this total the numbers of passwords with no letters ($41^2 42^6$ violate A), with no digits ($83^2 84^6$ violate B), and no punctuation ($78 \cdot 79^6 \cdot 77$ violate C, since no \ first, and no space first or last). But then we've subtracted some passwords more than once, so we add back in the numbers of passwords with neither letters nor digits ($31^2 32^6$ violate both A and B), those with neither letters nor punctuation ($26 \cdot 27^6 \cdot 25$ violate A and C), and those with neither digits nor punctuation ($68 \cdot 69^6 \cdot 67$ violate B and C). Finally, the passwords with only special symbols (no letters, digits, or punctuation) were initially counted, then subtracted three times and added back three times, so we need to subtract their number ($16 \cdot 17^6 \cdot 15$). This gives the number of permitted 8-character passwords.[†]

[†] Going from 8 to 10 characters gives 11,227 times more passwords. The rules allow 43% of all 8 character strings, and 54% of all 10 character strings.

$$\begin{aligned} 93^2 \cdot 94^6 - 41^2 \cdot 42^6 - 83^2 \cdot 84^6 - 78 \cdot 79^6 \cdot 77 \\ + 31^2 \cdot 32^6 + 26 \cdot 27^6 \cdot 25 + 68 \cdot 69^6 \cdot 67 - 16 \cdot 17^6 \cdot 15 \\ = 2,570,334,323,521,120 \end{aligned}$$

Another example arises in the next chapter: counting surjections.

5.3 Families of sets

Not everything about sets is easy, especially when we begin to look at *families of sets*, which are sets whose elements are sets.

5.3.1 Partitions

One example of a useful family is a *partition of a set S*, which is a collection of non-empty subsets of S so that every element of S is in exactly one subset. Simple examples include the partition of integers into the even and odd subsets, or the partition into $P_k = \{j \mid j \in \mathbb{Z} \wedge j^2 = k^2\}$, for $k \in \mathbb{N}$.

The notation for partitions is not easy to grasp due to alternation of quantifiers: $\Pi = \{P_1, P_2, \ldots, P_n, \ldots\}$ is a partition of S iff $\emptyset \notin \Pi$, $\forall_{P \in \Pi}\ P \subseteq S$ and

$$\forall_{s \in S}\ \exists_{P \in \Pi}\ \forall_{Q \in \Pi}\ s \in P \wedge (s \in Q \to P = Q).$$

Using a big operator for disjoint union, which produces an error if any element is shared by any two sets, we can write this more concisely: $\Pi = \{P_1, P_2, \ldots, P_n\}$ is a partition of S iff $\emptyset \notin \Pi$ and $S = \biguplus_{P \in \Pi} P$. We'll derive a formula to count partitions in subsection 8.2.2.

5.3.2 Further questions

Can a set contain a copy of itself? This leads to an infinite regress and to statements that we can write in notation but that do not have a valid interpretation. Consider the family A of the sets that do not contain themselves: $A = \{X \mid X \text{ is a set and } X \notin X\}$

Is $A \in A$? If not, then the rule says it must be, and if so, then the rule says it can't be. The self-reference here has no valid interpretation; we avoid defining such sets in this book.

There are also statements that do have valid interpretations, but we do not yet know whether they are true or false. A family of sets $A = \{A_1, A_2, \ldots, A_k\}$ is *closed under union* iff $\forall_{1 \le i,j \le k}\ A_i \cup A_j \in A$. The power set $A = \mathcal{P}(S)$ is closed under union, with every element of S in exactly half of the sets of A. A smaller example is

$$A = \{\emptyset, \{b\}, \{a, b\}, \{a, c\}, \{a, b, c\}\}.$$

The *Union Closed Conjecture*, originally stated by Frankl in 1979, is *open*, meaning that the answer is not known, though there has been progress since 2020.

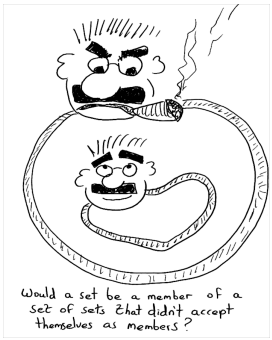


Figure 5.3: From bligblug, ©by Tom Canel. (used by permission)

Conjecture 5.3.1. *If A is a union closed family of sets, then there exists an element x that is contained in at least half of the sets of the family: $\exists_{x \in U}\ |\{i \mid x \in A_i\}| \ge |A|/2$.*

In discrete mathematics there are many puzzles like this that are easy to state and hard to solve. Even for unsolved puzzles, one can always begin by looking at special cases: e.g., if there is some set in A of size one, two, or three, then it is not too hard to show the conjecture holds true.

5.4 Summary

From simple definitions come a plethora of operations and properties that form the basic vocabulary for defining structures in computer science and mathematics. Learn the definitions (or remember the pictures that suggest the definitions) and you will be able to derive the properties that you need. Then learn the terms for the properties that you use most so you can communicate easily with others. After that you won't refer back to the definitions much, because they will have become second nature to you.

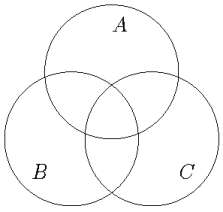
Using sets we were able to precisely state the rules for counting, although the informal rules are often sufficient for back-of-the-envelope checking.

Even discrete structures as simple as sets have puzzling questions that remain unsolved.

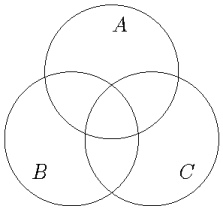
Figure 5.2: *Genji-mon* are the traditional crests marking the 54 chapters of the 11th century Japanese novel, "Tale of Genji." The central squiggle in this photo from fujiarts.com marks the final chapter; all other Genji-mon are possible answers for an incense game in which guests must identify, in a sequence of five scents, which scents repeat.
Starting at the upper right, as the Japanese would traditionally do, the first marker indicates a play of the game with three scents, which induce a partition $\{\{1,4,5\},\{2\},\{3\}\}$, abbreviated $145|2|3$. In other words, the first scent is reused as the 4th and 5th. What other partitions has the artist represented? (Mouse over for answers in pdf.)
The 52 partitions of $\{5,4,3,2,1\}$, plus the squiggle, are not enough for 54 Genji-mon; two Genji-mon must represent the same partition. (This claim depends on the pigeonhole principle, stated in the next chapter.) Can you find them in the list linked here? ?

5.5 Exercises and Explorations

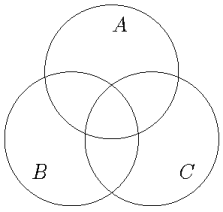
Quiz Prep 5.1. Shade areas to make Venn diagrams for these sets:



$(A \cap B) \cup C$



$(A \cap B) - C$



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

Quiz Prep 5.2. Answer the following:

1. In the universe of the real numbers, what is *complement* of the open interval from a to b:

$$\overline{(a,b)} = \{\quad\}.$$

2. Write the power set (set of all subsets) for each set

$A = \{a, b\}$ $\quad \mathcal{P}(A) = \{\quad\}$
$B = \{a, \emptyset\}$ $\quad \mathcal{P}(B) = \{\quad\}$
$C = \{\{a, b, c\}\}$ $\quad \mathcal{P}(C) =$

3. Find a set A for which $A \cap \mathcal{P}(A) \neq \emptyset$. ?

4. A group of six people ordered sandwiches. Two wanted no lettuce and three wanted no tomatoes. If two wanted everything (including both lettuce and tomatoes), how many wanted neither? ?
   Two people then decided they didn't want mayonnaise, which happened to make each sandwich unique. Of the eight subsets of $\{T, L, M\}$, which two were not ordered? ?

Hint:

Exercise 5.3. Draw Venn diagrams to illustrate properties from subsection 5.1.2.

Exercise 5.4. Here are two statements that are true for a family of sets $\mathcal{A}$ that partitions a set S. They are very close to the definition, but are different, because they are also true of some families that are not partitions. For each, give an example of a set S of integers and a family that is not a partition of S for which the statement remains true.

1. Every element $x \in S$ is in exactly one set of the family $\mathcal{A}$. In notation,

$$\forall_{x \in S} \left(\exists_{B \in \mathcal{A}} \left(\forall_{C \in \mathcal{A}} \left(x \in B \right) \wedge \left(x \in C \rightarrow B = C \right) \right) \right).$$

2. Element x is in S if and only if it is in exactly one set of the family $\mathcal{A}$:

$$\forall_{x\in U}\ \Big(x\in S \leftrightarrow (\exists_{B\in\mathcal{A}}\ \forall_{C\in\mathcal{A}}\ \ (x\in B)\wedge(x\in C\rightarrow B{=}C))\Big).$$

▶

Puzzle 5.5. Updating a Charles Dodgson (Lewis Carroll) puzzle: in a particularly aggressive paintball game with 20 combatants, 85% got hit in a leg, 80% in an arm, 75% upside the head, and 70% in the facemask. What is the minimum number of combatants that were hit in all four places? ? Maximum? ?

▶

Puzzle 5.6. On Settled Island, every native is either a "knight," who always tells the truth, or a "knave," who always lies. Natives of Settled know all about sets, which they denote $\{PQR\ldots\}$ (without commas). When counting, however, they primarily distingush even $\{0,2,4,\ldots\}$ and odd $\{1,3,5,\ldots\}$ numbers.

You meet 5 natives, A–E who made the following statements:

A said: In the set $\{B\}$ there is an even number of knights.
B said: In the set $\{C\}$ there is an even number of knights.
C said: In the set $\{D,E\}$ there is an odd number of knights.
D said: In the set $\{A,C\}$ there is an even number of knights.
E said: In the set $\{B,C\}$ there is an odd number of knights.

Who are the knights and who are the knaves?

Exploration 5.7. Suppose that three sets, A, B and C, satisfy two conditions, $(A\cup C)\subseteq(A\cup B)$ and $(A\cap C)\subseteq(A\cap B)$. Demonstrate, however you wish, that $C\subseteq B$.

Chapter 6
Relations and Functions

Mathematicians are like Frenchmen: whatever you say to them, they translate it into their own language, and at once it means something completely different.
—Johann Wolfgang von Goethe

You have been introduced to many functions in your math classes, like $\sin(x)$ and $f(x) = x^2$. What functions have in common is that they assign, or *map*, each input value from some domain set to a unique output value from a range set.

By using quantified statements about sets of tuples, we capture the familiar concept of a *function* in a precise manner using the abstract concept of a *relation*. We also define special types of functions and relations. These formal, sometimes abstract, definitions perfectly illustrate the Goethe epigram above, because they supersede our prior ideas of functions and become the authority on what is and is not a function.

Objectives: By working through this chapter and the exercises, you will be able to define relations from Cartesian product, and identify whether example relations are binary, reflexive, symmetric, or transitive. You will be able to define functions, to write notation for function definition with domain and range, $f : A \to B$, and function use, $f(a)$, and to identify whether functions are predicates, injections, surjections, or bijections. You will be able to count the number of possible functions and relations of different types, and begin to see how proof can formally show that certain functions satisfy the definitions. You will learn the pigeonhole principle, which is about the existence of bijections. Finally, you will see how quantified statements in the definitions of big O, big Ω, and big Θ characterize the growth of functions, such as bounds on the resources required by algorithms.

6.1 Relations

The abstract concept of a relation appears throughout computer science (from relational databases to entity-relationship diagrams) so it is a little surprising that the definition is a simple statement about sets.

A *relation* on two sets, A and B, is a subset of pairs $A \times B$; a relation $R \subseteq A \times B$. When $B = A$, we have a *binary relation*, $R \subseteq A^2$. A first example relation is "less than" on integers, $R = \{(m, n) \mid (m, n \in \mathbb{Z}) \wedge (m < n)\}$. Pair $(1, 2) \in R$, but $(2, 2) \notin R$; which we can abbreviate as $1\ R\ 2$, but $2 \not R\ 2$.* A

*Just as we write $1 < 2$ and $2 \not< 2$.

second example is the squaring relation $S = \{(x, x^2) \mid x \in \mathbb{R}\}$, which contains pairs $(2, 4)$ and $(-2, 4)$, but not $(2, -4)$ or $(2, 2)$. Non-numeric examples of relations on people include "is a child of," "is in the same class as," or "has sent email to." A non-binary relation on people and vehicles is "owns," and on students and numbers is "has a current GPA of."

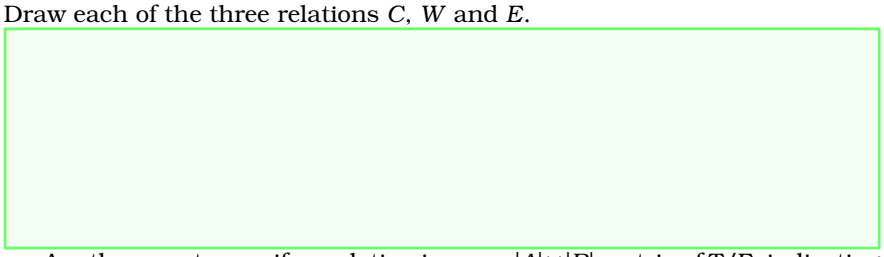
To specify a relation, we can simply list its set of pairs. For example, let's represent a set of seven people (alice, bob, chris, dana, eve, fern, and gary) by their lowercase initials: $P = \{a, b, \ldots, g\}$. Define the child-of ($C \subseteq P \times P$) and class-with ($W \subseteq P \times P$) relations:

$$C = \{(c, a), (c, b), (d, a), (d, b), (f, d), (f, e), (g, c)\}$$
$$W = \{(a, a), (a, e), (b, b), (b, g), (c, c), (d, d), (e, a), (e, e), (f, f), (g, b), (g, g)\}.$$

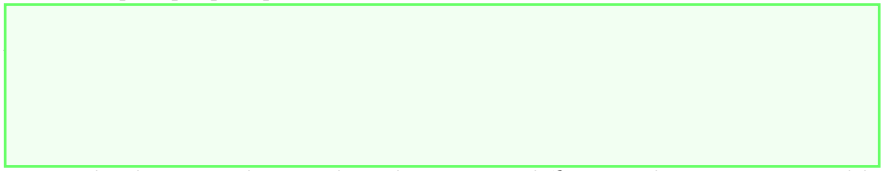
For the emails relation, $E \subseteq P \times P$, Grandma Alice broadcasts to everyone, every time, Grandpa Bob never emails, Chris and Dana exchange business email, cc'ed to themselves, Dana and Eve also email each other and their child, and the young ones, Fern and Gary, have each emailed everyone else, but neither emails themselves.

We can draw a relation $R \subseteq A \times B$ by making a dot or *vertex* for each element of A and B, and drawing an *edge* as a straight or curved arrow for each $(a, b) \in R$.* chapter 13 explores these diagrams, known as *graphs*. Draw each of the three relations C, W and E.

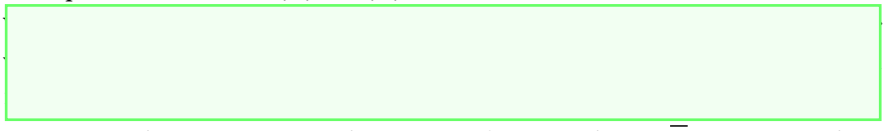
*An arrow for an edge $(a, a) \in R$ is called a *self-loop*.



Another way to specify a relation is as an $|A| \times |B|$ matrix of T/F, indicating which pairs are in the relation. Create the matrices for the relations $=$, $<$, and $\geq$ on $[0..2] \times [0..3]$.



To check our understanding, how many different relations are possible for a pair of sets of size $|A|$ and $|B|$? ?



Every relation $R \subseteq A \times B$ has a *complement* relation $\overline{R} \subseteq A \times B$ and an *inverse* relation $R^{-1} \subseteq B \times A$:

Complement: $\overline{R}$ consists of all pairs from $A \times B$ that are not in R: $\overline{R} = \{(x, y) \mid (x \in A) \wedge (y \in B) \wedge ((x, y) \notin R)\}$. The complement of the "sends email to" relation is the "does not send email to" relation. The complement of the "less than" relation on integers, $<$, is the "greater than or equal to" relation on integers, $\geq$. Complementing the relation complements the set of arrows in the corresponding drawing, and the Boolean entries in the matrix.

Inverse: R^{-1} is obtained by simply swapping the order of every pair: $R^{-1} = \{(y, x) \mid (x, y) \in R\}$. As you would expect, the inverse of the "sends email to" relation is the "receives email from" relation. The inverse of the "less than" relation on integers, $<$, is the "greater than" relation on integers, $>$. Going from a relation to its inverse reverses the direction of every arrow in the corresponding drawing, and transposes the matrix.

6.1.1 Binary relations

Many special properties of relations are defined by quantified statements, so I want to make some of those definitions here to practice with quantifiers and prepare for defining functions. We won't see the full power of these special relations until chapter 12.

Here are five properties for binary relation $R \subseteq A \times A$. It is good practice to write the negation for each. How could you recognize each property from a drawing of a relation?

Reflexive: Binary relation R is *reflexive* iff $\forall_{x \in A}\ (x\ R\ x)$.

Irreflexive: R is *irreflexive* iff $\forall_{x \in A}\ (x \not R\ x)$.

Symmetric: R is *symmetric* iff $\forall_{x,y \in A}\ (x\ R\ y \to y\ R\ x)$.

Antisym: R is *antisymmetric* iff $\forall_{x \neq y \in A}\ (x\ R\ y \to y \not R\ x)$.

Transitive: R is *transitive* iff $\forall_{x,y,z \in A}\ \Big(((x\ R\ y) \wedge (y\ R\ z)) \to (x\ R\ z)\Big)$.

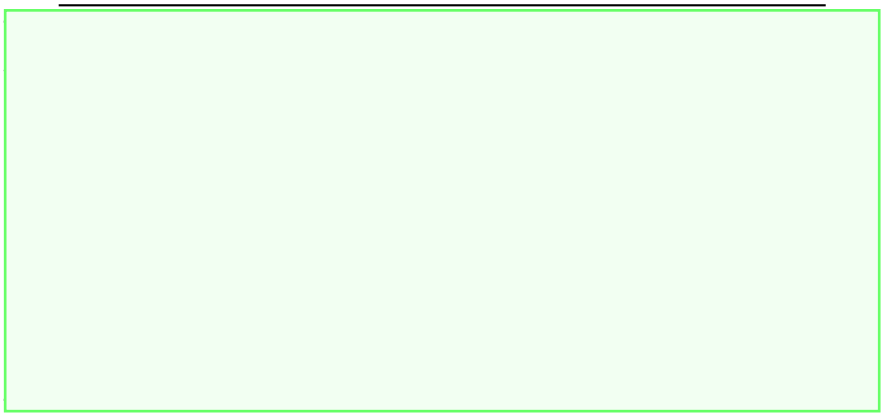
Fill in table 6.1 to identify the properties for the example binary relations from above: relations on integers ($=$, $<$, $\neq$, $\geq$), the subset relation on sets ($\subseteq$), and the relations on people (C, W, E).

Notice that reflexive and irreflexive are not opposites. Although it is true that no relation can be both reflexive and irreflexive, there are relations like E that are neither. In fact, if $n = |A|$, then of the 2^{n^2} binary relations, 2^{n^2-n} are reflexive and 2^{n^2-n} are irreflexive,* so as n grows, the fraction of all binary relations that are reflexive or irreflexive becomes vanishingly small.

*Can you explain these counts? See quizprep 6.5.

Table 6.1: We've defined eight binary relations above: on integers, we have $=$, $\neq$, $<$, and $\geq$, on sets of integers we have $\subseteq$, and on people we have child-of, class-with, and emails. For each, write two pairs that are in the relation and two pairs not in the relation. Then check off the properties each relation has. The first is done as an example (I'm using T and "." for contrast; answers can be revealed below.

	$=$	$\neq$	$<$	$\geq$	$\subseteq$	C	W	E
Pairs in	(0,0)	(,)	(,)	(,)	(,)	(,)	(,)	(,)
relation	(1,1)							
Pairs not	(0,1)							
in relation	(2,0)							
Reflexive	T							
Symmetric	T							
Transitive	T							
Irreflexive	·							
Antisym	T							



The same holds for symmetric and anti-symmetric: there are relations that are neither. There are $2^{n(n+1)/2}$ symmetric relations because when we choose, for each $a \le b \in A$ whether (a, b) is in the relation, we are making the same choice for (b, a). There are a few more choices for anti-symmetric relations: For the n repeat pairs (a, a), we independently choose in or out. For the pairs with $a \le b \in A$, we independently choose to include (a, b), (b, a), or neither; we cannot choose both. Thus, there are $2^n \cdot 3^{n(n-1)/2}$ anti-symmetric relations.

By the way, counting transitive relations is hard; there is no closed form expression. An expression is said to be in *closed form* if it can be calculated using a fixed number of mathematical operations on input values (addition, multiplication, exponentiation, factorial, choose). A summation is not considered closed form, because the number of inputs and additions in a sum can be arbitrarily large.

Relations with similar properties behave somewhat similarly. For example in table 6.1, both relations = and W (class-with) in are reflexive, symmetric, and transitive. Relations like these, called *equivalence relations*, always partition their sets into *equivalence classes*, as we will see in section 12.3.

Also in the table, $\subseteq$ is similar to $\ge$. Relations that are anti-symmetric and transitive, and either reflexive or irreflexive, are *partial orders*, as we will see in section 12.2. Relations $\ge$ and $<$ are *total orders* because $\forall_{a \ne b}$, either (a, b) or (b, a) is in the relation, but not both. On the other hand, we can find a pair of sets A, B with neither $A \subseteq B$ nor $B \subseteq A$, so $\subseteq$ is not a total order.

6.2 Functions

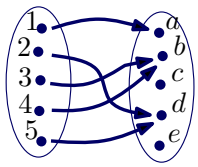
A *function* f is a relation on A and B that pairs, or *maps*, each element a from the *domain* A to exactly one element $b = f(a)$ from the *range* B. For example, the predicates used in quantified statements are functions whose range set is $\{T, F\}$, since they map each input to *true* or *false*.

Old mathematicians never die; they just lose their functions.

In notation, relation $f \subseteq A \times B$ is a *function* iff

$$\forall_{a \in A}\ \exists_{b_1 \in B}\ ((a, b_1) \in f) \wedge (\forall_{b_2 \in B}\ ((a, b_2) \in f \rightarrow b_2 = b_1)).$$

In English, this says that every element of the domain $a \in A$ is paired with an element of the range $b_1 \in B$, and if we think it is paired with a second element $b_2 \in B$, then we have actually rediscovered the first, because $b_2 = b_1$. (This is the idiom for "there is a unique $b \in B$" that was introduced in subsection 4.1.5.)



function

When this definition is satisfied, we change from relation notation to the familiar function notation: writing $f : A \rightarrow B$ to identify the domain and range, and writing $f(a)$ for the unique value that is paired with a. Mixing the

notation, $\forall_{a \in A,}\ (a, f(a)) \in f$ and $\forall_{a \in A, b \in B}\ ((a, b) \in f \to (b = f(a)))$. We depict a function by drawing a single arrow from each element of the domain to some element of the range.

How many functions from A to B are possible? For each element of A we choose exactly one element of B, so we have ☐ functions by the product rule.

We extend the $f(\,)$ notation to tuples for functions of more than one variable: if the domain of a function is a set of tuples, we suppress an extra set of parentheses. For example, the Pythagorean theorem gives the length of a vector (x, y), which is a function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ that we write as $f(x, y) = \sqrt{x^2 + y^2}$, suppressing the extra parentheses of $f((x, y))$.

We also extend the $f(\,)$ notation to sets. For a subset of the domain, $X \subseteq A$, we get a subset of the range, $f(X) = \{f(x) \mid x \in X\}$, called the *image* of X under f. In the function drawn in the margin, $f(\{2, 3, 4\}) = \{b, d\}$. In mathematical texts, B is called the *co-domain*, and the term *range* is reserved for the image of the domain, $f(A)$, which need not cover the entire co-domain, B. In computer science we tend to use *range* to mean *co-domain*.

In the other direction, the *pre-image* of $Y \subseteq B$ under f is a subset of the domain: $f^{-1}(Y) = \{a \in A \mid f(a) \in Y\}$. In the function in the margin $f^{-1}(\{a, b\}) = \{1, 3, 4\}$ and $f^{-1}(\{c, e\}) = \emptyset$. Using the notation f^{-1} to define pre-image does not imply that the inverse of f is a function, but it is consistent with the definition of inverse of a relation. (This type of offhand remark is what makes reading mathematics slow. To get it, you must disentangle the related but different concepts of 'inverse' for relations and for functions. If you do, feel proud. If not, ask.)

We can compose functions whose domain and range match to make a new function: given $f : A \to B$ and $g : B \to C$, the *composition* $h = g \circ f$ is a function from A to C defined so $h(a) = g(f(a))$. Notice that composition applies the functions from the side closest to the parameter first.

6.2.1 Some numerical functions

The operations on sets in the previous chapter are functions that apply to sets of any type of elements. Here are six functions on sets of numbers. The first three are defined with nested quantifiers (using a mathematical idiom for "smallest") before they are restated in words. The remaining three are defined from logic notation.

min For a set or tuple A, if $(x \in A) \wedge (\forall_{a \in A}\ a \geq x)$, then $\min(A) = x$, otherwise $\min(A)$ is undefined. In words, $\min(A)$ is the smallest value among the elements in A, if a smallest exists. For a finite set or tuple, $\min(A)$ is always defined, but infinite sets like $\mathbb{Z}$ or the real interval $(0, 1)$ need not contain a minimum element.

liminf For a set $A \subseteq \mathbb{R}$ define $\liminf(A) = x$, for $x \in \mathbb{R}$ iff $\forall_{a \in A} \Big((x \leq a) \wedge (\forall_{z > x \in \mathbb{R}} \exists_{b \in A} (x \leq b < z)) \Big)$. The value of $\liminf(A)$ need not be in A, but no greater number is less than or equal to all elements of A. For finite sets or tuples, $\liminf(A) = \min(A)$.

argmin For a tuple $A = (a_1, \ldots, a_n)$, $\arg\min(A) = i$ iff $\forall_{j \in [1..n]} (a_i < a_j) \vee ((i \leq j) \wedge (a_i \leq a_j))$. In words, $\arg\min(A)$ is the index of the first occurrence of a minimum element in tuple A.

Iverson bracket An expression in brackets, like $[x > 0]$, evaluates to '1' if the expression is true and '0' if the expression is false. In this example, $[x > 0]$ is a function from reals to $\{0, 1\}$.

sgn: signum For all $x \in \mathbb{R}$, $\operatorname{sgn} x = [x > 0] - [x < 0]$ returns the sign of the number (0 for zero).

absolute value For all $x \in \mathbb{R}$, the *absolute value*, $|x| = \operatorname{sgn}(x)x$, changes the sign if and only if x is negative.

In your algebra and calculus classes, continuity is an important property and most functions are continuous,* at least over most of their domain. In discrete mathematics, we don't even need to define continuous and many of our most important functions here and in the next chapter are not continuous.

*Topology has a succinct definition of continuous: the pre-image of every open set is open.

Is square root a function? If we look at the definition of function, we see that the question is incomplete, because we need to identify the domain and range sets. What we choose makes a difference:

- $\sqrt{x}\colon \mathbb{R} \to \mathbb{R}$ is not a function, because $\sqrt{-1}$ is not defined among the real numbers. (Proof: $(\sqrt{-1})^2 = -1$, but the square of any real number is non-negative. It was to avoid this difficulty that complex numbers were invented in the 16th century.)
- $\sqrt{x} \subseteq \mathbb{R}^{\geq 0} \times \mathbb{R}$, the inverse relation of squaring, is also not a function. The squaring relation $S = \{(x, x^2) \mid x \in \mathbb{R}\}$ is a function with range $\mathbb{R}^{\geq 0}$. Its inverse relation $S^{-1} = \{(y, z) \mid z^2 = y\}$, however, is not a function because both $(4, 2) \in S^{-1}$ and $(4, -2) \in S^{-1}$.
- $\sqrt{x}\colon \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ is a function that restricts the range to the non-negative branch, giving what is called the *principal square root* of x. This is how $\sqrt{x}$ is implemented in many calculators and computer languages. It is no longer the inverse of squaring reals, but is the inverse for non-negative reals, which, for example, is enough for the Pythagorean theorem.
- $\sqrt{x}\colon \mathbb{R}^{\geq 0} \to \mathbb{R}^2$ can also be a function if we define $\sqrt{x} = (-|y|, |y|)$ whenever $y^2 = x$, returning a unique pair. Note that if we defined $\sqrt{x} = (-y, y)$ we would not have a function, because $\sqrt{4}$ would need to map to two different pairs, $(-2, 2)$ and $(2, -2)$.

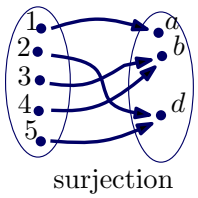
Complex arithmetic is all fun and games until someone loses an i.

In conclusion, the squaring function, $S\colon \mathbb{R} \to \mathbb{R}$, defined by $S(x) = x^2$, is a relation and a function. Its inverse relation, the square root, is not a function. By restricting the domain and range of S to non-negative reals, $S\colon \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ becomes a bijection (defined shortly), and its inverse, the principal square root, satisfies the definition of a function.

Notice how the formal definition encourages us to be more complete and precise (aka more pedantic and anal-retentive) about what is meant by 'square root.' This can be a good thing when coding or specifying what is to be coded, because the earlier an error or ambiguous specification is caught, the easier and cheaper it is to fix.

6.2.2 Types of functions

Quantified statements let us define special types of general functions as well. The most important of these are the bijections.



surjection

Partial function: A relation $R \subseteq A \times B$ is a *partial function* iff no $x \in A$ maps to more than one $y \in B$. *I.e.*, $\forall_{x\in A\ y,z\in B}\ ((x\ R\ y) \wedge (x\ R\ z)) \to (y{=}z)$. This drops one of the conditions of the definition of a function. In computer science, many of our programs are partial functions that give outputs only for legal inputs and give appropriate error messages (or inappropriate crashes) for illegal inputs. How many partial functions from A to B are possible?

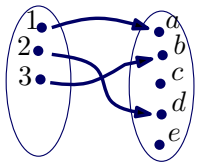
Surjection: A function $f\colon A \to B$ is a *surjection*, or *onto*, if and only if $f(A) = B$. In a drawing of a surjection, every element of B is the target of one or more arrows from A. Say this in notation:
Counting surjections is tricky. Start by listing all surjections for small domain and range sets, $A = [1..m]$ and $B = [1..n]$. When $m < n$ there can be no surjections. Do you agree with my counts in table 6.2?
We can create an inclusion/exclusion formula. Start with the set of all n^m possible functions, then subtract out those that miss one of the elements of the range: we choose which element to miss in n ways, then create a function in $(n-1)^m$ ways. Now, those that choose to miss two elements have been subtracted out twice, so we add that count back in: choose the pair to be missed in $\binom{n}{2}$ ways, and map in $(n-2)^m$ ways. But now any that misses a triple has been added back too often, so we need to subtract them out. . . . The resulting formula can be written as a

Table 6.2: Numbers of surjections of m onto n

$m\backslash n$	1	2	3
1	1	0	0
2	1	2	0
3	1	6	6
4	1	14	36
5	1	30	150

summation:

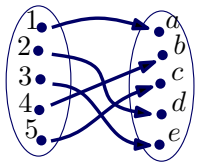
$$n^m - \binom{n}{1}(n-1)^m + \binom{n}{2}(n-2)^m - \cdots + (-1)^{n-1}\binom{n}{n-1} = \sum_{0 \le i \le n} (-1)^i \binom{n}{i}(n-i)^m.$$



injection

Injection: A function $f : A \to B$ is an *injection*, or *one-to-one*, iff each element of B is hit by mapping at most one element of A. We can express this using the 'at most one' idiom: [ ]. Replacing the conditional with its contrapositive, [ ], suggests that *two-to-two* should be the alternate name, since an injection maps two different elements of A to two different elements of B. (The definition of *function* already maps each element of A to a single element of B.)

How many injections from A to B are possible? Since no element of B can be reused, we get the permutation count, $P(|B|, |A|) = |B|(|B|-1)\cdots(|B|-|A|+1)$. Note that this correctly tells us that there are zero injections whenever $|A| > |B|$.



bijection

Bijection: A function $f : A \to B$ is a *bijection* if and only if it is an injection and a surjection. In a drawing of a bijection, every element of B has exactly one incoming arrow from A.

We can use quantifiers and write this as a single statement: a function $f : A \to B$ is a *bijection* (a one-to-one and onto function) iff

[ ]

Notice that this essentially says that the inverse relation f^{-1} is a function; bijection are exactly the set of functions that have inverses.

How many bijections from A to B are possible? Well, we must have $|A| = |B|$ to have any, and then we can apply the injection count to see that it is [ ].

We will increasingly derive consequences from definitions and prove new statements about discrete structures. For example, here is a type of function that we can prove is a bijection using the definition.

Suppose that $f : A \to A$ is its own inverse, meaning that $\forall_{a \in A}\, f(f(a)) = a$. Examples of functions with this property include negation (for logic or numbers), bitwise-XOR, $1/x$ for non-zero reals, and some cyphers like rot13.

f is an *involution*.

Lemma 6.2.1. *Any function $f : A \to A$ that is its own inverse is a bijection.*

Proof. We are given a function f satisfying $\forall_{a \in A}\, f(f(a)) = a$.

To show that f is surjective, we want to show that $\forall_{y\in A}\ \exists_{x\in A}\ f(x) = y$. For any given y, choosing* $x = f(y)$ works, since $f(x) = f(f(y)) = y$.

*Note: $\forall_y\ \exists_x$ so we are given a y, then can pick an x.

To show that f is also injective and thus a bijection, we want to show that $\forall_{x_1,x_2\in A}\ (f(x_1){=}f(x_2)) \to (x_1{=}x_2)$. So, suppose that we are given $x_1, x_2 \in A$ satisfying $f(x_1){=}f(x_2)$. Apply f to both sides: $f(f(x_1)) = f(f(x_2))$. But this means $x_1 = x_2$ since f is its own inverse. ∎

We can also show that the composition of two bijections is a bijection:

Lemma 6.2.2. *The composition of two bijections is a bijection.*

Proof. Let $f\colon A \to B$ and $g\colon B \to C$ be bijections. I want to show that the composition $g \circ f$ is a bijection—that it is a function that is both a surjection and an injection.

The function notation makes it easy to see that $g \circ f$ is a function, since each $a \in A$, is paired with $g \circ f(a) = g(f(a))$ as its unique value in C.

To show $g \circ f$ is a surjection, consider any $c \in C$. Since g is a surjection, $\exists_{b\in B}\ g(b) = c$; since f is a surjection, $\exists_{a\in A}\ f(a) = b$. But this a satisfies $g(f(a)) = c$, so $g \circ f$ is a surjection.

To show $g \circ f$ is an injection, suppose that we are given $a_1, a_2 \in A$ with $g(f(a_1)) = g(f(a_2))$. I want to show that $a_1 = a_2$. Note that because g is a bijection, we know that $f(a_1) = f(a_2)$. But then, because f is an injection, $a_1 = a_2$. ∎

6.3 Bijections and counting

We can now formally define what we informally stated in chapter 3: a set B has $|B| = n$ if and only if there is a bijection $f\colon [1..n] \to B$. We can generalize this definition to infinite sets as well: $|A| = |B|$ iff there is a bijection $f\colon A \to B$. Let's first look at consequences for counting finite sets, then briefly at infinite sets.

We first met the sum and product rules before we had notation for quantifiers, set operations, and functions; here is how they can now be stated for finite sets.

Theorem 6.3.1 (*sum rule*)**.** *For disjoint, finite sets A and B, $|A \uplus B| = |A| + |B|$.*

Theorem 6.3.2 (*product rule*)**.** *For finite sets A and B, $|A \times B| = |A| \cdot |B|$.*

Proofs are by building bijections; see Exercise 6.6.

The *pigeonhole principle* has a silly name and sounds obvious: you cannot put $n+1$ pigeons into n pigeonholes without having at least two share. Stated more mathematically, if I have two finite sets with $|A| > |B|$, then there is no

bijection $f: A \to B$. Sometimes its consequences are not completely obvious; we've already used it with Genji-mon and counting surjections.

- An airport with 1,500 or more landings a day must be able to accommodate two planes landing in the same minute.
- In a class of 35 students, if a majority are female and a majority are majoring in computer science, then at least one is both.
- Lossless compression cannot reduce the file size of all possible data files.

In general, with $|A| = m$ and $|B| = n$, for any function $f: A \to B$, there is some $b \in B$ for which $|f^{-1}(b)| \geq m/n$. That is, for at least one $b \in B$, the number of incoming arrows is at or above average. Let's prove this:

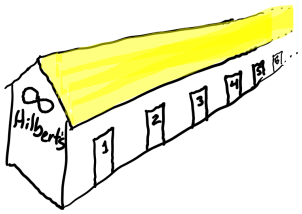
And NUH is the letter I use to spell Nutches
Who live in small caves, known as Nitches, for hutches
These Nutches have troubles, the biggest of which is
The fact there are many more Nutches than Nitches.
—Dr. Seuss, *On Beyond Zebra*

Lemma 6.3.3. *For any function $f: A \to B$ on finite sets, there is some $b \in B$ with $|f^{-1}(b)| \geq |A|/|B|$.*

Proof. Let $m = |A|$ and $n = |B|$. Since each element of A appears in $f^{-1}(b)$ for exactly one $b \in B$, the sum rule says $\sum_{b \in B} |f^{-1}(b)| = m$. If $|f^{-1}(b)| < m/n$ for each $b \in B$, then the n terms of the sum would total less than m. Thus, at least one $b \in B$ has $|f^{-1}(b)| \geq m/n$. ∎

There are some unexpected consequences of the definition that two infinite sets have the same cardinality iff there is a bijection between them. Some can be amusingly described as the movement of guests in Hilbert's Infinite Hotel, which has one room per positive integer in $\mathbb{Z}^+$, all occupied.

A bus drives up with an infinite number of new guests, one per positive integer. David Hilbert broadcasts to the existing guests that each guest in room i should move to $2i$, and that each new guest numbered j should proceed to room $2j-1$ and move in. The number of integers is the same as the number of evens, by the bijection $f(i) = 2i$. Even with no vacancy, there is no problem in doubling the number of guests in this hotel.



Buses numbered by $\mathbb{Z}^+$ drive up, each with infinitely many people, numbered by $\mathbb{Z}^+$. Hilbert tells each person to add their bus and person numbers, $n = b + p$, and proceed to room $\binom{n}{2} + p$. (Current guests are to use bus number zero.)

b\p	1	2	3	4	5...
0	1	3	6	10	15
1	2	5	9	14	20
2	4	8	13	19	26
3	7	12	18	25	33
4	11	17	24	32	41
5	16	23	31	40	50
⋮					

The table in the margin suggests that this pattern gives each person a room, filling the hotel! This sets up a bijection between $\mathbb{N} \times \mathbb{Z}^+$ and $\mathbb{Z}^+$ by using $\binom{n}{2}$ to count people with sums smaller than n, and adding p. The pairs of integers, the rational numbers, and even the number of finite length computer programs are all *countably infinite*, like the integers.

Some sets of people can not be accommodated, however. Suppose Hilbert tries to find rooms for a set of people labeled with all possible infinite sequences of coin flips $\{H, T\}^\infty$. We can prove that there is always someone who is not assigned to a room. Make the sequence whose ith symbol (H or T) differs from what the person in room i has as their ith symbol. (If there is no person in room i, then pick either symbol.) The resulting sequence belongs to a person P who does not have room j because room j is empty or is occupied by a person whose sequence differs in the jth symbol from P's.

This is Cantor's diagonalization argument, which shows that the number of H/T sequences is *uncountably infinite*, as are the real numbers and the number of functions on the integers. These have implications when you consider what functions are computable by programs, but that goes beyond the scope of this book.

6.3.1 Resource bounds and asymptotic notation

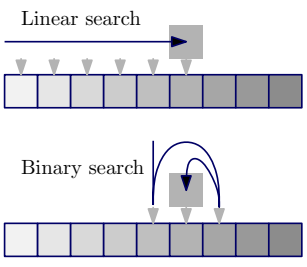
Computer scientists frequently report bounds on the resources (particularly time and memory) used by an algorithm as the problem size grows. Time and memory for any specific implementation depend on the underlying hardware and on programmer decisions. Because Murphy's law applies in a special way to computers, we are often interested in bounding the worst possible inputs, as a function of some measure of problem size. Furthermore, these bounds are often given in *asymptotic notation*, which hides specific constants and low-order terms and records only the relative growth of the time or memory. By comparing growth rates we can qualitatively compare resource requirements for different algorithms. And with a few experiments we can estimate implementation-dependent constants and extrapolate resource requirements over a larger range.

In this book I have chosen to use exact counts rather than asymptotic notation, but for those who are interested, I wanted to show how computer scientists use nested quantifiers and functions to precisely define the imprecision for reporting resource requirements.

Let's consider an example problem whose input consists of a chosen integer $x \in Z$ and an n-tuple of integers in non-decreasing order, $(a_1 \leq a_2 \leq \cdots \leq a_n) \in \mathbb{Z}^n$. The desired output has two cases: If x is found in the n-tuple, that is, if $\exists_{1 \leq i \leq n}$ such that $x = a_i$, then report i, the index where it is found. Otherwise report, "x is not in the tuple." (You could instead report the position where x belongs if it is not in the list.) We can imagine two different algorithms to solve this problem.

Linear search checks each element of the list in turn, starting with $i = 1$, and stopping when $a_i \geq x$.

Binary search compares x with the middle element of the tuple and continues the search in either the first or the second half of the tuple. (We'll derive a streamlined version of this algorithm in section 11.3.)



Let $f_A(x, a_1, a_2, \ldots, a_n)$ denote the number of element comparisons performed by an algorithm A on a particular input. When the algorithm A is clear from context, we omit the subscript. We'll use this as a proxy for running time. The actual running time of an implementation of algorithm A depends on many things: the language of the implementation, the quality of the compiler and the programmer, the underlying hardware, and the processor load conditions. But most costs can be applied either to a specific

comparison or to overhead. For a specific implementation of algorithm A, by doing a few timing tests with different inputs, we could determine constants c and d such that $cf_A(x, a_1, a_2, \ldots, a_n) + d$ is a good estimate of running time, even for extrapolating to larger inputs.

It takes work to calculate the number of comparisons for each possible input; often knowing the maximum is enough. Of course, as problems grow larger, the maximum is greater, so we define a function of n, the tuple size:

$$f_A(n) = \max_{x, a_1 \leq \ldots \leq a_n \in \mathbb{Z}} f_A(x, a_1, a_2, \ldots, a_n).$$

Spelling out the quantifiers and logic,

$$\begin{aligned} f_A(n) = y \text{ iff } & \exists_{x, a_1 \leq \ldots \leq a_n \in \mathbb{Z}}\ y = f_A(x, a_1, a_2, \ldots, a_n) \\ & \text{and } \forall_{x', a'_1 \leq \ldots \leq a'_n \in \mathbb{Z}}\ y \geq f_A(x', a'_1, \ldots, a'_n). \end{aligned}$$

Thus, $f_A(n)$ is the *upper bound* on the number of comparisons in A for any input (x value and non-decreasing n-tuple). By also bounding the time per comparison and overhead, we get the *worst-case time* for the algorithm A. We could also find the *best-case*, $\min_{x, a_1 \leq \ldots \leq a_n \in \mathbb{Z}} f_A(x, a_1, a_2, \ldots, a_n)$.

In the linear search algorithm A, the worst case is n and best case is 1 comparison. It is tempting to say that the *average case* is $n/2$ comparisons, but that would require defining a probability distribution on the set of all possible inputs, so we delay that until after chapter 14.

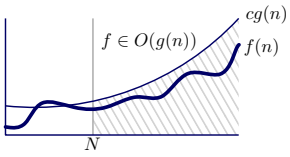
For binary search, should we check for equality ($x = A[i]$) in the loop, or only after, as suggested in section 11.3? If we do it in the loop, then the best case is 1 comparison, and worst is $2\lceil \log_2 n \rceil$, using the ceiling notation of chapter 7 to round up to the nearest integer. If we do it after, the best is $1 + \lfloor \log_2 n \rfloor$ and worst $1 + \lceil \log_2 n \rceil$. Since half of the inputs take the maximum and a quarter take one less, if we average over all possible inputs, checking after is less work. But this is a slight difference compared to the exponential improvement of using binary instead of linear search—an improvement that gets better as n grows larger.

Whether we consider worst, best, or some type of average case, there is another issue in reporting the times. Since we are using comparisons as a proxy for running time, we don't need to know the exact number of comparisons* To suppress the constants, and just look at the growth of functions, we define *asymptotic notation*, including big O for upper bounds, big Ω for lower bounds, and big Θ for both.

*Although, for this book, I have chosen to focus on exact counts, since it is easier to move from exact to approximate than the reverse.

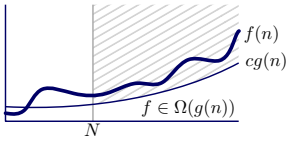
In English, a function $f : \mathbb{N} \to \mathbb{R}$ is said to be in the set $O(g(n))$ iff we can find a constant $c > 0$ so that, once n becomes large enough, $f(n)$ is positive, but always less than or equal to $cg(n)$. The quantified statement below makes this precise. Choosing c suppresses a constant factor; considering n large enough (choosing an N and considering all $n > N$) suppresses any

initial overhead to focus on the growth of $f(n)$ as n grows. The linear search algorithm is in $O(n)$, so it runs in worst-case linear time, and the binary search is in $O(\log n)$, so it runs in worst-case logarithmic time.



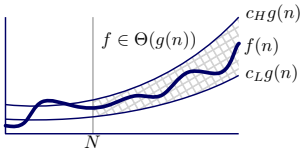
big O $f \in O(g(n))$ iff $\exists_{c \in \mathbb{R}^+, N \in \mathbb{N}}\ \forall_{n>N}\ 0 \le f(n) \le cg(n)$. Examples of functions in $O(n^2)$ include (pop-ups give possible values for c and N) $5n$?, $n\sqrt{n}$?, and $3n^2+2n+99$?. On the other hand, $n^3 \notin O(n^2)$, because no matter what N and c you choose, pick $n = \max(N, c) + 1$, and $n^3 > c^3 = cn^2$. For practice, what is the negation of $f \in O(g(n))$?

In this notation, n looks like a free variable in $f \in O(g(n))$, but in fact n is bound by a hidden "for all" quantifier. It would be better if the notation were $O_n(g(n))$, but $O(g(n))$ has become so entrenched that most computer scientists won't even notice this issue.



big Ω $f \in \Omega(g(n))$ iff $\exists_{c \in \mathbb{R}^+, N \in \mathbb{N}} \left(\forall_{n>N}\ (0 \le cg(n) \le f(n))\right)$. Examples of functions in $\Omega(n^2)$ include n^3 ? and $3n^2+2n+99$?. Notice that big O and big Ω are not opposites. In real analysis, a different definition is used : $f \in \Omega(g(n))$ iff $\exists_{c \in \mathbb{R}^+}\ \forall_{N \in \mathbb{N}} \left(\exists_{n>N}\ (cg(n) \le f(n))\right)$. That is, $f(n)$ goes above $cg(n)$ infinitely often.

big Θ $f \in \Theta(g(n))$ iff $\exists_{c_H, c_L \in \mathbb{R}^+, N \in \mathbb{N}} \left(\forall_{n>N}\ (0 \le c_L g(n) \le f(n) \le c_H g(n))\right)$. That is, a function $f \in \Theta(g(n))$ iff f is in both $O(g(n))$ and $\Omega(g(n))$. In English, for all sufficiently large n, the function f is trapped between two scaled copies of g.



*For a continuous, non-negative function: $n^3(1 + \sin(n))$.

Can you create a function that is neither $O(n^2)$ nor $\Omega(n^2)$? This would be a function that, for any chosen $c_1 > 0$, is greater than $c_1 n^2$ infinitely often and, for any chosen $c_2 > 0$, is less than $c_2 n^2$ infinitely often. The easiest way is to define function that behaves differently on even and odd inputs:* e.g.,

$$f(n) = \quad .$$

6.4 Summary

This chapter has built notation on the simple, but abstract concept of a relation. We define relation $R \subseteq A \times B$ to emphasize that a relation is just a set of pairs. We write $a\ R\ b$ as an abbreviation for $(a, b) \in R$ to emphasize the predicate defined by R and do formal proofs. In chapter 13 we define a graph $G = (A, B, R)$ to emphasize the connections between related elements. There

are several special types of relation this chapter has defined the most basic (a relation can be reflexive, symmetric, transitive, irreflexive, or anti-symmetric) and the most important (a relation can be a function).

Functions have their own notation for definition, $f : A \to B$ instead of $f \subseteq A \times B$, and use, $b = f(a)$ instead of $a \, f \, b$ or $(a, b) \in f$, but underneath they are simply special relations. Special types of functions include surjections, injections, and bijections; bijections are important for counting and other lossless transformations. The pigeonhole principle is one simple example.

6.5 Exercises and Explorations

Quiz Prep 6.1. Write the following conditions in notation using quantifiers.

1. Function $f: \mathbb{R} \to \mathbb{R}$ is bounded from above in the interval $[a, b]$.
2. Function $f: \mathbb{R} \to \mathbb{R}$ is strictly increasing.
3. Function $f: \mathbb{R} \to \mathbb{R}$ is non-decreasing.
4. There is a non-empty interval of the domain in which $f: \mathbb{R} \to \mathbb{R}$ is always smaller than $g: \mathbb{R} \to \mathbb{R}$.
5. Interval $[c, d] \subseteq \mathbb{R}$ is a maximal interval* in which $g: \mathbb{R} \to \mathbb{R}$ is less than or equal to $h: \mathbb{R} \to \mathbb{R}$.

*$[c, d]$ is a maximal interval with a property if any interval containing $[c, d]$ does not have the property.

6. Relation $R \subseteq A \times B$ is a function.
7. Function $f: A \to B$ is an injection.
8. Function $f: A \to B$ is a surjection.
9. Function $f: A \to B$ is a bijection.

Quiz Prep 6.2. Be able to specify a relation as a set, draw the graph of a relation, and fill in a table of properties like table 6.1.

Quiz Prep 6.3. Let $S = \{a, b\}$. Fill in the table with the numbers of the following: Please leave expressions like products or factorials unevaluated, to make it is easier to see where they came from.

	$\{1\}$	$\{1, 2\}$	$\{1, 2, 3\}$	$\{1, 2, 3, 4\}$
Relations on $S \times$	?	?	?	?
Functions from S to	?	?	?	?
Injections from S into	?	?	?	?
Surjections from S onto	?	?	?	?
Bijections between S and	?	?	?	?

Exercise 6.4. A binary relation $R \subseteq A \times A$ has an inverse, $R^{-1} = \{(b, a) \mid \forall_{a,b}\ a\ R\ b\}$, and a complement, $\overline{R} = (A \times A) \setminus R$. Show that

1. The inverse of a reflexive relation is reflexive and inverse of an irreflexive relation is irreflexive
2. The inverse of a symmetric relation is symmetric and inverse of an antisymmetric relation is antisymmetric.
3. The inverse of a transitive relation is transitive.
4. The complement of a reflexive relation is irreflexive and complement of an irreflexive relation is reflexive.
5. The complement of a symmetric relation is symmetric.
6. The complement of a transitive relation may or may not be transitive.
7. The inverse is never equal to the complement: $R^{-1} \neq \overline{R}$.

Exercise 6.5. How many of each of the following binary relations on $[1..3]$ are there? Be able to explain these counts. It may help to use a 3×3 Boolean matrix to represent all ordered pairs and determine which pairs are in the relation.

1. All relations ?
2. Reflexive relations ?
3. Irreflexive relations ?
4. Relations that are neither reflexive nor irreflexive ?
5. Reflexive relations that are symmetric ?
6. All symmetric relations ?
7. All antisymmetric relations ?
8. Relations that are neither symmetric nor antisymmetric ?
9. Relations that are not reflexive, not irreflexive, not symmetric, and not antisymmetric ?*
10. Is there a good way to count the transitive relations, except by brute force? I know of none.

Exercise 6.6. Build bijections to demonstrate the sum and product rules for finite sets, as stated in subsection 3.2.1:

\+ *sum rule*: For disjoint, finite sets A and B, $|A \uplus B| = |A| + |B|$.

Assume that we are given bijections $f_A : [1..|A|] \to A$ and $f_B : [1..|B|] \to B$. Define a function $g : [1..|A| + |B|] \to A \uplus B$ that is a bijection.

× *product rule*: For finite sets A and B, $|A \times B| = |A| \cdot |B|$.

Again, assume that bijections $f_A : [1..|A|] \to A$ and $f_B : [1..|B|] \to B$ are given. Let's build the inverse, $g^{-1} : A \times B \to [1..|A| \cdot |B|]$, because to build the function itself takes ceiling and mod functions, which are introduced in the next chapter.

►

Exercise 6.7. Let $(a_1, a_2, \ldots, a_n)$ be a sequence from $[1..n]$ with no repeated number. Show that if n is odd, then the product $\prod(a_i - i)$ is even. ►

Puzzle 6.8. This riddle was called old a hundred years ago: A man looking at a picture of an individual says, "Brothers and sisters have I none, but that man's father is my father's son." How must the pictured individual be related to the speaker? Draw a graph of either the "son of" or "father of" relation to help explain your answer.

Extension 6.9. A classic game theory scenario is the *prisoners' dilemma*: Police spot a car that looks like the stolen vehicle reported at the scene of yesterday's bank robbery, and turn on their sirens. The two friends in the car, Alan and Bob, want to avoid a three-year robbery sentence, so they quickly agree that they will claim to have stolen the car today, which carries a one-year sentence.

The police really want to solve the robbery, so they separate the pair and offer each their freedom (a year off the one-year sentence) if either gives evidence on the other. (Of course, if both give evidence, then the police will seek two-year sentences for each, so neither will be freed.) This can be summarized by a 2×2 matrix for a *payoff function*, $p : A \times B \to \mathbb{R}^2$, that maps A and B choices to *co-operate* (1) or *defect* (0) to the corresponding pair of prison sentences.

		B	
	A	1(*co-op*)	0(*defect*)
	1(*co-*)	$\langle -1, -1 \rangle$	$\langle -3, 0 \rangle$
	0(*def*)	$\langle 0, -3 \rangle$	$\langle -2, -2 \rangle$

Figure 6.1: Prisoners' dilemma *payoff matrix*

1. A must choose between *co-operate* (1) or *defect* (0), and wants to maximize his payoff (since prison time is a negative.) We say that one choice $a \in A$ dominates the other if, no matter what B chooses to do, A is no worse off for having chosen a. Does cooperate, defect, or neither dominate in the prisoners' dilemma?

2. For a payoff function, $p : A \times B \to \mathbb{R}^2$, in which A and B may have many options, write an expression that is true iff some $a \in A$ *dominates* all others.

3. Explore what happens with other payoff matrices.

Extension 6.10. A logic function with n variables has domain $\{T, F\}^n$ and range $\{T, F\}$. In other words, for each possible way to input an n-tuple of T and F, it must choose a T or an F.

1. How many different logic functions are possible on two input variables? (and, or, xor, if, iff are five familiar ones.)
2. How many different logic functions are possible on n input variables?
3. How many of the logic functions with two input variables actually use both? For example, $f(p, q) = (p \wedge q) \vee (p \wedge \overline{q})$ does not really depend upon q.
4. What is a good definition for a logic function on n variables *using the ith variable*?
5. How many of the logic functions with three input variables actually use all three?
6. Determine a formula for counting the number of logic functions with n input variables that use all n.

▶

Exploration 6.11. Garrison Keillor closed his monologues with, "Well, that's the news from Lake Wobegon, where all the women are strong, all the men are good looking, and all the children are above average." Formulate the last claim mathematically, using quantifiers and functions, and discuss under what conditions it might be possible.

Chapter 7
Math Review

If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.

—John von Neumann

We need to be familiar with the properties of division, floor, remainders (mod), summation, exponentiation, and logarithms, so we review these in this chapter. But let's do this with a purpose. All of these combine to give public key encryption by the RSA algorithm* [21], which is the main method of encryption protecting all your secrets on the Internet—from passwords to authentication of financial transactions. So let's consider the story of how I can announce publicly two numbers (my public key) that anyone can use to send me a message that only I can read. In the process, we'll also review bases, binary numbers, fractions, primes, and vectors.

*The algorithm creators, Ron Rivest, Adi Shamir, and Len Adleman, founded RSA Data Security in 1982, which EMC bought in 2006 for $2.1 billion.

Objectives: On completion of this chapter, the student will be able to do the following operations. Each operation reviews several concepts:

1. Convert a message string to a number, and numbers to binary. (Reviewing numbers represented as strings, summation notation, binary numbers (base 2) and other bases, Horner's rule)
2. Convert a number back to a message, or to another base. (Reviewing logarithms, floor, mod, ceiling, geometric series)
3. Hide information inside a published number. (Reviewing divisors, primes)
4. Keep numbers that we work with small even though we raise numbers to very large exponents. (Reviewing exponentiation, properties of mod)
5. Develop a family of bijections that are *one-way functions*: easy to compute but hard to invert unless you know some hidden information. (Reviewing fractions, vectors, slope, greatest common divisors)

In the process, we'll develop some of the number theory of integers mod a prime. You'll understand this best if you work out the material yourself,† so as you read, be ready with scratch paper, a notebook, or a fine-point pencil for marginal notes.

†Math is not a spectator sport
—Jerry Mortensen

7.1 Messages and bases

First, let's see how to convert a message string to numbers, and numbers to binary.

7.1.1 Strings to numbers

I could joke about decimal integers, but there's no point.

An integer can be written as a string from the alphabet of *digits* [0..9]. Consider a string of $k+1$ digits, from d_k, the most significant, to d_0, the least significant. (This may seem a strange way of numbering, but I hope you see how it simplifies the formula below.) The string of decimal digits $d_k d_{k-1} \cdots d_1 d_0$ is the number

$$d = \sum_{0 \le i \le k} d_i 10^i = d_k 10^k + d_{k-1} 10^{k-1} + \cdots + d_1 10 + d_0. \tag{7.1}$$

Notice that when we talk about the place values (ones, tens, hundreds, thousands), we are talking about powers of ten, starting from zero (10^0, 10^1, 10^2, 10^3). A number with d_k as the highest non-zero digit has $k+1$ digits.

*Base-16, or *hexadecimal*, uses the alphabet 0–9,A–F. So what are the decimal values of ACE_{16}, $CAFE_{16}$, and $DEC0DE_{16}$?

By generalizing Equation 7.1, a string from an alphabet of b elements can be written as a number:* Assign each symbol of the alphabet a unique number from $[0..b-1]$, then, in Equation 7.1, replace the '10's with 'b's and each d_i with the number of the ith symbol in the string. When the alphabet has two symbols (usually 0 and 1), you have the binary numbers.

A better way to convert a string to a number does not need powers of b: process the string one character at a time, from d_k down to d_0, distributing multiplication by b across addition:

$$d = (\cdots((d_k b + d_{k-1})b + d_{k-2})b + \cdots + d_1)b + d_0.$$

The calculation starts with the value of the first character, then repeatedly multiplies by b and adds the next character. This is *Horner's rule* for evaluating a polynomial, although in our application of the rule the polynomial is evaluated only at b, the base of the number system. (More in subsection 8.2.3.)

To convert a number d back to a string, or equivalently to get the string of digits in base b, we can reverse the process using division, floor, and mod, which are described in the next three subsections.

7.1.2 Floor and ceiling

Floor rounds a real number down to the next integer, and ceiling rounds up. Formally, for all $x \in \mathbb{R}$, the floor $\lfloor x \rfloor$ is the greatest integer less than or equal to x. So $m = \lfloor x \rfloor$ iff $m \in \mathbb{Z}$ and $m \le x$ and

$$\nexists_{n \in \mathbb{Z}} m < n \le x \equiv \forall_{n \in \mathbb{Z}} (n \le x) \to (n \le m) \equiv \forall_{n \in \mathbb{Z}} (m < n) \to (x < n).$$

The ceiling $\lceil x \rceil$ is the smallest integer greater than or equal to x. Here is a useful property of floor and of ceiling; let me prove part of it formally, step by step.

Lemma 7.1.1. *For all reals $x \in \mathbb{R}$, we have $x-1 < \lfloor x \rfloor \leq x$ and $x \leq \lceil x \rceil < x+1$, with equality iff x is an integer.*

I'll prove this for floor, and encourage you to make the changes to prove it for ceiling.

1. Let $m = \lfloor x \rfloor$; we know that $m \in \mathbb{Z}$ and $m \leq x$ — first condition in defn floor
2. Since $m < m+1$ and $m+1$ is an integer, — addition closed over $\mathbb{Z}$
3. we learn that $m \leq x < m+1$. — defn floor $\forall_n$, with $n = m+1$
4. Then $-m \geq -x > -m-1$ — Multiply by -1, reverses inequalities
5. So $x \geq m > x-1$. — add $x+m$ to all sides QED.

With these inequalities, we can prove several properties of floor and ceiling (Exercise 7.3). Try to prove that we can pull integers from inside floors or ceilings: For any $x \in \mathbb{R}$, $n \in \mathbb{Z}$, we have $\lfloor x+n \rfloor = \lfloor x \rfloor + n$ and $\lceil x+n \rceil = \lceil x \rceil + n$. Also, for all integers $n \in \mathbb{Z}$, $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$ and $\lfloor (n+1)/2 \rfloor = \lceil n/2 \rceil$.

7.1.3 Length of a given message

By the way, given an integer $n \in \mathbb{Z}^+$, how many digits are in its decimal string? How many bits* if I write n in binary? If n encodes a message in an alphabet of b symbols, how long is the message? The answers come from logarithmic functions, which we review in this section.

*bits = binary digits, each 0 or 1.

Recall that the base-b logarithm, $\log_b x$, is the exponent on b that gives x. That is, $x = b^{\log_b x}$. The natural log function uses base $e \approx 2.71828$: $\ln x = \log_e x$. In computer science we use log base 2 often enough to give it its own notation: $\lg x = \log_2 x$.

There are 10 types of people in the world: Those who understand ternary, those who don't, and those who thought this was a binary joke.

Back to digits: in base b, the non-negative integers with m-digits—those with d_{m-1} as the leading non-zero digit—are b^{m-1} through $b^m - 1$. (Check base 10: the single digit numbers, $m = 1$, are 1–9.) Thus, the first non-zero digit of n in base b is at position $\lfloor \log_b n \rfloor$, giving $\lfloor \log_b n \rfloor + 1$ digits. (Note that $\log_b 0$ is undefined—zero has no non-zero first digit.) For all $n \in \mathbb{Z}^+$, $\lfloor \log_b n \rfloor + 1 = \lceil \log_b (n+1) \rceil$ since log is a (slowly) increasing function.

You can remember the important properties of the log functions over $x > 0$ by thinking of $\lg x$ as roughly the number of bits† to write down $\lceil x \rceil$. For other bases b, $\log_b x$ is roughly the number of digits from $[0, b)$ to write x.

†$\lceil \lg n \rceil$ is also how many "$x \leq a$" questions are necessary and sufficient to guess any $x \in [1..n]$.

- If you double x in binary, you need just one more bit. If you multiply by b in base b, you need one more digit: $\log_b(bx) = 1 + \log_b x$.
- If you multiply xy, you need to add digits: $\log_b(xy) = \log_b x + \log_b y$.
- If you divide x/y, you can take away digits $\log_b(x/y) = \log_b x - \log_b y$.
- Powers (repeated multiplication inside) come out as multipliers (repeated addition outside): $\log_b(x^y) = y \log_b x$.
- Negative powers work, too: $\log_b(1/x) = -\log_b x$.

- To compute log base b if you know log base a, just divide by $\log_a b$ because that is how many digits from $[0, a)$ are replaced by each digit from $[0, b)$: $\log_b x = (\log_a x)/\log_a b$.

The last gives the relationship between the number of characters in a message and bits in the number: $\log_b x = (\lg x)/\lg b$, since

$$2^{\lg x} = x = b^{\log_b x} = \left(2^{\lg b}\right)^{\log_b x} = 2^{\lg b \cdot \log_b x}.$$

So if your calculator does not have log to some base, use any other log and remember to divide by log of your desired base. This justification used properties of exponents, which are usually taught before logs but which fit into my story later, in subsection 7.2.3.

Let's return now to encoding and decoding.

7.1.4 Mod for decoding

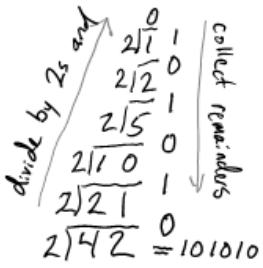


Figure 7.1: Decimal to binary: remainders on dividing by 2.

For all pairs of reals, $x, y \in \mathbb{R}$, with $y \neq 0$, define $x \bmod y = x - \lfloor x/y \rfloor \cdot y$. For positive integers, $x \bmod y$ is just the remainder when dividing x by y.* Recall the property of floors: $x/y - 1 < \lfloor x/y \rfloor \leq x/y$. Subtracting x/y from all sides, then multiplying by $-y$ we see that for positive y values $0 \leq (x \bmod y) < y$, and for negative y values $y < x \bmod y \leq 0$.

Using mod we can convert a positive integer d into its string of digits in base b, starting with the least significant digit d_0. We can think of this as decoding d to recover its ℓ-symbol message $d_\ell \cdots d_1 d_0$ in right-to-left order. Initialize $\ell = 0$. While $d > 0$, assign $d_\ell = d \bmod b$, increment $\ell = \ell + 1$, and update $d = \lfloor d/b \rfloor$. Note that each $d_\ell \in [0, b)$, as it should be.

Mod can also be used to create bijections for encoding and decoding. The classic 'Caesar cypher' chooses a *key value* $k \in [1, b)$ and shifts each character of the message, substituting $m_i = (d_i + k) \bmod b$ for each d_i.† This is a *private key* encryption method—the sender and receiver must each know, and keep secret, the key value k, because anyone who discovers it can decode by calculating $d_i = (m_i - k) \bmod b$.

With only $b - 1$ possible keys to choose from, this is very weak encryption. You can do a little better by grouping ℓ symbols of the message, converting to a number d, and choosing key $k \in [1, b^\ell)$. But even this is weak. For one thing, it falls to a *known cypher-text attack*: if I trick you into encoding d as m for me, I can calculate your private key $k = (m - d) \bmod b$ and read all your messages.

I want to allow anyone to encode a message for me using public information, yet be the only one who can decode using my private information. Seems impossible, but the next section shows how, by hiding large prime numbers in a published composite number. If what you've already read is

*Note that remainder is a special case of mod; we don't define mod by using remainder, but more generally using floor!

†What key k, in a 27-symbol alphabet ('blank', A–Z), shifts IBM to HAL?

not yet clear to you, however, this would be a good point to take a break and then review.

7.2 Encoding by exponentiation mod hidden primes

First, let's look at divisibility and primes, then review exponentiation. Then we'll state a 17th century theorem that ties these together and that has a beautiful, short proof by counting *necklace sequences*. The result gives an idea for using hidden primes to encode and decode messages.

7.2.1 Divisibility

For all integers $a, b \in \mathbb{Z}$, we say that a *divides* b, denoted $a|b$, iff $\exists_{m\in\mathbb{Z}}\ am = b$. Some use as equivalent expressions that a is a *divisor* of b, that b is divisible by a, that $b \bmod a = 0$, or that b is congruent to 0 $(\text{mod } a)$.*

*Sorry for the many ways to say the same thing. Many different people have invented different terms for these concepts over the centuries.

For example, 2 divides 6 (that is, 2|6 is true) because $2 \cdot 3 = 6$. Note that divides is a function $\mathbb{Z}^2 \to \{T, F\}$ and that the definition is based on multiplication—we do not say that 2|6 because $6/2 = 3$. Our chosen definition does not need to handle a special case if the first number might possibly be zero, even though division by zero is not defined. Check what this definition says about -1, 0, and 1: Zero divides only itself, every number divides 0, and both -1 and 1 are *divisor*s of every number.

Convince yourself, or (even better) a skeptical classmate, of the truth of these properties of divides for all integers $a, b, c \in \mathbb{Z}$. The most convincing demonstrations go back to the definition of divides.

- If $a|b$ and $b|c$, then $a|c$.
- If $a|b$ then $ac|bc$.
- If $a|b$ and $b \neq 0$ then $|a| \leq |b|$.
- If $a|b$ and $b|a$ then $|a| = |b|$.
- If $a|b$ and $a \neq 0$, then b/a is an integer.
- If $a \neq 0$, then $b \bmod a = c \bmod a$ iff $a|(b - c)$.

The first few of these properties give the classic demonstration that $\sqrt{2}$ is not a rational number; that no $p, q \in \mathbb{Z}$ satisfy $p/q = \sqrt{2}$.

Theorem 7.2.1. *$\sqrt{2}$ is irrational.*

Proof. Suppose, for the sake of deriving a contradiction, that there exist positive integers p and q with $p/q = \sqrt{2}$. Choose p and q to be a proper fraction†, with no common divisor greater than 1. But $p^2 = 2q^2$, so 2 divides p^2, and must also divide p. But then 4 divides p^2, so 2 divides q^2 and must‡ also divide q. Thus, p and q have 2 as a common divisor, which contradicts our choice. Therefore, there is no pair of positive integers such that $p/q = \sqrt{2}$. ☙

†Equivalently, choose the smallest positive p and q such that $p/q = \sqrt{2}$.

‡Why the two "must"s? On scratch paper, prove the contrapositive: any odd squared is odd.

7.2.2 Hiding two primes in a composite number

A positive integer N is *composite* iff it can be written as the product of two smaller, positive integers (factors). That is, there exist integers $1 < a, b < N$ with $N = ab$. An integer $p > 1$ is *prime* iff p is not composite; p's only positive divisors are 1 and itself. Zero and 1 are neither prime nor composite.

The fundamental theorem of arithmetic is that every integer > 1 can be written as a product of primes uniquely, up to order of the prime factors. It follows from the first of these properties, which apply for all integers a and b, and for all distinct primes p and q:

- $p|ab$ iff ($p|a$ or $p|b$).
- ($p|a$ and $q|a$) iff $pq|a$.

For a composite number $N > 0$, let a be the smallest possible factor greater than 1 (one) for which $N = ab$. We can observe two facts: First, a must be prime, or one of its own factors would be a smaller factor of N. Second, $a \leq \sqrt{N}$ since $b \geq a \geq 2$. Thus, one way to demonstrate that an integer N is composite is to test, for all primes $p \leq \sqrt{N}$, if p divides N.

Directly testing all primes up through $\sqrt{N}$ can still be a lot of work. The Prime Number Theorem says that up through x there are roughly $x/\ln x$ primes. Thus, if we create N by multiplying two 512- to 1024-bit prime numbers, someone directly testing would check more primes than there are atoms in the universe before factoring N. With today's algorithms and technology, composite numbers can be published and still hide two large primes.

How do you use a hidden prime and how can you find large primes to hide anyway? The answers to both of these questions come from exponentiation, so here is another digression.

7.2.3 Messages of given length

With an alphabet of b symbols, how many possible messages are there of length k? This is the reverse to the problem of subsection 7.1.3, and we've already seen the solution in chapter 3: the product rule says b^k, since we choose one of b symbols, k separate times. If we don't allow leading blank symbols, or leading zeros in decimal or binary strings, then we have $b^k - b^{k-1}$ by the sum rule.

Knowing the number of message strings of length x is b^x can help us remember the properties of exponentiation even for non-integer b and x. (We will assume that $b \geq 0$ whenever $x \notin \mathbb{N}$.) In these properties, for 'number' read "number of possible message strings using a b-symbol alphabet" and for 'exponent' read "string length," so the second says, "To make all possible message strings of length $x + y$ from b symbols, we independently choose

two strings, the first of length x and second of length y, then concatentate. Thus the number of possible strings is the product, $b^{x+y} = b^x b^y$."

- Multiplying a number by the base adds 1 to the exponent: $b \cdot b^x = b^{x+1}$.
- Multiplying numbers adds exponents: $b^x b^y = b^{x+y}$.
- Dividing numbers subtracts exponents: $b^x / b^y = b^{x-y}$.
- Powers multiply exponents: $(b^x)^y = b^{xy} = (b^y)^x$.
- Reciprocals are negative powers: $(1/b) = b^{-1}$.
- Exponents distribute over multiplication of bases: $(ab)^x = a^x b^x$.
- For base b, log and exponentiation are inverses: for all reals $x = \log_b b^x$, and for positive reals $y = b^{\log_b y}$.

Many interesting combinations of logs and exponents come up that are probably easier to derive than to memorize. Here's a cute one; you can trade exponent bases through log powers: $b^{\log_a x} = (a^{\log_a b})^{\log_a x} = (a^{\log_a x})^{\log_a b} = x^{\log_a b}$.

Exponential functions of integers grow very quickly, even when the base is small. You should know your first ten powers of two, and remember that 2^{10} is a little over* a thousand (10^3), so 2^{20} is over a million, 2^{30} is over a billion. The number of atoms in the universe is estimated at 10^{81} or 2^{270}.

*A programmer buys cheese by the kilo because he thinks he'll get 24 extra grams.

If we are taking exponents or doing other arithmetic with integers mod N, we can avoid storing numbers above N by taking mod before and after each calculation. Suppose that $c, d, N \in \mathbb{Z}$ with $N \neq 0$ and $e \in \mathbb{N}$; can you convince yourself that these are true?

- $(c + d) \bmod N = ((c \bmod N) + (d \bmod N)) \bmod N$.
- $cd \bmod N = ((c \bmod N)(d \bmod N)) \bmod N$.
- $c^e \bmod N = (c \bmod N)^e \bmod N$.

Encoders and decoders both compute functions like $x = m^e \bmod N$. To use the second and third properties to keep the magnitudes of intermediate numbers manageable, write the exponent in binary as

$$e = (\cdots((e_k \cdot 2 + e_{k-1})2 + e_{k-2})2 + \cdots + e_1)2 + e_0,$$

where each $e_i \in \{0, 1\}$. Start with $x = 1$ and, for $i = k$ down to 0, set $x = (x \cdot x \bmod N) \cdot (m^{e_i}) \bmod N$. This repeatedly squares x and, for each 1 bit in the binary expression of e, multiplies by m. This calculation is easily encoded into a spreadsheet, since all numbers are kept below N. It ends in $\lceil \lg e \rceil$ steps with $x = m^e \bmod N$, as desired. table 7.2 works through three examples.

7.2.4 The unexpected power of counting

A 17th century theorem, known as Fermat's Little Theorem,† is crucial to both finding and using hidden primes. It has a 20th century proof based on counting p-tuples [7] that is so short and clever that I'd like to convince you

†Fermat is famous for stating theorems and letting others find the proofs; his "Last Theorem" was proved only in 1995.

that the theorem is true before I explain how it can be used. So for now, just notice that this gives a way to start with a message $m \in [0, p)$, calculate an exponential, and get the message back.

Theorem 7.2.2 (Fermat's little theorem)**.** *For all integers m and primes $p > 1$, $m^p \mod p = m \mod p$.*

Let's count possible *necklaces*—circular sequences of p beads, each of which is one of m different colors. With $p = 3$, for example, a single color gives the single necklace *aaa*. Two colors give four: *aaa*, *aab*, *abb*, and *bbb*: other 3-tuples, like *aba*, are just alternative ways to write the second or third circular sequences as a straight sequence. Three colors give eleven sequences: 3 using one color, 6 using two, and 2 using all three colors: *abc* = *bca* = *cab* and *cba* = *bac* = *acb*. Note that these two are different because we may circularly rotate necklaces, but not flip them over.

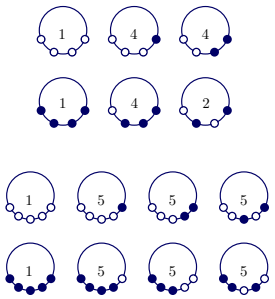


Figure 7.2: All 4-bead and all 5-bead necklaces with 2 colors, each with the number of straight sequences.

Necklaces that have a repeating color pattern, like *aaa* or *abab* = *baba*, produce fewer than p straight sequences when cut, but I claim that, for all primes p, the only p-bead necklaces that don't produce p different straight strings when cut are the single-color necklaces. If you've (unexpectedly) studied abstract algebra or group theory, then you already know that the number of straight strings from a p-bead necklace divides p. If not, try examples until you find this plausible, and remember that we should return to verify this formally (Corollary 7.3.4) once we are sure that it is worth the effort. Mathematics outside of textbooks invariably proceeds like this, jumping ahead and later backfilling with the details.

Proof of Thm. 7.2.2. Let n be the number of necklaces* (circular sequences) of p beads from m colors that use at least two colors. Each can be cut in p ways to form a different straight sequence. Thus, $np = m^p - m$, where the right side is the number of all p-tuples of m elements chosen with repetition, minus those tuples with a single color. We conclude that p divides $m^p - m$. ∎

*This is our second *combinatorial proof* by counting a set in two ways. In mathematics, any trick used three times becomes a method.

Fermat's little theorem can help us reject composite numbers in a search for large primes: if we find some $m \in [0, p)$ for which $m^p \mod p \neq m$, then p cannot be prime. The theorem doesn't say that composite numbers must fail this test, but many do fail for many m. A few composites, known as the *Carmichael numbers*, actually pass for all m; the smallest is 561. There are many fewer Carmichael numbers than primes, so there is a good chance that if a number passes the Fermat test for many m, then it is prime.

Perhaps more importantly, the theorem gives us an idea for public key cryptography: Let's find three numbers $E > 1$, D, and N so that the composite $E \cdot D$ has a Fermat-like property that for all $m \in [0, N)$, the $m^{ED} \mod N = m$. We publish N and E so that anyone can compute an encoded message

Table 7.2: Encryption with public key $N = 10403$, $E = 209$ and decryption with private $D = 2489$ for 1337, 42, and 9876. Binary representations of E and D give the steps of the encoding and decoding functions, $x = m^E \bmod N$ and $y = x^D \bmod N$, as described in subsection 7.2.3.

	Encoding (all ops mod $N = 10403$)			
E	step	values		
1	$m_1 =$	**1337**	**42**	**9876**
1	$m_2 = m_1^2 \cdot m_1$	4936	1267	7027
0	$m_3 = m_2^2$	270	3227	6091
1	$m_4 = m_3^2 \cdot m_1$	1593	5292	7845
0	$m_5 = m_4^2$	9720	388	10280
0	$m_6 = m_5^2$	8757	4902	4726
0	$m_7 = m_6^2$	4536	9077	10238
1	$x = m_7^2 \cdot m_1$	**8105**	**7098**	**8565**

	Decoding (all ops mod $N = 10403$)			
D	step	values		
1	$y_1 =$	**8105**	**7098**	**8565**
0	$y_2 = y_1^2$	6483	10278	7672
0	$y_3 = y_2^2$	1169	5222	9813
1	$y_4 = y_3^2 \cdot y_1$	6835	2475	7909
1	$y_5 = y_4^2 \cdot y_1$	9752	4451	2697
0	$y_6 = y_5^2$	7681	4089	2112
1	$y_7 = y_6^2 \cdot y_1$	3453	3093	9995
1	$y_8 = y_7^2 \cdot y_1$	1551	537	1801
1	$y_9 = y_8^2 \cdot y_1$	10281	897	2199
0	$y_{10} = y_9^2$	4481	3578	8609
0	$y_{11} = y_{10}^2$	1571	6394	3909
1	$y = y_{11}^2 \cdot y_1$	**1337**	**42**	**9876**

$y = m^E \bmod N$. I decode with $y^D \bmod N$ to recover m. Our N will actually hide two primes that will determine D from our choice of E, as described in the next section.

As an example, table 7.2 encodes and decodes three 4-digit numbers using $N = 10403$, $E = 209$ and $D = 2489$. These numbers are much smaller than would be used in a real encryption scheme, and are chosen so you should be able to factor this N rather easily to discover the hidden primes. Note that it would not be good to use the factors of N as the encoding and decoding exponents, even if they had the Fermat property, because then anyone could recover $D = N/E$. Thus, it will take some more work, via a digression into fractions, vectors, and invariants, to complete this.

7.3 Finding the encoding and decoding exponents

We need another theorem to support the calculation of the encoding and decoding exponents, E and D. This calculation is even older, dating back to Euclid, but I'm going to develop it from a 19th century look at fractions and vectors on a Cartesian grid so that we can review those. If your brain is full with Fermat's little theorem, this is a good place to break and review before continuing.

7.3.1 Fun with fractions

In fractions, there is a fine line between numerator and denominator.

I trust that you recall the proper way to add fractions, by putting them over a common denominator before adding: $\frac{a}{b} + \frac{c}{d} = \frac{ad}{bd} + \frac{cb}{bd} = \frac{ad+cb}{bd}$. Just for fun, though, define an operation that adds two fractions the wrong way, giving what is called their *mediant*:

$$\frac{a}{b} \oplus \frac{c}{d} = \frac{a+c}{b+d},$$

Start with the following sequence of three fractions on the number line, representing $-\infty$, 0, and $+\infty$:*

$$\frac{-1}{0} \quad \cdots\cdots\cdots\cdots \quad \frac{0}{1} \quad \cdots\cdots\cdots\cdots \quad \frac{1}{0}$$

Repeatedly choose two adjacent fractions and insert the mediant between them. Does it belong between? That is, if $a/b < c/d$ do we know that $a/b \overset{?}{<} (a+c)/(b+d) \overset{?}{<} c/d$? Can we get every fraction in this way? Do we ever get a fraction more than once, for example, do we see $1/2$ again as $3/6$ or $42/84$? How could we find fractions close to $\sqrt{2}$? We know from Theorem 7.2.1 that we won't find $\sqrt{2}$ itself as a fraction.

*Sequences between $0/1$ and $1/1$ that have all fractions with denominators less than n are known as *Farey sequences* because Cauchy learned of them from a short letter of John Farey in 1816, even though the mediant algorithm was published by Chuquet in the 1400s, and fully analyzed by Haros in 1802. Many mathematical results and structures are not named for their originators.

As you probably recall, a fraction p/q is in *lowest terms* if there is no integer $c > 1$ that divides both p and q. That is, $\forall_{c \in \mathbb{Z}^+}$ if $(c|p) \wedge (c|q)$ then $c = 1$. The pair p, q is then said to be *relatively prime*.

Let's demonstrate that starting with the sequence $(-1/0, 0/1, 1/0)$, repeatedly choosing any two adjacent fractions $a/b < c/d$, and inserting their mediant $(a+c)/(b+d)$ between them gives every rational number exactly once, in lowest terms. To make the demonstration convincing, we can observe some *invariants* – statements that always remain true during the procedure.

Lemma 7.3.1. *The following are invariants of generating mediants starting from* $(-1/0, 0/1, 1/0)$*:*

1. *For adjacent fractions* $a/b < c/d$*, we have* $bc - ad = 1$.
2. *Except for the initial* $-1/0$ *and* $1/0$*, all denominators are positive.*

Key idea. The formal proof of these invariants is by induction, which we study in depth in chapter 10. See Exercise 10.12.

Invariant 1 holds for the initial sequence. It remains true when you insert mediant $(a+c)/(b+d)$ between adjacent $(a/b, c/d)$ with $bc - ad = 1$ because the two new adjacent pairs satisfy $b(c+a) - a(b+d) = bc - ad = 1$ and $(b+d)c - (a+c)d = 1$.

Invariant 2 also holds for the initial sequence and remains true because denominators are formed by adding denominators. QED

The invariants let us answer the four questions about mediants asked above.

A1. The sequence of fractions is always in increasing order.

Combining invariants shows the gap between adjacent fractions is positive:

$$\frac{c}{d} - \frac{a}{b} = \frac{cb - ad}{bd} = \frac{1}{bd} > 0.$$

A2. Every fraction in the list is in lowest terms.

When a/b is formed, it is followed on the list by c/d with $cb - ad = 1$. Since any common factor of a and b must divide $cb - ad$, the only common factor can be 1.

A3. Any fraction $x = p/q$ can be made to appear in the list.

If we are looking for a number x in the list, the natural thing to do is to find adjacent fractions with $a/b \le x < c/d$, then replace one of these by their mediant so that x remains in the interval.

We can check that this starts by finding the interval $a/1 \le x < c/1$ with $c = a+1$. As we continue from there, we see the mediant denominator growing—it is always larger than the denominator of either fraction $(a/b, c/d)$, and it replaces one of the two.

We continue to replace mediants so that $a/b \le x < c/d$, stopping if we find x. Now, suppose that $x = p/q$, where integers p and q are not necessarily relatively prime, but $q > 0$.

I claim we must find x before either of the denominators, b or d, become greater than q. This is because $x = p/q$ must fit into the gap of width $c/d - a/b = 1/(bd)$ between the adjacent fractions. Moreover, since we have a strict inequality $x < c/d$, we know that $cq - pd$ is a positive integer. Combining these,

$$\frac{1}{bd} \ge \frac{c}{d} - \frac{p}{q} = \frac{cq - pd}{qd} \ge \frac{1}{qd},$$

so we learn that $b \le q$ and, therefore, we must have $d > q$. But then

$$\frac{1}{bq} > \frac{1}{bd} \ge \frac{p}{q} - \frac{a}{b} = \frac{pb - aq}{bq} \ge 0,$$

so $pb - aq$ must equal zero and $a/b = p/q$. So, $x = p/q$ is in the list.

A4. To approximate any real number x by the nearest rational with denominator at most q, maintain the two adjacent fractions with $a/b \le x < c/d$ as in A3. When the denominator $b + d > q$, report the fraction nearer to x.

Just an interesting fact we learn along the way.

For later reference, let's label as a Lemma the key fact from A1–A3.

Lemma 7.3.2. *For relatively prime integers a and b there are integers $c, d \in Z$ satisfying $bc - ad = 1$.*

Proof. Since a and b are relatively prime, the fraction a/b will appear in the list of mediants. When it does, let c/d be the following fraction: we know that $bc - ad = 1$. ∎

7.3.2 A vector view

Changing your perspective on a mathematical problem can help give intuition to find or understand a solution, so let's take a different view of what we have just done. If you are a visual thinker, like I am, you may find it explains the calculations of the previous section. Others may find the calculations more convincing.

A fraction a/b is the slope of a vector from the origin to point (b, a). When a/b is in lowest terms, then the line from the origin of slope a/b first hits an integer grid point at (b, a).

The two initial vectors $(1, 0)$ and $(0, 1)$ define a unit square whose only grid points are at the corners. From this perspective, the mediant of two vectors (b, a) and (d, c) is formed by the usual rule for vector addition: $(b, a) + (d, c) = (b + d, a + c)$, which completes the fourth corner of the parallelogram defined by the origin and the two input vectors.

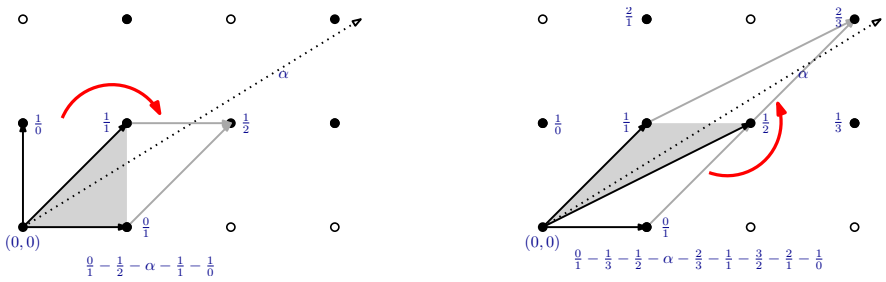


Figure 7.3: Fractions correspond to integer vectors and a/b in lowest terms means the vector from the origin to (b, a) does not pass through any other integer grid point. Diagonals of parallelograms construct mediants; you can cut a parallelogram into two triangles along the diagonal, and re-assemble into a new parallelogram, as indicated by curved arrows.

figure 7.3 illustrates that replacing one of the starting vectors with the mediant simply rearranges triangles of the parallelogram so that total area is preserved and grid points continue to appear only at parallelogram corners; these are the invariants of the mediant procedure. In fact, the algebraic invariant $bc - ad = 1$ is just the area calculation for the parallelogram.

If we draw the line of any given slope x, we can keep a pair of vectors* on opposite sides of that line to get upper and lower approximations to that slope. Since the parallelograms have grid points only at the corners,

*Alternate replacing the fraction above and below by the mediant and you get Fibonacci numbers whose slopes approximate the golden ratio $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ or its reciprocal $1/\phi = \phi - 1 \approx 0.618$.

all fractions generated are in lowest terms, and, if x is rational, it will be generated.

7.3.3 Common divisors

Lemma 7.3.2 is what we'll need to find the encoding and decoding exponents, so let's practice with it before we return to encryption.

The *greatest common divisor* of p and q is the largest number that divides both. In notation, $m = gcd(p, q)$ iff $m|p \wedge m|q \wedge (\forall_{n>m}\ (n \not| p \vee n \not| q))$.

Using the properties of divisors, we can demonstrate that for all integers $a, b, k, m \in \mathbb{Z}$, if $m|a$ and $m|b$ and every divisor k of a and b also divides m, then $|m|$ is the greatest common divisor of a and b. This gives us an unexpected way to obtain a *greatest* common divisor by taking a *smallest* element, as demonstrated in the next theorem.

Theorem 7.3.3. *For all integers $p, q \in \mathbb{Z}$, not both zero, define the set of integer combinations $S = \{qc - pd \mid c, d \in Z\}$. The smallest positive element in S is the greatest common divisor of p and q.*

Proof sketch. Let's name the greatest common divisor of p and q: $m = gcd(p, q)$. Note that all elements of S are multiples of m, since if $m|p$ and $m|q$ then $m|(qc - pd)$.

There exist integers $a, b \in \mathbb{Z}$ so that $p = am$ and $q = bm$ with a and b relatively prime. Lemma 7.3.2, says that there are $c, d \in Z$ with $bc - ad = 1$. Multiplying both sides by m, we find that $qc - pd = m$, so m is in S. ☙

Remember this property of necklaces used in the proof of Fermat's little theorem (Theorem 7.2.2): for all primes p, each p-bead necklace with at least two colors generates p different straight sequences? We can now prove this.

Corollary 7.3.4. *Let p be a prime. If the number of sequences obtained by cutting a given p-bead necklace is less than p, then all beads are the same color.*

Proof. If we have fewer than p sequences from p cuts, then there must be two cuts shifted $0 < q < p$ apart that give the same sequence. Note that $gcd(p, q) = 1$.

Identify some bead as 0. The beads at any integer multiple of q, counting around the necklace, must be the same color as bead 0. Theorem 7.3.3 says that there are integers $c, d \in \mathbb{Z}$ with $qc = pd + 1$. This means that bead $qc \bmod p = 1$ is an integer multiple of q from 0. And then so is every other bead. Either we get p sequences or all beads are the same color. ☙

7.3.4 Encryption

Recall that we want to use our $N = pq$ with its hidden primes to find exponents $E > 1$ and D that for all $m \in [0, N)$, $m^{ED} \bmod N = m$.

Choose an E that is relatively prime to $(p-1)(q-1)$, then use Lemma 7.3.2 one last time to find $D, k \in \mathbb{Z}$ such that $ED = (p-1)(q-1)k + 1$. Fermat's little theorem shows that this is the appropriate magic.

Corollary 7.3.5. *Let p and q be distinct primes and $N = pq$. For all integers k and $m \in [0, N)$, we have $m^{(p-1)(q-1)k+1} \bmod N = m \bmod N$.*

Proof. It is enough to check that $p|(m^{(p-1)(q-1)k+1} - m)$, since the proof for q will be symmetric and if both divide a number, then so does their product by a property of subsection 7.2.2.

If $p|m$ then we are done, so assume that p and m are relatively prime. In this case we can write Fermat's little theorem as $m^{p-1} \bmod p = 1$, and $m \cdot (m^{p-1} \bmod p)^{(q-1)k} \bmod p = m \cdot 1^{(q-1)k} \bmod p = m \bmod p$. ∎

For the example of table 7.2, I chose primes $p = 101$, $q = 103$ to get $N = 10403$, and chose $E = 11 \cdot 19 = 209$, which does not divide $(p-1)(q-1) = 10200$. Then I solved $ED - (p-1)(q-1)m = 1$ for integers D, m to get $D = 2489$, since $ED - 1 = 520200 = 51 \cdot (p-1)(q-1)$. This lets anyone encode, and me decode, 4-digit messages: $encode(1337) = 8105$ and $decode(8105) = 1337$; $encode(42) = 7098$ and $decode(7098) = 42$.

7.4 Summary

Using public-key cryptography as a motivating problem, this chapter reviewed properties of mathematical functions and objects that should mostly be familiar, if you look at any one thing. On the other hand, this chapter is intended to be a little overwhelming, because it brings together so many familiar things and views them from unfamiliar perspectives.

log and exp These are inverses, since for $a, b > 0$, if we have $a = b^c$, then $c = \log_b a$. The product rule for counting is one reason these are important.

floor and ceiling These operations round reals down or up to the next integer. Memory address calculations on the computer tend to use floor, rounding fractions down.

mod Modulo is an operation on integers that returns the remainder on division; its definition as $x \bmod y = x - \lfloor x/y \rfloor \cdot y$ assigns a specific result to all x and all positive or negative y; only $y = 0$ is forbidden.

primes The prime numbers, studied as compelling mathematical curiosities, ironically are applied to send encrypted communication on earth, and to send what is hoped to be clear communication to the stars.

fractions Even the well-known fractions have surprising properties when we look at them in new ways.

vectors Vectors are tuples (pairs, in this case) with additional operations of addition and scalar multiplication. Here they demonstrate that mathematical objects can be viewed from different perspectives to gain different insights.

In addition to reviewing the mathematics that computer science relies on, this chapter aims to demonstrate that the ability of mathematics to advance computer science requires, on one hand, formal definitions and precise reasoning, and on the other, creative understanding and insightful application. Ours is a deep and wonderful discipline.

This chapter continues a trend toward more formal proof, which culminates in chapters 9 and 10.

7.5 Exercises and Explorations

Quiz Prep 7.1. What are the values of these expressions?

? $\lfloor 3.9 \rfloor$	? $\lfloor -3.9 \rfloor$	? $42 \bmod 4$	? $gcd(20, 36)$	? 2^{10}	? $\log_2 32$
? $\lfloor 4.0 \rfloor$	? $\lfloor -4.0 \rfloor$	? $3 \bmod 11$	? $gcd(64, 65)$	? $(2^5)^2$	? $\log_3 9$
? $\lfloor 4.1 \rfloor$	? $\lfloor -4.1 \rfloor$	? $25 \bmod 5$	? $gcd(1, 9)$	? $2^{(3^2)}$	? $13^{\log_{13} 7}$

*Warning: incorrect statements in this problem!

Exercise 7.2. Find the mistake(s) in each of the following. *

1. In reasoning about a number d that is a multiple of 9, a student writes, "$9|d = \lfloor d/9 \rfloor \ldots$".
2. The negation of "n is not divisible by any prime number between 1 and $\sqrt{n}$" is "n is divisible by any prime number between 1 and $\sqrt{n}$."

▶

Exercise 7.3. Use definitions of floor and ceiling to establish these properties of floor and ceiling for $x, y \in \mathbb{R}$ and $N \in \mathbb{Z}$. Spelling out the definition of floor as the largest integer less than or equal to x, we see $\forall_{x \in \mathbb{R}}$ that $m = \lfloor x \rfloor$ iff $m \in \mathbb{Z}$ and $m \leq x$ and $m + 1 > x$.

1. First, show that $x - 1 < \lfloor x \rfloor \leq x$ and $x + 1 > \lceil x \rceil \geq x$, because these inequalities can help you establish the other properties.
2. Show that $\lfloor x \rfloor + \lceil -x \rceil = 0$.
3. Show that $\lceil \lceil x \rceil \rceil = \lceil x \rceil$.
4. Show that $\lfloor x + N \rfloor = \lfloor x \rfloor + N$.
5. Show that if $x < y$ then $\lfloor x \rfloor \leq \lfloor y \rfloor$.
6. Show that $\lfloor x \rfloor \leq \lceil x \rceil$, with equality iff x is an integer.
7. Show that $N = \lfloor N/2 \rfloor + \lceil N/2 \rceil$.
8. Show that $\lfloor (N + 1)/2 \rfloor = \lceil N/2 \rceil$.

Exercise 7.4. Partition a set of cardinality n (i.e., n elements) as evenly as possible into k subsets. Use floor, ceiling, and mod to say how many sets you get of what cardinalities. ▶

Exercise 7.5. $x - \lfloor x \rfloor$ is the *fractional value of* x and $x - \text{sgn}(x)\lfloor |x| \rfloor$ is the *fractional part of* x. Explain what each of these is in words. Use parallel language to make the similarities and differences stand out clearly. ▶

Exercise 7.6. Given positive reals $b, x \in \mathbb{R}^+$, express $\log_{1/b} x$ using logs of base b.

Exercise 7.7. Given positive reals $a, b \in \mathbb{R}^+$, express $\log_b a$ using logs of base a.

Exercise 7.8.

1. Give a combinatorial proof of the subcommittee identity $\binom{n}{k}\binom{k}{j} = \binom{n}{j}\binom{n-j}{k-j}$.

2. Show that, except for the ones, any two numbers from a row of Pascal's triangle have a common factor. That is, show for integers $0 < r, s < n$, that $gcd(\binom{n}{r}, \binom{n}{s}) > 1$.

Hint:

▶

Exercise 7.9. Let $p(x) = \sum_{0 \le i \le n} a_i x^i$ be a polynomial with integer coefficients $(a_1, a_2, \ldots, a_n)$ and degree greater than one. That is, $n \ge 1$ and $a_n \neq 0$. Show that there exists a non-negative integer k so that $p(k)$ is not prime.

▶

Exercise 7.10. On an infinite chessboard, a generalized knight moves by jumping p squares in one direction and q squares in a perpendicular direction, for given $p, q > 0$. Show that a knight needs an even number of moves to return to its starting position.

Hint:

Hint:

Exercise 7.11. Prove this easy upside-down version of Fermat's last theorem: There are no positive integers $n > 2$, x, y, and z that satisfy $n^x + n^y = n^z$.

Puzzle 7.12. From MAA Minute Math, via cut-the-knot: Call a number *prime-looking* if it is composite but not divisible by 2, 3, or 5. The three smallest prime-looking numbers are 49, 77, and 91. There are 168 prime numbers less than 1000. How many prime-looking numbers are there less than 1000?

Extension 7.13. There is a subtle difference between the number of k-tuples from a set of size b and the number of k-digit numbers in base b: we usually don't allow 0 as a leading digit. What are the following counts?

1. The number of k-tuples from a set of size b.

2. The number of i-tuples from a set of size b, for all $i \in [0..k]$.

3. The number of k-digit non-negative integers in base b.

4. The number of i-digit non-negative integers in base b, for all $i \in [0..k]$.

Extension 7.14. Lemma 7.3.1 dealt only with positive fractions; extend it to negative fractions as well.

Extension 7.15.

1. Convert 42 to base 5, 3, and 2.

2. Explain how to convert any base 2 number to base 4, 8, or 16, which use the digits 0-3, 0-7, and 0-9ABCDEF, respectively.

3. Create a spreadsheet that takes a decimal input number and a base, then expresses the digit sequence that represents that number in that base. Recall that the digits used in base b are $0 \ldots (b-1)$.

4. In your spreadsheet, convert messages in a given alphabet to numbers and back. In Excel, you can enter a chosen alphabet of symbols in a cell, say A1, (single quote if you want to start with space or enter just digits) and use functions =len(A1) for length, =find(B1,A1)-1 to find the number of the letter/digit in B1, starting from zero, and =mid(A1, C1+1,1) to turn the number in C1, which must be in $0 \leq C1 < \text{len(A1)}$, into the corresponding letter/digit. Your output can be one letter per cell, or can use the =replace function to collect output into a single string.

Hint:

Extension 7.16. For these questions, let $[x]$ denote the *fractional part of* x: that is, $[x] = x - \lfloor x \rfloor$. We are going to look at the set of all fractional parts, $F_a = \{[na] \mid n \in \mathbb{N}\}$. Show the following:

1. For any irrational number a, the elements we put into set F_a are distinct. That is, for all positive integers $m, n \in \mathbb{N}$, we have $[ma] = [na]$ iff $m = n$.

2. For any irrational number a, the set $\inf F_a = 0$. That is, for any $\epsilon > 0$, there is an $n \in \mathbb{N}$ with $[na] < \epsilon$.

3. For any irrational number a, the set F_a is *dense* in the interval $[0, 1]$. That is, for any reals $x \in [0, 1]$ and $\epsilon > 0$, there is an $n \in N$ such that $|x - [na]| < \epsilon$.

▶

Exploration 7.17. In your favorite programming language, or even your favorite spreadsheet, implement an algorithm that takes positive integers N, e, and $m < N$, and produces the encoded message, $m^e \bmod N$. The same algorithm does the decoding. Work out e, d, and N to demonstrate this.

Exploration 7.18. Experiment with using Fermat's little theorem to test for primes and Carmichael numbers. Determine, for some range of integers, how many m values you pick before you successfully detect a composite, and how many numbers that you pass as prime are actually not.

Exploration 7.19. How many simple substitution cyphers on [a..z]? A simple substitution cypher replaces each letter a–z with some other letter in an invertible manner – it is a bijection from a–z to a–z. You may have seen Cryptoquote in the newspaper, for which the standard example is AXYDLBAAXR = LONGFELLOW (i.e., A→L and X→O.)

1. How many different simple substitution cyphers are there?

2. Suppose that we want to avoid sending a letter to itself, so we replace letters from the first half (a–m) with letters from the second half (n–z) and vice versa. With this restriction, how many different simple substitution cyphers are there?

3. Counting the number of substitution cyphers in which no letter maps to itself is not easy, but you can take a swing at that, too.

▶

Chapter 8
Recursive definition of structures

Recursive loop:
See recursive loop.
—Borland Pascal Language Guide (1992)

Recursion* enables us to compactly and unambiguously specify arbitrarily large structures and sets so that even a mindless computer can follow the specification. The aim of this chapter is to eliminate ellipses (that is, ". . . "s) from definitions. chapter 10 will eliminate them from proofs.

*Google search: recursion

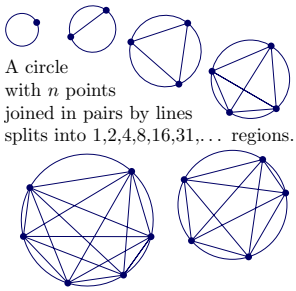
Objectives: After working through this section, you will be able to write recursive definitions for sets, relations, functions, and sequences. You will be able to identify elements that are in or not in sets that have been recursively defined, and to explain when and why the closure rule is needed. You will be able to concatenate strings, languages, and lists, and use Kleene-star to create Σ^*, the language of all strings on an alphabet Σ, or the set of all lists.

You will begin to appreciate how a data structure like a list can be implemented from its recursive definition, which is the basis for programming languages like ML or Haskell.

8.1 Recursive definition. . .

Recursive definition is especially important to computer science, because it tends to translate directly into construction and into rules for testing the validity of a construction.

8.1.1 . . . of sets


A circle with n points joined in pairs by lines splits into 1,2,4,8,16,31,. . . regions.

In subsection 3.1.1 we gave examples of sets defined by a list, a pattern, or a rule. Why do we need something else?

Infinite sets are a problem: I cannot list their elements in my finite lifetime, a rule can produce an infinite set only if it starts with some infinite set, and patterns are subject to interpretation. $\{1, 2, 4, 8, 16, \ldots\}$ seems to have an obvious pattern, until I tell you that the next number is 31.

To get around this, we make recursive definitions, which have a small number of rules that are easily checkable, but that can be applied an arbitrarily large number of times. As a computer scientist, I tend to think of a recursive definition as a computer program that can write out the reason for including an element in a set.

A *recursive definition* of a set S needs three things:

1. Base cases (one or more) that define initial elements of the set S,
2. Recursive rules (one or more) that tell how to generate additional elements of S from known elements of S, and
3. A closure rule, saying that the only elements of S are those that are generated from the base cases by a finite number of applications (possibly zero) of the recursive rules.

For example, the *natural numbers* $\mathbb{N}$, which were previously defined by pattern, can be more formally defined recursively. In this definition $s()$ stands for *successor*, which you can think of as $s(n) = n + 1$, as ++ from C/Java, or as a long unevaluated string so that the representation of 1 is $s(0)$, 2 is $s(s(0))$, etc.

1. Base: $0 \in \mathbb{N}$.
2. Recursive rule: If $n \in \mathbb{N}$, then the successor $s(n) \in \mathbb{N}$.
3. Closure: The only numbers in $\mathbb{N}$ are those generated from the base case by a finite number of applications of the recursive rule.

Is $\boxed{0 \in \mathbb{N}?}$ Is $\boxed{-1 \in \mathbb{N}?}$ Is $\boxed{42 \in \mathbb{N}?}$ Is $\boxed{\infty \in \mathbb{N}?}$ Is $\boxed{\pi \in \mathbb{N}?}$ Why do we need the Closure rule?

The set S of all multiples of 5 is better defined by a rule $S = \{5n \mid n \in \mathbb{Z}\}$, but can be defined recursively:

1. Base: $5 \in S$.
2. Recursive rule: If $a, b \in S$, then $a + b \in S$ and $a - b \in S$.
3. Closure: Only the numbers generated from the base case by a finite number of applications of the recursive rule are in S.

To prove that these two different definitions define the same sets, we will need the technique of mathematical induction from chapter 10.

Recursive definition is most useful when defining sets of other structures. For example, in propositional logic, if we pick a set of logic variables, we can recursively define the set of all possible fully-parenthesized formulas on these variables. These are the *well-formed formulae*, or 'WFF's:*

1. Base: Any isolated variable, or a constant 'true' or 'false' is a WFF.
2. Recursive Rule: If α and β are WFFs, then $\overline{\alpha}$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \to \beta)$, $(\alpha \leftrightarrow \beta)$, and $(\alpha \oplus \beta)$ are all WFFs.

*Example WFFs on $\{p, q, r\}$: p, $\overline{p}$, $(p \wedge q)$, $(p \vee \text{false})$, $(p \leftrightarrow (q \vee r))$, $((\overline{p} \wedge \overline{q}) \leftrightarrow \overline{(p \vee q)})$.
Non WFFs: $\overline{\vee}$, $(\wedge q)$, $(p \leftrightarrow q \vee r)$.

3. Closure: The only WFFs are the formulae that are defined from the base case by a finite number of applications of the recursive rule.

Note that the WFFs generated by this recursive definition have their full set of parentheses. An expression like $(p \leftrightarrow q \rightarrow r)$ is not a WFF, and is ambiguous, unless you've defined an operator precidence order, since the two ways to add parentheses to make a WFF, $((p \leftrightarrow q) \rightarrow r)$ or $(p \leftrightarrow (q \rightarrow r))$, are not logically equivalent.

8.1.2 ... of functions and relations

Functions whose domains sets are recursively defined are often best defined recursively. For example, the *factorial function* on $\mathbb{N}$ is defined formally as:

1. Base: $0! = 1$.
2. Rec. rule: for integers $n > 0$, let $n! = (n-1)! \cdot n$.

The function $F\colon \mathbb{N} \to \mathbb{N}$ for the *Fibonacci numbers* needs two base cases to get started.

It is not true that the original Italian spelling of 'Fibonacci' was 'Fibbooonnnnnaaaaaaaaaccccccccccccciiiiiiiiiiiiiiiiii.'

1. Base: $F(0) = 0$, $F(1) = 1$.
2. Rec. rule: for integers $n > 1$, define $F(n) = F(n-1) + F(n-2)$.

Unlike sets, functions do not need an explicit statement of closure. The base and rule must already define exactly one result for each element of the domain; if we tried to add in anything extra, then we would no longer have a function.

Recursive definitions of numerical functions, called *recurrences*, are often used in counting. Here are recurrences for two functions that we have seen before. We'll close the chapter with other examples that give a taste of the power of recurrences.

The *choose* function $C(n, r)$, which I continue to write using binomial coefficient $\binom{n}{r}$, can be defined with a two-parameter recurrence.

1. Base: $\binom{0}{0} = 1$. For integers n, r with $n < 0$, $r < 0$, or $r > n$, let $\binom{n}{r} = 0$.
2. Rec. rule: for integers $n > 0$ and $0 \le r \le n$, define $\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$.

We can write a recurrence to count surjections, which are the functions $f\colon A \to B$ that hit every element of the range, so $f(A) = B$. Define $S(m, n)$ as the number of surjections from domain $A = [1..m]$ to range B with n elements, and check that the following agrees with the counts in table 6.2. The justifications are not part of the definition, but can be turned into a proof that the resulting counts are correct, once we have the technique of mathematical induction.

1. Base: For all $m < n$, $S(m, n) = 0$ because of the pigeonhole principle.
   $S(0, 0) = 1$.
2. Rec. rule: for all $m \geq n > 0$, $S(m, n) = n(S(m-1, n) + S(m-1, n-1))$, because you can make a surjection by selecting the value $f(m)$ from n possibilities in the range, then either choose the rest of $f : [1..m-1] \to B$ as a surjection onto the entire n-element range B in $S(m-1, n)$ ways, or remove $f(m)$ from B and choose a surjection onto the remaining $n-1$ elements in $S(m-1, n-1)$ ways.

The factorial and Fibonacci definitions used addition and multiplication for $\mathbb{N}$. We can actually define those based only on successor. Here is addition $add(a, b)$

1. Base: for all $b \in \mathbb{N}$, let $add(0, b) = b$.
2. Rec. rule: for all $s(a), b \in \mathbb{N}$, let $add(s(a), b) = s(add(a, b))$.*

*That is, $(a + 1) + b = (a + b) + 1$.

And here is multiplication $mult(a, b)$

1. Base: for all $b \in \mathbb{N}$, let $mult(0, b) = 0$
2. Rec. rule: for all $s(a), b \in \mathbb{N}$, let $mult(s(a), b) = add(mult(a, b), b)$.†

†That is, $(a + 1)b = (ab) + b$.

We can also define relations, such as 'less than.' Recall that a binary relation on A is a subset of $A \times A$; to define a set of pairs, we do need a closure rule.

1. Base: $0 < s(0)$.
2. Recursive rule: If $a < b$, then $s(a) < s(b)$ and $a < s(b)$.
3. Closure: Only the pairs generated from the base case by a finite number of applications of the recursive rules are in the relation $<$.

Mathematicians like to define a family of operations on a simple function like successor and then derive all the basic properties of arithmetic from some simple definitions. Then any system that satisfies the definitions can be thought of as an arithmetic. In computer science we may be happy to accept the number system, but it becomes very useful to define data structures formally and precisely with few rules, as we do in the next sections.

8.1.3 ... of tuples, lists, and sequences

In subsection 3.1.2 we defined ordered pairs on a set A using Cartesian product, $A \times A$. To formally extend this to the sets of all k-tuples, A^k for $k \geq 0$, we use recursive definition:

1. Base: $A^0 = \{( )\}$ is the single 0-tuple.
2. Recursive Rule: for $k > 0$, the set of k-tuples $A^k = A^{k-1} \times A$.

A sequence of numbers or sets like this defines a unique value for each k. Like a function, it does not need an explicit statement of closure.

As was mentioned in subsection 3.1.3, several structures are extensions of tuples. We use the terms *list* when we speak about tuples that may have arbitrary finite length but more limited access, and *sequence* when that length may even be infinite.‡ A *string* is a tuple of letters from some *alphabet* set Σ, as we define in the next subsection.

‡A *series* is a sum over a sequence, replacing commas by plus signs.

Parentheses are often used for lists and sequences, but since that may lead to confusion with tuples, I use angle brackets for this section.

Let's focus on the set L of all possible lists of elements from A. We could say that L is just the union of all sets of tuples, $L = \bigcup_{k\geq 0} A^k$. A recursive definition can give a more explicit construction of L. Here I use a common notation from functional programming that originates with Lisp: the *list constructor* is a function $cons\colon A \times L \to L$ that takes an element and a list. It returns a new list with the element added to the front or *head* of the old list.

The set of all lists L is defined:

1. Base: The *empty list* $\langle\,\rangle \in L$.
2. Rec. Rule: For any element $a \in A$ and list $\ell \in L$, $cons(a, \ell)$ is a list in L.
3. Closure: Only lists generated from 1 by a finite number of applications of rule 2 are in L.

As with successor, we can imagine leaving the *cons* unevaluated, so the list $\langle a, b\rangle$ would be represented $cons(a, cons(b, \langle\,\rangle))$. Two partial functions $head\colon L \to A$ and $tail\colon L \to L$ are defined on every list but the empty list:* $head(cons(a, \ell)) = a$ and $tail(cons(a, \ell)) = \ell$. *Head* and *tail* of the empty list $\langle\,\rangle$ are undefined. Two given lists α and β are equal iff both are empty, or neither are empty, $head(\alpha) = head(\beta)$, and recursively $tail(\alpha) = tail(\beta)$.

*What are $head(\langle a, b, c\rangle)$ and $tail(\langle a, b, c\rangle)$?

Can you give recursive definitions for $length\colon L \to \mathbb{N}$, which returns the number of elements in a list, and *projection* $\pi_i\colon L \to A$, which for a list ℓ and $i \in [1..length(\ell)]$ returns the element in position i?

Defining $length\colon L \to \mathbb{N}$:

1. Base:
2. Rec. rule: For all ,

Defining projection $\pi_i\colon L \to A$:

1. Base: For all $\beta \in L$, define $\pi_1(\beta) =$).
2. Rec. rule: For all , .

A slower operation on lists is *concatenation*. The function $concat\colon L \times L \to L$, joins two input lists of length m and n into one list of length $m + n$ by

putting elements of the first list before those of the second. Notice how similar its recursive definition is to the definitions of addition and multiplication.

1. Base: for all $\beta \in L$, let $concat(\langle\,\rangle, \beta) = \beta$.
2. Rec. rule: For all elements $a \in A$ and lists $\beta, \gamma \in L$, we define concatenation as $concat(cons(a, \beta), \gamma) = cons(a, concat(\beta, \gamma))$.*

*That is, $(a\beta)\gamma = a(\beta\gamma)$.

We could define a function $rev\colon L \to L$ to reverse a list using *concat*:

1. Base: $rev(\langle\,\rangle) = \langle\,\rangle$.
2. Rec. rule: $\forall_{a\in A, \beta\in L}\ rev(cons(a, \beta)) = concat(rev(\beta), cons(a, \langle\,\rangle))$.

A better way is to give this function two input lists, $rev2\colon L^2 \to L$, called initially as $rev2(\beta, \langle\,\rangle)$.

1. Base: For all $\gamma \in L$, $rev2(\langle\,\rangle, \gamma) = \gamma$.
2. Rec. rule: $\forall_{a\in A,\ \beta,\gamma\in L}$, let $rev2(cons(a, \beta), \gamma) = rev2(\beta, cons(a, \gamma))$.

Adding this second *accumulator* argument is a general transformation applied in functional programming languages like ML and Haskell to make functions more efficient. Transforming the definition ensures efficiency of all implementations derived from the definition.

Sequences are like functions whose domain is $\mathbb{N}$. The Fibonacci numbers, for example, are commonly defined as a sequence rather than a function. Recursive definitions of sequences are often called *recurrences* or *recurrence relations*.

1. Base: $F_0 = 0$, $F_1 = 1$.
2. Rec. rule: for integers $n > 1$, define $F_n = F_{n-1} + F_{n-2}$.

8.1.4 . . . of notation for operations

As introduced in subsection 3.1.3, we use the Greek capital *Sigma* to stand for summation: $\sum_{1\le i} s_i = s_1 + s_2 + s_3 + \cdots$ adds the terms for every integer index i in the set or condition written below the Sigma. Some write sums with upper and lower index limits, like $\sum_{i=1}^{n} s_i$, but specifying indices by a set or by logical conditions, like $\sum_{i\in[1..n]} s_i$, gives flexibility—for example, we can easily sum over evens or primes.

This 'big operator' notation extends to other operators that apply to sets, lists, and sequences. We already used Greek capital Pi $\prod$ for product to define *factorial*†, $n! = \prod_{1\le i\le n} i$, and big 'and' and 'or' in defining the quantifiers. We use big operators for union and intersections, and for maximum and minimum by just subscripting 'max' or 'min.' Disjoint union $\uplus$ combines elements of non-overlapping sets; we used it to define a *partition of a set* in subsection 5.3.1.

†Look! No ellipses (. . .)!

To formally define Sigma notation, especially for infinite sequences, let me start with the operation of *prefix sum*, which actually produces a sequence of partial sums; these sequences are useful in their own right in section 8.2.

Prefix sum: Given a sequence $s_1, s_2, s_3, \ldots$, the *prefix sum* is the series of sums of the first k terms, for $k \geq 0$: using Sigma summation notation, $S_k = \sum_{1 \leq i \leq k} s_i$. Using a recursive definition,

1. Base: $S_0 = 0$.
2. Rec. rule: for integers $k > 0$, define $S_k = S_{k-1} + s_k$.

Now, for a finite sum, $\sum_{1 \leq i \leq n} s_i = S_n$, the final prefix sum. An infinite sum is defined to be the limit of the prefix sum values, if that limit exists. That limit may be infinite, e.g., for $\sum_{i \geq 1} 1$, or may not exist, e.g., for $\sum_{i \geq 1} (-1)^i$.

All other "big operator" notation, including and $\bigwedge$, or $\bigvee$, product $\prod$, minimum min, maximum max, union $\bigcup$, intersection $\bigcap$, disjoint union $\biguplus$, can be defined recursively by the same pattern. The base case begins with the appropriate *identity*, such as 1 for product, T for and, F for or, $\emptyset$ for union, U for intersection, and each term is combined with the previous prefix value.

8.1.5 ... of strings and languages

As defined in subsection 3.1.3, *strings* are lists containing characters from a given *alphabet**, Σ. The set of all possible strings is denoted $\Sigma^* = \bigcup_{i \geq 0} \Sigma^i$, which is recursively defined just below.

*Warning: Σ now a set.

We write strings without parentheses or brackets, and often name them with lowercase Greek letters like α, β, γ, μ, ν, σ, or τ. The *empty string* is commonly denoted by Λ.

We can apply all the operations for lists to strings. The main one is concatenation, so, for all $\alpha, \beta \in \Sigma^*$, we abbreviate $concat(\alpha, \beta)$ as simply $\alpha\beta$. Concatenation with the empty string does nothing: $\alpha\Lambda = \Lambda\alpha = \alpha$.

A *language* is a set of strings: $L \subseteq \Sigma^*$. We extend the definition of concatenation from strings to languages L and M by concatenating all possible pairs of strings: $LM = \{\alpha\beta \mid \alpha \in L, \beta \in M\}$. This is subtly different from Cartesian product, which is applied only to sets of tuples, because not all strings need to have the same length.† We get only part of the product rule: $|LM| \leq |L| \cdot |M|$. Can you find two sets of strings whose product $|LM| < |L| \cdot |M|$?

†We consider a similar problem in detail in subsection 9.2.1.

The operation of *Kleene-star* accumulates elements to form a set. Given a set, list, set of lists, alphabet, or language S,

1. Base: The empty list or empty string, as appropriate, is in S^*
2. Recursive Rule: For all $\alpha \in S^*$, then the concatenation $\{\alpha\}S \subseteq S^*$.
3. Closure: Only lists or strings generated from 1 by a finite number of applications of rule 2 are in S^*.

For example, with alphabet $\Sigma = \{b, c\}$: Is $\boxed{\Lambda \in \Sigma^*}$? Is $\boxed{b \in \Sigma^*}$? Is $\boxed{bc \in \Sigma^*}$? Is $\boxed{cc \in \Sigma^*}$? Is $\boxed{abc \in \Sigma^*}$?

Strings and languages are important topics in parsers, compilers and formal languages; we will see more of them in chapter 12.

8.1.6 . . . of other structures

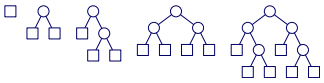


Figure 8.1: Trees

There are many other structures that are defined recursively. An important example in computer science is the set of all *binary trees*, T. Each binary tree comes with a pointer to its *root*, which is also defined in this recursive definition:

1. Base: $\square$ denotes an empty tree with no vertices; the root pointer can be *null*.
2. Recursive Rule: For any two binary trees with roots $\ell, r \in T$, the tree with root $\bigcirc$, left child ℓ, and right child r is also in T.
3. Closure: Nothing is in T except what is generated from 1 by a finite number of applications of rule 2.

Are each of the graphs depicted in figure 8.1 in T?

Let's partition trees by height: Define $\mathcal{T} = \{T_0, T_1, \ldots\}$ to be a family of sets of rooted binary trees, where each set T_i contains all the *trees of height i*. We modify the previous definition to define this family recursively:

1. Base: The empty tree with root $\square \in T_0$.

2a. Rec. Rule: For any two binary trees with roots $\ell \in T_i$ and $r \in T_j$, for $j \leq i$, the tree with root $\bigcirc$ whose left child is ℓ and right child is r is in T_{i+1}.

2b. Rec. Rule: For any two binary trees with roots $\ell \in T_j$ and $r \in T_i$, for $j \leq i$, the tree with root $\bigcirc$ whose left child is ℓ and right child is r is in T_{i+1}.

3. Closure: The only elements of T_i and sets in the family $\mathcal{T}$ are those generated from 1 by a finite number of applications of rules 2a and 2b.

What are the heights of the trees in figure 8.1? What would happen if I left out $\boxed{\text{rule 2b}}$?

As we shall see in chapter 13, trees are generally defined as connected, acyclic graphs. We explore there whether the different definitions really define the same set of structures.

8.2 Recurrences, series, and counting

I close this chapter with several examples of working with recursively defined sequences, or *recursions*. This is a grab bag of techniques applied to recursive

functions to derive results that will be useful in other chapters.

The first solves a counting puzzle by recognizing a sequence of numbers and showing that the recursion defining the sequence fits the puzzle, the second just forms a recursion for a new sequence. The examples after that analyze recursively defined sequences and series to find *closed form* expressions, either for exact results or for upper and lower bounds. (An expression is said to be in closed form if it can be calculated using a fixed number of mathematical operations (addition, multiplication, exponentiation), not dependent on the inputs. Summations and recursive functions are not closed form, because the number of calculations that they represent depend on their inputs.) The final two are examples of how calculus ideas come into discrete mathematics, too.

8.2.1 Combinatorial proof of a numerical identity

Here is a puzzle to illustrate the benefit of counting in two ways. Let's say that a set $S \subseteq [1..n]$ *counts itself* iff its smallest element is its size: $|S| = \min(S)$. How many sets count themselves for each value of n?

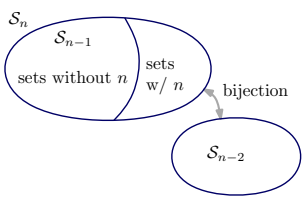
n	s_n	sets
0	0	
1	1	
2	1	
3	2	
4	3	
5	5	

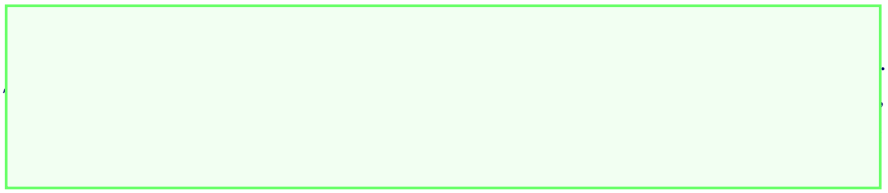
Scratch paper time. What are some examples of set that count themselves? Well, $\{1\}$, $\{2, b\}$ for any $b > 2$, $\{3, c, d\}$ for any $d > c > 3$, . . . It appears that we can create a self-counting subset of $[1..n]$ by choosing the minimum, k, and then choosing $k-1$ numbers from the $n-k$ numbers greater than k. Thus, the number of sets of size n that count themselves is $s_n = \sum_{1 \le k \le n} \binom{n-k}{k-1}$. (I could stop this *summation index* halfway, but the terms with $n - k < k - 1$ are zero by convention.)

*Cool tool: Online encyclopedia of integer sequences lets you look up sequences to find out what might create them.

Let's check that by listing in the margin the self-counting sets for $n \in [0..5]$, omitting braces to save space. The numbers of those sets, 0,1,1,2,3,5, look familiar.* Is there any reason we might expect self-counting sets to give us Fibonacci numbers? Can you explain why s_n might equal $s_{n-1} + s_{n-2}$ for $n > 1$?

Let's use $\mathcal{S}_n$ to denote the family of all self-counting subsets of $[1..n]$, so $s_n = |\mathcal{S}_n|$. Notice that $\mathcal{S}_{n-1}$ is the family of the sets of $\mathcal{S}_n$ that do not include n, so there are s_{n-1} of those. How many sets are in the difference, $\mathcal{S}_n \setminus \mathcal{S}_{n-1}$? That is, how many self-counting subsets of $[1..n]$ are not subsets of $[1..n-1]$?





By counting in two ways, we learn that the binomial coefficient sums $s_n = \sum_{1 \le k \le n} \binom{n-k}{k-1}$ are Fibonacci numbers.

8.2.2 Counting partitions

*Recall: A partition chops a set into a family of non-empty subsets so each element is in exactly one subset.

Not every count can be expressed in closed form. How many different partitions* can we have for the set $[1..n]$? Let's begin by looking at small examples in table 8.1. (I can even include $\emptyset$, which has one partition, namely the empty set itself. Strange as it seems, it fits the definition, and we will use it below.) The number of partitions grows quickly, so let's invent notation to write them compactly, e.g., writing $\{\{1,3\},\{2,4,5\}\}$ as $13 \mid 245$.

Table 8.1: Recursive count of all partitions of sets $[1..n]$ for $0 \le n \le 5$. A partition $\{\{1,3\},\{2,4,5\}\}$ is written compactly as $13 \mid 245$. For $n = 5$ I create notation to track the ways to group some of $[1..4]$ with 5, and partition the rest.

set	#	partitions
$\emptyset$	1	$\emptyset$
$\{1\}$	1	$\{1\}$
$[1..2]$	2	$\{\{1\},\{2\}\}$ and $\{\{1,2\}\}$ abbreviated $1\|2, 12$
$[1..3]$	5	$1\|2\|3, 12\|3, 1\|23, 2\|13, 123$
$[1..4]$	15	$1\|2\|3\|4, 12\|3\|4, 1\|23\|4, 2\|13\|4, 123\|4,$ (adds $\|4$ to above)
		$1\|2\|34, 12\|34, 1\|3\|24, 13\|24, 2\|3\|14, 23\|14,$ (one # with 4)
		$1\|234, 2\|134, 3\|124, 1234$ (two or all with 4)
$[1..5]$	52	$15{:}\ [1..4]\|5,\ 4\cdot 5{:}\ [abc]\|d5,\ 6\cdot 2{:}\ [ab]\|..5,\ 4{:}\ a\|..5,\ 12345$
		that is, $1\|2\|3\|4\|5, \ldots$ 50 more $\ldots, 12345$

†B_n for Bell numbers.

Let B_n denote the number† of partitions of the set $[1..n]$.

For the base case: $B_0 = 1$. For the general case, making the partitions for $[1..4]$ suggests that we count the partitions of $[1..(n+1)]$ by considering how many elements are in a set with $(n+1)$. Actually, let's use k as the number of elements *not* in a set with $(n+1)$. We can remove k elements $\binom{n}{k}$ ways, leaving a set that contains the element $(n+1)$. Independently, we can partition the k removed elements in B_k ways. Thus, we can count partitions recursively by $B_{n+1} = \sum_{0 \le k \le n} \binom{n}{k} B_k$, with base case $B_0 = 1$. This does not have a closed form expression.

8.2.3 Mathematical series

Recall that you get a series from a sequence by just summing the elements. There are several series that you should know. I find it easier to remember the key idea to derive these formulae, rather than to memorize the formulae themselves.

Let me start with the operation of *prefix sum*, which actually produces a sequence of partial sums. Prefix sum will help us analyze all other series that we look at: sums of consecutive integers, geometric series, harmonic numbers, and base conversion.

- Prefix sum: Given a sequence $s_1, s_2, s_3, \ldots$, the *prefix sum* is the series of sums of the first k terms, for $k \geq 0$: using Sigma summation notation, $S_k = \sum_{1 \leq i \leq k} s_i$. Using a recursive definition,
  1. Base: $S_0 = 0$.
  2. Rec. rule: for integers $k > 0$, define $S_k = S_{k-1} + s_k$.

  From calculus, you learned that an infinite series (sum of an infinite sequence) has a value if the prefix sums converge to a limit.

- Summing the first n integers:

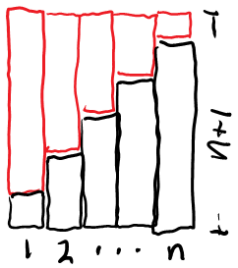
$$S_n = \sum_{1 \leq i \leq n} i = \frac{n(n+1)}{2}.$$

  Add up the numbers, once forward and once in reverse order, then divide by 2.

$$\begin{aligned} S_n &= 1 + \quad 2 \quad + \cdots + (n-1) + n \\ + \quad S_n &= n + (n-1) + \cdots + \quad 2 \quad + \quad 1 \\ \hline 2S_n &= n(n+1) \end{aligned}$$

  So, $S_n = n(n+1)/2$.

Wordless sum 1 to n:



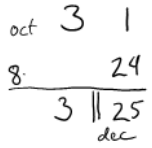
An infinite number of mathematicians walk into a bar. The first orders a beer, the second a half, the third a quarter. The bartender tells the rest, "Shuddup," and hands over 2 beers.

- Summing a *geometric series*: In a geometric sequence, two consecutive terms have the same ratio. For example $\langle a, ab, ab^2, \ldots \rangle$. By doing calculations with prefix sums, it is easy to figure out their values:

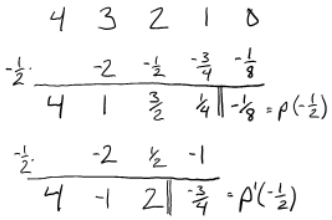
$$\begin{aligned} G_k &= \sum_{1 \leq i \leq k} ab^i \quad = a + ab + ab^2 + \cdots + ab^k \\ - \quad bG_k &= \sum_{1 \leq i \leq k} ab^{i+1} \quad = \quad ab + ab^2 + \cdots + ab^k + ab^{k+1} \\ (1-b)G_k &= a - ab^{k+1} \\ G_k &= a\frac{1-b^{k+1}}{1-b}. \end{aligned}$$

  The prefix sums can tell us if an infinite series converges. Here, when $|b| < 1$, series $\sum_{i \geq 1} ab^i$ converges to $a/(1-b)$.

Why computer scientists confuse Halloween and Christmas: OCT 31=DEC 25:



Evaluating $p(x) = \sum_{0 \le i \le 4} ix^i$ and its derivative $p'(x)$ at $x = -1/2$:



- Horner's rule for polynomials. In subsection 7.1.1, we mentioned Horner's rule as an efficient way to convert a string of digits into a number. In general, Horner's rule is a way to evaluate a polynomial $p(x) = \sum_{0 \le i \le n} a_i x^i$, with coefficients $(a_n, a_{n-1}, \ldots, a_1, a_0)$, at a specified value of x. It works by rewriting the polynomial as a recursive definition that goes from a base case at $i = n$, down to $i = 0$.

  1. Base: $p_n(x) = a_n$.
  2. Recursive rule: $p_i(x) = x \cdot p_{i+1}(x) + a_i$ for all $0 \le i < n$.

  The final polynomial $p_0(x) = p(x)$.

  If we carry out this definition with a specific value of x, then each $p_i(x)$ is just a number. This is easy to do in a table of three rows: write the coefficients, $a_n, \ldots, a_0$, in the first, write the value of x at the start of the second, and copy a_n as $p_n(x)$ to the start of the third. Then, for $i = n - 1$ down to 0, write $x \cdot p_{i+1}$ in the second row under a_i, then add to get p_i in the third row. The result is the last number in the third row.

  This recursive definition doesn't calculate powers of x, which gives it two advantages over the summation formula: It is faster because it uses fewer multiplications, and it tends to be more accurate in floating point because it is less likely to lose precision in the neighborhood of the roots of the polynomial due to cancellation from subtracting large, nearly equal numbers.

  By the way, if you need the derivative of your polynomial evaluated, too, just add a 4th and 5th row in the same way. Since the derivative of $p'_i(x) = x \cdot p'_{i+1}(x) + p_{i+1}(x)$, you can just use the values of row 3 as the coefficients, stopping at $i = 1$.

- The *harmonic numbers* $H_n = \sum_{1 \le i \le n} 1/i$ arise frequently in questions of probability. They form a divergent series—they do not converge to a limit. By rounding each fraction down to the nearest power of two, and then up to the nearest power of two, we get wordless proofs that $H_n \le \lg(n+1)$ and $H_n \ge 1 + (\lg n)/2$. Much tighter estimates are known, but these suffice for our uses.

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \cdots + \frac{1}{n} \\ &< 1 + \underbrace{\frac{1}{2} + \frac{1}{2}}_{1} + \underbrace{\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}}_{1} + \frac{1}{8} + \cdots + \frac{1}{2^{\lfloor \lg n \rfloor}} < \lg(n+1) \end{aligned}$$

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \frac{1}{9} + \cdots + \frac{1}{n} \\ &> 1 + \frac{1}{2} + \underbrace{\frac{1}{4} + \frac{1}{4}}_{1/2} + \underbrace{\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8}}_{1/2} + \frac{1}{16} + \cdots + \frac{1}{2^{\lceil \lg n \rceil}} > 1 + \frac{\lg n}{2} \end{aligned}$$

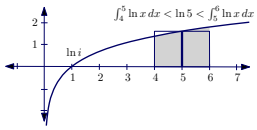
8.2.4 Estimating factorial

Here is another function to estimate. The factorial function $n!$ grows quickly. Exercise 10.8 asks for proof that, for all $n > 3$, factorial is $2^n < n! < n^n$. A closer approximation is Gosper's version of Stirling's formula:*

*For more, see Peter Luschney's pages

Lemma 8.2.1. *For $n \geq 1$, the factorial $n! \approx \sqrt{(2n + \frac{1}{3})\pi}(n/e)^n$.*

Rather than take the time to prove this, I'll derive a weaker expression from upper and lower bounds on $\ln(n!)$. Since $\ln x$ is strictly increasing, we can bound $\ln i$ by integrals for any positive integer i,



$$\int_{i-1}^{i} \ln x\,dx < \ln i < \int_{i}^{i+1} \ln x\,dx.$$

Since $\ln(n!) = \sum_{2 \leq i \leq n} \ln i$, we get the bounds

$$\int_1^n \ln x\,dx < \ln(n!) < \int_2^{n+1} \ln x\,dx,$$
$$n \ln n - n + 1 < \ln(n!) < (n+1)\ln(n+1) - n - 2\ln 2 + 1,$$
$$e \cdot (n/e)^n < \quad n! \quad < (n+1) \cdot ((n+1)/e)^n.$$

8.2.5 Generating functions

An important trick in analysis is to take a sequence $(a_0, a_i, a_2, \ldots)$ and make its elements the coefficients of an "infinite polynomial," $p(z) = \sum_{k \geq 0} a_k z^k$, which is more correctly called a *formal power series* or *generating function*†. Well-known polynomial operations like addition, multiplication, differentiation, composition and evaluation, extend easily to power series, so they can be intepreted as operating on the entire sequence at once. We may not care if a given power series converges, but if it does in the neighborhood of zero (in the complex numbers) then there are powerful mathematical tools that can tell us about the behavior of its coefficients. I briefly give three shallow examples of the deep subject of generating functions; for more, see Wilf's "generatingfunctionology" [27] or Graham, Knuth and Patashnik's "Concrete Mathematics" [9].

†because it can generate the sequence: e.g. $a_k = p^{(k)}(0)/k!$, if you remember your k-th derivatives from calculus

Consider the sequence in which $a_k = 1$ for all $k \geq 0$. We write down the generating function as a formal power series, then manipulate it to find a rational expression.

$$p(z) = \sum_{k \geq 0} z^k \quad = 1 + z + z^2 + z^3 + \cdots$$
$$\underline{\text{minus} \quad zp(z)} = \sum_{k \geq 1} z^k \quad = \quad z + z^2 + z^3 + \cdots$$
$$(1 - z)p(z) = 1$$
$$p(z) = \frac{1}{1 - z}.$$

This is not defined for $z = 1$, but is defined and converges for all $|z| < 1$; this gives another way to obtain values of *geometric series* like $p(\frac{1}{r}) = \sum_{k \geq 0} \frac{1}{r^k} = \frac{1}{1 - 1/r} = \frac{r}{r-1}$ for $r > 1$.

By taking the derivative of $p(z)$, we get the series $\sum_{k\geq 0} \frac{k}{r^k} = r/(r-1)^2$ from

$$p'(z) = \sum_{k\geq 1} kz^{k-1} = \frac{1}{(1-z)^2},$$
$$zp'(z) = \sum_{k\geq 1} kz^k = \frac{z}{(1-z)^2}.$$

Use the Fibonacci sequence to define a generating function $q(z)$, and manipulate it knowing that $F_k = F_{k-1} + F_{k-2}$:

$$\begin{aligned} q(z) &= \sum_{k\geq 0} F_k z^k && = z + z^2 + 2z^3 + 3z^4 \cdots \\ \text{minus}\quad zq(z) &= \sum_{k\geq 1} F_{k-1} z^k && = z^2 + z^3 + 2z^4 \cdots \\ \underline{\text{minus}\quad z^2 q(z)} &= \sum_{k\geq 2} F_{k-2} z^k && = z^3 + z^4 \cdots \\ (1-z-z^2)q(z) &= z \\ q(z) &= \frac{z}{(1-z-z^2)} \end{aligned}$$

The roots of the denominator are related to the *golden ratio* $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618034$ and $\hat{\varphi} = (1-\varphi) = \frac{1-\sqrt{5}}{2} \approx -0.618034$, so we can rewrite $q(z)$ by partial fractions:

$$q(z) = \frac{1/\sqrt{5}}{1-\varphi z} - \frac{1/\sqrt{5}}{1-\hat{\varphi} z}.$$

These two terms resemble $p(z)$ closely enough that we can determine a closed form for Fibonacci numbers:

$$F_k = \frac{\varphi^k - \hat{\varphi}^k}{\sqrt{5}}.$$

I find it surprising that each F_k is an integer, and completely amazing that the sequence is the Fibonacci numbers.

8.3 Summary

With recursive definitions we are able to define structures, functions, and operations of arbitrary (and even infinite) size by specifying a few base cases, and a rule that fills in the rest. We thus avoid ellipses (. . .), a good goal for a computer scientist, since ellipses need some human intelligence to fill in the missing terms.*

*Some programming languages, like Perl, do formally define ellipsis as a restricted, linear operator.

We can work with these definitions and sometimes recognize new structures as old friends in disguise, find closed forms for expressions, or bound the growth of functions or structures that we can't reduce to closed form.

In chapter 10 we will learn induction, which is the main way to prove properties of recursively defined structures.

8.4 Exercises and Explorations

Quiz Prep 8.1. Answer these questions on strings and languages:

1. What is the missing language L in these expression using concatenation? Please be precise – the left and right side of the equal sign should be the same set. Recall that $LM = \{st \mid s \in L \text{ and } t \in M\}$.
   (a) $\{\Lambda, a, ab\}L = \{b, ab, ba, aba, abb, abba\}$.
   (b) $L\{a, b\} = \{a, baa, b, bab\}$.
   (c) $L\{\Lambda, a\} = \{\Lambda, a, b, ab, ba, aba\}$
2. Give example languages M and N that satisfy
   (a) $|MN| = |M| \cdot |N|$.
   (b) $|MN| < |M| \cdot |N|$.
   (c) Explain the differences between the empty set, denoted $\emptyset$ or $\{\}$, the empty string Λ, and the language containing the empty string $\{\Lambda\}$.

Quiz Prep 8.2. Write the recursive definition for the set of all possible lists of elements from a set S.

Exercise 8.3. Find the mistake(s) in each of the following.*

*__Warning:__ incorrect statements in this problem!

1. Let $S(n) = \sum_{n=1}^{n} n \ldots$
2. For even $n \geq 0$, the double factorial $n!!$ is the product of all even numbers in $[1..n]$. For odd $n \geq 1$, the double factorial $n!!$ is the product of all odd numbers in $[1..n]$. Recursively define the double factorial for non-negative integers:

   Base: $0!! = 1$.
   Rec. Rule: for integers $n \geq 1$, $n!! = (n-2)!! \cdot n$.
3. We can recursively define the language S of all strings that have the same number of as and bs:

   Base: $\Lambda \in S$.
   Rec. Rule: if string $\sigma \in S$, then the strings $ab\sigma$, $ba\sigma$ $a\sigma b$, and $b\sigma a$ are in S.

►

Exercise 8.4. Write a recursive definition of a function that will find the maximum element in a list.

Puzzle 8.5. I and my wife Elizabeth went to a party with five other couples. At the start, all pairs of individuals who had not met before introduced themselves and shook hands; pairs who had met before did not shake hands. Later, I asked the other eleven guests (including Elizabeth) how many hands they had shaken, and got eleven different answers. How many hands did Elizabeth shake?

Puzzle 8.6. Calculate $\sum_{0 \le k \le 2n} (-1)^i k^2$. You could start with $n = 10$, which is the series $20^2 - 19^2 + 18^2 - 17^2 + \cdots + 2^2 - 1^2$.

Exploration 8.7. Implement the Fibonacci definition as a recursive function in your favorite programming language. Have it print '0' and '1' each time it reaches one of those base cases. Explain what you observe and ways to make the computation more efficient.

Exploration 8.8. Write a recursive definition for the integers using successor $s()$ to give the positive numbers and predecessor $p()$ to give negative numbers. Every number should have a unique representation. Then write the recursive definitions for operations of addition, subtraction, and multiplication, which are closed under integers, and exponentiation for exponents in $\mathbb{N}$. Write recursive definitions for equality and less than.

Chapter 9
Proof

I mean the word proof not in the sense of the lawyers, who set two half proofs equal to a whole one, but in the sense of a mathematician, where half proof = 0, and it is demanded for proof that every doubt becomes impossible.

—Carl Friedrich Gauss

In previous chapters, we have seen a few formal proofs by truth table or by symbol manipulation using rules of inference. We have seen many more arguments, sketches, and demonstrations that are meant to be convincing, and that contain the key ideas that could be turned into a formal proof by symbol manipulation. In fact, I have in earlier chapters attempted to avoid the words 'proof' and 'prove' except when referring to a formal proof. In this chapter we take a philosophical and practical look at the role of proof in understanding and communicating about discrete structures.

Objectives: After reading this chapter, you will appreciate that proof is about communication, first to oneself and second to others. You will be able to indicate the type of proof by beginning with some common phrases, and use a two-column proof format that tries to set out

what is done: the steps of the proof,
why it is done: the aims of the steps, and
how it is done: the justification of each step.

You will be able to apply writing exercises, not only to proofs but to other writing. The chapter closes with a review of definitions and properties of the previous chapters, which you should be able to use or prove.

9.1 What is a proof

If you look up proof in Webster's dictionary you'll find the legal and mathematical definitions combined in the first definition:

1a: the cogency of evidence that compels acceptance by the mind of a truth or a fact.
b: the process or an instance of establishing the validity of a statement especially by derivation from other statements in accordance with principles of reasoning.

defn 7: alcoholic strength indicated by a number that is twice the percent by volume of alcohol present

Keith Devlin names these two alternatives (in opposite order) at the start of an article on when can a proof be accepted with certainty.

> What is a proof? The question has two answers. The right wing ("right-or-wrong," "rule-of-law") definition is that a proof is a logically correct argument that establishes the truth of a given

> statement. The left wing answer (fuzzy, democratic, and human centered) is that a proof is an argument that convinces a typical mathematician of the truth of a given statement.

As a computer scientist, I could give the pragmatic answer that a proof is good if the program based on it runs and produces correct answers. Kara crashes if she is told to walk into trees, pick up a clover that isn't there, or put down a second clover. She also crashes if no rule applies or if more than one rule applies (in the default "deterministic" mode). We would like to convince ourselves and others that our program can never encounter these situations—especially if we are being paid upon job completion. This is one of the main reasons that formal proof is such an important topic in this book.

In fact, the process of proving, mentioned in Webster's 1b, is often more important than the final proof. The compact, precise language of mathematical symbols, once we become sufficiently fluent in it, helps record our thinking so that we can avoid getting stuck in ruts, and can leverage from the work of others. Imre Lakatos' book, "Proofs and Refutations," points out that when we don't know if a claim is true, then searching in parallel for its proof and its refutation (a counterexample) helps us refine our understanding of the range of possibilities and limitations for solutions.

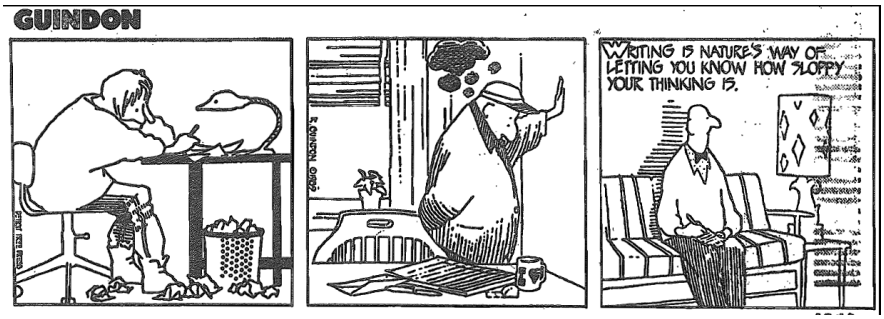


Figure 9.1: Cartoon by Richard Guindon, used by permission.

You may be surprised how much of the process of proving is about communication and writing mathematics precisely. Leslie Lamport opens his book "Specifying Systems" [17] by quoting Richard Guindon, "Writing is nature's way of letting you know how sloppy your thinking is." Leslie continues, "Mathematics is nature's way of letting you know how sloppy your writing is," and later, "Formal mathematics is nature's way of letting you know how sloppy your mathematics is."

9.1.1 The roles of definitions and properties

We use the precise definitions of several discrete structures to determine properties that can help us implement, use, and debug these structures; they help us make sure we are communicating our intent to the computer and understanding its results. At least as importantly, precise specification and reasoning helps us communicate to humans—to bosses or clients — so that all can agree on tasks to be done (and we can demonstrate that we need to be paid more if they change their mind) and to ourselves, since we can only think about what we can express in language. That is why formal reasoning about discrete structures is a key component of this book.

Proof that a cat has nine tails: No cat has eight tails. A cat has one tail more than no cat. Therefore, a cat has nine tails.

Definitions are good if they are consistent and expressive—if from a small number of definitions we can derive many useful properties without deriving any contradictions. We must start with some undefined terms (primitives or axioms—for us these are basic logic, set, and tuple operations: $\wedge$,$\vee$, $\neg$, $\in$, $\forall$, variables, braces, and parentheses suffice). From these basics we can define other operations (e.g., $\subseteq$, $\cup$, $\cap$, $\times$, complement, counting, concatenation, ...), derive their properties (associativity, commutativity, distribution laws, de Morgan's laws, ...), and so on. These properties, once proved, become shortcuts so that we don't have to prove everything from the basics. When you start, however, and whenever you are uncertain, do go back to the basics to avoid circular reasoning* (e.g., assuming A to prove A.) We also may need to forget the informal way we use certain words, e.g., logical 'or,' $p \vee q$, always allows for both p and q to be true, even though sometimes we informally mean 'exclusive or,' $p \oplus q$.

*Of course this joke is relevant; every joke in the margins is relevant.

9.1.2 Proof types

Remember that proofs are written down to communicate. One of the important things to communicate early is what will be the logical structure or type of your proof. Some stock phrases can indicate to a reader what type of argument you will use.

Proof of $\forall x \in S$: Defining and proving statements 'for all' is so common that the universal quantifiers are frequently omitted. A statement may simply use x as a generic variable when the set containing x can be understood from context. This matches the common approach to proving such statements: begin by assuming that someone has given you a generic value for x, and prove the statement for that value, then add the quantifier by universal generalization. Such a proof often begins with a phrase like, "Suppose that we are given an $x \in S$," or more briefly, "Given $x \in S$."

It is important that the value is generic—that any value could be given, not just a specific example. While working through specific examples can help

determine the reasons a statement is true, to convincingly establish "for all," you want an argument that can apply to any given value.

When the set S is the natural numbers or some other recursively defined set, then the technique of mathematical induction is often used to prove $\forall x \in S$ statements. chapter 10 gives a detailed template for creating induction proofs without having to think, except in the interesting part of the proof.

Direct proof of conditional: Many properties that we may want to prove have the form, "if a, b, and c are true, then q is true," although this may be expressed in English in different words: "Assume a, b, and c. Then q." or "An a that is b and c is also q."A direct proof assumes that the given a, b, and c are true, and tries to work from that knowledge to derive that q is true. Lemma 9.2.1 will show an example: a proof that if a string $x \in L(M \cap N)$ then $x \in LM \cap LN$, which, after rephrasing what is to be proved, begins "Suppose that. . . " and continues to show the statement to be proved is true.

It does no good to also assume that q is true and from that to demonstrate other true things. What you need to rule out is the possibility that q can be false while a, b, and c are true. That, you'll recall, is the only way for the conditional $(a \wedge b \wedge c) \rightarrow q$ to be false.

Proof of contrapositive: Because $(p \rightarrow q) \equiv (\overline{q} \rightarrow \overline{p})$, we can start a direct proof of a conditional by assuming the negation of q to show the negation of p. Begin your proof with, "We will prove the contrapositive, so assume $\overline{q}$. . . "

Proof by contrapositive is still a direct proof, unlike proof by contradiction. (Proof by inverse or converse are no good, since those are not logically equivalent to the original conditional.)

*The Chinese word for *contradiction*, 矛盾, is 'spear shield,' from a 3rd century BC story of a merchant with unstoppable spears and impenetrable shields at the same market stall.

Proof by contradiction: To show something is true, we can assume that it is false and derive a contradiction.* This is useful way to find a proof of $p \rightarrow q$ because you can start by assuming that p is true and q is false—you get more to work with at the start of your proof. Begin your proof by saying, "Assume, for the sake of deriving a contradiction, that p is true and q is false," and conclude your proof with, "This is a contradiction, which shows that our initial assumption is false. Therefore, $p \rightarrow q$." The classic example is the proof that $\sqrt{2}$ is not rational in Theorem 7.2.1, which gets to assume that a fraction $p/q = \sqrt{2}$ exists to derive a contradiction showing it does not.

Once you find a proof by contradiction, you can often turn it into a shorter, clearer direct proof. This is especially true when your assumption is that

something does not exist, and you find your contradiction by showing it does. You can then extract a shorter, direct proof to show it exists.

Proof of "if and only if": Because $p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p) \equiv (p \rightarrow q) \wedge (\overline{p} \rightarrow \overline{q})$, proof of "iff" often breaks down into two proofs, one for each direction. Start such a proof by saying, "We first prove $p \rightarrow q$, then the converse $q \rightarrow p$ [or inverse $(\overline{p} \rightarrow \overline{q})$]. . . " It is possible to do both directions at once by a series of iff statements, but you must be careful that they really are equivalences—that they work going forward and backward. Since set equality $A = B$ is defined as $\forall_x\ (x \in A) \leftrightarrow (x \in B)$, proving two sets are equal is a special case of proving $p \leftrightarrow q$, or actually $p(x) \leftrightarrow q(x)$ for a generic element x. The proofs for language concatenation with intersection (Lemma 9.2.1) and union (Lemma 9.2.2) illustrate the two ways to attempt to prove 'iff' statements.

Proof of $\subseteq$: Replace the statement $A \subseteq B$ by its definition, $\forall_{x \in U}\ (x \in A) \rightarrow (x \in B)$, to see that we want to show a quantified conditional statement. So assume a generic element x is given, and show that if $x \in A$ then $x \in B$. Proof can proceed directly (assume $x \in A$, derive $x \in B$), by contrapositive (assume $x \notin B$, derive $x \notin A$), or by contradiction (assume both, derive a contradiction).

Proof of = for sets: As mentioned under "if and only if," we can use the definition of $A = B$, which goes back to individual elements and uses "iff." We can instead use the important property that $A = B$ iff $(A \subseteq B$ and $B \subseteq A)$. We often prove statements about equality of sets by proving these two statements of about subsets. Start your proof by saying, "To show $A = B$, we first show that $A \subseteq B$, then that $B \subseteq A$." Often one direction is easier than the other; I typically start with the easy direction as the warm-up.

Proof by exhaustive checking: Suppose that we have a predicate $q(x)$. We can show $\exists_{x \in S}\ q(x)$ by exhibiting one element of S. For a finite set S, we can show $\forall_{x \in S}\ q(x)$ by checking all elements of S. These proofs by exhaustive checking are boring, but effective. Start your proof by saying, "We can check that. . . " (In advanced mathematics these often become "The reader can check. . . " and this is often where errors lurk because the author was being lazy or sloppy. So do enough of the exhaustive check to be sure that it is right, even if you don't write all the details for the reader.)

Proof of $\exists x \in S$: A proof of existence that produces a specific example is called a *constructive proof*; most exhaustive proofs are constructive. Some proofs do not produce a specific example; here is a *non-constructive proof* that one of two candidate pairs satisfies the next lemma, without identifying which pair.

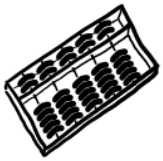
Lemma 9.1.1. *There exists a pair of irrational numbers, a and b, such that a^b is rational.*

Proof. We know that $\sqrt{2}$ is irrational. Consider $\sqrt{2}^{\sqrt{2}}$: if it is rational, then $a = b = \sqrt{2}$ gives a rational a^b. Otherwise, let $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$, and observe that a^b is rational:

$$a^b = \left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\cdot\sqrt{2}} = \sqrt{2}^2 = 2.$$

∎

In fact, $\sqrt{2}^{\sqrt{2}}$ is irrational, and even transcendental (it is not the root of any finite polynomial with integer coefficients), but the proof of that fact uses deeper mathematics that would completely obscure the simplicity of this proof. Most non-constructive proofs show that an example exists by contradiction or by counting; these can be quite hard to turn into constructive proofs.



Combinatorial proof: Last, but not least, counting the same quantity in two different ways can give concise, elegant proofs. For example, knowing that the binomial coefficient $\binom{n}{r}$ is the number of ways to choose r things out of n, you can easily observe that $\binom{n}{r} = \binom{n}{n-r}$ because each way you choose r of n items to keep is also a way of choosing $n - r$ to discard. Speaking more formally, "We establish a bijection between the ways to choose r out of n items and the ways to choose $n - r$ out of n items."

9.2 Modified two-column proof form

There can be no hard and fast rules for the form of a proof, since communication via proof is a creative activity—find some form that works for you, and maybe some modifications that communicate well to others. Nevertheless, putting some restrictions on the form can make it easier both for us to write correct proofs and for others to read our proofs.

The traditional "*two-column proof*" for geometry, created in 1913, had enough criticism in recent decades that some of you may have never seen it. The idea is to write the steps in one column and the reasons in a second column, showing "what" in the left column and "how" in the right. Let's modify this, adding words on "why" to the steps so that you can read the left side as a narrative, and the right side as comments on the narrative. Here are some examples.

Warning: only one of these is true!

9.2.1 Example two column proofs

For all languages L, M, N, on an alphabet Σ, can we show that $L(M \cap N) \stackrel{?}{=} LM \cap LN$ and that $L(M \cup N) \stackrel{?}{=} LM \cup LN$?

Here is a list of definitions and properties that I will assume that we know. (I try not to state definitions like these inside a proof, because they break up the flow. Instead, I just use them. I include the statements here so you can see the primitive definitions I will rely on.)

- A language is a set of strings; a string is an ordered sequence of letters from Σ. In brief, $L, M, N \subseteq \Sigma^*$, and a string $a \in \Sigma^*$.
- For two languages $A, B \subseteq \Sigma^*$, the concatenation or product $AB = \{a\beta \mid \text{for every possible choice of } a \in A, \beta \in B\}$.
- For two sets A, B, the intersection $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$.
- Various properties of logical 'and' $\wedge$ and 'or' $\vee$: idempotent, distributive, commutative, associative.

Back to the two things to show. In each case, we want to show two sets are equal, so we can try to show that each is a subset of the other. Let's look at intersection first.

Lemma 9.2.1. *For all languages L, M, N on alphabet Σ, we have $L(M \cap N) \subseteq LM \cap LN$: concatenation with an intersection is a subset of the intersection of concatenations.*

1. We will show that $L(M \cap N) \subseteq (LM \cap LN)$, or equiv.,	
2. that $\forall x \in \Sigma^*$, if $x \in L(M \cap N)$ then $x \in (LM \cap LN)$	Definition of subset.
3. Let us do this by direct proof:	To show $p \to q$, we suppose p is true
4. Suppose that $x \in L(M \cap N)$;	and show q must be true.
5. that is, $\exists a \in L$ and $\exists \beta \in (M \cap N)$ with $a\beta = x$.	Defn concatenation of languages
6. But then $\beta \in M$ and $\beta \in N$,	Exist. instantiation & defn $\cap$
7. so we can write $x = a\beta$ with ($a \in L$ and $\beta \in M$) and ($a \in L$ and $\beta \in N$).	Properties of 'and' (idempot., commut., assoc.), or by checking a truth table for equivalence.
8. But then $x \in LM$ and $x \in LN$,	Defn concat of languages
9. so $x \in (LM \cap LN)$, as desired.	Defn of intersection
10. Thus, we have shown that $L(M \cap N) \subseteq (LM \cap LN)$.	Direct proof of conditional and defn of subset.

∎

What if we wanted to show the opposite inclusion, that $(LM \cap LN) \subseteq L(M \cap N)$? Well, we better have a problem because this is false, as we can prove by a counterexample: Choose $L = \{a, ab\}$, $M = \{b\}$, and $N = \{\Lambda\}$, then*

*Can you see that defns from 3.1.2 imply $L\emptyset = \emptyset$ and $L\{\Lambda\} = L$?

$$L(M \cap N) = L\emptyset = \emptyset, \text{ but}$$
$$LM \cap LN = \{ab, abb\} \cap \{a, ab\} = \{ab\}.$$

Let's try to prove this false direction and see how far we get:

1. We want to show that $(LM \cap LN) \subseteq L(M \cap N)$, or equiv.,	
2. that $\forall x \in \Sigma^*$, if $x \in (LM \cap LN)$ then $x \in L(M \cap N)$	Defn of subset.
3. Let us do this by direct proof:	To show $p \to q$, we suppose p is true
4. Suppose that $x \in (LM \cap LN)$;	and show q must be true.
5. that is, $x \in LM$ and $x \in LN$	Defn of intersection.
6. Then $\exists a \in L$ and $\exists \beta \in M$ such that $x = a\beta$.	Defn concat of languages
7. and $\exists \gamma \in L$ and $\exists \delta \in N$ such that $x = \gamma\delta$.	
8. If $\beta = \delta$ then we are good: $a = \gamma$ by definition of equality of strings and we can finish the proof.	
9. But if $\beta \neq \delta$ then we are stuck, and that is exactly what happens in the counterexample.	

Note that I am careful not to reuse variable names a, β for LN in line 7 that are already in use for LM in line 6. Reusing a variable that already has another meaning is one of the most common mistakes in a proof—if I did that here, I could complete the proof of a false theorem. Because I was careful, my attempted proof instead indicates how the proposed lemma could fail: L must contain a string and one of its proper prefixes—this is how I knew to include ab and a in the counterexample. (In cases where no string in L is a prefix of another, then the proof can be completed successfully and the result holds!)

Two-column proofs aren't always so long. For example, let's prove that for all languages $L, M, N \in \Sigma^*$, concatenation does distribute over union: $L(M \cup N) = LM \cup LN$. We'll do this in traditional two-column format by a sequence of equalities in the left column, with justifications on the right.

Lemma 9.2.2. *For all languages L, M, N on alphabet Σ, concatenation distributes across union: $L(M \cup N) = LM \cup LN$.*

1. $L(M \cup N) = \{a\beta \mid a \in L \wedge (\beta \in M \vee \beta \in N)\}$	Defn concat and union
2. $= \{a\beta \mid (a \in L \wedge \beta \in M) \vee (a \in L \wedge \beta \in N)\}$	Distribution $\wedge/\vee$.
3. $= \{a\beta \mid a \in L \wedge \beta \in M\} \cup \{\gamma\delta \mid \gamma \in L \wedge \delta \in N)\}$	Defn union, renaming variables.
4. $= LM \cup LN$.	Defn concatenation.

Since each step is small, a proof in traditional two-column format should be easy to verify. There are some disadvantages:

- The intent of each step must be inferred, as it is not stated.
- The format encourages heavy use of symbols, which make it hard to read for those not fluent in the notation.
- Small steps can be tedious.

These are why I prefer to modify the two-column format to include more narrative text in the left column, as in the proof of Lemma 9.2.1.

To make the proof of Lemma 9.2.2 more narrative, it may be enough to summarize the proof in words before giving the notation:* "After expanding $L(M \cap N)$ using the definitions of concatenation and union, we identify that each string is in LM or LN."

*It is always a good idea to summarize in words what you will say in notation.

As with any communication, you need to know your audience—their mathematical background suggests what steps you can do in great leaps, and what steps you need to expand in detail. L. Lamport [16] suggests that hyperlinked proofs could allowing a reader to click on a higher level property (e.g., de Morgan's law for sets) and get the expansion of the proof of that law.* For yourself, you should expand your proof one level deeper than what you find convincing.

*I use margin notes and tooltips to expand on some proofs in this book.

Modified two-column proofs still tend to favor a particular order: proceeding from what the audience knows (the given information) to what they don't (the target statement to be proved). On scratch paper we often work from both ends, given and target, until we meet in the middle. We may even work entirely backwards, from the target. Sometimes these orders are the best way to write the final proof as well; just make sure to indicate what things we know, and what things we are trying to show. Adding words ("We want to show" or "It is enough to prove" vs. "We know" or "We are given") helps. You can also use notation, such as marking predicates you'd like to establish with a question mark ($\overset{?}{<}$, $\overset{?}{\leq}$, $\overset{?}{=}$, $\overset{?}{\subseteq}$, ...).

Once you are accustomed to modified two-column proofs, you can drop the right column, or incorporate the non-trivial "how"s into the narrative. Most arguments in mathematical and computer science work are given at a higher level, with many steps or justifications omitted under the assumption that the intended audience has the knowledge to supply them, given sufficient motivation.† This should only be done after you have demonstrated that you are able to supply the steps and justifications, so don't try to write at a high level too early, or for anything critical; go one level deeper than seems clear to you. Even if you are writing at a high level, I recommend writing reasons for each step of your proofs on your scratch paper, to avoid leaps of faith, reuse of variables, or other logical mistakes that might be overlooked in a narrative proof form. Even better, find a skeptical friend who is willing to question your proof attempts, and force you to demonstrate that you understand the details.

†The classic proof in a lecture:
Prof: From steps 7 and 12, it is obvious that we get 13...
Back row student: Is it?
Prof, after 10 min of staring at the board: Yes.

9.3 Communication

Remember that your proofs are generally for a human (often yourself) to read. To communicate well you need to use a shared vocabulary and shared assumptions, both of which this book aims to remind you of/introduce. You also need some idea of the way your reader is expecting you to build on this

shared context. Since this is difficult to do in a first draft,[‡] I give five specific exercises that I use to improve my own writing.

George Gopen [8] says that a reader of English expects to find context first in a sentence, paragraph, or section, and new information after, with the information being stressed at the end.[*] New information of one sentence or paragraph often becomes context for the next, or the author may add more information in the same context. If the author provides information where it is expected then the prose will "just flow" – the reader will not have to backtrack and spend mental energy to disentangle words, but can save that energy for understanding ideas.

The ideas of context first and managing reader expectations are behind most of the picky rules of writing, such as comma usage and which vs. that. They are also behind rules of and advice for mathematical writing:

- Define any terms or variables before you use them; don't define variables or notation that you will not use, or will use only once. Don't use variables outside their scope. Avoid reusing a variable—one of the most common mistakes in proofs is to have two different variables have the same name.
- Start with words that indicate the purpose of a variable or expression:
  - To choose a generic x to show $\forall_x$: "Assume that x" or "Suppose that x"
  - To name an x that comes from the problem statement: "Given an x that"
  - To define x to replace a complex expression: "Define x" or "Let x."
  - To recall a previously defined variable or property: "Since," "Because," "We know that," or "But."
  - To state a conclusion: "So," "Thus," or "Therefore."
  - To state a goal: "We want to show."
  - To identify cases: "We will have n cases: First. . . , Second. . . , Finally. . . ," "On the other hand," "Conversely,"
  - To rephrase what you want to show: "Equivalently," or "Rewriting this as."
  - Reserve "If" for if-then logic; don't use "If" in place of any of the above.
- Always separate mathematical notation by words. Consider summarizing a proof in words before you give it in notation. Punctuate properly—punctuation helps your reader know what to expect. Stay in present tense.
- Reserve the equals sign, '$A = B$,' for expressions in which both sides are really equal, or are being defined or assigned to be equal, with a variable named A being assigned the value of B. (Many computer languages distinguish even between these, either using == for testing equality or := for assignment.) Please don't use '=' when you mean logically equivalent ($\equiv$ or $\Leftrightarrow$), the logic operator iff ($\leftrightarrow$), or even "I hope to show that these are equal" ($\stackrel{?}{=}$).

[‡] The five steps to good writing: write, rewrite, rewrite, rewrite, rewrite.

[*] In contrast, I've been told that scholarly writing in German starts a paragraph by telling you what it is not about, in the middle states the point, then ends with all the exceptions.

In a first draft, it is usually best to record ideas quickly, with little concern for the ideal organization and presentation. But then improve what you've written by applying writing exercises. Here are my favorite five:

1. Context first (Gopen). Find the context and new information in sentences in a paragraph or paragraphs in a section and ensure that context comes first. For a mathematical example, let's rewrite a typical, first-draft proof that any even number a times any integer b is even: "When a is even ab is even because we know there is a c so that $ab = 2cb$, where $2c = a$. But ab is even, since $ab = 2cb$ and the 2 shows that ab is even." This has all the right statements, but you have to read past each statement for its reason, then reread to verify it.

   Don't mimic this example.

   Notice that the context is the assumption that a is even, which was not clearly distinguished from the desired conclusion. The reader is assumed to know the definition of 'even' so, in the first sentence, the new information is the naming of integer c certifying that a is even. The naming of c gives context for the second sentence. Realizing this, we can reorder and shorten: "For any a that is even, there is some integer c satisfying $a = 2c$. But then $ab = 2cb$ is even."

   Mimic this instead.

2. Strengthen parallels (Gopen / Strunk & White) Especially for technical writing, find parallels and strengthen them. E.g., use consistent terminology (if you called it an algorithm, don't call it a method, technique, or approach later) and sentence structure (if you find yourself saying "another xyz" make sure the first was labeled as an "xyz" and, ideally, start out by saying how many "xyz"s the reader will see. Furthermore, if you write "the first is a blah that performs a smurge," then you better write "the second is a foo that performs a bar.") Using the same sentence structure makes the similarities and differences obvious without you even needing to comment. I expect that you were told in middle school to "vary your sentences to make your writing interesting." Forget that—now you are supposed to have interesting ideas, and by using the identical sentence structure you can highlight the clarity of your ideas, especially for comparing and contrasting.

3. Use vivid verbs (A. Snoeyink/MC van Leunen) Underline all verbs, giving being verbs (is, was, are) double underlines, and passive verbs wiggly underlines. Consider rewriting any paragraph in which less than half of the verbs are single underlines. Being and passive verbs can be fine, but overusing them hides the actors, lengthens sentences, and makes technical prose even more boring.

4. Omit needless words (Strunk & White). Go through a document and cross out as many words as possible without changing the meaning.

Easy candidates are phrases like "in terms of," "for the purpose of," "by means of," "very," "quite"... Sometimes you [are going to have to → must] add a word to remove others.

5. Paragraph outline (Gopen/MC van Leunen): Identify the point you are making in each paragraph; if there is more than one, consider splitting. Check that these points are made in a logical order.

9.4 What may I use?

One of the big questions in doing proofs is "What facts am I allowed to use?" I close this chapter with some pages on this because it is a good way to collect a summary of definitions and properties that we have seen thus far.

It is important to distinguish between definitions and the properties that can be derived from definitions. For example, set equality is defined as $A = B$ iff $\forall_{x \in U}, (x \in A \leftrightarrow x \in B)$, but we often use the property that $A = B$ iff $(A \subseteq B)$ and $(B \subseteq A)$. You may always use the definitions, and once a property has been established, you may use it. The thing to avoid is circularity: you cannot allow the derivation of property A to contain hidden within it the assumption that property A is true. To avoid this, we can put the properties in order,* and insist that the proof of a property use only properties that have been proved earlier. The rest of this section gives example properties in the order for this book. It may look a little overwhelming, but these are all part of the knowledge of a working computer scientist. In fact you need only remember a few primitives and definitions to derive the many properties. On the other hand, once you have a firm grasp of the basics, you'll want to remember many of the properties because they let you work faster and at a higher level of abstraction, which is important for exams and for specifying and developing robust software in the working world.

*actually, a partial order—we'll define those in section 12.2

JSS: I need to clean up the formatting.

9.4.1 Primitives

Our primitives come from sets and logic; we can reduce any proof back to these.

A mathematician's view of how other disciplines would prove a mathematical theorem, such as the theorem, "All odd numbers greater than two are prime:"

Set construction: We may make sets by listing elements in braces, giving a rule $\{x \mid p(x)\}$, or recursive definition (base, recursive rule, closure).
Element of: $x \in S$ iff x is an element of set S.

We may replace any expression with a logically equivalent expression. Equivalence (denoted $\equiv$) can be verified by a truth table. We name many important equivalences so we can remember them and not always have to make the truth table. (Tell me of typos you find!!)

Idempotence of 'and' $\wedge$, 'or' $\vee$ and not: $p \equiv p \wedge p \equiv p \vee p \equiv \overline{\overline{p}}$
Commutativity: $p \wedge q \equiv q \wedge p$ and $p \vee q \equiv q \vee p$.
Associativity: $p \wedge (q \wedge r) \equiv (p \wedge q) \wedge r$ and $p \vee (q \vee r) \equiv (p \vee q) \vee r$.
Distribution: $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$ and $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$.
deMorgan's laws for logic: $\overline{p \wedge q} \equiv \overline{p} \vee \overline{q}$ and $\overline{p \vee q} \equiv \overline{p} \wedge \overline{q}$.
Tautology and contradiction: $p \vee \overline{p} = p \to p = p \leftrightarrow p = T$ and $p \wedge \overline{p} = p \leftrightarrow \overline{p} = p \oplus p = F$.
Absorption and identity: $p \wedge T = p$, $p \wedge F = F$, $p \vee T = T$, and $p \vee F = p$.
if: $p \to q \equiv \overline{p} \vee q \equiv \overline{q} \to \overline{p}$.
iff: $p \leftrightarrow q \equiv (p \to q) \wedge (q \to p)$.
To prove $p \to q$ we may assume p and show q. (Or assume $\overline{q}$ and show $\overline{p}$.)
To prove $p \leftrightarrow q$ we prove both $p \to q$ and $q \to p$.
Quantifier expansion: If $S = \{s_1, s_2, s_3, \ldots\}$ then $\forall_{x \in S}\ p(x) \equiv \bigwedge_{x \in S} p(x)$ and $\exists_{x \in S}\ p(x) \equiv \bigvee_{x \in S} p(x)$.
Quantifier negation: If $S = \{s_1, s_2, s_3, \ldots\}$ then $\overline{\forall_{x \in S}\ p(x)} \equiv \exists_{x \in S}\ \overline{p(x)}$ and $\overline{\exists_{x \in S}\ p(x)} \equiv \forall_{x \in S}\ \overline{p(x)}$.

Physicist: 3 is prime, 5 is prime, 7 is prime, 9 is not prime, but 11 is prime, and 13 is prime. 9 must have been experimental error; all odd numbers greater than two are prime.

Chemist: 3 is prime, 5 is prime, 7 is prime. That's enough data; all odd numbers greater than two are prime.

Concatenation and equality of tuples or strings can be formally defined recursively, but since we used them before we talk about recursive definition, we can take as primitive that, e.g., $((a, \ldots, b), (c, \ldots, d)) = (a, \ldots, b, c, \ldots, d)$, and $(a_1, a_2, \ldots) = (b_1, b_2, \ldots)$ iff $\bigwedge_{i \geq 1} a_i = b_i$.

Similarly, we may want to use cardinality (counting) before we define it using bijective functions, so $|A|$ can be considered primitive.

Biologist: 3 was known to be prime, but, hey! 5 is prime! Let's publish, and apply for a grant to study 7 and 9.

9.4.2 Basic definitions and properties that follow from them

Just as reminders, here is a collection of definitions. U is the universal set.
Set equality: $A = B$ iff $\forall_x, (x \in A \leftrightarrow x \in B)$,
Subset: $A \subseteq B$ iff $\forall_{x \in U}\ x \in A \to x \in B$.
Complement: $\overline{A} = \{x \mid x \in U \wedge x \notin A\}$
Union: $A \cup B = \{x \mid x \in A \vee x \in B\}$
Intersection: $A \cap B = \{x \mid x \in A \wedge x \in B\}$
Power set $\mathcal{P}(A) = \{S \mid S \subseteq A\}$
Cartesian product $A \times B = \{(x, y) \mid \forall_{x \in A}\ \forall_{y \in B}\}$
Language concatenation $LM = \{a\beta \mid \forall_{a \in L}\ \forall_{\beta \in M}\}$
Kleene star: $L^* = \bigcup_{i \geq 0} L^i$, where $L^0 = \{\Lambda\}$, $L^1 = L$, $L^2 = L \times L$, . . . *

*I include this one line with ellipses for information—the real definitions is recursive in section 8.1.

The following properties of set operations are proved and named so that we can use them. Their proofs should use only primitives, definitions, and properties above them. (In fact, each proof can be expanded to use only definitions and primitives. Exercise: do some of these; they build character.)
Equality and subset: $A = B$ iff $(A \subseteq B)$ and $(B \subseteq A)$.
Idempotence: $A = A \cup A = A \cap A = \overline{\overline{A}}$

Architect: 3 is prime, 5 is prime, 7 is prime, the engineers will figure out how to make 9 prime, 11 is prime, . . .

Commutativity: $A \cup B = B \cup A$ and $A \cap B = B \cap A$.
Associativity: $A \cup (B \cup C) = (A \cup B) \cup C$ and $A \cap (B \cap C) = (A \cap B) \cap C$ and $A \times (B \times C) = (A \times B) \times C$ and $L(MN) = (LM)N$.
Distribution: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ and $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ and $A \times (B \cap C) = (A \times B) \cap (A \times C)$ and $A \times (B \cup C) = (A \times B) \cup (A \times C)$ and, as we have proved as Lemmas 9.2.2 and 9.2.1, $L(M \cup N) = LM \cup LN$ and $L(M \cap N) \subseteq LM \cap LN$ *Note: not equality!*
de Morgan's laws for sets: $\overline{A \cup B} = \overline{A} \cap \overline{B}$ and $\overline{A \cap B} = \overline{A} \cup \overline{B}$
Absorption and identity: $A \cup \emptyset = A$, $A \cap \emptyset = \emptyset$, $A \cup U = U$, $A \cap U = A$, $A \times \emptyset = \emptyset$, $A \times \{()\} = A$, $L\emptyset = \emptyset$, and $L\{\Lambda\} = L$.
Absorption for subset: $A \subseteq B$ iff $A \cap B = A$ iff $B \cup A = B$.

Psychologist: 3 is prime, 5 is prime, 7 is prime, 9 is latently prime but repressing it, 11 is prime; all odd numbers greater than two are prime.

Some of the counting properties we can't prove until we get induction, but they still make good checks of understanding.
Inclusion/exclusion: for two sets $|A \cup B| = |A| + |B| - |A \cap B|$, for three sets $|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$, generalizes to more sets. . .
Power set: $|\mathcal{P}(A)| = 2^{|A|}$.
Cartesian product: $|A \times B| = |A| \cdot |B|$.
String concatenation: for strings $a, \beta \in \Sigma^*$, the length $|a\beta| = |a| + |\beta|$.
Language concatenation: $|LM| \leq |L| \cdot |M|$ with equality if each string can be made in only one way.

9.4.3 Definitions with variations: Functions, relations, graphs

Accountant: 3 is prime, 5 is prime, 7 is prime; uh, did you really need 9 to be prime? Because I could shift 2 more into this account and you'll have 11 which is prime.

Many of these are elaborated upon in later chapters.

A *relation* on sets A, B is a subset of $A \times B$.
A *function* $f\colon A \to B$ is a relation $F \subseteq A \times B$ that sends each element of A to exactly one B. The formal definition is $\forall_{x \in A}\ \exists_{y \in B}\ ((x, y) \in F)$ and $\forall_{x \in A} \forall_{y_1, y_2 \in B}\ ((x, y_1) \in F \wedge (x, y_2) \in F) \rightarrow (y_1 = y_2)$. We usually use the notation $f(x)$ for the value in B that x is mapped to, but we could also say $(x, f(x)) \in F$.

The *image* of a set $S \subseteq A$ under function $f\colon A \to B$ is the set $f(S) = \{f(x) \mid x \in S\}$. Note $f(S) \subseteq B$.
The *pre-image* of a set $T \subseteq B$ under function $f\colon A \to B$ is the set $f^{-1}(T) = \{x \mid f(x) \in T\}$. Note $f^{-1}(T) \subseteq A$.
The function f is *surjective* (f is a surjection, aka onto) iff $f(A) = B$. That is, every element of B is mapped from some $x \in A$.
A function f is *injective* (f is an injection, aka one-to-one) if no two elements of A map to the same element of B. Formally, $\forall_{y \in B} \forall_{x_1, x_2 \in A}\ (f(x_1) = y) \wedge (f(x_2) = y) \rightarrow (x_1 = x_2)$.
A function f is *bijective* (f is a bijection, aka one-to-one and onto) if it

is injective and surjective. For a bijection, the reverse mapping is also a function, called the *inverse* and denoted $f^{-1}: B \to A$.

Engineer: 3 is prime, 5 is prime, 7 is prime, 9 is prime, 11 is prime, 13 is prime, 15 is prime, 17 is prime, 19 is prime...

For a relation R, let's write $a\ R\ b$ as an abbreviation for $(a, b) \in R$ and $a \not R\ b$ for $(a, b) \notin R$. We will see much more about relations later, but here are some variants:
A relation on $A \times A$ is *reflexive* iff $\forall_{x \in A}\ x\ R\ x$, *irreflexive* iff $\forall_{x \in A}\ x \not R\ x$,
symmetric iff $\forall_{x \neq y \in A}\ x\ R\ y \leftrightarrow y\ R\ x$,
antisymmetric iff $\forall_{x,y \in A}\ x\ R\ y \leftrightarrow y \not R\ x$,
transitive iff $\forall_{x,y,z \in A}\ x\ R\ y \wedge y\ R\ z \to x\ R\ z$,
an *equivalence relation* iff it is reflexive, symmetric, and transitive,
a *partial order* iff it is antisymmetric and transitive and either reflexive or irreflexive.

A *graph* G is a pair of sets (V, E), with V a finite set of *vertices* and $E \subseteq V \times V$ a set of pairs called *edges*. We will see more on graphs later.

Computer Scientist: 1 is prime, 1 is prime, 1 is prime, 1 is prime, ...

9.5 Summary

You may be surprised that I consider proof relevant to computer science. After all, what is proof but the manipulation of cryptic symbols with arcane rules? In computer science we get to create things by ... the manipulation of cryptic symbols with arcane rules. Just as an architect has a better chance of designing a functional and elegant building if he or she explores and learns by sketching concepts and making models with media that have key properties of the desired final product, so we will have better software by exploring and learning with our scratch paper, where mistakes are less costly.

Proof is about communication, and communication can be difficult when the language is unfamiliar. Think of how you learn other languages: you start with a small vocabulary, practice under supervision, make many mistakes, and learn from them. Then you go out and start using the language, continuing to learn vocabulary, make mistakes, and improve your understanding of what communicates. As you become fluent, you learn to recognize elegant expressions from inelegant expressions, and can even begin to appreciate poetry. You can only reach that level by time spent in practice, making mistakes, and learning from them.

Don't expect that the first thing you say will be said well. Plan what you want to say on scratch paper, then rewrite, using some of the suggestions on communication. It can take less time to do a detailed sketch and a rewrite than to try to write a first draft clean enough to submit or to serve as the basis for code you will write.

9.6 Exercises and Explorations

Quiz Prep 9.1. Choose any pair of the following statements about sets A and B and write a two-column proof that they are equivalent:

a $A \subseteq B$
b $\overline{B} \subseteq \overline{A}$
c $A \cup B = B$
d $A \cap B = A$
e $A \setminus B = \emptyset$

Quiz Prep 9.2. Suppose that we have a function $f : A \to A$ with the property that, for all $x \in A$, $f(f(x)) = x$.

1. We can prove that f is a *surjection*, which means: (fill in definition, including quantifiers.)
2. From this, I know:
3. I want to show:
4. Complete the proof.
5. We can prove that f is an *injection*, which means: (fill in definition, including quantifiers.)
6. From this, I know:
7. I want to show:
8. Complete the proof.

Quiz Prep 9.3. Prove or show a counterexample: For all positive integers $a, b, c \in \mathbb{Z}^+$ with $c < \min(a, b)$, if $c \nmid a$ and $c \nmid b$ then $c \nmid (ab)$.

Quiz Prep 9.4. I want to show, for all reals x, that $\lfloor x \rfloor + \lceil -x \rceil = 0$. Write a sequence of eight to twelve line numbers that would constitute a clear and valid proof. (There are many sequences because there is some choice of ordering. Choose a sequence that, for example, defines variables before using them.) ?

1. Also, $\lceil -x \rceil = -3$.
2. Let $m = \lfloor x \rfloor$, which is the greatest integer less than or equal to x.
3. Let $n = \lfloor -x \rfloor$, which is the greatest integer less than or equal to $-x$.
4. Let $m = \lceil x \rceil$, which is the smallest integer greater than or equal to x.
5. Let $n = \lceil -x \rceil$, which is the smallest integer greater than or equal to $-x$.
6. Let $x = 3.1415926$.
7. $m + 1 > x$, because m is the greatest integer less than or equal to x.
8. $m + 1 > x$, because $m \geq x$.
9. $m + 1 < x$, because m is the smallest integer greater than or equal to x.
10. $m + 1 < x$, because $m \leq x$.
11. $n - 1 > -x$, because n is the greatest integer less than or equal to $-x$.
12. $n - 1 > -x$, because $n \geq -x$.
13. $n - 1 < -x$, because n is the smallest integer greater than or equal to $-x$.
14. $n - 1 < -x$, because $n \leq -x$.
15. Suppose that I am given a real number x.
16. Thus, $\lfloor x \rfloor + \lceil -x \rceil = 0$.
17. Thus, $m + n - 1 < x - x < m + n + 1$.
18. Thus, $-1 < m + n < 1$, and $m + n$ is an integer, so $m + n = 0$.
19. Thus, $x - x \leq m + n < x + 1 - x - 1$, so $0 \leq m + n < 0$.
20. Thus, $x - x - 1 < m + n < x + 1 - x$.
21. Notice that $\lfloor x \rfloor = 3$.
22. Notice that $m \leq x$, by definition of floor of x.
23. Notice that $n \leq -x$, by definition of floor of $-x$.
24. Notice that $m \geq x$, by definition of ceiling of x.
25. Notice that $n \geq -x$, by definition of ceiling of $-x$.

Exercise 9.5. Prove the following by examining all cases.

1. For all $x \in \mathbb{R}$ we have $x - 1 < \lfloor x \rfloor \leq x$ and $x \leq \lceil x \rceil < x + 1$, with equality only for integers.

2. For integers $n \in \mathbb{Z}$, $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$.

3. For integers $n \in \mathbb{Z}$, $\lfloor (n + 1)/2 \rfloor = \lceil n/2 \rceil$.

Exercise 9.6. In section 4.2, item A6 suggested that you could show that "there are no lock or unlock operations in the trace" is equivalent to, "every operation in the trace is an access," if you added the condition that "each trace entry records exactly one of the operations $\{a, l, u\}$ applied by one process to one file." Here are the two statements, using the notation from that section:

$$\forall_i \, \forall_p \, \forall_f \, ((t_i \neq l(p,f)) \wedge (t_i \neq u(p,f))),$$

$$\forall_i \, \exists_p \, \exists_f \, (t_i = a(p,f)).$$

1. Write an expression for the added condition
2. Assuming the added condition is true, prove that the two statements are equivalent.

▶

Puzzle 9.7. Show that no set of 9 consecutive integers can be partitioned into two subsets such that the product of the elements in the first set is equal to the product of the elements in the second set.

▶

Puzzle 9.8. I give you a rectangle of size $n \times m$, where both $n, m > 1$ and their product mn is even, and enough dominoes (1×2 tiles) to cover it. I remove the squares at (x_1, y_1) and (x_2, y_2). Prove that you can cover the other squares with dominoes iff $x_1 + y_1 + x_2 + y_2$ is odd. (What type of proof do you use?) ▶

Puzzle 9.9. A function $f : S^2 \to \mathbb{R}$ is a *distance metric* for set S iff for all choices of x, y, z, it is satisfies three conditions:

1. Non-negative: $f(x, y) \geq 0$ with equality if and only if $x = y$.
2. Symmetric: $f(x, y) = f(y, x)$.
3. Triangle inequality: $f(x, y) \leq f(x, z) + f(z, y)$.

Hint:

E. Semyenov proposes that the *triangle inequality* is redundant, and offers this demonstration.

Suppose that 1 & 2 hold, and assume, for the sake of deriving a contradiction, that 3 does not, but that for all x, y, and z, $f(x, y) > f(x, z) + f(z, y)$. Choose $y = z$ and $x \neq y$, and the triangle inequality says $f(x, y) > f(x, z) + 0 = f(x, y)$. This is a contradiction, since $f(x, y)$ cannot be greater than itself. Therefore, our assumption is wrong—the triangle inequality can't be false while the other two conditions hold.

Do you believe it?

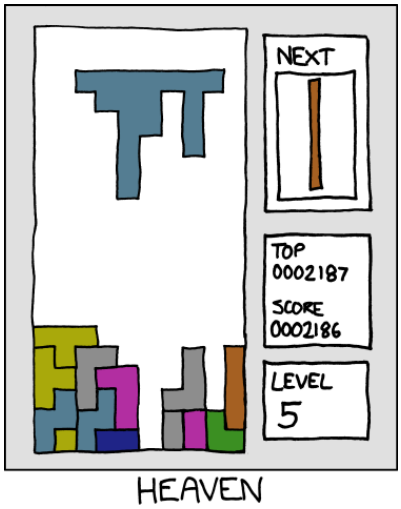
Puzzle 9.10. Alice has fifteen cookies, Beth has nine, and Carla has none, but offers 24 cents to the other two, and each girl eats one-third of the cookies.

Hint:

Beth wants Alice to divide the 24 cents evenly with her, but Alice says that since she brought fifteen cookies to Beth's nine, she should get 15 cents and give Beth 9 cents. What is the fair division of the 24 cents between Alice and Beth?

Extension 9.11. Write formal proofs for assertions or demonstrations of chapter 3 or chapter 7, especially if there are any that you find confusing or doubtful.

Chapter 10
Mathematical Induction



—xkcd.com/888

The proof technique of *mathematical induction* lets us show that a predicate* $P(x)$ is true for every member of a recursively defined set, X. It is ubiquitous in computer science, because so many of our structures and procedures are best defined recursively. I introduce an 8-step template for *strong induction* that confines the thinking primarily to step S7.

Ok, that's true only if after we know $P(x)$ and X – that is, after we can state precisely what we are trying to prove about what. The process of defining the problem and creating the proof go hand-in-hand; the induction template is another tool to break large problems into smaller pieces.

*Recall that a predicate is a function that maps each x to either true or false: $P: X \to \{0, 1\}$.

Objectives: After working through this chapter and the many exercises and puzzles, you will be able to do proofs of $\forall n, P(n)$ by strong induction, following my 8-step template. You will be able to explain what is known and what is to be shown at each step, and find where attempts at induction proofs of false statements fail because they do not fit into the template.

10.1 Strong induction

To prove $\exists_{x \in X}\ P(x)$ is usually easy—we just exhibit some x that works.

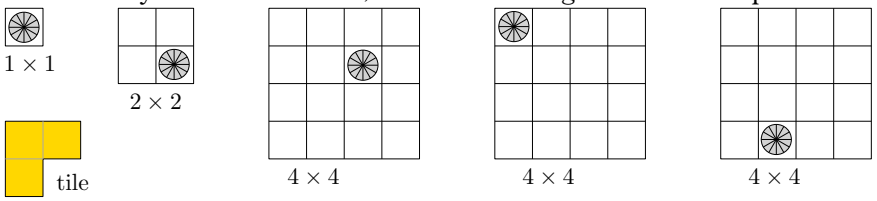
To prove $\forall_{x \in X}\ P(x)$ is harder, because we have to show that every possible $x \in X$ works. As working computer scientists, however, we want to be able to assure clients that, for all possible inputs $x \in X$, our program P computes the true value $P(x)$. We also need to be able to reassure ourselves.

When the set X is recursively defined, such as the natural numbers $\mathbb{N}$, we can essentially set up a computer program that writes a proof that $P(x)$ holds for each possible value of $x \in \mathbb{N}$. This proof often mimics the recursive definition: We establish $\forall x \in \mathbb{N}, P(x)$ by checking a small set of base cases, say $x \in [0..b]$, and then setting up a machine that can show, for any given $n > b$, that $P(n)$ is true, under the assumption that $P(k)$ is true for all smaller $0 \leq k < n$. Let's start with two interesting (that is, difficult) examples.

10.1.1 Examples

Tiling with Ls: A celebrity with initials LL wants to tile her laundry room with gold L-shaped tiles, each three unit squares. The room is a $2^n \times 2^n$ grid of squares, and there is a 1×1 drain at some position i,j, but she doesn't remember exactly where. When the master tile layer asks the clever but lazy apprentice to put the tile cutter on the truck, the apprentice says it won't be needed. Prove that no matter where the drain is, they can tile any $2^n \times 2^n$ grid, with one grid cell removed, by L-shaped tiles without cutting.

Start with small examples, to make sure we understand the question. How would you tile these 1×1, 2×2 and 4×4 grids with L-shaped tiles?

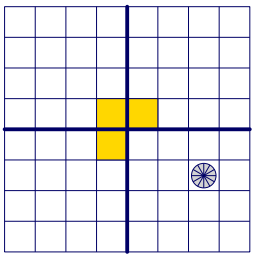


*Large grids may be tiled in many ways; finding one way is enough.

The apprentice wants to demonstrate that for each possible floor (a $2^n \times 2^n$ grid with $n \geq 0$ and one grid cell missing) there exists a way to tile.* Being lazy, the apprentice does *not* want to do this by tiling every possible floor, so divides floors into groups by their size parameter, n, and finds a way to turn the larger task into four smaller instances of exactly the same task.

Lemma 10.1.1. *Any $2^n \times 2^n$ grid with $n \geq 0$ and one grid cell missing can be tiled by three-cell L-shaped tiles without cutting.*

Proof. Let's prove that for all $2^n \times 2^n$ grids with $n \geq 0$ and one grid cell missing, it is possible to tile using L-shaped tiles, by induction on n.



Base $n = 0$: There is only one 1×1 grid, and it needs no tiles, since its only grid cell must already be missing. ✓

Ind. Step: For a given $2^n \times 2^n$ grid missing one cell, with $n > 0$,

Ind. Hyp.: we may assume that any $2^k \times 2^k$ grid that is missing one cell with $0 \leq k < n$ can be tiled using Ls.

To tile a given $2^n \times 2^n$ grid, divide it into four squares of $2^{n-1} \times 2^{n-1}$ by drawing lines through the center. One of the four squares is already missing a grid cell. Place an L-shaped tile at the center to take one grid cell from

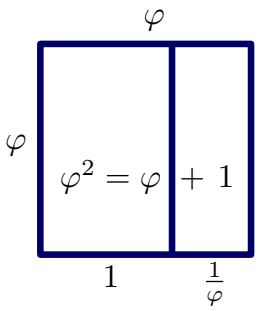
each of the other three squares. By the induction hypothesis, each of these four smaller squares can be tiled, which tiles the entire floor. ☺

Note how alternation of quantifiers affects the proof. An adversary chooses the floor (and n), then we may choose how to tile. We decide to lay a single tile near the center that steals a cell from the three $2^{n-1} \times 2^{n-1}$ squares that were not already missing a cell. We can do this, since $n > 0$. Rather than making more tile decisions, we recognize that we have four smaller copies of the original problem, which can be tiled by the magic of recursion. Try on the grids above.

Fibonacci numbers: The golden ratio, $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$, is a number that shows up in surprising places in nature and art. It is an irrational number, which, as you can check*, happens to satisfy $\varphi^2 = \varphi + 1$, $1/\varphi = \varphi - 1$, and $1/\varphi^2 = 1 - 1/\varphi$. subsection 8.2.5 suggested a surprising closed form expression in φ for the Fibonacci numbers.

*but not on your calculator because that checks approximate equality

Lemma 10.1.2. *For all $n \geq 0$, the nth Fibonacci number $F_n = (\varphi^n - (-\frac{1}{\varphi})^n)/\sqrt{5}$.*



Proof. We can prove this by induction on n.
Base $n = 0$: By definition $F_0 = 0$, and $(\varphi^0 - (-\frac{1}{\varphi})^0)/\sqrt{5} = 0$. ✓
Base $n = 1$: We can check that $(\varphi - (-\frac{1}{\varphi}))/\sqrt{5} = (2\varphi - 1)/\sqrt{5} = 1 = F_1$. ✓
Ind. Step: for a given $n > 1$, we prove that $F_n = \frac{\varphi^n - (-1/\varphi)^n}{\sqrt{5}}$.
Ind. Hyp.: We assume that, for all $0 \leq k < n$, we know that $F_k = \frac{\varphi^k - (-1/\varphi)^k}{\sqrt{5}}$.

The proof expands the recursive definition of Fibonacci numbers for all $n > 1$, uses the induction hypothesis twice (for $k = n-1$ and $k = n-2$), then does some algebra—simplifying expressions with φ according to the above equalities.

$$
\begin{aligned}
F_n &= F_{n-1} + F_{n-2} && \text{defn } F_n \\
&= \frac{\varphi^{n-1} - (-1/\varphi)^{n-1}}{\sqrt{5}} + \frac{\varphi^{n-2} - (-1/\varphi)^{n-2}}{\sqrt{5}} && \text{IH twice} \\
&= \frac{1}{\sqrt{5}} \cdot \left(\varphi^{n-1} + \varphi^{n-2} - (-\frac{1}{\varphi})^{n-1} - (-\frac{1}{\varphi})^{n-2}\right) && \text{factor } \frac{1}{\sqrt{5}} \\
&= \frac{1}{\sqrt{5}} \cdot \left((\varphi+1)\varphi^{n-2} - \left(1 - \frac{1}{\varphi}\right)(-\frac{1}{\varphi})^{n-2}\right) && \text{distrib +*} \\
&= \frac{1}{\sqrt{5}} \cdot \left(\varphi^2\varphi^{n-2} - (\frac{1}{\varphi})^2(-\frac{1}{\varphi})^{n-2}\right) && \text{prop } \varphi^2, \frac{1}{\varphi^2} \\
&= \frac{\varphi^n - (-1/\varphi)^n}{\sqrt{5}}. && \text{prop expon}
\end{aligned}
$$

Thus, we have established the formula for all $n \geq 0$ by induction. ☺

We needed to know the values of both F_{n-1} and F_{n-2} to compute F_n, so $n = 2$ is the first case that can be handled by the induction step, and we need base cases for $n = 0$ and $n = 1$. After that, induction can complete a proof for any value of n that we request, therefore the theorem is true for all n. Note that induction does not tell us how to come up with the formula (for that, see 8.2.5), but it does give a surefire way of proving this formula is correct.

10.1.2 8-step template for strong induction

Here is an 8-step template to work out an induction proof that $\forall_{x \in X}\ P(x)$, where each x has some *size* $n = |x|$. This applies directly to laundry-room floors with x being an $2^n \times 2^n$ grid with one cell missing and $P(x)$ being the claim that "x can be tiled with L-shaped tiles." It applies to the $x = n$th Fibonacci number, with $P(n)$ being the claim that "$F_n = (\varphi^n - (-1/\varphi)^n)/\sqrt{5}$."

Most template steps require little thought, but simply create a framework to help you understand and communicate what you are doing in the steps that do require thought: proving a 'for all $x \in X$' statement by splitting into specific base cases (step S3), and specific general cases (steps S4–S7) that assume that proofs have already been given for all 'smaller' cases.

As you become adept at strong induction, you can omit steps of the template, but when you begin, I suggest that you do even the most mindless of the steps. On an assignment or test, even if you cannot figure out step S7, you at least earn partial credit for doing the other steps right. And writing down the framework prepares you for steps S3 and S7, too.

S1. State the 'for all' statement that you want to prove: Here, $\forall_{x \in X}\ P(x)$.
Sometimes we strengthen the statement of what we want to prove in order to have stronger assumptions in S5.

S2. Say "we prove this by induction on" and state the induction parameter.
I'll use the size $n = |x|$ for this template, but sometimes you have a choice, as in the chocolate bar example coming next.

S3. Prove the base case, often $n = 0$ or $n = 1$ or both.
Here you use the definition to check the truth of $P(x)$ for specific instances of x with small values of the induction parameter. If you have trouble getting an induction rolling, do an extra base case or two.

S4. Write "Induction Step: for a given x with size $n >$ the base cases,"
This mindless step is a reminder that in the induction step you are proving $P(x)$ for a specific, given instance x that has size n. No $\forall x \in X$ quantifier allowed here! You should imagine, however, that it is your adversary who gives you x of size n, since the proof in S5–S7 must work for any given x and n not already handled in a base case.

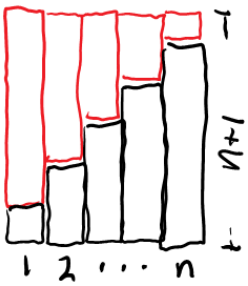
S5. State the Induction Hypothesis (IH): "I can assume, for all y of size k, with base cases$\leq k < n$, that. . . " (e.g., that $P(y)$ is true.)
This repeats the phrasing of S1, but now for all y of size $k < n$ in place of for all x, because while trying to prove $P(x)$, we get to assume that we know $P(y)$ for all ys smaller than x, including the base cases.

S6. State what you are going to prove about your specific value of x of size n that was given to you in S4: e.g., I want to prove $P(x)$.
Again, no $\forall$ quantifier, because we have a specific x of size n to work with.
Sometimes S4–S6 are combined for short proofs.

S7. Do the proof for the specific x and n, often by expanding the basic definition, applying the IH, then doing some calculation.
Once you've chosen what you plan to prove in S1, you don't really have to think until somewhere in the middle of this step.

S8. Declare victory. "Therefore, we have proved $\forall_{x \in X}\ P(x)$ by induction."

For another simple example, let's prove a fact that we already observed in subsection 8.2.3: that the first n positive integers sum to $\frac{n(n+1)}{2}$.



Lemma 10.1.3. *For $n \geq 0$, the sum $\sum_{0 \leq i \leq n} i = \frac{n(n+1)}{2}$.*

Proof. For easier notation, define $S_n = \sum_{0 \leq i \leq n} i$.

1. I want to prove, for all $n \geq 0$, that $S_n = \frac{n(n+1)}{2}$,
2. by induction on n.
3. Base $n = 0$: $S_0 = 0 = \frac{0(0+1)}{2}$. ✓
4. Induction Step (IS): For a given $n > 0$,
5. Induction Hypothesis (IH): I can assume, for all $0 \leq k < n$, that $S_k = \frac{k(k+1)}{2}$,
6. and I want to prove that $S_n = \frac{n(n+1)}{2}$.
7. By definition, $S_n = S_{n-1} + n$, and by the IH with $k = n - 1$, $S_{n-1} = \frac{(n-1)n}{2}$.
   So $S_n = \frac{(n-1)n}{2} + \frac{2n}{2} = \frac{n(n+1)}{2}$
8. ∎

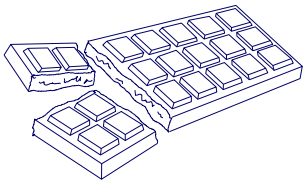
- You should never need ellipses (. . .) in an induction proof; if you do, then you either need to define some notation before you start* or you haven't broken the proof down into elementary steps.
- The claim that you are trying to prove (e.g., $P(x)$ in the template; $S_n = n(n+1)/2$ above) is true or false, so don't do arithmetic on $P(x)$.
- Be sure that you actually use the induction hypothesis.† You get to assume that $P(y)$ is true for all y of size k in base cases$\leq k < n$; if you don't use that, you aren't doing induction. (Many proofs need only that $P(y)$ is true for instances of size $k = n - 1$, but some, like the Fibonacci proof, need more.)

*E.g., to talk about the sum of the first n positive integers, define $S_n = 1 + 2 + \cdots + n$.

†One of the most common mistakes is not to use the induction hypothesis.

- Check that your base case starts at the right place: $n = 0$ or $n = 1$ is common.
- Check that your inductive step (IS) matches with your base cases—the first time through, the IS should use only statements that you've established as base cases. E.g., proofs about Fibonacci numbers usually need two base cases.
- Taken together, the base cases and the induction step cases must cover the entire quantification domain X.
- We get to assume for all y smaller than x, that $P(y)$ is true, and try to prove $P(x)$ is true. Don't assume $P(x)$ and try to prove $P(y)$.
- More generally, don't confuse the roles of x and n with y and k in S5–S7. In step S4, your adversary gives you one generic, non-base-case value x of size n in S4, and it does not change in S5–S7. You may consider any or all ys of size $k < n$ in those steps and know from the induction hypothesis (IH) that $P(y)$ is true.
- You don't need closure for induction—we don't care if there are other cases where the theorem is true that aren't covered by our proof. You do need closure for recursive definition—we don't want extra elements being tossed into our sets, languages, structures, or relations.

Here is a less numerical example. Suppose that you have a chocolate bar made up of r squares by c squares that you want to share. At each step, you choose a piece of chocolate with two or more squares and break along a vertical or horizontal line between squares. Eventually, it will be reduced to single squares. Does it make a difference if you break the long or the short way?



We can prove by induction that the number of breaks required is $rc - 1$, no matter which way you break. Decide what would you write in these boxes before looking at my answers. Note how the first two steps reduce the pair r, c to a single induction parameter.

Answers in boxes are hidden in pdf. If you printed, cover the boxes until you've provided your own answers.

1. We want to prove that, for any chocolate bar C with $n = rc \geq 1$ squares, and any way to break it, it will be reduced to isolated squares after exactly $b(C) = n - 1$ breaks.

2. We prove this by induction on [ ].

3. Base case $n =$ [ ]: (Do the base case) [ ], which is right since it is already an isolated square.

4. Ind Step: Consider a given chocolate bar C with [ ] squares (an inequality),

5. IH: We may assume that, for all bars D of k squares with [ ] (upper & lower bounds), the number of breaks $b(D) =$ [ ].

6. I want to prove that the number of breaks $b(C) =$ [ ]. (Should I add ∀? [?])

7. For the n-square chocolate bar, C, with one break I get two smaller parts, say A with m squares and B with $n - m$ squares.

   By IH, $b(A) =$ [ ], and $b(B) =$ [ ].

   Therefore, $b(C) = 1 + b(A) + b(B)$
   $= 1 +$ [ ] $+$ [ ] $= n - 1$.

   This establishes the induction step,

8. and by induction we have proved that for all chocolate bars with n squares, exactly $n - 1$ breaks will turn it into isolated squares. ∎

10.2 Variants

There are several important variants of induction that can be used once you master the basics of the 8-step template. I present examples of the following four variants in this section:

You may want to do a few exercises to help master the basics before you work through this section.

Weak induction: In many induction proofs, only the $k = n - 1$ case of the induction hypothesis is needed in the induction step. As a result, most introductions to induction ask students to prove $P(n) \rightarrow P(n + 1)$ in the inductive step, known as weak induction.
In computer science we often use divide and conquer: solving a problem or forming a data structure by combining two or more smaller instances that are not necessarily of the previous size (e.g., binary trees, subsection 8.1.6). Since these call for strong induction, I see no need to introduce different notation for weak induction. I prove one result using both strong and weak induction in subsection 10.2.1.

Nested quantifiers: As mentioned above, sometimes the "for all" statement we wish to prove actually depends upon other quantified variables that must be chosen first. subsection 10.2.2 illustrates this with a Tetris example where it is easy to see the solution, but fiddly to prove it formally.

Strengthening the Induction Hypothesis: Paradoxically, it is often easier to prove a stronger statement than we actually need. Because we get to assume that the statement we are trying to prove is true for smaller instances of the problem, strengthening the statement gives us a stronger Inductive Hypothesis. subsection 10.2.3 continues the Tetris example to demonstrate this.

Minimal counterexample: One of my favorite variants combines proof by contradiction with induction in the following way: you assume that a counterexample of the theorem exists, and choose the smallest one, under an appropriate, discrete notion of smallest. You show that the smallest (base) cases are not counterexamples. Then you show that any counterexample that is not a base case can be manipulated to produce a smaller counterexample. But this is a contradiction that shows that the falsity of the assumption that a counterexample exists. See subsection 10.2.4, Lemma 12.1.1, and Lemma 12.2.2 for examples.

The rest of this book uses induction; here are three more examples coming later:

Invariants: chapter 11 has several examples of induction used to establish algorithm invariants—statements that remain true while an algorithm is running. In the induction step, your adversary gives you some input and state of the data structures at a particular step, and you must show that your algorithm does the right thing in its next step. You often need to strengthen the statement of the invariants so you can assume a strong enough Induction Hypothesis that your proof can go through. Refining the invariants and their proof identifies and avoids possible coding errors, which would be much more expensive to identify and fix in the test and debugging phase.

Induction on more than one variable: section 11.5 shows that an algorithm with two parameters terminates by induction on the pair—it gives three options to do so on page 208.

Equivalent structures: subsection 13.5.1 shows that three definitions of rooted trees actually give top-down, bottom-up, and global views of exactly the same set of structures. The formal proof ensures that we haven't missed anything in the boundary cases of these definitions, which is easy to do without the assistance of the formality.

10.2.1 Weak vs. strong induction

You may have previously seen *weak induction*, which asks you in the inductive step to prove, for all n, that $P(n) \Rightarrow P(n+1)$. I prefer *strong induction*, which assumes that for all $k < n$ we know $P(k)$ is true to show $P(n)$ is true. Since strong induction applies easily in every case that weak induction applies, and in some cases that weak induction does not, I encourage you to use strong induction, too.* The first two examples of the previous section, summing $1 + \cdots + n$ and tiling with Ls, can use either weak or strong induction; the other two examples are easier with strong induction. The Fibonacci closed form in Lemma 10.1.2 needs the formula to hold for the *two* previous

*Weak induction can mimic strong for $P(n)$ by proving that $\hat{P}(n) = \bigwedge_{j \le n} P(j)$ is true for all n; strong induction does not need to complicate the predicate to be proved.

numbers. The chocolate bar needs the formula to hold for *all* bars with a smaller number of rows or columns. Even for the tiling, weak induction emphasizes the number in the exponent rather than that we are given a tiled floor.

Here is an example showing first a weak and then a strong induction proof of the same lemma, in similar an 8-step forms. The weak induction starts with a base case for 12 and shows that you can get from postage of n cents from knowing the postage of $n-1$ by either replacing a 4 with a 5, or else by replacing three 5s with four 4s, using some additional notation to help describe this operation. The strong induction starts with four base cases for [12..15], and eventually reduces every number to one of them. The weak induction may seem more "mathematical" because we must think harder when only the immediately preceding case is assumed in the IH. I find the strong induction proof simpler.

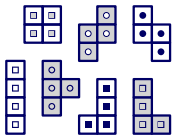
Lemma 10.2.1. *Any amount of postage 12 cents or more can be formed using 4- and 5-cent stamps.*

For the proof by weak induction, we restate the lemma to include some non-negative integer variables.

1. *State what you want to prove,*	We want to show $\forall n \geq 12\ \exists a, b \in \mathbb{N}$ such that $n = 4a + 5b$,
2. *State induction parameter.*	by induction on n.
3. *Do Base case(s).*	for $n = 12$: Since $n = 3 \times 4$, we set $a = 3$ and $b = 0$.✓
4. *State "Ind. Step: for given $n >$ base,"*	Ind. Step: for a given $n > 12$, we may assume
5. *State IH, true for all base$\leq k < n$.*	IH: $\exists a, b \in \mathbb{N}$ with $n - 1 = 4a + 5b$.
6. *State "We want to prove", for given n.*	We want to prove that we can form n cents.
7. *Do the proof,*	From the IH, assume there are $a, b \in \mathbb{N}$ with $n - 1 = 4a + 5b$. If $a > 0$ then $n = (a-1)4 + (b+1)5$ has $(a-1), (b+1) \geq 0$. Otherwise $a = 0$, and since $n > 12$, we know $b \geq 3$. Then $n = (a+4)4 + (b-3)5$ has $(a+4), (b-3) \geq 0$.
8. *and declare victory.*	☺

The proof by strong induction needs no restatement.

1. *State what you want to prove,*	Prove $\forall n \geq 12$ we can make n-cents from 4- and 5-cent stamps,
2. *State induction parameter.*	by induction on n.
3. *Do Base case(s).*	We can make $n \in [12..15]$ easily:
	$12 = 3 \times 4$✓ $\quad 13 = 2 \times 4 + 5$✓
	$14 = 4 + 2 \times 5$✓ $\quad 15 = 3 \times 5$✓
4. *State "Ind. Step: for given $n >$ base,"*	Ind. Step: for a given $n > 15$,
5. *State IH, true for all base$\leq k < n$.*	IH: $\forall 12 \leq k < n$, we can make k cents from 4- and 5-cent stamps,
6. *State "We want to prove", for given n.*	We want to prove that we can form n cents.
7. *Do the proof,*	Choose $k = n - 4$. Since $n \geq 16$, we have $12 \leq k < n$.
	By the IH, we can make k cents with 4- & 5-cent stamps.
	Add a 4-cent stamp to make n.
8. *and declare victory.*	☙



10.2.2 With nested quantifiers

The Tetris pieces are the seven different ways to connect four squares, allowing rotations but not reflections. We play original (no gravity) Tetris—when a row disappears every block above moves down one row. Suppose that your Tetris program gives you zigs, and zigs only. You can still rotate them to be horizontal or vertical. Can you play forever? Does your answer depend on the well width (number of columns) w? We will see in this subsection that for even w we can play forever, and in the next subsection that for odd w we cannot.

In pictures and in proofs, it may be easier to think about just crossing out rows that would otherwise disappear. To compensate, rather than having a fixed height $h_{\max}$, we play in an infinitely tall well of width w. Later pieces teleport over crossed-out rows as if they weren't there. We then say that we can play forever in a well of width w iff there is a strategy σ and a height h such that every grid cell above h is eventually covered by some piece i. The negation would say that we cannot play forever in width w iff for any strategy σ and chosen h, there exists some grid cell g above h that all pieces i fail to cover.

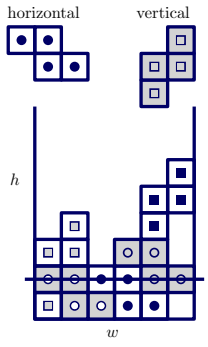


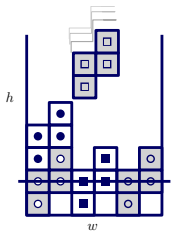
Figure 10.1: Tetris with zigs

Notice the alternation of quantifiers in this claim that for all even w we can play forever: $\forall$ even w, $\exists$ strategy σ, $\exists h \geq 0$, $\forall$grid cells g, $\exists i$ with grid $g_y > h$ and zig i covering g. As mentioned in subsection 4.1.3, it is this alternation of quantifiers that makes problems difficult; we get a different statement if we change the order.

In the rest of this section we show formally that for all even $w > 0$ there is a strategy and a height h, so that all grid cells above h are eventually covered. Our adversary will first choose w, then we can choose the strategy and h to depend upon w, then our adversary can ask about any grid cell

above h and we are supposed to demonstrate that it will be covered. In fact, we can choose $h = 1$; only the strategy needs to depend upon w.

Idea of the strategy: Once the adversary chooses an even width w, we want to show 'there exists' a height and strategy. We choose $h = 1$ and a strategy that plays all the zigs vertically and keeps filling in the lowest reachable spot that is farthest left. We can see the pattern pretty easily: the first row will not be fully covered, but every row after $h = 1$ will be fully covered. This is enough to convince someone that a proof exists. For practice, I spell out this strategy in detail and prove that it works for any even $w > 0$ that is given to me.



Lemma 10.2.2. *Playing original Tetris with width w and zigs only, for all even $w > 0$ there is a strategy and a height h so that all grid cells above h are covered.*

Proof. Start by choosing $h = 1$ and by making the strategy more formal: group the w columns into strips of 2, numbered $2m - 1$ and $2m$ for $1 \leq m \leq w/2$. The strategy is to play a vertical piece in each strip in turn, with the ith piece in strip m covering grid cells $(2m-1, 2i-1), (2m-1, 2i), (2m, 2i), (2m, 2i+1)$.

We can invert this to calculate which zig covers a cell (x, y) with $y > 1$: it must be in strip $\lceil x/2 \rceil$, and will be the $\lfloor y/2 + (x \bmod 2) \rfloor$th zig in that strip. By looking at the even and odd columns separately, we can see that no two zigs in a strip overlap, and since every grid cell with $y > 1$ is in a zig, this strategy works. ∎

Notice how in this problem being formal makes us pin down the strategy to the point that we could even write it as computer program. This program depends on the width w, though, so we are asking the reader of the proof to check that it works for all even w. We can be even more formal and use induction to check all even $w > 0$. Nested within that proof, once we choose the strategy and $h = 1$, we find a statement "for all cells (x, y) with $y > h$," so we use a second induction to prove that.

Proof by nested induction. We again choose $h = 1$ and the strategy of playing vertical zigs in strips. We want to prove that for all even $w > 0$ all grid cells above h are covered. We do this by induction on w.

Base $w = 2$**:** Here we want to prove that every cell (x, y) with $y > h$ is covered. Since this is again a 'for all', a formal proof can use induction (yes, within the base case of the induction on w.)

> Let's prove that using n vertical zigs covers all cells up to $2n$ in column 1 and cells between 2 and $2n + 1$ in column 2. Check the base case for $n = 1$. Now, for any chosen $n > 1$, we assume that $n - 1$ pieces cover up to $2n - 2$ in column 1 and $2n - 1$ in column 2. Adding that last vertical zig adds two to both columns. Thus, the result we wanted is

true by induction. (Note: we actually prove more than we wanted—our result describes the shape after n zigs, and all we care is that things eventually get covered. But being able to assume the shape in the IH really helps.)

Ind Step: Now we are back to our given even $w > 2$, and we may assume:
IH: For all even $0 < k < w$, in a game of width k, the vertical zig strategy covers all cells above $h = 1$.

We want to show that the strategy works for width w. But we can partition w into a game of width 2 and one of width $0 < w - 2 < w$, and apply the induction hypothesis to both games. This completes the proof. ∎

10.2.3 Strengthen what is to be proved

We should expect this to be harder since our adversary has many choices, not just picking the width as in the previous subsection.

For odd column widths w, we can turn the controls over to our adversary and be sure that, no matter their strategy, they cannot play forever. That is, for any odd w, our adversary can choose any strategy and height h, and there will be some cell above h that is never covered. We prove this by induction on w.

What is a strategy? We can think of it as a sequence of where each zig in turn is to be placed: whether it is played horizontal or vertical, and the position in the grid when it is dropped (by specifying the position of a distinguished square, or the lowest square in the rightmost column). The strategy must respect the rules, and not overlap zigs or extend zigs outside this well.

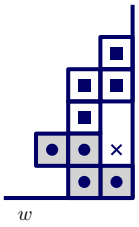


Figure 10.2: Placing zigs along right wall

Here is the first key idea: look at the zigs that a chosen strategy places touching the right wall. Each one that is placed horizontally leaves a square (marked x in figure 10.2) that cannot be covered: the square below (since we are playing with no gravity) protects it from the bottom, the square on the same row cannot disappear until the x is filled, and neither a horizontal nor a vertical zig can fill it from the top. Thus, if our adversary claims to have a strategy that covers all grid cells above height h, then above h they must play only vertical zigs in the right two columns. We can imagine the adversary's strategy as playing the other pieces above h in a width $w - 2$ game.

So we would like to run the adversary's strategy and just ignore all zigs that don't extend above h or outside the right two columns. There is a subtle problem; can you see it in figure 10.3? In this smaller game, we are not starting the width $w - 2$ game with an empty well, because some zigs at h or $h - 1$ may be poking into it from below. The second key idea is to again strengthen the statement that we want to prove: allow the adversary to add any collection of initial blocks in the first two rows. Proving a stronger theorem gives us a stronger assumption in the induction hypothesis.

Lemma 10.2.3. *Playing original Tetris with width w and zigs only, for all odd $w > 0$, for all strategies, heights $h \geq 0$, and initial choice of blocks for the first two rows, there exists some grid cell above h that is not covered.*

Proof. We prove this by induction on w.

Base: $w = 1$; In a single column you can't play any zigs at all. The adversary can block up the first two rows, but by row $\max(3, h + 1)$ you will have an uncovered grid cell above h.

Ind Step: for an odd width $w > 1$, I get to assume:

Ind Hyp: For all odd widths $0 < v < w$, for all strategies, for all initial two rows, and $\forall j$, there is some uncovered cell above height j.

Now, for width w, I want to show that for all strategies, for all initial two rows, and $\forall h$, there is some uncovered cell above height h.

Consider the zigs played that touch the right wall. If any zig is played horizontally above height h, then it protects an uncovered cell, as I argued with figure 10.2. Therefore I need to worry only about strategies in which all vertical zigs are played in the right two columns above h. In such a case, however, I cut off the right two columns and all rows h or below, and view the strategy as playing a smaller game of width $w-2$ that may start with the fragments of cut zigs in its first two rows, as in figure 10.3. By the induction hypothesis, there will be an uncovered cell in this game. ∎

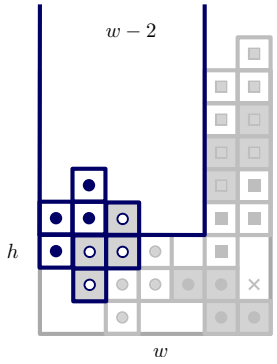


Figure 10.3: Make IH stronger

There are other possible proofs, but many of them require reordering the zigs in a strategy, which really needs an aboveness relation that I don't define until section 12.2. By strengthening the claim, I avoid that completely.

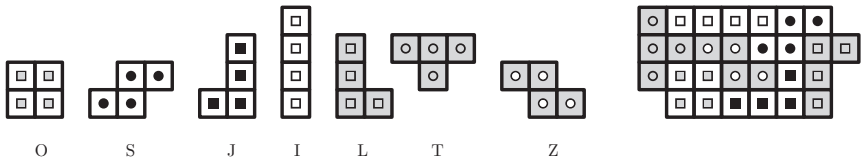


Figure 10.4: Sometimes impossibility proofs are easy, when you get the right idea: Prove that you cannot pack the 7 Tetris pieces into a 4×7 rectangle.

10.2.4 Minimal counterexample

One of my favorite variants combines proof by contradiction with induction: proof by a minimal counterexample. Here is a mathematical example.

A list of real number *coefficients*, $(a_o, a_1, \ldots, a_n)$ with $a_n \neq 0$, gives us a non-zero *polynomial of degree n*: $p(x) = \sum_{0 \leq i \leq n} a_i x^i$. A real number r is a *root of polynomial p* iff $p(r) = 0$. We can show:

Lemma 10.2.4. *Any non-zero polynomial of degree $n \geq 0$ has at most n roots.*

Proof by minimum counterexample. Suppose, for the sake of deriving a contradiction, that there exists at least one non-zero polynomial with more roots than its degree. From all such examples, choose a polynomial $p(x) = \sum_{0 \leq i \leq n} a_i x^i$ whose degree n is minimum, and let $(r_0, r_1, \ldots, r_n)$ be $n+1$ distinct roots of p.

Proof that every number in $\mathbb{N}$ is interesting.
Base: 0 and 1 are interesting (as Booleans, or identities for + and ×.)
Suppose that there are uninteresting numbers in $\mathbb{N}$. Let k be the smallest such; that is an interesting property.

Notice that the degree of p cannot be $n = 0$, because the non-zero polynomials of degree 0 are constants, and have no roots.

Now, form the polynomial $q(x) = a_n(x - r_1)(x - r_2)\cdots(x - r_n)$. As polynomials, $p \neq q$, since $p(r_0) = 0$, but $q(r_0)$ is a product of non-zero terms. The difference, $p - q$, is therefore a non-zero polynomial, and since the $a_n x^n$ terms cancel, it has degree less than n. But it has n roots, $(r_1, r_2, \ldots, r_n)$, so would be an example of smaller degree, which contradicts the minimality of p.

Thus, any non-zero polynomial of degree $n \geq 0$ has at most n roots. ☙

We can do a proof by induction going forwards, but then we have to know how polynomial division works.

Proof by forward induction. We prove that any non-zero polynomial of degree $n \geq 0$ has at most n roots by induction on n.
Base: When $n = 0$, for all reals x, the polynomial $p(x) = a_0 \neq 0$, by conditions of the problem. So p has no roots. ✓
Ind. Step: for a given non-zero polynomial p of degree $n > 0$, we may assume as
Ind. Hypothesis: any non-zero polynomial of degree $0 \leq k < n$ has at most k roots.
We want to prove that the given polynomial p has at most n roots. If p has no roots, then we are done, so we worry about the cases in which p has a root r.

Use polynomial division (just like long division for numbers) to obtain the polynomial $q(x)$ satisfying $p(x) = (x - r)q(x)$. If you are used to polynomial division, this is fine, otherwise set up the division and work out some details to make sure that $q(x)$ is indeed a polynomial, and that the remainder is zero. You should find that the highest coefficient of $q(x)$ is $b_{n-1} = a_n$, and the rest, for $i \leq n$, can be generated recursively as $b_{i-1} = a_i + r * b_i$, which happens to give $b_{-1} = p(r) = 0$.

Now, since $p(x) = (x - r)q(x)$, it is not hard to see that any number that is not a root of q and not r cannot be a root of p. The induction hypotheses says that q has at most $n - 1$ roots, so p has at most n roots. (Fewer if r is already a root of q.) ☙

10.3 Summary

Induction is important in computer science because we often want to know that an algorithm or data structure works correctly (and efficiently) for all possible inputs. The next chapter gives several examples of how induction is used to establish invariants—properties of our algorithms or data structures that remain true while our algorithm is running. Iterative algorithms can often get by with weak induction, but recursive algorithms often need strong

induction, which is why I strongly encourage you to use the template for strong induction.

The template emphasizes that you are trying to prove a property of all elements in a set by assigning each element a size (usually an integer). The base cases then prove that the smallest elements have the property. The inductive step proves that an element has the property under the assumption that all smaller elements have the property. As a computer scientist, I can view an induction proof as a program that can generate a valid proof that any particular element has the property.

The usual problems when teaching a proof technique is that one must start with simple proofs that don't show the many variations and complexities of the technique. The examples in the text are relatively complex and subtle; some of the exercises and puzzles for this chapter are even more complex. Read them more than once.

10.4 Exercises and Explorations

Exercise 10.1. Can you show that the first n odd integers sum up to n^2?

We want to show that $1 + 3 + 5 + \cdots + 2n - 1 = n^2$, except this formula is not quite right at $n = 0$.

Stating it correctly with a recursive definition instead of ellipses: Let $S_0 = 0$, and for integers $n > 0$ define $S_n = S_{n-1} + 2n - 1$. Prove by induction that for all $n \geq 0$, $S_n = n^2$. Summation notation also works: show for all $n \geq 0$ that

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

I urge you to follow the template; there are many example proofs on the web that are not always careful about where they start, or about use of n and k, or that they are proving a 'for all' statement.

▶

Exercise 10.2. What is the sum of the first n squares? For all integers $n \geq 0$, prove that $1^2 + 2^2 + 3^2 + \cdots + n^2 = n(n+1)(2n+1)/6$. In summation notation, show that

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

Try to use the induction template to make sure you handle the ns and ks in a correct manner. There is nothing magical about those variable names, but good habits help us to be correct without thinking hard. ▶

Exercise 10.3. Is the sum of the first n cubes equal to the square of the sum of the first n natural numbers? Show that:

$$1^3 + 2^3 + 3^3 + \cdots + n^3 = (1 + 2 + 3 + \cdots + n)^2$$

Define notation to avoid the ellipses: either by summation Σ or recursive definition of prefix sums. You may get the value of the right hand side from Lemma 10.1.3.

Exercise 10.4. An *arithmetic series* $a_1, a_2, \ldots, a_n$ is a series in which the differences between consecutive terms are the same: $\exists d \, \forall 1 \leq i < n \, (a_{i+1} - a_i = d)$. Show that the sum of an arithmetic series with n terms is just n times the average of the first and last terms:

$$\sum_{1 \leq i \leq n} a_i = n \frac{a_1 + a_n}{2}.$$

Exercise 10.5. A *geometric series* $a_1, a_2, \ldots, a_n$ is a series in which the ratios between consecutive terms are the same: $\exists r\, \forall 1 \leq i < n\ (a_{i+1} = ra_i)$. Show that the sum of an geometric series with n terms is the following combination of first and last terms:

$$\sum_{1 \leq i \leq n} a_i = \frac{a_1 - ra_n}{1 - r}.$$

Exercise 10.6. What is the value of the binary number that is a string of n ones? Define $S_0 = 0$ and for $n > 0$, define $S_n = 2S_{n-1} + 1$. Show that $S_n = 2^n - 1$.

Do an induction to determine a closed form expression for the value for this sequence that is defined recursively. An expression in *closed form* uses a fixed number of standard operations, like arithmetic, factorial, binomial coefficients, but no summation or recursion in which the number of operations grows as n grows.

▶

Exercise 10.7. What do the first n fractions of the form $1/(i(i+1))$ sum to? Show $\forall n \in \mathbb{N}$ that $\frac{1}{1\cdot 2} + \frac{1}{2\cdot 3} + \cdots + \frac{1}{(n-1)\cdot n} = \frac{n}{n+1}$. In summation notation, show that

$$\sum_{1 \leq i < n} \frac{1}{i \cdot (i+1)} = \frac{n}{n+1}.$$

Feel free to add notation, like defining S_n as the sum. It may help to write the fraction $1/(i(i+1))$ as the sum of two fractions.

Go ahead and start filling in the template without thinking, however; you'll need to think only in step S7.

Exercise 10.8. Prove that for all $n > 3$ we have $2^n < n! < n^n$.

Exercise 10.9. For what integers n is $n^2 \leq 2^n$?

You'll have to do a few base cases, then induction can take over.

Exercise 10.10. Can euro 2 and 5 cent coins make any value greater than 3? For all $n \geq 4$, show that there exist non-negative integers h and k such that $2h + 5k = n$.

Warning: Be careful not to re-use k in the induction proof; you'll want to rename either this variable or the one in the induction template. ▶

Exercise 10.11. Prove that for all non-negative integers $n \in \mathbb{N}$, the number $n^3 + 2n$ is divisible by 3. Recall that integer a is divisible by d if there exists an integer m such that $a = md$.

Exercise 10.12. Prove Lemma 7.3.1 by induction on the number of medi-ants created. ▶

Exercise 10.13. What is wrong with Polya's classic proof that all horses are the same color? I'd suggest rewriting this to follow my template and see if that clarifies what goes wrong.

CLAIM: In any set of h horses, all horses are the same color.

Faulty proof. by Induction on h.

Base: For $h = 1$. In any set containing just one horse, all horses clearly are the same color.

Induction Step: For $k \geq 1$ assume that the claim is true for $h = k$ and prove that it is true for $h = k + 1$.

Take any set H of $k + 1$ horses. We show that all the horses in this set are the same color. Remove one horse from this set to obtain the set H_1 with just k horses. By the induction hypothesis, all the horses in H_1 are the same color. Now replace the removed horse and remove a different one to obtain the set H_2. By the same argument, all the horses in H_2 are the same color. Therefore, all the horses in H must be the same color, and the proof is complete. ∎

▶

Exercise 10.14. What is wrong with this induction proof that $a^n = 1$ always?

Claim: Let a be any positive number. For all positive integers $n \in \mathbb{Z}^+$, we have a $a^{n-1} = 1$.

Faulty proof. by induction on n

Basis: If $n = 1$, we have $a^{n-1} = a^0 = 1$. ✓

Inductive Step: We want to prove this for an integer $n > 1$.

Inductive Hypothesis: Assume, for all natural numbers $k < n$, that $a^{k-1} = 1$.

We now want to show that it is true for n by writing

$$a^{n-1} = \frac{a^{n-2} \cdot a^{n-2}}{a^{n-3}},$$

since $n - 1 = (n-2) + (n-2) - (n-3)$. But if we use the inductive hypothesis twice, for $k = n - 1$ and $k = n - 2$, we can replace $a^{n-2} = 1$ and $a^{n-3} = 1$, so $a^{n-1} = 1 \cdot 1/1 = 1$. ∎

▶

Exercise 10.15. Can you show that *not* and *and* suffice to express all logic formulae? Consider a well-formed logical formula (wff), using the operations of negation, $\rightarrow$, $\vee$, and $\wedge$ and full parentheses. The set of all wffs can be defined recursively: A logical variable is a wff, and if f and g are wffs, then the following four statements are also wffs: $(\overline{g})$, $(f \rightarrow g)$, $(f \vee g)$, and $(f \wedge g)$. Prove one of the following using mathematical induction on the length of a wff.

1. For any wff there is a logically-equivalent wff using only the operations of negation and $\wedge$ and parentheses.
2. For any wff there is a logically-equivalent wff using only the operations of $\wedge$ and $\vee$ and parentheses applied to logic variables or their logical negations.

Puzzle 10.16. Think of a number, square it, write it down in base 10, and add up the digits. For example, $13^2 = 169 \rightarrow 16$. Repeat until you get into a cycle, e.g., $16^2 = 256 \rightarrow 13$. Show that every number reduces to one of three cycles. Can you also figure out which cycle a number will end at without going through the squaring and reduction process?

Puzzle 10.17. In how many ways can you tile a $2 \times n$ rectangle with dominoes? Dominoes, spots down, are 2×1 rectangles, or 1×2 if rotated. How many ways can you tile a $2 \times n$ rectangle for different values of n? For small values of n, the answers are:

n	1	2	3
#	1	2	3

But this pattern does not continue.

Puzzle 10.18. Can you show that the number guessing game has a worst case of exactly $\lceil \log_2 n \rceil$ guesses?

I'm thinking of a natural number x between 1 and n. You can ask questions of the form, "Is x larger than a?" and I will tell you (truthfully) yes or no. Show that, in the worst case, $\lceil \log_2 n \rceil$ questions are necessary and sufficient to exactly determine my number.

For sufficiency, you may want to argue that you can round n up to the next power of 2 to make your proof easier: e.g., new $n = 2^{\lceil \log_2 n \rceil}$.

To show that this many are necessary is the harder direction. You'll want to imagine that you delay picking a number until forced to by the guesser's questions. Pick your answers to keep the maximum number of options open to you. ►

Puzzle 10.19. Suppose there are n identical cars stopped on a circular track, and that the total gas in all tanks is enough for one car to make it around the track (even if it had to stop and start n times.) Show that there is at least one car that can make it around if it siphons all the gas from each of the stopped cars it passes. ►

Puzzle 10.20. Nim is a simple two players game with two stacks of chips, matches, or other counters. At each turn, a player removes some (non-zero) number of counters from one of the stacks. In this variant, the player who removes the last match wins.

If the two piles contain the same number of matches at the start of the game, show that the second player can always win with the following strategy: always take the same number of matches that your opponent took, but from the other pile.

Puzzle 10.21. Take the red cards (hearts and diamonds) from a deck, shuffle them, pair them up arbitrarily, then glue pairs back to back. Show that you can still lay out the pairs so that you'll see, face up: A, 2, 3, . . . , 10, J, Q, K. (The suits showing do not matter.)

Generalize to a $2n$-card deck, consisting of $1 \ldots n$ of hearts and diamonds.

Play with a 10 card deck (A..5) on scratch paper until the problem is clear.

Puzzle 10.22. In the city of Arrangemént, the old city wall has been replaced by a one-way ring road, and all other roads are one way, starting and ending on the ring road. Show that there must be some "block" that you can legally circle, respecting the one-way arrows. (Blocks in Arangemént may have odd shapes, but have no roads through them.)

Puzzle 10.23. Ten pirates are going to split 100 gold coins. The pirates have a strict seniority. The oldest makes a proposal for a division of the coins, and if it acceptable to half the pirates, including himself, they divide the coins as proposed. If not, the remaining pirates make the proposer walk the plank. Pirates are smart, greedy, and bloodthirsty: each has first priority of staying alive, second of getting as most money as possible, and third of making others walk the plank if it doesn't conflict with first or second priority. What should the oldest propose to get the most money?

Does the proposal change if the proposer is not allowed to vote? (That is, if the proposal needs a strict majority to accept?)

Extension 10.24. Using properties from Subsections 7.2.1 and 7.2.2, prove the "Fundamental Theorem of Arithmetic" that every integer $n > 1$ can be written as the product of one or more primes in an increasing sequence in exactly one way (up to order).

1. Initially drop the "in exactly one way," and just show that every integer $n > 1$ can be written as a product of primes.
2. Then prove the full theorem, perhaps by minimal counterexample.

Extension 10.25. Prove the *Cauchy-Schwartz inequality* by induction:Given two sequences of reals, $(a_1, a_2, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_n)$,

$$\Big(\sum_{1\le i\le n} a_i b_i\Big)^2 \le \Big(\sum_{1\le j\le n} a_j^2\Big)\Big(\sum_{1\le k\le n} b_k^2\Big).$$

Extension 10.26. Prove the *Pigeonhole Principle* by induction:That is, show that for all integers $n \ge 1$, for all sets A of $n+1$ elements, all sets B of n elements, and all functions $f\colon A \to B$, there exist elements $a_1, a_2 \in A$, with $a_1 \ne a_2$, such that $f(a_1) = f(a_2)$. ▶

Extension 10.27. Prove *Fermat's little theorem*, Theorem 7.2.2, by induction on m: For all primes p and positive integers $m \in Z^+$, the exponential $m^p \bmod p = m \bmod p$. ▶

Extension 10.28. Prove the *Chinese remainder theorem*: given integers $n_1, n_2, \ldots, n_k$ that are pairwise relatively prime (no two have a common factor) and have product $N = \prod_{i\in[1..k]} n_i$, any integer $m \in [0, N)$ is uniquely determined by its remainders $(m \bmod n_i)_{i\in[1..k]}$.

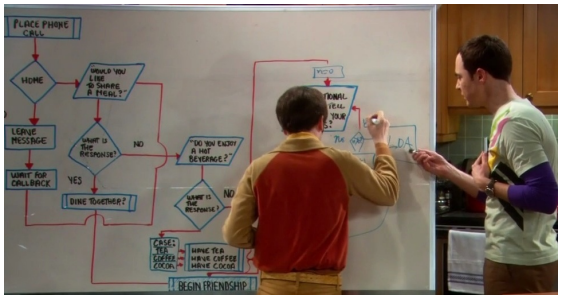
You'll want to choose good notation, use induction on k, and use Lemma 7.3.2 within the induction step.

Exploration 10.29. An interesting recursive way to represent every non-negative integer $n \in \mathbb{N}$ as a set S_n is the following:

- Base: $S_0 = \emptyset$ represents 0.
- Recursive rule: For $n > 0$, let $S_n = S_{n-1} \cup \{S_{n-1}\}$ represent n.
- Closure: All sets representing integers come from the base case, by a finite number of applications of the recursive rule.

1. Show that S_n has n elements.
2. Show that $i < j$ iff $S_i \subseteq S_j$.
3. Show that for S_i and S_j we have three choices: either $S_i \subseteq S_j$, $S_i = S_j$, or $S_j \subseteq S_i$.
4. Recursively define functions for addition and multiplication.

Chapter 11
Algorithms and invariants



A loop counter and an escape to the least objectionable activity. Howard, that's brilliant! I'm surprised you saw that.

—Dr. Sheldon Cooper, Big Bang Theory, ep. 2–13, "The Friendship Algorithm"

Induction plays a crucial role in computer science because we use it to establish *invariants*—statements that remain true while an algorithm is running. I demonstrate by asking you to derive four important algorithms:

P1 Given an array $A[0..n-1]$ of n numbers, find the maximum.

P2 Binary search a sorted array $A[0..n-1]$ to determine where a given x belongs using $\lceil \log_2(n+1) \rceil$ comparisons.

P3 Sort a given array $A[0..n-1]$ of n numbers into non-decreasing order, $A[0] \le A[1] \le \cdots \le A[n-1]$. (We'll use insertion sort.)

P4 Find the greatest common divisor of two non-negative integers, $gcd(a, b)$.

Objectives: By working through a series of questions about four algorithms in this chapter, you will see the importance of clearly and precisely stating the input and desired output, and how identifying invariants can help us derive an algorithm, in much the same way that we derive a proof. On mastering this material, you will be able to derive algorithms from invariants that will run correctly (once you fix the inevitable typos), and recoup the time spent learning this technique by reducing the time you spend debugging.

11.1 Preliminaries

To write down the steps of an algorithm we use *pseudocode* with operations common to many programming languages: logic ($<$, $>$, ==,!*not*, &*and*, |*or*), calculation (+,−,∗,/), increment (++,−−), assignment (=), indexing[], and function calls. For control flow (if-else-endif, for, while) we use C++/Java or

higher-level languages; we allow "for all i in" a set or list. Lines mentioned in the text are numbered.

Most classroom programming assignments are simple, and we can simply write down the steps of a algorithm. That is, we say what to do, and expect others to infer what is true before, during, and after the algorithm executes. This is poor communication because it suppresses the context that is needed to understand the algorithm. One should always specify what is true of the input, what we desire to have true of the output, and, ideally, the invariants that remain true during an algorithm. Time spent doing so helps avoid errors, decrease debugging time, and simplify maintenance.

For more critical or intricate programming tasks, you also need to guarantee that the algorithm that you write terminates and returns the correct output. Formal proof can help do that. As usual, the search for a proof can be more valuable than the proof itself; trying to prove a program correct help clarify the conditions you need on the input and what you need to maintain as an invariant, increasing your understanding of a problem and the chances that your code is correct. In this chapter you will work through several examples, starting with the simple problem of finding a maximum in a list.

Remember the problem solving steps from chapter 1: Draw diagrams and work through examples on scratch paper to help you understand what is sought. Break tasks into smaller tasks. Ask yourself what can go wrong—check the types, check the ranges of indices. Then, when writing down your solution, don't forget the context: before writing the steps of any algorithm, always specify the input and desired output. Then answer three questions, in this order:

Tragically, the computer scientist died in the shower after reading a shampoo bottle's instructions: Lather. Rinse. Repeat.

- Does this algorithm terminate?
- Does it produce correct results?
- How many steps does it take?

11.2 Max in a list(A)

Our first algorithm is to find the maximum in a list. Even this simple algorithm leads to interesting questions when we also search for min or look at average performance over all possible inputs. But let's begin by considering *invariants*, which are statements that are true at all times during a procedure.

1. I read you a sequence of n bid amounts, $b_1, b_2, \ldots, b_n$, and you want to sell to the highest bidder. What will you keep track of?

Remembering the entire sequence of bid amounts would be overkill, since tracking one amount and two integers is enough. The properties of these three numbers are the *invariant* of a procedure for tracking the highest bid and bidder. Notation helps us state an invariant precisely, so let us leave this scratch-paper musing and begin writing an algorithm for finding the maximum in a list.

We are given an array $A[0..n-1]$ of n entries that can be compared by $<$. Think of each entry $A[j]$ as a bid amount in your favorite currency to distinguish it from the integer *position* or *index* $j \in [0, n)$ at which it is stored. We want to find the* maximum bid amount mx, which satisfies $\forall_{j\in[0,n)}\ mx \geq A[j]$, and a position mi where it is found, so $mx = A[mi]$.

*Note articles: since mx be repeated in A, I say "*the* max bid" but "*a* position."

The implementation in Algorithm 1 is iterative. There will be a *loop variable* i that starts at 1, and will be incremented after each time through the loop. At the start of each loop, we maintain an invariant that says that mx is the maximum of the first i array entries, $A[0..i-1]$. Stated in notation, the *invariant* has three parts: $0 \leq mi < i$, $mx = A[mi]$ and $\forall_{j\in[0,i)}\ mx \geq A[j]$. The last time through the loop body $i = n-1$, so when it is incremented to $i = n$, the check at the top of the loop exits.

2. Work the example suggested in the margin, making a table to track the values of mi, mx and i at the test $i < n$ in line 2. I put i at the end of the table, since it is incremented just before the test.

For $A = [77_0, 55_1, 77_2, 99_3, 88_4, 66_5]$, track values of mi, mx and i at the test $i < n$ in line 2.

It may seem more natural to track values at the 'endfor' after the loop body, and you may choose to do so, if you prefer. If you do, however, notice that your base case must handle two possible ways to reach the 'endfor' for the first time: either the test initially fails, and you don't pass through the loop body, or it initially succeeds and you do.† By choosing to track the values at the test in line 2, my base case handles only the initialization, and my inductive step handles any and all executions of the loop body. When the test fails because $i = n$, I don't have to distinguish whether it failed in the base case or the induction step.

†This is a subtlety of making the proof cover all possible cases; the code is unchanged.

To show that Algorithm 1 is correct, we must show that, for all input values n and $A[0..n-1]$, it terminates and returns the correct value. For any given n and A, proving that A terminates is trivial: when line 2 finds the loop variable has reached n, the program halts.

For any given n and A, to prove that Algorithm 1 finds the correct value, we actually prove something stronger—that each trip through the 'for' loop preserves the three-part invariant. Specifically, for any given n and A, we prove that for all $1 \leq i \leq n$, whenever line 2 tests that $i < n$,‡ the three-part

‡Note that last test is with $i = n$, since when $i < n$ is false, the loop ends.

Max in a list(A)
Input: An array $A[0..n-1]$ of n comparable entries.
Output: Find a position mi of the maximum entry mx in A:
$0 \le mi < n$, $mx = A[mi]$ and $\forall_{j\in[0,n)}\ mx \ge A[j]$.
Invariant: In line 2, just before testing loop variable $i < n$, we know that $0 \le mi < i$, $mx = A[mi]$ and $\forall_{j\in[0,i)}\ mx \ge A[j]$.

```
1: mx = A[0]; mi = 0;
2:     for (i = 1; i < n; i++)
3:         if mx < A[i]
4:             mx = A[i]; mi = i;
           endif
       endfor
```

Algorithm 1: Find a maximum entry in a unordered list in array $A[0..n-1]$.

invariant holds: $0 \le mi < i$, $mx = A[mi]$ and $\forall_{0\le j<i}\ mx \ge A[j]$.

We prove this by induction on the value of the loop variable, i. (We could instead do induction on the number of trips through the loop body, which is $i-1$.) There will be some confusion here because we will be speaking of the loop variable "i", its current value i, and its previous value $i-1$ (which was named "i" at the time, but which I'll call k to match my induction hypothesis).

Base $i = 1$ is the initial case, when we have been through the loop 0 times. Line 1 has set $mi = 0$ and $mx = A[0]$, so the invariant holds at the test of line 2: $0 \le mi < 1$, $mx = A[mi]$, and $\forall_{0\le j<1}\ mx \ge A[j]$. ✓

Ind Step: At the test in line 2 with a given $1 < i \le n$,
IH: I may assume that, when line 2 tested previous loop variable values $k \in [1, i)$, the invariant held: $0 \le mi < k$, $mx = A[mi]$ and $\forall_{0\le j<k}\ mx \ge A[j]$. (I'll use only the $k = i-1$ cases of the IH.)

Now we want to show that the invariant holds for the given value $i > 1$.

To have reached the test in line 2 with $i > 1$, we must have gone through the previous loop with value $k = i-1$.

3. Quick check: did $k = i-1$ make the trip through the body of the previous loop?

The IH holds for $k = i-1$ so at the test in the previous loop, we knew that $0 \le mi < k$, $mx = A[mi]$ and $\forall_{0\le j<k}\ mx \ge A[j]$. Next, line 3 tested if $mx < A[k]$. If not, then mx was already a maximum in $A[0..k]$; otherwise line 4 updated mx and mi with the new maximum $A[k]$ and position $mi = k$.

In either case, since $k = i-1$, we now know that $0 \le mi < i$, $mx = A[mi]$ and $\forall_{0\le j<i}\ mx \ge A[j]$. Thus, the invariant has been re-established for the

current value of i.

QED by induction.

4. Notice that the loop ends with $i = n$. Spell out what the invariant says for this final test, and why that implies that mi and mx have the correct final values.

5. Strengthen the invariant to say that the index mi is the *first* occurrence of the maximum mx in $A[0..i-1]$. Write this formally using a quantified logical expression; attempt this yourself before you peek to see my answer.

6. What steps of the induction proof would you modify to show that the algorithm already establishes this strengthened invariant?

7. Convert the invariant and algorithm to index the array of n numbers starting from 1 instead of 0.*

*Example languages whose default arrays start at index 1: FORTRAN, COBOL, MATLAB, Julia, and Lua.

This question is easy enough that one might "just start coding" to make the changes. To minimize debugging, however, first change the input, output, and invariant, then derive the algorithm changes from the changes to the invariant.

What if we want to compute both the minimum and the maximum? Let's continue to use the array $A[1..n]$ here, and later consider the changes if we return to $A[0..n-1]$.

MinMax(A)
Input: An array $A[1..n]$ of n comparable entries.
Output: return the maximum entry mx in A and minimum entry mn in A.

Assume initially that you are allowed to reorder elements of A during the algorithm; later we remove that assumption. By simplifying the problem we can find an initial solution faster, and then improve it by *refactoring*, which is rewriting and improving working code. It can be more fun to build from success to success than to be debugging failures until reaching success at the end.

Algorithm 2: Finding a maximum entry in array $A[1..n]$.

8. I'd like to find both the min and max in array $A[1..n]$ with the smallest number of comparisons to elements in the array. Obviously, we could run $mx = Max(A)$ from section 11.2, and a similar algorithm $mn = Min(A)$. How many comparisons does that perform?

9. For all integers $1 \leq i \leq n/2$, if $A[2i-1] > A[2i]$ then swap the entries $A[2i-1]$ and $A[2i]$. Do this on the example arrays $A = [77_1, 55_2, 77_3, 99_4, 88_5, 66_6]$ and $B = [77_1, 55_2, 77_3, 99_4, 88_5, 66_6, 99_7]$. Exactly how many comparisons of array elements is this? (Give an expression that works for both even and odd n.)

10. Double check: prove that comparing $A[2i-1] > A[2i]$, for integers $1 \leq i \leq n/2$, does not access elements outside of $A[1..n]$.*

*Several internet-wide virus infections have exploited *buffer overrun* errors.

11. What can you say in English about the array that results from the procedure in 9? In notation we can say that $\forall_{1 \leq i \leq n/2}\ A[2i-1] \leq A[2i]$.

12. It is important to be precise. Each of these five plausible statements **is**

false for some input array; do you see why?

- After the procedure in 9, A is sorted. ?
- After the procedure, $\forall_{j\in[1,n)}\ A[j] \le A[j+1]$.?
- After this procedure entries at odd positions cannot be larger than entries at even positions.?
- After this procedure every entry at an odd position is at most the entry at the next even position.?
- The maximum value must be at an even position and the minimum at an odd position.?

After the procedure of question 9, only the values in odd positions are candidates for the minimum, and only values in even positions (plus $A[n]$ if n is odd) are candidates for the maximum. Thus we can run the previous max algorithm, and a related min algorithm, each on roughly half the array. Let's develop the invariant and algorithm together.

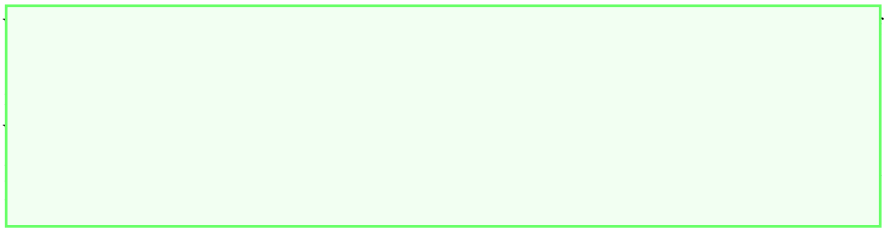
13. Let's initialize at the high end of the array, since that is where even and odd behave differently. How might you initialize in both odd and even cases to minimize comparisons?

14. Next, I'd loop for integers $1 \le i < n/2$ with the **Invariant** that before each loop

$$mn = \min(A[n-1+(n \bmod 2)..n], A[1..2(i-1)])$$
$$mx = \max(A[n-1+(n \bmod 2)..n], A[1..2(i-1)]).$$

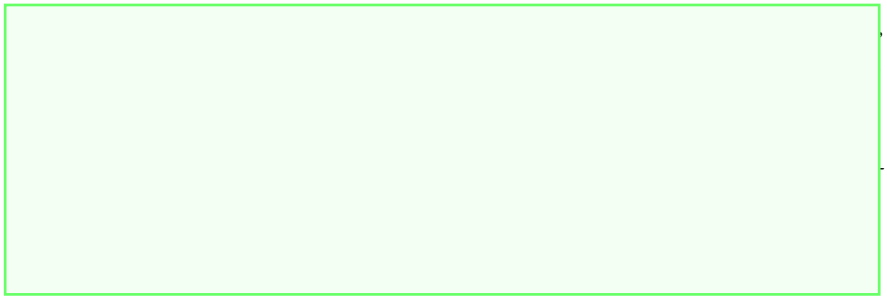
Write the loop body. The first box hides a hint.

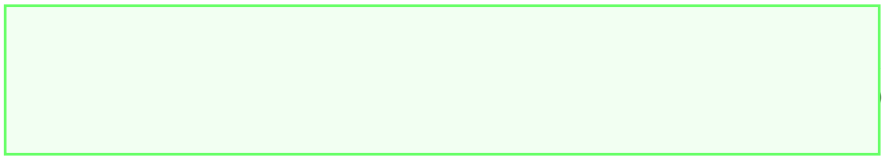
15. For both even and odd cases, double check the value of i when the loop terminates. Make sure that it fits with the initialization so no array entries are missed.

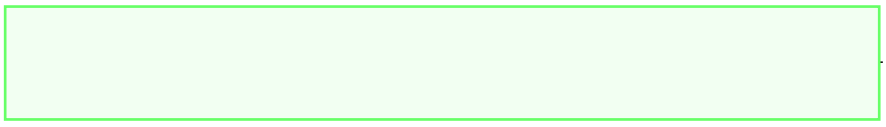


16. Prove by induction that your algorithm correctly finds the min and max in any given array, $A[1..n]$. That is, establish the invariant in 14 for all iterations through the loop.

We don't need to do induction on n or the array elements, because those are fixed while this algorithm runs. We assume we've already swapped pairs, so we know $\forall_{1 \le j \le n/2}\ A[2j-1] \le A[2j]$.







17. What is the total number of comparisons to find both min and max? (Don't forget the comparisons in 9. You can do evens and odds separately, if you wish.) Your calculation may differ, but you should get an equivalent final answer:

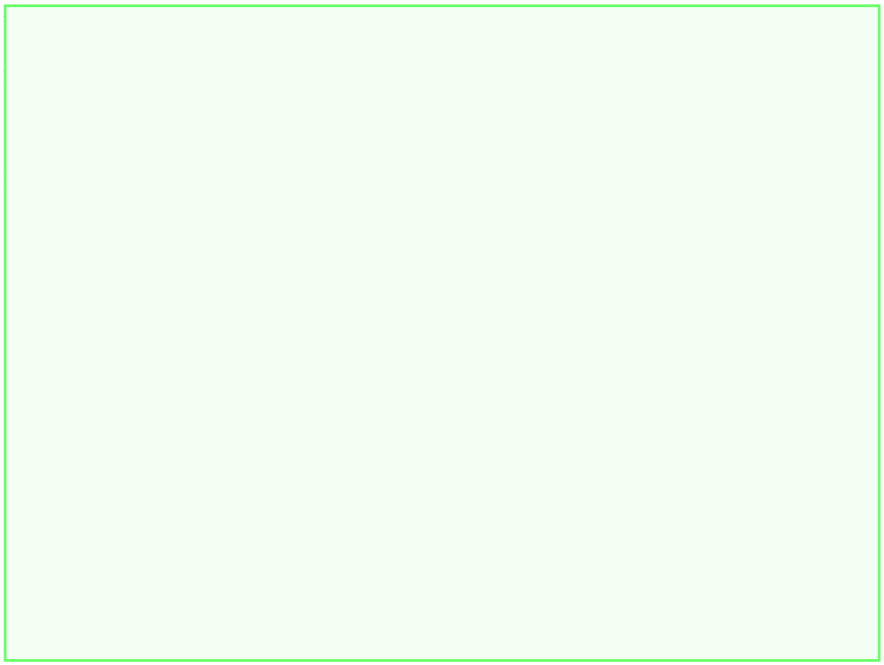
18. Change your algorithm to work when the array entries are stored in $A[0..n-1]$ and may not be moved.

Refactor your code in stages. First, make it internally do the comparison and swap of a pair at a time, rather than doing the swaps in preprocessing. Then change the indexing. As before, change the invariants first, and derive the code changes from the invariant.

Note: I decided to keep the same for loop limits, but change the index calculations; your code is likely to be different. If you write tricky code like this, then include the invariants in the comments. You'll thank yourself when if you read it 2 months later, as will anyone who has to maintain the code.

We can prove that any comparison-based algorithm that correctly finds min and max on all input arrays of size n must perform at least $f(n) = \lceil \frac{3n}{2} \rceil - 2$ comparisons on at least one of the possible input arrays. We use a technique called an *adversary argument*.

Usually we play the role of algorithm creator and want to show that, for all large n, there exists (by construction) an algorithm that, for all input arrays of length n given by the cruel world, computes the correct min+max, and performs at most $f(n)$ of comparisons. To negate this, we want to show that there is a large n so that, for any algorithm, there is an array that needs more than $f(n)$ comparisons. Our opponent is now the algorithm creator, who still can try all algorithms, and we are the adversary who picks the array



Algorithm 3: Finding both min and max in $A[0..n-1]$ without disturbing A.

that frustrates each attempt. (Note that we are still playing the existence quantifier.)

Actually, we'll prove something a little stronger than the negation, because we let the algorithm designer pick n as well as the algorithm. Through the next few questions we will show that, for any $n \in \mathbb{Z}^+$ and algorithm $MM(A)$ that finds min+max in arrays of size n by comparisons, we can create an array $A[1..n]$ such that $MM(A)$ must perform $f(n) = \lceil \frac{3n}{2} \rceil - 2$ comparisons to correctly find the min and max for A.

We, as the adversary, want to show that there exists an array that forces our opponent's algorithm to perform many comparisons. To keep track of the comparisons, let us assume that we keep the array—the algorithm never sees specific values, but may ask us how two entries compare, or instruct us to move entries around. Note that all algorithms of this section can be made to work this way: e.g., by storing *mi*, the index of the current max, they can compare with $A[mi]$ instead of a stored value *mx*.

A key observation is that, since we hold the array, we can delay assigning actual values to array entries, as long as we answer requests for comparisons between entries in a manner that is consistent with our previous answers. So, create four labels for array entries—*untested*, *small*, *mid*, and *big*—with the following definitions for how each entry is labeled at any time.

untested: An entry is labeled untested iff it has not been compared with another entry. Untested entries may be assigned any value without contradiction.

mid: An entry is labeled mid when we assign it a unique, permanent integer value. There will remain values labeled small and labeled big when we do this.

small: An entry labeled small has been compared only with entries in mid and big, so may be assigned any value smaller than min(mid).

big: An entry labeled big has been compared only with entries in mid and small, so may be assigned any value larger than max(mid).

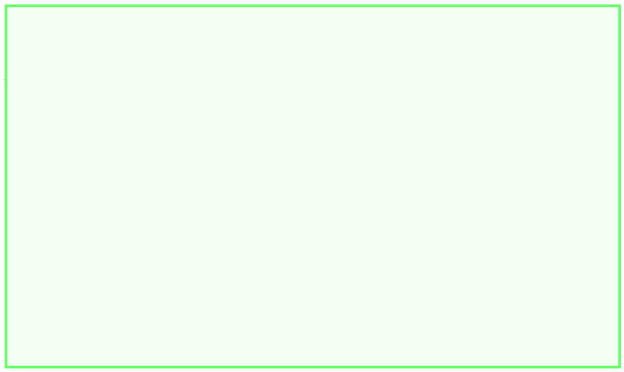
19. How must we initially label each array entry to satisfy all invariants?

20. Candidates for min and max array entries may have which labels? The goal of the algorithm is to reduce to one min and one max candidate. Initially, how many candidates are there in total, counting both min candidates and max candidates?

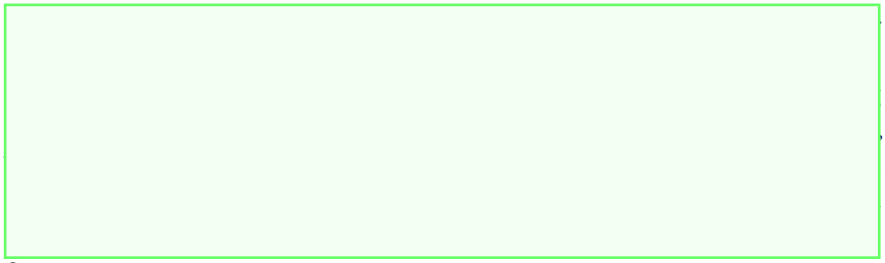
We, as adversary, must respond to the algorithm's request for comparisons between two entries. We can look at the labels and decide what to report ($<$ or $>$) and how we must relabel as a result. For example, comparing two untested entires, we report $<$ and re-label the one small, and the other big. Before the comparison both were min and max candidates (total of four); after the small is a min candidate only and the big is a max candidate only (total of two). Comparing small to small, we report $<$, assign the second the value min(mid) $-$ 1 (or 0 if there is no entry labeled mid yet), and relabel it mid, reducing by one candidate.

When there is a choice, we base our decision on what makes the smallest change in the number of candidates for min and max. Comparing small to untested, we report $<$ and change the untested to big, because that loses only one candidate. Comparing a small to a big, we report $<$ and leave labels and candidates unchanged. With mid to mid, we use the numbers to decide what to report.

21. Fill in this table for how to resolve comparisons between labeled entries (U,M,S,B) and minimize the number of candidates lost, Δ. I use $*$ to indicate that an output label is unchanged. (If you are unsure of the table structure, drag the rectangle down and to the right to see the row and column headings before you reveal the rest of the table.)



22. Now, argue that in order to have one min and one max candidate remain, the algorithm must perform at least $\lceil 3n/2 \rceil - 2$ comparisons. It may be easiest to consider even n and odd n in separate cases.



Concepts:

- Algorithm properties: termination, correctness, step counts.
- Invariant: something that remains true during algorithm execution.
- Before listing steps of an algorithm, state input, output, and invariants.
- Often the algorithm can be derived from the invariants (7, 13–14).
- Even, odd, floor, ceiling
- Adversary argument for lower bound (19–22)

11.3 Iterative binary search

"I'm thinking of a number between 1 and 1000." I'm sure that you have played this game in which you guess and are told if you are too high or too low. If you always guess the middle of the remaining numbers, what is the maximum number of guesses you'll need? This is binary search, and every computer science student should learn how to derive a correct, non-recursive implementation of binary search from its invariants. It is a good question for job interviews because it tests not only preparation but also thinking and communication skills.

I would like code for the following.

Binary search(A, x)

Input: A sorted array $A[0..n-1]$ of n numbers with $A[0] \le A[1] \le \cdots \le$

$A[n-1]$ and a number x.

Pretend that $A[-1] = -\infty$ and $A[n] = \infty$ (without actually modifying A).

Output: Return the unique index $0 \le i < n$ with $A[i] \le x < A[i+1]$.

Invariants: Maintain two indices $0 \le lo < hi \le n$ for which $A[lo] \le x < A[hi]$.

We access $A[i]$ only for $lo < i < hi$.

Warning note: I have left two mistakes in the Output and Invariants that you should be able to detect as soon as you start these exercises.

23. Read the desired output, and decide what these calls should return:

$Bsrch([5,10,15],10)$, $Bsrch([5,10,15],12)$, $Bsrch([5,10,15],1)$,
$Bsrch([5,10,15],42)$, $Bsrch([5,10],5)$, $Bsrch([\,],42)$

A key idea for $Bsrch(A,x)$ is that we maintain a shrinking *interval*, $[lo,hi)$, rather than just a middle point. Many students attempt to maintain the middle point (and maybe the width) of the interval, but that is hard to do correctly. Other students write recursive algorithms that copy half of the array, thereby doing exponentially more work than they should.

24. With what values should we initialize lo and hi?

25. Let's use a while loop. What is the condition on lo and hi for which we can stop?

26. Suggest a calculation to choose an integer mid midway between lo and hi.

27. How would you decide if x is in interval $[A[lo],A[mid])$ or $[A[mid],A[hi])$, and why must it be in exactly one of these two half-open intervals?

28. Write the binary search algorithm so that the invariant is preserved.

By deriving the steps from the invariant we obtain a very short algorithm that does exactly what we need. It makes no access out of the range $[0..n-1]$, yet it correctly handles lists with 0 elements and returns the flag index -1 if x is below the range.

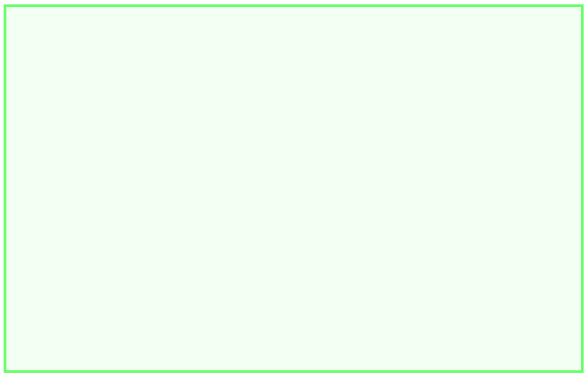
29. You may have expected $Bsrch(A,x)$ to return *false* if x is not in the sorted array A. What I have chosen to return is more useful, because it tells the

Binary search(A, x)

Input: A sorted array $A[0..n-1]$ of n numbers with $A[0] \le A[1] \le \cdots \le A[n-1]$ and a number x. Pretend that $A[-1] = -\infty$ and $A[n] = \infty$ (without actually modifying A).

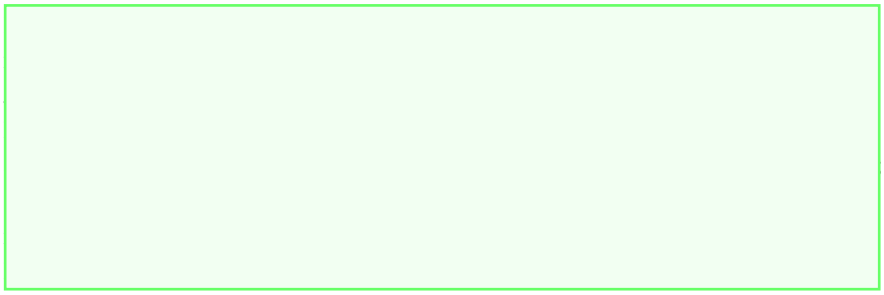
Output: Return the unique index $-1 \le i < n$ with $A[i] \le x < A[i+1]$.

Invariants: Maintain two indices $-1 \le lo < hi \le n$ for which $A[lo] \le x < A[hi]$. We access $A[i]$ only for $lo < i < hi$.

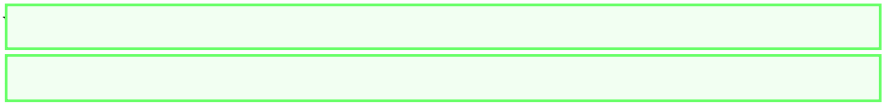


Algorithm 4: Binary search derived from its invariants

position of the last copy of x if there are duplicates, and tells you the position at which you want to insert x if you want to add it. What tests will correctly determine whether x is in the array after $idx = Bsrch(A, x)$?



30. How many comparisons does Algorithm 4 do in the worst case? In a list with n elements, there are $m = n + 1$ possible outcomes. We can recursively define a function $T(m)$ that counts the number of comparisons to determine the unique outcome. We use m to avoid carrying a "+1" around while doing the induction. $T(1) = 0$, and for all $m > 1$, $T(m) = 1 + T(\lceil m/2 \rceil)$. Show that $T(m) = \lceil \log_2 m \rceil$.



Concepts:

- Binary search
- Deriving an algorithm from its invariants (24–28)
- Recurrences (recursive functions) for counting steps of an algorithm (30)

11.4 Sorting by insertion

There are many good sorting algorithms for various types of data. (And some bad ones that teachers like to teach anyway; the worst culprit is Bubblesort—never use it.) Insertion sort in Algorithm 5 is great for small or nearly-sorted lists.

31. Prove that Algorithm 5 terminates.

32. Prove that no numbers are lost, even though we may assign $A[j+1] = A[j]$ and/or $A[j+1] = tmp$.

33. Prove by induction that Algorithm 5 correctly sorts its input array $A[0..n-1]$.

34. What are the minimum and maximum numbers of times that this algorithm can assign the value of j? What permutations of $[1..n]$ give the

Insertion sort(A)
Input: An array $A[0..n-1]$ of n numbers
Output: Values of $A[0..n-1]$ permuted so $A[0] \leq A[1] \leq \cdots \leq A[n-1]$.
Invariant 1: At 1 with loop value i, the values of $A[0..i-1]$ have been permuted so that $A[0] \leq A[1] \leq \cdots \leq A[i-1]$, and the values of $A[i..n-1]$ are unchanged.
Invariant 2: At 2, tmp is the former value of $A[i]$, and $tmp <$ former values of $A[j+1..i-1]$, now moved to $A[j+2..i]$.

```
1: for (i = 1; i < n; i++)
       tmp = A[i]; j = i − 1;
2:         while j >= 0 & tmp < A[j]
               A[j + 1] = A[j]; j = j − 1;
           endwhile
           A[j + 1] = tmp;
       endfor
```

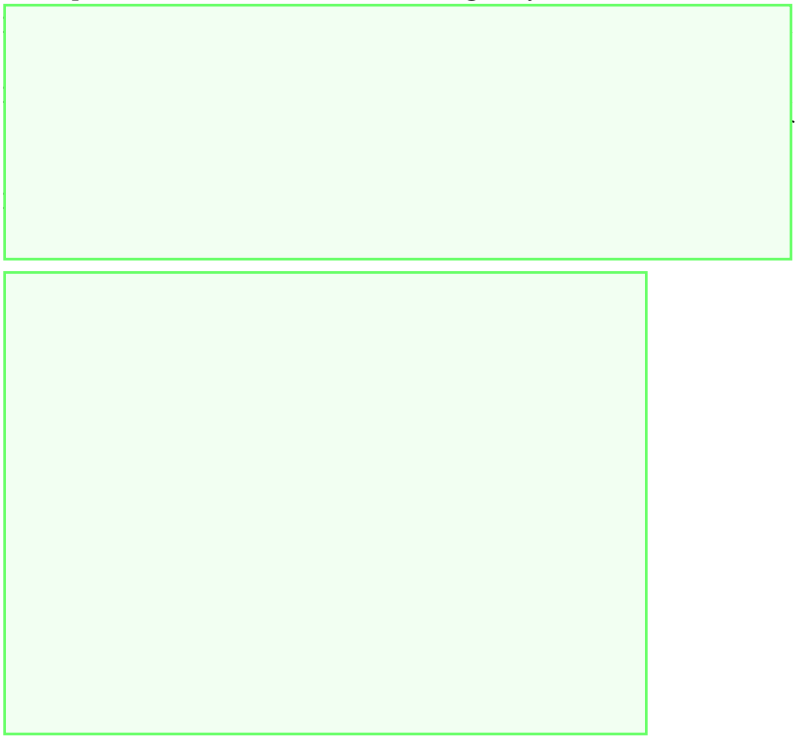
Algorithm 5: Insertion sort $A[0..n-1]$.

minimum and the maximum?

35. Modify Algorithm 5 to create a permutation P of $[1..n]$ so the value found at P[input index] = output index.

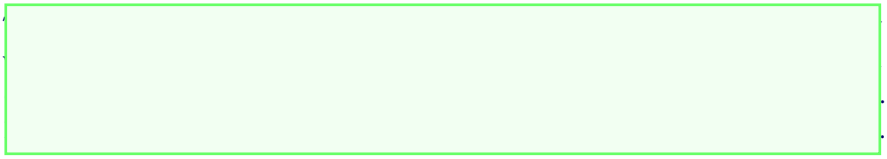
We want to keep track of where the item originally at position i ultimately goes to. This requires thought. We can keep track of where each item came from and invert to get where each goes to. Add to both invariants that i is the original position of tmp, and that $F[k]$ always tells the original position of the item at k. The comments in Algorithm 6 contain the statements that would be used in an induction to formally prove that F and P are set up correctly.

Insertion sort(A)
Input: An array $A[0..n-1]$ of n numbers
Output: Values of $A[0..n-1]$ permuted so $A[0] \leq A[1] \leq \cdots \leq A[n-1]$ and permutation P so that the value originally at $A[i]$ ends at $A[P[i]]$.



Algorithm 6: Modified insertion sort of $A[0..n-1]$ records where elements come from and reports permutation P so the item originally at i ends at $P[i]$.

36. Strengthen the invariant to show that insertion sort is *stable*, that is, it does not swap the order of numbers that are equal. (If in the input $A[i] = A[j]$ for $i < j$, then output indices $P[i] < P[j]$.)



The sharp-eyed may notice that elements initially assigned to F are never used in the code, only in invariant 0. So we could drop that initialization if we wanted to work invariant 0 into the other two. The next modification, however, does make use of the initialized values.

37. If we permute just the from indices, we can avoid moving the data in A, a

trick worth knowing for large data records, or when you want to sort shared data. Explain how Algorithm 7 does this.

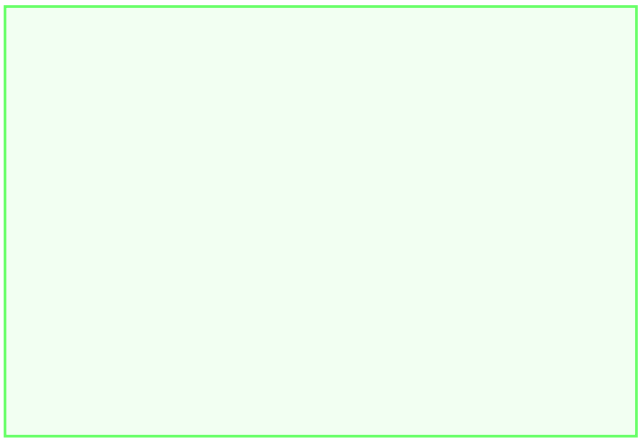
Insertion sort(A)

Input: An array $A[0..n-1]$ of n numbers

Output: Values of $A[0..n-1]$ permuted so $A[0] \leq A[1] \leq \cdots \leq A[n-1]$ and permutation P so that the value originally at $A[i]$ ends at $A[P[i]]$.

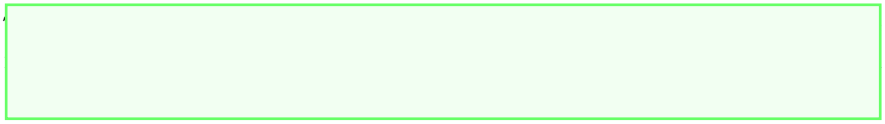
Inv 0: $F[0..n-1]$ maintains all $F[k]$ equal to the position in A of the item that is kth in the current insertion sort array order.

Inv 1: At the start of line 1 with loop value i, the values of $F[0..i-1]$ have been permuted so that $A[F[0]] \leq A[F[1]] \leq \cdots \leq A[F[i-1]]$, and the values of $F[i..n-1]$ are unchanged.

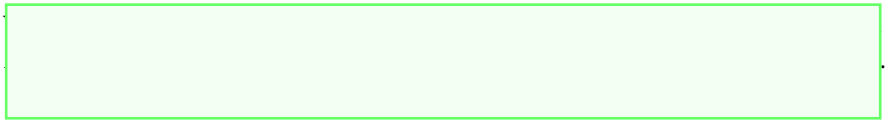


Algorithm 7: Insertion sort $A[0..n-1]$ without moving data

38. Define an inversion for A to be a pair (i,j) with $i < j$ but $A[i] > A[j]$. What are the minimum and maximum numbers of inversions possible, and what permutations give these numbers?



39. Show that the number of times j is assigned a value is $n-1$ plus the number of inversions of A.



40. Of the $n!$ permutations of $[1..n]$, how many have an inversion with a chosen pair (i,j) for $i < j$?

41. By summing the previous result, what is the average number of inversions over all $n!$ permutations? (This is the average number of steps for the algorithm, if we assume that all input permutations are equally likely.)

Concepts:

- Sorting a list
- Finding a permutation (35)
- Stable sorting (36)
- Inversions of a list (38–40)
- Average running time, assuming a distribution of the input (41)

11.5 Greatest common divisor

Euclid's Elements, Book VII, Prop. 2 presents an ancient algorithm for greatest common divisors in the verbose, but still understandable fashion of 300 BC. Euclid's algorithm is, in some sense, the reverse of the mediant algorithm of Lemma 7.3.1, but this presentation will be self-contained. I quote a translation of some of Euclid's preliminary definitions and his lemma in figure 11.1 to underscore the benefits of modern notation for precision and clarity.

Recall that for integers a, d we say that d *divides* a, written $d|a$, iff there exists an integer m so $a = md$.

$\forall_{a,b\in\mathbb{N}}$, define

$$gcd(a,b) = \begin{cases} 0 & \text{if } a = 0 \text{ and } b = 0, \\ \max\{d : d|a \wedge d|b\} & \text{otherwise.} \end{cases}$$

42. Examples: What are $\boxed{gcd(0,0)}$, $\boxed{gcd(18,24)}$, $\boxed{gcd(3,5)}$, and $\boxed{gcd(0,4)}$?

Algorithm 8 presents slow and fast versions of Euclid's algorithm, *SE* and *FE*, with the same input and output:

1. A **unit** is that by virtue of which each of the things that exist is called one.
2. A **number** is a multitude composed of units.
3. A number is **a part** of a number, the less of the greater, when it measures the greater;
4. but **parts** when it does not measure it.
5. The greater number is a **multiple** of the less when it is measured by the less. . . .
11. A **prime number** is that which is measured by an unit alone.
12. Numbers **prime to one another** are those which are measured by an unit alone as a common measure.

Proposition 1. Two unequal numbers being set out, and the less being continually subtracted from the greater, if the number which is left never measures the one before it until an unit is left, the original numbers will be prime to one another.

Figure 11.1: Euclid's definitions and preliminary lemma before the algorithm for greatest common divisor that bears his name. Translation by Heath [11].

Input: integers $a, b \geq 0$.

Output: the greatest common divisor $gcd(a, b)$

Both $SE(a, b)$ and $FE(a, b)$ in Algorithm 8 compute $gcd(a, b)$ for all $a, b \in \mathbb{N}$.

43. Show that if $SE(a, b)$ is called with parameters $a, b \geq 0$, then all recursive call parameters are ≥ 0. This will be important in the next question to show that the SlowEuclid algorithm is correct.

44. Show that for all integers $a, b \geq 0$, algorithm $SE(a, b)$ terminates.

The condition $a, b \geq 0$ is important, since, for example, $SE(0, -1)$ does not terminate. (For defensive programming without adding an extra test to each call, line 2 should test $b <= 0$, and then raise an exception if $b < 0$.)

To prove termination we would like to show that the pair (a, b) becomes smaller in some way. Here are three options; most students try an informal version of the second, but the others have fewer cases.

Option 1: Define a pair order: $(c, d) > (e, f)$ iff $\max(c, d) > \max(e, f)$ OR $(\max(c, d) = \max(e, f)) \wedge (d > f)$. This is known as *lexicographic* or *dictionary* order: resolving the comparison at the first index where the tuples differ.

Now, observe that a recursive call is always made to a smaller pair:

- When $a < b$, we recursively call $SE(b, a)$. This is the easy case: $(a, b) > (b, a)$, since [ ] and [ ].
- When $a \geq b > 0$, we call $SE(b, a - b)$. Show that $(a, b) > (b, b - a)$:

SE(a, b) /* *Slow Euclid* */

Input: Integers $a, b \geq 0$.

Output: The greatest common divisor $gcd(a, b)$.

```
if b > a
    return SE(b, a);
else if b == 0
    return a;
else
    return SE(b, a − b);
endif
```

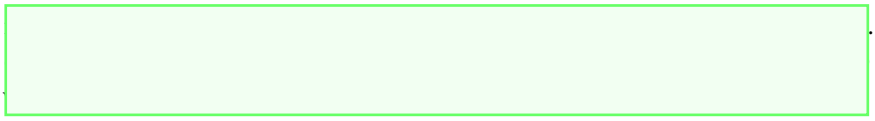
FE(a, b) /* *Fast Euclid* */

Input: Integers $a, b \geq 0$.

Output: The greatest common divisor $gcd(a, b)$.

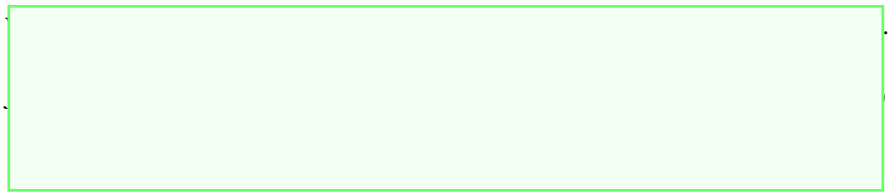
```
if b == 0
    return a;
else
    return FE(b, a mod b);
endif
```

Algorithm 8: Two algorithms for greatest common divisor: *SE*(a, b) uses repeated subtraction as described by Euclid, and *FE*(a, b) uses mod.

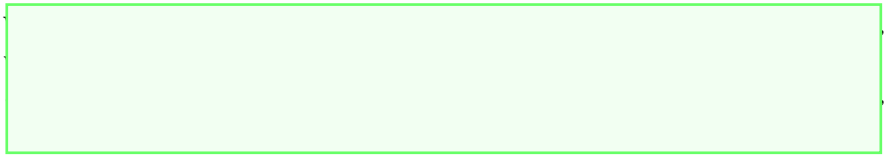


Notice that the only pairs smaller than $(0, 0)$ have negative numbers, which question 43 implies we will never reach. Thus *SE*(a, b) must terminate.

Option 2: define a function $f(a, b) = \max(a, b)$.



Option 3: define $f(a, b) = 2\max(a, b) + [a < b]$. Here we are using Iverson notation, where $[a < b]$ evaluates to 1 if $a < b$ and 0 otherwise.



45. Show that for all $a > b \geq 0$, $(d|a \wedge d|b)$ iff $(d|b \wedge d|(a-b))$. This, with mathematical induction, is all we need to show that $SE(a, b)$ returns the correct result, $gcd(a, b)$.

46. $SE(999999, 1)$ would be slow; show that $FE(a, b)$ computes the same result as $SE(a, b)$ by showing that many steps of $SE(a, b)$ may be replaced by one step of $FE(a, b)$.

The intuition is that mod or division is repeated subtraction. The proof will check not only this intuition, but also that the algorithm handles the start and end conditions correctly. We can show by induction that several steps of $SE(a, b)$ can simulate one step of $FE(a, b)$, leaving no doubt that they compute the same result.

47. Recall the Fibonacci sequence: $F_0 = 0$, $F_1 = 1$, $F_{n+1} = F_n + F_{n-1}$, for n>0.

Run FastEuclid on adjacent Fibonacci numbers, such as $FE(21, 13)$. Make a table of the values of each call. What familiar numbers do you observe?

48. Show that for all integers a, b with $0 \le b \le F_i$ the call $FE(a, b)$ makes at most i recursive calls. (That is, every call after the first: $FE(a, 0)$ makes 0 recursive calls.) This is challenging because it takes some thought in 'step S7' of the 8-step induction template.

49. These algorithms are *tail-recursive* because in each branch the last thing they do is make the recursive call. Tail-recursive algorithms are easy to convert to iterative algorithms. I leave you with the task of rewriting both algorithms to use 'while' statements instead of recursion.

Concepts:

- Divides and gcd
- Induction on two parameters (44)

- Using a slow algorithm to explain a faster one (46)
- Recurrences to count steps
- Fibonacci numbers (47–48)
- Tail recursion (49)

Chapter 12
Binary Relations and Applications

No one really understood music unless he was a scientist, her father had declared, and not just a scientist, either, oh, no, only the real ones, the theoreticians, whose language was mathematics. She had not understood mathematics until he had explained to her that it was the symbolic language of relationships. "And relationships," he had told her, "contain the essential meaning of life."

—Pearl S. Buck, *The Goddess Abides*

The definition of relation and definitions of types of binary relations from chapter 6 were abstract. Now that we have the tools of recursive definition and inductive proof, I want to demonstrate that this abstraction helps us understand the nature of computation, using two quite complex examples. The first defines my favorite *partial order*, aboveness, which reveals one reason why geometry in 3d is harder than in 2d. The second uses several *equivalence relation*s to create, for a given regular language, the simplest possible Kara program to recognize whether or not a string is in the language. This chapter tells the stories of these two examples, after recalling and extending some definitions from chapter 6.

Objectives: After working through this chapter and the exercises, you will review which binary relations are reflexive, symmetric, and transitive, and be able to force these properties by closure operations. You will be able to recognize and give examples of comparability, partial order, and equivalence relations. You will be able to prove that an *equivalence relation* partitions its domain into *equivalence classes*. Through complex examples of aboveness for line segment drawings and determining the simplest Kara program to recognize the strings of a *regular language*, I hope that you will begin to appreciate how proofs using the somewhat abstract properties of partial orders and equivalence relations can reveal useful structure that would otherwise be hidden.

12.1 Binary relations extended

Recall from section 6.1 that a relation is just a set of pairs, $R \subseteq A \times B$, and a binary relation is pairs from the same set, $R \subseteq A^2$. Here are nine special types for a binary relation, $R \subseteq A \times A$, that are defined by quantified statements. The first five are review.

Reflexive: A relation R is *reflexive* iff $\forall_{x \in A}\ (x\ R\ x)$.
Symmetric: R is *symmetric* iff $\forall_{x,y \in A}\ (x\ R\ y \to y\ R\ x)$.

Transitive: R is *transitive* iff $\forall_{x,y,z \in A}\ ((x\ R\ y \wedge y\ R\ z) \rightarrow x\ R\ z)$.
Irreflexive: R is *irreflexive* iff $\forall_{x \in A}\ (x \not R\ x)$.
Antisym: R is *antisymmetric* iff $\forall_{x \neq y \in A}\ (x\ R\ y \rightarrow y \not R\ x)$.
Comparability: R is a *comparability* relation iff $\forall_{x \neq y \in A}\ (x\ R\ y \vee y\ R\ x)$.
Equiv. relation: R is *equivalence relation* iff it is reflexive, symmetric, and transitive.
Partial order: R is a *partial order* iff it is antisymmetric, transitive and either reflexive or irreflexive. The pair of set and relation (A, R) is called a *poset*.
Total order: R is a *total order* iff it is a partial order and a comparability relation.

Example binary relations in section 6.1 included relations on integers like "equals" ($=$), "less than" ($<$) and their negations "not equals" ($\neq$) or "greater than or equal to" ($\geq$), the relation on sets of subset $\subseteq$, and relations on people like "is a child of," "is in the same class as," or "has sent email to." Which of these are comparability[?], partial order[?], total order[?], or equivalence relations[?]?

12.1.1 Closure operations for relations

For a binary relation $R \subseteq A^2$, three *closure* operations can make new relations. These operations are used only occasionally, and almost exclusively when a relation is being defined, so there isn't a standard notation for them.

- The *reflexive closure*, $R \cup \{(a, a) \mid a \in A\}$, adds each element paired with itself. This new relation is clearly reflexive.

- The *symmetric closure*, $R \cup R^{-1} = \{(a, b), (b, a) \mid (a, b) \in R\}$, adds the reverse of every pair. This new relation is clearly symmetric.

- The *transitive closure* is a relation $\hat{R} \subseteq A^2$ that can be defined recursively:

  Base: $R \subseteq \hat{R}$.
  Rec. Rule: $\forall_{a,b,c \in A}$, if $a\hat{R}b$ and $b\hat{R}c$ then $a\hat{R}c$.
  Closure: The only pairs in $\hat{R}$ are those obtained from the base case by a finite number of applications of the recursive rule.

  This definition adds the shortcut for every two pairs until no new shortcuts can be added; since it doesn't stop until all shortcuts are present, $\hat{R}$ is clearly transitive.

As an example, define the "ancestor of" relation as the transitive closure of the "parent of" relation: the ancestors of a person are their parents and the ancestors of their parents. Assuming no unusual loops (e.g., that no child has adopted their grandparent, and no-one is their own parent), the

"ancestor of" relation is an irreflexive partial order. We could make it a reflexive partial order, calling each person their own 0-step ancestor, by also taking the reflexive closure.

A reflexive or symmetric closure adds all missing pairs to a relation in a single step. Transitive closure needs a recursive definition because adding some pairs can make other pairs eligible for addition. By the recursive definition that my ancestors are my parents and the ancestors of my parents, I avoid the ellipses that I would need if I said they were my parents, my grandparents, my great-grandparents, my great-great-grandparents, ...

I said above that the transitive closure of a relation is "clearly" transitive. In mathematics, when a writer says something is "clear" or "obvious" *they are being lazy*, and you should ask them to elaborate. How would a formal proof $\hat{R}$ is transitive begin? We want to prove that, $\forall_{a,b,c \in A}\ ((a\hat{R}b \wedge b\hat{R}c) \rightarrow a\hat{R}c)$. We can start with any given $a, b, c \in A$ for which $a\hat{R}b$ and $b\hat{R}c$, and observe that the recursive rule says that $a\hat{R}c$. So this is indeed easy.

What is more interesting is that $\hat{R}$ is the smallest transitive relation that contains R. We can prove

Lemma 12.1.1. *Any transitive relation T that contains a binary relation, $R \subseteq T$, also contains its transitive closure: $\hat{R} \subseteq T$.*

Proof. Here is a proof by minimal counterexample, which is an induction variant from subsection 10.2.4.

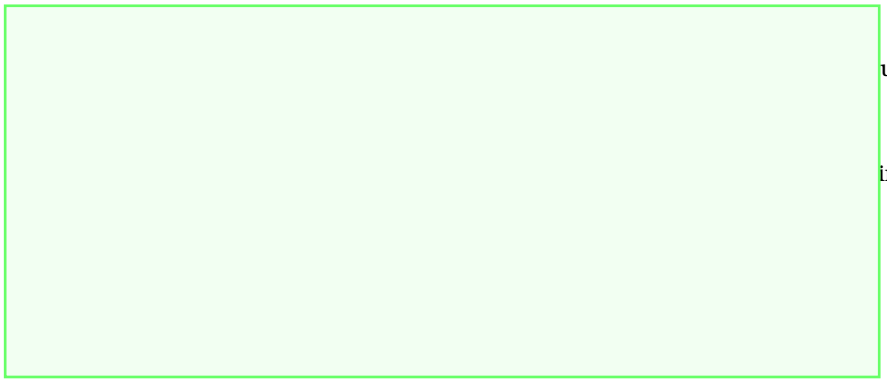
1	Given a transitive relation T that contains R, suppose, for the sake of deriving a contradiction, that $\exists_{(a,c)\in\hat{R}}$ with $(a,c) \notin T$.	Given for proof by contradiction
2	Choose $(a,c) \in \hat{R}$ that uses the recursive rule the fewest times.	Closure rule for trans. closure
3	Note that $(a,c) \notin R$,	since $R \subseteq T$
4	so $(a,c) \in \hat{R}$ because $\exists_b$ with $(a,b) \in \hat{R} \wedge (b,c) \in \hat{R}$.	Recur. rule
5	(a,b) and (b,c) each use rec rule fewer times,	And (a,c) was fewest $\notin T$
6	so $(a,b) \in T$ and $(b,c) \in T$.	
7	Thus, $(a,c) \in T$.	T is transitive.
8	This contradiction establishes the lemma.	

∎

Proof by minimum counterexample is a form of proof by contradiction. Sometimes it is easier to find a contradiction than a specific conclusion. Once you have found the idea that makes a proof work, a proof that goes forward may be easier to verify and to communicate to others. Try writing your own 8-step induction proof* before peeking at mine in figure 12.1. Which proof do you find most convincing? Which is most concise?

* This proof had 8 steps by coincidence; don't confuse that with the 8-step induction template.

Suppose that a relation is both symmetric and transitive. Must it then

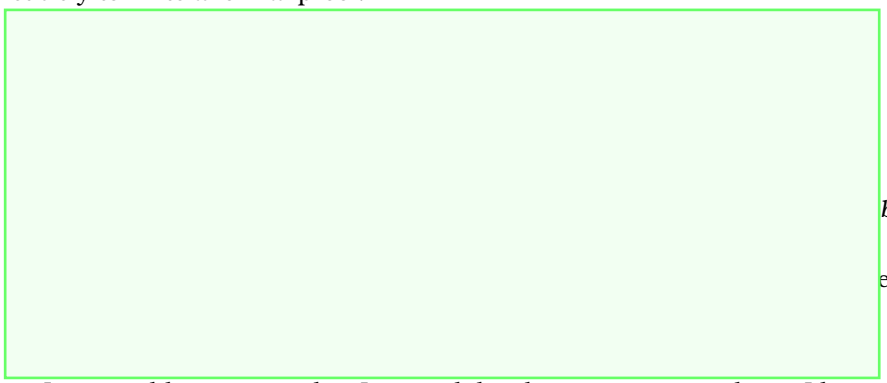


ule.

in T.

Figure 12.1: 8-step induction proof of Lemma 12.1.1; try your own first, then compare the three proofs.

be reflexive? Check this on scratch paper: symmetry says that whenever we have aRb we also have bRa, transitivity then says we have aRa, right? So let's try to write a formal proof.



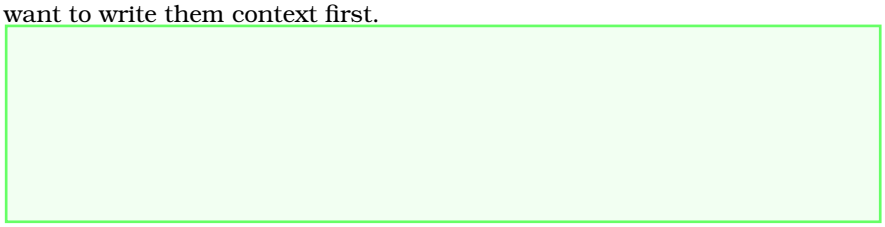
bRa.

ens)

I am unable to prove what I wanted, but by attempting to do so, I learn something new. The only way a symmetric and transitive relation $R \subseteq A^2$ can be non-reflexive is by having one or more $a \in A$ that are not related to anything. Write a two column proof of this lemma.

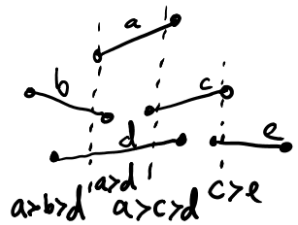
Lemma 12.1.2. *If $R \subseteq A^2$ is a symmetric and transitive relation, and $\forall_{a \in A}\ \exists_{b \in A}\ (a\ R\ b)$, then R is also reflexive.*

Proof. Many of the ideas come from the proof attempt above, but you will want to write them context first.



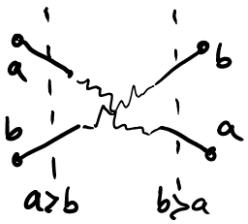
12.2 Aboveness: A partial order

On graph paper, draw a set S of n line segments that do not touch or cross. Define a binary *aboveness* relation on these segments: $\forall_{a,b \in S}$, we say that $a \succ b$ iff we can draw a vertical line that hits a above b. Prove that:



Lemma 12.2.1. *The aboveness relation $\succ$ is anti-symmetric.*

Proof. Suppose, for the sake of deriving a contradiction, that I have two segments with $a \succ b$ and $b \succ a$.



Draw an example that shows that this relation is not transitive. Three segments is enough:

Construct a new relation $\succ^*$ as the transitive closure of $\succ$. That is, $\forall_{a,c \in S}$, we say that $a \succ^* c$ iff there is a sequence of $k \geq 1$ segments $b_1, b_2, \ldots, b_k \in S$ with $a = b_1$, $c = b_k$, and $b_i \succ b_{i+1}$ for all $1 \leq i < k$. Can we prove that $\succ^*$ is a partial order?

*Read as 'superior to.'

The hard part is to show that $\succ^*$ is anti-symmetric because we cannot have a cycle in $\succ$. (Since we know there are no cycles of length 2, this might also be a good place to prove there is no minimal counterexample.) But let's start with the easy part.

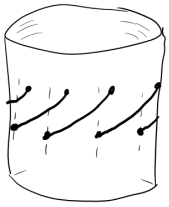
To show that $\succ^*$ is a partial order, we must show it is transitive, anti-symmetric, and irreflexive. Transitive is easy, because $\succ^*$ is a transitive closure. For practice, here are the details:

For all $r, s, t \in S$ if $r \succ^* s$ and $s \succ^* t$ then we know that there are two sequences $b_1, b_2, \ldots, b_k \in S$ with $r = b_1$, $s = b_k$, and $b_i \succ b_{i+1}$ for all $1 \leq i < k$,

and $c_1, c_2, \ldots, c_\ell \in S$ with $s = c_1$, $t = c_\ell$, and $c_i \succ c_{i+1}$ for all $1 \leq i < \ell$. We can concatenate these two sequences (after dropping the duplicate $s = c_1$) to get $r \succ b_2 \succ \cdots \succ b_k \succ c_2 \succ \cdots \succ c_{\ell-1} \succ t$, which shows $r \succcurlyeq t$.

To show that $\succcurlyeq$ is anti-symmetric and irreflexive, we want to show that there is no cycle in the $\succ$ relation: that there is no sequence $b_1, b_2, \ldots, b_k \in S$ with $b_k \succ b_1$, and $b_i \succ b_{i+1}$ for all $1 \leq i < k$. We already know that there is no such sequence of length 1 or 2.

Why, if I want to show $\succcurlyeq$ is both anti-symmetric and irreflexive, should I worry about cycles in $\succ$? First, the practical: $\succ$ is more basic than $\succcurlyeq$, and is easier to analyze through pictures, so I can find a proof more easily if I replace each $\succcurlyeq$ with a chain of $\succ$ relations. Second, the formal: if $\succcurlyeq$ is not irreflexive then there is a cycle in $\succ$. If $\succcurlyeq$ is not anti-symmetric then we can link two chains $r \succcurlyeq s$ and $s \succcurlyeq r$ into a cycle in $\succ$. Thus, if we show there is no cycle in $\succ$, then we have shown that $\succcurlyeq$ is both anti-symmetric and irreflexive.

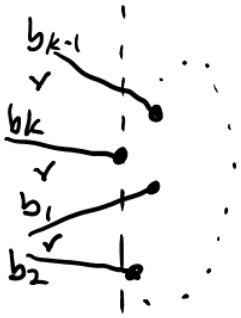


By the way, if we rolled the graph paper into a cylinder, and then drew line segments, the $\succcurlyeq$ relation might not be anti-symmetric. Anti-symmetry is a property of the plane; what in your proof depends on being in the plane?

Lemma 12.2.2. *The relation $\succcurlyeq$, which is the transitive closure of the aboveness relation $\succ$, is a partial order.*

Proof. Suppose, for the sake of deriving a contradiction, that there is a cycle $b_1, b_2, \ldots, b_k \in S$ with $b_k \succ b_1$, and $b_i \succ b_{i+1}$ for all $1 \leq i < k$. Without loss of generality, we can suppose that we assigned indices so that b_k is a segment whose rightmost point is furthest to the left. (This is a choice we can make in the plane, but not in the cylinder.)

I claim that $b_{k-1} \succ b_1$ so that $b_1, b_2, \ldots, b_{k-1}$ is a shorter cycle. Why?



Thus $\succcurlyeq$ is irreflexive and anti-symmetric, and is a partial order. ∎

This means is that a 2-dimensional game like pick-up sticks, or Mikado, is easy: in any direction you choose, some stick will be the top of the pile and can be translated to infinity without disturbing others. In 3-d this is not true; you can make collections of sticks (suitably fattened, but without introducing notches) so that in no direction can you translate one stick away from the rest.

12.3 Equivalence relations and finite state automata

In many tasks—parsing code for keywords or variable names, searching DNA sequences, scanning network traces to identify *denial of service* (DOS) attacks—we want to recognize strings that belong to a given language (a set of strings, as defined in subsection 3.1.3.) In this section, we see how to automatically make the simplest possible Kara programs to recognize the important class of regular languages.

Usually, finding the simplest possible program to do anything is hard, if not impossible.* In this case, we will be able to use equivalence relations to reveal and exploit structure of the strings inside and outside the language that would be otherwise hidden in particular examples—equivalence relations will help us see the forest that would otherwise be hidden by the trees.

*The simplest program to test if your code can go into an infinite loop? Not possible, because from it you can create a program that goes into an infinite loop iff it says it does not.

The rest of this section deals with material that you will see again if you take a class on languages and automata; the key fact to learn is that equivalence relations partition their sets into equivalence classes—that is the structure that we will exploit more than once.

12.3.1 Regular languages and simplified Kara

Regular languages are a useful class of languages that can be defined recursively:

1. Base: The empty language $\emptyset$ is regular, the language of the empty string $\{\Lambda\}$ is regular, and, for all $a \in \Sigma$, the language $\{a\}$ is regular.
2. Recursive Rule: If languages L and M are regular, then LM, $L \cup M$. and L^* are all regular.
3. Closure: Only languages that can be constructed from base cases by a finite number of applications of the recursive rule are regular.

Here are some example regular languages over the alphabet that Kara can write: $\Sigma = \{\text{🍇}, \square\}$:

- Any language with a finite number of strings is regular.
- The language of strings in which every $\square$ is immediately preceded by a 🍇: $L = \{\text{🍇}, \text{🍇}\square\}^*$.
- The language of strings in which no 🍇s follows a $\square$s: $L = \{\text{🍇}\}^*\{\square\}^*$.
- The language of strings where the number of $\square$s is a multiple of 3: $L = \{\text{🍇}\}^*(\{\square\}\{\text{🍇}\}^*\{\square\}\{\text{🍇}\}^*\{\square\}\{\text{🍇}\}^*)^*$.
- The substrings of $\{\text{🍇}\text{🍇}\square\}^*$ obtained by starting and ending at arbitrary characters: $L = \{\text{🍇}\square, \square, \Lambda\}\{\text{🍇}\text{🍇}\square\}^*\{\Lambda, \text{🍇}, \text{🍇}\text{🍇}\}$.

One example language that is not regular, as we shall see in Corollary 12.3.4, is the language of strings with the same number of 🍇s as $\square$s.

We can simplify Kara to build regular language recognizers on $\Sigma = \{\clubsuit, \square\}$.

1. Write the string to be tested in clovers and blanks from left to right on a $1 \times n + 1$ tape.
2. Start Kara at the leftmost end of the string, facing right.
3. Put a tree one space after the end of the string. (For the empty string, that puts the tree next to Kara.)
4. Kara may optionally stop if she senses the tree ahead.
5. After each read, Kara must move one square to the right. (It doesn't matter what Kara writes; she won't read it again.)
6. If Kara senses the tree and stops, then the string *accepted* as being in the language; if Kara crashes, then it is not. Kara is said to *recognize* the language of all accepted strings.

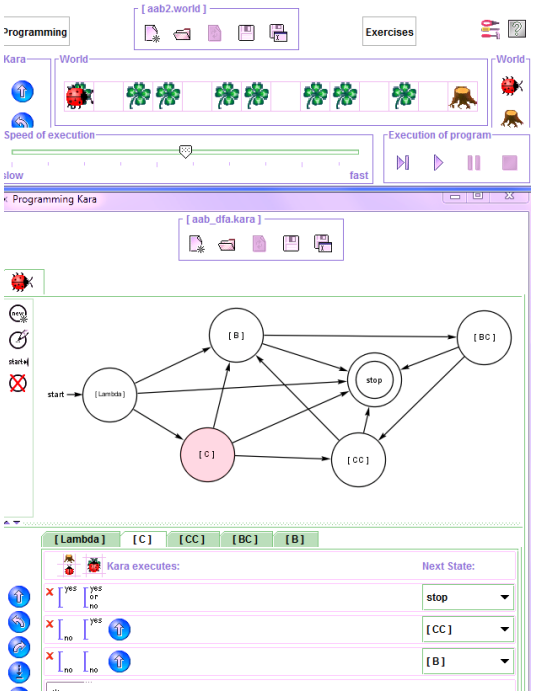


Figure 12.2: Kara program to *recognize* the language of substrings of $\{\clubsuit\clubsuit\square\}^*$. A string is in the language iff Kara stops without crashing.

Consider the example world at the top of figure 12.2. Kara on a clover, so the string to be tested is in the language of substrings of $\{\clubsuit\clubsuit\square\}^*$. We can draw a simplified Kara program as a state diagrams: a graph with a vertex for each state, and edges labeled with the clover, blank, or tree to be read as Kara transitions to the next state. Add the labels at left to make a program that correctly recognizes strings of this language.

Let's adopt a drawing convention to reduce clutter: rather than drawing the arrows to Stop that are labeled with trees, put double circles around the states they come from. These are *accepting states*; all states of figure 12.2 would be accepting. To be complete, however, we should add a non-accepting "garbage state" to catch all strings that would otherwise crash Kara. Then Kara would process each string to the end.

Each simplified Kara program can be written as a 5-tuple $K = (S, \Sigma, \delta, s_0, F)$ containing a set of states S, alphabet $\Sigma = \{\clubsuit, \square\}$, a transition function $\delta\colon S \times \Sigma \to S$ that always moves to the next symbol, a distinguished start state $s_0 \in S$, and a set of final or accepting states $F \subseteq S$. This is a complete description of machine K, capturing all the relevant detail from K's state diagram. *K accepts string a* iff by starting at the start state and processing a to the end, it ends in an accepting state. *K recognizes the language L_K* of all strings that it accepts.

There is a nice binary relation on strings of Σ^* that comes from the state diagram: Define any two strings $a, \beta \in \Sigma^*$ to be *equivalent with respect to K*, denoted $a \equiv_K \beta$, iff from the start state they reach the same state. (Figure

out which strings end in which states in figure 12.2.)

- This relation is reflexive, symmetric, and transitive (why?), so it is an equivalence relation. It partitions strings of Σ^* into a number of classes that equals the number of states.
- If $a \equiv_K \beta$ then $a \in L_K \leftrightarrow \beta \in L_K$—one string is accepted iff the other is accepted.
- For any $\gamma \in \Sigma^*$, if $a \equiv_K \beta$ then $a\gamma \equiv_K \beta\gamma$—once strings travel together, they stay together.

The relation $\equiv_K$ from a machine K tells me about its language L_K; I'd like a language L to tell me about its machines. We will do that in several steps (using at least three more equivalence relations) in the rest of this chapter.

12.3.2 Recognizing a regular language with superKara

Let's show that, for any regular language L, we can create a Kara program to recognize L. Let's initially cheat, and give Kara two superpowers: First, we allow Kara to *teleport* from one state to another state without reading or moving. An edge labeled with the empty string Λ in the state graph gives Kara the option to teleport if she wishes. Second, if Kara has a choice of where to go on reading a clover/blank (or of teleporting and reading later), she can copy the world, *clone* herself, and try all possibilities.* We change the domain and range of the transition function $\delta\colon S \times \Sigma \cup \{\Lambda\} \to \mathcal{P}(S)$.

*Also known as non-determinism.

A string is accepted by superKara iff there exists some clone that stops at the end of a in an accepting state. We can think of Kara as running independent clones in parallel, or as having an uncanny ability to choose a path that leads to acceptance, if one exists. figure 12.3 shows three diagrams that use these superpowers; which languages are these?†

†Hint: The first pair in figure 12.3 accept a common language, as do the last pair.

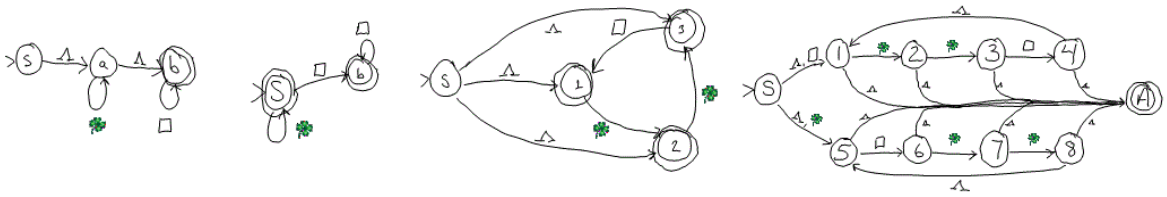


Figure 12.3: Four example diagrams that use the teleportation and cloning powers of superKara. Which languages do these diagrams accept? I.e., from the start state, which string end in a double-circled accepting state?

Any regular language $L \subseteq \Sigma^*$ has a recursive definition that builds L from simpler regular languages. We can build a superKara state diagram accept-

ing L by induction – combining state diagrams for the simpler languages. Each of our diagrams will have a single accepting state. For example, state diagrams in figure 12.4 accept base case languages for an example alphabet $\Sigma = \{a, b\}$.

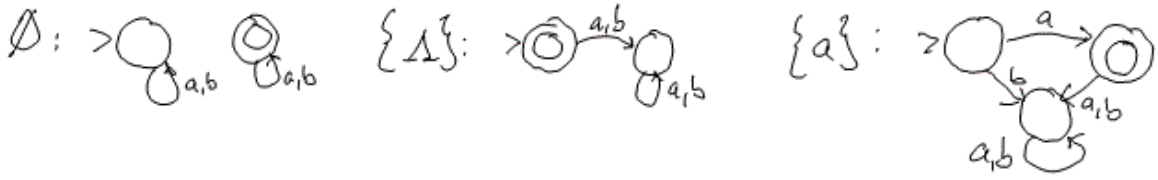


Figure 12.4: Machines with $\Sigma = \{a, b\}$ that accept base case languages $\emptyset$, $\{\Lambda\}$, and $\{a\}$. The machine for $\{b\}$, similar to the machine for $\{a\}$, would complete the set of base cases.

figure 12.5 suggests how to combine machines for languages L and M to make machines that accept LM, $L \cup M$. or L^*. Here we use superKara's ability to teleport and clone to make the constructions easy, and to always have one start state and one accepting state.

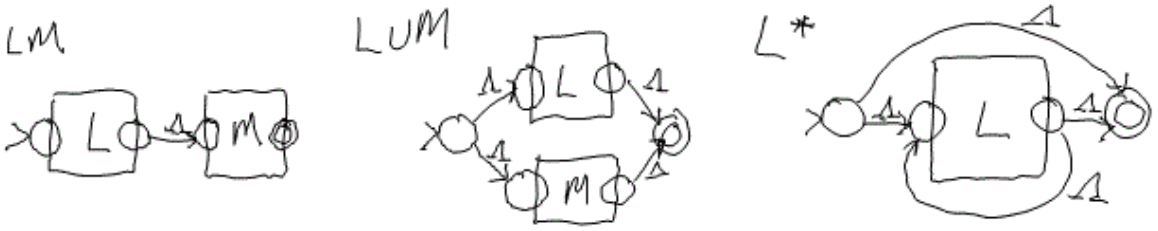


Figure 12.5: Combining machines for languages L and M to make machines that accept LM, $L \cup M$. or L^*.

For any a regular language $L \subseteq \Sigma^*$, we simply follow the (finite number of) steps of L's recursive definition to build a machine K that accepts every string in L and rejects every string not in L. We would prove that K recognizes L by induction on the number of recursive steps in the definition of L.

12.3.3 Simple Kara simulates superKara

Next, we would like to remove Kara's superpowers of teleportation and cloning, so that each state,symbol pair goes to exactly one state. To begin, let R be the relation on states with sRt iff there is an edge from s to t labeled with Λ, and form $\hat{R}$ from R by reflexive and transitive closure.* Relation $\hat{R}$ is not necessarily symmetric, but is reflexive and transitive—a partial order. Define the Λ-*closure* of any set of states A as the set of states related to them by $\hat{R}$: $\Lambda\text{-closure}(A) = \{t \mid \exists_{s \in A}\ s\hat{R}t\}$.

* $s\hat{R}t$ iff there is a path from s to t labeled with Λs.

From the machine $K = (S, \Sigma, \delta\colon S \times \Sigma \cup \{\Lambda\} \to \mathcal{P}(S), s_0 \in S, F \subseteq S)$ we define a new machine $K' = (S', \Sigma, \delta'\colon S' \times \Sigma \to S', s'_0 \in S', F' \subseteq S')$ that essentially runs all possible clones of K in parallel. We define

- $S' = \mathcal{P}(S)$; states of K' are the sets of old states that clones of K can be in while processing an input string. $\emptyset$ always serves as a garbage state.
- $s'_0 = \Lambda$-closure(s_0); start in the set of states that clones in K can reach by teleporting from s_0 (including s_0),
- $F' = \{A \mid A \subseteq S \wedge A \cap F \neq \emptyset\}$; a string is accepted iff, after processing to the end, there exists a clone in an accepting state of K.
- $\forall_{A \in S', b \in \Sigma}\ \delta'(A, b) = \Lambda\text{-closure}(\bigcup_{t \in \Lambda\text{-closure}(A)} \delta(t, b))$. In words, if superKara has clones in the states of A, she teleports clones to all places they can reach, advances clones to all possible next states reading b, then teleports clones to all possible output states. The resulting set of states of K is a state of S' for K'.

This new machine K' is completely deterministic; for each state (set of old states) there is a single new state to go to for each letter in Σ. We could prove by induction on the length of a string a that K' accepts a iff K accepts a. Thus, K recognizes the language L.

The new machine can be huge—the superKara construction from a regular language L already created many states, and this conversion to simple Kara constructs all subsets of these states. We can reduce the construction to only the states that are reachable from the start s'_0—doing so also gives, for each state, a string that ends in that state. Using yet another relation we can do better, and make the smallest machine that recognizes L.

12.3.4 An equivalence relation $\equiv_L$

In order to capture the idea of strings traveling together with respect to a language, define $a \equiv_L \beta$ iff $\forall_{\rho \in \Sigma^*}\ (a\rho \in L \leftrightarrow \beta\rho \in L)$—that is, two strings are equivalent iff every possible extension puts both inside or both outside the language L.*

*$\equiv_L$ is like $\equiv_K$, but different. Can you explain how?

Lemma 12.3.1. *$\equiv_L$ is an equivalence relation.*

12.3.5 Proof in detail:

We need to show that $\equiv_L$ is reflexive, symmetric, and transitive. All three are easy; let me write the proofs in full detail, then more briefly.

Detailed proof. To show that $\equiv_L$ is an equivalence relation, we need to show that it is reflexive, symmetric, and transitive.

Refl: I want to show that $\forall_{a\in\Sigma^*}\ a \equiv_L a$. Expanding the definition, this means I want to show that $\forall_{a,\rho\in\Sigma^*}\ a\rho \in L$ iff $a\rho \in L$ is always true. But $p \leftrightarrow p$ is always true, so $\equiv_L$ is reflexive.

Sym: I want to show that $\forall_{a,\beta\in\Sigma^*}$ if $a \equiv_L \beta$ then $\beta \equiv_L a$. But $a \equiv_L \beta$ means $\forall_{a,\beta,\rho\in\Sigma^*}\ (a\rho \in L \leftrightarrow \beta\rho \in L)$, and since $\leftrightarrow$ is symmetric, $(\beta\rho \in L \leftrightarrow a\rho \in L)$, and $\equiv_L$ is symmetric.

Trans: $\forall_{a,\beta,\gamma\in\Sigma^*}$ if $a \equiv_L \beta$ and $\beta \equiv_L \gamma$ then $a \equiv_L \gamma$. But this expands to $\forall_{a,\beta,\gamma,\rho\in\Sigma^*}$, if $(a\rho \in L \leftrightarrow \beta\rho \in L)$ and $(\beta\rho \in L \leftrightarrow \gamma\rho \in L)$, then $(a\rho \in L \leftrightarrow \gamma\rho \in L)$. But $\leftrightarrow$ is transitive—for all p, q, r, if $p \leftrightarrow q$ and $q \leftrightarrow r$, then $p \leftrightarrow r$, so $\equiv_L$ is transitive.

QED

Brief proof. We aim to show that $\equiv_L$ is reflexive, symmetric, and transitive.

Refl: This is trivial. $\forall_{a,\rho\in\Sigma^*}\ a\rho \in L$ iff $a\rho \in L$, so $a \equiv_L a$.

Sym: This is trivial since $\leftrightarrow$ is symmetric. $\forall_{a,\beta,\rho\in\Sigma^*}$ if $(a\rho \in L \leftrightarrow \beta\rho \in L)$ then $(\beta\rho \in L \leftrightarrow a\rho \in L)$, so if $a \equiv_L \beta$ then $\beta \equiv_L a$.

Trans: This is almost trivial. $\forall_{a,\beta,\gamma,\rho\in\Sigma^*}$ if $(a\rho \in L \leftrightarrow \beta\rho \in L)$ and $(\beta\rho \in L \leftrightarrow \gamma\rho \in L)$, then $(a\rho \in L \leftrightarrow \gamma\rho \in L)$, so if $a \equiv_L \beta$ and $\beta \equiv_L \gamma$ then $a \equiv_L \gamma$.

QED

Any equivalence relation partitions its set into *equivalence classes*. For $\equiv_L$ defined on strings Σ^*, the equivalence class of string $a \in \Sigma^*$ is the set $[a] = \{\beta \in \Sigma^* \mid a \equiv_L \beta\}$.

Lemma 12.3.2. *Every string in Σ^* is in exactly one equivalence class.*

Proof. This proof uses only the properties that $\equiv_L$ is reflexive, symmetric, and transitive, so it applies to any equivalence relation. First, every string is in some equivalence class: $\forall_{a\in\Sigma^*}\ a \in [a]$ because $\equiv_L$ is reflexive.

Second, if it looks like a is in a second equivalence class, it is actually the same one: I want to show $\forall_{a,\beta\in\Sigma^*}$ that if $a \in [\beta]$ then $[a] = [\beta]$.
So assume $a \in [\beta]$, which means that $a \equiv_L \beta$.
Consider any $\gamma \in \Sigma^*$.

$$\begin{aligned} \gamma \in [a] &\Leftrightarrow \gamma \equiv_L a && \text{by definition of } [a] \\ &\Leftrightarrow \gamma \equiv_L \beta && \text{by transitivity and symmetry of } \equiv_L \\ &\Leftrightarrow \gamma \in [\beta]. && \text{by definition of } [\beta] \end{aligned}$$

Thus, the equivalence classes $[a]$ partition Σ^*. QED

12.3.6 The smallest machine for L

We can make a machine K_L for L from the equivalence classes of $\equiv_L$.

- Make a state for each equivalence class: $S_L = \{[a] \mid a \in \Sigma^*\}$.
- Define the transition function $\delta_L \colon S_L \times \Sigma \to S_L$ by $\forall_{[a]\in S_L}\ b \in \Sigma$ $(\delta_L([a], b) = [ab])$.
- The accepting states are $F_L = \{[a] \mid a \in L\}$.
- The start state is $[\Lambda]$.
- So, $K_L = (S_L, \Sigma, \delta_L, F_L, [\Lambda])$.

I leave it to you to prove that K_L recognizes L, probably by using induction on the length of a string to show that K_L accepts every string in L and rejects every string not in L. The next lemma does prove that K_L has the smallest number of states of all machines to accept L.

Lemma 12.3.3. *No machine with fewer states than the number of equivalence classes of $\equiv_L$ can accept the language L.*

Proof. Consider a machine K with fewer states than the number of equivalence classes of $\equiv_L$. By the pigeonhole principle, there must exist two strings, a and β, with $a \equiv_K \beta$ and $a \not\equiv L\beta$. In words, a and β end in the same state of K, but there is some string ρ for which exactly one of $a\rho$ and $\beta\rho$ is in the language L. But both or neither of $a\rho$ and $\beta\rho$ are in the language accepted by K, so K does not accept language L. ∎

This has a quick and important corollary: There are languages that are not regular. Simple Kara cannot remember an arbitrarily large count.

Corollary 12.3.4. *The language $M = \{a^m b^m \mid m \in \mathbb{N}\}$ is not accepted by any machine with a finite number of states.*

Proof. There are an infinite number of equivalence classes for $\equiv_M$: For all $m \in \mathbb{N}$, the strings a^m is in it's own equivalence class with respect to $\equiv_M$, because adding b^n to it puts it into the language M iff $m = n$. ∎

It is unusual to be able to prove that we can make the smallest program to do something, especially something as useful as recognizing a whole class of languages. I think this is the second-best demonstration of the benefit of equivalence relations; second-best because I have to figure out the equivalence classes from a language, and that doesn't seem easy. But we will fix that by getting the classes from a machine.

12.3.7 Reducing K' to K_L

subsection 12.3.2 and 12.3.3 gave a construction of a way-too-big machine K' that recognizes L. The relation defined on this machine, $\equiv_{K'}$, partitions

strings of Σ^* into classes that are a finer partition of the classes of $\equiv_L$. So we just have to figure out what states of K' have to be merged to make the states of the minimum-state machine K_L.

Let's actually start with states of K' merged together and separate those that must be separated. Initially, let $\Pi_0 = \{S_0, S_1\}$ be a partition of states S' into accepting states S_1 and non-accepting states S_0.

Given a partition Π_i, we consider each possible set of states $T \in \Pi_i$ and each $b \in \Sigma$ and, you guessed it, define equivalence relations $\simeq_{i,b} \subseteq T \times T$ with states $s \simeq_{i,b} t$ iff the next states on reading b, namely $\delta'(s, b)$ and $\delta'(t, b)$, are in the same set in partition Π_i. (Why are these equivalence relations?) If T has more than one equivalence class under $\simeq_{i,b}$, then we form Π_{i+1} from Π_i by removing T and replacing it with its equivalence classes under $\simeq_{i,b}$.

When no choice of T or b change the partition Π_i, we stop. Each element of the partition Π_i corresponds to a state S_L of our minimum-state machine K_L. This requires proof, which can be done by induction, but that is for a later computer science class on automata theory or compilers. Here, let me just close by working out the example for the language of substrings of $\{aab\}^*$ in figure 12.6.

12.4 Summary

The language of relations is common in computer science. By studying special types of relation, we learn properties common to all relations of that type.

The examples of partial orders and equivalence relations are complex, even though they follow from simple definitions. You will see finite state machines and regular languages in subsequent computer science classes, so the one thing to learn from the last example is that **equivalence relations are helpful tools because they partition their set into equivalence classes.** Of course, to learn that, you also need to know what are binary relations and equivalence relations on a set, how to show a relation is an equivalence relation, what is a partition, and what are equivalence classes. These are very general and abstract concepts, and most examples of them (like '=') are trivial, so I wanted to show an important, non-trivial example.

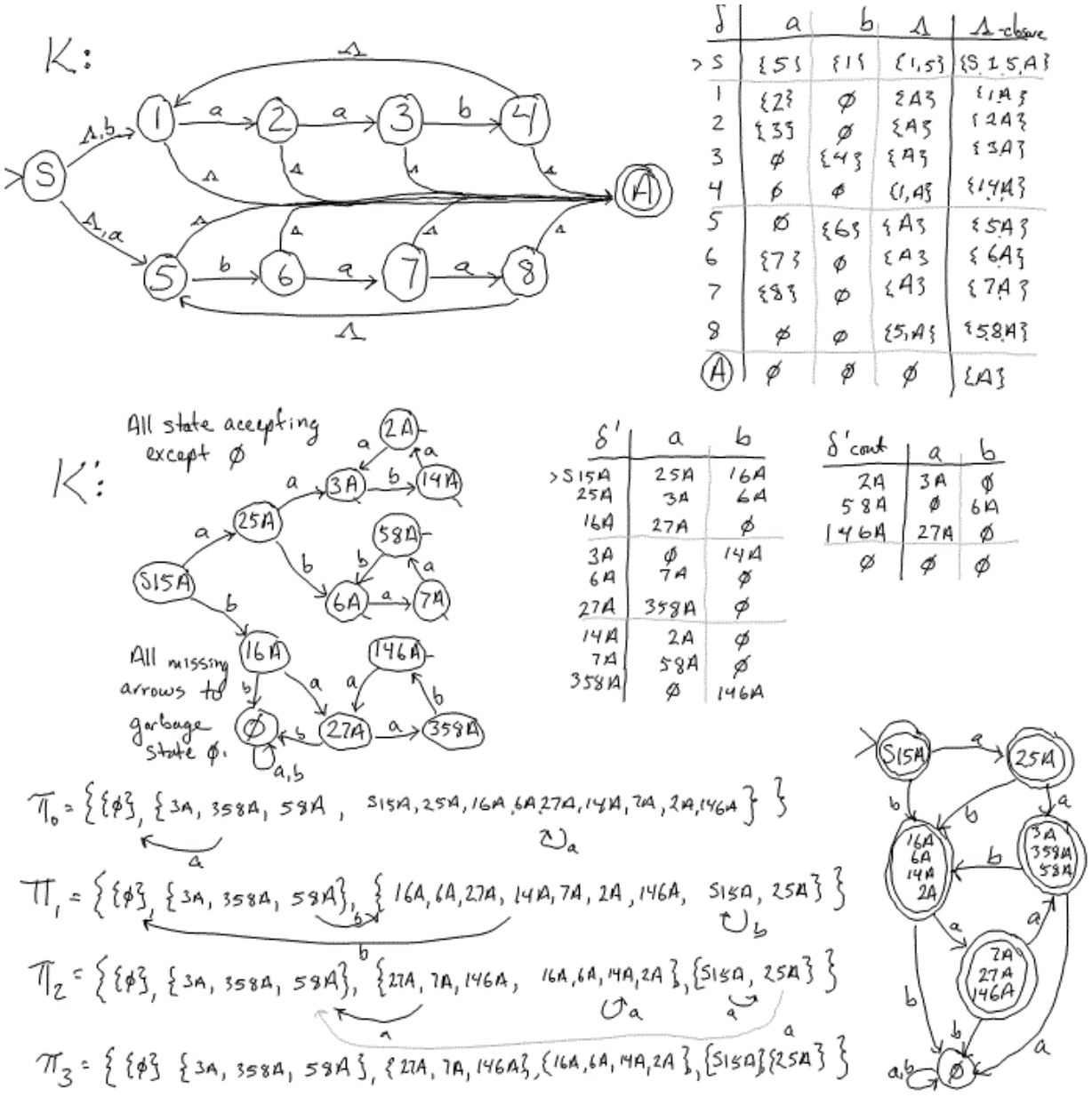


Figure 12.6: Constructing and minimizing a finite state machine for the language of substrings of $(aab)^*$ shows the role of equivalence relations and their partitions.

12.5 Exercises and Explorations

Quiz Prep 12.1. On the set of all intervals of the real line, $\{[a,b] \mid a \leq b\}$, we can define the following binary relations. Classify each as a total order, partial order (not total), equivalence relation, or neither.

a ? Contained: $[a,b] \prec [c,d]$ iff $[a,b] \subseteq [c,d]$.
b ? Nesting: $[a,b] \prec [c,d]$ iff $[a,b] \subseteq [c,d]$ or $[c,d] \subseteq [a,b]$.
c ? Narrower: $[a,b] \prec [c,d]$ iff $b-a \leq d-c$.
d ? Intersecting: $[a,b] \prec [c,d]$ iff $[a,b] \cap [c,d] \neq 0$.
e ? Disjoint: $[a,b] \prec [c,d]$ iff $[a,b] \cap [c,d] = 0$.
f ? Sliding: $[a,b] \prec [c,d]$ iff $a < c$ and $b < d$.
g ? Lexicographic: $[a,b] \prec [c,d]$ iff either $a < c$ or ($a = c$ and $b \leq d$).

Quiz Prep 12.2. Show that the divides relation is a partial order for the sets of positive integers, $\mathbb{Z}^+$, and non-negative integers, $\mathbb{N}$, but not the set of all integers, $\mathbb{Z}$. Recall that $a|b$ iff $\exists m \in \mathbb{Z}\ ma = b$.

Hint:

Exercise 12.3. Prove that subset ($\subseteq$) is the universal reflexive partial order. That is, if you are given a poset $(S, \preceq)$ with $\preceq$ being reflexive, then you can make a family of sets $\mathcal{F}_S$ and a bijection $f : S \to \mathcal{F}_S$ so that for all $a, b \in S$, $a \prec b$ iff $f(a) \subseteq f(b)$.

Hint:

▶

Exercise 12.4. For a poset $(S, \prec)$ with $n = |S|$ elements, a *chain* of length k is a totally ordered sequence of k elements from S: $s_1 \prec s_2 \prec \cdots \prec s_k$. An *antichain* of size k is a subset $X \subseteq S$ of k elements, no two of which compare: $\forall a, b \in X\ a \not\prec b$.

Show that any poset has either a chain of length at least $\sqrt{n}$ or an antichain of size at least $\sqrt{n}$.

Exercise 12.5. You are a xenobiologist studying Zorches. When two Zorches get together, one or both may "freeb" (glow blue). Write aFb if a freebs when close to b. Some Zorches freeb near your spaceship, and you eventually realize that this is because they are seeing their own reflection, aFa.

A particular group of four Zorches, $Z = \{Aye, Bee, Cea, Dii\}$, change their freeb pattern every day, but it is always reflexive (each freebs with its reflection), symmetric (for any pair, either both or neither freeb), and transitive (if aFb and bFc then aFc). How many days can they go before repeating a freeb pattern? Let's break this into smaller questions.

1. How many elements are there in the Cartesian product $Z \times Z$? ?

2. How many different relations from Z to Z are possible?

3. How many different reflexive relations from Z to Z are possible?

4. How many different reflexive and symmetric relations from Z to Z are possible?

5. How many different equivalence relations from Z to Z are possible?

►

Exercise 12.6. Answer these questions from the end of subsection 12.3.1 on the relation $\equiv_K$, which defines two strings to be equivalent with respect to machine K iff from the start state they reach the same state.

1. Show that $\equiv_K$ is indeed an equivalence relation.
2. Explain why the other two statements use 'if' rather than 'iff': (If $\alpha \equiv_K \beta$ then $\alpha \in L_K \leftrightarrow \beta \in L_K$, and if $\alpha \equiv_K \beta$ then, for any $\gamma \in \Sigma^*$, $\alpha\gamma \equiv_K \beta\gamma$.

Exercise 12.7. Show that if a binary relation R is irreflexive and transitive, then it is anti-symmetric.

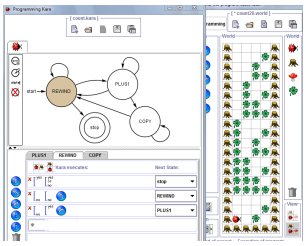
Exercise 12.8. In a wolf-pack, suppose that the domination relation is a transitive comparability relation. That is, for all $a \neq b$, either aDb or bDa (and possibly both), and for all a, b, c, if aDb and bDc then aDc. A pack-leader is a wolf that dominates all others. Show that every pack has at least one pack leader. What additional condition could I place on D to ensure that there is exactly one pack leader?

Chapter 13
Graphs and Trees

The secret to productivity in so many fields – and in origami – is letting dead people do your work for you. . . . When you get math involved, problems that you solve for aesthetic value only, or to create something beautiful, turn around and turn out to have an application in the real world.

—Robert Lang, TED Feb 2008, 2:19 & 15:25.

In this book, graphs and trees are discrete structures that are defined from sets and tuples: e.g., a *directed graph* is a finite set of *vertices* V and *edges* $E \subseteq V \times V$, representing a relation between the vertices (e.g., the "influences," "likes," or "friends" relationship). This abstract concept is not to be confused with the *graph of a function*, which is a drawing of the values of taken on by the function. Our graphs need not even be drawn, although small ones almost always are because our visual system can perceive connection, density, and pattern information very rapidly. For an example, in Kara's programming window, transitions out of a state are shown in two ways: as edges of the *state diagram* and rows of a table. In a well-drawn state diagram it takes only a glance so check if there exists a path from the start to the stop state, but in a table this takes careful inspection.



Objectives: You will be able to apply your knowledge of sets and tuples to understand the basic definitions of graphs and trees, You will meet many variations of graphs, including embedded graphs with planar drawings and triangulations. You will be able to use your knowledge of proof techniques to derive a series of properties that follow from these definitions; some of these properties have been known for centuries, such as that a tree on v vertices has $v - 1$ edges, and a connected planar graph with v vertices, e edges, and f faces satisfies Euler's relation, $f - e + v - 1 = 1$. Others are recent, such as a way to encode triangulations using $4v$ bits, v of which are 1s, which is related to schemes for compressing characters and terrain in 3-d video games. Throughout your study of computer science you will meet many applications and puzzles that demonstrate the usefulness of both graphs and trees.

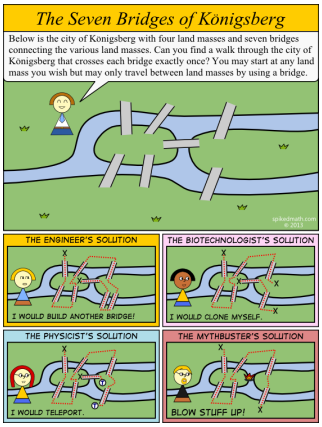


Figure 13.1: From Spikedmath

The study of graph theory is considered to have begun with L. Euler's 1736 paper on the puzzle of the bridges of Königsberg [4]: Is it possible to walk across all seven bridges without crossing a bridge twice? [?] This chapter is dominated by questions and puzzles that introduce basic definitions and variations of graphs, and develop some of my favorite applications that are relevant to multiplayer video games: drawing terrain, location positions, and compressing models. I've selected several problems whose solutions

depend upon properties of *trees*, which are special types of graphs that are ubiquitous in computer science. (We've seen rooted trees defined recursively in subsection 8.1.6.)

The emphasis of this chapter is less the specific results than to practice problem-solving using our precise language for sets, tuples, counting, relations, and proof techniques. Graph theory and its applications is just the domain in which we find interesting problems to solve. I stronly suggest that you first copy or summarize each question in a notebook (paper or electronic), then try answer it yourself. After you have wrestled with the problem yourself, then look at hidden hints or answer sketches and revisit your answer. Since proof is about communicating ideas clearly, it is good to do this as a group of two to four classmates.

In my experience, those who go beyond what is assigned emerge with a clearer understanding. The practice will not only help with sets, tuples, and proof, but you can expect to see graph data structures and graph algorithms in future classes across all areas of computer science. You can also find entire courses on graph theory, graph algorithms, and graph drawing.

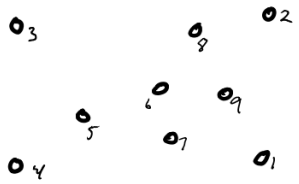
13.1 A draw-it-yourself chapter outline

Get your scratch paper ready, because I would like to introduce the main concepts for (undirected) graphs in an informal way by having you draw them. Each of the terms mentioned here is more formally defined later in the chapter.

Basic graph definitions: degree, paths, cycles. In section 6.1, we noticed that we could draw a binary relation $R \subset A \times A$ by drawing a labeled dot (*vertex*) for each element of A and an arrow for each pair $(a, b) \in R$. This is a *directed graph*, as defined in the first sentence of this chapter. If the relation is symmetric and irreflexive*, then the graph is an *undirected graph*; we can draw each pair (a, b) as a line or curve with no arrowheads, since whenever we have $(a, b) \in R$, we also have $(b, a) \in R$.

*That is, $\forall (u, v) \in R$ we know both $u \neq v$ and $(v, u) \in R$.

Draw four points that form a box, then draw five points inside the box so no three of the nine points lie on a common line. Make a few copies, then answer the following questions.



1. What is the maximum number of line segments (undirected edges) that you can draw between the nine dots (vertices†) without repeating an edge? ? This is the *complete graph* K_9.

2. The *degree* of a vertex is the number of edges incident on it. For example, every vertex in K_9 has degree 8. Notice that the sum of all degrees counts every edge twice. One implication is that the number of odd degree vertices is always even. Check the graphs you draw and the ones in figure 13.2.

†*vertices* is the plural of *vertex*: one vertex, two vertices.

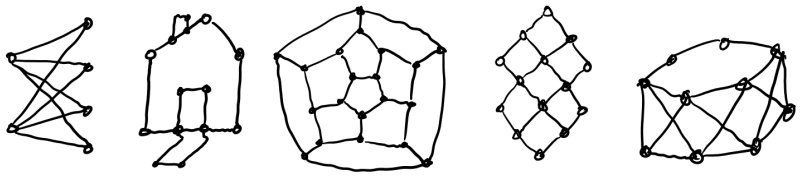


Figure 13.2: Which have paths (and cycles) that visit all vertices with no repetition? All edges?

3. In a graph, a *path* from u to v is a sequence of vertices $(w_0, w_1, \ldots, w_k)$ with $w_0 = u$, $w_k = v$ and, for all $1 \leq i \leq k$, the pair (w_{i-1}, w_i) is an edge. An undirected graph is *connected* iff for any two vertices u, v there exists a path from u to v.

Paths are allowed to repeat vertices and edges, but suppose we forbid repetition. Which is easier: For a given graph, determine if there is a path that visits all edges without repetition, or determine if there is a path that visits all vertices without repetition? Try with the graphs of figure 13.2.

4. A *cycle* is a path that begins and ends at the same vertex and repeats no edge. Which of the graphs of figure 13.2 have cycles that visit all edges? Which have cycles that visit all vertices without repeating a vertex?

5. Draw any graph in which every vertex has degree at most two, and observe that it consists of a collection of disjoint* paths and cycles.

*Disjoint means that the paths and cycles partition the sets of edges and vertices.

Trees: different, equivalent definitions. Draw straight line edges to form a connected graph on your 9 points. What is the smallest number of edges that will make the graph connected?[?] Does the number depend on where the points are placed?[?]

6. A graph is called a *tree* iff it is connected and has no cycles. Observe that your graph is a tree under this definition, and that if you add any edge you will form a cycle. Are all the graphs of figure 13.3 really trees?

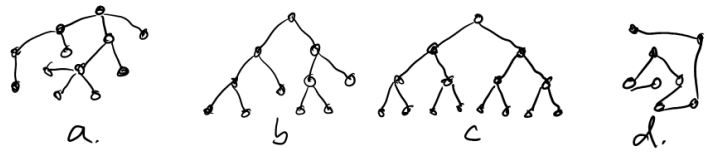


Figure 13.3: Some example trees: a) free b) binary, c) full binary, d) path

7. In 8.1.6, we've seen a recursive definition of rooted trees that combines trees at their root nodes. Can you make your tree from 6 and the trees of figure 13.3 in this way? (Does it matter what vertex you choose as root?)

8. In a tree, a vertex of degree 1 is called a *leaf*. In your data structures class, you may have seen a different recursive definition of trees that adds one leaf at a time. Can you make your tree from 6 and the trees of figure 13.3 by repeatedly adding a leaf?

9. Any graph that you form by starting with one vertex then adding one vertex and one incident edge until you have n vertices must have $n-1$ edges. We will see that all trees can be made this way. In fact, any graph G with n vertices that has two of these three properties is a tree, and so has the third property also:
   1. Graph G is connected.
   2. Graph G is acyclic.
   3. Graph G has $n-1$ edges.

Planar embeddings and planar graphs. "Graph" is an abstract concept: vertices are given identities but no fixed locations. When we draw a graph we *embed* it in the plane by assigning locations to each vertex and edge, but the drawing does not change the abstract graph.

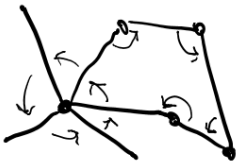
In *planar drawings* or *planar embeddings*, edges are not allowed to touch except at shared vertices. In question 6 you probably chose a planar embedding for your tree, though nothing in the question forbade edge crossings. You probably also drew edges as straight line segments, although the question didn't forbid more meandering curves.

10. A classic question is to determine the family of *planar graphs*: graph G is planar iff there exists an assignment of its vertices and edges to points in the plane that is a planar drawing. All but the last graph in figure 13.2 are planar, although the first picture is not a planar drawing. Create a planar drawing to show that it, too, is planar.

11. Create a graph with few edges that has no planar drawing; that for all assignments of vertices to points in the plane there exists some pair of edges that are not incident on the same vertex, yet still intersect.

12. Does your graph still require crossings even if you are not required draw each edge (u, v) as a straight line segment $\overline{uv}$, but may draw it as a curve or sequence of segments that wends its way from u to v?

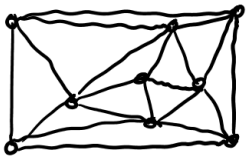
In a planar graph the edges around a vertex have a particular order. For example, we might define *next* for giving the next edge counter-clockwise (ccw) around a vertex, and its inverse *prev*, so that, for all vertices u, v, w, we have $(v, u).next = (v, w)$ iff $(v, w).prev = (v, w)$, and $(v, u).next^n = (v, u)$ iff the degree of v divides n.



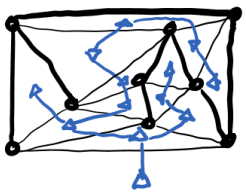
13. From this we can define the concept of a *face*, which is a region bounded by a cycle of edges and vertices. What I've just given, however, is an incomplete description, and not a definition; not every sequence defines a face. Can you create a definition of a face?

A good definition should, for example, say that everything outside the drawing of a planar graph is a single face. Does your definition apply to trees? Does it apply to disconnected graphs? (For now it is fine to just define faces for connected graphs. To handle disconnected graphs, you'll need to know that any cycle separates the plane into a bounded inside region and unbounded outside region.)

14. For planar graphs we have the graph and must find the points so that we can draw it with no edges crossing. In some applications we are given the points and consider which graphs can be drawn. With your nine points, grow a graph by adding non-crossing line segments until you cannot add more. You should find that every face inside the box is a triangle—your graph is a *triangulation*. How many faces do you have?[?] (Don't forget the unbounded face.) Do the numbers of edges or faces depend on where you place the five points inside the box?[?].



15. Darken a subset of edges of your triangulation of that form a tree on the nine vertices. Add a dot for each face, and connect dots in faces that are separated by an edge that has not been darkened. What do you see?[?] Notice that if you have v vertices and f faces, the pattern you see would count edges $e = (v - 1) + (f - 1)$ if it holds true.



Characters in first-person shooters, and the terrains that they run around on, are often represented by collections of triangles that represent surfaces in three dimensions. The explorations in this chapter will culminate in a way compress a class of triangulated surfaces losslessly—storing them in few bits.

13.2 Foundational definitions

Let's begin with foundational definitions of three important graph variants: directed, undirected, and bipartite graphs. You will notice that we have enough machinery in sets, tuples, functions, and relations to make these definitions, often in more than one way. This leads to some flexibility in the definitions.

Don't let this flexibility confuse you. If I state a definition in terms of a relation and repeat it in terms of sets of tuples, the purpose is that readers who already understand relations and those who are more comfortable with sets and tuples should both agree that exactly the same object has been defined. (If not, then one of us has made a mistake. Either I need to correct what I've written or a reader needs to refine their understanding.) You should adopt whatever definition is more meaningful or memorable for you, as long as it defines exactly the same object and is not circular. The flexibility is in the definition, not in the concept; we modify a concept only if we agree collectively that the modification is more useful for some purpose.

Suggestion: Draw examples for each definition in a notebook or the margins. Consider the extremes for each definition, such as graphs with no edges, or with as many edges as possible. Count the number of objects defined. These checks help to ensure that we agree on the concepts.

16. The first sentence of this chapter defines a *directed graph* or *digraph*: a pair $G = (V, E)$, where V is a finite* set of *vertices* and $E \subseteq V \times V$ is a set of *edges*. Sometimes I say *node* and *arc* for *vertex* and *edge*, especially when I work with two graphs and need to distinguish between them. An edge (u, u), called a *self-loop*, could appear in a directed graph. The maximum number of edges in a directed graph with n vertices is ; if there are no self-loops the maximum is .

*Infinite graphs exist, but are beyond the scope of this book.

As already mentioned, we often draw pictures of directed graphs. We do so by specifying additional information: a position (and maybe style) for each vertex, and a straight or curved arrow for each edge. This is often called a *drawing* or *embedding* of a graph. Crossings may be present in an embedding, or even necessary in any embedding, but they are not part of the underlying graph. Thus, figure 13.4 shows three drawings of the same digraph.

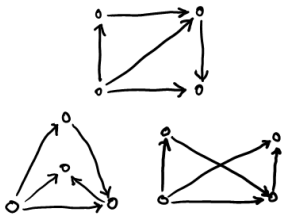


Figure 13.4: 3 drawings, 1 graph

17. Any binary relation $R \subset E \times E$ on a finite set can be drawn as a directed graph with edge set R, and that the properties of the relation show up in the drawing. Complete these statements; the first is done for you.

- A reflexive relation has a self-loop for every vertex;
- An irreflexive relation has .
- A symmetric relation has an edge iff .
- An anti-symmetric relation never has .

- In a transitive relation, any two-edge *path* (u, v, w) with $(u, v) \in R$ and $(v, w) \in R$, .

What about equivalence relations and partial orders? We'll return to those (in 27 and 34) after introducing some more terms.

*When someone says "graph," assume they mean undirected.

An *undirected graph** is a pair (V, E) in which E is an irreflexive, symmetric relation. That is, $E \subseteq V \times V$ in which $\forall u \in V$, $(u, u) \notin E$, and $\forall u, v \in V$, if $(u, v) \in E$ then $(v, u) \in E$. (If we drop the 'irreflexive' condition, we get a variant that allows self-loops.) We draw edges of an undirected graph with no arrows, since joining u and v represents both (u, v) and (v, u).

An equivalent definition could say that E consists of unordered, two-element sets of vertices, rather than ordered pairs, but people tend not to do that. I suspect that we become used to seeing an ordered pair, (u, v), as an edge, so seeing a set $\{u, v\}$ might cause confusion.

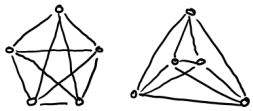


Figure 13.5: Two drawings of complete graph K_5

18. The undirected graph with $n = |V|$ vertices that has all possible edges, $E = \{(u, v) \mid u \neq v \in V\}$, is the *complete graph*, denoted K_n. It is considered to have edges, one for each unordered pair from V. (The *complete digraph* on n vertices has edges, as noted in 16.)

19. Another variation: a graph $G = (V, E)$ is *bipartite* iff the vertices V can be partitioned into two sets U and W, (that is, $U \cup W = V$ and $U \cap W = \emptyset$), so that $E \subseteq U \times W$. A bipartite graph $G = (U, W, E)$ can be considered undirected or directed from U to W. If $m = |U|$, $n = |W|$, then the *complete bipartite graph* $K_{m,n}$ includes edges, namely .

Figure 13.6: Complete bipartite graph $K_{2,3}$

20. How could we change the definition to allow bipartite graph edges to be directed arbitrarily? There is a choice to be made here: Do we allow an edge to appear in either or both directions, as long as we have no edges from $U \times U$ or $W \times W$? Or do we allow only a single copy of each undirected edge, directed either toward U or W? The definitions are different.

21. Two vertices are said to be *adjacent* if they are joined by an edge; a vertex is said to be *incident on* its edges. The *neighbors* of a vertex v are the vertices joined by edges starting from v: $N(v) = \{w \mid (v, w) \in E\}$. The *degree of a vertex* is the number of its neighbors: $d(v) = |N(v)|$. What is the

degree of every vertex in K_n? ? For a variant of undirected graphs that allow self-loops, graph theorists modify the definition to count self-loops a second time: $d(v) = |N(v)| + [(v,v) \in E]$.[†]

[†] Recall Iverson notation: $[(v,v) \in E]$ is 1 if edge (v,v) is in E, and 0 if not.

22. For a directed graph, the *in-degree* of vertex v is the number of incoming edges, $d_{\text{in}}(v) = |\{u \mid (u,v) \in E\}|$, and the *out-degree* is the number of outgoing edges, $d_{\text{out}}(v) = |\{w \mid (v,w) \in E\}|$. Argue that total in-degree equals total out-degree.

23. For undirected graphs we can prove that the sum of all degrees is even, which also means that the number of vertices of odd degree is even.

Lemma 13.2.1. *In any undirected graph, the number of odd degree vertices is even.*

Proof. If we add up all the degrees, then we count every edge twice. Thus,

$$\sum_{\{v \in V \mid d(v) \text{ is odd}\}} d(v) = 2e - \sum_{\{v \in V \mid d(v) \text{ is even}\}} d(v),$$

and since the right side is even, the left side must have an even number of terms in the sum. QED

This is the type of proof sketch that a mathematician will accept, knowing that if asked, they could fill in the details. However, as a character-building exercise, let's fill in all the details. If you already have enough character, you may skip ahead to 24.

Recall that a number n is *even* iff it can be written as twice an integer: $\exists_{k \in \mathbb{Z}}\ n = 2k$. A number n is *odd* iff $n+1$ is even, that is $\exists_{k \in \mathbb{Z}}\ n+1 = 2k$, which means $\exists_{k \in \mathbb{Z}}\ n = 2k-1$. Here are two facts that we can prove. If you believe the facts, you can skip over the proofs. If you don't believe these proofs, try your hand at writing proofs by induction.

Lemma 13.2.2. *Any sum of even numbers is even.*

Proof. Let $n_1, n_2, \ldots, n_k$ be even numbers. I want to prove that $S = \sum_{1 \le i \le k} n_i$ is even.

By definition of even, there exist integers $m_1, m_2, \ldots, m_k$ such that for all $1 \le i \le k$, we have $n_i = 2m_i$. We expand S to see that it is twice an integer:

$$S = \sum_{1 \le i \le k} n_i = \sum_{1 \le i \le k} 2m_i = 2\Big(\sum_{1 \le i \le k} m_i\Big).$$

QED

Lemma 13.2.3. *A sum of odd numbers is even iff the number of odd numbers is even.*

Proof. Let $n_1, n_2, \ldots, n_k$ be odd numbers. I want to prove that $S = \sum_{1 \le i \le k} n_i$ is even iff k is even. By definition of odd, there exist integers $m_1, m_2, \ldots, m_k$ such that for all $1 \le i \le k$, we have $n_i = 2m_i - 1$. We can try to write S as twice an integer:

$$S = \sum_{1 \le i \le k} n_i = \sum_{1 \le i \le k} 2m_i - 1 = \Big(\sum_{1 \le i \le k} 2m_i\Big) - k = 2\Big(\sum_{1 \le i \le k} m_i\Big) + (-k).$$

Using the previous lemma we may conclude that S is even iff k is even. QED

Now, here is a more detailed proof of **Lemma 13.2.1**, that the number of odd degree vertices is even.

Proof. First, observe that, in an undirected graph with e edges no self-loops, the set size $|E| = 2e$, since each edge contributes two pairs to E: $(v, w) \in E$ iff $(w, v) \in E$. The sum of all degrees is even, because it counts the two pairs for each edge:

$$\begin{aligned}\sum_{v \in V} d(v) &= \sum_{v \in V} |\{(v, w) \in E\}| = \left|\bigcup_{v \in V} \{(v, w) \in E\}\right| \\ &= |\{(v, w) \mid v \in V \ \forall (v, w) \in E\}| \\ &= |E| = 2e.\end{aligned}$$

But we can split that sum into the terms with even and with odd indegrees,

$$\sum_{v \in V} d(v) = \sum_{\{v \in V | d(v) \text{ is odd}\}} d(v) + \sum_{\{v \in V | d(v) \text{ is even}\}} d(v) = 2e,$$

which we can rewrite with odds on the right side and evens on the left:

$$\sum_{\forall v \in V \ d(v) \text{ is odd}} d(v) = 2e + \sum_{\forall v \in V \ d(v) \text{ is even}} -d(v).$$

From Lemmas 13.2.2 and 13.2.3, the number of odd degree vertices is even. ☙

24. Computer scientists may prefer a different proof. The combinatorial proof above considers a graph as it is; one could instead create the graph: start with the set of vertices, then add edges one by one. Show by induction that an invariant of this creation process is that the number of odd-degree edges is even. (It is crucially important to observe that every graph can be created. This is trivial for adding edges, but may not be for other processes, or other graph types, such as connected graphs to come in 42.)

25. Suppose that we convert a directed graph $G = (V, E)$ into an undirected graph G' by dropping self-loops and adding the reverse of every edge. Write an expression for the number of edges in G' based on the sets V,E that make up G:

13.3 Modify, count, draw, and color graphs

Questions in this section skim the surface of the many things that can be done with graphs. Later sections will go deeper into topics needed for particular applications.

13.3.1 Modifying graphs

26. Here are two ways to make new graphs from old: A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. Graph G' is the *vertex-induced subgraph* iff $E' = E \cap (V' \times V')$; that is, we keep all edges having both ends in V'. Give an example of a graph G and a subgraph of G that is not a vertex-induced subgraph of G.

27. A *clique* in a graph or digraph is a vertex-induced subgraph that is complete—it contains all possible edges. Argue that any equivalence relation, considered as a graph, can be partitioned into *cliques*, one for each equivalence class.

28. The *complement of a graph* $G = (V, E)$ is the graph $\overline{G} = (V, (V \times V) \setminus E)$, which is the graph with an edge in $\overline{G}$ iff that edge is missing from G. What is the complement of the bipartite graph $K_{4,5}$?

29. One more graph modification: For a graph $G = (V, E)$, the operation of *contraction of an edge* $(u, v) \in E$, denoted $G \setminus (u, v)$, constructs a new graph $G' = (V' = V \setminus \{v\}, E')$

where E' eliminates (u, v) and replaces v by u in all other edges. Can you come up with a good way to define the edge set E' using set notation?

A graph $G' = (V', E')$ is a *minor* of $G = (V, E)$ if it can be obtained by contracting and omitting edges. Minors can reveal simple structure in large graphs, as we shall see.

13.3.2 Counting graphs

30. From the count of edges in 18, many undirected graphs are there on n vertices? How many directed graphs? How many bipartite (undirected) graphs when the vertex set is partitioned into sets of size n and m?

For these counts, we consider the vertex sets to be labeled, and consider graphs to be the same if the edges, which are pairs of vertex labels, are exactly the same; e.g., the two graphs of figure 13.7 are different.

How many labeled, vertex-induced subgraphs does $G = (V, E)$ have? One for each subset of V, so .

How many labeled subgraphs of G have all the vertices V? One for each subset of E, so . There may be a good way to count the total number of labeled subgraphs of G, but I doubt it, because subsets V and E can't be chosen separately: you can only delete a vertex if you delete all its edges, too.

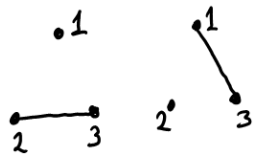


Figure 13.7: Two labeled graphs that are isomorphic without labels

31. A different, and much harder question, is how many graphs are there if you consider two graphs the same if it is possible to relabel the vertices of one so that the edge sets become identical. Relabeling defines an equivalence relation, *graph isomorphism* (from Greek *iso*=equal, and *morphē*=shape), which we know partitions the set of all graphs into equivalence classes.

Hint:

The difficulty in counting is that not all equivalence classes have the same size. Draw an example *unlabeled graph* from each isomorphism class of (undirected) graphs on 4 vertices. But first read the next question, because it can cut your work almost in half.

32. Prove that a relabeling makes two graphs isomorphic iff it makes their complements isomorphic. (You'll want to use the definition of complement from 28 and to write out the meaning of isomorphism in symbols.) One consequence is that it is enough to determine the isomorphism classes for graphs of up to $\binom{n}{2}/2$ edges because those with more edges will be the complement of those with fewer. (Can you ever have an odd number of classes? ?)

33. How many labeled graphs fall into each class of question 31? Note that the total number of labeled graphs with 4 vertices and i edges is

$$\binom{\binom{4}{2}}{i} = \binom{6}{i}$$

.

Hidden here are the numbers of ways to label graphs that have two or three edges:

- There are two classes for two edges:
- There are three classes for three edges:

13.3.3 Drawing partial order graphs

34. Recall that a relation is a partial order iff it is anti-symmetric, transitive, and either reflexive or irreflexive. Prove that the graph of any partial order with a finite number of elements has a *downward drawing*: you can place the vertices so that all edges go from higher to lower y coordinates. (This statement says nothing about possible edge crossings because they are assumed not to affect the graph.)

Hint:

35. The *Hasse diagram* of a partial order is a graph in which we omit any edge (u, v) that can be inferred because there is a path from u to v. That is, we draw edges of a relation whose transitive closure is the partial order. For example, the Hasse diagram for the *poset** $[1..12]$ under $\leq$ is just a vertical path from 12 down to 1. Draw the Hasse diagram for $[1..12]$ under the 'divides' relation ($|$).

*Recall that a poset is a set with a partial order relation

36. The Hasse diagram simplifies downward drawings (34) by removing edges from almost all graphs of partial orders. What family of graphs is not simplified? Hint:

Figure 13.8: Hasse diagram for divides

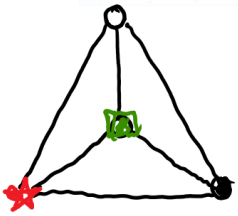
13.3.4 Coloring

It is a challenging puzzle in general to color the vertices of an arbitrary graph with the minimum number of colors so that no edge joins two vertices of the same color. For special types of graphs it is possible to determine the maximum, minimum, or even the exact number of colors. (Unfortunately,

the map four-color theorem takes more machinery than I want to introduce in this book, but we revisit this in section 13.6.)

37. Show that the number of colors to color any subgraph of a graph is at most the number of colors to color the graph. Give an example to show that contracting an edge of a graph may increase the number of colors required. Define some notation to be able to state claims precisely and concisely.

38. Show that any graph with maximum vertex degree d can be colored with at most $d + 1$ colors.



39. Show that the complete graph K_n requires n colors. (You can use 38 for the upper bound.)

40. Show that a graph is bipartite iff its vertices can be colored with two colors. (Exercise 9.8 nicely illustrates the power of two-coloring.)

13.4 Paths and cycles

A *path* in a graph is a sequence of vertices $P = (v_0, v_1, \ldots, v_k)$ with each adjacent pair, $\forall 0 \leq i < k$, joined by an edge $(v_i, v_{i+1}) \in E$. The *length* of a path P is its number of edges, k. A path is *simple* if it visits no vertex more than once. (Notice that these definitions allow repeated edges on a path, but not on a simple path.)

A *cycle* is a path that starts and ends at the same vertex and has no repeated edge. The easiest way to talk about a k-edge cycle is to let the vertices be $(v_0, v_1, \ldots, v_{k-1})$ and index mod k, so $v_k = v_0$. A *simple cycle* has no repeated vertex.

41. How would we say mathematically that path P never repeats an edge?

42. An undirected graph is *connected* if there exists a path from every vertex to every other. Let's write this out with all the quantifiers in front. A graph $G = (V, E)$ is connected iff, for all pairs $u, w \in V$, there exists a non-negative integer k, and a sequence $v_0, v_1, \ldots, v_k \in V^{k+1}$ with $v_0 = u$, $v_k = w$, and, for all $0 \leq i < k$, $(v_i, v_{i+1}) \in E$. What is the negation?

(BTW, a directed graph is called *strongly connected* if there is a path from every vertex to every other.)

43. The *connected components* of a graph G are the *maximal connected subgraphs* of G. Recall from Q 26 that a subgraph takes a subset of vertices and edges to make a new graph G', which is connected iff there is a path between any pair of its vertices, and is maximal if adding any more vertices and edges from G will make it disconnected. For example, we can think of the star of David ✡ as a graph with six vertices and six edge that has two triangles as connected components. If we added six vertices at the crossings, then there would be 18 edges in a single connected component. For an undirected graph $G = (V, E)$, write a recursive definition of the set of vertices in the connected component that contains $v \in V$.

44. See if you can prove the following, which we will need in 48.

Lemma 13.4.1. *In a connected graph with vertices of even degree, removing a single edge leaves the graph connected.*

45. What graphs can be drawn without lifting the pen or retracing a line? Such graphs are called *Eulerian*, in honor of L. Euler's study of the classic Königsberg bridges problem.

Consider refining statements that would be true about graphs that can and cannot be drawn.

X: A graph can be drawn if and only if there exists a path that visits every edge exactly once.
Y: A graph with k edges can be drawn iff there exists a path of length k that visits every edge.
$\overline{X}$: A graph cannot be drawn if and only if all paths that visit every edge repeat some edge.
$\overline{Y}$: A graph with k edges cannot be drawn iff all paths of length k miss some edge.

*This type of disambiguation is not just "academic." It an important part of specifying what is to be solved, especially if you are to be under contract to provide a solution.

Let's check this for extreme graphs: what if we have no edges? Will we say such a graph can be drawn, or do we want to visit all vertices, too?* Let's insist that we want to visit every vertex, too, so we'll want to add a condition that *our graph is connected*.

We've seen in 41 how to say that no edge of a path repeats. How would we say mathematically that a path $P = (v_0, v_1, \ldots, v_k)$ visits all edges of a graph $G = (V, E)$? We can create the set of edges $E_P = \{(v_i, v_{i+1}), (v_{i+1}, v_i) \mid 0 \leq i < k\}$, and check that $E_P = E$.

Y_2: A graph $G = (V, E)$ with k edges can be drawn if and only if G is connected and $\exists$ a length-k path, $P = (v_0, v_1, \ldots, v_k)$, whose edge set $E_P = E$.

46. Can you think of a more elegant way to say that no edge repeats? Rather than just making the edges of P, we could also make its graph $G_P = (V_P, E_P)$, where $V_P = \{v_i \mid 0 \leq i \leq k\}$ and E_P is as defined above. Argue that the following is equivalent to Y:

Y_3: A graph $G = (V, E)$ with k edges can be drawn iff there is a length-k path P whose graph $G_P = G$.

47. Now start with a path P, and look at the degrees of vertices of its graph G_P. It is pretty clear that we need all vertex degrees to be even, except for paths where the start and end are different and have odd degrees. Recall that "clearly" in mathematics means that a writer is getting lazy. Explain why this should be clear.

48. What may not be clear is that this condition is sufficient: in a connected graph, if all degrees are even, except possibly for two vertices that will become the start and end, then there is a path that visits every edge once.

Lemma 13.4.2. *Every connected graph with at most two vertices of odd degree has an Eulerian path. If there are odd degree vertices, they must be the start and end of the path, otherwise the path can be taken to start and end at any vertex.*

Here are two proofs: which do you prefer?

Proof. In this proof, we assume that we have a connected graph G with two odd degree vertices, v_1 and v_2. If initially all vertices have even degree, then either we have the graph with a single vertex, which satisfies the lemma with a path of length zero, or we can remove any edge to form two vertices of odd degree. By Lemma 13.4.1 this graph remains connected,* and if we can find an Eulerian path joining this pair, we can close it to a cycle by adding back the edge.

*This condition is easy to overlook, but important for correctness. Fortunately we did it in 44.

So, it is enough to prove that G has an Eulerian path that starts at v_1 and ends at v_2, We prove this by induction on the number of edges, n.

Base case $n = 1$: The single edge (v_1, v_2) path satisfies the lemma. ✓ (Initially I was going to use all simple paths and cycles as base cases, but discovered that I could reduce to a single edge.)

In the induction step, consider a connected graph G with $n > 1$ edges and two odd degree vertices, v_1 and v_2. We want to find an Eulerian path between them.

IH: We may assume that any graph with $0 \leq k < n$ edges that is connected and has two odd degree vertices has an Eulerian path between them.

If G has a vertex of degree 1, say v_1, then remove v_1 and its incident edge (v_1, w) to make a smaller graph G'. Note that $w \neq v_2$, because the single edge case was covered in the base case, so G' has odd degree vertices w and v_2. It is also connected, because we can find a path between any pair of its vertices by finding a path in G and replacing any instances of w, v_1, w with w to avoid using the removed edge. (Note that v_1 is not in G' so it won't be used as the start or end of the path.) Therefore, the induction hypothesis gives an Eulerian path from w to v_2; we prepend v_1 to get an Eulerian path for G.

Otherwise v_1 has degree > 2. Find any simple path Q from v_1 to v_2 and remove an edge (v_1, w) so that $w \neq v_2$ and (v_1, w) is not the first edge of Q. This forms a smaller G' that again has odd degree vertices w and v_2, and is again connected—this time because any path between a pair of vertices in G can be transformed to a path in G' by replacing the edge (v_1, w) or (w, v_1) by Q or its reverse. Again, the IH gives an Eulerian path for G' from w to v_2; we prepend v_1 to get an Eulerian path for G.

This completes the proof. ∎

We can instead focus on loops of edges.

Proof. We will assume that G is connected, and that all vertex degrees are even. If we begin with two odd degree vertices, connect both to a new vertex so all degrees are even. After we find an Eulerian cycle, remove the new vertex and edges to break it into an Eulerian path.

We prove that any connected graph with only even-degree vertices, G, has an Eulerian cycle by induction on its number of edges, n.

Base case $n = 0$; The graph with one vertex and no edge has a one-vertex Eulerian cycle. ✓

In the induction step, consider a connected graph G with $n \geq 1$ edges and even degree vertices.

IH: We may assume that any graph with $0 \leq k < n$ edges that is connected and has even degree vertices has an Eulerian cycle.

Find any cycle C in G by removing one edge (u, v), finding a path joining u to v (which exists in the remaining graph by Lemma 13.4.1), then closing it with edge (u, v). Now, remove the edges of cycle C from the graph G, and gather the connected components. Each has even degree, so the IH says each has an Eulerian cycle. Now, as you walk around C, the first vertex you hit of a component, take a detour to visit its Eulerian cycle before continuing. Since G was connected, and components of $G - C$ are maximally connected—they need edges of C to grow into a single connected component—this walk will make an Eulerian cycle for the entire graph G.

This completes the proof. ∎

49. The task of finding a *Hamiltonian path*, which visits every vertex exactly once and need not visit all the edges, has no known simple criterion like this one—it is in the family of NP-hard problems, which means that you can, for any graph, efficiently verify that a given path is Hamiltonian, but that no algorithm is known to efficiently find a path or prove that none exists.

50. What can be said about directed graphs? When does a directed graph have a path that visits every edge?

13.5 Trees

Trees are special graphs that are ubiquitous in computer science, in many variations (undirected, directed, rooted, balanced, . . .)

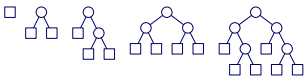
51. Here is one way to define the simplest variant: An undirected graph is a *forest* iff it has no cycles (it is *acyclic*). An undirected graph is a *tree* iff it is a connected forest. Draw three undirected examples: a tree, a forest that is not a tree, and a graph that is neither. For what values of n is the complete graph K_n a tree? ?

52. A *spanning tree* of a undirected graph G is a subgraph that includes all vertices of G and is a tree. Draw a spanning tree for your graph in Q 51.

13.5.1 Equivalent definitions for rooted trees

In the introduction, questions Q 6–Q 8 suggested three different possible definitions of *rooted trees*: by graph properties, top-down recursive, and leaf-based recursive. These will define three sets graphs, S, T, and L, which we will prove are, in fact, the same; each of the three definitions gives the same notion of rooted trees. Because previous experience with trees may make some readers too ready to believe this, this section steps away from the question/hidden answer format used in the rest of this chapter to make it easier to compare the details of the different definitions and proofs. You are still encouraged to try writing your own proofs before reading the ones written here.

The first definition: let S be the set of connected, acyclic graphs that have a vertex chosen as *root*. This definition means that a single vertex can be a rooted tree, but the graph with no vertices is not a rooted tree. This contrasts with the recursive definition of binary trees from subsection 8.1.6. In computer science, every binary tree node has two pointers to children, even if they are set to null.



The second definition, which generalizes the recursive definition of binary trees to the set of all *rooted trees* T, changes the base case to the single-vertex tree.

1. Base: The single-vertex tree $\bigcirc \in T$, with this vertex as the root.
2. Recursive Rule: For any two rooted trees $s, t \in T$, form a new tree whose root is the root of t by making the root of s be a child of the root of t. This tree is in T.
3. Closure: Only graphs generated from the base by a finite number of applications of the recursive rule are in T.

The third definition is also recursive, but builds trees by adding leaves instead of combining at the root. Note how the root never changes for this definition of a set of rooted trees, L.

1. Base: The single-vertex tree $\bigcirc \in L$, with this vertex as the root.

2. Recursive Rule: For any tree $t \in L$ and any vertex $v \in t$, form a new tree by adding as a child of v a new vertex $\bigcirc$ that has no children itself. This tree is in L.
3. Closure: Only graphs generated from 1 by a finite number of applications of rule 2 are in L.

Let us prove formally that $S = T = L$; that all three definitions define the same set of rooted trees. I will prove first that $S = T$ by showing that every T-tree is connected and acyclic, then that every S-tree can be made from T-trees.

Proving $T \subseteq S$: I want to show that if a graph G is a T-tree, then G is an S-tree. Since if G is not a T-tree this "if" statement is (vacuously) true, I can start my proof by assuming that G is a T-tree. I want to show that G is an S-tree, which means that I want to show that G is connected (there exists a path between any two vertices) and has no cycles (all paths of length > 0 that leaves a vertex and returns must repeat some edge.)
Notes: I chose to do my induction on the number of vertices, but could use the number of edges, or on the number of times the recursive rule is used to make G. My step S7 has to work for any given G, which is why it has a "take G apart and reassemble" feel. For this first proof, I try to err on the side of too much detail rather than too little, just to make sure I've got things right.

1,2. We show that all T-trees G are connected and acyclic, by induction on the number of vertices n.
3. Base $n = 0$: the empty graph has no vertices or cycles at all, so satisfies the definitions trivially.
   Base $n = 1$: the single vertex graph has the 0-edge path, and no paths of length > 0, so it also satisfies the definitions.
4. Ind. step: Someone gives me a T-tree G with $n > 1$ vertices;
5. IH: I may assume that, for any T-tree H with $0 \leq k < n$ vertices, H is connected and acyclic.
6. I want to show that G is connected and acyclic.
7. Since G has $n > 1$ vertices, it must have been assembled by joining the roots of two T-trees, G_1 and G_2, by an edge e. Since each of G_1 and G_2 have fewer than n vertices, the IH applies (twice) to show that each are connected and acyclic.
   Now, to see that G is connected, we show there exists a path between any two vertices u and v. Since G_1 and G_2 are connected, if u and v are together in one of these trees, we know there is a path between them. Otherwise, we can assume, by renaming if necessary, that $u \in G_1$ has a path to the root of G_1 and $v \in G_2$ has a path to the root of G_2, and we can connect these roots by edge e to make a path from u to v.

Finally, to see that G is acyclic, we consider any path of length > 0 that starts and ends at the same vertex. If the path stays completely in G_1 or G_2, we know it must reuse an edge, because G_1 and G_2 are acyclic. If it uses vertices of both, then it must cross edge e at least twice, because that is the only connection between G_1 or G_2. Therefore, G is acyclic. ☙

Proving $S \subseteq T$: I want to show that if a graph G is a S-tree, then G is an T-tree. I can start by assuming that G is a S-tree. I want to show that G is an T-tree, which means that it is either a base case or can be made by joining roots of two T-trees by a new edge.
Notes: Notice carefully which things I know and which I don't when I start taking G apart. I always know that G is connected and acyclic, because that is given to me. I don't initially know that its fragments are. I even have to prove that there exists an edge e from the root to delete (I choose to delete such an edge because I know that is how I want to put G back together), I have to show that deleting e leaves two connected pieces, that these pieces are also acyclic, and have fewer vertices than G, and only then do I know that I am able to apply my Ind. Hypothesis. Once I do, I am done; this is in contrast to the previous proof where I could use my IH right away, and most of the work came after. (That happens with recursive definitions.)

1,2. We show that all S-trees G are T-trees by induction on the number of vertices n.
3. Base $n = 0, 1$: the empty graph and single vertex graph are both S-trees and base case T-trees. ✓
4. Ind. step: Someone gives me an S-tree G with $n > 1$ vertices, which means that G is connected and acyclic.
5. IH: I may assume that, for any S-tree H with $0 \leq k < n$ vertices, H is also a T-tree.
6. I want to show that G can be built from two T trees by the recursive rule.
7. Since G has $n > 1$ vertices, and is connected, there must be some edge e that connects the root to a child. Delete e.
   I claim that the graph falls into two connected components, G_1 and G_2: The endpoints of e cannot remain connected after we delete e because otherwise G would have a cycle. Furthermore, there cannot be more than two components after we delete e, since adding e back can unite at most two components and G is a connected graph with one component. (BTW, these are all consequences of paths being an equivalence relation, as you prove in Exercise 13.2.)
   I also claim that G_1 and G_2 are acyclic, because any cycle in one of them would also be a cycle in G, and we know that G is acyclic. This means that G_1 and G_2 are also S-trees.

Since each loses at least the root of the other, they also have less than n vertices, and we can apply the IH twice to see that G_1 and G_2 are T-trees. Therefore G is made by joining the roots of two T-trees, G_1 and G_2, and is itself a T-tree. ∎, by induction.

Useful facts about trees: Now that I know that $S = T$, I can prove that every tree with $n > 1$ vertices does have a *leaf*, which is a vertex of degree one that is not the root. I'll sketch the inductions for two ways, knowing that you are able to fill in the detailed steps.

Lemma 13.5.1. *Every S-tree with $n > 0$ vertices has $n - 1$ edges.*

This is chocolate bar splitting

Proof. Since S-trees are exactly the T-trees, we prove this by induction for T-trees on the number of vertices n.

Base $n = 1$: single vertex, no edge ✓

Ind. Step: Given a T-tree G with $n > 1$ vertices, we may assume for all $1 \leq k < n$ that a k-vertex tree has $k - 1$ edges. By removing one edge, G splits into T-trees of n_1 and $n_2 = n - n_1$ vertices. The total edges of G are $1 + (n_1 - 1) + (n_2 - 1) = n - 1$. ∎

Lemma 13.5.2. *Every S-tree with $n > 1$ vertices has a leaf.*

Combinatorial proof. In an S-tree $G = (V, E)$, the sum of all degrees $\sum_{v \in V} = 2|E| = 2|V| - 2$. Since G is connected, no vertex has degree zero, and to have the sum be $2|V| - 2$, there must be at least two vertices of degree 1. Even if one of those is the root, there still remains a leaf of degree 1. ∎

Proof by induction. on the number of vertices as an alternative.

Base $n = 2$: the only two-vertex tree has a leaf.

IH: every tree with $2 \leq k < n$ vertices has a leaf. Any given tree with $n > 2$ vertices was made by joining the roots of two smaller trees, one of which had at least 2 vertices, so had a leaf. Only the roots change degrees during the join process, so the leaf persists. ∎

Proving $S \subseteq L$: As before, assume that G is a S-tree. I want to show that G is an L-tree, which means that it is either a base case or can be made by adding a leaf to an L-tree.

Notes: Let's do this direction first, because it needs Lemma 13.5.2 to take G apart as an S-tree so that we can reassemble it as an L-tree. If we hadn't proved that separately, then we'd need to prove it within this induction, which would make things messier.

1,2. We show that all S-trees G are L-trees by induction on the number of vertices n.

3. Base $n = 0, 1$: the empty graph and single vertex graph are both S-trees and base case L-trees. ✓
4. Ind. step: Someone gives me an S-tree G with $n > 1$ vertices, which means that G is connected and acyclic.
5. IH: I may assume that, for any S-tree H with $0 \leq k < n$ vertices, H is also a L-tree.
6. I want to show that G can be built from an L tree by the recursive rule.
7. Since G is an S-tree with $n > 1$ vertices, it has a leaf v with parent u; remove v and edge (u, v) to make graph G'.
   I claim that G' is an S-tree with $n - 1$ vertices: First, it is connected because any two vertices $s, t \in G'$ could be connected by a path in G, and if that path visited v it would do so by $\ldots uvu \ldots$, which we can replace by u to get a path in G'. Second, it is acyclic because G was acyclic, and removing an edge and vertex cannot make more cycles. Third, it lost vertex v.
   Thus, the IH applies to G', showing that it is an L-tree, and we can replace v and (u, v) to show that G is also an L-tree.
8. ☠

Proving $L \subseteq S$: I want to show that if a graph G is a L-tree, then G is an S-tree. As before, assume that G is an L-tree, and show that G is connected (there exists a path between any two vertices) and has no cycles (all paths of length > 0 that leaves a vertex and returns must repeat some edge.)
Notes: This direction is easier. We could even use weak induction because we need only the $k = n - 1$ case of the IH.

1,2. We show that all L-trees G are connected and acyclic, by induction on the number of vertices n.
3. Base $n = 0, 1$: the base case L-trees, the empty graph and single vertex graph, are both S-trees. ✓
4. Ind. step: Someone gives me an L-tree G with $n > 1$ vertices;
5. IH: I may assume that, for any L-tree H with $0 \leq k < n$ vertices, H is connected and acyclic.
6. I want to show that G is connected and acyclic.
7. G must have been made by adding a vertex v and edge (u, v) to an $(n-1)$-vertex L-tree, G'. The IH tells me that G' is connected and acyclic. But then G must be connected, because we can connect v to any other vertex by first connecting to u and then taking a path in G', and all other pairs are connected in G'. Furthermore, G is acyclic, because no cycle of length > 1 can touch v without using (u, v) twice, and cycles that don't touch v would be cycles in G', but there are none.
8. ☠, by induction.

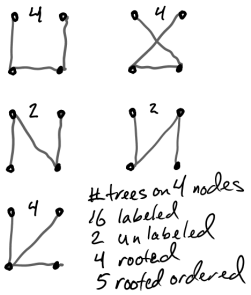
We conclude with this characterization of trees from Q 9.

Theorem 13.5.3 (2 of 3⇒tree)**.** *Any graph G with n vertices that has two of these three properties is a tree, and so has the third property also:*

1. *Graph G is connected.*
2. *Graph G is acyclic.*
3. *Graph G has $n-1$ edges.*

Proof. If we have 1 and 2, we have an S tree, and Lemma 13.5.1 gives the third. 1 and 3 were enough to prove that there is a leaf (Lemma 13.5.2) that can be plucked while leaving the graph connected. Thus, by induction, any connected graph with $n-1$ edges can be built as an L-tree. And for any acyclic graph, each connected component is a tree with one fewer edges than vertices, so to total $n-1$ edges, we must have only one connected component. ∎

13.5.2 Counting Trees



53. How many different trees on n vertices are possible?

For this, we must decide what we mean by different. If we think of vertices drawn as points in the plane, or labeled with distinct ids, we could mean that trees are the same if they have the same set of edges. So let's count *labeled trees* on the vertex set $[1..n]$.

Here is a way to disassemble a labeled tree and form an $(n-2)$-tuple of integers $(a_i) \in [1..n]^{n-2}$ known as the *Prüfer code*: for steps $i = 1$ to $n-2$, remove the smallest leaf, and record the number of its neighbor as a_i. It is clear that this gives a sequence for every tree. In fact it is invertible: determine how to return to the tree from the sequence.

54. Because there is a bijection from labeled trees to Prüfer codes, the number of labeled trees on $[1..n]$ is ____.

55. But if we think of vertices as unlabeled and indistinguishable, we could define an equivalence relation that considers two trees to be the same if there is some way to permute the vertex labels of one so that both have the same set of edges. (This is a tree version of *isomorphism* from 31.) It builds character to write down this relation, and formally prove it is an equivalence relation. What are the two unlabeled trees on four vertices?

The number of *unlabeled trees* is much harder to count: it grows a little slower than 3^n. If you are interested, you can start with sequence A55 in the On-Line Encyclopedia of Integer Sequences and its references. The number

of *rooted unlabeled trees*, sequence A81 is a little easier, but still has no closed form.

56. There is yet another notion of difference for trees that is natural in data structures: the vertices are unlabeled, but the edges coming out of each vertex u are stored in some order—the neighbors $N(u)$ are stored not in a set, but a (circular) sequence. Two *ordered trees* would be considered the same if we can relabel vertices so that corresponding neighbor sequences become the same. Draw pairs of trees that distinguish between this definition and those of 55 and 53.

57. One nice way to write down a rooted ordered tree uses strings from $\{[,]\}^*$ that have balanced brackets. For the base case, represent a single node tree by the string [ ]. For the recursive rule, if S and T are strings of balanced brackets that represent rooted trees, then $cons(S, T)$ makes the root of S a child of the root of T. E.g. $cons([\,], [[\,]]) = [[\,][\,]]$ makes a single node a child of a two-node tree, to produce a binary tree on three nodes.

We can omit the outermost pair of brackets, so that an ordered tree with $m + 1$ nodes becomes a string with m pairs of brackets, *open* [and *close*], in which no prefix has more close] than open [. We can count these by choose and the sum rule. What five strings represent the five rooted ordered trees on 4 vertices?

58. It is easy to count the number of strings of $2m$ brackets with m open: just choose locations for the opens: . This set contains unbalanced strings that have a prefix with more closes than opens. To count these, create a bijection between the set of unbalanced strings having m [and m], and the set of all strings with $m - 1$ [and $m + 1$]: Run your finger along an unbalanced string until you pass the first position that has more close] than open [, then swap braces [⇔] for the rest of the string. Argue that this produces a string with $m + 1$ close] and $m - 1$ open [braces, and that this is invertible–for every string of $2m$ with $m + 1$ close], you can make a unique unbalanced string of m open and close that violates the prefix property.

59. This gives the count of binary trees of $m + 1$ nodes as the mth *Catalan number*:

.

60. The *full binary trees*, in which every non-leaf node has two children, a left and a right, are the most useful subset of the ordered trees of 56. They can also be represented as balanced strings of braces. Here we represent a leaf by an empty string, and a non-leaf node with child strings L and R by $[L]R$. Now with n non-leaf nodes we get a string of n pairs of balanced braces, also counted by the nth Catalan number.

The fact that ordered trees and full binary trees each have a bijection to balanced strings of braces means they have a bijection to each other. Describe this bijection, and its implication for storing ordered trees in a binary tree data structure.

61. The most common way to store a tree would use pointers for each edge. Since each pointer must be at least $\lg n$ bits long to select one of n nodes, the total memory for pointers is at least $n \log n$ bits. The connection to balanced braces shows that $2n$ bits suffice to store a tree structure, a significant savings. You can think of reading a string of braces as a recipe to traverse and build a tree: for each open [, you create a new edge and go down it to a new node, and for each close] you go back up the edge you on which you came down to you current vertex. We will use this idea to compress planar graphs in 78.

13.6 Planar Graphs and Triangulations

62. Recall that in 14 you created a *triangulation* on your 9 points by adding as many edges connecting points as you could without two edges crossing. How many edges could you add to your set of 9 points? Is there more than one way to add edges? Do you always get the same number of edges? The same number of triangles?

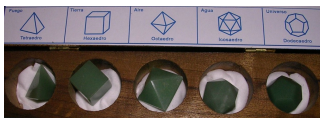
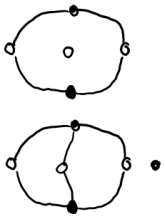


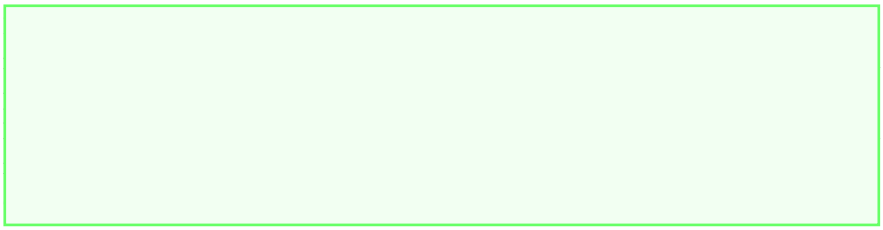
Figure 13.9: Platonic solids. (photo: Yvette Soler)

63. Your drawing should satisfy Euler's relation on the number of vertices, edges, and faces of a planar graph: $f - e + v = 2$, if you remember to count the face outside the rectangle. This is like putting the graph on a sphere, so check that Euler's relation also holds for objects like the Platonic solids: the regular tetrahedron, hexahedron (aka cube), octahedron, dodecahedron, and icosahedron.

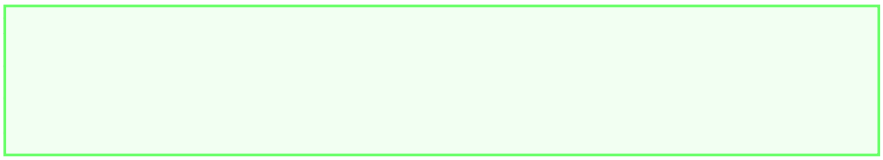
64. One important property of drawings in the plane: any cycle separates the plane into one bounded region (or more if the cycle is allowed to intersect itself) and one unbounded region. (We get the same on a sphere if we arbitrarily distinguish some region as "unbounded," perhaps by adding a "point at infinity.") This is one form of the *Jordan curve theorem*, which is usually stated in continuous term that rule out wild behavior such as infinite changes of direction. (Since cycles in a graph are finite, all we need is that edges aren't wild, and sraight lines or arcs certainly qualify.)



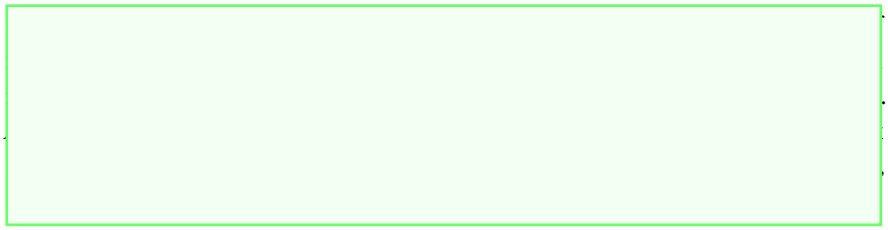
A consequence of the Jordan curve theorem is that any drawing of $K_{3,3}$ in the plane must have a crossing. Start with a 4-cycle $K_{2,2}$ on a sphere, then add each remaining vertices and their edges. Explain why this must happen as depicted in the margin if we try to avoid crossings.

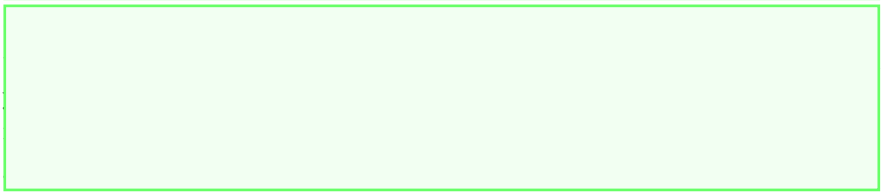


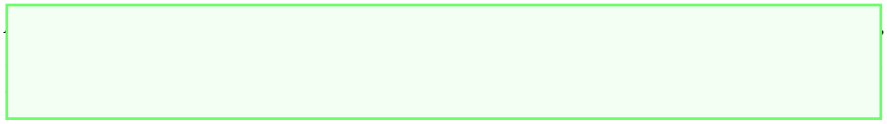
65. We can prove Euler's relation in the form $e = (f-1)+(v-1)$ by partitioning the edges into two spanning trees. First, make one for vertices: For every vertex p, try to choose an edge by the following rule: if an edge goes straight down from p, choose it, otherwise choose the lowest edge that goes to the right from p. Argue that this chooses $v-1$ edges, and that they form a connected graph. Thus, by 9, they form a tree spanning all vertices.



66. Argue that the unchosen edges form a spanning tree of the faces. Here we don't initially know their number, so must argue that they are connected and acyclic. It helps in drawing pictures on your scratch paper to draw the duals of the unchosen edges as arcs connecting nodes in the faces incident on the unchosen edges.

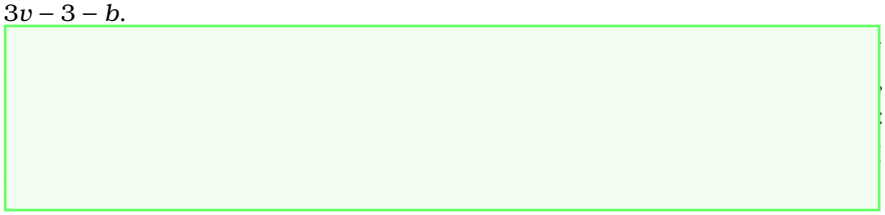




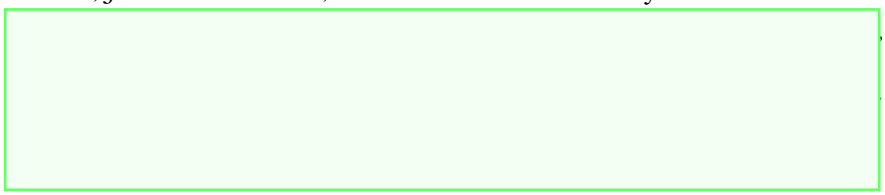


67. We can determine how the number of triangles and edges is related to the number of vertices by combining Euler's relation with the fact that summing face degrees counts edges twice. This is important to know when allocating space to store meshes for video game scenes or characters. Argue that if b is the number of points on the boundary of a triangulation, then the number

of triangles inside the hull is exactly $2v - 2 - b$, and number of edges is $3v - 3 - b$.



68. The Platonic solids are those in which every vertex has the same degree d_v and every face has the same degree d_f. Now that we know that Euler's relation, $f - e + v = 2$ holds, show that there can be only five Platonic solids.



69. One of the great things about learning new mathematical tricks and tools in computer science is that you get to use them in unexpected applications. Suppose that you have a massive multiplayer game in which players are traversing a terrain represented as a triangular mesh in which each triangle knows the coordinates (x, y, z) of its vertices, and has pointers to the three neighboring triangles. Assume that if you ignore the z-coordinates, you see a triangulation in the plane with no triangle edges crossing.

    Notice that you can determine whether a non-boundary edge would be chosen for a spanning tree of 65 by knowing the other vertex of its two neighboring triangles and the "downward" direction (easiest if this is parallel to the negative y axis, but with a little trigonometry, it can be any direction.) This requires no modification or marking of the triangulation itself.

70. Turn this into a way to visit every triangle by treating the spanning tree as walls of a maze and walking with your left hand on the wall. All you need to remember is how you entered your current triangle.

71. Show that if you number each triangle the first time you visit it, then any line in the downward direction encounters triangles in increasing order. In other words, if I draw triangles when I first encounter them, and am standing at $y = -\infty$, then I'll be drawing in back to front order and will end up seeing only the triangles I am supposed to see. This is called the "painter's algorithm," and multiple players can be using it simultaneously on the same mesh.

72. Euler's relation holds on the plane and sphere because there are no holes or handles. On a donut or *Klein bottle* world the count would be a little different: $f - e + v = 0$. To extend the formula would take me too far afield, so I leave that for your own exploration or another course. But do look at

your proof in 66 and figure out where it would break down on a donut.

73. No-one knows if two point sets in *general position* always have *compatible triangulations*[*]: Given any two point sets $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$, is there always a set of pairs $E \in [1..n]^2$ that defines a triangulation of both sets? That is, $E_A = \{(a_i, a_j) \mid (i,j) \in E\}$ are a maximal non-crossing set of segments for A, and $E_B = \{(b_i, b_j) \mid (i,j) \in E\}$ are a maximal non-crossing set of segments for B. I think the answer is "yes," if the points are in *general position*: no line contains more than two points and if A and B have the same numbers of points on their convex hulls (the shape formed if you snap a rubber band around a set of points). You can make two point sets for which the answer is "no" if you violate one of those two conditions. Having compatible triangulations might give a nice way to morph one set of points onto another.

[*]The Open Problems Project, P38

74. Another open problem is to connect the dots: For any triangulation with e edges and an m extra points, can you find a path that connects all the points and has less than $2e$ crossings with edges? Here the question is not to compute the fewest crossings, but to guarantee that there is always a way to cross few edges. This would be a good order of player positions if you wanted to determine which triangle each player was standing on by having your search algorithm step through the triangles between consecutive players positions.

13.6.1 Drawing and encoding planar graphs

Walter Schnyder [23] showed how to partition the edges of a triangulation in the plane into three directed spanning trees, colored red, green, and blue, that cross in a very controlled way: except for three root vertices on the infinite face, every vertex has three outgoing edges, one red, one green, and one blue in counter-clockwise order. Any incoming edges arrive between the outgoing edges of the other two colors. This strange property has surprising implications for drawing planar graphs on graph paper and compressing them.

75. Show from this property, and the fact that no graph edges cross, that all single color paths end at a root, and that two paths of different colors cross in at most one vertex.

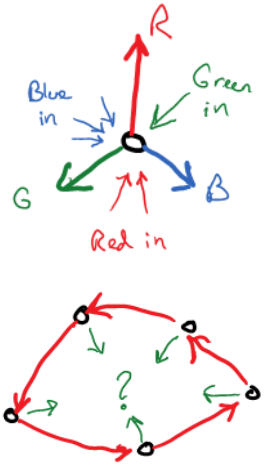


Figure 13.10: Showing that red cycles are impossible

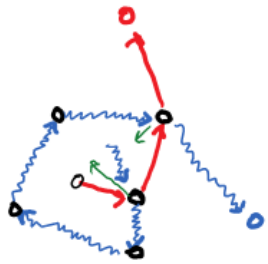


Figure 13.11: Red and blue intersecting twice.

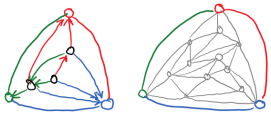


Figure 13.12: Vertices r, g, and b and their spanning trees.

76. Let's see how to find a coloring. First, assume that the graph is fully triangulated, so that every face is a triangle. If necessary, add curved edges outside the convex hull so that the infinite face is a triangle with three vertices: r at top and gb horizontal, as in figure 13.12. These three vertices, the only exceptions to the coloring rules, each have a single outgoing edge: color edge (r, g) green, (g, b) blue, and (b, r) red. Direct all other edges incident on r toward r and color them red. Now, can you figure out a procedure that will color the rest of the edges by the rules? The first box is a hint.

Show by induction that your procedure will succeed.

77. Notice how every vertex, except rgb, lies on a red, a blue, and a green path to the vertices r, g, and b, respectively. These paths define three regions; include each path with the region to its left. For each vertex, count up the number of vertices in the regions on and to the left of the red and the green paths, and make those the coordinates of that vertex. This assigns each

vertex a coordinate on an $(n-2)\times(n-2)$ grid. The graph can now be drawn compactly, with straight lines.

I don't have a concise proof that the drawing has no edge crossings—there are too many cases for how the four vertices of the two edges can lie in their two spanning trees. A more indirect proof may be better, but that is best left to a graph theory course. But perhaps you could work through one case using the observations of 75.

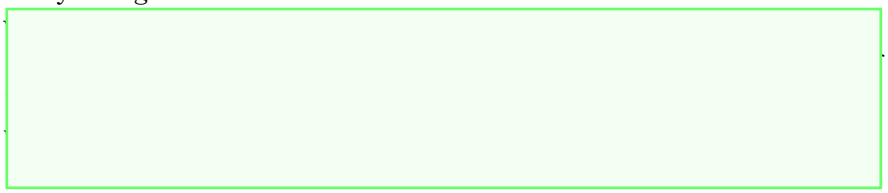
78. We can use the Schnyder trees to count and to compress triangulations. This type of compression is used in video games because most scenes and characters are represented by meshes of triangles that are painted with textures.

The left-to-right (counter-clockwise) way to assign colors in 76 leads to a clockwise way to find a spanning tree of all vertices that uses one edge leaving each vertex. Start at the top of root r, and trim the edges to g and b, leaving stubs behind. Then repeat until you return back to the top of r: continue clockwise around your current vertex v until you meet either an edge (u, v) that has not been trimmed, or the edge (v, w) that brought you to v. In the first case, go "down" edge (u, v) (opposite its direction), make u your current vertex, and trim the other two edges out of u to stubs. In the second case, return "up" edge (v, w) and continue with w as your current vertex.

Prove that this visits every non-root vertex once, so that you have a spanning tree of all vertices but g and b, and each vertex has two stubs.

79. The cool thing is that you can repeat this clockwise traversal to rebuild the triangulation from the spanning tree. Start at r and create g and b for the two stubs. Now, traverse like in 61: each time you go down, make an edge to a new vertex. Each time you hit a stub, create an edge to close the triangle clockwise from the current vertex. Each time you go up, you change your current vertex. Convince yourself that this gives back the original triangulation.

80. The traversal in 61 used two symbols. Here it looks at first as if we need three, for down, up and stubs. However, at each traversed vertex we hit both stubs before we go up, so we can encode down as 1 and both stubs and up as 0, and simply keep track at each vertex how many zeros we've seen so far. How many 0s and 1s suffice, therefore, to encode a triangulation? How many triangulations are there?



13.7 Exercises and Explorations

Quiz Prep 13.1. Draw a graph . . .

1. that has a cycle (highlight it) that is not simple. ?
2. that is two-colorable (and therefore bipartite), and not a tree.
3. that is planar and requires four colors K4 works
4. that has more than one Eulerian cycle.
5. that has no Eulerian path.
6. that has a Hamiltonian path (highlight the path).
7. with no Hamiltonian path.
8. that is connected, but may or may not be strongly connected if each edges is given a direction.
9. that is connected, but cannot be strongly connected if every edge is directed toward one of its vertices.

Exercise 13.2. In an undirected graph $G = (V, E)$, define the relation "is reachable from" on vertices: $\forall a, b \in V$ we say that $a \; R \; b$ iff there exists a path starting at b that ends at a. Prove that R is an equivalence relation: that is, it is reflexive, symmetric, and transitive. What are the equivalence classes of R in graph terminology? ▶

Exercise 13.3. Prove that an undirected graph G is a tree iff it has a unique path between any pair of vertices.

Prove that an undirected graph G is a tree iff adding any edge (u, v) that is not already present forms a single cycle.

Exercise 13.4. In the *Max in a list* algorithm of section 11.2, show that an algorithm that performs fewer than $n - 1$ comparisons will return the wrong maximum on some list. Consider the graph whose vertices are list entries with an edge between two vertices iff their list entries are directly compared. ▶

Puzzle 13.5. Sprouts is a pencil and paper game invented by J. H. Conway and M. S. Paterson. Start with n spots on the paper. Each player in turn draws a line connecting two spots that are the ends of less than three lines without crossing any existing lines or spots. This line sprouts a new spot, which can be used once more. The last player who is able to do this wins. Since new spots are created, it may not be obvious that this game terminates. Show that it does, and figure out the minimum and maximum number of plays as a function of n. (There is a conjecture that the first player can guarantee a win iff $n \bmod 6 \in \{3, 4, 5\}$, and this has been verified by computer for $n < 45$ spots.)

Puzzle 13.6. Dominoes are wooden rectangles with numbers on each end. Under what conditions can an arbitrary collection of dominoes be lined up end to end so that the numbers match where they touch? (I have a double-9 set, and I've seen double-12, but as a puzzle you can think of numbers being $[1..n]$ on each side. You don't get the all the dominoes in a set, though. . .)

Hint:

Puzzle 13.7. Prove the *Futurama theorem* [13], which writer Ken Keeler created for the Season 6 episode, "The Prisoner of Benda:" Professor Farnsworth and Amy have created a mind-swapping machine. After using it on several characters, they discover that no pair of minds can survive a second swap, although two minds can swap back with the help of some fresh bodies.

If n people are in the wrong bodies because of some unknown set of swaps. how many helpers are needed to sort everyone out? (Convince yourself that one extra body is not enough.) Keeler's Futurama theorem says that you never need more than two. Describe how, and prove that your method works.

You'll want to begin by assigning names to bodies and minds – defining notation can help clarify questions. Since bodies are put into the machine, I suggest numbering bodies so that body i wants mind i, but initially contains mind m_i.

Hint:

Figure 13.13: Swapping minds

Exploration 13.8. Many puzzle-based video game include graph puzzles. Two examples:

Minesweeper: On a grid of vertices, the game creates invisible edges from each mine to its eight neighbors. Clicking on a vertex reveals a mine (you lose) or its in-degree. Whenever in-degree is zero, then the computer recursively clicks all eight neighbors for you, too.

Bejeweled: On a grid of colored vertices, create edges between pairs of same-color vertices whose positions differ by one in x or by one in y. Clicking on any vertex removes the entire connected component; larger components earn larger scores. Vertices drop down to fill in, and the game continues.

Notice how these games hide information, or change it dynamically; these are two examples of *information/action handles* that can be added to mathematical puzzles to turn them into games [5]. Other handles include online (e.g., Tetris) or discrete repair (e.g., Planarity).

Identify a graph from a puzzle in a game that you have played, and any information/action handle that was used. Describe whether looking at it as a graph gives you new insights into the solution. Manage your time, though. You may not count more than 5 minutes of replaying the game as "doing homework."

Exploration 13.9. In one classic peg puzzle, you have a line of ten holes that start with 4 blue pegs on the left, two empty holes, then 4 blue pegs on the right. A peg may move into an empty hole that is adjacent, or by jumping over a single peg of the opposite color, but blue pegs must move right and red pegs must move left. The goal is to get the pegs to swap: all 4 red at left and all 4 blue at right.

First, if we were to build a graph to capture all possible arrangements and the movements between them, what would we find?

- How many different arrangements of colored pegs in holes are possible? ?
- How many of these arrangements have a pair of reds blocking a pair of blues? ?
- In any given arrangement, what is the maximum number of possible moves? ?
- How many jumps will have to take place to go from the initial arrangement to the solution, and how many single step moves?

So the graph here would be too big to build by hand, but easily managed by computer. With the graph you could answer many questions: How many arrangements allow no moves? How many of those can you reach from the start position? How many arrangements allow the maximum number of moves? And, of course, is there a solution? ▶

Exploration 13.10. Radia Perlman summarizes her 1985 paper [18] as a poem (a pastiche on Joyce Kilmer's "Trees").

> Algorhyme
> I think that I shall never see

A graph more lovely than a tree
A tree whose crucial property
Is loop-free connectivity.
A tree which must be sure to span
So packets can reach every LAN.
First the Root must be selected
By ID it is elected.
Least cost paths from Root are traced.
In the tree these paths are placed
A mesh is made by folks like me
Then bridges find a spanning tree.

This work is the source of the Spanning Tree Protocol (STP), which routers use to prevent *bridge loops* that mess up network routing tables. Find this paper and read it to create a summary of how the bridges find a subgraph guaranteed both to be a tree and to span.

Exploration 13.11. Listen to Robert Lang's TED talk on how math has affected the art of origami design.

Many origami models fold completely flat before they are shaped into their 3-d forms. If you unfold such a model, you'll see creases for mountain and valley folds (turning the paper over makes mountains into valleys, and valleys into mountains). Lang mentions four mathematical rules for crease patterns of flat origami:

1. Crease patterns are planar graphs whose faces can be colored with two colors.

2. At any vertex, the number of valley and mountain folds always differ by two. [Maekawa's theorem]

3. Around any vertex, the odd numbered angles add up to 180 degrees, as do the even. [Kawasaki's Theorem.]

4. A sheet can never penetrate a fold.

Prove the first three rules. Note that Kawasaki's theorem is trivial if all creases extend through a point p, so create a fold where some creases end at p to see if you still believe it.

Chapter 14
Discrete Probability

If the host is required to open a door all the time and offer you a switch, then you should take the switch. But if he has the choice whether to allow a switch or not, beware. Caveat emptor. It all depends on his mood.
My only advice is, if you can get me to offer you $5,000 not to open the door, take the money and go home.

—Monty Hall (né Monte Halperin) [25]

The origin of the study of discrete probability was an exchange of letters between between Pascal and de Fermat about how to split the pot in an interrupted gambling game by carefully counting the set of possible ways that the game could have continued. From this has developed a number of simple, carefully-chosen definitions that lead to some surprising and powerful conclusions. Probability helps us work with large or uncertain data sets and processes; the mathematics that I illustrate with coin flips and dice rolls directly applies in trendy applications of computing like data mining, risk analysis, quantum physics, network modeling, decision theory, and statistical thermodynamics.

((todo. this chapter is not complete or even ready for reading.))

Objectives: After working through this chapter and its exercises you will be able to use the language of discrete probability to **((todo.** This chapter is only half here. finish this))

probability of events in sample spaces random variables expectation

Averaging smooths out special cases. If we know that the average is large, then we know that some cases must be large.

bookkeeping device, analyze typical rather than worst-case behavior, discover possibilities by studying average.

14.1 Definitions

A *random experiment*, such as flipping a coin, throwing two dice, shuffling a deck of cards, or being dealt a hand of 5 cards, is a repeatable experiment that may not always have the same outcome. Each individual outcome is an *elementary event*, any set of outcomes is an *event*, and the set of all possible outcomes is called the *sample space*.

For example, the sample space of rolling two 6-sided dice, one white and one De Morgan, has 36 elementary events, including snake eyes, ⚀⚀. In the game of Craps, the first roll of two dice can lead to the events of rolling

craps and losing with 2,3, or 12 spots, {⚀⚀, ⚁⚁, ⚂⚂, ⚃⚃}, rolling a *natural* and winning with 7 or 11, {⚀⚃, ⚁⚄, ⚂⚃, ⚃⚄, ⚄⚅, ⚅⚀, ⚃⚃, ⚅⚅}, or rolling any other combination and having to roll again.

For a sample space S, the set of all possible events is the power set 2^S. The subset $\emptyset$ is the *null event*, which can never happen, and whole set S is a *certain event*, which must happen. Two events $A, B \subseteq S$ are called *mutually exclusive* if $A \cap B = \emptyset$; craps and natural are mutually exclusive, but snake eyes and craps are not.

*The mathematician will say, "Not necessarily all sets," and define σ-algebras, which are non-empty families of sets that are closed under complement and finite union, but we need not worry about that refinement here.

A *probability distribution* for sample space S assigns to each subset* of the sample space S a real number satisfying three simple axioms:

A1. For all $A \subseteq S$, the probability $Pr\{A\} \geq 0$, and
A2. $Pr\{S\} = 1$,
A3. For mutually exclusive events $A, B \subseteq S$, we have $Pr\{A \cup B\} = Pr\{A\} + Pr\{B\}$.

From these axioms, we can already draw some conclusions. Can you justify each of these?

C1. $Pr\{\emptyset\} = 0$.

C2. If $A \subseteq B$, then $Pr\{A\} \leq Pr\{B\}$.

C3. For any $A, B \subseteq S$, we have $Pr\{A \cup B\} = Pr\{A\} + Pr\{B\} - Pr\{A \cap B\} \leq Pr\{A\} + Pr\{B\}$.

Notice that any probability distribution is a function defined on subsets of the sample space, $Pr\colon 2^S \to \mathbb{R}$. It is traditional to use slightly different notation because there are other functions (in particular, random variables, defined in section 14.2) that have the same domain and range but are not necessarily probability distributions.

Two events $A, B \subseteq S$ are *independent* iff $Pr\{A \cap B\} = Pr\{A\} \cdot Pr\{B\}$; that is, iff knowing that you have a sample in one event gives you no clue whether that sample is in the other. Thus, whether you roll even and craps is independent, but whether you roll even and natural is not.

We say that a 6-sided die is *fair* iff the outcome of each roll is independent, and the probability of each face is $1/6$, giving the *uniform distribution*. Axioms A1–A3 determine that the probability for any other event $A \subseteq [1..6]$ equals $|A|/6$. For example, the probability of ⚀ is $1/6$, the probability of less than 3 spots is $2/6$, and the probability of *even* is $1/2$. With two fair dice, one white and one De Morgan, each of the 36 elementary events would be assigned a probability of $1/36$. The probability of snake eyes is $1/36$, of craps is $4/36 = 1/9$, and of a natural is $8/36 = 2/9$. With probability $1 - 3/9 = 2/3$, you'll need to roll again.

With two white dice, there are only 21 different events, because we cannot distinguish between the two ways to roll a 3, for example, if we cannot tell which die shows which value. The 6 events of rolling doubles (⚀⚀ . . . ⚅⚅) would each have probability of 1/36, and the other 15 events of rolling two different numbers would have probabilities of 2/36 = 1/18. You can check that 6/36 + 15/18 = 1.

14.1.1 Monty Hall and sample spaces

Let's look at how phrasing a popular puzzle, the Monty Hall question [26], in the language of sample spaces and events helps us clarify assumptions. You are a contestant on a televised game show. Monty Hall will show you three doors; one door hides a new car, the remaining two doors each hide a goat. You will choose one door, and Monte Hall, because he knows where the car is, he will then open a door that you did not choose and show you a goat. Do you switch to the other unopened door, stay with you first choice, or does it not matter?

As I have phrased the problem, you want to switch. In order to see why, let's consider the sample space of 27 triples $S = \{1, 2, 3\}^3$, where the triple (A, B, C) indicates that you initially chose door A, Monty Hall opened door B, and in the end you learn that the car was behind door C.

Before the game begins, any triple is possible. Once the car is placed, 2/3 of the triples are impossible, but you don't know which. You can think of the car placement as a random experiment* that partitions S into 3 disjoint events, $S = S_1 \uplus S_2 \uplus S_3$, with each event S_C being the set of the nine triples ending with with C. You might as well assume that each event has probability 1/3. (We will revisit this assumption below.)

*For the individual contestant, who gets one chance to play, it isn't really a random experiment, but for Monty (and for the composite of all contestants) it is.

Let us also assume that your initial choice A and the location C of the car are independent: This implies that you have no inside information or extra-sensory perception, and that Monty is neither predicting your guess, nor cheating by moving the car after your guess.

Now, suppose that you initially choose door $A = 1$. figure 14.1 lists the nine possible elementary events of the form $(1, B, C)$ that can remain in the sample space after your choice. Each column C contains three elementary events remaining in the set S_C, which we assumed has probability 1/3.

The problem statement constrains Monty's choice of his door B in two ways: he cannot open the door that you opened, and he cannot open the door that hides the car. figure 14.1 indicates that if your initial guess was correct ($C = 1$), then Monty has a choice between two doors with goats, but in the other two events Monty's choice is forced, so he is essentially

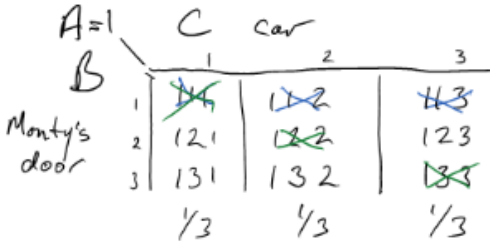


Figure 14.1: Elementary events possible after choosing door $A = 1$; Monty cannot chose the options crossed off.

telling you where the car is.

If you stay with your initial choice, you win only in event S_1, but if you switch, you win in both events S_2 and S_3. Since each event happened with probability 1/3, you double your chance of winning if you switch.

Calculating probabilities makes sense only for experiments that can be repeated under the same conditions, so it is important to be precise about these conditions.* A slight change to the problem statement can change the probabilities. For example, suppose that after you choose door A, Monte Hall will always open door $B = 1 + (A \bmod 3)$, no matter what is behind it. If he reveals the car (event S_B with probability 1/3), you lose immediately. Otherwise you now know you are in one of the other two events, each of probability 1/3, and it doesn't matter whether you keep your door or switch.

*After Maralyn vos Savant [26] popularized this puzzle, the real Monty Hall showed her statement was ambiguous because it didn't force him to offer the switch: on the show, he could open A if that hid a goat, or offer cash not to open any door [25].

Your chances are much better if Monty Hall *must* show you a goat than if he *just happened* to show you a goat, even though the snapshot of the game at the instant you make your switch/stay decision looks the same. This sensitivity to the conditions and assumptions is one reason this puzzle generates much debate.† Mathematical language encourages us to be precise about the conditions and assumptions that underly our conclusions.

†vos Savant reports tens of thousands of letters on this puzzle, including over 1,000 from PhD's incorrectly arguing that switching gave no advantage [14, 25].

We assumed that the car was equally likely to be behind any of the three doors. What if we have kept track of the history, and expect it to be behind door i with probability p_i? If we choose the door with smallest probability and switch, then we would win with probability $1 - \min\{p_i\}$. So Monty's best strategy is to make that minimum as large as possible, by setting all $p_i = 1/3$.

We also assumed that A and C were independent. Suppose, however, that Monty knows that you plan to initially choose door j with probability q_j and then switch; he can put the car behind the door you are most likely to choose, and you win with probability $1 - \max\{q_j\}$. So your best strategy is to choose your door A uniformly at random—setting all $q_j = 1/3$. This makes the maximum as small as possible.

14.2 Random variables and expectation

A *random variable* X is a function on subsets of a sample space $X\colon 2^S \to \mathbb{R}$, Think of it as a variable whose value you won't know until you actually carry out a random experiment. Examples include the number of spots on two dice, which you don't know until your roll them, or the amount you win on a lottery ticket (minus the price you paid for it), which you don't know until the numbers are called (this amount is almost always negative).

Although we do not know the value of a random variable, we can determine its *expected value* or *expectation*: the sum over the possible values of the random variable, each weighted by the probability that it takes on that value: $E(X) = \sum_{r\in\mathbb{R}} rPr\{X = r\}$. For example, what is the expected number of spots

on one roll of a fair die? Let X be the random variable for number of spots showing with one roll. The expected number of spots showing,

$$E(X) = \quad 5$$

What is the expected number of spots showing on two fair dice? The sample space is $[2 \ldots 12]$, and you roll 2 spots in 1/36 ways, 3 spots in 2/36 ways, ..., 7 spots in 6/36 ways, 8 spots in 5/36 ways, ..., 12 spots in 1/36 ways, from which we could calculate the expected number of spots. But there is a much easier way using the next lemma.

We can do arithmetic on random variables to make new random variables. If $X\colon 2^S \to \mathbb{R}$ and $Y\colon 2^T \to \mathbb{R}$ are random variables and $a \in \mathbb{R}$ is a real number, we can make new random variables

$A = aX$ Do the experiment for X and multiply the number you get by a.

$B = X + Y$ Do the experiment for X, then for Y, and sum the results.

$C = XY$ Multiply the results of experiments for X and for Y.

Note that $X+X$ may be ambiguous: does it involves two separate experiments, or is it the same as $2X$, which doubles the result of one experiment? To avoid confusion, name two separate variables if you mean two experiments. For example, to roll two six-sided dice, let random variable X_b be the number of spots on a roll of a De Morgan die, and X_w be the spots on a white die. Their sum, $X_b + X_w$, is a function from sets of two dice rolls to the values in $[2..12]$. It is surjective, hitting both even and odd values, where $2X_b$ hits just the evens.

*A mathematician would say that expectation is a linear functional.

The next lemma says that the expectation of a sum is the sum of expectations.* Thus, rolling one De Morgan and one white dice, $E(X_b + X_w) = E(X_b) + E(X_w) = 7$.

Lemma 14.2.1. *For random variables X and Y, and real a, the expectation $E(aX+Y) = aE(X) + E(Y)$.*

1.	$E(X + Y) = \sum_{r\in\mathbb{R}} rPr\{X+Y = r\}$	defn of expectation
2.	$= \sum_{r\in\mathbb{R}} \sum_{x\in\mathbb{R}} rPr\{X=x \wedge Y=r-x\}$	defn rnd var $X + Y$
3.	$= \sum_{x\in\mathbb{R}} \sum_{r\in\mathbb{R}} rPr\{X=x \text{ and } Y=r-x\}$	reorder sums
4.	$= \sum_{x\in\mathbb{R}} \sum_{x+y\in\mathbb{R}} (x+y)Pr\{X=x \wedge Y=y\}$	replace $r = x + y$
5.	$= \sum_{x\in\mathbb{R}} \sum_{y\in\mathbb{R}} (x+y)Pr\{X=x \wedge Y=y\}$	$\sum_{x+y\in\mathbb{R}} \equiv \sum_{y\in\mathbb{R}}$
6.	$= \sum_{x\in\mathbb{R}} x \sum_{y\in\mathbb{R}} Pr\{X=x \wedge Y=y\}$	distribute &
7.	$\quad + \sum_{y\in\mathbb{R}} y \sum_{x\in\mathbb{R}} Pr\{X=x \wedge Y=y\}$	reorder sum
8.	$= \sum_{x\in\mathbb{R}} xPr\{X=x\} + \sum_{y\in\mathbb{R}} yPr\{Y=y\}$	first term sums all y
9.	$E(X) + E(Y)$	defn of expectation.

∎

Stop and work through that detailed derivation, because it will give you an appreciation for being able to use the *linearity of expectation* without needing to derive it afresh each time. No single step is hard.[†]

[†] In fact, once you work through it, you'd probably be happy if I combined steps, but because of the importance I made this proof unusually detailed.

An *indicator random variable* takes on the value one if an event happens and zero otherwise: $I\colon 2^S \to \{0,1\}$. Expectation for an indicator random variables is especially simple: $E(I) = 0 \cdot Pr\{I=0\} + 1 \cdot Pr\{I=1\} = Pr\{I=1\}$. As we will see, indicator random variables are useful bookkeeping devices for analyzing events that may occur in computer programs. Here is another example.

In **??** we saw that, for a given r, if you choose a large enough complete graph, K_n, then no matter how you two-color the edges, you will always find a complete subgraph K_r that is monochromatic. Here we use expectation to show the opposite for small graphs:

Lemma 14.2.2. *For a complete graph K_n with $n \le 2^{r/2}$, there exists a coloring with no monochromatic K_r.*

Proof. Let each edge of the graph independently choose red or blue with probability $1/2$. The probability that a given r-clique is monochromatic is , because once the first edge chooses its color, every other edge must choose the same.

The number of r-cliques in complete graph K_n is . The expected number of monochromatic r-cliques is the product of these; can you explain why?

For $r \ge 2$, $\binom{n}{r} < n^r$ and $r^2/2 \le \binom{r}{2} - 1$. By the hypothesis of this lemma, $n \le 2^{r/2}$, so the expected number of monochromatic r-cliques is less than one. But since the number of monochromatic r-cliques in any coloring is an integer, for the average over all colorings to be less than one, then at least one coloring has no monochromatic r-cliques. ∎

((todo. Decide how much to say about games here, or make a game theory.)) The proof of Lemma 14.2.2 actually implies that at least half the random colorings of K_n, with ??, do not have monochromatic r-cliques. But checking that a coloring has no monochromatic cliques is still time consuming!

This is a non-constructive proof and gives no idea how to actually find such a coloring. You can turn this into a Maker-Breaker game: Start with n vertices and no

edges. Maker proposes an new edge (i, j) and Breaker assigns the color red or blue. Maker wins if a monochromatic r-clique is created.

Maker can win if the game is played for 2^{2r+1} rounds using a recursive strategy, starting with a set of vertices V. The ith time, select and remove a vertex v_i from V, then ask for the colors of all edges (v_i, w) for all $w \in V \setminus \{v_1\}$. Keep vertices in V whose edges are the majority color, and discard the rest, then proceed to the next recursive call. If you end with at least $N(0) = 1$ vertex, then you began the jth-from-last recursive call with at least $N(j) = 2N(j-1) + 1 = 2^{j+1} - 1$. Thus, you began with at least $N(2r-1) = \cdot 4^r - 1$ vertices.

On the other hand, Breaker can make the game go at least $2^{r/2}$ rounds as follows: Let V be the set of vertices whose edges Maker has requested colors for, **((todo. finish))**.

Calculation with random variables eventually becomes so natural that they are often used to specify event sets and their probabilities. E.g., one can write $Pr\{X_b + X_w = 2\} = 1/36$, because the only way to get 2 is to roll snake eyes, ⚀⚀, and $Pr\{X_b + X_w = 7\} = 1/6$ because six different rolls add to 7. Recall that probability and random variables are both functions that map event sets to reals. So if we defined $f = Pr\colon 2^S \to \mathbb{R}$ and $g = X_b + X_w\colon 2^S \to \mathbb{R}$, then this notation would be written $f(g^{-1}(7)) = 1/6$, meaning that $g^{-1}(7)$ is the set of events (dice rolls) with value 7, and $f(\cdot)$ is the probability of those events.

14.2.1 Non-transitive dice

Let's use this notation to analyze a scam suggested by James Grime and Brian Brushwood: non-transitive dice.

Make three pairs of dice in traffic-light colors: a Green pair with five ⚂s and one ⚅, a Yellow pair with three ⚂s and three ⚄s, and a Red pair with one ⚀ and five ⚃s. Let a friend choose one die, and you choose the color above on a traffic light (with wrap around: you choose Red if they chose Green). Now, you both roll, and the die showing the most spots wins that roll. First person to win 7 rolls wins the game.

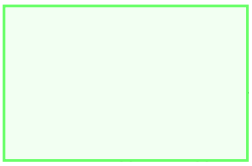
From a table of possibilities (table 14.1) you can calculate the probability that Green beats Yellow on a single roll. Work out the possibilities for Yellow-Red and Red-Green pairs, and find their probabilities, too. [?] You'll observe a non-transitive relationship: $G > Y > R > G$. That is, Green likely beats Yellow, Yellow beats Red, and Red beats Green.

Below you can work out you probability of being first to win 7 rolls by counting the possible roll sequences. [?]

First, however, James and Brian suggest that the scam can continue after you tell your friend about the traffic light order: you choose first and let your friend choose the color that beats it. But now, double the stakes, double the target, and each roll two dice. You are still more likely to win,

Table 14.1: Rolling one die for pairs of colors: events, probabilities, and winners.

	Green	
	5/6 ⚀	1/6 ⚅
Yellow		
3/6 ⚀	$\frac{15}{36}$ G	$\frac{3}{36}$ G
3/6 ⚄	$\frac{15}{36}$ Y	$\frac{3}{36}$ G



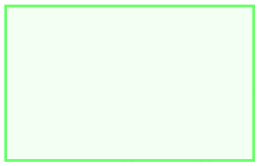
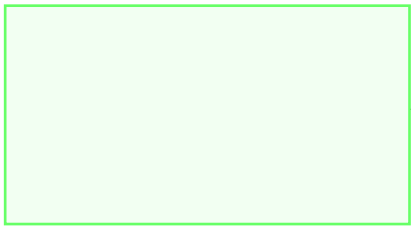
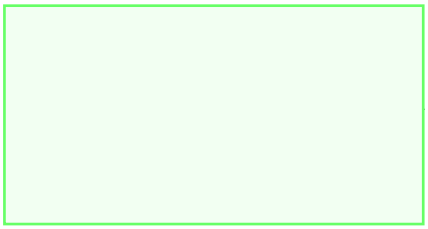


Table 14.2: Rolling two dice for pairs of colors: events, probabilities, and winners.

	Green		
	25/36 ⚀⚀	10/36 ⚀⚅	1/36 ⚅⚅
Yellow			
1/4 ⚀⚀	$\frac{25}{144}$ G	$\frac{10}{144}$ G	$\frac{1}{144}$ G
2/4 ⚀⚄	$\frac{50}{144}$ Y	$\frac{20}{144}$ G	$\frac{2}{144}$ G
1/4 ⚄⚄	$\frac{25}{144}$ Y	$\frac{10}{144}$ Y	$\frac{1}{144}$ G





because the order reverses when rolling two dice: $G < Y < R < G$. Can you work out the probabilities for winning one roll ?, and for being first to 14 winning rolls? ? (The probabilities of a Red pair over a Green pair is not much over 50%, so there you'll still need some luck.)

When the probability of winning one roll is p, what is the probability of winning the best two out of three, best four out of seven, or n out of $2n-1$? To work out an exact answer, we could use either of two natural sample spaces: 1) play until someone wins for the nth time, or 2) play $2n-1$ rounds and see who won the most. If we let k be the number of times we lose, for $0 \leq k < n$, then we want to know in 1) the probability of all sequences of length $n+k$ with n wins and k losses, and in 2) the probability of all sequences of length $2n-1$ with $2n-1-k$ wins and k losses. These counts give two formulae that each look similar to the binomial theorem: figure them out, and check that they agree for small n.

$$\square = \square^{k}.$$

As n grows, these expressions become quite hard to calculate because numbers like p^n become very small, while $\binom{2n-1}{k}$ becomes very large. In most computer languages the computer approximates real numbers in scientific notation, limiting both the size of the exponent and the precision of the mantissa. Try to find the n values for the single and pair rolls so that you win the best n of $2n-1$ at least 90% of the time—for pairs, direct computation gives me 'infinity,' which is incorrect.

Probability gives some useful tools for bounding probabilities using expectations. The first is really trivial, once you get past the notation and the fact that it is named.

Theorem 14.2.3. *Markov Inequality. For a non-negative random variable $X \geq 0$ and positive real a, the probability $Pr\{X \geq a\} \leq \frac{E(X)}{a}$.*

Here is a way to work out what this means on your scratch paper: we dump a unit amount of probability (as dirt) on the real number line, but don't let it spill onto the negative side. We want to know how much dirt is past $x = a$, which is $Pr\{X \geq a\}$.

The inequality uses the average, $E(X)$; dirt on both sides of this position must balance, so let's push this position as close to zero as we can and check if the inequality remains true. We can push all dirt in $[0, a)$ down to 0, and all dirt in $[a, \infty)$ down to a. This gives an average of $aPr\{X \geq a\} \leq E(X)$.

This is too weak to help us directly, even though the number of wins in N rolls is a non-negative random variable and we can figure out the expected number of wins is $N \cdot p$, since it is a sum of indicator random variables. But we can apply it to another positive distribution, the *variance*, which will help:

Theorem 14.2.4. *Chebyshev* Inequality. For a non-negative random variable $X \geq 0$ and positive real a, the probability $Pr\{(X - E(X))^2 \geq a\} \leq \frac{E(X)}{a}$.*

*Or Tchebycheff, using the French transliteration of the Russian.

14.3 Examples: balls into bins

[†] These questions come from algorithms for bucket sort and hashing.

Let's look at some balls and bins questions related to expectation and using random variables.[†] Suppose that you have n balls and m bins. Throw each ball independently into a random bin with uniform probability—each bin being equally likely.

Q1. What is the expected (average) number of balls per bin?
Q2. What is the expected number of balls already in the bin that each ball hits?
Q3. What is the expectation for the sum of the squares in each bin?
Q4. What is the expected number of empty bins?
Q5. How many bins would we look at before finding the first empty bin?
Q6. What is the expected maximum number of balls in any bin?

For many of these we can give exact values; for others we will have to be content with upper bounds. Let's define and give (completely arbitrary) names to two random variables that will be useful:

Let $Z_{i,k} = \begin{cases} 1 & \text{if ball } i \text{ goes into bin } k, \\ 0 & \text{otherwise.} \end{cases}$

By the assumption of uniformity, the probability $Pr\{Z_{i,k} = 1\} =$. And since every ball goes into some bin, $\sum_{1 \le k \le m} Z_{i,k} =$.

Let $X_{i,j} = \begin{cases} 1 & \text{if balls } i \text{ and } j \text{ go into the same bin,} \\ 0 & \text{otherwise.} \end{cases}$

By the assumption of uniformity and independence, the probability $Pr\{X_{i,j} = 1\} =$. There is another relationship to Z:

Lemma 14.3.1. *For throwing n balls into m bins, $X_{i,j} = \sum_{1 \le k \le m} Z_{i,k} Z_{j,k}$.*

Proof. $Z_{i,k} Z_{j,k} = 1$ iff both ball i and ball j go into bin k. Summing over all bins counts i, j at most once, and only if they both land in the same bin. ∎

[*] These are the first two terms Taylor series of e^x, and the error term is positive.

We need one useful fact from calculus to find upper bounds:

Observation 14.3.2. *For all reals x, we have*[*] $1 + x \le e^x$.

A1. What is the expected (average) number of balls per bin?

The average is the number of balls over the number of bins: n/m. But as a warm-up, let's use random variables $Z_{i,1}$ to determine the expected number of balls in bin 1:

$$E\Big(\sum_{1 \le i \le n} Z_{i,1}\Big) = \sum_i E(Z_{i,1}) = \sum_i Pr\{Z_{i,1} = 1\} = \sum_{1 \le i \le n} \frac{1}{m} = \frac{n}{m}.$$

A2. What is the expected number of balls already in the bin that each ball hits?

Here we have $\binom{n}{2}$ pairs, each of which has a $1/m$ chance of landing in the same bin. But let's continue to use random variables. We can sum $X_{i,j}$ for pairs with $i < j$:

$$E\Big(\sum_{1\le i<j\le n} X_{i,j}\Big) = \sum_{i<j} E\left(X_{i,j}\right) = \sum_{1\le i<j\le n} \frac{1}{m} = \frac{n(n-1)}{2m}.$$

A3. What is the expectation for the sum of the squares in each bin?

This quantity is interesting if, for example, each ball coming into a bin compares itself to all balls already there. If you have only one bin, then the total number of comparisons grows as a quadratic function in n, but if the number of bins equals the number of balls, we shall see that the number of comparisons is only linear in n. Here is where random variables really start to show their power. We rearrange expressions and reinterpret.

1.	$E\Big(\sum_{1\le k\le m}(\sum_{1\le i\le n} Z_{i,k})^2\Big)$	for each bin, count balls and square
2.	$= E\Big(\sum_k(\sum_{1\le i\le n} Z_{i,k})(\sum_{1\le j\le n} Z_{j,k})\Big)$	expand square, rename index
3.	$= E\Big(\sum_k \sum_i \sum_j Z_{i,k}Z_{j,k}\Big)$	distribute $\cdot$ over $+$
4.	$= E\Big(\sum_i \sum_j \sum_{1\le k\le m} Z_{i,k}Z_{j,k}\Big)$	reorder sum
5.	$= E\Big(\sum_i \sum_j X_{i,j}\Big)$	by Lemma 14.3.1
6.	$= \sum_i \sum_j E(X_{i,j})$	linearity of expectation
7.	$= \sum_{1\le i\le n} E(X_{i,i}) + 2\sum_{1\le i<j\le n} E(X_{i,j})$	sum $i=j$, $i<j$, $i>j$ separately
8.	$= \sum_{1\le i\le n} 1 + 2\sum_{1\le i<j\le n} 1/m$	i always with itself, $1/m$ with j
9.	$= n + \frac{n(n-1)}{m}$.	

QED

A4. What is the expected number of empty bins?

Define a new random variable to give an upper bound on a bin being empty:

Let $Y_k = \begin{cases} 1 & \text{if bin } k \text{ is empty after } n \text{ balls,} \\ 0 & \text{otherwise.} \end{cases}$

The probability $Pr\{Y_k = 1\} = (\frac{m-1}{m})^n = (1-\frac{1}{m})^n \le e^{-n/m}$, since to keep bin k empty, each ball must independently choose one of the $m-1$ other bins. So, an upper bound on the expected number of empty bins is

$$E\Big(\sum_{1\le k\le m} Y_k\Big) = \sum_{1\le k\le m} E\left(Y_k\right) \le \frac{m}{e^{n/m}}.$$

A5. How many bins would we inspect before finding an empty bin?

This will depend on how we do the inspection and how many bins are empty. The application is to a data structure for hashing by open addressing, which starts at a random bin, say 1, and scans through 2, 3, . . . , until finding an empty bin. This is complex, though, so let's change the problem to consider simpler variants first.

Suppose that $0 \le a \le 1$ is the fraction of non-empty bins, and that we inspect bins at random, or that the number of bins is infinite.

If we do inspection by repeatedly picking a random bin, then we

There are several choices for how to interpret this problem. Rather than considering all the choices before we decide which to try to solve, let's change the problem to make it easier to solve and explore variations. Solving simpler versions first gives us ideas and confidence; knowing that we are exploring the statement of the problem as well as the solution keeps us from thinking we"ve solved a problem before we've fully understood it.

First variation: Suppose that we know the fraction $a \in [0, 1]$ of bins that are non-empty. Suppose that we look for an empty bin by inspecting (and possible re-inspecting) bins uniformly at random. Then there is a simple formula for the probability that we have to inspect the kth bin: □.

Create an indicator random variable for each k for the first k bins inspected all being non-empty.

$$P_k = \begin{cases} 1 & \text{Bins } 1 \ldots k \text{ were all empty,} \\ 0 & \text{otherwise.} \end{cases}$$

so let's consider some variations. We'll change the problem to make it easier to solve Suppose that we have an infinite supply of bins, each non-empty with probability a. The probability that the first k bins we inspect are all non-empty is $Pr\{P_k = 1\} = a^k$.

The expected number of non-empty bins is bounded by

$$E\Big(\sum_{1 \le k \le m} P_{k-1}\Big) < \sum_{1 \le k} a^k = \frac{1}{1-a} - 1 = \frac{1}{1/a - 1}.$$

A different way to look at this would be to have b non-empty bins out of a **((todo. fix ocr and complete))** Then Choose non-empty bins from the $m - k + 1$ Set of all bins, without replacement The upper bound of = still holds Finally, if we have n balls in m bins

A6. What is the expected maximum number of balls in any bin?

Here we can bound the expected maximum, rather than compute it. I'll do only the special case $m = n$. How big can $\sum_{1 \le i \le n} Z_{i,k}$ grow? We can have

all balls in one bin, $\max_{1\le k\le m}\sum_{1\le i\le n} Z_{i,k} = n$, but that is very unlikely, as it happens in only $m/m^n = 1/m^{n-1}$ ways. The maximum expectation is easy to calculate $\max_k E\left(\sum_i Z_{i,k}\right) = \max_k n/m = n/m$, but we want expected max, which is going to be larger.

The idea is to find a k so that the number of balls in a single bin exceeds k with probability $< 1/n^2$. This means that with n bins, the probability that any exceeds k is $< 1/n$. In the expectation formula, the values $\le k$ are rounded up to k and those $> k$ are rounded up to n, giving an upper bound on the expected maximum.

So, throwing n balls into m bins, in how many ways can more than k keys land in bin 1? An over-count would be $\binom{n}{k}\frac{1}{m^k}$; choosing k of the n keys to land in bin one, and letting the rest fall as they may. (This is an over-count because instances with $j > k$ keys landing in bin 1 are counted $\binom{j}{k}$ times.) Can you determine a value of k to make this less than $1/n^2$?

Now, with probability $p < 1/n$ the maximum bin size is $\le n$, and with probability $(1-p)$ it is $\le k$, so the expectation for the maximum is $< (1-p)k + pn < k + 1 = 3\ln n/\ln\ln n$.

Look back at the observations that this uses: First, in all cases the maximum is $\le n$. Second, in all but $< 1/n$ of the cases, the maximum is $\le k$, because this is how we chose k. Thus, the rare cases contribute at most 1 to the expectation, while the others contribute k.

14.4 Exercises and Explorations

Quiz Prep 14.1. Compute the probabilities that demonstrate these claims from the text. Define events E = roll an even number of spots on two dice, F = roll at least one five, C = roll craps, and N = roll natural.

1. Compute $Pr\{E\}$ = ?, $Pr\{C\}$ = ?, $Pr\{N\}$ = ?, and $Pr\{F\}$ = ?.
   $Pr\{E \cap C\}$ = ?, $Pr\{E \cap N\}$ = ?, $Pr\{E \cap F\}$ = ?, and $Pr\{F \cap N\}$ = ?

2. In a sample space defined by n outcomes, how many pairs of mutually exclusive events are possible?

3. Rolling evens and craps is independent, but rolling evens and natural is not.

Exercise 14.2. Pick an integer m at random from $[1..n]$. What is the probability you have a perfect square—that there exists $k \in Z$ such that $m = k^2$? ▶

Exercise 14.3. The student who decided to answer the exam's true/false questions by flipping a coin was still the last to finish, because he went back to check his answers. What is the expected number of flips to see the same side twice in a row? ? How many flips is the student expected to need to answer and check n questions, if each check that doesn't agree with an answer becomes the new answer. ? How many if for each check they erase the previous answer and try again? ?

Exercise 14.4. Your friend holds out 7 cards, face down, and says 2 are hearts and 5 are spades. If you pick k cards, what are your chance of getting at least one a red card? Complete this table:

k	1	2	3	4	5	6	7
prob	?	?	?	?	?	?	?

▶

*A carnival employee, employed by the "house."

Exercise 14.5. Chuck-a-luck, or Birdcage, is a carnival gambling game whose barker* rolls three 6-sided dice for six players. Each player chooses one of the numbers 1–6, and receives \$1 for each die that shows their number, or pays \$1 if none of the dice does. Although the barker may advertise "three losers and three winners," giving the impression that the game is break-even, the house rakes in a significant percentage. How and how much? (You may want to list the possible outcomes: it helps to assume that the dice have three different colors.) What should the payouts be for the players to break even? ▶

Exercise 14.6. I flip two fair coins (a dime and quarter) and tell you that one of the coins shows heads. What is the probability that the second also shows heads? There is actually not enough information to answer this question: you need to know a bit about my strategy. What is your answer if:

1. I always tell you what is showing on the dime. ?
2. I pick one of the coins at random and report what is showing. ?
3. Whenever I throw two tails, I throw again; I report results only when there is at least one head. ?
4. Whenever possible, I report that one coin shows tails. ?

Exercise 14.7. I flip a coin that comes up heads with probability p. How many runs of heads should I expect on n flips? Two 'H' flips are in the same run if there is no 'T' flipped between them.

Exercise 14.8. A fair 6-sided die is rolled until a 6 comes up. What is the expected number of spots before that event? For example, I just rolled $1, 2, 2, 6$, which is 5 spots. My next attempt had 33.

Puzzle 14.9. You are given a shuffled deck containing $2n$ cards, n red and n De Morgan, which you will turn over one by one. With each card you may bet any amount up to your credit on which color will appear. If you are right, you win that amount, and if you are wrong, you lose that amount. This means that you can guarantee to double your money by betting zeros until there is one card left, and then betting all your credit. What is the largest amount that you can guarantee to win?

Exploration 14.10. Here are three scenarios that each begin with n red and n blue balls. Each has a protocol for removing a ball until only one color remains. The question is how balls many of the remaining color do you expect to find?

To make sure you understand each scenario, determine, with r red and b blue remaining, the probability that you remove red. Then click on the '?'.

1. There are two bins, one containing the red balls, and one containing the blue. You flip a fair coin, and for heads you remove a red ball and for tails you remove a blue. Stop when one bin is empty. ?
2. All balls are in one bin. You have no coin, so you simply pull out a ball at random and remove it. ?
3. All balls are in one bin, but when you pick a ball at random, you put it back and instead remove a ball of the opposite color. ?

Simulate each scenario in your favorite programming language, and see if you can explain the results that you find. (The first is easy to explain; the others are much harder.)

Bibliography

Do not consider it proof just because it is written in books, for a liar who will deceive with his tongue will not hesitate to do the same with his pen.
—Maimonides

[1] Edsger W. Dijkstra. My hopes of computing science. In *Proc. 4th Int. Conf. on Software Engineering*, Munich, September 1979.

[2] S.S. Epp. *Discrete Mathematics With Applications*. Cengage Learning, 2010.

[3] M. Erickson. *Aha! Solutions*. MAA Problem Book Series. Mathematical Association of America, 2009.

[4] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, 8:128–140, 1736. Opera Omnia (1) 7 (1911-56), 1-10

An engineer, a physicist and a mathematician find themselves in an anecdote, indeed an anecdote quite similar to many that you have no doubt already heard.

[5] M. Fellows. The heart of puzzling: Mathematics and computer games. In *Proceedings of the 1996 Computer Games Developers Conference*, pages 109–120. Miller Freeman, 1996.

[6] M. Gardner. *aha! A two volume collection: aha! Gotcha & aha! Insight*. Spectrum Series. Mathematical Association of America, 2006.

[7] Solomon W. Golomb. Combinatorial proof of Fermat's "little" theorem. *Amer. Math Monthly*, (10), December 1956.

[8] G.D. Gopen. *The sense of structure: writing from the reader's perspective*. Pearson Longman, 2004.

[9] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994.

After some observations and rough calculations, the engineer realizes the situation and starts laughing.

[10] P.R. Halmos. *Naive Set Theory*. Undergraduate Texts in Mathematics. Springer, 1998.

[11] Thomas L. Heath. *The Thirteen Books of Euclid's Elements*, volume 2. Dover, 1956.

[12] R. P. Boas Jr. "if this be treason. . . ". *The American Mathematical Monthly*, 64(4):pp. 247–249, 1957. Collected in "Lion Hunting & Other Mathematical Pursuits," Boas, R.P. and Alexanderson, G.L. and Mugler, D.H., MAA, 1996.

[13] Ken Keeler. The Prisoner of Benda, August 2010. Futurama, season 6, episode 10.

[14] Stefan Krauss and X. T. Wang. The psychology of the Monty Hall problem: Discovering psychological mechanisms for solving a tenacious brain teaser. *Journal of Experimental Psychology*, 132(1):3–22, 2003.

[15] Imre Lakatos. *Proofs and refutations*. Cambridge University Press, Cambridge, 1976. The logic of mathematical discovery, Edited by John Worrall and Elie Zahar

A few minutes later, the physicist understands too and chuckles to himself happily as he now has enough experimental evidence to publish a paper.

[16] Leslie Lamport. How to write a proof. *The American Mathematical Monthly*, 102(7):600–608, 1995.

[17] Leslie Lamport. *Specifying Systems*. Addison-Wesley, July 2002.

[18] Radia Perlman. An algorithm for distributed computation of a spanningtree in an extended lan. *SIGCOMM Comput. Commun. Rev.*, 15(4):44–53, September 1985.

[19] G. Polya. *How to solve it*. Princeton Science Library. Princeton University Press, Princeton, NJ, 2004. A new aspect of mathematical method, Expanded version of the 1988 edition, with a new foreword by John H. Conway.

[20] Raimond Reichert. *Theory of Computation as a Vehicle for Teaching Fundamental Concepts of Computer Science*. PhD thesis, May 2003.

[21] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[22] K. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Companies,Incorporated, 2011.

[23] Walter Schnyder. Embedding planar graphs on the grid. In *Proc. 1st Annual ACM-SIAM Symp. On Discr. Alg. (SODA)*, pages 138–147, 1990.

This leaves the mathematician somewhat perplexed, as he had observed right away that he was the subject of an anecdote, and deduced quite rapidly the presence of humor from similar anecdotes, but considers this anecdote to be too trivial a corollary to be significant, let alone funny.

[24] G.J. Summers. *Test Your Logic*. Dover Recreational Math Series Dover Publications, 1972.

[25] John Tierney. Behind Monty Hall's doors: Puzzle, debate and answer? *New York Times*, pages 1, 20, July 21, 1991.

[26] Maralyn vos Savant. *The power of logical thinking*. St. Martin's Press, New York, 1997.

[27] Herbert S. Wilf. *Generatingfunctionology*. A. K. Peters, Ltd., Natick, MA, USA, 3rd edition, 2006. 2nd ed at http://www.math.upenn.edu/~wilf/DownldGF.html.

Selected Solutions

The mathematician likened proving a theorem to seeing the peak of a mountain and trying to climb to the top. One establishes a base camp and begins scaling the mountain's sheer face, encountering obstacles at every turn, often retracing one's steps and struggling every foot of the journey. Finally when the top is reached, one stands examining the peak, taking in the view of the surrounding countryside and then noting the automobile road up the other side!

—Robert J. Kleinhenz

Extension 1.3. Gray code. You want to design a *position coder* that can report the angle of a rotating shaft. You could encode 2^b positions by striping the shaft with the b-bit binary count pattern from figure 1.2 in b concentric circles and positioning b brushes or photodiodes to sense the bit pattern. Unfortunately, adding or subtracting 1 may change many bits of a binary number, and in a mechanical device they never all change simultaneously. Thus, we get spurious readings as the shaft rotates from one count to the next.

Frank Gray observed that the same 2^b b-bit binary numbers could be put into a circular order so that from one number to the next only a single bit changes: toggling from 0→1 or 1→0. His *reflected binary Gray code* can be constructed as follows. For one bit, use order 0,1. For two bits, use order 00,01,11,10, which places 0s in front of the one-bit code, then 1s in front of the reverse of the 1-bit code. For k bits do the same: write 0s in front of the $(k-1)$-bit code, then 1s in front of the reverse of the $(k-1)$-bit code. On scratch paper, try writing the orders for the 3-,4-, and 5-bit codes, then check figure 1.4 in the text.

The patterns to notice are 1) every other move toggles of the least significant bit, and 2) in between, scan left to find the rightmost 1 bit, and toggle the bit to the left of it.

Program Kara (or another TM simulator) to list numbers in Gray code order. figure 1.2 shows the state diagram for a Kara program with five states that copies rows on Kara's 2-d tape. The state names are FindLSB, Find1, FLIP, COPY, and stop; transitions are shown only for the FindLSB state. On a 1-d tape, 4 states would be enough, since you wouldn't need to copy.

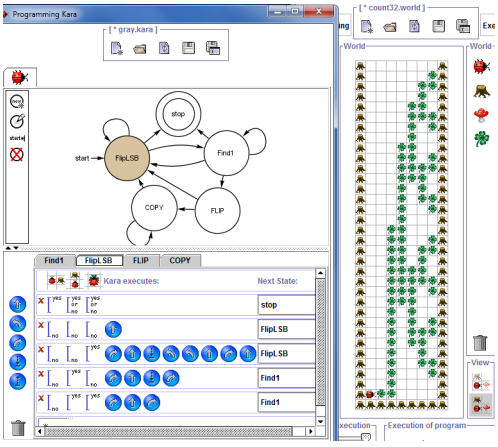


Figure 1.2: A 5-state machine can generate a Gray code on Kara's 2-d tape in a column bounded by stumps. With a 1-d tape, four states is enough.

◀

Exploration 1.4. *Busy Beaver Turing machines* fill an initially blank tape

with a string of as many clovers as possible, and then halt. (Halting is the challenge – it is trivial to make Kara run forever filling the tape with clovers.) Kara is a little different from the standard Turing machine, but if we use an $n \times 1$ world and limit the number of move-ahead actions, we can create a Kara version of the Busy Beaver problem:

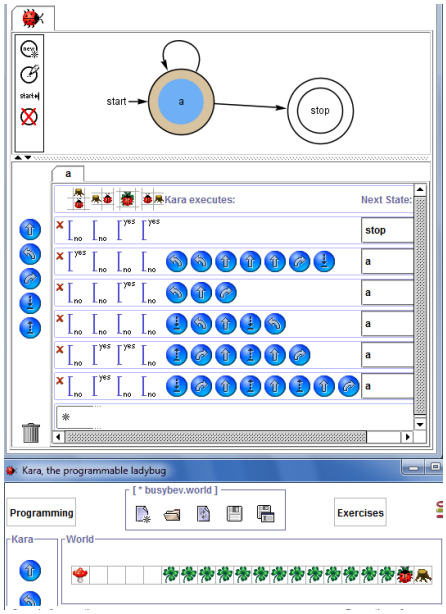


Figure 1.3: A Kara "Busy Beaver" with one state plus 'stop' can generate 14 clovers.

◀

Exercise 2.5. Operator precedence: In arithmetic expressions we have precedence rules that says that in an expression like $1 \cdot 5 - 8/4 + 2^3$ evaluate the exponential, then multiplication and division (left to right), and finally addition and subtraction (left to right). (Some learn these as PEMDAS.) In logic, the order is parentheses, negation ($\neg$), and ($\wedge$), or ($\vee$), and if,iff ($\rightarrow, \leftrightarrow$), with operations evaluated from *right to left*. Insert parentheses in these expressions so they they will evaluate correctly even if you follow only the parentheses rule.

1. $p \vee q \wedge r$ is _____.
2. $p \rightarrow \neg q \rightarrow r$ is _____.
3. $p \leftrightarrow q \rightarrow r$ is _____.
4. $p \vee \neg q \rightarrow r$ is _____.

5. $p \wedge q \to p \vee q$ is _____.

6. $p \vee \neg q \leftrightarrow \neg(\neg p \wedge q)$ is _____.

◀

Exercise 2.6. Find the mistake(s) in each of the following. *

*Warning: incorrect statements in this problem!

1. The negation of $0 < x < 5$ is $0 \geq x \geq 5$.

   $0 < x < 5$ is an abbreviation mathematicians use for the 'and' statement $(0 < x) \wedge (x < 5)$, so de Morgan says the correct negation is the 'or' statement $(0 \geq x) \vee (x \geq 5)$. Expand the 'and' before negating.

   Some strongly-typed programming languages will complain about $0 < x < 5$. Others (C++, FORTRAN, BASIC, etc.) will quietly evaluate $0 < x$ as 1 for true or 0 or -1 for false, then compare that number to 5 and get 'true' every time—not the intended behavior.

2. p only if q means $q \to p$.

   Actually, it means p can be false without repercussions, but p is true only if q is also true. We cannot have p true and q false. This is the definition of $p \to q$.

   When we say p iff q, p if and only if q, we are first saying the contrapositive $q \to p$ (p if q) and second the conditional $p \to q$ (p only if q). Once you get used to using $p \leftrightarrow q$ (p iff q) you, like me, will probably forget this and it will seem backwards if it is ever called to your attention.

◀

Exploration 2.16. Use your favorite spreadsheet program to create truth tables for logic operations or to solve logic puzzles. You can use TRUE/FALSE and logic operations (check your documentation for AND, NOT, OR, IF), or 0/1 and arithmetic operations (*, –, max, mod). First you'll want to set up a way to count through all possible T/F sequences for the variables, and then put in the logic expressions using those variables. Spreadsheets let you "fill down"† to replicate the formulas in a row to fill the entire table.

†The first spreadsheet program, Visicalc in 1979, already had replication commands.

Just a bit more specific detail on ways to make a truth table in a spreadsheet: It helps to know the commands fill-right and fill down (control-r and control-d in Excel) that copy formulas. Try this: Put in row 1 the text of the variable names. Fill the variables in row 2 with FALSE, or zero, whichever you prefer. For row three, the last variable toggles the value above, using `=NOT(C2)` or `=1-C2`. Other variables toggle whenever the next column goes 1 to 0 or TRUE to FALSE. That is, in cell A3, put `=IF(B2>B3,NOT(A2),A2)`, and copy that to all but the last variable's column. Then fill down to get all possible assignments to variables.

Next, put the text of the formulas in row 1, and the spreadsheet math in row 2. Spreadsheets support logic (AND, OR, NOT) on TRUE/FALSE, or you can use arithmetic operations on numbers to achieve the same operations (*, MAX, 1–). The ampersand (&) concatenates strings in Excel, and is not the operator for 'AND.' Check the documentation on IF; in Excel, $A \to B$ is written as =IF(A,B,TRUE), or you can use the equivalent =OR(NOT(A),B). I use =(A=B) for the biconditional, $A \leftrightarrow B$.

◀

Exercise 3.4. Assume that sets A and B do not contain tuples. Under what conditions does $A \times B = B \times A$? Be complete.

For every $a \in A$ and $b \in B$, we must have both $(a, b) \in B \times A$ and $(b, a) \in A \times B$.

This is vacuously true if A or B is the empty set $\emptyset$, because then both products are empty. It is also true if $A = B$.

If A and B are sets of tuples, we can make many other examples by from sets that are powers of another set, C. For example, $A = C^m$ and $B = C^n$ or $B = C^+$.

◀

Exercise 3.5. Combinatorial Pizza offers small, medium, and large pizzas with 14 possible toppings from 3 categories:

- Cheese (2): Mozzarella, Feta
- Veggie (7): Mushrooms, Peppers, Onions, Olives, Capers, Artichoke, Pineapple
- Meat (5): Salami, Pepperoni, Ham, Salmon, Anchovies

Combinatorial Pizza has five specials:

- Sampler: three different toppings from any categories.
- Balanced Diet: one topping from each of the three categories.
- Carnivore: You may choose one to four different kinds of meat.
- Vegan: Any size pizza with three Veggie toppings. (You can order more than one of a topping—my daughter likes triple pineapple.)
- Gut-buster: A large pizza with up to five toppings. (Ever had quintuple anchovies?)

The volume of a pizza of thickness a and radius z: pi z z a. –Wolfram Alpha

Let's assume that you don't care in what order the pizza chef puts the toppings on your pizzas. (E.g., They put the cheese first, even if you ask otherwise.) So, what you are ordering is a set of toppings (possibly a multi-set for the Vegan and Gut-buster).

In answering a question like this, the formula is more informative than the number, so be sure you show the formula. Each letter has a pop-up with the correct number so that you can check yourself.

1. How many different ways can you order a medium or large "Sampler" pizza? ?

Choose size and 3 of 14 toppings: $2\binom{14}{3} = 728$. (It is better if you don't multiply out because I can see where your numbers come from. I just do it for curiosity.)

2. How many different ways are there to order a large "Balanced Diet" pizza — one topping from each of the three categories? ?
   $5 \cdot 7 \cdot 2 = 70$ by product rule.
3. How many different ways are there to order two small "Balanced Diet" pizzas? ?
   Here we are counting a multiset of 2 pizzas from 70 options, allowing repetition. We can use the sum rule to add the ways to choose one pizza twice, 70, to the ways to choose two pizzas, $\binom{70}{2}$. Alternatively, we can add a new pizza type, "same," and choose a set of two pizzas from 71 types. $70 + \binom{70}{2} = \binom{71}{2} = 2,485$.
4. How many different "Carnivore" pizzas can be made? ?
   Choose size, then a subset of meat items that is not the empty set or set of all five: $3(2^5 - 2) = 90$.
5. How many different small "Vegan" pizzas are there? ?
   We can repeat choices, so we are sampling with repetition in $\binom{7-1+3}{3} = 84$ ways. How do we get that? Think about throwing 3 balls into 7 cups in a line, which we will represent by just the 6 vertical sides between them. Any string with 3 balls and 6 bars can be interpreted as a choice: e.g., $|\circ\circ||\circ|||$ is double topping 2 and topping 4, and $\circ|||\circ|||\circ$ is toppings 1, 4 and 7. We thus just have to choose the positions of the balls among $6 + 3 = 9$ symbols.
6. How many different "Gut-buster" pizzas are there? ?
   If we introduce a 15th topping of 'air', then this is exactly like the previous: $\binom{15-1+5}{5} = 11,628$. If you want to choose only 3 of the 5 toppings, then you choose double 'air.'

◀

Exercise 3.6. On a common single dial padlock, with numbers 1-40, a combination* is a 3-tuple.

*Should be called a "3-tuple" lock, because order is important and repeats are allowed.

How many combinations does such a lock have?
40^3

Since you can test all third digits with a single slow turn, how many pairs of the first two digits are there?
40^2

On some locks, it is enough to dial the first and second digit to within ±2. In that case, how many "slow turns" suffice to try all combinations?
8^2

◀

Exercise 3.7. In activity games, board games, and card games, players are often arranged in a circle. Sometimes the capability or personalities of the players to your left or right (or both) make a difference in your chances to win the game. See if you can get the same answer as the pop-up before looking at the reasons.

1. How many different orders are there for n players? Two orders are considered the same if and only if every player has the same player to their right.
   ? —$(n-1)!$ Seat someone to break the circle, then permute the rest.

2. What if $n = 2m$ players come as m pairs that want to sit next to each other? Now how many orders?

   $(m-1)! \cdot 2^m$

   —Order the pairs in $(m-1)!$ ways, then choose the order within each pair in 2^m ways.

3. You need to choose a set of k of the n people to help you; in how many ways can you do so if order does not matter?

   $\binom{n}{k}$

4. You want to pick the k of $n = 2m$ people that came in pairs so you take at most one of any pair; now how many ways?

   $\binom{n}{k}2^k$

   —Choose the pairs you'll break up, then choose one from each pair.

◀

Exercise 3.8. In how many ways can I choose k numbers from $[1..n]$, disregarding order, so that no two chosen numbers are consecutive (differ by 1)?

$\binom{n-k+1}{k}$. Here is a bijection between the ways to choose numbers and arrangements of cards of two colors: In a row of $n-k+1$ blank cards, choose k to color red. Insert a new blank card after each red card but the last, which is $k-1$ insertions. Then number all the cards, reporting the numbers of the red cards.

◀

Puzzle 3.11. How many positive integers have the property that their digits are strictly increasing as you read them from left to right? (Examples: 1, 128, 123,456,789.) How many positive integers have digits that are strictly decreasing from left to right? (Examples: 1, 42, 9630.)

The number of non-empty substrings of '123456789' is $2^9 - 1 = 511$, and of '9876543210' is $2^{10} - 1 = 1023$.

◀

Puzzle 3.12. A good exercise is to count the number of 5-card poker hands of different values. Let's assume a standard 52-card deck, with 13 cards (A, 2–9, 10, J, Q, K, A) in each of 4 suits (♡, ♢, ♣, ♠) and no jokers or wild cards. Ace is listed twice as it can be either low or high in straights, but not both in the same hand. See if you get the same counts as I do. As a warm-up, the number of possible hands is $\binom{52}{5} = 2,598,960$.

royal straight flush 10, J, Q, K, A of the same suit: 4 ways —The only choice is the suit.

straight flush 5 consecutive cards of the same suit, minus the royal straight flush: 36 ways. —9 choices for starting card, A–9, times 4 choices for suit.

four of a kind 624 ways. —Choose all four cards of one number, then any other card: $13 \cdot 48$.

full house Three of one card, two of another: 3,744 ways. —Choose the card number for the triple, then 3 suits, then the number for the pair, and two suits: $13\binom{4}{3} \cdot 12\binom{4}{2}$.

flush All five cards of the same suit, minus all straights: 5,108 ways. —Choose one suit, then 5 of the 13 cards for $4 \cdot \binom{13}{5}$, minus all 40 straight flushes.

straight 5 consecutive numbered cards of any suit, minus all flushes: 10,200 ways. —10 choices for starting card, A–10, times 4^5 choices for suit, minus all 40 straight flushes.

triple 54,912 ways. —Choose the number for 3 suits, then the suits, then two of the remaining 12 numbers and their suits: $13 \cdot \binom{4}{3} \cdot \binom{12}{2} \cdot 4^2$.

two pair 123,552 ways. —Choose the lone number, its suit, then the two numbers for pairs, and their two suits: $13 \cdot 4 \cdot \binom{12}{2}\binom{4}{2}^2$

single pair 1,098,240 ways. —Choose the number for the pair, then its suits, then the three remaining numbers and suits: $13\binom{4}{2}\binom{12}{3} \cdot 4^3$.

none of the above 1,302,540 ways. —Choose any five distinct card numbers, $\binom{13}{5}$, except the 10 straights, and independently choose any sequence of five suits, 4^5, except the 4 flushes, so $(\binom{13}{5} - 10)(4^5 - 4)$.

◀

Exercise 4.4. Find the mistake(s) in each of the following.*

*__Warning:__ incorrect statements in this problem!

1. Given sets A and B with elements from the universe U, to show that $A \subseteq B$, we must show that $\exists_{x \in U}\ (x \in A) \wedge (x \in B)$.

   There are two mistakes here. The quantifier from the definition of subset is $\forall$, not $\exists$, and the operation should be $\rightarrow$, not $\wedge$. This answer claims that for A to be a subset of B, it is enough for there to be a single element that is in both sets. But this is false for $A = \{1, 2\}$ and $B = \{2, 3\}$; the element 2

is in both A and B, but A is not a subset of B. The correct definition for $A \subseteq B$ says $\forall_{x \in U}\ (x \in A) \to (x \in B)$.

2. Given sets A and B, to show that $A \subseteq B$, we must show that for all x, both $x \in A$ and $x \in B$.

   This statement says that A and B both contain every element in the universe, which is probably not what was intended (although the universe does satisfy the definition for being a subset of itself). What is needed is the conditional: if $x \in A$ then $x \in B$. For elements of the universe that are not in A the statement is trivially true, so we don't care if they are in B or not. Elements in A, must be in B, as desired.

3. Since $a \cdot 1 = a$ for all integers a, if we know $b \cdot m = b$, where b and m are integers, then $m = 1$.

   A counterexample proves this wrong: We can have $b = 0$ and $m = 2$.

   The problem arises because in a statement like, "if we know $b \cdot m = b$, where b and m are integers," then b and m are specific integers that someone else has chosen for us. The statement "$a \cdot 1 = a$, for all integers a" applies only if either b or m equals one, but we have no control over that.

   If we were told that, "For all integers b, $b * m = b$," then we could use this as follows: Since this works for all integers, choose $b = 1$, giving $1 \cdot m = 1$. But we also know that $1 \cdot m = m \cdot 1 = m$, by substituting m for a, so $m = 1$.

◀

Puzzle 4.6. Jack Palmer's 1924 jazz lyric says, "Everybody loves my baby, but my baby don't love nobody but me."

1. If we think the double negative is for emphasis, we might say, "Everybody loves my baby, but my baby doesn't love anybody but me." Translate this into an expression quantified over a set P of people with my baby $b \in P$ and me $m \in P$. Denote a loves b by the predicate $\ell(a, b)$.

   $(\forall_{x \in P}\ \ell(x, b)) \wedge (\nexists_{y \in P}\ (y \neq m) \wedge \ell(b, y))$

2. Show that this leads to the narcissistic conclusion, "I am my baby."

   By simplification, we can split the statement into the left, $\forall_{x \in P}\ \ell(x, b)$, and as an equivalent of the right, $\forall_{y \in P}\ (y = m) \vee \overline{\ell(b, y)}$.

   Substitute b for x in the left by universal instantiation. We learn $\ell(b, b)$.

   Substitute b for y in the right to obtain $(b = m) \vee \overline{\ell(b, b)}$. In the conjunction, $((b = m) \vee \overline{\ell(b, b)}) \wedge \ell(b, b)$, distribute, then recognize and eliminate the contradiction to get $(b = m) \wedge \ell(b, b)$, which means that I am my baby and I love myself.

3. What genre of music best fits the original lyric?

   The original lyric might fit a country music song: the right side translates as $\nexists_{y \in P+} (y \neq m) \wedge \overline{\ell(b,y)} \equiv \forall_y (y = m) \vee \ell(b,y)$, which says that my baby loves everyone who isn't me, and I'm not sure if s/he loves me.

4. How might you write the lyric to convey the intended meaning?

   Perhaps the lyric should have said, "my baby loves nobody else but me," which could translate to $\forall_{y \in P} ((y = m) \vee (y = b) \vee \overline{\ell(b,y)})$.

◀

Exercise 5.4. Here are two statements that are true for a family of sets $\mathcal{A}$ that partitions a set S. They are very close to the definition, but are different, because they are also true of some families that are not partitions. For each, give an example of a set S of integers and a family that is not a partition of S for which the statement remains true.

1. Every element $x \in S$ is in exactly one set of the family $\mathcal{A}$. In notation,

$$\forall_{x \in S} \left(\exists_{B \in \mathcal{A}} (\forall_{C \in \mathcal{A}} (x \in B) \wedge (x \in C \to B = C)) \right).$$

   Simple example: $S = \{1\}$ and $\mathcal{A} = \{\{1,2\}\}$. This statement allows elements not in S into sets of the family $\mathcal{A}$.

2. Element x is in S if and only if it is in exactly one set of the family $\mathcal{A}$:

$$\forall_{x \in U} \left(x \in S \leftrightarrow (\exists_{B \in \mathcal{A}} \forall_{C \in \mathcal{A}} (x \in B) \wedge (x \in C \to B = C)) \right).$$

   Simple example: $S = \{1\}$ and $\mathcal{A} = \{\{1,2\},\{2\}\}$. This statement says that elements not in S can't appear in only one set of the family $\mathcal{A}$; we want them not to appear in any.

◀

Puzzle 5.5. Updating a Charles Dodgson (Lewis Carroll) puzzle: in a particularly aggressive paintball game with 20 combatants, 85% got hit in a leg, 80% in an arm, 75% upside the head, and 70% in the facemask. What is the minimum number of combatants that were hit in all four places?

2, or 10%

Maximum?

14, or 70%

This is related to inclusion/exclusion counting, and can be solved by sprinkling people into a Venn diagram to satisfy the hit counts: 17 Leg + 16 Arm + 15 Head + 14 Mask = 62 total.

The maximum is easy: it can't exceed the number hit in any part, so 14 LAHM. You have six left and need 3 L, 2 A, 1 H. Some could double or triple up on hits to leave others unscathed.

On the other hand, if 20 people get three hits each, you'd be two short, so at least two were hit LAHM. And you can achieve this minimum: 2 LAHM + 6 LAH + 5 LAM + 4 LHM + 3 AHM = 20 people.

◀

Exercise 6.6. Build bijections to demonstrate the sum and product rules for finite sets, as stated in subsection 3.2.1:

\+ *sum rule*: For disjoint, finite sets A and B, $|A \uplus B| = |A| + |B|$.

Assume that we are given bijections $f_A : [1..|A|] \to A$ and $f_B : [1..|B|] \to B$. Define a function $g : [1..|A| + |B|] \to A \uplus B$ that is a bijection.

$$g(i) = \begin{cases} f_A(i) & 0 < i \leq |A|, \\ f_B(i - |A|) & |A| < i \leq |A| + |B|. \end{cases}$$

We know this is a bijection because it has an inverse that is also a function:

$$g^{-1}(y) = \begin{cases} f_A^{-1}(y) & y \in A, \\ |A| + f_B^{-1}(y) & y \in B. \end{cases}$$

How does this use "disjoint?" Since an element is in A or B, but not both, the inverse assigns a unique value to each $y \in A \uplus B$.

How does this use "finite?" To add $|A|$, set A at least must be finite.

× *product rule*: For finite sets A and B, $|A \times B| = |A| \cdot |B|$.

Again, assume that bijections $f_A : [1..|A|] \to A$ and $f_B : [1..|B|] \to B$ are given. Let's build the inverse, $g^{-1} : A \times B \to [1..|A| \cdot |B|]$, because to build the function itself takes ceiling and mod functions, which are introduced in the next chapter.

Actually, I will include the function as well, but ignore it until you are familiar with ceiling and mod.

$$g^{-1}(a, b) = f_A^{-1}(a) + |A| \cdot (f_B^{-1}(b) - 1),$$

$$g(n) = \left(f_A(1 + ((n-1) \bmod |A|)),\ f_B\left(\left\lceil \frac{n}{|A|} \right\rceil\right)\right).$$

◀

Exercise 6.7. Let $(a_1, a_2, \ldots, a_n)$ be a sequence from $[1..n]$ with no repeated number. Show that if n is odd, then the product $\prod(a_i - i)$ is even.

By the pigeonhole principle, not all odd a_i can go to even positions i, so some term $(a_i - i)$ must be even.

◀

Extension 6.10. A logic function with n variables has domain $\{T, F\}^n$ and range $\{T, F\}$. In other words, for each possible way to input an n-tuple of T and F, it must choose a T or an F.

1. How many different logic functions are possible on two input variables? (and, or, xor, if, iff are five familiar ones.)

   16

2. How many different logic functions are possible on n input variables?

   2^{2^n}

3. How many of the logic functions with two input variables actually use both? For example, $f(p, q) = (p \wedge q) \vee (p \wedge \overline{q})$ does not really depend upon q.

   10

4. What is a good definition for a logic function on n variables *using the ith variable*?

   Function $f : \{T, F\}^n \to \{T, F\}$ *uses variable* v_i iff there is some assignment to the variables $v_1, v_2, \ldots, v_n$ so that $f(v_1, v_2, \ldots, v_n) \neq f(v_1, v_2, \ldots, v_{i-1}, \overline{v_i}, v_{i+1}, \ldots v_n)$.

5. How many of the logic functions with three input variables actually use all three?

   218

6. Determine a formula for counting the number of logic functions with n input variables that use all n.

   Here is an inclusion/exclusion formula that starts with all functions, then, for each variable, removes all functions that do not use that variable. Functions not using a pair of variables have been removed twice and need to be added back, then. . . , until we finally reach the $2^{2^0} = 2$ functions that do not use any of the n variables, but are all T or all F. The final count is $\sum_{0 \le i \le n} (-1)^i \binom{n}{i} 2^{2^{n-i}}$.

◀

*__Warning:__ incorrect statements in this problem!

Exercise 7.2. Find the mistake(s) in each of the following. *

1. In reasoning about a number d that is a multiple of 9, a student writes, "$9|d = \lfloor d/9 \rfloor \ldots$".

   There is a type conflict, and probably a misuse of '=': $9|d$ is a true/false statement and $\lfloor d/9 \rfloor$ is an integer so they cannot be equal. It would make sense to say $9|d$ iff $d/9 = \lfloor d/9 \rfloor$. Here the '=' is testing numerical equality and the 'iff' is for the equivalence of two logical statements.

2. The negation of "n is not divisible by any prime number between 1 and $\sqrt{n}$" is "n is divisible by any prime number between 1 and $\sqrt{n}$."

   The correct negation is, "There exists a prime between 1 and $\sqrt{n}$ that divides n." Putting quantifiers first helps us determine the right negations.

◀

Exercise 7.4. Partition a set of cardinality n (i.e., n elements) as evenly as possible into k subsets. Use floor, ceiling, and mod to say how many sets you get of what cardinalities.

n mod k of $\lceil n/k \rceil$ and the rest of $\lfloor n/k \rfloor$. Note that you can get 0 of the larger and k of the smaller, but cannot get the reverse. Where does this asymmetry come from?

◀

Exercise 7.5. $x - \lfloor x \rfloor$ is the *fractional value of* x and $x - \text{sgn}(x)\lfloor |x| \rfloor$ is the *fractional part of* x. Explain what each of these is in words. Use parallel language to make the similarities and differences stand out clearly.

The *fractional value* is the non-negative number for how much greater x is than its floor, the greatest integer less than x. The *fractional part* is the non-negative number that represents all digits behind the decimal point for x. For $x \geq 0$ the fractional value and fractional part are the same, but for $x < 0$ they are the same if x is a multiple of one half, otherwise they are different and sum to 1.

◀

Exercise 7.8.

1. Give a combinatorial proof of the subcommittee identity $\binom{n}{k}\binom{k}{j} = \binom{n}{j}\binom{n-j}{k-j}$.

   The left side counts the number of ways to choose a committee of k from n people, with a subcommittee of size j chosen from the k. The right side counts the same quantity by first chosing the j for the subcommittee from n people, and then choosing the remaining $k-j$ for the committee from the remaining $n-j$ people.

2. Show that, except for the ones, any two numbers from a row of Pascal's triangle have a common factor. That is, show for integers $0 < r, s < n$, that $gcd(\binom{n}{r}, \binom{n}{s}) > 1$.

   Suppose that $0 < j < k < n$, and use the subcommittee identity $\binom{n}{k}\binom{k}{j} = \binom{n}{j}\binom{n-j}{k-j}$. Since $\binom{n}{j}$ divides the right, it also divides the left side. But since $k < n$, $\binom{k}{j} < \binom{n}{j}$, so there must be a common factor of $\binom{n}{k}$ and $\binom{n}{j}$.

Hint:

◀

Exercise 7.9. Let $p(x) = \sum_{0 \le i \le n} a_i x^i$ be a polynomial with integer coefficients $(a_1, a_2, \ldots, a_n)$ and degree greater than one. That is, $n \ge 1$ and $a_n \neq 0$. Show that there exists a non-negative integer k so that $p(k)$ is not prime.

Notice that for integers $m > 0$, $p(ma_0)$ has every term divisible by a_0. If any $p(ma_0)$ is not prime, then we are done. So we need only worry about the case that $p(0) = a_0$ is prime, and show that for some m, $p(ma_0) \neq a_0$. But this is true, because polynomial $p(x) - a_0$ has at most n roots.

◀

Extension 7.16. For these questions, let $[x]$ denote the *fractional part of* x: that is, $[x] = x - \lfloor x \rfloor$. We are going to look at the set of all fractional parts, $F_a = \{[na] \mid n \in \mathbb{N}\}$. Show the following:

1. For any irrational number a, the elements we put into set F_a are distinct. That is, for all positive integers $m, n \in \mathbb{N}$, we have $[ma] = [na]$ iff $m = n$.

   $[ma] - [na] = (m-n)a - \lfloor ma \rfloor + \lfloor na \rfloor = 0$ iff $(m-n) = 0$, because otherwise a would be rational, namely $a = (\lfloor ma \rfloor - \lfloor na \rfloor)/(m-n)$.

2. For any irrational number a, the set $\inf F_a = 0$. That is, for any $\epsilon > 0$, there is an $n \in \mathbb{N}$ with $[na] < \epsilon$.

   Hint:

   Given any $\epsilon > 0$, choose $M \in \mathbb{N}$, so $1/M < \epsilon$. Divide the interval $[0, 1]$ into M subintervals of length $1/M$. By the pigeonhole principle, there must be two distinct integers $m, n \in [1..M+1]$ so that the fractional parts $[ma]$ and $[na]$ are in the same subinterval; that is, $0 < [ma] - [na] \le 1/M$. Expanding, we see that $\lfloor ma \rfloor - \lfloor na \rfloor < (m-n)a \le \lfloor ma \rfloor - \lfloor na \rfloor + 1/M$, so $[(m-n)a] \le 1/M < \epsilon$.

3. For any irrational number a, the set F_a is *dense* in the interval $[0, 1]$. That is, for any reals $x \in [0, 1]$ and $\epsilon > 0$, there is an $n \in N$ such that $|x - [na]| < \epsilon$.

   For a given $\epsilon > 0$, the previous solution finds $M, k \in \mathbb{N}$ satisfying $0 < [ka] \le 1/M < \epsilon$. If we again divide $[0, 1]$ into subintervals of length $1/M$, we find that each subinterval contains at least one number from $\{i[ka] \mid i \in \mathbb{N}\}$.

   The given x lies in some subinterval, and is at most $1/M < \epsilon$ away from any of the numbers of $\{i[ka] \mid i \in \mathbb{N}\}$ that lie in the same subinterval.

◀

Exploration 7.19. How many simple substitution cyphers on [a..z]? A simple substitution cypher replaces each letter a–z with some other letter in an invertible manner – it is a bijection from a–z to a–z. You may have seen Cryptoquote in the newspaper, for which the standard example is AXYDLBAAXR = LONGFELLOW (i.e., A→L and X→O.)

1. How many different simple substitution cyphers are there?

   26!

   —You have 26 choices for the first letter, 25 for the next, ..., since you may not choose a letter twice if you want to be able to decode the message. In other words, counting the bijections is just counting the possible permutations of the letters. Of course, this includes undesirable cyphers that leave some or all letters unchanged.

2. Suppose that we want to avoid sending a letter to itself, so we replace letters from the first half (a–m) with letters from the second half (n–z) and vice versa. With this restriction, how many different simple substitution cyphers are there?

   13! * 13!

   —This is like independently permuting the first and second halves of the alphabet, so the numbers multiply. Note that this much smaller than 26!.

3. Counting the number of substitution cyphers in which no letter maps to itself is not easy, but you can take a swing at that, too.

   The number of cyphers that map no letter to itself is less than the number of permutations, 26!, but more than 25!. It is the number of *derangements*, which I will denote $D(26)$.

   It is not hard to come up with a general recurrence for derangements of $[1..n]$, if we keep numbers and positions separate. Let a_i be the number in position i, which in a derangement cannot be i. Find the position j with $a_j = n$.

   If $a_n \neq j$, then throw out n at position j, move all numbers at positions greater than j down one, and subtract one from each number greater than j. Numbers at positions less than j are either left alone or decrease by one if they were greater than j, so they remain out of place. Numbers ending at positions j or greater are either less than j or both moved one position and decremented, so they are also out of place. We thus have a derangement of $[1..n-1]$.

   On the other hand, if $a_n = j$, then discard numbers and positions j and n, subtract one from all numbers greater than j, and you are left with a derangement of $[1..n-2]$.

   These manipulations are reversible, and each smaller derangement gives a different derangement of $[1..n]$. Thus, $D(n) = (n-1)(D(n-1) + D(n-2))$, with base cases $D(0) = 1$ and $D(1) = 2$.

   Here are two simpler expressions for this count: $D(n) = n! \sum_{0 \leq i \leq n} \frac{(-1)^i}{i!}$ and, for $n > 0$, $D(n) = \left\lfloor \frac{n!}{e} + \frac{1}{2} \right\rfloor$.

◀

Exercise 8.3. Find the mistake(s) in each of the following.*

*__Warning:__ incorrect statements in this problem!

1. Let $S(n) = \sum_{n=1}^{n} n \ldots$

   The sum uses n as both the summation variable and the upper limit, making this highly ambiguous. We can't even tell if $S(n) = \sum_{i=1}^{n} n$ or $S(n) = \sum_{i=1}^{n} i$ was intended without context.

2. For even $n \geq 0$, the double factorial $n!!$ is the product of all even numbers in $[1..n]$. For odd $n \geq 1$, the double factorial $n!!$ is the product of all odd numbers in $[1..n]$. Recursively define the double factorial for non-negative integers:

   Base: $0!! = 1$.
   Rec. Rule: for integers $n \geq 1$, $n!! = (n-2)!! \cdot n$.

   Positive odd integers never reach the base case. To complete the definition, we need to add $1!! = 1$ and have the recursive rule apply for $n \geq 2$.

3. We can recursively define the language S of all strings that have the same number of as and bs:

   Base: $\Lambda \in S$.
   Rec. Rule: if string $\sigma \in S$, then the strings $ab\sigma$, $ba\sigma$ $a\sigma b$, and $b\sigma a$ are in S.

   There are two problems, here. First, this is missing a closure rule, so the set of all strings $\{a, b\}^*$ satisfies the Base and the Rec. Rule, but also contains strings with more as than bs. Second, and more subtle, if we added the Closure rule, that the only strings in S are those derived from the base case by a finite number of applications of the recursive rule, then there are strings with the same number of as and bs that are not in S: $aabbbbaa$ is one example.

   It is possible to recursively define the set S of strings with the same number of as as bs if you change strings in the middle, and not only at the ends.

   Base: $\Lambda \in S$.
   Rec. Rule: $\forall a \in S$, if we write $a = \beta\gamma$ with $\beta, \gamma \in \{a, b\}^*$, then $\beta ab\gamma \in S$ and $\beta ba\gamma \in S$.
   Closure: The only strings in S are those derived from the base case by a finite number of applications of the recursive rule

◀

Exercise 9.6. In section 4.2, item A6 suggested that you could show that "there are no lock or unlock operations in the trace" is equivalent to, "every operation in the trace is an access," if you added the condition that "each trace entry records exactly one of the operations $\{a, l, u\}$ applied by one process to one file." Here are the two statements, using the notation from that section:

$$\forall_i \, \forall_p \, \forall_f \, ((t_i \neq l(p,f)) \wedge (t_i \neq u(p,f))),$$

$$\forall_i \, \exists_p \, \exists_f \, (t_i = a(p,f)).$$

1. Write an expression for the added condition

$$\forall_i \, \exists_o \in \{a, l, u\} \, \exists_p \, \exists_f \, (t_i = o(p,f)) \\ \wedge \left(\forall_{o',p',f'}(t_i = o'(p',f')) \rightarrow (o{=}o' \wedge p{=}p' \wedge f{=}f')\right)$$

2. Assuming the added condition is true, prove that the two statements are equivalent.

   Assume that an adversary has chosen the time i, and consider t_i. **((todo. finish))**

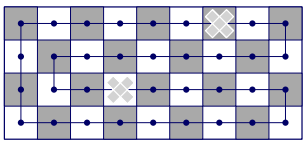
◀

Puzzle 9.7. Show that no set of 9 consecutive integers can be partitioned into two subsets such that the product of the elements in the first set is equal to the product of the elements in the second set.

You can't use zero, because then only one product would be zero. You can't use negatives, because one set will have even cardinality and the other odd. You need an even number of each prime factor, but intervals that start before 20 contain single factors of either 19, 13, or 7. Finally, five or more numbers of at least 20 multiply to greater than 32×10^5, and four or fewer at most 30 multiply to less than 81×10^4, so they cannot be equal.

◀

Puzzle 9.8. I give you a rectangle of size $n \times m$, where both $n, m > 1$ and their product mn is even, and enough dominoes (1×2 tiles) to cover it. I remove the squares at (x_1, y_1) and (x_2, y_2). Prove that you can cover the other squares with dominoes iff $x_1 + y_1 + x_2 + y_2$ is odd. (What type of proof do you use?)



The proof of this is beautiful if you do two things: First, color the grid with a checkerboard pattern; even $x + y$ get one color, and odd the other. Observe that $x_1 + y_1 + x_2 + y_2$ is odd iff the two removed squares have opposite colors. Second, find a path that goes square to square, visiting each square

once and returning to its start. Now, if you remove squares of opposite colors, you can cover the rest of the board by placing dominoes along the two fragments of the path. And if you remove squares of the same color, no-one can cover the rest of the board, since you are left with more of one color than the other and any domino covers one square of each color.

◀

Exercise 10.1. Can you show that the first n odd integers sum up to n^2?

We want to show that $1 + 3 + 5 + \cdots + 2n - 1 = n^2$, except this formula is not quite right at $n = 0$.

Stating it correctly with a recursive definition instead of ellipses: Let $S_0 = 0$, and for integers $n > 0$ define $S_n = S_{n-1} + 2n - 1$. Prove by induction that for all $n \geq 0$, $S_n = n^2$. Summation notation also works: show for all $n \geq 0$ that

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

I urge you to follow the template; there are many example proofs on the web that are not always careful about where they start, or about use of n and k, or that they are proving a 'for all' statement.

We have to prove that $S_n = 1 + \cdots + (2n - 1) = n^2$. We prove this by induction on n.

Base case: $n = 0$, the sum of zero numbers = 0, and $0^2 = 0$.)

Induction step: For a given $n > 0$,

Induction hypothesis: $\forall 0 \leq k < n, S_k = k^2$

We have to prove $S_n = n^2$:

$$\begin{aligned} S_n &= S_{n-1} + (2n - 1) && \text{Defn of } S_n \\ &= (n - 1)^2 + 2n - 1 && \text{Using IH with } k = n - 1, \\ &= n^2 - 2n + 1 + 2n - 1 && \text{By algebra} \\ &= n^2. \end{aligned}$$

So, by induction, $\forall n \geq 0, 1 + \cdots + (2n - 1) = n^2$.

QED

◀

Exercise 10.2. What is the sum of the first n squares? For all integers $n \geq 0$, prove that $1^2 + 2^2 + 3^2 + \cdots + n^2 = n(n + 1)(2n + 1)/6$. In summation notation, show that

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

Try to use the induction template to make sure you handle the ns and ks in a correct manner. There is nothing magical about those variable names, but good habits help us to be correct without thinking hard.

Let $S_n = \sum_{0 \le i \le n} i^2$. I want to prove, for all $n \ge 0$, that $S_n = n(n+1)(2n+1)/6$ by induction on n.

Base: $n = 0$: $S_0 = 0 = 0 \cdot 1 \cdot 1/6$ ✓

IS: for a given $n > 0$, I assume:

IH: for $0 \le k < n$, $S_k = k(k+1)(2k+1)/6$.

Now, we can expand the definition of S_n, apply the IH with $k = n-1$ and simplify.

$$\begin{aligned} S_n &= S_{n-1} + n^2 \\ &= (n-1)n(2(n-1)+1)/6 + n^2 \\ &= (n(n-1)(2n-1) + 6n^2)/6 \\ &= (n/6)((n-1)(2n-1) + 6n) \\ &= (n/6)(2n^2 - 3n + 1 + 6n) \\ &= (n/6)(2n^2 + 3n + 1) \\ &= (n/6)(n+1)(2n+1) \end{aligned}$$

QED.

◀

Exercise 10.6. What is the value of the binary number that is a string of n ones? Define $S_0 = 0$ and for $n > 0$, define $S_n = 2S_{n-1} + 1$. Show that $S_n = 2^n - 1$.

Do an induction to determine a closed form expression for the value for this sequence that is defined recursively. An expression in *closed form* uses a fixed number of standard operations, like arithmetic, factorial, binomial coefficients, but no summation or recursion in which the number of operations grows as n grows.

I will use induction on n.

1. Base $n = 1$, and $S_1 = 2S_0 + 1 = 2 \cdot 0 + 1 = 1 = 2^1 - 1$.

2. Induction hypothesis: for a given $n > 1$, $S_{n-1} = 2^{n-1} - 1$. To prove this statement, I have only to prove, $S_n = 2^n - 1$.

3. By definition, $S_n = 2S_{n-1}+1 = 2\cdot(2^{n-1}-1)+1 = 2\cdot 2^{n-1}-2+1 = 2^{(n-1+1)}-1 = 2^n-1$.

4. QED

◀

Exercise 10.10. Can euro 2 and 5 cent coins make any value greater than 3? For all $n \ge 4$, show that there exist non-negative integers h and k such that $2h + 5k = n$.

Warning: Be careful not to re-use k in the induction proof; you'll want to rename either this variable or the one in the induction template.

We prove by induction on n that, for all $n \ge 4$, there exist non-negative integers h and k such that $2h + 5k = n$.

Base $n = 4$: choose $h = 2$, $k = 0$ to get $2h + 5k = n$.

Base $n = 5$: choose $h = 0$, $k = 1$ to get $2h + 5k = n$.

Ind Step: consider some $n > 5$, and assume IH: for all $4 \leq j < n$, there exist non-negative integers h' and k' such that $2h' + 5k' = j$. We want to show that there exist non-negative integers h and k such that $2h + 5k = n$.

But for $n > 5$, we know that $n - 2 \geq 4$, so we can apply the induction hypothesis with $j = n - 2$ to show that there exist non-negative integers h' and k' such that $2h' + 5k' = n - 2$. Let $h = h' + 1$ and $k = k'$, and observe that $2h + 5k = 2h' + 2 + 5k' = (n - 2) + 2 = n$. This proves the induction step.

By induction, we know the result for all $n \geq 4$.

◀

Exercise 10.12. Prove Lemma 7.3.1 by induction on the number of mediants created.

We prove these by induction on the number of mediants generated. We add that for all fractions a/b, we have $a, b \leq 0$ with not both equal to zero.

Base $n = 0$: the pairs $-1/0, 0/1$ and $0/1, 1/0$ satisfy the invariant.

Ind Step for $n > 0$: We want to prove that the invariant holds for all sequence containing n mediants, using as the induction hypothesis that they hold for all sequences containing $0 \leq k < n$ mediants.

Suppose that someone has given us a sequence with n mediants. Intervals without mediants were checked in the base case, so consider a mediant $(a + c)/(b + d)$ that was generated between a/b and c/d because they were adjacent in some sequence with $0 \leq k < n$ mediants. By the IH, we may assume that $bc - ad = 1$ and the denominators of a/b and c/d are positive except for possibly one zero. The reader can then check that $b(c + a) - a(b + d) = bc - ad = 1$ and $(b + d)c - (a + c)d = 1$, and that the denominator is positive for the mediant.

◀

Exercise 10.13. What is wrong with Polya's classic proof that all horses are the same color? I'd suggest rewriting this to follow my template and see if that clarifies what goes wrong.

CLAIM: In any set of h horses, all horses are the same color.

Faulty proof. by Induction on h.

Base: For $h = 1$. In any set containing just one horse, all horses clearly are the same color.

Induction Step: For $k \geq 1$ assume that the claim is true for $h = k$ and prove that it is true for $h = k + 1$.

Take any set H of $k + 1$ horses. We show that all the horses in this set are the same color. Remove one horse from this set to obtain the set H_1 with just k horses. By the induction hypothesis, all the horses in H_1 are the same color. Now replace the removed horse and remove a different one to obtain the set H_2. By the same argument, all the horses in H_2 are the same

color. Therefore, all the horses in H must be the same color, and the proof is complete. ∎

I copied from someone's online description rather than from the original source, here, so one of the things that is not good is the way it switches from h to k going from the base to inductive step. It would be better to stay with h, and simply say: "For $h \geq 1$ assume that the claim is true and prove that it is true for $h+1$. Take any set H of $h+1$"

The problem arises in assuming that, because the first subset of horses were all of the same color, that the second must be as well. It is easier to conclude this with groups of $k \geq 3$, where it is clear that $\exists h\ (H_1 \cap H_2 = h)$.

If we go backwards from the inductive step, to $k = 2$, we have only two horses. We examine both horses individually $H_1 = \{h_1\}$ and $H_2 = \{h_2\}$, but since we don't have a third horse to compare each to, these two need not be the same color.

Stated another way, the induction step proof works only for sets of at least $h = 3$ three horses, and the base case covers $h = 1$, so $h = 2$ is never properly covered. If we lived in a world where we could treat $h = 2$ as another base case (i.e., all sets of two horses are the same color) then we could indeed prove that all horses are the same color.

◀

Exercise 10.14. What is wrong with this induction proof that $a^n = 1$ always?

Claim: Let a be any positive number. For all positive integers $n \in \mathbb{Z}^+$, we have a $a^{n-1} = 1$.

Faulty proof. by induction on n

Basis: If $n = 1$, we have $a^{n-1} = a^0 = 1$. ✓

Inductive Step: We want to prove this for an integer $n > 1$.

Inductive Hypothesis: Assume, for all natural numbers $k < n$, that $a^{k-1} = 1$.

We now want to show that it is true for n by writing

$$a^{n-1} = \frac{a^{n-2} \cdot a^{n-2}}{a^{n-3}},$$

since $n-1 = (n-2)+(n-2)-(n-3)$. But if we use the inductive hypothesis twice, for $k = n-1$ and $k = n-2$, we can replace $a^{n-2} = 1$ and $a^{n-3} = 1$, so $a^{n-1} = 1 \cdot 1/1 = 1$. ∎

The first inductive step for $n = 2$ does not match with the base cases, because it needs base cases for $k = n-1$ and $k = n-2$, which are 1 and 0, and we gave the base case only for $n = 1$. We could fix this if we had a base cases for $n = 1$ and $n = 2$, and the induction started with $n = 3$ and the proof would work.

◀

Puzzle 10.18. Can you show that the number guessing game has a worst case of exactly $\lceil \log_2 n \rceil$ guesses?

I'm thinking of a natural number x between 1 and n. You can ask questions of the form, "Is x larger than a?" and I will tell you (truthfully) yes or no. Show that, in the worst case, $\lceil \log_2 n \rceil$ questions are necessary and sufficient to exactly determine my number.

For sufficiency, you may want to argue that you can round n up to the next power of 2 to make your proof easier: e.g., new $n = 2^{\lceil \log_2 n \rceil}$.

To show that this many are necessary is the harder direction. You'll want to imagine that you delay picking a number until forced to by the guesser's questions. Pick your answers to keep the maximum number of options open to you.

Let me rephrase this: I have a list of n consecutive numbers, and by asking a question I can split the list and determine which sublist contains x. I want to prove that $\lceil \log_2 n \rceil$ splittings are both necessary and sufficient to reduce to a single number.

Spelling out the quantifiers, I want to show two things: Sufficient: There exists a strategy that for all choices of a number x from a list of n will find x in at most $\lceil \log_2 n \rceil$ guesses, and Necessary: For any strategy, there exists some list of n numbers and choice of x that requires at least $\lceil \log_2 n \rceil$ guesses.

I'll prove each of these separately, by induction on n. For Sufficient, let's pick the strategy of dividing the list in half by our guess. Let G_n be the number of guesses by this strategy. I want to prove that for all $n > 0$, $G_n \leq \lceil \log_2 n \rceil$

Base $n = 1$: No guesses are needed; we know x. And $G_1 = 0 = \lceil \log_2 1 \rceil$.

Ind Step: For any $n > 1$, I get to assume the following IH: for all $0 < k < n$, we know $G_k \leq \lceil \log_2 k \rceil$. I want to prove that $G_n \leq \lceil \log_2 n \rceil$.

If n is even, then we can write $n = 2m$ and splitting in half gives two lists of size m. If n is odd, then $n = 2m - 1$ and splitting gives lists of size m and $m - 1$. Note that $\lceil \log_2 2m - 1 \rceil = \lceil \log_2 2m \rceil$, because the function $\lceil \log_2 \rceil$ increases after powers of two, which is an even to odd transition.

$$
\begin{aligned}
G_n &\leq 1 + G_m && \text{our guess + guess on larger half} \\
&\leq 1 + \lceil \log_2 m \rceil && \text{IH for } k = m \\
&= 1 + \lceil (\log_2 2m) - 1 \rceil && \text{property of log} \\
&= \lceil \log_2 2m \rceil && \text{property of ceil} \\
&= \lceil \log_2 n \rceil && \text{in both even and odd cases.}
\end{aligned}
$$

This completes the induction step. By induction, we now know that for all $n > 1$, $G_n \leq \lceil \log_2 n \rceil$.

For Necessary, we need to accept any strategy, but can pick x to make the strategy work hard. In fact, we delay picking, but keep a list of possible values and simply answer each question consistent with x being in the larger sublist. Let H_n be the maximum number of questions any strategy uses to pick one out of a list of n possible values. We show that $H_n \geq \lceil \log_2 n \rceil$, by induction on n.

Base $n = 1$: No guesses are needed; x is known. Fortunately $H_1 = 0 = \lceil \log_2 1 \rceil$.

Ind Step: For any $n > 1$, I get to assume the following IH: for all $0 < k < n$, we know $H_k \geq \lceil \log_2 k \rceil$. I want to prove that $H_n \geq \lceil \log_2 n \rceil$.

If a question is asked that is outside the list of remaining possible values, answer, but don't count it. A question inside the list will split the list into two pieces, one of size $m > 0$ and one of size $n - m > 0$; let's assume that $m \geq n/2$, so that is the larger piece. We can then apply the IH to bound H_m. The proof is almost the same as before, except for the inequalities being reversed.

$$\begin{aligned}
H_n &\geq 1 + H_m && \text{1+guess on remaining possibles} \\
&\geq 1 + \lceil \log_2 m \rceil && \text{IH for } k = m \\
&= 1 + \lceil (\log_2 2m) - 1 \rceil && \text{property of log} \\
&= \lceil \log_2 2m \rceil && \text{property of ceil} \\
&= \lceil \log_2 n \rceil && \text{in both even and odd cases.}
\end{aligned}$$

This completes the induction step. By induction, we now know that for all $n > 1$, $H_n \geq \lceil \log_2 n \rceil$.

◀

Puzzle 10.19. Suppose there are n identical cars stopped on a circular track, and that the total gas in all tanks is enough for one car to make it around the track (even if it had to stop and start n times.) Show that there is at least one car that can make it around if it siphons all the gas from each of the stopped cars it passes.

In any configuration with n cars and enough gas for one to make it around, there exists a car that can complete the circle I want to prove this by induction on n.

Base: $n = 1$: by assumption, the one car has enough gas to complete the circle.

Ind Step: For a given $n > 1$,

IH: I get to assume that for all $1 \leq k < n$, some car out of k cars can complete the circle. I want to show, for n cars, that one can complete the circle.

First, observe that some car, call it car 1, can drive to reach the next car, car 2. If not, then if we put the distances each car can drive together, we would not reach around the circle; there would be gaps just before each car. This would contradict our assumption that the total amount of gas is enough to go around at least once.

But now, we can create a smaller, $(n-1)$-car instance of the problem: put all cars in their initial positions except car 2, which we remove, giving its gas to car 1 in car 1's initial position. By induction, this smaller instance has a solution. But then the larger, n-car instance does too, because whichever car picks up car 1's gas will, on that gas alone, have enough to reach car 2 and pick up its gas, too. This completes the induction step. Induction shows that no matter how many cars n we start with, some car can complete the circle.

◀

Extension 10.24. Using properties from Subsections 7.2.1 and 7.2.2, prove the "Fundamental Theorem of Arithmetic" that every integer $n > 1$ can be

written as the product of one or more primes in an increasing sequence in exactly one way (up to order).

1. Initially drop the "in exactly one way," and just show that every integer $n > 1$ can be written as a product of primes.

2. Then prove the full theorem, perhaps by minimal counterexample.

For all $n > 1$, we can write n as the product of one or more primes
We prove this induction on n.
Base: n=2: 2 is a prime.
Induction Step: for a given $n > 2$, I want to prove that n is the product of one or more primes.
Induction Hypothesis: for all $2 \leq k < n$, I may assume that k is the product of one or more primes.
I want to show that n is a product of one or more primes
If n is prime, then n works
If n is composite, then $\exists a, b \neq 1$ so that $ab = n$.
Since $2 \leq a < n$ and $2 \leq b < n$, by the IH, a and b are products of one or more primes, so we know that n is a product of at least two primes.
Therefore, I have proved for all $n > 1$ that n is the product of one or more primes.

◀

Extension 10.26. Prove the *Pigeonhole Principle* by induction:That is, show that for all integers $n \geq 1$, for all sets A of $n + 1$ elements, all sets B of n elements, and all functions $f : A \to B$, there exist elements $a_1, a_2 \in A$, with $a_1 \neq a_2$, such that $f(a_1) = f(a_2)$.

We can prove this by induction on n.
For $n = 1$, set A has 2 elements (a_1 and a_2) and set B has 1 element. Thus, there exits only one function from A to B. This function maps both elements in A to the single element in B. Thus, $f(a_1) = f(a_2)$.
Induction step: For a given $n > 1$, we want to prove that for all sets A of $n + 1$ elements, all sets B of n elements, and all functions $f : A \to B$, there exist elements $a_1, a_2 \in A$, with $a_1 \neq a_2$, such that $f(a_1) = f(a_2)$.
Induction hypothesis: We can assume, for all $k < n$, that for all sets A of $k + 1$ elements, all sets B of k elements, and all functions $f : A \to B$, there exist elements $a_1, a_2 \in A$, with $a_1 \neq a_2$, such that $f(a_1) = f(a_2)$.
Assuming the induction hypothesis, we will prove that for a given $n > 1$, for all sets A of $n + 1$ elements, all sets B of n elements, and all functions $f : A \to B$, there exist elements $a_1, a_2 \in A$, with $a_1 \neq a_2$, such that $f(a_1) = f(a_2)$.
If there is another $x_2 \in A$ such that $f(x_1) = f(x_2)$ then we have found our two pigeons and are done. So suppose that x_1 gives $f(x_1)$ uniquely. But then we remove x_1 from A leaving n elements, $f(x_1)$ from B leaving $n - 1$, and both from f to make a new function $g : A - \{x_1\} \to B - \{f(x_1)\}$. By the induction hypothesis with $k = n - 1$, this new function g sends two elements $a_1, a_2 \in A - \{x_1\}$ to the same value

$g(a_1) = g(a_2) \in B - \{f(x_1)\}$. Since f does the same as g on a_1, a_2, this establishes the induction step.

Therefore, we have proved that for all $n \geq 1$, for all sets A of $n+1$ elements, for all sets B of n elements, and for any function $f: A \to B$, there exist elements a_1 and a_2 in A with $a_1 \neq a_2$ such that $f(a_1) = f(a_2)$.

◀

Extension 10.27. Prove *Fermat's little theorem*, Theorem 7.2.2, by induction on m: For all primes p and positive integers $m \in Z^+$, the exponential $m^p \bmod p = m \bmod p$.

Proof. We prove this picking any prime p, then doing induction on m.

Base $m = 1$: For all primes p, we see $1^p = 1 \equiv 1 \pmod{p}$.

Ind step: For $m > 1$,

IH: assume, for all $1 \leq k < m$, that $k^p \equiv k \pmod{p}$.

We want to show that $m^p - m \equiv 0 \pmod{p}$, that is, that $m^p - m$ is a multiple of p. Choose $k = m - 1$.

$$\begin{aligned} m^p &= (k+1)^p \\ &= k^p + \binom{p}{1}k^{p-1} + \binom{p}{2}k^{p-2} + \cdots + \binom{p}{p-1}k + 1. \end{aligned}$$

Putting all the binomial coefficients involving p on the right,

$$m^p - k^p - 1 = \binom{p}{1}k^{p-1} + \binom{p}{2}k^{p-2} + \cdots + \binom{p}{p-1}k,$$

we get that both sides of the equation are divisible by p, so $m^p - k^p - 1 \equiv 0 \pmod{p}$. But by the IH, $k^p \equiv k \pmod{p}$, so $m^p - k - 1 \equiv 0 \pmod{p}$, and $m^p \equiv m \pmod{p}$, as desired. ∎

◀

Exercise 12.3. Prove that subset ($\subseteq$) is the universal reflexive partial order. That is, if you are given a poset $(S, \preceq)$ with $\preceq$ being reflexive, then you can make a family of sets $\mathcal{F}_S$ and a bijection $f: S \to \mathcal{F}_S$ so that for all $a, b \in S$, $a \prec b$ iff $f(a) \subseteq f(b)$.

According to the hint, for each $b \in S$, define $f(b) = \{a \in S \mid a \prec b\}$. The family $\mathcal{F}_S = f(S)$, the set of all these sets.

We have two directions to prove: First, suppose $a \prec b$. Consider any $c \in f(a)$. By definition, $c \prec a$, so by transitivity $c \prec b$ and $c \in f(b)$. Thus $f(a) \subseteq f(b)$.

Second, suppose $f(a) \subseteq f(b)$. Since $\prec$ is reflexive, $a \in f(a)$. But then $a \in f(b)$, meaning $a \prec b$. ∎

◀

Exercise 12.5. You are a xenobiologist studying Zorches. When two Zorches get together, one or both may "freeb" (glow blue). Write aFb if a freebs when close to b. Some Zorches freeb near your spaceship, and you eventually realize that this is because they are seeing their own reflection, aFa.

A particular group of four Zorches, $Z = \{Aye, Bee, Cea, Dii\}$, change their freeb pattern every day, but it is always reflexive (each freebs with its reflection), symmetric (for any pair, either both or neither freeb), and transitive (if aFb and bFc then aFc). How many days can they go before repeating a freeb pattern? Let's break this into smaller questions.

1. How many elements are there in the Cartesian product $Z \times Z$?

   4*4=16

2. How many different relations from Z to Z are possible?

   $2^{16} = 65,536$, since each pair in $Z \times Z$ is independently either in or out of the relation.

3. How many different reflexive relations from Z to Z are possible?

   $2^{4 \cdot 3} = 2^{12} = 4096$, since for $x, y \in Z$, we know that (x, x) is in, but if $x \neq y$ then (x, y) may be in or out of the relation.

4. How many different reflexive and symmetric relations from Z to Z are possible?

   $2^{\binom{4}{2}} = 2^6 = 64$, since for $x < y \in Z$, once we decide if (x, y) is in, then we know whether (y, x) is in the relation, too.

5. How many different equivalence relations from Z to Z are possible?

   There are 15. Because equivalence relations partition the underlying set into equivalence classes that freeb together, we can count partitions of the four Zorches:

   - 1 way to partition into 4 sets of one,
   - $\binom{4}{2} = 6$ ways to partition into 3 sets by choosing which two freeb,
   - $\binom{3}{1} = 3$ ways to partition into 2 sets of two by choosing who freebs with "Aye,"
   - $\binom{4}{1} = 4$ ways to partition into sets of one and three by choosing who doesn't freeb,
   - 1 way to have one set in which every Zorch freebs.

◀

Exercise 13.2. In an undirected graph $G = (V, E)$, define the relation "is reachable from" on vertices: $\forall a, b \in V$ we say that $a \; R \; b$ iff there exists a path starting at b that ends at a. Prove that R is an equivalence relation: that is, it is reflexive, symmetric, and transitive. What are the equivalence classes of R in graph terminology?

Recall that a path is a sequence of one or more vertices, in which adjacent vertices are joined by edges. We can see that the "reachable" relation is

Reflexive: For all $a \in V$, vertex a is reachable from a by the 0-edge path (a), so R is reflexive.

Symmetric: For all $a, b \in V$, if $a \; R \; b$, then there is a path $(b, v_1, v_2, \ldots, v_k, a)$. We can simply reverse the path, $(a, v_k, \ldots, v_2, v_1, b)$, to show that $b \; R \; a$, so R is symmetric.

Transitive: For all $a, b, c \in V$, if $a \; R \; b$ and $b \; R \; c$ then there are paths $(b, v_1, v_2, \ldots, v_k, a)$ and $(c, w_1, w_2, \ldots, w_\ell, b)$. We join these to form a path $(c, w_1, w_2, \ldots, w_\ell, b, v_1, v_2, \ldots, v_k, a)$ that shows $a \; R \; c$, so R is transitive.

The equivalence classes are the connected components.

◀

Exercise 13.4. In the *Max in a list* algorithm of section 11.2, show that an algorithm that performs fewer than $n - 1$ comparisons will return the wrong maximum on some list. Consider the graph whose vertices are list entries with an edge between two vertices iff their list entries are directly compared.

Form a graph G with n vertices representing the n array entries and an edge joining two entries whose values are compared. If G has fewer than $n - 1$ edges, then G is not connected.

Now, if we choose *mi* as the array entry with maximum value *mx*, the adversary can find a component that does not contain the vertex for *mi*, and increase all values in that component by the same amount so at least one value is greater than *mx*. All comparisons performed by our algorithm on these new values produce the same results as before—all comparisons are between unchanged values or values that have the same amount added—so our algorithm must again report *mi*, even though $A[mi]$ is no longer the maximum value in the array.

◀

Puzzle 13.7. Prove the *Futurama theorem* [13], which writer Ken Keeler created for the Season 6 episode, "The Prisoner of Benda:" Professor Farnsworth and Amy have created a mind-swapping machine. After using it on several characters, they discover that no pair of minds can survive a second swap, although two minds can swap back with the help of some fresh bodies.

If n people are in the wrong bodies because of some unknown set of swaps. how many helpers are needed to sort everyone out? (Convince yourself that one extra body is not enough.) Keeler's Futurama theorem says that you never need more than two. Describe how, and prove that your method works.

You'll want to begin by assigning names to bodies and minds – defining notation can help clarify questions. Since bodies are put into the machine, I suggest numbering bodies so that body i wants mind i, but initially contains mind m_i.

Start by arranging bodies so that body m_i is just ahead of body i. Note that this puts bodies into one or more cycles so that each mind wants to move into the body just ahead; we can fix each cycle with the same two helper minds, A and B, as long as they have not swapped with each other or with any minds in the cycles.

Break a cycle between the lowest numbered body and its mind, so that the body, which we'll call 1, is facing the front and mind, m_1, is at the back, facing the whole line. Have body A swap with the front body, whose mind wants to be last. Have body B swap with the body at the back, then continue swapping forward to body 2, whose mind wants to be first. This puts minds 2-k into the right bodies. Now, swap body A with body 2 and body B with body 1, which fixes the cycle, leaving the minds that were in A and B in opposite bodies without directly swapping them yet. Fix all cycles, and if there was an odd number, conclude with a final swap of A and B to return everyone to their original bodies.

◀

Exploration 13.9. In one classic peg puzzle, you have a line of ten holes that start with 4 blue pegs on the left, two empty holes, then 4 blue pegs on the right. A peg may move into an empty hole that is adjacent, or by jumping over a single peg of the opposite color, but blue pegs must move right and red pegs must move left. The goal is to get the pegs to swap: all 4 red at left and all 4 blue at right.

First, if we were to build a graph to capture all possible arrangements and the movements between them, what would we find?

- How many different arrangements of colored pegs in holes are possible?
  C(10, 2)*C(8, 4)=3150
- How many of these arrangements have a pair of reds blocking a pair of blues?
  7*C(6,2)*C(4,2)=630
  There are 7 places to place the blocking four, then choose positions for the remaining two of each color. There are other ways to block, such as having two blocking one at either end, in $2\binom{7}{3}\binom{4}{2} = 420$.
- In any given arrangement, what is the maximum number of possible moves?
  4
  2 per hole (where each may be a jump or step).

- How many jumps will have to take place to go from the initial arrangement to the solution, and how many single step moves?
  Each peg needs to move 6 in total, but for each pair of a red and a blue, one of them will get the bonus of jumping the other, so the $6 \cdot 8 - 4^2 = 8 \cdot 4 = 32$ moves will have 16 jumps and 16 steps.

So the graph here would be too big to build by hand, but easily managed by computer. With the graph you could answer many questions: How many arrangements allow no moves? How many of those can you reach from the start position? How many arrangements allow the maximum number of moves? And, of course, is there a solution? ◀

Exercise 14.2. Pick an integer m at random from $[1..n]$. What is the probability you have a perfect square—that there exists $k \in Z$ such that $m = k^2$?

$\lfloor \sqrt{n} \rfloor / n$, since there are exactly $\lfloor \sqrt{n} \rfloor$ perfect squares in $[1..n]$.

◀

Exercise 14.4. Your friend holds out 7 cards, face down, and says 2 are hearts and 5 are spades. If you pick k cards, what are your chance of getting at least one a red card? Complete this table:

k	1	2	3	4	5	6	7
prob	2/7	11/21	5/7	6/7	20/21	1	1

There are several ways to do this, but the best it to recognize that the chance of getting no red is $\binom{5}{k}/\binom{7}{k}$, so the number we want is $1 - \binom{5}{k}/\binom{7}{k}$. This lets us complete the table as:

k	1	2	3	4	5	6	7
prob	2/7	11/21	5/7	6/7	20/21	1	1

◀

*A carnival employee, employed by the "house."

Exercise 14.5. Chuck-a-luck, or Birdcage, is a carnival gambling game whose barker* rolls three 6-sided dice for six players. Each player chooses one of the numbers 1–6, and receives \$1 for each die that shows their number, or pays \$1 if none of the dice does. Although the barker may advertise "three losers and three winners," giving the impression that the game is break-even, the house rakes in a significant percentage. How and how much? (You may want to list the possible outcomes: it helps to assume that the dice have three different colors.) What should the payouts be for the players to break even?

The best way to think of this is to have all six players pay \$1 up front. The payouts to players are then \$2 for one die showing the number, \$3 for two, and \$4 for three. Of the $6^3 = 216$ outcomes, in $6 \cdot 5 \cdot 4 = 120$ all three dice are different and the house pays \$6, in $6 \cdot 5 \cdot 3 = 90$ there is a pair and the house pays \$5, and in 6 there is a triple and the house pays \$4. Thus, the house expects to make

$90/216 + 2 \cdot 6/216 > 0.47$ cents on each game—a take of almost 7.9%. A possible break-even game could pay \$2 for one, \$4 for two, and \$6 for three.

◀