

OUTSIDE OF A DOG, A BOOK IS A MAN'S BEST FRIEND.

INSIDE OF A DOG, IT'S TOO DARK TO READ.

GROUCHO MARX

RICK NEFF

FIRST THREE ODDS

DOUBLE HALVE DIVIDE RECIPROcate

PUBLISHED BY THE AUTHOR

Contents

Preface	ix
0.1 In-Paired Vision	ix
0.2 More Double Vision	x
0.3 Autobiographical Aside	xi
0.4 Word Play Work Plan	xi
1 Introductions	3
1.1 Life is But a Dream	3
1.2 First Meeting	3
1.3 First Impressions	4
1.4 First Assignment	6
2 Sets and Logic	11
2.1 Set the Stage	11
2.2 Start with Propositions	12
2.3 Modeling Stuff	16
2.4 Exclusive and Inclusive	20
2.5 Rules to Build By	22
2.6 Truth Tables	26
2.7 Go Visual	28
2.8 Propositional Membership	31
2.9 Conditional Containment	33
2.10 Understanding Conditional Logic	38
2.11 Conditional Vocabulary	41
2.12 Logical Equivalence	45
2.13 Explanation and Implementation	49
2.14 Summary of Terms and Definitions	50
3 Functions	55
3.1 Encapsulation for Fun and Profit	55
3.2 A Plethora of Terms	57
3.3 Image and Preimage, Pre-In-Post Fix	61
3.4 Home Onto the Range	62
3.5 Mix and Match	63

3.6	Abstractions and Types	65
3.7	Representations	66
3.8	What's In A Name?	69
3.9	The Loop Free Way	70
3.10	Further Adventures in Functionology	74
3.10.1	Floor and Ceiling	74
3.10.2	Invert and Compose	76
3.10.3	Sequences and Strings	77
3.10.4	Arithmetic Progressions	78
3.10.5	Geometric Progressions	79
3.10.6	Summations	82
3.10.7	The Harmonic Series	89
3.11	Predicates	91
3.12	Quantifiers	92
3.13	Negating Quantifiers	95
3.14	Free and Bound	106
3.15	Summary of Terms and Definitions	116
4	Relations	121
4.1	Products and Subsets	121
4.2	Four Main Properties	123
4.3	Relational Databases	131
4.4	Equivalence Relations	136
4.5	Congruences	137
4.6	Arithmetic Modularity	139
4.7	Equivalence Classes and Partitions	140
4.8	Same Old Same Old	142
4.9	How Mathematicians Think	146
4.10	Summary of Terms and Definitions	147
5	Combinatorics and Probability	151
5.1	Frequently Subjective	151
5.2	Elementary and Basic	152
5.3	Perms and Combs	156
5.4	Choosing	160
5.5	Multichoosing	165
5.6	Basic Probability Theory	170
5.7	A Dicey Analysis	175
5.8	A Matter of Order	178
5.9	Summary of Terms and Definitions	183
6	Number Theory and Practice	185
6.1	Infinitely Many	185
6.2	Divisibility Redux	186
6.3	Building Blocks	189

6.4	Fundamental and Standard	190
6.5	Knowns and Unknowns	192
6.6	Primes Up To N	195
6.7	A Dearth of Primes	196
6.8	Greatest and Least	198
6.9	Infinite Alternatives	201
6.10	A Combination of Theorems	205
6.11	Theorems of Congruence	207
6.12	Modular Inverse	208
6.13	Modular Exponentiation	210
6.14	Solving Simultaneously	212
6.15	Bijectivity Guaranteed?	214
6.16	A Most Beguiling Theorem	216
6.17	Linear Systems	217
6.18	Residue Number Systems	219
6.19	The RSA Cryptosystem	220
6.20	Summary of Terms, Definitions, and Theorems	223
7	Trees	227
7.1	Growing Trees Left and Right	227
7.2	Nomenclature	230
7.3	Branching Out	230
7.4	Rooted or Not	234
7.5	Expressions	235
7.6	Fast and Glorious	235
7.7	Balancing Act	241
7.8	Optimal Search	245
7.9	Compression Confession	246
7.10	Encoding and Decoding	249
7.11	Walking It Through	250
7.12	Summary of Terms and Definitions	253
8	Graphs	257
8.1	Applications Abound	257
8.2	Origins	257
8.3	Visually Appealing	259
8.4	Cycles, Links, Paths	260
8.5	Simplicity Abandoned	263
8.6	Add Direction	263
8.7	Adjacency Counts	265
8.8	Handshaking	267
8.9	Get Directed	268
8.10	Special Graphs	270
8.11	Graph Representation	276

8.12 Graphs and Set Operations	280
8.13 Exploration	284
8.14 Summary of Terms, Definitions, and Theorems	287
9 Languages and Grammars	293
9.1 Help or Hurt	293
9.2 Order Matters	294
9.3 Logic, Beads, and Strings	296
9.4 Common Mold	297
9.5 Language Language	299
9.6 Grammar Composition	300
9.7 Parsing Up a Tree	303
9.8 Moving Closer to Useful	305
9.9 Irregular	306
9.10 Regular	307
9.11 Star Power	308
9.12 State Machinery	308
9.13 Recognition	310
9.14 Tiny Computer	313
9.15 Back and Forth	313
9.16 Taxonomy	316
9.17 Conclusion	318
9.18 Summary of Terms and Definitions	318
Appendices	
Appendix A Proofs of Some Theorems of Number Theory	325
Appendix B A Sieve to Remember	331
Appendix C An Astounding Fact	337

PREFACE

Two explanations are in order. First, the opening epigraph of this book, and second, its title. Just to be entitled, first the title's explanation: a seven-word aphorism that captures a formula for computing a well-known mathematical constant. How these seven words do all that is instructive, but best left to an animated presentation!¹

The word play by Groucho Marx is there to invite you to connect mathematics and humor, two things you may initially think have nothing to do with each other.² Dogs and books aside, there is also a connection to be made between *outside* and *inside* in a logical context. You are invited to find it!³

0.1 In-Paired Vision

This is a textbook for the study of the field of knowledge called *discrete* (not *discreet*) *mathematics*, which is a catch-all for anything not under the *continuous* mathematics umbrella (think calculus and friends). The idea is not to try to provide a complete coverage of this vast field — that would be impossible for any textbook to do. Instead, the following six topic pairs will be *very lightly* treated, with the expectation that going deeper into each will come as you continue your studies in computer science:⁴

1. Sets and Logic
2. Functions and Relations
3. Combinatorics and Probability
4. Number Theory and Practice
5. Trees and Graphs
6. Languages and Grammars

Functions (which build on sets (which build on logic)) embody transformations. Transformation describes many natural processes as well.⁵ So sets, logic, functions — these are natural, foundational topics in a study of discrete mathematics. A natural companion to this study is functional programming, the computing paradigm that adds considerable power when joining the more familiar paradigms of procedural and object-oriented programming, presumably already in your toolbox. Using a language that supports functional programming is also a natural way to learn discrete mathematics, since functions are a key aspect, and illuminate many other aspects of discrete math.

Characterized as *multi-paradigmatic*, **Python** is a popular language that supports functional programming, as well as object-oriented and procedural programming. In this course of study we will deemphasize the latter two and focus on functional, starting with a quick review in Table 1.

¹ This is the first of many side or margin notes where you will find links to external sites you can visit to find supplementary material — such as *this one — please click me to see!* All such links will be emphasized and underlined for easy identification.

² Not the first author to make the attempt, but certainly one of the most engaging, John Allen Paulos, in his eyebrow-raising book, *Mathematics and Humor*, connects them in many delightful ways.

³ In this book from time to time you will be invited to “make a connection” or “find a connection” (in some cases, a very specific one) between two (or more) ideas. Connecting concepts is vital to learning and retaining them. See Steven Pinker’s *College Makeover*.

⁴ Think of this treatment as the *absolute bare minimum* you must learn to be prepared to understand computers and computing at more than a superficial level.

⁵ Learning involves transformation from an old state of less knowledge to a new state of more knowledge.

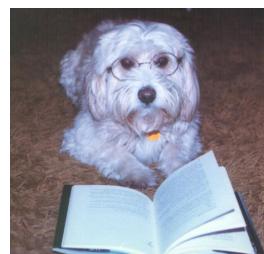


Figure 1: Sam, the book-loving dog.

Procedural and OO	Functional
Variable values vary.	Variable values never change once assigned.
Collections are mutable.	Collections are immutable.
Code is stateful (function calls can have side effects and may return values that are dependent on state).	Code is stateless (function calls can be replaced by their return values without changing program behavior, and will return the same value every time for a given input).
Functions are partial (not defined on all inputs, so exceptions can be thrown).	Functions are total (defined on all inputs, so exceptions are not thrown and need not be handled).

Table 1: Procedural, Object-Oriented, and Functional Programming compared.

⁶I share the perspective of Professor Ganesh Gopalakrishnan in his 2019 CRC Press book *Automata and Computability: A Programmer’s Perspective*:

“In my humble experience, all courses that stood out from my own student-days involved learning new concepts and reinforcing them through programming.

For illuminating concepts we use a subset of Python based on functional programming — which means recursion features prominently as do higher order functions such as map, reduce, fold, and filter.

[...] given that all modern programming languages (even C++) nowadays include some elements of functional programming, these are valuable concepts to carry with you beyond this course. [...] Besides, ideas from functional programming are powering many industries, increasing programmer productivity and helping reduce bugs.”

When viewed through the functional lens, procedural programming and object-oriented programming look the same. How they differ is in the *types* of the variables. Object-oriented adds user-defined types to the standard primitives and composites of procedural. Procedural and object-oriented also differ on how they treat nouns and verbs. It gets pretty topsy-turvy when normal everyday distinctions between nouns (things) and verbs (actions) are blurred. When nouns get verbed and verbs get nouned, it’s double blurry, and so we wisely focus on functional!⁶

0.2 More Double Vision

Speaking of double, combinatorics, the study of combining, or counting, and number theory, the study of integers (useful for counting — especially the positive ones) are a core part of discrete math. How many ways are there of combining things in various combinations? Variations, arrangements, shufflings — these are things that must be accounted for in the kind of careful counting that is foundational for discrete probability theory. But they also come into play in modern practices of number theory like cryptology, the study of secret messages.

Double encore: two more discrete structures, with a cornucopia of applications, are trees and graphs — two of the most versatile and useful data structures ever invented. For a stellar example, a *binary tree*⁷ — perhaps the easiest type of tree to understand — can represent any conceptual (as opposed to botanical) tree⁸ whatsoever. How to do so takes us into deep representational waters, but even more so with graphs. Because graphs can represent relations, listed above in conjunction with functions, they truly are rock solid stars of math and computer science. The relationship between functions and relations is like that of trees and graphs — the special to the general.

Last double: finishing off and rounding out this relatively small set of topics is languages and grammars. Language is being used, rather

⁷Please watch [an animation](#) created by a former TA.

⁸You are likely familiar with hierarchical organization charts, family trees, tournament trees, perhaps even decision trees, to name a few.

prosaically, as a vehicle for communication, even as I write these words, and you read them. There is fascinating scientific evidence for the facility of language being innate, inborn, a human instinct. Steven Pinker, who literally wrote the book on this,⁹ calls language a “discrete combinatorial system,” which is why it makes sense that it be included in this book.

And what is grammar? Pinker in his book is quick to point out what it is not. Nags such as “Never end a sentence with a preposition,” “Don’t dangle participles,” and (my personal favorite) “To ever split an infinitive — anathema!” Such as these are what you may have learned in school as teachers tried their best to drill into you how to use good English. No doubt my English teachers would cringe at reading the last phrase of the last sentence of the last paragraph, where “it” is used in two different ways, and “this book” can ambiguously refer to more than one book.

No, in the context of discrete mathematics, grammar has nothing to do with proper English usage. Instead, a grammar is a *generator* of the discrete combinations of a language system. These generated combinations are called *phrase structures* (as opposed to word structures), hence, in this sense grammars are better known as *phrase structure grammars*.

0.3 Autobiographical Aside

I wrote this book for many reasons, which mostly relate to the following language-ish math-ish relationships and transformations: Words have Letters, Numbers have Digits, Letters map to Digits, hence Words map to Numbers.

I love words. I love numbers. Does that mean English was my first choice of discipline, Math second? No! English is my native tongue, but Math to me feels *pre-native*, or the language I came into this world furnished with.¹⁰

A first recollection of my inchoate interest in math is from third grade. There I was, standing at the blackboard, trying to figure out multiplication. The process was mysterious to me, even with my teacher right there patiently explaining it. But her patience paid off, and when I “got it” the thrill was electrifying! Armed with a memorized times table, I could wield the power of this newly acquired understanding. Much later in life came the realization that I could immensely enjoy helping others acquire understanding and thus wield power!

0.4 Word Play Work Plan

Back to words. As I said, I love them. I love the way words work. I love the way they look. I love the way they sound. Part of this stems from my love of music, especially vocal music. Singers pay attention to nuances of pronunciation, diphthongs¹¹ being but one example. Take

⁹ *The Language Instinct* by Steven Pinker. This highly popular book is subtitled *How the Mind Creates Language*.

¹⁰ Not like Karl Friedrich Gauss, John Von Neumann, nor any such extraordinary mathematical prodigy. Not even close!

¹¹ How laaaa-eekly is it that you know what these are?

another example. The rule for pronouncing the e in the. Is it ee or uh? It depends not on the look of the next word but its sound. The honorable wants ee not uh because the aitch is silent. The Eternal, but the One — ee, uh. Because one is actually won.

You may have noticed that, among other infelicities, the preceding paragraph plays fast and loose with *use* versus *mention*. There is an important distinction to be made between the use of a word, and the mention of a word. For example:

Idaho has mountains. (A use of the word.)

“Idaho” has five letters. (A mention of the word.)

Consider another operation where words are mentioned in order to take them apart and recombine them: “READILY” becomes “REALITY” via a “REA”—“D”—“I”—“L”—“Y” slicing, swapping “L” for “D” and “T” for “L” simultaneously¹² — not sequentially — and then stitching the fragments back together. The letters involved in this transformation, ‘D’, ‘L’, and ‘T’, have something to do with the title of this book. What is the connection?

Consider another operation where these two closely tied words are now surrounded on either side by the same word, for example, ‘HOW’:

HOW READILY REALITY HOW

The operation of surrounding on either side with the same thing — a single letter in this case — can also be applied to one of the ‘HOW’s to form a word that will make the four words into a meaningful sentence. What is that letter, and which instance of ‘HOW’ should it surround?

We use words and “words” to aid discovery by unpacking meaning from them. This is speaking metaphorically, but if the metaphor is a good one — that meaning gets packed into words, (or encodings thereof (another layer of packing material)) — then unpacking can result in rich interconnected structures that strengthen learning, that make it stick¹³ in the brain.

Let me say a few words about the three fictional characters you are about to meet. I have given them personalities that intersect with my own, as well as those of other teachers and learners I have known. Using their voices as an echo of my voice is a way of sharing my excitement for the subject matter¹⁴ and also making for a more engaging read — that has been my aim, at any rate, and that I hit that target to your benefit is my high hope!

This book is dedicated to lovers of learning everywhere.

¹² See here.

¹³ Two sources: *make it stick* (the book), and in *Why Information Grows* on page 81 is a great discrete mathematical description of some of the difficulties of learning: “The social and experiential aspects of learning imply that there is a limit to the amount of knowledge and knowhow an individual can accumulate. That limit makes the accumulation of knowledge and knowhow at the collective level even more difficult, because such an accumulation requires that we break up knowledge and knowhow into chunks that are smaller than the ones an individual can hold.

Chopping up knowhow compounds the difficulties involved in individuals’ accumulation of knowhow not only because it multiplies the problem of accumulating knowledge and knowhow by the number of individuals involved but also because it introduces the combinatorial problem of connecting individuals in a structure that can reconstitute the knowledge and knowhow that were chopped up in the first place. That structure is a network. Our collective ability to accumulate knowledge is therefore limited by both the finite capacity of individuals, which forces us to chop up large volumes of knowledge and knowhow, and the problem of connecting individuals in a network that can put this knowledge and knowhow back together.”

¹⁴ I resonate with Hardy and Wright, who said in the preface of the first edition of *their book*:

“Our first aim has been to write an interesting book, and one unlike other books. We may have succeeded at the price of too much eccentricity, or we may have failed; but we can hardly have failed completely, the subject-matter being so attractive that only extravagant incompetence could make it dull.”

Part I: Foundations



Sand is a discrete kind of thing. Clump together enough grains of sand and you get a sea of sand, a desert. As opposed to a sea of water, an ocean, a continuous substance. Zoom in close enough, however, and an ocean is a collection of individual water molecules, again discrete.

More metaphorically, could the image suggest that we sometimes think of discrete mathematics as a “dry” topic?! Or perhaps it means that if we work alone, we could get lost in the “sands of time”—meaning that it will be a long and lonely adventure if we try to do it all by ourselves!

But this desert image also conjures up all that might be buried beneath the sand. We’ve all heard stories of possible lost libraries, cities and the knowledge that the desert civilizations once had being swallowed up in the sand, resulting in it being lost until archaeologists uncover it again. That is somewhat representative of discrete math, or math in general. Math can explain the world around us, but sometimes the math is hidden, and we have to bring it up to the surface. (Musings of former discrete math students.)



Richard Feynman once said: *I have a friend who’s an artist and has sometimes taken a view which I don’t agree with very well. He’ll hold up a flower and say “look how beautiful it is,” and I’ll agree. Then he says “I as an artist can see how beautiful this is but you as a scientist take this all apart and it becomes a dull thing,” and I think that he’s kind of nutty. First of all, the beauty that he sees is available to other people and to me too, I believe. Although I may not be quite as refined aesthetically as he is [...] I can appreciate the beauty of a flower. At the same time, I see much more about the flower than he sees. I could imagine the cells in there, the complicated actions inside, which also have a beauty. I mean it’s not just beauty at this dimension, at one centimeter; there’s also beauty at smaller dimensions, the inner structure, also the processes. The fact that the colors in the flower evolved in order to attract insects to pollinate it is interesting; it means that insects can see the color. It adds a question: does this aesthetic sense also exist in the lower forms? Why is it aesthetic? All kinds of interesting questions which the science knowledge only adds to the excitement, the mystery and the awe of a flower. It only adds. I don’t understand how it subtracts.*

— The Pleasure of Finding Things Out: The Best Short Works of Richard P. Feynman

1

Introductions

1.1 Life is But a Dream

He awoke with a start. What time was it? ... Did his alarm not go off? ... No, it was 3:45, he just woke up too early. He lay there, mind racing, trying to figure out what had woken him up. His wife was breathing deeply, she was sound asleep, so it wasn't her stirring. No ... it was that dream. That strangely awesome, awesomely strange dream where he was outside, staring into the clear night sky, verily basking "in the icy air of night" — when suddenly he felt a very real vertigo, as if he were falling *up* into that endless star-sprinkled void.

Falling up. A contradiction in terms. Yes, but to be studied, not ignored, as life's contradictions are as inevitable as gravity. No, this was the title of a book. Yes, that treasured book he had acquired years ago,¹ and still enjoyed. Clever poems and cute drawings. It was because the author was so playful with words and the thoughts they triggered that he liked it, and his other book, what was it? Ah yes, *Where The Sidewalk Ends*. Oh well, enough of that. He would have to get up soon. Today he would meet his two new tutees. TWOTees for short. A male and a female had seen his ad and responded. Was he expecting more than two to? Yes ... but two would be fine. Two would be enough to keep him busy.

¹ *Falling Up* by Shel Silverstein is a sweet treat.

1.2 First Meeting

"What I'd like to do today is just make introductions and see where we're at," he said. "My name is Tiberius Ishmael Luxor, but you can call me Til. T-I-L, my initials, my preference." While escorting his two visitors into his study, his mind wandered, as it always did when he introduced himself. Tiberius, why had his parents, fervent fans of Captain Kirk and all things Star Trek, named him that? And Ishmael? Was Moby Dick also a source of inspiration? Or the Bible? He had never asked them, to his regret. Luxor, of course, was not a name they had chosen. He rather liked his family name. The first three letters, anyway. Latin for light. He often

played with Luxor by asking himself, Light or what? Darkness? Heavy? Going with the other meaning of light seemed frivolous.

"Uh, my name's Atticus. Atticus Bernhold Ushnol," his male visitor said, noticing that Til looked distracted. "What do you prefer we call you?" asked Til, jerking himself back to full attention. "Well, I like Abu, if that's okay with you!"

Til smiled. A man after his own heart. "How acronymenamored of you!" he said.

"Is that even a word?" asked his female guest, addressing Til. Feeling just a little annoyed, she turned to Abu and said "You really want us to call you the name of Aladdin's monkey?" Abu deadpanned, "Aladdin who?!"

"You can't possibly have not seen Aladdin! What planet are you from?" she said, completely incredulous.

"Wait a minute," said Til. "Let's discuss the finer points of names and origins after you introduce yourself."

"Ila. My middle name is my maiden name, Bowen. My last name is my married name, Toopik, which I suppose I'm stuck with." Ila always felt irritated that marriage traditionally involved changing the woman's name, not the man's. "And yes, it's Eee-la, not Eye-la. And I especially prefer you don't call me Ibt, if you get my drift!"

Til laughed. Abu grinned but said nothing, so Til went on. "We need variety, so Ila is fine. Not everyone needs to be acronymized!"

1.3 First Impressions

Later that day as Til reflected on their first meeting, he saw himself really enjoying the teacher-learner interactions to come. His ad had had the desired effect. Abu and Ila were motivated learners, albeit for different reasons. He was somewhat surprised that not more interest had been generated. But he knew it was unlikely up front. So two it would be. Two is a good number. Three is even better (odd better?) if he counted himself as a learner. Which he always did. Learning and teaching were intricately intertwined in his mind. Teaching was like breathing, and both teaching and learning were exhilarating to Til.

Til's wife Tessie interrupted his reverie. "Dear, what did you learn today from your first meeting with — what did you call them — Abu and Ila?"

"Interesting things," said Til. "Abu is the general manager at a landscaping company (I think he said nursery as well). Ila works for some web design company (I forget the name) doing programming. She wants to learn discrete math because she's been told it will make her a better programmer!"

"However, Abu's not a programmer, and not really all that techy, I

gathered, but says he *loves* learning, practical or not. I predict this may put him at odds at times with Ila, who said she *only* wants practical learning. I told her Euclid would mock that attitude, and she was surprised. But she agreed to look into that. She didn't know who Euclid was, if you can imagine!" Til opened both eyes wide in brief mock surprise, and his wife rolled hers.

"Anyway, Abu's pretty laid back, so her barbs won't bother him — much I hope," he said.

"Are they married, any kids?" she said.

"Abu is single, but would like to be married." Til rubbed his chin. "Ila is married, but she didn't mention her husband's name, or really anything about him, except that he's not a nerd like her. Before they even warmed up to my affinity for acronyms, Ila confessed to being a DINK,² and Abu immediately responded, 'I guess that makes me a SINKNEM!'³ So he's pretty quick on the uptake."

"That's good!" she said. "So Ila's husband is not nerdy like her, but ...?"

"But she also didn't say what he does," he said. "She taught herself programming, how cool is that?!"

"Very cool, Mr. Luxor," she said affectionately, gently rubbing his shoulders.

"Mrs. Luxor, I'll give you 20 minutes to stop that!" he said.

"What are their interests other than discrete math?" she said, ignoring his pathetic ploy.

"Well, let's see, Abu rattled off his other interests as flowers, stars and poetry!" he said. "And Ila likes sports and the outdoors. Hopefully that includes stars!"

"That would be great if you all have stars in common," she said.

Til nodded, but his eyes unfocused as his thoughts started to wander. His wife withdrew, wordlessly worrying that Til was thinking about sharing his dream about falling up into the stars with Abu and Ila. That never seemed to go well. But Til was remembering the exchange he had with Abu, who lingered a few minutes after Ila left.

He had said, "Abu, since you like flowers, you must be okay with their STEM⁴less reputation!" Without waiting for him to answer he went on. "Since I gave Ila an invitation to learn something about Euclid, I invite you to look up what Richard Feynman said about flowers and science, specifically whether knowing the science made him appreciate the flowers more or less!"

Well, enough musing for now. His two tutees were eager and grateful, and not just because they thought the fee he had set was reasonable. Little did they know the price of knowledge he would exact of them.

² Dual Income No Kids

³ Single Income No Kids Not Even Married

⁴ Science Technology Engineering Mathematics — of course — flowers really do have stems.

1.4 First Assignment

"Wow, can you believe the homework Til assigned on our first day?" Ila said. "It's a little intimidating, yes," said Abu. "But I'm kinda looking forward to the challenge! I'll have to bone up on my high school math a little, I'm afraid." "Same here," said Ila, "and I suppose he's right that this will give him an idea of our mathematical maturity level." "Exactly," Abu said. "That problem about the circles and polygons threw me for a loop, until I thought a little trigonometry review might help." "Yeah, same here" said Ila. "But I must say I like the word problems, even though they don't seem to have much to do with math."

"He did say we could collaborate, and even encouraged it," Abu said. "I'm game," said Ila. "My husband is okay with our getting together online once or twice a week. He's very supportive of my learning and trying to improve my computer science skills. Means more money from my employer!" Abu grinned. "I just want the knowledge, I don't really care about more money — but yeah, let's put our heads together and see what we can do!"⁵

⁵ Good collaboration is a “me lift thee and thee lift me, and we both ascend together” phenomenon. Bad collaboration is copying someone else’s work and submitting it as your own, to get “credit” for doing some assigned task, while not getting the understanding doing the task was meant to give you.

Exercise 1 The operation called dehydration takes a word and deletes all letters in it from H to O. What is the original (rehydrated) text of this dehydrated sequence of words?

TE TE AS CE TE WARUS SAD T TA F AY TGS F SES AD SPS AD
SEAG WAX F CABBAGES AD GS AD WY TE SEA S BG T AD WETER
PGS AVE WGS

Hint 1 Lewis Carroll is your friend here.

KINGS AND THE SEA IS BOILING HOT AND WHETHER KING
OF SHOES AND SHIRTS AND SEARING HOT OR CABBAGES AND
THE TIME HAS COME THE MARGRAS SAID TO TALK OF MANY THINGS
LOOKING-CLASS AND WORKING-CLASS:
coffee „The Margrass and the Carpenter,“ in his 1872 book *Workers* from
readers in schools the answer: This is from the numerous books of Carrolls
that did not get published, such as the series of books Carroll *Answers*

Exercise 2 What English word has the largest finite consonant-vowel-ratio (CVR)? For example, the CVRs of the words (including the TLA) in the previous sentence are 3:1, 5:2, 3:1, 2:1, 2:1, 5:2, 1:1, 2:1, 3:2, 2:3, and 3:0 — say $n:0$, which is $n/0$, is infinity for any

positive integer n . Which is to say that “words” with no vowels are disqualified (because CVRs must be finite).

Hint 2 Look for words with just one vowel in them. Count y as a vowel, even when it is the first letter of a word.

AT A ASA A TAWSMIA

Exercise 3 Do you think mathematicians discover or invent patterns? While pondering that question, consider what G. H. Hardy famously wrote: A mathematician, like a painter or a poet, is a maker of patterns. If his patterns are more permanent than theirs, it is because they are made with ideas. What do you think he meant by that?

Pick a number (a positive integer). Form the product of that number, one more than that number, and one more than twice that number. Repeat for several different numbers. There is a pattern to these products. What is it?

Hint 3 The picked number is referred to as ‘that number’ three times. It doesn’t change during the forming of the product. Try dividing all the products by 6.

AT A ASA A TAWSMIA

Exercise 4 What does “654321” have to do with “CLAHCK” — other than that they both match the “length 6” pattern?

Hint 4 Split the 6-digit number into two 3-digit numbers, and think of the word CLOCK (what CLAHCK really should be) as a sequence of 5 letters.

AT A ASA A TAWSMIA

Exercise 5 In Figure 1.1, the innermost circle has radius 1. It is circumscribed by an equilateral triangle, which is circumscribed by a circle, which is circumscribed by a square, which is circumscribed by yet another circle, and so the pattern continues. What is the radius of the outermost circle?

Hint 5 Another name for this problem is the Limiting Radius problem. Try a trigonometric approach, as Abu said.

Եթե α և β անգամ մեծ են, ապա $\sin(\alpha + \beta)$ և $\cos(\alpha + \beta)$ ավելի մեծ են, քանի որ $\sin^2 x + \cos^2 x = 1$. Այս հարցում առաջանակ է առաջանակը:

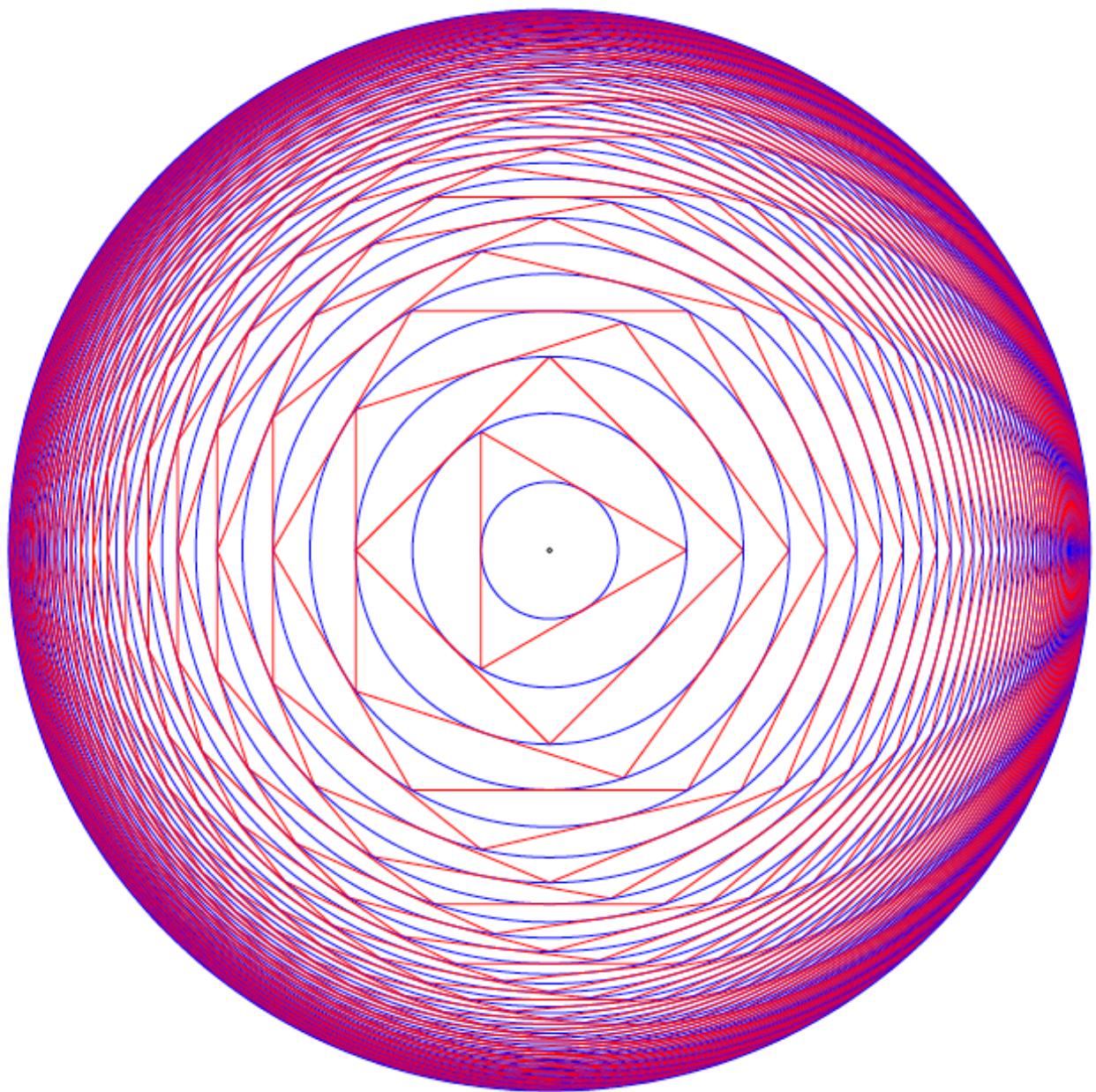


Figure 1.1: Circumscribed Polygons and Circles

2

Sets and Logic

2.1 Set the Stage

“Let’s begin at the beginning,” said Til, hearing a song in the back of his head. “When you read you begin with A, B, C; when you sing you begin with Do, Re, Mi; when you count you begin with 1, 2, 3.” Abu chimed in, “Aren’t those called the counting numbers?” “Yes,” replied Til. “They are AKA the positive integers; or speaking collectively, the *set* of positive integers.”

“Remind me, please, what AKA means,” asked Ila. “Also Known As?” ventured Abu. He was warming up to Til’s acronymenamorization, and was pleased to realize that he was really going to enjoy this learning adventure with Til and Ila.

“Right!” said Til. “Except hold on. How do you count the number of elements in the set with no elements, the so-called *empty set*? ” Without waiting for a reply, Til went on. “You need zero — which for centuries was not even thought to be a number.”¹

“Isn’t zero considered a natural number, along with the counting numbers?” Abu said. “Yes,” said Til. “These days zero is indeed a full-fledged member of **the set of natural numbers**. But some still see zero as **UNnatural** and thus exclude it from the set. Already ambiguity rears its head, and so we have to be sure to clarify our definitions before making arguments that depend on whether zero is in or out.”

“Why don’t we just decide which it is and be done with it?” said Ila, exasperated. “Mathematicians have logical reasons for using both, in different contexts,” said Til. “Computer scientists mainly do zero-based counting, or indexing, as both hardware and software coax them to this side.” “I like being noncommittal,” said Abu. “Keeps my options open!” Ila rolled her eyes.

Til went on. “A collection that serves the mathematical idea of a set² is an **unordered** one — the elements can be listed in any order and it *does not* change the set. Here are some simple examples.” (See Table 2.1.)

“Remind me what those three dots are saying,” said Ila. “I know!”

¹ See *Zero: The Biography of a Dangerous Idea* and, bringing the danger and the strangeness to the present day, two more: *Void: The Strange Physics of Nothing* and *The Book of Nothing: Vacuums, Voids, and the Latest Ideas about the Origins of the Universe*.

² It is impossible to improve upon [Georg] Cantor’s succinct 1883 definition: *A set is a Many which allows itself to be thought of as a One.* One of the most basic human faculties is the perception of sets. [...] When you think about your associates and acquaintances you tend to organize your thought by sorting these people into overlapping sets [...]. Or the books that you own, the recipes that you know, the clothes you have — all of these bewildering data sets are organized, at the most primitive level, by the simple and automatic process of set formation, of picking out certain multiplicities that allow themselves to be thought of as unities. Do sets exist even if no one thinks of them? The numbers 2, 47, 48, 333, 400, and 1981 have no obvious property in common, yet it is clearly possible to think of these numbers all at once, as I am now doing. But must someone actually think of a set for it to exist? [...] The basic simplifying assumption of Cantorian set theory is that sets are there already, regardless of whether anyone does or could notice them. A set is not so much a “Many which allows itself to be thought of as a One” as it is a “Many which allows itself to be thought of as a One, if someone with a large enough mind cared to try.” For the set theorist, [...] a set **M** consisting of ten billion random natural numbers exists even though no human can ever see **M** all at once. A set is the form of a possible thought, where “possible” is taken in the broadest possible sense. — Rudy Rucker, *Infinity and The Mind: The Science and Philosophy of the Infinite*, pages 191-192.

Math	Python Set	Python List	Python Tuple
{ } or \emptyset	{ }	[]	()
{A, B, C}	{A, B, C}	[A, B, C]	(A, B, C)
{Do, Re, Mi}	{Do, Re, Mi}	[Do, Re, Mi]	(Do, Re, Mi)
{1, 2, 3}	{1, 2, 3}	[1, 2, 3]	(1, 2, 3)
{0, 1, 2, ...}	{0, 1, 2}	[0, 1, 2]	(0, 1, 2)

Table 2.1: Five sets in four different styles

³The set \mathbb{N} of natural numbers (including zero) is AKA the nonnegative numbers. Clarification: in this context numbers mean integers, numbers with no fractional parts, not even decimal points followed by nothing (or nothing but zeros). The set \mathbf{P} is *not* the set of positive integers, that's called \mathbb{Z}^+ . \mathbf{P} names the **prime** numbers (but not always). \mathbb{Z}^- is the negative integers, but there's no abbreviation for the nonpositive integers, which is \mathbb{Z}^- together with 0. All together, \mathbb{Z} is the short name for all integers, positive, negative, and zero — and is short for **Zahlen**, which is the German word for numbers.

Abu jumped in. “They say *and so on*, meaning *continue this way forever*.” Til smiled and said, “Indeed! However, in a computer programming language, like Python, continuing forever is a bad idea! So we just truncate the set³ after listing its first three members. The three dots are called *ellipsis*, by the way, and the symbol name for this set is \mathbb{N} .”

Abu showed his doodles. “Aren’t duplicates disallowed in sets? Like, aren’t these all the same set, {A,B,C}?”

{A,B,C}, {A,A,B,C}, {A,A,A,B,C,C}

“Yes, they are!” said Til. Ila said, “Wouldn’t these also be considered the same set?”

{B,A,C}, {C,A,B}, {B,C,A}

“You’re both right — good insights!” Til continued. “So remember these two features, constraints, conditions, what have you. Sets, one, guarantee element uniqueness — no duplication — and two, they exhibit element unorderliness. So make it a rule to get rid of repeated elements when listing sets, and if it pleases you, list the elements in some natural order, but always keep in mind that their order *does not matter*.”

2.2 Start with Propositions

⁴George Boole is who this data type is named for, mostly because of his seminal book shortnamed the *The Laws of Thought*. (What is its longer name?)

“Now let me draw your attention to a way to treat statements about sets as **propositions** with Boolean values⁴ (true or false).” Til was warming up to his topic. “Let’s be clear on definitions, first. A **proposition** is a sentence (or a statement) that is either true or false, nothing else. The sentence can be about anything at all (so not just sets), as long as ascertaining its truth or falsity is feasible. The study called **propositional logic** deals with standalone and compound propositions — which are propositions composed of other propositions, standalone or themselves composite. We’ll go over how to compose or combine propositions to create these compounds after a few exercises in *model building*. Let’s start with some simple examples of standalone propositions, and for contrast, **non-propositions**.”

“A proposition: *Man is mortal*.”

“Another proposition: *The sky is up*.”

“A proposition about sets: *The English Alphabet can be split into the set of vowels and the set of consonants.*”

“A proposition about numbers: *The square of a number is always greater than that number.*

“One of these propositions is false. Which one?” Til interrupted himself to see if his interlocutors were paying attention. “Ila, you take this one,” Abu said. Ila smiled and said, “The squares of 0 and 1 are 0 and 1, so not greater — equal. That means the last so-called proposition is not true.” “Right,” Til said. “But remember, it’s still a proposition even if it’s false.” “Oh, right!” Ila blushed a little, chagrined that Til had corrected her. “I thought it was a non-proposition too,” Abu said, noticing Ila’s discomfort and wanting to distract her from it. “Could we change the wording a little and make these propositions about sets? Like *Man is a member of the set of mortals*, or *The sky is in the set of things that are up?*” Being wrong is in the set of things I hate, thought Ila, still smarting a little. Til nodded, and said, “By the way, use the symbol \in to say *is a member of* or *is in*, as in $x \in A$ — x is in A . Now, what are some non-propositions?”

“Why bother?” said Ila. Abu blinked, and she went on. “That’s a question, not a statement. So not a proposition, right?” “Right,” said Til. “Go to the back of the boat,” Abu blurted out, and silently added *he said sternly*. He thought it best not to say that out loud. Sometimes his penchant for punnery got him into trouble! He also didn’t want to set Ila off. He knew she had a sense of humor, but he hadn’t quite figured it out yet. “A command, excellent!” chuckled Til, looking quizzically at Abu. Abu got the distinct impression that Til could read his mind about the pun he only thought.

“We are done for today and you are leaving,” said Til. “That’s an example, by the way, of a compound proposition — two standalone propositions (We are done for today, you are leaving) joined with *and*. There are of course many other ways to compose propositions to make them compounds, but we’ll look at those next time. Let me just leave you with this: building mental models requires raw materials — thought stuff — and the knowhow to combine and arrange this stuff usefully. Both come more easily and more plentifully as you do exercises and work through problems. Hence your homework!”

Exercise 6 Write three propositions and three non-propositions. Make them pithy.

Did you know that now is the time for all good men to come to the aid of their country?
Hint 6 Examples of (non-)pithy (non-)propositions: Ed is tall. Now is the time for all good men to come to the aid of their country. Duck!

AT A GLANCE

Exercise 7 Put the proposition “ x is a member of the set of English alphabet consonants” in symbolic logic terms, using C as the symbol naming the set of consonants.

called for:

Hint 7 This is very straightforward. Three symbols are all that are

AT A GLANCE

Exercise 8 Proposition or not? If so, true or false? “ $2 + 2 = 4$.”

HINT 8 EASY

AT A GLANCE

Exercise 9 Proposition? True or false? “ $2 + 1 = 4$.”

Hint 9 Brush up on your arithmetic!

AT A GLANCE

Exercise 10 Proposition? True or false? “Toronto is the capital of Germany.”

Hint 10 With apologies to any Canadians or Germans.

AT A GLANCE

Exercise 11 Proposition? True or false? “Read these questions carefully.”

Hint 11 And carry out all instructions.

AT A GLANCE

Exercise 12 Proposition? True or false? “ $x + y + z = q$.”

Hint 12 Perhaps, it depends?

AT A GLANCE

Exercise 13 Proposition? True or false? “What time is it?”

Hint 13 Time to get this.

AT A GLANCE

Exercise 14 Proposition? True or false? “ $2x + 3 = 9$.”

Hint 14 It depends again?

AT A GLANCE

Exercise 15 Proposition? True or false? “Jump!”

Hint 15 Have faith that you got this.

AT A GLANCE

Exercise 16 “Weeks are shorter than months.” Is this a simple proposition or a compound proposition?

Hint 16 This is a gift to be simple.

AT A GLANCE

Exercise 17 Simple or compound? “Days are longer than hours and minutes.”

Hint 17 Be careful!

Правильные ответы:
Правильные ответы: “Дни дольше часов” и “Дни дольше минут.”
Правильный ответ

Exercise 18 Simple or compound? “Hours are longer than minutes and seconds are shorter than minutes.”

Hint 18 The answer should be obvious.

Правильный ответ
Правильный ответ

Exercise 19 Simple or compound? “People can fly or pigs can sing.”

Hint 19 Don't you wish?

Правильный ответ
Правильный ответ

2.3 Modeling Stuff

“True propositions are simply **facts**.” Til started in as soon as Abu and Ila appeared, much to their surprise. It’s like we never left, thought Ila. “So we call a set of true propositions a knowledge base, which is just a database of facts. This represents what we know about the world (or a subset of the world) or, in other words, the meaning or interpretation we attach to symbols (such as sentences — strings of words). Take a simple world.”

- It is true that snow is white.
- It is false that grass is white.

“Let’s assign the propositional variables p and q to these propositions (made pithier) to make it easier to manipulate them.”

- p = “snow is white”

- $q = \text{"grass is white"}$

"Now, the compound proposition $p \text{ and } \text{not } q$ represents our interpretation of this world."⁵

"So, roughly speaking, a **model** is what meanings or interpretations we give to symbols. In general, models are all over the map, but in propositional logic, models are assignments of truth values to propositions, which, for convenience, we will continue to use variables for."

Abu spoke up. "You mentioned before that truth values are Boolean values, true or false, right?" "Precisely," said Til. "That means True or False. Period. End of sentence. No partly true or somewhat false, nor any other kind of wishy-washiness." Abu *almost* spoke his little joke: *I used to think I was indecisive, but now I'm not so sure!* — good thing he was getting better at thinking better of it. Not that Til would mind, but Ila got easily annoyed when he indulged.

Til continued. "True-or-false logic is a crucial feature of any programming language — Python is no exception. So, we will study the programming logic of the basic logical connectives by looking at the following four functions, below which I've put the corresponding logic symbol. The first three are built-in-functions (BIFs) in Python, by the way:"

and	or	not	xor
\wedge	\vee	\neg	\oplus

Abu and Ila considered the table for a silent moment. Then Ila spoke the burning question on her mind. "What's Python's non-built-in way to do xor?" "The up caret will do it — still built-in but not a function," said Til. "But more on that later. For now just focus on not — not is **not** a connective in the sense of connecting *two* propositions together, but it behaves as a logical *operator* nonetheless."

Abu looked at Til and said, "I have two questions. Does operator mean the same thing as connective? And is not not — well, and and or too, really — aren't they just pretty ordinary words in everyday life?"

"I'll answer your second question first," Til said, "but again, focus on not. So yes, in everyday life, **not** expresses negation, flipping truth values. When notted (AKA negated) true becomes false, false becomes true. "Birds are singing" or "Birds are not singing," if one is true the other is false. As to whether or not operator and connective mean the same thing — they do and they don't. When we talk about infix versus prefix versus postfix, and functions and their arities, you'll see what I mean."

"Now watch this —" Til paused for effect. "You can always make the negation of a proposition, for example, *birds are singing*, with just *It is not the case that birds are singing*."

Ila eagerly interjected, "So just put those six words in front of any proposition, and that will negate it?" Til nodded, and Abu huffed. "Sounds like the lazy way to do it!"

⁵ One could imagine a stranger world where the model is p and q (*i.e.*, where snow is white and so is grass) or even not p and q (where snow is green, say, and grass is white).

“It is more verbose,” Til said, “so you have to use more words, which is not really lazy. Real laziness means avoiding work! By using that six-word prefix you avoid the mental effort of figuring out where to insert the *not* in your proposition. Putting “It is not the case that” in front of a proposition is a double sin — lazy *and* wasteful. But in symbolic logic, you can be lazy and succinct! Prefix any proposition p with \neg to get $\neg p$, and you just inverted its truth sense, no muss no fuss.”

Exercise 20 Express in English (“anglify”) the negation of “Two plus two is four.”

Hint 20 For negation, **do not** use the “It is not the case that” prefix.

AT a fSA 0S 1EWsNA

Exercise 21 Anglify the negation of “Two plus two equals four.”

Hint 21 Make it sound like good English.

AT a fSA 0S 1EWsNA

Exercise 22 Negate / anglify “Toronto is the capital of Germany.”

Hint 22 Fool me once . . .

AT a fSA 0S 1EWsNA

Exercise 23 Negate / anglify “A total eclipse happens infrequently.”

Hint 23 Did you see the one on August 21st, 2017?

AT a fSA 0S 1EWsNA

Exercise 24 Negate / anglify “Special measures must be taken to deal with the current situation.”

Hint 24 Before it gets out of hand.

AT a ~~as~~ A ~~as~~ **Answers**

Exercise 25 Let p be the proposition “I studied.” and q be the proposition “I got an A on the test.”

Anglify $p \wedge q$.

Hint 25 Substitute p and q as appropriate.

AT a ~~as~~ A ~~as~~ **Answers**

Exercise 26 Anglify $\neg p \wedge q$.

Hint 26 You got lucky.

AT a ~~as~~ A ~~as~~ **Answers**

Exercise 27 Anglify $p \wedge \neg q$.

Hint 27 Try studying harder:

εδεαδιεμάθε δηλούτ, μεανσά ανδ, — ανδ it σωσης δεττερ μερε. (Γολγίαστι) Αν δε την πιθανότητα I ~~as~~ **Answers**

Exercise 28 Anglify $\neg p \wedge \neg q$.

Hint 28 Are you surprised?

AT a ~~as~~ A ~~as~~ **Answers**

Exercise 29 Anglify $p \vee q$.

Hint 29 Continue to use the same p and q .

AT A GLANCE | **ANSWER**

Exercise 30 Anglify $\neg p \vee q$.

Hint 30 Which is it?

AT A GLANCE | **ANSWER**

Exercise 31 Anglify $p \vee \neg q$.

Hint 31 It's better to study.

AT A GLANCE | **ANSWER**

Exercise 32 Anglify $\neg p \vee \neg q$.

Hint 32 Reconsider how you study.

AT A GLANCE | **ANSWER**

2.4 Exclusive and Inclusive

Abu and Ila studied the message they just got from Til:

“Contrast xor, the *exclusive* or, with the *inclusive* or:

- Inclusive: “ p or q ” is true if either p is true or q is true, *or both*.
- Exclusive: “ p xor q ” is true if either p is true or q is true, *but not both*.”

“As far as implementation goes, I invite you to compare the bitwise XOR (\wedge) operator I mentioned earlier with a function named `xor` that makes clever use of the `not` operator in an `if/else` expression (see Figure 2.1).”

```
def xor(p, q):
    return not q if p else q

actual = [True ^ True,
          True ^ False,
          False ^ True,
          False ^ False,
          xor(True, True),
          xor(True, False),
          xor(False, True),
          xor(False, False)]

expected = [False, True,
            True, False,
            False, True,
            True, False]

print(actual == expected)
```

Figure 2.1: Two equivalent implementations of *exclusive or*

"Please run this code and verify that all four cases work correctly both ways." Til's message concluded.

Ila sighed. "I guess Til just expects us to know Python already." Abu replied, "Or pick it up on our own, I suppose." Ila frowned. "I wish it were more like JavaScript. That's a language I understand. With no semicolons or curly braces, Python just confuses me." Abu said, "I don't know about those things you said, but —" He grinned his cheesiest grin. "I guess not knowing any other programming language will make learning Python a breeze for me — nothing to unlearn!" "Well aren't you King of the Planet!" retorted Ila. "Well," Abu laughed and winked, "just so you know what planet I'm from, I did see Aladdin. And since Star Wars is all the rage — Chewbacca is to Han Solo as Abu is to Aladdin, right?" "Hardly," said Ila. "So, knowing his antics you still want to go by Abu?!"

"Oh, look, Til just sent another message," Abu ignored her taunt. "Looks like we get to do more exercises!" Ila refocused, "I think he just wants us to say whether the *inclusive or* is the intended meaning in these sentences, or does it make more sense as the *exclusive or*?"

Exercise 33 Which or? "A side of fries or chips comes with your sandwich."

Hint 33 How hungry are you?

Answer 33 Exclusive

Exercise 34 Which or? "A high school diploma or a GED are needed for this position."

Hint 34 It's not hard to tell.

Answer 34 A GED

Exercise 35 Which or? "To get your license, you must provide your social security card or birth certificate."

Hint 35 And pass a test.

Answer 35 Inclusive

Exercise 36 Which or? “We accept Mastercard or Visa credit cards.”

Hint 36 What’s in your wallet?

AT a **Visa** **MasterCard**

Exercise 37 Which or? “You can cash in your tickets or exchange them for a prize.”

Hint 37 A cash prize?

AT **cash** **prize**

Exercise 38 Which or? “Take it or leave it.”

Hint 38 Last chance.

AT **a** **last** **chance**

2.5 Rules to Build By

“Today,” said Til, “I want to introduce a very important principle to you, and then we’ll talk about how we combine propositions to create compound propositions. If you will diligently apply it, this principle will result in deeper and more satisfactory learning. What we’ll talk about today, as far as propositional logic goes, is only the start, the veneer, the surface, the superficial.” Til paused, deep in thought.

Ila was getting a little antsy — what was this all-important principle? Her agitation did not go unnoticed by Abu, but just as he leaned forward to give her a calming look, Til fairly shouted:

“Never settle for the superficial! — Always look below the surface!”

Ila felt her spine tingle as she stole a glance at Abu, whose eyes went big at this display of unusual intensity from their tutor. But as Til’s eyes bored into hers, she couldn’t shake the nagging suspicion that superficiality had always been her approach to learning, and that Til somehow knew that about her.

“I’ll give this principle a name and we’ll discuss it more later,” Til said, “but please keep it in the forefront of your mind. As you may have gathered, the things I’ll try to furnish you with in our little sessions are pretty

much just the building blocks — it's up to you to build something with them.”

“For example, a huge application area of logic is proof. Formal proof. Mathematical proof. I'll only lightly touch upon methods and strategies of building valid arguments and rigorously proving mathematical statements. Not as in ‘beyond reasonable doubt’ types of proofs — ‘beyond all doubt’ is the only viable standard. But as I said, going beyond the basics is for you to do. Are you with me?”

Abu and Ila looked at each other, then looked at Til and nodded solemnly.

“Good,” said Til. “Let's start by seeing how to create compound propositions by applying some rules over and over again. Think of these rules as a recipe for making propositional *molecules* from propositional *atoms*. There are four of them.”

1. A *proposition* is a variable, that is, a single element from the set $\{p, q, r, s, t, \dots\}$;
2. alternatively, a *proposition* is a *proposition* preceded by a *not*;
3. alternatively, a *proposition* is a *proposition* followed by a *connective* followed by a *proposition*.
4. A *connective* is a single element from the set $\{\text{and}, \text{or}, \text{xor}\}$.

“Note that the second and third rules have a *recursive* flavor, so called because they refer back to themselves. In other words, they define a proposition in terms of itself, and also simpler versions of itself.” Abu recalled the back-and-forth he and Ila had had the last time they met:

- Abu: I'm happy.
- Ila: I'm happy you're happy.
- Abu: I'm happy you're happy I'm happy.
- Ila: I'm happy you're happy I'm happy you're happy.

Fortunately it ended there. He thought about mentioning this potentially never-ending exchange as an example of a different flavor of recursion that leads to less and less simple versions of itself — but he refrained.

“We actually need more syntactic structure to make sense of certain compounds,” Til said. “For example — building a proposition from scratch by starting with proposition and then seeing what applying rules can yield. Table 2.2 shows this same sequence in shortened form:

Applying rule 3 yields proposition connective proposition;
 applying rule 1 and choosing p yields p connective proposition;
 applying rule 4 and choosing and yields p and proposition;
 applying rule 3 yields p and proposition connective proposition;
 applying rule 1 and choosing q yields p and q connective proposition;
 applying rule 2 yields p and q connective not proposition;
 applying rule 1 and choosing r yields p and q connective not r;
 applying rule 4 and choosing or yields p and q or not r.”

Applying	Choosing	Yields
Rule 3	—	proposition connective proposition;
Rule 1	p	p connective proposition;
Rule 4	and	p and proposition;
Rule 3	—	p and proposition connective proposition;
Rule 1	q	p and q connective proposition;
Rule 2	—	p and q connective not proposition;
Rule 1	r	p and q connective not r;
Rule 4	or	p and q or not r.

Table 2.2: Applying rules, making choices, yielding results

“Now note that this is ambiguous. Does it mean p and, say s where s is q or not r? Or does it mean s or not r where s is p and q? In other words, which is the right interpretation, p and (q or not r), which is called *associating from the right*, or (p and q) or not r, which is *associating from the left*.

Ila observed, “You didn’t mention parentheses as part of the rules.” Til nodded and said, “You’re right. The parentheses are the extra *syntactic sugar* we need to sprinkle on the expression to *disambiguate* its meaning— force the choice of associating from the right or associating from the left.”

Abu chimed in, “Does it even matter?” “Let’s find out,” said Til. “First with the truth-value assignment of true to p, false to q, and true to r. Making the assignment in p and (q or not r) — in other words, replacing p, q, and r with true, false, and true, respectively, gives us true and (false or not true).”

“Now before simplifying this expression, collapsing it down to a single truth value, true or false, let’s quickly review how each of these works — the simplest ‘connective’ first:

- not flips truth values, true becomes false, false becomes true.
- or, to be true, requires at least one of the two propositions appearing on its left-hand-side and its right-hand-side to be true. If both of them are false, their *disjunction* (another name for or) is false.
- and, to be true, requires both left-hand-side and right-hand-side propositions to be true. If even one of them is false, their *conjunction* (another name for and) is false.”

Abu’s head was starting to swim. He made some quick notes to help him keep these straight. Ila looked at him, slightly amused.

“So back to the task,” said Til. “We have true and (false or not true), which simplifies to true and (false or false) then to true and false then to simply and finally, false. Now do it with (p and q) or not r and you get (true and false) or not true, then (true and false) or false, and finally, false or false, which is still false.”

"So they're the same!" said Abu. "Hold on," said Ila. "What about a different assignment of truth values? I just tried it with p, q and r all set to false, and here's what I got for (p and (q or not r)):"

1. false and (false or not false)

2. false and (false or true)

3. false and true

4. false

"Then I did it with ((p and q) or not r):"

1. (false and false) or not false

2. false or not false

3. false or true

4. true

"Different!" said Abu. "So it *does* matter!"

Til was pleased. "Very good," he said. "Now, instead of going about this haphazardly, let's tabulate all possible truth-value assignments — AKA models — for both forms, and see how often they're the same or not. First problem, how do we generate all possible assignments?"

"I know," said Ila, "we could do a binary decision tree." "Wonderful idea!" said Til. "And I have just the one we need!" (See Figure 2.2.)

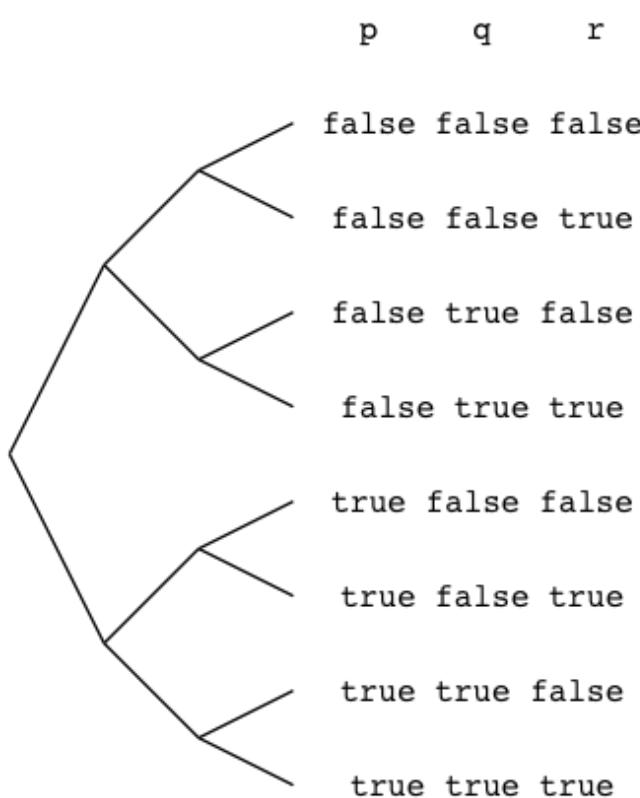


Figure 2.2: Binary decision tree truth assignments. In this binary tree, each **node** (wherever it branches up or down) represents a decision to be made. Branching up represents a choice of false, branching down true. Going three levels (branch, twig, leaf) gives the eight possibilities. Just like two choices (2^1) expand to four (2^2) when for each of two choices we make another binary choice; thence to 8, 16, 32, ... or 2^n where n is the number of choices made in any binary true-or-false yes-or-no in-or-out on-or-off... fashion.

2.6 Truth Tables

This was fascinating to Abu. And he must have looked it, because Ila was looking at him as though she found his fascination fascinating. Til went on. “A **truth table** is just a tabulation of possible models. To make it from the tree, just pick off the leaves from top to bottom, making a row of each leaf, as shown in Table 2.3.”

Table 2.3: Truth table for all possible models for p and $(q \text{ or not } r)$

p	q	r	not r	$(q \text{ or not } r)$	p and $(q \text{ or not } r)$
false	false	false	true	true	false
false	false	true	false	false	false
false	true	false	true	true	false
false	true	true	false	true	false
true	false	false	true	true	true
true	false	true	false	false	false
true	true	false	true	true	true
true	true	true	false	true	true

“If we want to save space we could also abbreviate true as T and false as F, as in Table 2.4.”

Table 2.4: Truth table for all possible models for p and $(q \text{ or not } r)$ with T and F

p	q	r	not r	$(q \text{ or not } r)$	p and $(q \text{ or not } r)$
F	F	F	T	T	F
F	F	T	F	F	F
F	T	F	T	T	F
F	T	T	F	T	F
T	F	F	T	T	T
T	F	T	F	F	F
T	T	F	T	T	T
T	T	T	F	T	T

⁶The *arithmetization* of logic (using 0 and 1) is also instructive because of how it enables certain useful encodings. What this means is to *operationalize* logic by making it more like doing arithmetic. For example, not (as in $\text{not } 1 = 0$, $\text{not } 0 = 1$) becomes the subtraction-from-one operation. Compound propositional formulas can be encoded in useful ways as well. One useful encoding turns a Boolean formula ϕ into a polynomial equation E in many variables. This clever encoding is such that ϕ is satisfied if and only if E has integral roots.

“Numerically it makes sense to have, and indeed many programming languages do have, 0 represent false. Having 1 represent true has advantages, too, although some languages treat anything non-zero as true as well. Still, it’s less typing, so we’ll use 0 and 1 to save space.”⁶

"One more thing," continued Til. "Let's also replace words with symbols⁷ — so and becomes \wedge , or becomes \vee , and not becomes \neg , and thus our truth table becomes much more compact." (See Table 2.5.)

"Likewise for the second parenthesization." (See Table 2.6.)

"Let's combine the truth tables for the binary logical operators, \wedge , \vee , and \oplus , with an output column for each." (See Table 2.7.)

"Next time we'll explore propositions involving set membership in various forms," concluded Til. "Til we meet again!"

Exercise 39 Compare the truth table outputs in Table 2.5 with those in Table 2.6.

Hint 39 They differ in just two places.

AT A GLANCE OF ANSWERS

Exercise 40 Construct a truth table for $p \wedge p$.

Hint 40 Use Table 2.7 and pretend the second p is q .

AT A GLANCE OF ANSWERS

Exercise 41 Construct a truth table for $p \vee p$.

Hint 41 This has a simpler truth table with just one column.

AT A GLANCE OF ANSWERS

Exercise 42 Construct a truth table for $\neg p \vee q$.

Hint 42 A baker's dozen.

AT A GLANCE OF ANSWERS

⁷ Abu and Ila came up with some mnemonics to help keep these straight. The \wedge symbol looks like an 'A' (for 'And') without its middle bar. The \vee is a 'V' not because it's the upside-down version of \wedge but because *vel* is Latin for *or*. Abu also noted that \wedge resembles a triangular tent. For such a tent to not fall it must have the left AND the right held up. Then, not to be outdone, Ila pointed out that \vee resembles a valley. If something is in a valley, it could be on one side, or the other side, or at the bottom on both sides.

p	\wedge	(q)	\vee	\neg	r
0	0	0	1	1	0
0	0	0	0	0	1
0	0	1	1	1	0
0	0	1	1	0	1
1	1	0	1	1	0
1	0	0	0	0	1
1	1	1	1	1	0
1	1	1	1	0	1

Table 2.5: Truth table for $p \wedge (q \vee \neg r)$

$(p \wedge q)$	\vee	\neg	r	
0	0	1	0	
0	0	0	1	
0	0	1	1	0
0	1	0	0	1
1	0	1	1	0
1	0	0	0	1
1	1	1	1	0
1	1	1	0	1

Table 2.6: Truth table for $(p \wedge q) \vee \neg r$

p	q	$p \wedge q$	$p \vee q$	$p \oplus q$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0
3	5	1	7	6

Table 2.7: Truth table for three binary logical operators with a summary line giving a decimal encoding of each column (important for Exercise answers).

Exercise 43 Construct a truth table for $p \wedge \neg q$.

Hint 43 Remember that \wedge is only true when both sides are true.

Exercise 44 Construct a truth table for the proposition $p \wedge (q \oplus \neg r)$.

Hint 44 Use the expanded or more compact form, whichever is easier.

AT A GLANCE

Exercise 45 Construct a truth table for the proposition $p \vee (\neg q \oplus \neg r)$.

Hint 45 It helps to copy an existing table and then modifying it — sometimes easier than starting from scratch.

AT A GLANCE

Exercise 46 Construct a truth table for the proposition $(p \vee \neg q) \oplus (\neg r \wedge \neg s)$.

Hint 46 With four variables your truth table will have $2^4 = 16$ rows.

AT A GLANCE

2.7 Go Visual

Abu and Ila were expecting this, but it still jarred. Til started right in, “To forge the link between sets and logic, let’s go visual and then formal. Drawing pictures helps us visualize the abstract, as set theorists and logicians discovered early on. The most common visual representation is the diagram named after its inventor.”⁸

⁸ John Venn, see here. In Venn diagrams, sets are usually represented as circles or ellipses. If they overlap, the shape of the region of overlap is *not* a circle or ellipse — it’s a convex lens-like shape (for a two-set diagram). It’s more consistent to make everything a rectangle.

"Figure 2.3 shows two sets, A drawn as a blue rectangle, B , drawn as a yellow rectangle, and their intersection the green rectangle."

"The union of A and B would be everything blue, per Figure 2.4."

Ila and Abu studied these two simple figures. "What if we made A and B smaller so they don't overlap?" said Abu.

Til made it so with Figure 2.5, "Good! That would mean A and B have an *empty intersection*." Ila said, "As in no elements in common?" "Yes," said Til. "And there's a term for that — two sets A and B are **disjoint** if A intersect B is empty. And if you have more than two sets, they are called **pairwise** (or **mutually**) **disjoint** if no two of them have a non-empty intersection. In other words, pick any two of many sets, the intersection of the pair of them is always empty. For example, in Figure 2.6, A , B , and C are mutually disjoint."

Exercise 47 Which pairs of the following sets are disjoint?

1. The set of all even numbers.
2. The set of all odd numbers.
3. The set of all nonnegative powers of 2.

Hint 47 Be careful with the third set.

per uñijie eneñy ofuer nownegatiue boñer ol ñ is eneñ.
atue hafitwiss qñisloñit pescarue tpe señorit boñer ol ñ is l, ari oqqa nunn-
AT 47 ANSWER Tpe eneñs ari qñisloñit tñow tpe oqqa. Mo oñuer tuo sets

Exercise 48 Which pairs of the following sets are disjoint?

1. The set of all white elephants.
2. The set of all gray pachyderms.
3. The set of all purple cows.

Hint 48 This is too easy.

AT 48 ANSWER

"Now we'll make these three sets bigger so they overlap, and thus are no longer pairwise disjoint, per Figure 2.7."

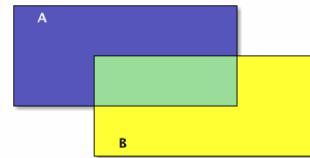


Figure 2.3: Set A overlapping Set B

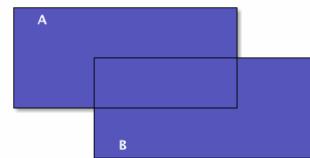


Figure 2.4: Blue union

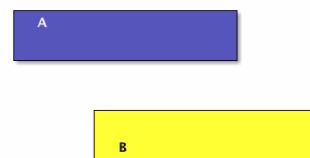


Figure 2.5: Set A not overlapping Set B

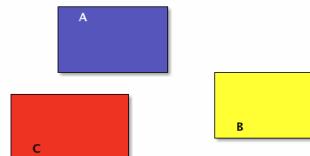


Figure 2.6: Sets A , B , and C , all disjoint from each other

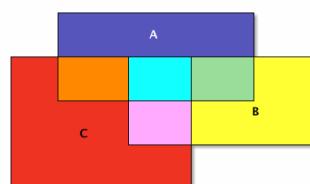


Figure 2.7: Sets A , B , and C , none disjoint from any other

Color	Set Operations
Red	$A \cap \bar{B} \cap C$
Yellow	$\bar{A} \cap B \cap \bar{C}$
Pink	$\bar{A} \cap B \cap C$
Blue	$A \cap \bar{B} \cap \bar{C}$
Orange	$A \cap \bar{B} \cap C$
Green	$A \cap B \cap \bar{C}$
Sky Blue	$A \cap B \cap C$

Table 2.8: Different-colored regions as set operations

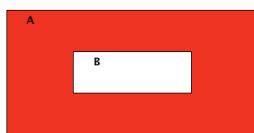


Figure 2.8: Set B in superset A

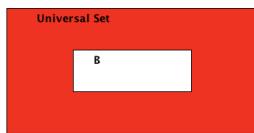


Figure 2.9: Set B 's complement is the red region — everything in the universe *except* what's in B .

The different-colored regions have different set operations that define them. Let's define the different set operations, and then show them in Table 2.8."

Ila thought that colorblindness would make this approach problematic, but was hesitant to bring that up.

"First let's go back to two sets," Til said, "and make B entirely contained in A . In other words, in Figure 2.8 every element of B is also an element of A (but not vice-versa)."

"Now note," said Til, "that in all of these diagrams, we haven't been showing — just left implied — *the Universal set* that encompasses all other sets. However, to understand set complement it helps to show an explicit Universal set. So let's make set A be that Universal set as shown in Figure 2.9."

Til cleared his throat. "Okay, we've done our warmups, now we're ready to formally define these set operations and see how they correspond to logic operations. If A and B are sets, then:

- The **union** of A and B , denoted $A \cup B$, is the set with members from A or B or *both*.
- The **intersection** of A and B , denoted $A \cap B$, is the set with members in common with A and B .
- The **difference** of A and B , denoted $A \setminus B$, is the set with members from A but *not* B .
- The **complement** of B , denoted \bar{B} , is the set with members *not* in B ."

Ila noticed Abu's eyes were glazing over, so she said, "Hey Abu, tell me if you agree that if an element is in A but not in B , or if it is in B but not in A , then it is *not* in the **intersection** of A and B !"

Abu struggled for a few seconds, trying to grasp Ila's question, but finally nodded his agreement.

Til beamed. "Excellent, you two!" Now it was Abu's and Ila's turn to beam. "Now let's add numbers to identify the regions so we can cross reference them in Table 2.9."

"For bearing with me in some tedious but necessary exposition," Til said, "I have another tiny assignment for you as we conclude for today."

#	Color	Set Ops	A	B	C
1	Red	$A \cap \bar{B} \cap C$	0	0	1
2	Yellow	$\bar{A} \cap B \cap \bar{C}$	0	1	0
3	Pink	$\bar{A} \cap B \cap C$	0	1	1
4	Blue	$A \cap \bar{B} \cap \bar{C}$	1	0	0
5	Orange	$A \cap \bar{B} \cap C$	1	0	1
6	Green	$A \cap B \cap \bar{C}$	1	1	0
7	Sky Blue	$A \cap B \cap C$	1	1	1

Table 2.9: Numbered different-colored regions as set operations with three additional columns

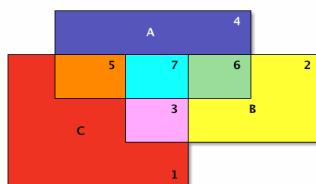


Figure 2.10: Sets A , B , and C , not disjoint, numbered

Exercise 49 What is the correlation between the first three columns and the last three columns of Table 2.9?

a *subset*.

Hint 49 The first three columns give three different ways to identify

Exercise 50 What is the set difference between the set of letters in the English alphabet and the set of letters in the Hawaiian alphabet?

Himt 50 The Hawaiian alphabet has one less than half as many letters.

AT&T 20 Years Ago

2.8 Propositional Membership

As they settled in for their next session, Ila and Abu were determined to engage Til in some small talk, but to no avail! He ignored their attempts and started his spiel.

“Let p be the proposition ‘ $x \in A$ ’ and let q be the proposition ‘ $x \in B$,’” Til rattled off, with obvious relish. “Recall that ‘ \in ’ means ‘is in’ and ‘ \notin ’ means ‘is not in’ and observe:” (See Table 2.10.)

Logical Statement	Equivalent Set Operation Statement
$\neg p$	$x \notin A$ (or $x \in \bar{A}$)
$p \vee q$	$x \in A \cup B$
$p \wedge q$	$x \in A \cap B$
$p \wedge \neg q$	$x \in A \setminus B$ (or $x \in A \cap \bar{B}$)
$p \oplus q$	$x \in (A \cup B) \setminus (A \cap B)$

Table 2.10: Logical and equivalent set operation statements

“The last row in Table 2.10, the \oplus (**xor**) logical operator, corresponds to what is called the **symmetric difference** of sets A and B . With that, it’s time to bring De Morgan’s laws for logic **and** sets into the picture. But first, let’s make explicit the connection between logical and set operations.” (See Table 2.11.)

Logic	Symbol	Symbol	Set
conjunction	\wedge	\cap	intersection
disjunction	\vee	\cup	union
negation	\neg	—	complement

Table 2.11: The correspondence between logical and set operations

“Note carefully — De Morgan’s laws for logic state that the negation of a conjunction is the disjunction of negations; likewise, the negation of a disjunction is the conjunction of negations. De Morgan’s laws for sets state that the complement of an intersection is the union of complements; likewise, the complement of a union is the intersection of complements. (See Table 2.12.)”

Table 2.12: Logic and Sets Equivalence

Logic	Equivalent Logic	Sets	Equivalent Sets
$\neg(p \wedge q)$	$\neg p \vee \neg q$	$A \cap B$	$\overline{A \cup B}$
$\neg(p \vee q)$	$\neg p \wedge \neg q$	$A \cup B$	$\overline{A \cap B}$
$p \oplus q$	$(p \vee q) \wedge \neg(p \wedge q)$	$(A \cup B) \cap \overline{A \cap B}$	$(A \cup B) \cap (\overline{A} \cup \overline{B})$

“Let’s visualize this in pictures. The symmetric difference of A and B is everything red (see Figure 2.11), and, as I said, this set operation corresponds directly to the \oplus (**xor**) logical operation.”

“Let’s redraw our colorful Venn diagram to emphasize the separation between each numbered (colored) region — separate in the sense that each is a non-overlapping subset (see Figure 2.12).”

Table 2.13: Logic and sets connection

Set	As Union of	Logic \equiv Set Operation Description
A	4 \cup 5 \cup 7 \cup 6	$(p \wedge \neg q \wedge \neg r) \vee (p \wedge \neg q \wedge r) \vee (p \wedge q \wedge r) \vee (p \wedge q \wedge \neg r)$
B	2 \cup 3 \cup 7 \cup 6	$(\neg p \wedge q \wedge \neg r) \vee (\neg p \wedge q \wedge r) \vee (p \wedge q \wedge r) \vee (p \wedge q \wedge \neg r)$
C	1 \cup 3 \cup 5 \cup 7	$(\neg p \wedge \neg q \wedge r) \vee (\neg p \wedge q \wedge r) \vee (p \wedge \neg q \wedge r) \vee (p \wedge q \wedge r)$

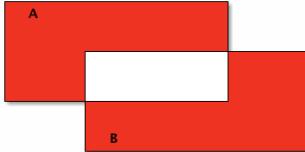
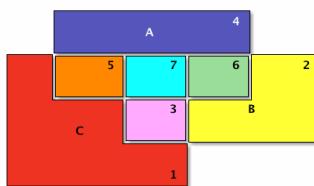


Figure 2.11: The symmetric difference of sets A and B . The upper red region corresponds to $A \setminus B$, and the lower red region corresponds to $B \setminus A$. The union of these two differences is the symmetric difference, just as the exclusive or is one or the other, but not both.

“Given how clearly you can now see the separate numbered regions, I’m sure you can reconstruct each set in terms of unions of those regions. For homework!”

Exercise 51 Explain the connection between logic and sets in Table 2.13 where

- p is the proposition ‘ $x \in A$ ’;
- q is ‘ $x \in B$; and
- $r = ‘x \in C’.$



Hint 51 Provide an explanation for the equivalence of the logical and set operation descriptions (columns 3 and 2) for each set (column 1) in the table. So, use wording like “Set A is explained as the union of numbered regions 4, 5, 7 and 6, because . . .”

Figure 2.12: Not disjoint but separated visually

Exercise 52 *Let*

- set $A = \{\text{'verve'}, \text{'vim'}, \text{'vigor'}\}$, and
 - set $B = \{\text{'butter'}, \text{'vinegar'}, \text{'pepper'}, \text{'vigor'}\}$.

For each of the following set operations, give its resulting members as a set of strings:

1. The set of words that are in $A \cup B$; call this set C .
 2. The set of words that are in $A \cap B$; call this set D .
 3. The subset of set C of words that start with 'v'.
 4. The subset of set C of words that end with 'r'.
 5. The subset of set C of words that start with 'v' and end with 'r'.
 6. The subset of set D of words that have six letters.

Hint 52 This is very straightforward. But see if you can write some Python code to help you find the answers!

ATLAS 29 TWSRA

2.9 Conditional Containment

“Our time today is very limited, so let’s dive right in.” Though short on time, Til was feeling a little more loquacious than usual, so he just started reeling off a flurry of definitions:

“The scenario that set B is entirely contained in set A — recall Figure 2.8 — has another shorter way to say it that involves a conditional **if-then** statement.”

- “ B is a **subset** of A ($B \subseteq A$) means that **if** $x \in B$ **then** $x \in A$.”
 - “ A is a **superset** of B ($A \supseteq B$) means that B is a **subset** of A .”
 - “ B is a **proper subset** of A ($B \subset A$) means that there is some element

of A that is not in B . In other words, A is strictly speaking ‘bigger’ than B .’

- “ A is a **proper superset** of B ($A \supset B$) means that B is a **proper subset** of A .”

“With these definitions in mind, please study Table 2.14 very carefully.”

Ila thought, “And then get ready to do homework — lots of homework!” — right before Til said essentially that, right after he gave them a quick explanation of the *conditional* operator (\rightarrow), which he mysteriously said was pronounced *only if*, and the *biconditional* operator (\leftrightarrow), which he said was pronounced *if and only if*. Then he dismissed them and that was that!

Table 2.14: The conditional logic of set containment. The last row’s biconditional means that if each of two sets is a subset of the other, then the sets are equal — they are the same set. In terms of numbers, this is analogous to $(a \leq b \wedge b \leq a) \rightarrow a = b$, which suggests viewing the subset relation as a “less-than-or-equal-to” relation, which mirrors the similarity of the symbols. So $A \subseteq B \Leftrightarrow a \leq b$, where a is the *size of* (number of elements in) A , written as $a = |A|$ and ditto for $b = |B|$. This is **not** the absolute value of A or B — so note the overloaded use of the vertical bars here.

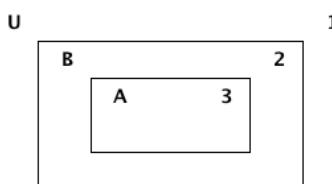


Figure 2.13: Set-subset conditional connection

Logical Statement	Equivalent Set Statement
$p \rightarrow q$	$A \subseteq B$
$p \leftarrow q$ (or $q \rightarrow p$)	$A \supseteq B$ (or $B \subseteq A$)
$p \leftrightarrow q$	$(A \subseteq B) \wedge (B \subseteq A)$ (or $A = B$)

Exercise 53 Here are some Python assignment statements illustrating subset size compared to superset size:

```
A = {1, 3, 7, 9}
a = 4
B = {1, 3, 6, 7, 8, 9, 10, 21, 42, 63, 68, 76}
b = 12
A_is_a_subset_of_B = True
a_is_less_than_or_equal_to_b = True
```

Revise this sequence of assignments to avoid using the literals 4, 12 and `True`, instead replacing them with calls to built-in Python functions or operators.

Hint 53 Any decent Python documentation website will reveal these.

ANSWER 23 The `len` and `issubset` functions’ `<=` operator.

Exercise 54 The Venn diagram in Figure 2.13 shows three numbered regions of Universal set U and set B with a superset-subset-of relationship with set A .

Make a connection between the logical conditional operator (\rightarrow) and the definition of a subset. Refer to the three numbered regions in your answer.

Hint 54 Making this connection means: (1) express the logical connection in terms of the three numbered regions. More emphatically, making this connection will help you understand why the conditional operator works the way it does. To answer fully, you must give the definition of a subset, and you must refer to the three numbered regions in the Venn diagram. whichever makes the most sense to you. Express either (or both) in subset in terms of the definition of the logical conditional operator; dittoinal operator in terms of the definition of subset, or (2) express terms of the three numbered regions.

AT A GLANCE ANSWERS

Exercise 55 Let p be the proposition “I studied.” and let q be the proposition “I got an A on the test.”

Express the proposition $p \rightarrow q$ as an English sentence (“anglify”).

Hint 55 Use if-then keywords.

AT A GLANCE ANSWERS

Exercise 56 Anglify $\neg p \vee \neg q$.

Hint 56 Use the same p and q .

AT A GLANCE ANSWERS

Exercise 57 Anglify $\neg p \rightarrow (p \vee q)$

Hint 57 It doesn't have to make sense!

AT A GLANCE ANSWERS

Exercise 58 Anglify $\neg p \rightarrow \neg q$.

Hint 58 Still use the same p and q .

AT A GLANCE ANSWERS

Exercise 59 Let p be the proposition “You applied for admission at BYU-Idaho.” and let q be the proposition “You were accepted.”

Express “You applied for admission at BYU-Idaho and were accepted.” as a symbolic compound proposition using logical connectives (“symbolize”).

Hint 59 Don't overthink this.

ANSWER $\neg q \vee p$

Exercise 60 Symbolize “You did not apply for admission at BYU-Idaho but were still accepted.”

Hint 60 Use p and q as before.

ANSWER $\neg p \wedge q$

Exercise 61 Symbolize “You applied for admission at BYU-Idaho but were not accepted.”

Hint 61 That should not happen!

ANSWER $\neg p \vee \neg q$

Exercise 62 Symbolize “Either you did not apply for admission at BYU-Idaho and didn't get accepted or you did apply and got accepted.”

Hint 62 That goes without saying.

ANSWER $\neg p \vee q$

Exercise 63 There are many different ways in English to put the same conditional statement, ‘if p then q ’. Here are four:

1. $p \rightarrow q$ (pronounced) p only if q .
2. omit the ‘then’: if p , q .
3. reversed: q if p .

4. ‘whenever’ means ‘if’: whenever p , q (or q whenever p).

Find at least four other ways.

Hint 63 Versus sufficient (sufficiency).

Hint 63 Some have to do with conditions being necessary (necessity)

- 8. $d \rightarrow b$ is equivalent to b .
- 7. b entails d .
- 6. d is implied by b .
- 5. b implies d .
- 4. b is a sufficient condition for d .
- 3. b is sufficient for d .
- 2. b is a necessary condition for d .
- 1. b is necessary for d .

Exercise 63 **ANSWER**

Exercise 64 Symbolize “Your applying for admission at BYU-Idaho is necessary for your being accepted.”

Hint 64 Did you do the previous exercises?

ANSWER

Exercise 65 Symbolize “Your being accepted is a sufficient condition for your applying for admission at BYU-Idaho.”

Hint 65 Don’t confuse sufficiency with necessity.

ANSWER $d \rightarrow b$.

Exercise 66 Symbolize “Your applying for admission at BYU-Idaho is necessary and sufficient for your being accepted.”

Hint 66 This is usually the case.

ANSWER

2.10 Understanding Conditional Logic

Ila huffed, “I mostly get the logical connectives, but why we don’t just call them operators is what I don’t get. I mean, they’re the same as the logical operators in JavaScript, and Python too, and using two different words for the same thing is just confusing.” Abu thought Ila tended to complain a lot when she was stressed. He also thought that the strain of their homework was getting to her.

Ila vented some more. “I also don’t get the conditional connective, operator, whatever you want to call it. It just seems illogical that ‘p only if q’ is — how did Til put it?”

“I believe he said ‘p only if q is true, except when p is more true than q,’” said Abu.

“That’s what I’m talking about, what does ‘more true than’ mean? Something is either true or it’s not. How can it be ‘more true’? Truer than true?!”

“Well, here’s how I see it,” said Abu. “When they have the same truth value, both true or both false, neither is more or less than the other, they’re equal.”

Ila blushed. “Oh — now I get it. A true p is greater than a false q. And the last case, when p is false and q is true, p is less true than q. Way less, because it has zero truth.”

“Irrelevant to the truth of q,” said Abu. “I drew its truth table, like Til asked, and only one row is false, the rest true. Perhaps you drew it this way too?”

p	q	$p \rightarrow q$
0	0	1
0	1	1
1	0	0
1	1	1

Ila furrowed her brows. “Mine has the last column centered under the arrow, but otherwise it’s the same as yours.”

“I like centered better,” said Abu. “Til will be glad we get it, I think, but do you think he was going to explain it some more?”

“He will if we ask, but — that look in his eye that said I just explained this in the simplest, most understandable way — I really don’t want to ask him,” said Ila.

“I’ll do it,” said Abu. “I don’t mind appearing ignorant, because appearances aren’t deceiving in my case — I really *am* ignorant!”

“Well, I hate asking dumb questions,” said Ila.

Abu winked and said, “Never been a problem for me!”

Exercise 67 True or false? “If $2 + 2 = 4$, then pigs can fly.”

Hint 67 Check your PCK (Procedure Capability Knowledge)!

• اسلافی سی.

„اے جو میرے ساتھ ہے“ تجھے سمجھا دے، اسی ساتھ سمجھا دے جو میرے ساتھ ہے۔

Exercise 68 True or false? “If $2 + 7 = 5$, then Elvis Presley is still alive.”

Fielde!

Hint 68 Check your arithmetic, and your RDF (Relativity Distortion)

AT A ASA 83 تہذیب

Exercise 69 True or false? “If pigs can fly, then dogs can’t fly.”

Plying is not one of them.

Hint 69 Pigs can do a lot of things that dogs can't do, and vice versa.

اسلافی سی → اسلافی سی اور اسی سے

اے جو میرے ساتھ ہے، اسی ساتھ میرے ساتھ ہے، اسی ساتھ میرے ساتھ ہے۔

Exercise 70 True or false? “If dogs have four legs, then ostriches have two legs.”

Hint 70 Normally speaking, that is.

AT A ASA 07 تہذیب

Exercise 71 True or false? “ $2 + 1 = 3$ if and only if $1 + 2 = 3$.”

Hint 71 This is not hard.

اے جو میرے ساتھ ہے، اسی ساتھ میرے ساتھ ہے، اسی ساتھ میرے ساتھ ہے۔

Exercise 72 True or false? “ $1 + 2 = 3$ if and only if $3 + 1 = 6$.”

Hint 72 This is easy.

AT A GLANCE

Exercise 73 True or false? “ $1 + 3 = 2$ if and only if the earth is flat.”

Hint 73 Flat-earthers need not apply.

AT A GLANCE

Exercise 74 True or false? “ $1 < 2$ if and only if $2 < 3$.”

Hint 74 I'm convinced. Are you?

AT A GLANCE

Exercise 75 Write “It is necessary to sign up to win the contest.” as an ‘If p then q ’ statement (“symbolize-as-if-then”).

Hint 75 The necessary thing is the outermost circle.

AT A GLANCE

Exercise 76 Symbolize-as-if-then “I get a cold whenever I go outside.”

Hint 76 Whenever means if, remember?

AT A GLANCE

Exercise 77 Symbolize-as-if-then “It is sufficient to be an A student to receive the scholarship.”

Hint 77 If that were only true!

ANSWER 77 It is sufficient to be an A student to receive the scholarship.

Exercise 78 Symbolize-as-if-then “As long as you leave now, you will get there on time.”

Hint 78 Think about it.

ANSWER 78 As long as you leave now, you will get there on time.

Exercise 79 Symbolize-as-if-then “I'll get half off if I act now.”

Hint 79 Pretty straightforward.

ANSWER 79 I'll get half off if I act now.

2.11 Conditional Vocabulary

Abu greeted Ila and they chatted briefly before opening Til's latest message together.

“Some important terms and notation that you need to know when analyzing conditionals are:”

- Dual
- Antecedent
- Consequent
- Vacuous
- Trivial
- Set-builder notation

“Something is the ‘dual’ of something else when they have a tightly paired relationship of operating or being treated similarly. The **Duality Principle** says you can interchange duals in a valid Boolean expression and the result is also valid. For example, De Morgan's laws of negation of conjunctions and disjunctions are duals of each other, because \wedge and \vee are duals. Also, 0 (false) is the dual of 1 (true), and vice versa, and \neg is its own dual (self-dual).”

“The **antecedent** is what comes before the conditional operator, and the **consequent** is what comes after it. For example, in ‘If sunset heralds night then sunrise heralds day’ the antecedent is ‘sunset heralds night’ and ‘sunrise heralds day’ is the consequent.”

“The terms **vacuously true** and **trivially true** are a little subtle. Since the conditional $p \rightarrow q$ is true whenever the antecedent p is false, it is called a *vacuous* (empty, void) truth, because any consequent whatsoever follows from a false antecedent. The dual of a vacuously true conditional is one that is trivially true. This is when the consequent q is true — because whether or not the antecedent is true has no effect on the conditional — it is always (trivially) true.”

The next part really caught Ila’s attention, as she had been studying Python’s “list and set comprehensions” and sharing her findings with Abu — who found these ideas more fascinating the more he looked at them.

“Next,” wrote Til, “I want to say something about **set-builder notation**. This notation essentially describes the process for building a set by giving a conditional expression (which is why I’m lumping this with other vocabulary about conditionals) that governs membership in a set. For example, $\{x \mid x \geq 13 \wedge x \leq 19\}$ is the set of numbers in the teens. The vertical bar separates the description of a generic member of the set (x in this example) from the conditional expressed in terms of that generic member. I said numbers, but I meant integers more specifically. So let’s narrow the scope of the conditional expression by putting the domain before the vertical bar: $\{x \in \mathbb{Z} \mid x \geq 13 \wedge x \leq 19\}$ Of course, it could also be written as $\{x \mid x \in \mathbb{Z} \wedge x \geq 13 \wedge x \leq 19\}$. The complement of the set of numbers in the teens is $\{x \in \mathbb{Z} \mid x < 13 \vee x > 19\}$, but I will also point out that set-builder notation can be problematic. For example, $\{x \in \mathbb{Z} \mid x < 13 \wedge x > 19\}$, which is a convoluted way to specify the empty set, because the condition is contradictory. Another way this notation could be problematic is if the condition for membership is infeasible to evaluate — meaning it could take forever. For example, $\{x \in \mathbb{Z} \mid x \text{ is an odd perfect number}\}$. There are many known *even* perfect numbers⁹ but nobody has ever found an *odd* one. Nor can anyone (yet) prove that there are none. Nor can anyone (yet) prove that there are infinitely many perfect numbers, although, as Euclid proved centuries ago, all the even ones have a certain form. But I digress. We’ll come back to primes and Euclid and other fascinating number facts later.”

“Also important to know are the three variants of the conditional statement,” Til continued, “known as the **converse**, the **inverse** and the **contrapositive**:

- The *straight* conditional is $p \rightarrow q$ (not a variant);
- its *converse* is $q \rightarrow p$ (swap antecedent and consequent);
- its *inverse* is $\neg p \rightarrow \neg q$ (negate both antecedent and consequent); and,
- its *contrapositive* is $\neg q \rightarrow \neg p$ (the inverse of the converse).

⁹ 6, 28, and others like $2^{p-1}(2^p - 1)$ where $2^p - 1$ is prime. 6 is of this form when $p = 2$, and 28 is when $p = 3$. See [the definition and some history](#) for more information.

(See also Figure 2.14.)”

Exercise 80 Write the converse, inverse, and contrapositive of (“variantize”) “If it rains today, we won’t go to the park.”

Hint 80 Symbolize first, then form the variants, then translate the symbols back into words.

AT A SAA 80 TAWSMIA

Exercise 81 Variantize “If you do your homework, I’ll give you a pat on the back.”

Hint 81 Symbolize first, then form the variants, then translate the symbols back into words.

mon, f gñue lñor a bñat ou tñre ñracf, tñren lñor qm, f qo lñor fmouemoukf. „
tñren I mon, f gñue lñor a bñat ou tñre ñracf.“ „The contrapositive is “If I
qo lñor fmouemoukf.“ „The inverse is “If I gñue lñor qm, f qo lñor fmouemoukf,
AT A SAA 81 TAWSMIA“ „The converse is “If I gñue lñor a bñat ou tñre ñracf, tñren lñor

Exercise 82 Variantize “Whenever I babysit, I get sick.”

Hint 82 You should be getting the hang of it.

AT A SAA 82 TAWSMIA

Exercise 83 Variantize “Every time there is a quiz, I go to class.”

Hint 83 It’s as though correlation implies causation.

contrapositive is “If I qm, f gñ o to class, tñren tñre is m, f a dmñ.“ „
The inverse is “If tñre is m, f a dmñ, tñren I qm, f gñ o to class.“ „The
AT A SAA 83 TAWSMIA“ „The converse is “If I gñ o to class, tñren tñre is a dmñ.“ „

```
def cond(p, q):
    """straight conditional
    """
    return (not p) or q

def converse(p, q):
    """swaps p and q
    """
    return cond(q, p)

def inverse(p, q):
    """negates p and q
    """
    return cond(not p, not q)

def contrapositive(p, q):
    """swaps and negates
    p and q
    """
    return cond(not q, not p)
```

Figure 2.14: The conditional and its variants implemented

Exercise 84 Variantize “I wake up late when I stay up past my bed-time.”

Hint 84 Go to bed on time then!

AT a fSA 48 1ewsaA

Exercise 85 Construct a truth table for $p \rightarrow \neg q$.

Hint 85 “NAND”.

AT a fSA 48 1ewsaA
fut
for the last row we have b and d are both

Exercise 86 Construct a truth table for $p \oplus q$.

Hint 86 Zero if either both or neither

AT a fSA 48 1ewsaA

Exercise 87 Construct a truth table for $\neg q \rightarrow q$.

Hint 87 Remember, true > false.

AT a fSA 48 1ewsaA
isnt si p nebyo si si T

Exercise 88 Construct a truth table for $p \vee \neg q$.

Hint 88 Think $p \rightarrow q$.

AT a fSA 48 1ewsaA

Exercise 89 Construct a truth table for $p \rightarrow (\neg p)$.

Hint 89 Refer to Exercise 87.

ANSWER 8 Essentially the same as the answer to Exercise 8.

Exercise 90 Construct a truth table for $p \leftrightarrow q$.

Hint 90 The implication goes both ways.

ANSWER 9

Exercise 91 Construct a truth table for $p \leftrightarrow (\neg p)$.

Hint 91 “A contradiction — always.”

ANSWER 10 If p is false — a contradiction — if p is true.

Exercise 92 Construct a truth table for $(p \rightarrow q) \rightarrow (q \rightarrow r) \rightarrow (r \rightarrow s)$.

Hint 92 Take it one step at a time.

ANSWER 11

Exercise 93 Construct a truth table for $((p \vee q) \wedge (\neg p \vee r)) \rightarrow (q \vee r)$.

Hint 93 You will be taught another name for this pattern.

ANSWER 12 You will be taught another name for this pattern.

2.12 Logical Equivalence

“Since we can’t meet this week, I have some very important points to make and then some more exercises for you to do,” Til’s message said. “Being a logical language, the **propositional calculus** (another name for the propositional logic) obeys certain laws. For example, you’ve already seen De Morgan’s laws. There are also the *commutative*, the *associative*, and the *distributive* laws, as well as some miscellaneous laws known as *idempotence*, *absorption*, and *double negation*. But the most important law is that some propositions are *equivalent* to each other.”

“So, we say that two compound propositions p and q are **logically equivalent** when they have the same truth values under every model. This equivalence is expressed symbolically as ‘ $p \equiv q$ ’ and operationally by saying that the biconditional ‘ $p \leftrightarrow q$ ’ is always true.”

“There are three terms for this type of *iff* (*if and only if*) expression, its opposite, and neither it nor its opposite:

- A **tautology** is a proposition that is **always true**. For example, $p \vee \neg p$.
- A **contradiction** is a proposition that is **always false**. For example, $p \wedge \neg p$.
- A **contingency** is a proposition that is neither a tautology nor a contradiction, but is **sometimes true, sometimes false**. For example, $p \rightarrow q$.”

Exercise 94 Use truth tables to verify De Morgan’s laws:

1. $\neg(p \vee q) \equiv \neg p \wedge \neg q$
2. $\neg(p \wedge q) \equiv \neg p \vee \neg q$

Hint 94 You’ve had practice with previous exercises.

AT A FSA GE TAWSEN

Exercise 95 Use truth tables to verify the commutative laws.

1. $p \vee q \equiv q \vee p$
2. $p \wedge q \equiv q \wedge p$

Hint 95 To “commute” means “to exchange.”

AT A FSA GE TAWSEN

Exercise 96 Use truth tables to verify the associative laws.

1. $(p \vee q) \vee r \equiv p \vee (q \vee r)$
2. $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$

Hint 96 It’s just like addition or multiplication.

AT A FSA GE TAWSEN

Exercise 97 Use truth tables to verify the distributive laws.

1. $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$.
2. $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$.

Hint 97 Think “and” false if not p , “or” true if p .

ANSWER Both are tautologies.

Exercise 98 Use truth tables to verify some miscellaneous laws, letting $1 = \text{true}$, $0 = \text{false}$:

1. $p \wedge 1 \equiv p$ (*idempotence*)
2. $p \vee 0 \equiv p$ (*idempotence*)
3. $p \wedge 0 \equiv 0$ (*absorption*)
4. $p \vee 1 \equiv 1$ (*absorption*)
5. $\neg\neg p \equiv p$ (*double negation*)

Hint 98 These are simpler than most truth tables.

ANSWER

Exercise 99 Match the following set identities with their counterparts in the miscellaneous logic laws, letting U be the Universal set:

1. $A \cap U = A$.
2. $A \cup U = U$.
3. $A \cup \emptyset = A$.
4. $\underline{\underline{A}} \cap \emptyset = \emptyset$.
5. $\underline{\underline{A}} = A$.

Hint 99 The miscellaneous logic laws are shown in Exercise 98.

ANSWER

Answers correspond to each logic law in Exercise 98.

Exercise 100 Use De Morgan’s Laws to find the negations of the propositions:

1. Winning the first round is necessary for winning the trophy.
2. Winning the tournament is sufficient for winning the trophy.

3. Getting an A on the final exam is necessary and sufficient for passing this class.

Hint 100 See Exercise 63 to remind yourself of the difference between necessary and sufficient.

AT a **AS A 101 TAWNSA**

Exercise 101 Show that $(p \leftrightarrow q) \wedge (p \leftrightarrow r)$ and $p \leftrightarrow (q \wedge r)$ are not logically equivalent (i.e., the biconditional does not distribute over and).

Hint 101 A truth table is useful again here.

are false.

AT a **AS A 101 TAWNSA**

Exercise 102 Show that $\neg(p \leftrightarrow q)$ and $p \leftrightarrow \neg q$ are logically equivalent.

Hint 102 Don't think too hard about it.

AT a **AS A 101 TAWNSA**

Exercise 103 Determine if $((p \vee q) \wedge (\neg p \vee r)) \rightarrow (q \vee r)$ is a tautology.

Hint 103 Try using a spreadsheet.

AT a **AS A 101 TAWNSA**

Exercise 104 Find a compound proposition involving the propositional variables p , q , and r that is true when p and q are true and r is false, but is false otherwise.

Hint 104 It's exactly as described.

AT a **AS A 101 TAWNSA**

2.13 Explanation and Implementation

Abu finally got to ask their question about the conditional. Ila was at her most alert.

“Think of it this way,” said Til. “Inside versus outside.¹⁰ In your mind’s eye, visualize a couple of circles, two different sizes, the smaller inside the larger. Make them concentric. Okay? Now consider the possibilities. Can something be inside both circles simultaneously?”

Abu and Ila nodded together.

“That’s your p-and-q-both-true case. Now, can something be outside both circles?”

“Yes, obviously,” said Ila. Til was studying both their faces carefully. “That’s your p-and-q-both-false case,” he said. “And can something be inside the outer circle without being inside the inner one, Abu?”

“Yes,” said Abu, hesitantly — “that’s when p is false and q is true, I think.”

Ila felt a tingle go up her spine. She blurted out, “I see it! The only impossibility is having something inside the inner circle but not inside the outer circle — p true and q false. Impossible means false — makes perfect sense!”

Til beamed. “Excellent, Ila! How about you, Abu? Make sense?”

Abu grinned extra wide and nodded his head.

Exercise 105 Figure 2.14 shows the implementation of the conditional operator in terms of its simplest logical equivalence. Figure 2.15 shows the implementation of the biconditional (iff) operator $p \leftrightarrow q$ in terms of the conditional operator.

Find two shorter ways to correctly implement the biconditional function.

Hint 105 Use something you have seen before (see Figure 2.1).

second even sforfer mān niffl refuru b == d;

Answer 102 Use first sforfer mān niffl refuru xor(b, d) and true

Exercise 106 This is an exercise in implementing logic gate Boolean functions to generate “personalities” (output strings) of compound Boolean functions.

Specifically, your task is to implement the `not1`, `or2` and `and2` functions, **arithmetically** (i.e., do not use any conditional logic — `if` or the like). Use **only** the operators `+`, `-`, and `*`. The inputs to these three functions will be zeros or ones only. You’ll know you got it right

¹⁰ Now is a good time to reread the second paragraph in the Preface.

```
def iff(p, q):
    """biconditional:
       p if and only if q.
    """
    return cond(p, q) and
           cond(q, p)
```

Figure 2.15: The biconditional operator implemented in terms of the conditional operator

if True is printed when running the code:

```

def f1(p, q, r):
    return or2(and2(p, q), not1(r))

def f2(p, q, r):
    return and2(p, or2(q, not1(r)))

def f3(p, q, r):
    return or2(p, and2(q, r))

def personality3(boolfunc3):
    outputs = ''
    i1 = 0
    while i1 <= 1:
        i2 = 0
        while i2 <= 1:
            i3 = 0
            while i3 <= 1:
                outputs += str(boolfunc3(i1, i2, i3))
                i3 += 1
            i2 += 1
        i1 += 1
    return outputs

print(personality3(f1) == '10101011' and
      personality3(f2) == '00001011' and
      personality3(f3) == '00011111')

```

Hint 106 Don't be afraid to work in groups!

AT A GLANCE

2.14 Summary of Terms and Definitions

A **set** is a **collection** of **objects**, where objects are individual, discrete, separate things that can be named or identified somehow.

An object belonging to a **set** is called a **member** or an **element** of the set. In symbols, for a set S and object x :

$$x \in S$$

is the statement " x is an element (or member) of S " — or shorter, " x is in S ". For non-member y , the statement " y is not in S " is symbolized as

$y \notin S$.

\emptyset denotes the empty set, the set with no elements.

0 is the number of elements in the empty set.

$x \in \emptyset$ is false for every element x .

$y \notin \emptyset$ is true of every element y .

A **proposition** is a sentence (or statement) that is either true or false.

A **truth value** is one of True or False—not neither, and not both.

A **model** is the set of meanings given to symbols. In propositional logic, models are **assignments** of truth values to propositions.

An **assignment** is the pairing of a proposition with either a True or a False truth value.

A **truth table** is a tabulation of possible models. It has 2^n rows, where n is the number of propositional variables, and as many columns as needed to show the inside-out evaluation of each (sub)proposition.

The **union** of A and B , denoted $A \cup B$, is the set with members from A or B or both.

The **intersection** of A and B , denoted $A \cap B$, is the set with members in common with A and B .

The **difference** of A and B , denoted $A \setminus B$, is the set with members from A but *not* B .

The **complement** of B , denoted \overline{B} , is the set with members *not* in B .

The **xor** or **symmetric difference** of A and B is the set difference $(A \cup B) \setminus (A \cap B)$.

Two sets A and B are **disjoint** if $A \cap B = \emptyset$.

More than two sets are **pairwise** (or **mutually disjoint**) if no two of them have a non-empty intersection.

B is a **subset** of A , denoted $B \subseteq A$, means that if $x \in B$ then $x \in A$.

A is a **superset** of B , denoted $A \supseteq B$, means that B is a **subset** of A .

B is a **proper subset** of A , denoted $B \subset A$, means that there is some element of A that is not in B .

A is a **proper superset** of B , denoted $A \supset B$, means that B is a **proper subset** of A .

$|A|$ denotes the **size of** (number of elements in) set A .

The **Duality Principle** deals with duals — \wedge is the dual of \vee , 0 (false) is the dual of 1 (true), and vice versa, and \neg is its own dual (self-dual) — and says if duals are interchanged in a valid Boolean expression the result is also valid.

The **antecedent** is the p in a $p \rightarrow q$ conditional.

The **consequent** is the q in a $p \rightarrow q$ conditional.

A **vacuously true** conditional $p \rightarrow q$ is when p is false.

A **trivially true** conditional $p \rightarrow q$ is when q is true.

set-builder notation describes how to build a set by giving a condition-for-membership to control what goes in the set, e.g., the set of teens: $\{x \mid x \geq 13 \wedge x \leq 19\}$.

The **straight conditional** is $p \rightarrow q$.

The **converse** is $q \rightarrow p$ (swap antecedent and consequent).

The **inverse** is $\neg p \rightarrow \neg q$ (negate both antecedent and consequent).

The **contrapositive** is $\neg q \rightarrow \neg p$ (the inverse of the converse).

A **tautology** is a proposition that is **always true**.

A **contradiction** is a proposition that is **always false**.

A **contingency** is a proposition that is neither a tautology nor a contradiction, but is **sometimes true, sometimes false**.

Two compound propositions p and q are **logically equivalent**, signified $p \equiv q$ when they have the same truth values under every model. It means that $p \leftrightarrow q$ is a tautology.

p	q	0	$p \wedge q$	$p \wedge \neg q$	$\neg p$	$\neg p \wedge q$	q	$p \otimes q$	$p \vee q$	$\neg(p \vee q)$	$p \equiv q$	$\neg q$	$q \rightarrow p$	$\neg p$	$p \rightarrow q$	$\neg(\neg(p \wedge q))$	1
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	1	0	0	0	1	1	1	1	0	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Figure 2.16: All truth tables for two propositional variables, p and q

3

Functions

3.1 Encapsulation for Fun and Profit

“Let’s talk about functions today, shall we?” said Til, jumping right in. Ila stared with wonder and amazement. It was like Til had no problem whatsoever dispensing with the normal pleasantries of conversation. Like that time when she arrived a few minutes before Abu and tried to make small talk with Til. “So, it seems that you’re interested in stars!” she had said. “Yes, I am.” Til had replied, and then got that faraway look in his eyes, like he was suddenly out there among the stars, abruptly leaving the nearby flesh-and-blood human behind.

Like he had that time by arriving with a cheery greeting, Abu broke the awkward silence this time by saying “Are we talking about Python functions?”

“Partly, yes,” said Til. “You’ve already seen some of those, so we’ll continue to use them as a vehicle for ‘concretizing’ the abstract mathematical notion of a function.”

Abu and Ila studied the figures Til was displaying.

“I have in Figure 3.1,” Til continued, “a sample function that is defined and *invoked* — another way to say *called*. This definition and invocation have the same effect as what the code in Figure 3.2 does.”

```
def list_some_computations_on(a, b, c, d):
    """Add, divide, subtract, multiply
    four numbers pairwise in a certain
    fashion.
    """
    return [a + b, d / b, d - a, c * d]

print(list_some_computations_on(1, 2, 3, 4))
```

Figure 3.1: A function to list some computations on four numbers. Calling it with those numbers will print the output [3, 2.0, 3, 12].

```
a = 1
b = 2
c = 3
d = 4
print([a + b, d / b, d - a, c * d])
```

Figure 3.2: Variables storing numbers and then some computations performed on them. The same result ([3, 2.0, 3, 12]) will be printed out.

Til turned and said, “Abu, you’re not a programmer – not yet anyway! Why do you think the first way is preferable?”

Abu rose to the occasion. “As an aspiring programmer — of Python at least — I’ve heard of the programming principle called DRY,¹ so I can imagine that programmers would cringe at having to expand a bunch of calls with lots of different numbers (see Figure 3.3) the way Figure 3.2 does.”

¹ Don’t Repeat Yourself. AKA the SOAR principle — Stamp Out and Abolish Redundancy.

Figure 3.3: Multiple calls with different numbers

```
print(list_some_computations_on(1, 2, 3, 4))
print(list_some_computations_on(5, 6, 7, 8))
...
print(list_some_computations_on(97, 98, 99, 100))
```

Seeing Abu’s figure, Ila nodded vigorously. “As a working programmer, I can tell you you are absolutely right, Abu!”

Til smiled and went on. “Quite so! So functions encapsulate modules (pieces of code). In fact, encapsulation means modularization. These ‘modules’ we define as functions become part of our vocabulary, and a new module is like a new word we can use instead of saying what is meant by that word, in other words, its definition, every time we want to use it.”

Suddenly serious, Til said, “But stripped down to its core — a **function** is just an object that takes objects and gives other objects.”

Ila said “You said a while back that we would mostly avoid using the object-oriented features of Python. So I take it you don’t mean that kind of object here?”

Til nodded, “Correct. Not an instance of a class — that’s object-oriented programming-speak, Abu — but object in the completely general sense of *thing*. When I said ‘stripped down to its core’ I meant its *mathematical* core — independent of any realization in a computer language. So again, a function is simply an object, a machine if you will, that produces outputs for inputs. Now, mathematicians and computer scientists describe various kinds and attributes of functions and their machinery with a plethora

of terms that have been adopted over the years — so let's start with the one term that justifies the way we began by learning about sets.”

3.2 A Plethora of Terms

Abu was sure he would enjoy this learning challenge. Ila was a little worried, as she had never been good at memorizing a lot of vocabulary words. They both were surprised when the first term Til defined was very familiar, and then were not so surprised that they had never seen nor heard the second term before.

“Functions need sets for their complete and rigorous definitions,” Til said. “The set of all *possible inputs* for a function is called its **domain**. Distinguish that from the set of all *possible outputs* for a function, which is called its **codomain**. So the domain of our sample function, which does the four basic arithmetic operations on certain pairs of its inputs, is the set of numbers, and its codomain is the same set.”

“But which set of numbers?” Ila scratched her head, and continued, “there are lots of different sets of numbers, as we've seen.”

“Yes, there are!” said Til. “So we need to specify exactly *which* set of numbers we mean. For starters, let's just say that the set of positive integers is both domain and codomain for this function. Now note that this doesn't work. Why not? Because the third operation d / b produces 2.0, a number with a fractional part, even though it's only a decimal point followed by nothing but a zero. Still a *floating point* number, as it's called.”²

Til looked closely at his twotees to make sure they were paying attention. “So — this is important — in Python — and in other languages, for that matter — there's a difference between *mathematical division* (the single slash we have) and *integer division*, which is represented by a double slash. We want the double slash in order to stay in \mathbb{Z}^+ , which, recall, is the symbol for the set of positive integers. For a different reason, \mathbb{Z}^+ also won't work as the codomain. Suppose a and d are the same number. The second operation does d minus a . Subtract the same number from itself and you get zero — which is outside the set of positive integers.”

“So why not make the set of all integers the domain and the codomain?” Abu said. “I see a problem with that,” said Ila. “The second input, b , can't be zero, because you're dividing by it.” “Very astute!” said Til, obviously pleased with their thinking. “How do we fix our domain, then?”

“How about this,” said Abu, as he showed them his display. “The domain can be $\mathbb{Z} \setminus \{0\}$, and the codomain can be \mathbb{Z} .” Til said, “Ah, excellent! But let's be even more nuanced in our thinking. How many inputs does our function take?”

“I wondered about that,” said Abu. “But first, tell me — is there a way to tell Python what set is the domain and codomain of a function?” “No,” said Til, “but there are ways to work around that, which we'll discuss

² In computers, these numbers, which are also called *floats*, are *approximations of real numbers*. More in the purview of continuous math than of discrete math, real numbers are not of particular concern at this point.

³Not to be confused with “an exchange of diverging or opposite views, typically a heated or angry one;” or “reasons given with the aim of persuading others that an action or idea is right or wrong.” The argument “vector” (`argv`) is how function inputs are sometimes identified in C-like languages.

⁴The argument “count” (`argc`) is just the length of the argument vector. In some functions in C-like languages `argc` is necessary because `argv` has no ‘length’ attribute that can be retrieved.

Figure 3.4: A function to do some computations on four numbers and return them. Calling it with those numbers will print the output `(3, 2, 3, 12)` with parentheses instead of square brackets.

later. But, to overload an everyday word, we call the inputs to a function its **arguments**.³ And a not-so-everyday word — **arity** — means the *number* of input arguments a function has.”⁴

“I’m not really seeing a nuance here,” said Ila. “Our sample function has four inputs — arguments — so its arity is four, right?” “Yes,” said Til, “but what would you say is the number of outputs it has?” “Four as well?” said Abu. “Or is it one??” Ila, what do you think, is it one or four?” said Til. “Well, it’s four numbers bundled in one list,” said Ila. “Right!” said Til. “Here’s the thing. We can redefine our function (with // this time) to return four outputs *without* wrapping them in a list (see Figure 3.4).”

```
def do_some_computations_on(a, b, c, d):
    """Add, integer divide, subtract, multiply
       four numbers pairwise in a certain
       fashion.
    """
    return a + b, d // b, d - a, c * d

print(do_some_computations_on(1, 2, 3, 4))
```

“So, to be precise,” continued Til, “the original version’s codomain is really the set of *lists* of four integers. Math stipulates that a function has only one output, so we have to use a list or some other container to return multiple things as one thing. But in Python it looks like it *can* return multiple things — and that’s one nuance I want you to be aware of.”

“I wonder,” said Abu, “how big a problem it is to say the codomain is the set of lists of four integers instead of just the set of integers?” Ila jumped in, “I think it’s because we haven’t yet defined what a list is, exactly. Right?”

“Right,” said Til. “We can, and we will, mathematically define a list, but let’s postpone that for now. The other nuance is that we don’t really have a single *domain* for our function. Because . . . ?”

Abu saw it. “Because it has four inputs, not one! But its domain is not the set of lists of four numbers either, is it?” Ila said, “This is starting to get way too complicated, if you ask me!”

“Agreed,” said Til. “But the problem really lies with the function itself — it tries to do too much! This 4-ary function would perhaps be better if we broke it into four 2-ary functions. Then only the domain of the second of two arguments to the function that divides the first argument by the second argument would need to exclude zero. Speaking of 2-ary, we need more vocabulary — **k-ary** describes a function with **k** inputs. We also say **unary** for a function with 1 argument, **binary** for a function with 2

arguments, **ternary** for 3, etc. Actually, for four or more, the k-ary way works better.”

Abu thought of something. “Can there be a function that takes *zero* arguments — a 0-ary function?” “Absolutely,” said Til. “You can also call it a **nullary** function.” Ila thought “Seems wasteful,” then said, “How would a nullary function be any different than a defined constant?” “The result would be the same, yes —” Til said, “assuming when the function is called it doesn’t do anything other than return a constant — no random behavior, no side effects. But we’ll come back to this idea when we talk about higher-order functions. There are a dozen or so terms that we still need to define, but we’re done for now.” Til ended their session as abruptly as he began it.

Exercise 107 What is the codomain of the function whose domain is \mathbb{Z}^+ and whose output is the square root of its input?

Hint 107 Think about the square root of 2, for example.

square roots of integers that are irrational numbers.

ANSWER The codomain would have to be \mathbb{R}_+ to accommodate

Exercise 108 Look up the definitions of the following Python builtin functions and describe their domains, codomains, and arities:

1. *abs*
2. *chr*
3. *int*
4. *len*
5. *pow*

Hint 108 These are not just sets of numbers.

AT A GLANCE **ANSWER**

Exercise 109 The notation $f : D \rightarrow C$ is the math “signature” for a function f that takes its domain D to its codomain C . The sets D and C can be the same set or different sets. Which of the five Python builtin functions in Exercise 108 best matches the function signature $f : \mathbb{Z} \rightarrow \mathbb{N}$?

a number-type set.

Hint 109 Eliminate any functions whose domain or codomain is not

absolute values are natural numbers.

ANSWER The address function is the one that matches best. Interfaces,

Exercise 110 Wiktionary defines codomain as:

The target set into which a function is formally defined to map elements of its domain; the set denoted Y in the notation $f : X \rightarrow Y$.

Wolfram MathWorld defines codomain as:

A set within which the values of a function lie (as opposed to the range, which is the set of values that the function actually takes).

The (two-part) question is: Why bother with the term codomain at all? Why not just use range?

right answer.

Hint 110 The question is asking for an opinion — there's no one

AT&T USA OLI Test

Exercise 111 More recent books don't use the term "range" at all, instead preferring which other word?

Hint 111 A little research will reveal it.

of a function's properties **imposes** as defined in § 3.3.

ANSWER III The term "matrix" is used as the name of the set of all

3.3 Image and Preimage, Pre-In-Post Fix

“Did you understand Til’s message about the image and preimage of a function?” Ila said, after they settled in and exchanged pleasantries. She was glad Abu was not averse to small talk like Til appeared to be. She didn’t know how soon he would come into his study where they were waiting, but wanted him to find them discussing what they were learning when he did.

“I think so,” said Abu. “Tell me if you think I have this right (refer to Figure 3.5) — both how I defined this unary square function and how I interpret those terms. So, as I understand it, the image of 2 under the square function is 4. The square function’s preimage of 4 is 2. The image of 3 is 9, the preimage of 25 is 5, and so on.”

“That’s right!” said Ila. “Til said that **image** refers to the output of a function applied to some input, and the input is called the **preimage** of the output — so you nailed it.”

“Indeed I did, and indeed you did, Abu” said Til, as he joined them. “Ila, is that your alternate definition of square we’re seeing in Figure 3.6?”

“Yes,” said Ila. “I thought it would suggest squaring better — exponentiating by 2 — plus I like that Python’s exponentiation operator is the same as JavaScript’s!” Abu said, “Well, I like them both, because they use the familiar — how did you describe it in your message, Til? — here it is, **infix notation** is when you have a *binary* function whose symbol comes *between* its two arguments.”

Til nodded, but said nothing, so Ila jumped in. “Do you have a preference, Til? I’m with Abu, I like infix too.” Til said, “I do have a preference. Let’s compare and contrast the three ways, and then I’ll tell you what it is. So, infix means the symbol, or name, comes **in-between**, **prefix notation** means the symbol comes *before* the function’s arguments, and **postfix notation** means the symbol comes *after* the arguments. Prefix and postfix accomodate arbitrary **k-ary** functions, not just binary ones. Infix is by far the most common, and postfix the least common. If you remember, you’ve already seen an example of prefix when we were talking about the logical not operator, `¬`.”

Abu said, “I remember you said that when we would talk about infix versus prefix versus postfix, and functions and their arities, we would see what you meant when Ila asked you whether or not operator and connective mean the same thing, and you said they do and they don’t. Please, explain!”

Ila agreed. “Yes, I would love to hear your explanation — I’ve been waiting.”

Til looked pleased. “I’m glad you remembered that point. And I will explain after you answer one question — when you call a function, is that prefix, infix or postfix?”

```
def square(number):
    """Squaring is multiplying
    a number by itself.
    """
    return number * number
```

Figure 3.5: A square function

```
def square(number):
    """Squaring is raising a
    number to the power of 2.
    """
    return number ** 2
```

Figure 3.6: An alternate square function

Ila looked thoughtful, Abu puzzled. “I think all function calls are prefix,” said Ila, “because the function name always comes first.” “In your experience that is the case,” said Til, “but could you also have listed all the arguments, still surrounded by a pair of parentheses, and then the function name last?” “I suppose so,” said Ila, “but that seems kind of weird to call a function by mentioning it last.” Abu chimed in, “Maybe the point you’re getting at is that it’s infix that’s weird, because you’re restricted to just two arguments, and you put the name between them — although, that’s how we say it — two *plus* two not *plus* two two or two two *plus*.” Til was very pleased. “Notice what you just did, Abu? By contrasting the three ‘fix’ notations you implicitly turned the *plus operator* into a *function!*”

Abu looked thoughtful, and it was Ila who looked puzzled now. Then she suddenly grinned. “I see what you mean! You answered my question by asking another question!” Abu said, “What do you mean, Ila?” She paused a moment, still grinning, then said, “It all makes sense now. Operators *are* functions! Connectives *are* functions! They may have different arities, and whatever ‘fix’ notation you please, but they still take inputs and produce outputs.”

“Very good you two!” beamed Til. “But don’t forget the nuances — the reasons why we use different words. So, I’ll tell you my preference now. It’s prefix — because I believe in saying what you’re going to do and then saying what you’re going to do it to. Add two and two. Two plus two — or two times two, or two raised to two, it’s all the same — it interrupts the flow!”

3.4 Home Onto the Range

“Speaking of nuances,” said Abu, “I remember you wanted us to bring up and discuss the difference between a function’s *possible* outputs and its *actual* outputs.” “Thank you for reminding me, Abu” said Til. “Ila, what is the name of the set of *possible* outputs?” “Codomain,” said Ila. “But I don’t see how actual and possible are different.” “Let’s give a name to the actual outputs set,” said Til, “and then see how it’s *potentially* (not necessarily) different. So the set of all *actual* outputs is called the **range** of the function. *If* it is different than the codomain, would the range necessarily be larger or smaller than the codomain, Abu?” Abu thought for a moment, then said, “Smaller, I think. You can’t actually have more than what’s possible — that would be impossible!” “Ha, ha,” said Ila. “Can you give us an example of a function whose range is smaller than its codomain?”

“Your very own square function, Ila!” Til smiled and continued, “but first hold on — not just smaller in size — the range must be a *subset* of the codomain.” Abu said, “That’s because actual has to be possible!”

“Right,” said Til. “So, which sets are domain and codomain for the square function, Abu?” “I’d say \mathbb{Z} for the domain and \mathbb{N} for the codomain. Because squaring a negative number makes it positive, and the square of zero is zero.” Ila addressed Abu. “But not every natural number is a square of an integer, so I think the codomain should be the set $\{0, 1, 4, 9, 16, 25, \dots\}$. Or should that be the range?” Abu said, “It’s certainly smaller — a subset — of the naturals. What’s the verdict, Til?” “Keep in mind,” Til said, ignoring the question, “that in general, squaring applies to all real numbers, so the domain is really \mathbb{R} , and the codomain is $\mathbb{R}^+ \cup \{0\}$ ⁵ — to underscore your point about squares being nonnegative, Abu.” Abu looked pleased as Ila said, “That didn’t really answer my question, but I suspect you were about to answer it with another question, judging from that look in your eye!” Til laughed and said, “You’re starting to read my mind, Ila! I’ll give you the answer prefaced by another vocabulary word and then my question — an **onto** function is one whose range is the same as its codomain (that is, an **onto** function produces (for the right inputs) all possible outputs). Would the square function be considered onto?” “It depends,” said Ila, “on which version of square you mean. The square with signature $f_1 : \mathbb{R} \rightarrow \mathbb{R}^+ \cup \{0\}$ or the square with signature $f_2 : \mathbb{Z} \rightarrow \mathbb{N}$?” “The latter, f_2 ” said Til. “Then no,” said Ila, “because $\mathbb{N} \neq \{0, 1, 4, 9, 16, 25, \dots\}$.” Abu waited for Til to say something, but Til just looked at him as if he expected him to speak next. So he did. “That means f_1 would be onto, because it produces all possible outputs for its inputs.”

Til smiled, and said, “I’ll leave you two sharpwits to find other examples and explore the corner cases, if you find any. But one more thing before you go — we say a function takes its domain **to** its codomain, which could mean **onto**, if the range is the codomain. But what if it isn’t? Instead of saying not onto, what’s another prefix similar to ‘on’ that we can prepend to ‘to’ to describe the not onto case?” “How about ‘in’ — so **into**?” ventured Abu. “Bingo!” said Til, “and bye for now.”

3.5 Mix and Match

“Help!” Ila implored Abu. “I’m drowning in terminology! Til wasn’t kidding when he said there was a plethora of terms to learn.” “I know,” said Abu, “I totally get how difficult it is to keep them all straight. Take this list he sent us:”

- **surjective** is a synonym for onto.
- **surjection** is a noun naming a surjective function.
- **one-to-one** describes a function each of whose outputs is generated by only one input.
- **injective** is a synonym for one-to-one.
- **injection** is a noun naming an injective function.
- **bijective** describes a function that is both one-to-one and onto.

⁵ Or, it could be argued that \mathbb{R} is both domain and codomain, to make a virtue of the necessity of having a different range than codomain. It would seem that mathematicians delight in the nuance that the range — $\mathbb{R}^+ \cup \{0\}$ in this case — is necessarily different than the codomain.

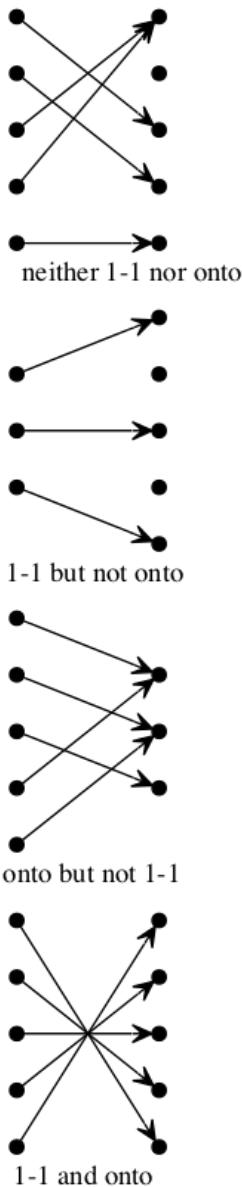


Figure 3.7: Simple abstract functions from/to sets with three or five elements. These show the conditions for functions being onto versus one-to-one. A necessary condition for a function to be onto is that the size of its codomain is no greater than the size of its domain. In symbols, given $f : A \rightarrow B$, f is onto $\rightarrow |A| \geq |B|$. A necessary (but not sufficient) condition for a function to be bijective is for its domain and codomain to be the same size.

- **bijection** is a noun naming a bijective function.
- **recursive** describes a function that calls itself.

“He said all but the last term can be understood by carefully studying this picture (Figure 3.7):

“I get one-to-one versus onto,” said Ila. “The difference is pretty clear in the abstract, but all those blank-jection terms — good grief!” “I thought of a way to kinda know which is which,” said Abu. “Take surjection. Sur as in surface, like on the surface, so surjection is onto. Take injection. Typically when you get a shot, one needle goes into one arm, so injection and one-to-one match.” Ila smiled and said, “Thanks, Abu — that helps. And bi means two — both — so a bijection is both an injection and a surjection.” “Yep — good!” said Abu. “So are you good to tackle this hairy-looking exercise now?” “Maybe,” said Ila. “But I’m hoping you have some good intuition about this one, because it scares me silly just looking at it!”

Exercise 112 A function is one-to-one if and only if it is onto, provided its domain and codomain are the same size.

Put this statement in symbolic form using the adjectives injective and surjective, and argue for its truth using the following formal definitions:

Let f be a function from domain A to codomain B , $f : A \rightarrow B$.

f is injective if $f(m) \neq f(n)$ whenever $m \neq n$, for all $m \in A$ and for all $n \in A$.

f is surjective if for all $b \in B$ there exists an $a \in A$ such that $f(a) = b$.

Hint 112 The statement of the problem in symbolic form is: $|A| = |B| \rightarrow (f \text{ is injective} \leftrightarrow f \text{ is surjective})$. As a conditional, this is true unless the antecedent is true and the consequent is false. So you must argue that given the equality $|A| = |B|$, it must be true that the function f is injective if it is surjective, and it is surjective if it is injective. Reason so as to reject an absurd conclusion if it would contradict the premise that sets A and B are the same size.

AT aasa SII tewsaA

“Well, that wasn’t too bad, now was it!” exclaimed Abu. “Ila, stop rolling your eyes, you loved that little logical-thinking exercise!” Ila rolled her eyes again. Abu sighed and said, “Maybe you could help me understand recursive functions better? I know we talked about it a little with those recursive rules for building compound propositions, but what’s a good example in Python?” Ila was pleased. She searched her files for a few seconds, then showed her display and said, “I don’t know how good

it is, but I like this simple recursive multiplication of a list of numbers (see Figure 3.8). The ‘recursion’ — that’s what a recursive function does — or rather, let’s say it *recurses* by multiplying the first number in the list by the result of calling itself with the *rest* of the list.” Abu furrowed his brows. “Is that what $L[1:]$ means — the rest of the list from element 1 on — element 0 being the first element?” “Yes,” said Ila, “that’s called list slicing in Python. I rather like that syntax!” “I guess it’s okay. But what about the ‘if not L ’ — what does that mean?” Ila was enjoying this teacher-learner interaction. “That’s the *escape* clause. If you keep calling yourself with a list that shrinks by one every time, eventually you run out of list. What do you do then?” “I think,” said Abu, “that you’d better stop calling yourself!” “Right!” said Ila. “You have to escape from the recursion by detecting the empty list — we’ll ask Til if there’s another way — but as far as I understand it, ‘if not L ’ essentially does that emptiness check. And that’s when you return 1 because multiplying by 1 doesn’t change the result.” “That’s amazing!” said Abu. “That’s recursion!” said Ila. “One more question,” said Abu, “then I have to run. What is the backslash ‘\’ doing on the second ‘return’ line?” “Oh,” said Ila, “that’s the ‘line continuation’ character. I put it there to make the line width shorter, but read it just as if it were on one line.” “Thanks,” said Abu, “that makes sense. Well, see you next time!”

3.6 Abstractions and Types

“Today we’re going to get a little more abstract — so hold on tight!” said Til. At that, Ila and Abu subconsciously but synchronously tightened their grip on their chair arms. If Til noticed, he gave no sign. He held their gaze for just a moment, then said “Similar to what you did for an exercise, the function in Figure 3.9 is a one-liner whose domain, codomain and range are the same set: $\{0, 1\}$. This is problematic for a couple of reasons.”

Abu and Ila relaxed their grip as it was starting to be uncomfortable, and hindered their concentration to boot.

“For one,” said Til, “Python is a so-called *dynamic-typed* language where variables don’t have types,⁶ only values do. Remember, Abu, you asked if there was a way to tell Python what sets a function’s domain and codomain are, and I said no, but that there were ways to work around that that we’d discuss later?” “Yes, I remember,” said Abu. “Well, later is now!” said Til.

Ila’s programmer’s spidey sense was tingling, as she was all too familiar with type errors.

“The second reason this is problematic,” continued Til, “is that somehow restricting callers of this function to only pass 0 or 1 is a *non-trivial* task — an onerous task. The language itself doesn’t help, very much any-

```
def multiply(L):
    if not L:
        return 1
    else:
        return L[0] * \
            multiply(L[1:])
```

Figure 3.8: Recursive multiplication. The call `multiply([2, 3, 10, 29])` proceeds
`2 * multiply([3, 10, 29])` whence
`2 * 3 * multiply([10, 29])` whence
`2 * 3 * 10 * multiply([29])` whence
`2 * 3 * 10 * 29 * multiply([])` whence
`2 * 3 * 10 * 29 * 1` whence
`2 * 3 * 10 * 29` whence
`2 * 3 * 290` whence
`2 * 870` whence
`1740`, the finally returned value.

```
def not_with_0_or_1(arg_is_0_or_1):
    """Logical not with 0 for
       false and 1 for true.
       Subtracts argument from
       1, so 1 - 0 yields 1,
       and 1 - 1 yields 0.
    """
    return 1 - arg_is_0_or_1
```

Figure 3.9: Not with 0 or 1. Domain, codomain, and range are the set $\{0, 1\}$. Or at least, that’s what they *should* be!

⁶ You almost certainly know that types are programming abstractions that include Boolean, integer, float, string, etc. In dynamic (as opposed to static) typing, not only do variables *not* have declared types, they can store *any* type during their lifetime.

way. As it stands, calling this function with any numeric type for which the subtraction operation is legal will not be detected as an error, but it *might* be an error nonetheless. A logical error.”

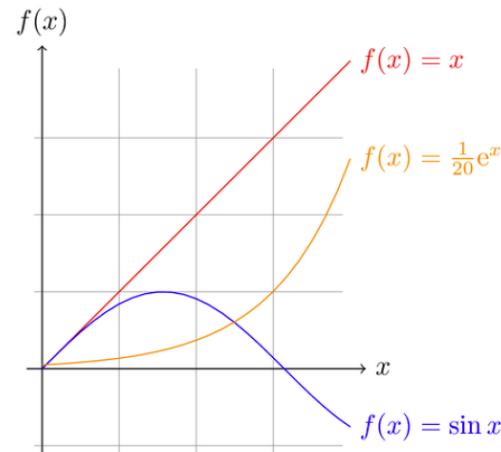
Abu was hanging on to every word, understanding only about half of them. Ila said, “But the function itself *can* help by testing its argument and balking if given something besides 0 or 1, right?” “Yes,” said Til, “but balking can mean many things — throwing an exception being the most extreme, other than just aborting execution altogether. It thus becomes the programmer’s responsibility to do type-checks or deal with raised exceptions.”⁷

⁷ Static-typed languages, on the other hand, empower *the compiler* to enforce strict typing.

3.7 Representations

“We’ve been looking at functions a certain way,” said Til, “mostly with a Python programming bias — but functions can be viewed in many other ways besides code.⁸ Through your study of these ways, I hope you’ll appreciate the bottom line — computer scientific functions are useful abstractions of mathematical functions, which in turn are useful abstractions of the *process* of transformation. And the fact that functions have multiple ways of viewing them is a consequence of how useful each way has proven to be. You’ve seen my list of a few of these representations. You can also see a few functions graphed in Figure 3.10 and one of them partially tabulated in Table 3.1.”

Figure 3.10: A few functions graphed.



x	.05e ^x
0	0.050
1	0.136
2	0.369
3	1.004
4	2.730
5	7.421
6	20.17
7	54.83
8	149.0
9	405.2

Table 3.1: A short table of 10 integral values of x mapping to $\frac{1}{20}e^x$.

Ila felt her throat constrict — as it did whenever she was faced with an intellectual challenge — like dealing with a long list of different terms that *mean the same thing*.

Til paused as if he sensed Ila’s discomfort, although she kept her expression the same ‘I’m interested in learning this’ semi-smile. He went on. “I hope, eventually, that you’ll be able to move comfortably between all these different representations, not just of functions, but of many other

areas of math as well. When you can do this you'll have what we call *representational fluency* — to be fancy about it.”

Ila couldn't help herself. She said, “I don't know if I'll ever be comfortable or have, whatever you said, fluency —” “Representational,” said Til. “I just —” started Ila, — okay, I'll just come out and say it. I hate that there are so many different ways to say the same thing!”

Abu piped up, “I know what you mean, Ila. But I can see that having more than one representation can be a good thing. Remember that Richard Feynman quote I shared with you? I believe just like Feynman's knowledge of chemistry and physics enhanced his understanding and appreciation of flowers, so likewise our minds can be richer, not poorer, by having more ways of seeing.”

Ila replied, “Maybe you're right — but I'm not convinced. Til, do you really believe that the more representations we have, the clearer our understanding becomes?”

Til said, “Yes, but we must be careful. Each representation has its own precision — and its own vagueness.”

“What do you mean?” said Abu.

“Let me put it this way,” said Til. “We use words, symbols, pictures, etc., to understand our world and reach mutual understanding with other people. But each representation of an idea we seek to illuminate is just a facet of the whole, and subject to misinterpretation if that's all we focus on. We need to be both holistic *and* reductionistic — not one at the exclusion of the other.”

“Could you give us an example of what you mean?” said Ila.

“Sure,” said Til. “Look at how we write $1/2$ versus $\frac{1}{2}$. The first suggests one divided by two, the second or third the fraction one-half — what we *get* when we divide one by two. The result *and* the process. Both are important.”

“Okay,” said Ila, “but what does that have to do with representational fluency?”

“Representational space deals with relations between ideas, concepts, aspects of reality. Representational fluency speaks to how nimbly we deal with aspects of reality like time and space, and our movement through each. Causality, cause and effect — three-dimensional spatial relationships, above, below, right, left, before or behind — proximity and locality — are you still with me?” Til noticed both Ila and Abu starting to drift.

“Speaking of space,” said Abu, jerking himself back into focus, “the representation of functions as graphs in 2-d space is one I've thought was most helpful — ever since I took algebra.”

“How about you, Ila?” said Til. “Yes, I agree with Abu,” said Ila, “visualizing functions as graphs is very helpful. But that reminds me, this everyday, ordinary meaning of graphs is not the only meaning of the word — I've also seen it referring to those webby-looking things with dots and

lines. Just another example of confusing terminology, this time using the *same* word to refer to two *different* things.”

“Right you are,” said Til, ignoring her complaint, “and we’ll get to those types of graphs, important discrete structures that they are, later.” Til could see that Ila had a complainer imp that often bent her to its will, but he was used to that.

“For now, let’s explore just one representation of functions — actually two-in-one — the *association* and the *mapping* (or *map*⁹). A function-as-association between elements of a domain and elements of a codomain, for ‘small’ domains and codomains, inspired this so-called *associative array*, yet another name for the same concept.”

Even Abu, normally not one to get anxious about learning new terminology, was being flooded with information overload warnings.

“In Python this is known as a *dictionary* — I suspect you’ve seen some of these if you’ve looked at a lot of Python code,” said Til.

Abu thought he had heard wrong — he knew what a dictionary was, of course, but could only say, “Isn’t that what you use to look up definitions of words?”

Til smiled. “Yes, of course, the ‘word → definition’ mapping provided by an ordinary dictionary generalizes in this conceptualization to any ‘key → value’ mapping. Figure 3.11 shows a simple example.”

“You will explore these other representations (mostly code) through examples and by doing exercises — but also by applying the gersy principle.”

“The *what* principle??” said Ila and Abu, almost in unison. “Remember when I gave you a 2x5 word principle and said I’d give it a name and we’d discuss it more later?” said Til, fixing his gaze on Ila. “It’s later, isn’t it!” said Abu. At that Til nodded as he turned to look at Abu, and said, “And what were those 2x5 words, Abu?” Abu grinned and said,

“Never settle for the superficial! — Always look below the surface!”

“That’s twice five words, yes, but what was that name you said a minute ago?” asked Ila. Til replied, “Gersy — ‘guh’ (hard ‘g’) ‘eee’-‘are’-‘ess’-‘why’! Not to be confused with the cattle breed, Jersy.” “For the life of me,” said Ila, “I can’t see how gersy goes with that principle — which, by the way, I’ve been trying to practice.” “As have I,” said Abu.

“The connection comes,” said Til, smiling, “from my observation that the two sentences

1. John is **eager** to please.
2. John is **easy** to please.

are very similar — at least *on the surface*. Superficially, they differ only in the five letters ‘g’, ‘e’, ‘r’, ‘s’ and ‘y’ — hence, ‘gersy’. You have to look below the surface to discern that they have very different meanings. Sentence 1 is more a statement about John’s approach to life. Sentence 2

⁹This is what it’s called in C++. Other languages call it a *hashtable* or a *hashmap*.

```
mymap = {}
mymap['one'] = 'red'
mymap['two'] = 'blue'
mymap['three'] = 'green'

for key in mymap.keys():
    print(key + ': ' + mymap[key])
# the same dictionary in
# literal syntax
for key in {'one': 'red', \
            'two': 'blue', \
            'three': 'green'}:
    print(mymap[key] + ': ' + key)
```

Figure 3.11: A mapping from number names to color names. Prints out:

```
one: red
two: blue
three: green
red: one
blue: two
green: three
```

is more about his personality.”

Abu wondered where Til got those two sentences as he processed this revelation. He would ask Ila later what she thought.¹⁰ For him it was a big reveal into the way this man’s mind worked.

Ila was skeptical. “Is this play on words supposed to make it easier to remember the principle?” “Only if you make it so,” said Til. “In general, if you think about it, names are pretty arbitrary handles for the things they name — ‘cow’, for example. How did the word ‘cow’ come to refer to that four-legged mammal we humans have found so useful?” “I guess,” said Abu, “that that applies to any word we learn and store in our mental dictionary!”

3.8 What’s In A Name?

“Speaking of names we give things,” said Til, pleased to have finally shared his favorite learning principle’s name’s origin, “let’s take a look at one of the most prominent functional-programming-language features of Python, or indeed any language claiming to be functional. Its name is **lambda** — and its definition is an *anonymous, nameless* function. For example, compare the *named* double function with its nameless lambda equivalent in Figure 3.12.”

“For another example, say we want a function whose definition is *if the argument x is even return x halved, otherwise return one more than thrice x*. What name would you give that function? Forget about that creative burden — if this function is going to be called just once, you need merely surround its lambda expression with parentheses and follow it with an argument, also in parentheses, as in Figure 3.13.”

“Still with a single mention of the lambda expression, we can do many calls to our function if we want to just by using a simple loop (see Figure 3.14).”

Abu was intrigued, but before he could formulate a question

“That’s not the only way to do a loop, you know,” said Ila, showing her quickly typed alternative (see Figure 3.15).

“That’s right — there are many ways to do it. But here’s the deal,” said Til with eyes aglow. “You know the old saying — just because you can doesn’t mean you should? I urge you to start seeing loops that way. Eliminating loops in your code can be abundantly advantageous. Look for reasons why as we continue our journey! Until next time — be wise, and remember to apply the gersy principle!”

¹⁰They later discovered a web site that was just *a tip of this iceberg*. Will you go deeper?!

```
def double(n):
    return n * 2

lambda n: n * 2
```

Figure 3.12: Double and double again, the second time sans name. The advantages are compelling. No need to think up a name, as the body of the function — what it does — suffices. Also not required are the def and return keywords, as well as the parentheses surrounding the argument list.

```
(lambda x: x * 3 + 1 \
    if x % 2 else x // 2)(27)
```

Figure 3.13: An anonymous function called with 27. If done in a REPL, the answer will be 82. The lambda expression makes use of Python’s 1 = true, as x % 2 returns 1 if x is odd. If x is even, x / 2 will be an integer — but x // 2 is used to avoid getting a float answer.

```
n, m = 0, 7
while n <= m:
    print('{0} --> {1:2d}'.format\
        (n, (lambda x: x * 3 + 1\
              if x % 2 else x // 2)＼
        (n)))
    n += 1
```

Figure 3.14: Loopy lambda prints results:
1 --> 4
2 --> 1
3 --> 10
4 --> 2
5 --> 16
6 --> 3
7 --> 22

```
for n in [1, 2, 3, 4, 5, 6, 7]:
    print('{0} --> {1:2d}'.format\
        (n, \
        (lambda x: x * 3 + 1 \
            if x % 2 else x // 2)＼
        (n)))
```

Figure 3.15: A better loopy lambda

3.9 The Loop Free Way

"How in the world can we do without loops?" Ila was giving Abu an earful. "I must use them like 200 times a day!" Abu, amused, listened for a while, then when Ila paused for breath, he jumped in. "More to unlearn?!" Ila was taken aback, but before she could retort, a text from her husband arrived and she stopped to read it. "We'll talk later," she said, waving distractedly at Abu, who waved goodbye back with a wan smile.

Much later, when they arrived at Til's, he was excited to see them and actually welcomed them with a big smile before starting in. "Look at the simple loop construct in the `morphify` function in Figure 3.16. The function takes two arguments, `f` and `l`, another function and a list, and iterates calling the function `f` with each element of the list `l`, and collecting the results of these calls into a new list, which is returned."

Ila recalled her recursive multiplication function and quickly produced Figure 3.17, which, with a wink to Abu, she displayed and said, "You can do that same thing recursively!"

"Excellent!" said Til. "And you've just shown that loops are unnecessary — you can always use recursion instead!" Ila was ready for that 'bombshell'. She said, "But why should we prefer recursion to looping? There's a lot more function calling — that seems wasteful."

"True," said Til. "I would instead urge you to prefer *mapping* to looping. And I don't mean the map-as-dictionary mapping representation of a function. I mean the concept of *higher-order* functions, that I mentioned earlier, if you recall. These are functions that *take functions as arguments* — like our `morphify` and `morphifyr` functions do — or that *return functions* that can then be stored, or passed to other higher-order functions." Abu was desperate for something, anything concrete here, and Til obliged. "The Python `map` function is very much like your `morphifyr`, except that it has no need for an external *accumulator* (see Figure 3.18)."

"Isn't a loop still there just under the covers when you use a mapping function like that? Or at least, recursion?" said Ila. Abu was sure he would need some more time to digest all this, but he was sure Til was sure he would seek Ila's help.

"It may seem like mapping is implicit looping, or recursion, which is like looping," said Til. "But that is your Von-Neumann-architecture-infused mindset at work, where computation proceeds serially as a sequence of executed instructions. Mapping need not be so conceived! Imagine a computer with many processors capable of executing the same code on different data items, one item preloaded into each processor. Then the mapping over the complete data set can happen in parallel, that is, the function *simultaneously* transforms each item, rather than sequentially one at a time.¹¹ This is *powerful magic!*"

Ila, surprised at how passionately Til was speaking, thought she would

```
def morphify(f, l):
    results = []
    for e in l:
        results.append(f(e))
    return results

print(morphify(lambda x: x ** 2,
              [1, 2, 3]))
```

Figure 3.16: The `morphify` function and a call that prints the list `[1, 4, 9]`.

```
def morphifyr(f, l, a):
    if l:
        a.append(f(l[0]))
        morphifyr(f, l[1:], a)

    accum = []
    morphifyr(lambda x: x ** 2,
              [1, 2, 3], accum)
    print(accum)
```

Figure 3.17: The `morphifyr` function (recursive version) and a call. Populates the passed-in list `accum`, which printed yields the same list as before, `[1, 4, 9]`.

```
first_3_squares_map = \
map(lambda x: x ** 2,
    [1, 2, 3])
print(first_3_squares_map)
print(list(first_3_squares_map))
```

Figure 3.18: The `map` function and storing the call to it in a variable which is printed. To print the list as before, `[1, 4, 9]`, the `map` object must be converted to a list.

¹¹ This scheme is known as *Single Instruction, Multiple Data*, or SIMD.

have to really take stock of her entire experience with programming. She realized, though, that it would take some time before she could let go of her imaginary servant who did all the housekeeping chores of loop initialization, loop execution, and loop update, while she just sat back and reaped the benefits.

Abu was thinking he was going to be seeking Ila's help *a lot* in the near future. But the prospect of learning this powerful magic was compelling!

"By the way," said Til. "You don't have to use lambda expressions with `map`, you can use named functions as well. For example, take the `personality3` function that you may recall from Exercise 106, and see what concise expressiveness `map` and other functional-programming constructs can afford it, regardless of the choice of named or unnamed functions. But you decide which you prefer, the way of Figure 3.19 with lambdas, or the way of Figure 3.20 without."

```
from operator import concat
from functools import reduce

def personality3(boolfunc3):
    inputs = [(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
    outputs = map(boolfunc3, map(lambda x:x[0], inputs), map(lambda x:x[1], inputs), map(lambda x:x[2], inputs))
    return reduce(concat, map(str, outputs))
```

Figure 3.19: The `personality3` function with lambdas.

```
from operator import concat
from functools import reduce

def first(x):
    return x[0]

def second(x):
    return x[1]

def third(x):
    return x[2]

def personality3(boolfunc3):
    inputs = [(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
    outputs = map(boolfunc3, map(first, inputs), map(second, inputs), map(third, inputs))
    return reduce(concat, map(str, outputs))
```

Figure 3.20: The `personality3` function without lambdas.

Abu and Ila gazed back and forth at the two figures for several seconds. “So, that line with four `map` calls in it,” began Abu, hesitantly, “it looks like the outputs assigned there are being passed to the *fifth* `map` in the last line — that’s a lot of `maps`, by the way — could you walk us through those, please?” He was desperately trying to wrap his head around this new way of thinking, despite not having Ila’s loopy mindset to reset.

Ila was feeling more confident about her expanding understanding of it, and was sure it would solidify if she attempted to explain it to Abu. “May I?” she said, and Til replied “Yes, absolutely!”

“I think it’s working, Abu,” said Ila, typing furiously as she spoke, “as if `boolfunc3` were being called 8 times, like — here, look at my display (see Figure 3.21) — and keep in mind that `boolfunc3` is referring to another function, for example `f1` that I copied into this code.”

```
def f1(p, q, r):
    return or2(and2(p, q), not1(r))

boolfunc3 = f1

# the three lists are the 1st, 2nd, and 3rd elements of each 3-tuple
map(boolfunc3, [0, 0, 0, 0, 1, 1, 1, 1], \
     [0, 0, 1, 1, 0, 0, 1, 1], \
     [0, 1, 0, 1, 0, 1, 0, 1])
# which then becomes the equivalent of
[boolfunc3(0, 0, 0), boolfunc3(0, 0, 1), boolfunc3(0, 1, 0), boolfunc3(0, 1, 1),
 boolfunc3(1, 0, 0), boolfunc3(1, 0, 1), boolfunc3(1, 1, 0), boolfunc3(1, 1, 1)]
```

Figure 3.21: The `f1` function (alias `boolfunc3`) function called 8 times via `map`.

Abu studied Ila’s code for a few minutes, then the lightbulb lit up. “I think I see what’s going on! At first I thought `map` could only take a single list as its second argument, but now I believe it can take however many it needs — dictated by the arity of the function it’s mapping, I guess — so, yes, `boolfunc3` refers to a *ternary* function, which `f1` is.” “That’s right!” exclaimed Ila. “I hadn’t thought about it in terms of arity, but now I see that too!”

Til was abundantly pleased at the learning and teaching he was seeing unfold before his eyes, and resolved to capitalize on Abu’s and Ila’s swift blossoming into true collaborators.

Puzzled again, Abu said, “Til, what’s that `reduce` in the `return` line doing?” “That,” said Til, “is reducing a sequence of strings produced by the fifth and final `map` to a single string, the “concatenation” — the “smooshing together” — of all those other strings. This is thanks to the `concat` operator, or function — which, by the way, is a function that takes

two strings — in general, what reduce requires is a binary function that takes two of the same type of input. And here let me stress the idea that I said we'd come back to when we got to higher-order functions — how important it is that these functions being mapped over or called on to reduce something *do NOT* have any random behavior, or side effects.”

“I think,” said Ila, “that I know what those are — but mostly for Abu’s benefit” — she turned and grinned at Abu — “would you explain side effects, please?” Abu eagerly nodded yes.

“Side effects,” said Til, “mean something like print statements or changing some state that’s *not* local to the function, for example, the infamous **global variable**. Setting its value in various parts of your code and using its value in other, far distant, parts, is a fertile bed for bugs. These are effects happening “outside” or “on the side” of the purpose of the function. A central requirement of functional programming is that functions be free of side effects. So recall your nullary function — AKA defined constant. We want the value returned by that function — or any k-ary function, for that matter — to be able to replace that function call without changing any other behavior in our program.”

“But how do we know,” said Abu, “that concat, or str, or any other function we didn’t write, is side-effect-free?”

Standing up to signal they were done, Til said, “Figuring that out, my twotees, is for you a GPAO!”¹²

¹² Gersy Principle Application Opportunity, as Til revealed as he escorted them out. He also insisted they pronounce it with three equally-stressed syllables, G-PA-O (Gee-Paw-Oh, not Gee-Pea-Ay-Oh).

3.10 Further Adventures in Functionology

Abu and Ila were studying the list of GPAOs Til had given them. “There’s something I don’t quite get, Abu,” said Ila. “How can Til, our tutor, who it just cracks me up, by the way, that he calls us his ‘twotees’¹³ —” to which Abu just smiled. Ila ranted on. “How can he just throw so much at us to grapple with on our own while he sits back and takes it easy?” Abu scratched his chin, and said, “I know, it seems wrong. But when you were explaining that code to me, well, I saw the look in Til’s eyes. I saw pure joy — and it didn’t look like the joy of getting out of work!” “Hmm, I see,” said Ila, who didn’t really see — mostly it was the quantity of the material he had assigned them to learn, not the fact that he wanted them to tackle it on their own. But she thought Abu was a patient learner, and she was an eager teacher — when she understood something, which she had to work at. She was a little envious that Abu seemed to just sponge up knowledge — when it was presented in the right way. She resolved to help them both find that right way!

3.10.1 Floor and Ceiling

Til’s GPAO list began with what he called two highly useful functions with the real numbers as domain, the integers as codomain (and range), and everyday names.

“I think I’m just going to have to practice with these definitions, Ila,” said Abu. “So the **floor** of a number is the *largest integer* less than or equal to that number. That makes 3 the floor of 3.14, I think.”

“Yes,” said Ila, “and the **ceiling** of a number is the *smallest integer* greater than or equal to that number. The floor of x is the ‘round down’ function — just another way to say it. Likewise, the ceiling of x is ‘rounding up’ — so the ceiling of 3.14 would be 4 — very straightforward.”

“Well good, let’s see how fast we can knock out these exercises!” said Abu.

Exercise 113 Define the “take the fractional part of” function `fracpart` in terms of `floor` as follows:

```
from math import floor
def fracpart(number):
    return number - floor(number)
```

For example,

```
fracpart(3.14159)
```

is

0.1415899999999988

Suppose `fracpart` were the primitive, builtin function. How would you define `floor` (assuming it were not builtin) using `fracpart`?

Hint 113 Be careful, the `fracpart` function as defined gives the wrong answer for negative numbers.

leftovers = x - int(x)).
Even now get this right and we're done! Let's do it with `math.floor(x)`:
AT A GLANCE But the calculations for x and $\text{fracpart}(x)$ are different — always — after

Exercise 114 The floor of x is denoted $\lfloor x \rfloor$, and the ceiling of x is denoted $\lceil x \rceil$. Graph these functions $f(x) = \lfloor x \rfloor$ and $c(x) = \lceil x \rceil$ for real number values of x in the interval from -5.0 to 5.0, inclusive.

friend here.

Hint 114 A tool like <https://www.desmos.com/calculator> is your

AT A GLANCE

Exercise 115 Which of these statements about floor and ceiling are correct, for any real number x and any integers n and m ?

1. $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$.
2. $\lfloor -x \rfloor = -\lceil x \rceil$.
3. $\lceil -x \rceil = -\lfloor x \rfloor$.
4. $\lfloor x + n \rfloor = \lfloor x \rfloor + n$.
5. $\lceil x + n \rceil = \lceil x \rceil + n$.
6. $\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil = n$.
7. $\lfloor \frac{n}{2} \rfloor + \lfloor \frac{n+1}{2} \rfloor = n$.
8. $\lceil \frac{n}{2} \rceil + \lceil \frac{n+1}{2} \rceil = n$.
9. $\lfloor \frac{n+m}{2} \rfloor + \lfloor \frac{n-m+1}{2} \rfloor = n$ (for any m , not just $m = 0$ as in statement 7).
10. $\lceil \frac{n+m}{2} \rceil + \lceil \frac{n-m+1}{2} \rceil = n$ (for any m , not just $m = 0$ as in statement 8).

import floor, ceil are ready to serve.

Hint 115 Write some code to help you decide. In Python, from math

as:

AT A GLANCE and if n is odd, then $\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil = n$ and $\lfloor \frac{n+m}{2} \rfloor + \lfloor \frac{n-m+1}{2} \rfloor = n$ for all m .

Exercise 116 Define the function “round to nearest integer” using the floor and ceiling functions, together with the “Iverson bracket” function — which you must define. A number whose fractional part is less than one-half should be rounded down, otherwise up.

inner, don't overthink it.
Hint 116 The Iverson bracket (search for which) function is a one-

AT **AS A ILL ANSWER**

3.10.2 Invert and Compose

“Well, that was fun!” said Abu. For a certain twisted definition of fun, thought Ila. “Next,” said Ila, suppressing her urge to sass, “Til wants us to briefly explore the inverse and composition of functions.”

“So he says,” she began, reading aloud to help them both unpack the definition, “the **inverse** of a function $f : A \rightarrow B$, denoted f^{-1} , is a function from B to A such that for all $a \in A$, and for all $b \in B$, if $f(a) = b$, then $f^{-1}(b) = a$.

“In other words,” said Abu, “the inverse maps the image of a function to its preimage.” “Yes,” said Ila, “that sounds right. So, for example, the inverse of $4x$ would be $\frac{1}{4}x$.” “And the inverse of x^2 ,” added Abu, “would be \sqrt{x} . I think we could easily find other examples, but let’s move on.” “Agreed,” said Ila.

Abu read this one: “The **composition** of two functions f and g , denoted $f \circ g$, is the operation of applying f to the result of applying g to something. That is, $(f \circ g)(x)$ is $f(g(x))$. To form the composition of (to compose) f and g , they must be **composable** (be compatible), meaning the codomain of g must be the same as (or a subset of) the domain of f .¹⁴

“Here,” said Ila, “I found this¹⁴ on Wikipedia.”

Intuitively, composing two functions is a chaining process in which the output of the inner function becomes the input of the outer function.

“So,” said Abu, again quoting Til, “functions f and g compose by passing as the argument of f , the outer function, the result of calling g , the inner function; hence, f of g of x .”

Abu paused, a little unsure about his parsing of this next part. But he soldiered on. “Note that $(f \circ g)(x)$ is not the same as $(g \circ f)(x)$ — i.e., composition is not commutative. If $f(x) = x^2$ and $g(x) = 3x + 1$, then $(f \circ g) = (3x + 1)^2 = 9x^2 + 6x + 1$, whereas $(g \circ f)(x) = 3(x^2) + 1 = 3x^2 + 1$.¹⁵”

“So here’s some code I just wrote,” said Ila, “that, to use Til’s term, ‘concretizes’ his example — using the letter ‘o’ for the ‘o’ symbol, fog is f applied after g , and gof is g after f (see Figure 3.22).”

```
def f(x): return x * x
def g(x): return x * 3 + 1
def fog(x): return f(g(x))
def gof(x): return g(f(x))
print('f(2) =', f(2),
      'and g(2) =', g(2))
print('fog(2) =', fog(2),
      'and gof(2) =', gof(2))
```

Figure 3.22: Composition of functions f and g , both ways. Prints:

$f(2) = 4$ and $g(2) = 7$
 $fog(2) = 49$ and $gof(2) = 13$

“You are amazing,” said Abu. “How do you type so fast?” Ila, pleased, said “Practice. Lots of practice!”

Exercise 117 *What conditions/restrictions must be placed on a function for it to have an inverse (be invertible)?*

Hint 117 A little research will fill the bill.

ANSWER 117 For a function to be invertible it must be a bijection.

Exercise 118 *Write code in the same fashion as Ila’s in Figure 3.22 to demonstrate the non-commutativity of the composition of two functions of your own choosing.*

a difference.

Hint 118 Make them simple functions, but different enough to make

AT A GLANCE

3.10.3 Sequences and Strings

Til was really pouring it on. “Let’s next define a **sequence** and distinguish it from a **set**. The difference is that a sequence is an **ordered** collection. So a list is just a sequence, and if you think about it, a sequence is really just a special kind of function.”

Abu stopped reading to think about it. Ila, to whom this was child’s play due to her programming experience, plowed on. “A finite **sequence** of elements of a set S is a function from the set $\{1, 2, 3, \dots, n\}$ to S — the number n is the **length** of the sequence. Alternatively, the domain of the sequence function can be $\{0, 1, 2, \dots, n - 1\}$.”

“So, hold on a minute,” Abu interrupted. “Let me see if I’ve got this. With n concretized as 10, `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` in Python list style, the sequence length is still n (ten) in this case, right?” “You got it!” said Ila. “Okay,” said Abu, “go on.”

Ila paused as she began reading the next part, hesitant to say out loud the notation a_n — was it pronounced ‘ay en’, ‘ay at en’, ‘a subscript n’, ‘a-down-half-a-line-n’ — what? She thought, How do I pronounce ‘`a[n]`’ in my code — presumably the same thing? — and realized she couldn’t really think of a time when she said such parts of her code out loud. She was annoyed with subscripts anyway, but then thought, I tolerate superscripts okay, why not subscripts? The answer came like a flash — a

long-buried memory in a math class where the teacher had embarrassed her when she had asked about the meaning of, of, oh! The case where a subscript *and* a superscript were used *at the same time!*

Abu, noticing her hesitation, and apparent consternation from the look on her face, prompted her gently. “I think it’s pronounced ‘a sub n’ —”, he said, and was immediately relieved that she didn’t take offense, but shook her head and gave him a faint smile, as if to say thanks. Her confidence restored, she resumed reading. “Generically, a sub n denotes the image of the (nonnegative) integer n , AKA the n^{th} term of the sequence. An *infinite* sequence of elements of S is a function from $\{1, 2, 3, \dots\}$ to S . Shorthand notation for the entire sequence is a_n . Again, the domain can be either \mathbb{Z}^+ or \mathbb{N} .”

Abu said, “Let me read through his example for a minute.” They both paused and read silently:

“For example,” Til wrote, “ a_n , where each term is just three times n , i.e., $a(n) = 3n$, is a sequence whose terms (generically a_1, a_2, a_3, \dots) would be 3, 6, 9, 12, … (all the multiples of three — infinite in number).”

“You okay?” asked Ila. “I think so — thanks,” said Abu. “Please continue reading out loud.”

Ila, still grateful for Abu’s sensitivity, was a little worried that he was not really okay with all this, but she said nothing. “So Til wrote something very interesting next,” she said. “He says another term for a *finite sequence* is a **string**, that venerable and versatile data structure available in almost all programming languages. Strings are always finite — if it’s an infinite sequence, it is most *emphatically* not a string!”

Abu envisioned a Python program that defined a string with an opening quote mark, a gazillion characters after that . . . but no closing quote! “Yes,” he said, “that makes sense. It’s like he said when we first started talking about sets how computers don’t deal with infinite sets very well.”

Ila nodded with a smile bigger than any Abu had ever seen her show. Obviously she had shrugged off whatever was bothering her earlier.

“We’re making progress!” said Ila. Ah yes, the opposite of congress! thought Abu. He was happy because of how happy Ila was, and didn’t want to spoil her good mood by subjecting her to his quirky sense of humor.

“Moving on, Til writes that there are two fundamental types of integer sequences — called *progressions*.” Ila couldn’t help it. “Confusingly enough — again with this more-than-one-word-to-say-the-same-thing business!”

3.10.4 Arithmetic Progressions

Abu grinned, and took over reading. Unlike Ila, he was not bothered by this “business.” It amazed him that Ila’s mood could swing back to

annoyance so quickly.

An arithmetic progression is a sequence of the form $a, a+d, a+2d, a+3d, \dots$. In other words, the sequence starts with a , and the common difference between the next term and the current term is d . For example: $20, 25, 30, 35, \dots$ where $a = 20, d = 5$.

3.10.5 Geometric Progressions

Ila, recovered from her pique, asked, “Do you see how you could write that example in code?” Abu nodded, and she went on. “The second fundamental type of integer sequence is a **geometric progression**, which is a sequence of the form $a, ar, ar^2, ar^3, ar^4, \dots$. In other words, the sequence starts with a , and the common *ratio* between the next term and the current term is r . For example: 3, 6, 12, 24, 48, … where $a = 3, r = 2$.”

“Difference versus ratio, I get it,” said Abu.

Ila stared into space for a couple seconds, turned back to her computer, did a quick search, typed furiously, and then said excitedly, “Abu! I just found a BIF in Python that creates arithmetic progressions! I thought something like that must exist, and here it is (see Figure 3.23).”

Abu was intrigued. He did a quick calculation, and said “Ila, I just noticed something interesting. You just chose those numbers randomly, right?” “Yeah, so?” said Ila. “Take that last example,” said Abu. “The ending value 141 does not equal 140, which is $21 + 7 \cdot 17$, so it looks like the sequence stops as soon as adding the delta value would be bigger than the ending value.” “Uh-huh,” said Ila, “that’s because the second argument, the ending value, is what’s called *exclusive* — which, by the way, I got from the builtin Python documentation — `help(range)` — see, you can get that help for any function, which is super chill. So like in my first example that only goes to 17 because the 18 is excluded. What it means is that you really stop if adding the delta value would *equal* or exceed the ending value.” Abu looked at Ila with admiration. “You really get this, don’t you!” he said. “This, yes. But don’t sell yourself short — I think you get stuff I don’t, at least at first.” “We make a good team, then!” said Abu, and Ila just beamed.

Exercise 119 What is the a (starting value) and the d (delta, or increment) for the infinite sequence $[7, 14, 21, 28, 35, 42, 49, 56, 63, 70, \dots]$?

Hint 119 Take the difference between any term and the one before it to get d . It should be obvious what a is.

ANSWER **ALL** **THE** **STATISTICS** **ARE** **SEEN**, **THE** **DATA** **IS** **ALSO** **SEEN**.

```
print(list(range(10, 18)))
print(list(range(4, 31, 3)))
print(list(range(1, 37, 4)))
print(list(range(21, 141, 17)))
```

Figure 3.23: Arithmetic progressions created by the (listified) range function, parameterized by a starting value (inclusive), an ending value (exclusive), and the (optional, default value 1) delta (increment) between successive values. Prints:

```
[10, 11, 12, 13, 14, 15, 16, 17]  
[4, 7, 10, 13, 16, 19, 22, 25, 28]  
[1, 5, 9, 13, 17, 21, 25, 29, 33]  
[21, 38, 55, 72, 89, 106, 123, 140]
```

Exercise 120 What is the a (starting value) and the r (ratio) for the infinite sequence [6, 12, 24, 48, 96, 192, 384, 768, 1536, 3072, ...]?

Hint 120 Take note the difference between two consecutive terms but their what?

AT a **ASI** **ANSWER**

Exercise 121 Using the range function, create an arithmetic progression that has a negative delta.

Hint 121 And isitify it to make it printablae.

AT a **ASI** **ANSWER**

Exercise 122 Is there a Python BIF that creates a geometric progression?

Hint 122 Have you found a good source of Python documentation yet?

AT a **ASI** **ANSWER**

Exercise 123 Many, many other types of sequences exist. For example, identify the formula or rule (“rulify”) that generates the terms of the sequence [3, 4, 6, 9, 13, 18, 24, 31, 39, 48, ...].

Hint 123 Try subtracting each term from the one before it.

AT a **ASI** **ANSWER**

Exercise 124 Rulify the sequence [1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, ...].

Hint 124 Treat each number as a binary number and convert it to a decimal number.

AT A GLANCE

Exercise 125 Rulify $[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots]$.

Hint 125 Starting with the third term, look at the differences between each term and its predecessor. Compare those differences with the original sequence.

AT A GLANCE

Exercise 126 Rulify $[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, \dots]$.

Hint 126 Differences won't help here. Look at the next term when the current term is even, versus when it is odd. This should ring a bell!

AT A GLANCE

Exercise 127 Rulify $[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, \dots]$.

Hint 127 Compare this with the sequence in Exercise 125.

If the sequence starts with 2, it follows a pattern where every second term is odd and every third term is even. If the sequence starts with 1, it follows a pattern where every second term is even and every third term is odd. If the sequence starts with 3, it follows a pattern where every second term is odd and every third term is even. If the sequence starts with 4, it follows a pattern where every second term is even and every third term is odd.

Exercise 128 Rulify $[3, 5, 7, 11, 13, 17, 19, 23, 29, \dots]$.

Hint 128 Notice anything odd about these numbers?

AT A GLANCE

Exercise 129 Sequences can be non-numeric as well. What is the pattern for the sequence $[f, t, o, d, h, d, r, f, th, od, do, ha, di, re, fir, thr, \dots]$? How long can it continue?

Hint 129 Don't think too hard about this.

comqđesefly sđeñeq oñt.
fle of tñis doof. It can continue until the shortest word ("s") is
that contains a word with (but still) words out the first word
the first word is that contains the word "at".
AT ñ ñSA 129 TewsmA

Exercise 130 How many solutions are there in single-digit integers (positive or negative) to the equation $3x + 4y = 7$?

Hint 130 Use the `range` function with a nested loop.

AT ñ ñSA 130 TewsmA

Exercise 131 How many pairs of numbers each taken from the set $\{2, 3, 4, 5, 6, 7\}$ that when multiplied together equal one more than a multiple of 9 are there?

Hint 131 See the hint to Exercise 130.

AT ñ ñSA 131 TewsmA

Exercise 132 How many pairs of numbers that when multiplied together equal one more than a multiple of 11 are found in the set $\{2, 3, 4, 5, 6, 7, 8, 9\}$?

Hint 132 See the hint to Exercise 130.

AT ñ ñSA 132 TewsmA

3.10.6 Summations

Coming back together after several days off, Abu was excited to tell Ilia: "I think I'm getting the hang of this language. And Python and math are really quite close!"

Ilia managed a thumbs up in reply, but was unable to generate much enthusiasm, having had too little sleep the night before. Why did her

husband have to keep her up so late so often? She grimaced, shook off her fatigue, and said, “Do you think there’s a Python equivalent for the next GPAO topic Til gave us, the symbol Σ for summations, or sums?”

Abu shook his head no, and then reconsidered. “Let’s see. Oh yeah, here it is! The `add` function — operator — from the `operator` package no less — can be used with `reduce` to turn sequences into summations.” Here Abu copied the example he found online and showed it to Ila. “Once again,” he said, triumphantly, “behold the power of functional programming (see Figure 3.24)!”

“Don’t be so dramatic,” said Ila, and read aloud Til’s definition. “A **summation** (sometimes confusingly called a **series**),” (Arrgh! thought Ila, but went on,) “denoted by the Greek letter Sigma (Σ) collapses a sum of many related terms into a single term parameterized by a variable called the **index** of summation that ranges from a lowest to a highest value. Thus, with i as the index of summation ranging from 1 to n ,

$$f(a_1) + f(a_2) + \cdots + f(a_n)$$

collapses to

$$\sum_{i=1}^n f(a_i).$$

For example,

$$\sum_{k=1}^4 k^2 = 1^2 + 2^2 + 3^2 + 4^2 = 1 + 4 + 9 + 16 = 30$$

where $f(k) = k^2$.

“Okay, cool, summing the first four squares,” said Abu. “Wait,” said Ila, “let’s finish what he said and then talk about it.” She picked up, “Note that k is being used instead of i , and in general, it doesn’t matter what the index variable is, nor what its lowest value is — it can start at 1, or 0, or indeed any nonnegative integer.”

“Interesting,” said Abu. “Just like the `range` function — although I think that can start negative too.” “Yes,” said Ila, “however, I think math prefers not to have negative indexes.” “You’re probably right, although I don’t think ‘indexes’ is the right word for the plural of ‘index’.” Glancing ahead in Til’s exposition, Ila said, “Oh, you’re right — but that’s coming next —” Abu, excited, broke in with “I had a thought — it’s nice that Til used the words ‘ranges’ and ‘ranging’ — it shows that Python’s using ‘range’ as the function’s name — well, it just juices my idea that Python and math are really close!”

“Let’s move on,” said Ila, a little irritated at his interrupting her. But as she had been reading ahead, she had only been half-listening to Abu, and what she read added to her irritation and confirmed her fear that Til, never one to leave well enough alone, in person or in writing, would take it to the next level. “When there’s more than one index they’re called

```
from operator import add
from functools import reduce
print(reduce(add, range(1, 101)))
```

Figure 3.24: Reduce-with-add a range to get the sum 5050. As Abu and Ila will discover later, this can be done more simply, thus: `sum(range(1, 101))`

indices, and with these we can nest summations (to any depth, just like loops):

$$(1 \times 1 \times 1 + 1 \times 1 \times 2) + \\ (2 \times 2 \times 1 + 2 \times 2 \times 2) +$$

$$(3 \times 3 \times 1 + 3 \times 3 \times 2) = 42$$

Table 3.2: Nested summation term by term and final sum.

```
sum = 0
for k in range(1, 4):
    for i in range(1, 3):
        sum += k * k * i
print(sum)
```

Figure 3.25: Nested summation, sums and prints 42.

¹⁵This is a formula computed via arithmetic operations on the variables in the summation, *except* for the index of summation. It is “closed” as opposed to just listing all the summands in “open” form, where addition of the summands is the only operation.

¹⁶Can you see why?

$$\sum_{k=1}^3 \sum_{i=1}^2 k \times k \times i = \sum_{k=1}^3 (k \times k \times 1 + k \times k \times 2).$$

This is expanded in Table 3.2 and Pythonized in Figure 3.25.”

“Okay, that wasn’t so bad,” capitulated Ila. “Oh wait, it gets worse.” “Allow me,” said Abu.

“An arithmetic progression,” continued Til’s explanation, “from any lower bound (index $i = l$) to a higher upper bound (index $i = u, u \geq l$) has a *closed-form formula*¹⁵ for its sum:

$$\sum_{i=l}^u a + i \cdot d = (u - l + 1)(a + d(u + l)/2).$$

For $a = 0, d = 1, l = 1, u = n$ we have the sum of the first n positive integers, a well-known special case:

$$\sum_{i=1}^n i = 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}.$$

If we take this summation and form a sequence of its partial sums as n increments, the resulting sequence 1, 3, 6, 10, 15, 21, 28, …, is AKA the ‘triangular’ numbers.¹⁶

Back to Ila. “The upper bound of a summation can be ∞ , in which case the Σ is called an **infinite series**, and can either **converge** (produce a finite number) or **diverge** (grow “infinitely large,” or in other words, grow “without bound”). Naturally, the sum of an arithmetic or geometric progression where the terms are all positive integers will diverge to ∞ if the sum has an infinite number of terms. Interestingly, when the common ratio of a geometric progression is a positive fraction less than 1, the sum will converge, and is given by another closed-form formula.”

$$\sum_{i=0}^{\infty} ar^i = \frac{a}{1-r}, 0 < r < 1$$

Abu could see that writing some code would help unpack all these definitions and formulas. “My turn,” he said. “Til is really making us think, huh?” Ila glared at him.

Abu, unperturbed, kept reading. “So, the sum of the reciprocals of the powers of two is a simple example where $a = 1$ and $r = \frac{1}{2}$.

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

We can use this special case to show why this formula works — and it is easily generalized. Call S the sum

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots.$$

We can factor out the common ratio $\frac{1}{2}$ from each term to get

$$\frac{1}{2}(2 + 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots).$$

We see that

$$S = \frac{1}{2}(2 + S) \rightarrow 2S = 2 + S \rightarrow S = 2.$$

An interesting variation of this series is the sum of the reciprocals of the powers of two *scaled by which power*:

$$\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \frac{5}{32} + \frac{6}{64} + \dots = \sum_{n=1}^{\infty} \frac{n}{2^n} = 2.$$

Showing why this also sums to 2 is left as an exercise for you!"

"Oh, brother," said Ila. "Here's another fun one." She picked up where Abu left off. "The sum of the reciprocals of the products of consecutive pairs of positive integers:

$$\begin{aligned} \sum_{n=1}^{\infty} \frac{1}{n(n+1)} &= \sum_{n=1}^{\infty} \left(\frac{1}{n} - \frac{1}{n+1} \right) = \\ \left(1 - \frac{1}{2}\right) + \left(\frac{1}{2} - \frac{1}{3}\right) + \left(\frac{1}{3} - \frac{1}{4}\right) + \left(\frac{1}{4} - \frac{1}{5}\right) \dots &= \\ 1 \left(-\frac{1}{2} + \frac{1}{2}\right) \left(-\frac{1}{3} + \frac{1}{3}\right) \left(-\frac{1}{4} + \frac{1}{4}\right) \left(-\frac{1}{5} + \frac{1}{5}\right) \dots &= 1. \end{aligned}$$

And converging somewhere between 1 and 2 is this special sum with a fascinating history, the sum of the reciprocals of the squares:¹⁷

$$\frac{1}{1} + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \frac{1}{25} + \frac{1}{36} + \dots = \sum_{n=1}^{\infty} \frac{1}{n^2}.$$

"My word," said Abu, "it gets deeper and deeper!" He took over again:

"If the scale factor, or term multiplier, is a constant, distributivity of multiplication over addition justifies the factoring out of that constant from each term. Hence:

$$c \cdot a_1 + c \cdot a_2 + \dots + c \cdot a_n = \sum_{i=1}^n c \cdot a_i = c \cdot \sum_{i=1}^n a_i.$$

Associativity and commutativity of addition and subtraction justify:

$$\sum_{i=1}^n (a_i \pm b_i) = \sum_{i=1}^n a_i \pm \sum_{i=1}^n b_i.$$

Sums can be broken into two or more pieces:

$$\sum_{i=a}^z s_i = \sum_{i=a}^m s_i + \sum_{i=m+1}^z s_i, a < m < z.$$

¹⁷ Euler ingeniously found the correct closed-form solution to this famous summation, the challenge of which was known in his day as the *Basel problem*.

Nested summations with two indices of summation controlling a binary function can be interchanged:

$$\sum_{j=1}^n \sum_{k=1}^j f(j,k) = \sum_{k=1}^n \sum_{j=k}^n f(j,k).$$

"Definitely going to have to write some Python to make sense of these," said Ila. "But my brain is fried. Let's pick this up and work on these exercises later." Abu agreed. "Absolutely. And let's be glad there remains but one more GPAO to tackle before we resume our meetings with Til!"

Exercise 133 Why is the sum of the reciprocals of the powers of two scaled by which power equal to two?

Hint 133 Showing why this is so is an exercise in pattern recognition, but the right approach is to exploit by rearranging terms of the summation, but the infinite sum the commutativity and associativity of addition — realising that the sum can be done in any order.

runme(J, J0000)))
begin(response(qqq' wab(jawpqa n: n \ s * n' /
from functionimbout response
from operatorimbout qqq

How apostle a little Paulous;

$\frac{1}{1-x}$	$\frac{1}{1-2x}$	$\frac{1}{1-3x}$	$\frac{1}{1-4x}$	$\frac{1}{1-5x}$	\dots
$= 1 + x + x^2 + x^3 + x^4 + \dots$	$= 1 + 2x + 2x^2 + 2x^3 + 2x^4 + \dots$	$= 1 + 3x + 3x^2 + 3x^3 + 3x^4 + \dots$	$= 1 + 4x + 4x^2 + 4x^3 + 4x^4 + \dots$	$= 1 + 5x + 5x^2 + 5x^3 + 5x^4 + \dots$	\dots
$= 1 + x + x^2 + x^3 + x^4 + \dots$	$= 1 + 2x + 2x^2 + 2x^3 + 2x^4 + \dots$	$= 1 + 3x + 3x^2 + 3x^3 + 3x^4 + \dots$	$= 1 + 4x + 4x^2 + 4x^3 + 4x^4 + \dots$	$= 1 + 5x + 5x^2 + 5x^3 + 5x^4 + \dots$	\dots
$= 1 + x + x^2 + x^3 + x^4 + \dots$	$= 1 + 2x + 2x^2 + 2x^3 + 2x^4 + \dots$	$= 1 + 3x + 3x^2 + 3x^3 + 3x^4 + \dots$	$= 1 + 4x + 4x^2 + 4x^3 + 4x^4 + \dots$	$= 1 + 5x + 5x^2 + 5x^3 + 5x^4 + \dots$	\dots
$= 1 + x + x^2 + x^3 + x^4 + \dots$	$= 1 + 2x + 2x^2 + 2x^3 + 2x^4 + \dots$	$= 1 + 3x + 3x^2 + 3x^3 + 3x^4 + \dots$	$= 1 + 4x + 4x^2 + 4x^3 + 4x^4 + \dots$	$= 1 + 5x + 5x^2 + 5x^3 + 5x^4 + \dots$	\dots

of fmo' firs firme startin'g mifus fmo fo firs bomei zero:

сумы оѣ тѣс „парт“ соїтумы — тѣс сумы оѣ тѣс 1есѣнѣюсаѣ оѣ тѣс 1оморес
сумы оѣ тѣс 1иреніюса 1омъ 1иаѣтизѣ тѣс 1иѣтире сумы 1иѣст тѣс (1иѣмитѣ)
1оїзюнизѣ 1омозъ 1иѣтизѣ тѣс сумы оѣ саси 1омъ 1иѣст тѣс 1иѣст 1иѣст 1иѣст тѣс
Анализ 133 Всегда сдвиг тѣс сумы $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots$ вправо тѣс

Exercise 134 Explore the counterpart of Σ , the symbol used to collapse a sum of terms; namely, the Greek letter (which?) used to collapse a product of terms.

Hint 134 Π is for Product.

AT A GLANCE

Exercise 135 With at least two examples of a binary function, verify the nested summation interchange rule.

Hint 135 Use your creativity.

Line

ט' ט' ט' ט' ט'

JJ' JS' 8' θ' JT0' JJ' JS' J3' JT0' JJ' JS' J3' JT4' JS' J3' JT2' JT' JT'
[S' 3' 4' 8' 2' 2' 8' 1' 2' 2' 8' 1' 2' 2' 8' 1' 2' 2' 8' 1' 2' 2' 8' 1' 2' 2']
JT' JT2' JT' JT'

a))

(e) ((1' 1') (2' 2') (3' 3') (4' 4') (5' 5') (6' 6') (7' 7') (8' 8') (9' 9') (10' 10'))

BESTS:

```
print(result == result)
```

RESULTS

resuftss.sut()

brünft (resurfs)

resufts4 += (jmpq x, λ: fnbje([x + λ]))(j, k)

FOR KIDS

result = []

ပြန်လည်သင့်အမျိုးအစား (Reinforcement)

Reshetskiy = (fun x. x + y)(fun x. x + y)

for i in range

$\text{Eisen}_3 = []$

BRITISH JOURNAL OF

卷之三

http://www.elsevier.com/locate/jmaa

178

LCAGCCSE = [1]

`b1 = f1(cabs(wab(cabs('1234567'))))`

`lesser? x = (greater x λ: (abs(([[x, λ]])))(λ: x`

100 | THE JOURNAL

```
resu[fs] = []
```

Exercise 136 Consider the following function:

```
from operator import add
from functools import reduce

def calculate_pi_very_slowly(max_iterations):
    return 4 * reduce(add, map(lambda n: \
        1.0 / (2 * n + 1) * (-1) ** n, \
        range(0, max_iterations + 1)))
```

How slow is very slowly?

Hint 136 Research how to do Python benchmarking.

AT A GLANCE

3.10.7 The Harmonic Series

“One of the most, if not *the* most, counterintuitive sums is known as the *harmonic series* — simply described as the sum of the reciprocals of the positive integers.” Abu noticed Ila tense up, as she always did when something was described as counterintuitive. But he was glad she was eager to tackle their last topic so they could meet with Til again.

“We’ve seen in a geometric series terms getting smaller and smaller, and their infinite sum converging to a definite finite number. Whenever we have an infinite series that adds up more and more of less and less, it would seem that this would be the case. That’s what our intuition tells us, but for the harmonic series it turns out *not* to be the case! Here is just one of many different ways¹⁸ to see this:”

Oh boy, thought Abu. This is going to be hairy.

“Let $H_n = \sum_{i=1}^n \frac{1}{i}$ be the partial sum of the reciprocals of the positive integers, up to the reciprocal of n . There are 9 one-digit numbers, 1 to 9, whose reciprocals are greater than $1/10$. Therefore by matching these inequalities term by term and adding up all nine of them:

¹⁸ Another way will be shown later, but a GPAO awaits anyone wanting to see the whole plethora of proofs out there.

$$\frac{1}{1} > \frac{1}{10}$$

$$\frac{1}{2} > \frac{1}{10}$$

$$\frac{1}{3} > \frac{1}{10}$$

$$\frac{1}{4} > \frac{1}{10}$$

$$\frac{1}{5} > \frac{1}{10}$$

$$\frac{1}{6} > \frac{1}{10}$$

$$\frac{1}{7} > \frac{1}{10}$$

$$\frac{1}{8} > \frac{1}{10}$$

$$\frac{1}{9} > \frac{1}{10}$$

Or in sum:

$$H_9 > \frac{9}{10}.$$

Likewise, there are 90 two-digit numbers, 10 to 99, whose reciprocals are greater than 1/100. Therefore

$$H_{99} > \frac{9}{10} + \frac{90}{100} = 2\left(\frac{9}{10}\right).$$

In general, where the subscript of H has k 9s in total:

$$H_{99\dots 9} > k\left(\frac{9}{10}\right).$$

So the sum grows forever unbounded!"

And with that epitome of unboundedness, Til's massive missive came to an abrupt end.

Exercise 137 Investigate and determine how many terms of the harmonic series you have to add up to reach a given number x .

Hint 137 The answer will be a function of x

```

x += π
butnf(μu' u` λonq(exp(x - 0.2πππ))
t̄t μu >= x:
μu += π \ u
for u in range(j, 100000):
μu' x = 0' π
flow wifn twbolf exp

meantest mīgēs: Yoi can uenifl this with the following code:
it's smirly abdoroximātē(y) e_x_y tñbale y ≈ 0.2πππ rotmpleq to the
first readif (or exceedq) x after the number.
TIL TERSNA

```

3.11 Predicates

True to form, there was not even a hint of acknowledgment by Til that it had been quite a while since they last met. Nor any manner of small talk. Just onward and upward.

“Functions that return true or false are used all the time,” said Til, as he welcomed Abu and Ila back into his study, “and they have a special name — **predicate**. This designation applies to any function whose domain is whatever you want — any conceivable set — but whose codomain is confined to the set {true, false}. A predicate is AKA a **property**. We could also call them **deciders** — or *decision-makers* — that decide whether or not a given object has some property.”¹⁹

“Let me explain a nuance,” continued Til, “and then we’ll look at a couple of examples. As used in *predicate logic* — a superset of propositional logic — the noun **predicate** means — to quote from the dictionary now — *that part of a proposition that is affirmed or denied about the subject*. For example, in the proposition ‘We are mortal,’ ‘mortal’ — or ‘is-mortal’ or ‘are-mortal’ — is the predicate. Any statement about something could ‘predicate’ — the verb form — some general term of that thing. However, things could get confusing. For example, ‘The main purpose of the Law is to enforce social order.’ It’s not easy to pull out a term which, predicated of ‘the Law’, has the same meaning. So if we’re saying $P(x)$, where x stands for ‘the Law’, what does P stand for — ‘to-enforce-social-order-main-purpose-of’?! No, we mostly never need to say what P stands for by itself — just let $P(x)$ stand for any sentence with an x in it.”

That was one heck of a nuance, thought Abu. He would definitely need to talk to Ila about this. Parsing English was never his strong suit.

“Okay,” Til went on, after pausing to catch his breath a little, “you’ve already seen two predicates I referred to earlier as *adjectives* — *injective* and *surjective* — three, actually, counting *bijective*. Some word that describes an object indicates that a property identified by that word is possessed by the object. So, saying x^2 is an “injective function” — which it is or isn’t, depending on its domain, of course²⁰ — is saying that the function x^2 has the *injective* property.”

¹⁹ Recall § 2.11. Set-builder notation in a nutshell: $\{x \mid P(x)\}$ where P is any predicate.

²⁰ The statement “ $\text{injective}(x^2)$ ” is true if the domain in question is restricted to positive numbers, but false if the domain includes negative numbers.

3.12 Quantifiers

"Let's introduce an accompanying concept to predicates with a familiar example," continued Til. "We know all the colors in the 'rainbow-colors' domain, so we could say, *for all the colors in the rainbow, each is beautiful* — or more succinctly, *every rainbow color is beautiful*. These **quantifiers all** and **every** apply to the whole of our universe of discourse."

Ila was seeing the rainbow painted on the wall of her childhood bedroom. Her husband had nixed the painting of a rainbow on their master bedroom wall!

"A **universe of discourse** (AKA **domain of discourse**) is the set of all things we've set forth to talk about, i.e., *all things under consideration*. It must be specified — unless the *real* universe is our universe of discourse — to make sense of propositions involving predicates."

Abu was considering all the things he had learned so far from Til, which was a set of immense size, he realized!

"The **universal quantifier**, denoted \forall — yes, an upside-down 'A' — think 'A' for 'All' — says something is true for *all* members of this universal set, no exceptions. By contrast, \exists — the backwards 'E' (for 'Exists') — is the **existential quantifier**, and says something is true for *some* member (i.e., at least one) of this universal set."

Ila remembered seeing verbiage like "for all" and "for some" before — she resolved to ask Til if he didn't mention this.

"We can use these symbols to be more succinct. Saying something like $\forall x B(x)$ rather than $B(\text{red}) \wedge B(\text{orange}) \wedge B(\text{yellow}) \wedge B(\text{green}) \wedge B(\text{blue}) \wedge B(\text{indigo}) \wedge B(\text{violet})$ — because, of course, each and every one is true — is much shorter, which is especially important when the domain is large — or infinite!"

Again with the infinite, thought Abu. He had a hard enough time dealing with the finite! As if reading his mind, Til continued with a finite example — but not for long!

"So if $\{2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42\}$ is the domain x comes from, $\forall x Even(x)$ is true. But the same quantified statement is false when the domain is that same set only with 42 replaced by 41. And indeed, $\forall x Even(x)$ is infinitely shorter than a conjunction of unquantified propositions $Even(2) \wedge Even(4) \wedge \dots$, one for each member of the set of even integers! Here's another easy example: Let P be the predicate *is a ball*. So $P(x)$ means x is a ball. Is $P(x)$ true?" Til asked this very quickly, as if to say *wake up!*

Ila, whose eyes had been glazing over, jerked back to full attention, and then noticed Abu looked a little sheepish. Perhaps he had allowed his attention to wander a little too, so she thought a moment and said, "I don't think — and I'm sure Abu would agree — that you can say one way or the other." Abu nodded his head vigorously, trying to clear the fuzz.

"That's right!" said Til. "A predicate²¹ is *not*, I repeat, *NOT* a proposition. So let's make it into one."

With a flourish Til added and displayed $\exists x P(x)$. "Meaning?" he said, expectantly.

Abu answered quickly. "Meaning there exists an x such that x is a ball. Or, I guess you could say, there is something that is a ball."

"Is this true, Ila?" said Til. Ila was wary, and for good reason, given the cautionary tone of voice Til had just used.

"It depends." she said. "What's the *universe of discourse*!?"

"Very good!" said Til. "Let's say it's the set of all kitchen appliances in your house." Ila smiled and said, "In that case, it's false."

"So negate it," said Til, "and make a false statement into a true one — in our kitchen appliances universe, still — and $\neg\exists x P(x)$ says what, Abu?"

Abu was quick to reply, "There does not exist an x such that x is a ball. Or, there is no ball."

"Ila," continued Til, "contrast that with $\exists x \neg P(x)$." "That means," said Ila, just as quickly as Abu, "There exists an x such that x is not a ball. So there is something that is not a ball."

Abu asked, "But that's *not* the same as saying *Everything is not a ball*, right?"

Ila answered, "Right! That would be $\forall x \neg P(x)$: *For every x, it is not the case that x is a ball*."

Til said, "Now move the negation to the outside: $\neg\forall x P(x)$ means *Not everything is a ball*. I'll give you the rule for negating quantifiers in a minute, but just to get a better feel for it, you two do another example, say with the predicate R for *is red*. Cover all the cases, and be as succinct as possible."

"Okay," said Ila, "but I want to use y instead of x — does that fly with you, Abu?" "Y of course!" said Abu, trying but failing to keep a straight face. But Ila seemed not to notice the sly smile that played on his lips. Til did, of course, and winked at Abu when Ila wasn't looking.

"First," said Ila, $\forall y R(y)$ means *Everything is red*."

"Negated," jumped in Abu, " $\neg\forall y R(y)$ means *Not everything is red*."

"Which means the same thing," continued Ila, "as $\exists y \neg R(y)$, or *There is some non-red thing*."

To which Abu concluded, "Remove the negation to get $\exists y R(y)$, which means *There exists some red thing*, or *There is something that is red*."

"Very good!" said Til. "Now let's examine a *Sillygism* — recalling that the three dots stacked in a triangle — '∴' — means 'therefore':"

Object-oriented is good.
Python is good.
∴ Python is object-oriented.

²¹Or more precisely, a propositional function (or formula) consisting of a predicate and some variable(s).

"That certainly *sounds* silly," said Abu. "Very true!" said Til. "It's better as a *Syllogism*:"

Object-oriented is good.
Python is object-oriented.
. . . Python is good.

"Now it's right. In general, for $G(x) = x$ is good, and $O(x) = x$ is object-oriented:"

$\forall x (O(x) \rightarrow G(x))$
 $O(\text{Python})$
. . . $G(\text{Python})$

"While we're on this tangent," said Til, "here's a well-known Syllogism:"

All men are mortal.
Socrates is a man.
. . . Socrates is mortal.

"This pattern — *all something are something else* — is expressed most naturally as a universally quantified implication: $\forall x (P_1(x) \rightarrow P_2(x))$.²² So let $S(x)$ be 'x is a student who knows logic', and $T(x)$ be 'x has taken a course in discrete mathematics'."

$\forall x (S(x) \rightarrow T(x))$.

"But more succinctly, if the domain of discourse is decided on in advance to be *students who know logic*, we can simply say:"

$\forall x T(x)$.

"Now," said Til, "here are a couple of heuristics — not ironclad rules, just rules of thumb. Exemplified by the first line of the syllogistic form we just saw — as a rule of thumb, *universal* quantifiers are followed by *implications*. For another example, the symbolic form of

Every prime greater than 2 is odd.

is

$\forall x ((\text{Prime}(x) \wedge \text{GreaterThanTwo}(x)) \rightarrow \text{Odd}(x))$

or — here's a shorthand for that —

$\forall x > 2 (\text{Prime}(x) \rightarrow \text{Odd}(x))$

but *not*

$\forall x > 2 (\text{Prime}(x) \wedge \text{Odd}(x))$.

²²This is equivalent to the subset-to-set relationship: $\{x \mid P_1(x)\} \subseteq \{x \mid P_2(x)\}$.

As a rule of thumb, *existential* quantifiers are followed by *conjunctions*.
For example, the symbolic form of

There exists an even number that is prime.

is

$$\exists x (Even(x) \wedge Prime(x))$$

but *not*

$$\exists x (Even(x) \rightarrow Prime(x))$$

which says, nonsensically, some number is even only if it's prime."

3.13 Negating Quantifiers

"Now for that other rule — not of thumb — that I said I'd give you in a minute:"

When negating a quantified proposition, move the negation sign from the left side to the right side of the quantifier, and CHANGE THE QUANTIFIER from universal to existential (or vice versa).

"Take, for example, the negation of the statement *some cats like liver*. It is *not* the statement *some cats do not like liver*. The negation is *no cats like liver*, or *all cats dislike liver*. So, setting *cats* as the universe of discourse, and letting *L* be the predicate *likes liver*:

$$\exists x L(x)$$

says

Some cats like liver.

$$\neg \exists x L(x) \equiv \forall x \neg L(x)$$

says

All cats dislike liver

or

No cats like liver.

$$\neg \forall x L(x)$$

is

It is not the case that all cats like liver

or

Not all cats like liver.

And finally,

$$\neg \forall x \neg L(x)$$

is

Not all cats dislike liver,

or

Some cats like liver,

which is what we started with."

Exercise 138 In this and subsequent exercises, unless otherwises stated, the domain of discourse is the set of all people, and:

$P(x)$	=	<i>x is older than 21</i>
$S(x)$	=	<i>x is a student</i>
$F(x)$	=	<i>x is a friend</i>
$C(x)$	=	<i>x is cool</i>
$N(x)$	=	<i>x is from Nepal.</i>

Express in good English ("anglify") the statement $\exists x P(x)$.

Hint 138 Don't use the word 'exists'.

AT ð ñSA 831 TEWNSA

Exercise 139 Anglify $\forall x P(x)$.

Hint 139 Don't use the word 'all'.

AT ð ñSA 831 TEWNSA

Exercise 140 Anglify $\exists x \neg P(x)$.

Hint 140 Use the word 'someone'.

AT ð ñSA 831 TEWNSA

Exercise 141 Anglify $\forall x \neg P(x)$.

Hint 141 *No one*, sounds better than *Everyone is not*.

Exercise 142 Anglify $\exists x S(x)$.

Hint 142 *Some* <something> *is a* <something else>.

Exercise 143 Anglify $\forall x S(x)$.

Hint 143 *Start with Everyone*.

Exercise 144 Anglify $\neg \exists x S(x)$.

Hint 144 *Start with There is no*.

Exercise 145 Anglify $\exists x \neg S(x)$.

Hint 145 *Start with There is a*.

There is someone who is not you.
There is someone who is not you.

Exercise 146 Anglify $\neg \forall x \neg S(x)$.

Hint 146 *Start with Not all*.

Not all are alike.

Exercise 147 Anglify $\forall x \neg S(x)$.

Hint 147 Start with 'No one'.

AT a ~~asA~~ ~~g4l r4wsuA~~

Exercise 148 Anglify $\forall x (F(x) \rightarrow C(x))$.

Hint 148 Start with 'All'.

AT a ~~asA~~ ~~g4l r4wsuA~~

Exercise 149 Anglify $\exists x (F(x) \wedge C(x))$.

Hint 149 Start with 'Some'.

AT a ~~asA~~ ~~g4l r4wsuA~~

Exercise 150 Anglify $\forall x (F(x) \wedge C(x))$.

Hint 150 Start with 'Everyone'.

AT a ~~asA~~ ~~g4l r4wsuA~~

Exercise 151 Anglify $\exists x (F(x) \rightarrow C(x))$.

Hint 151 Start with 'Some'.

AT a ~~asA~~ ~~g4l r4wsuA~~

Exercise 152 Anglify $\forall x (S(x) \rightarrow N(x))$.

Hint 152 Are you getting the hang of this?

AT a ~~asA~~ ~~g4l r4wsuA~~

Exercise 153 Anglify $\exists x (S(x) \rightarrow N(x))$.

Hint 153 Start with "Someone and use if and then".

Mədən.

Hint 153 Start with "Someone and use if and then".

Exercise 154 Anglify $\forall x (S(x) \wedge N(x))$.

Hint 154 Start with "Everyone".

Hint 154 Start with "Everyone".

Exercise 155 Anglify $\exists x (S(x) \wedge N(x))$.

Hint 155 Some student, ...

Hint 155 Some student, ...

Exercise 156 Translate into logical expressions using predicates, quantifiers, and logical connectives ("translog") the statement "Everyone's a critic."

and $F(x) = "x \text{ is your friend}"$.

Hint 156 Use the predicates $C(x) = "x \text{ is a critic}"$, $P(x) = "x \text{ is perfect}"$

Hint 156 Use the predicates $C(x) = "x \text{ is a critic}"$, $P(x) = "x \text{ is perfect}"$

Exercise 157 Translog "No one is perfect."

Remember (and change the flavor of the quantifier).

Hint 157 Remember to push the negation sign to the right of the

Hint 157 Remember to push the negation sign to the right of the

Exercise 158 Translog “At least one of your friends is perfect.”

quantifiers?

Hint 158 Remember which connective to use with the existentialAT a **asA** **g2l** **rewsua****Exercise 159** Translog “All of your friends are critics.”

quantifiers?

Hint 159 Remember which connective to use with the universalAT a **asA** **g2l** **rewsua****Exercise 160** Translog “Everyone is a critic or someone is your friend.”

ments.

Hint 160 This looks like the disjunction of two quantified state-AT a **asA** **g2l** **rewsua****Exercise 161** Translog “No one is a critic and everyone is your friend.”

ments, one of them negated.

Hint 161 This looks like the conjunction of two quantified state-AT a **asA** **g2l** **rewsua****Exercise 162** Translog “Some state has no neighboring states.”**Hint 162** Use a unary and a binary predicate.AT a **asA** **g2l** **rewsua**

Exercise 163 Translog “Whenever there is an error, at least one error message is displayed.”

Hint 163 There are two different domains at play here.

$\exists x E(x) \rightarrow \exists y M(y) \vee D(y)$.
 이전에 “ $E(x)$ ”는 어떤 메시지를 표시하는지를 “ $M(y)$ ”는 “ $D(y)$ ”는 “ y ”가 오류인지를 표시하는지를 정의합니다.

Exercise 164 Translog “All the programs have been scanned, but at least one program has a virus.”

Hint 164 Use three predicates with programs as their domain.

AT A FILE TRANSFER

Exercise 165 Express the statement “All horses have hooves” using quantifiers; then form the negation of the statement so that no \neg is to the left of a quantifier; then express the negation in simple English (“quaneganglify” for all three tasks).

Hint 165 Use the predicates $H(x)$ = “ x is a horse”, and $L(x)$ = “ x has hooves”.

“ $\forall x H(x) \rightarrow L(x)$ ”는 모든 말이 “ $L(x)$ ”를 만족하는지 여부를 표시합니다.
ANSWER $\forall x H(x) \rightarrow L(x)$

Exercise 166 Quaneganglify “No horse can fly.”

Hint 166 Use the predicates $H(x)$ = “ x is a horse”, and $F(x)$ = “ x can fly”.

AT A FILE TRANSFER

Exercise 167 Quaneganglify “Every bat is blind.”

Hint 167 This is like Exercise 165.

„*Bear* *x* *can* *dance*(*x*)” \rightarrow *At least one bear can dance.*

Exercise 168 *Quaneganglify “No bear can dance.”*

Hint 168 Use *B* for Bear and *D* for Dance (as in “can dance”).

AT least one bear can dance.

Exercise 169 *Quaneganglify “There is at least one penguin that can swim and catch fish.”*

Hint 169 Use three predicates with one quantifier

„*fish-eater* *x* *can* *swim*(*x*) \wedge *can* *catch*(*x*)” \rightarrow *At least one fish-eater can swim and catch fish.*

Exercise 170 The notation $\exists!x P(x)$ means “There exists a unique *x* such that *P(x)* is true.” True or false? $\exists!x \in \mathbb{Z} (x > 1)$.

Hint 170 How many numbers are greater than 1?

At least one number is greater than 1.

Exercise 171 True or false? $\exists!x \in \mathbb{Z} (x^2 = 1)$.

Hint 171 The definition is in Exercise 170 if you missed it.

1-bar 1: out and about. Hint: $\exists!x$ $\exists!y$ $x = y$.

Exercise 172 True or false? $\exists!x \in \mathbb{Z} (x + 3 = 2x)$.

Hint 172 Just do the algebra.

AT A GLANCE

Exercise 173 True or false? $\exists!x \in \mathbb{Z} (x = x + 1)$.**Hint 173** Algebra won't help here, but logical thinking will.

ALGEBRA WORKS!

Exercise 174 True or false? $\exists x \in \mathbb{R} (x^3 = -9)$.**Hint 174** Algebra works!

AT A GLANCE

Exercise 175 True or false? $\exists x \in \mathbb{R} (x^4 > x^2)$.**Hint 175** Take advantage of the fact that both sides of the inequality

are positive (or zero).

ALGEBRA WORKS!

Exercise 176 True or false? $\forall x \in \mathbb{R} ((-x)^2 = x^2)$.**Hint 176** A negative times a negative equals a what?

AT A GLANCE

Exercise 177 True or false? $\forall x \in \mathbb{R} (2x > x)$.**Hint 177** Be careful here.

ALGEBRA WORKS!

Exercise 178 True or false? $\exists!x P(x) \rightarrow \exists x P(x)$.

Hint 178 Does “exactly one” imply “at least one”?

AT A GLANCE

Exercise 179 True or false? $\forall x P(x) \rightarrow \exists!x P(x)$.

Hint 179 Think about small universes — as in really small.

If the statement is false,
the statement is true if there are more than one thing in
a universe that has exactly one thing in it.
AT A GLANCE

Exercise 180 True or false? $\exists!x \neg P(x) \rightarrow \neg \forall x P(x)$.

Hint 180 Don't get tied up in notes.

AT A GLANCE

Exercise 181 True or false? $\forall x P(x) \rightarrow \exists x P(x)$.

Hint 181 Study the answer to Exercise 179 and the code in Figure 3.26.

If the statement is true, it is always true if there are more than one thing in a universe that has exactly one thing in it.
AT A GLANCE

Exercise 182 Let $E(x)$ be the statement “ x is an excuse”, $I(x)$ be the statement “ x is ignored”, $L(x)$ be the statement “ x is a lie”, and $R(x)$ be the statement “ x is a reason”, and let the domain for x consist of all English words. Express each of these statements using quantifiers, logical connectives, and $E(x)$, $L(x)$, and $R(x)$.

1. Some excuses are ignored.
2. Not all lies are reasons.
3. Some excuses are not reasons.

4. Some excuses are lies.

Hint 182 This should be getting easier

AT A GLANCE

Exercise 183 Let $T(x)$, $F(x)$, $R(x)$, and $D(x)$ be the statements “ x is a saint”, “ x is one of my friends”, “ x is a soldier”, and “ x is willing to drink”, respectively. Express each of these statements using quantifiers, logical connectives, and $T(x)$, $F(x)$, $R(x)$, and $D(x)$.

1. No saint is willing to drink.
2. No soldiers ever decline a drink.
3. All my friends are saints.
4. My friends are not soldiers.
5. Does 4 follow logically from 1, 2 and 3? If not, is there a correct conclusion?

Hint 183 This is easier than it looks.

- ρλ δ' σο μο σολφίειν εαιν ρε α μηνηδ οή μηνε.
τωιλλημαδ το φηνηδ ρλ ι αυηδ δ' αυηδ σολφίειεισ αιει τωιλλημαδ το φηνηδ
2. Αερ' ή ισθιειαγλ λογγομσ λιονισ ι' δ αυηδ δ'. Μο μηνηδ οή μηνε ιι
4. $\forall x E(x) \rightarrow \neg B(x)$
3. $\forall x E(x) \rightarrow L(x)$
5. $\forall x B(x) \rightarrow D(x)$
- ANSWER 183** 1. $\forall x L(x) \rightarrow \neg D(x)$

Exercise 184 Consider the two functions shown in Figure 3.26.

The function `forall` operationalizes the Universal Quantification of $P(x)$, the proposition that is true if and only if $P(x)$ is true for all x in S . `forall` loops through each value in a set S of finite size to see if the predicate P is always true. If it encounters a value for which P is false, then it short-circuits the loop and returns False. Otherwise it finishes the loop and returns True.

The function `forsome` operationalizes the Existential Quantification of $P(x)$, the proposition that is true if and only if there exists an x in the set S such that $P(x)$ is true. `forsome` loops through each value in a set S of finite size to see if the predicate P is ever true. If it encounters any value for which P is true, then it short-circuits the loop and returns True. Otherwise it finishes the loop and returns False.

```

def forall(P, S):
    for x in S:
        if not P(x):
            return False
    return True

def forsome(P, S):
    for x in S:
        if P(x):
            return True
    return False

even = lambda x: x % 2 == 0
odd = lambda x: x % 2 == 1

all_evens = [2, 4, 6, 8, 10]
not_all_evens = [2, 4, 5, 8, 10]
all_odds = [1, 3, 5, 7, 9]
not_all_odds = [1, 3, 6, 7, 9]

print(forall(odd, not_all_odds))
print(forall(odd, all_odds))
print(forall(even, not_all_evens))
print(forall(even, all_evens))

print(forsome(odd, not_all_evens))
print(forsome(odd, all_evens))
print(forsome(even, not_all_odds))
print(forsome(even, all_odds))

```

Find and explore the builtin Python functional (loop-free) equivalents of these two functions.

Hint 184 Test your findings with the even and odd predicates, plus some you find or create.

AT A GLANCE

3.14 Free and Bound

Today's session was feeling to Abu like it would be the culmination of a long exploration into the many facets of functionology. Ila was acting a little antsy, so they were both glad when Til looked up from what he was doing, and, as if taken by surprise, hurriedly acknowledged they were there.

Then, without so much as a hi-how-are-ya, Til said, “Before we leave predicates and quantifiers, let's talk about two related, very important concepts. To turn a first-order logic formula into a proposition, variables must be **bound** either by

1. assigning them a value, or
2. quantifying them.

As in real life, *freedom* versus *bondage* applies to math!“

Abu was in awe of the intensity with which Til spoke that last sentence. Ila was merely curious. “Could you please explain what you mean by freedom versus bondage?” she said.

“Certainly!” said Til. “In formulas, variables can be *bound* or they can be *free*. Bondage means being under the control of a particular value or quantifier. Say if ϕ is some logical expression and x is a variable in ϕ , then in both $\forall x \phi$ and $\exists x \phi$ the variable x is *bound* — a variable in ϕ that is neither quantified nor assigned a value is *free*.“

Til could see this explanation was less than satisfying. “Let me say one more thing and then we'll look at an example. As is the case in propositional logic,²³ some predicate formulas are true, others are false. It really depends on how the you interpret the predicates and functions. But if a formula has *any* free variables, in general, you *cannot* determine its truth under *any interpretation*. For example, let L be interpreted as *less-than*. Whether the formula $\forall x L(x, y)$ is true or not, we can't say. The presence of the free variable y prevents that. But insert an $\exists y$ right after the $\forall x$, thus binding y , and that, my twotees, makes it true!”

Abu thought it fascinating that you could nest one quantifier inside another like that. And that's exactly what Til called it:

“Finally — *this is important* — when nesting *same-flavored* quantifiers, the order in which they appear does not matter — but if they're different — then it does matter. Here's a quick example using the 2-

Figure 3.26: Defining and testing the `forall` and `forsome` functions. Prints:
 False
 True
 False
 True
 True
 False
 True
 False

²³ AKA the *propositional* (or *predicate*) *calculus*.

argument predicate $Q(x, y)$ asking if $x + y = x - y$:

- $\forall x \forall y Q(x, y)$ is false;
- $\forall x \exists y Q(x, y)$ is true;
- $\exists x \forall y Q(x, y)$ is false; and
- $\exists x \exists y Q(x, y)$ is true.

“Convince yourselves of these facts — and that’s all for now!”

Exercise 185 Write the formal definitions of injective and surjective given in Exercise 112 using quantifier symbols instead of words.

Hint 185 $A_m, n \in A$ is short for $A_m \in A \wedge A_n \in A$.

Ω.
माना यह कि $f : A \rightarrow B$ एक फलन है जिसके लिए $a_1, a_2 \in A$ के लिए $f(a_1) = f(a_2)$ का अर्थ है कि $a_1 \neq a_2$. तो यह कि f एक इनजेक्टिव फलन है।

Exercise 186 Translate these statements into English, where the domain for each variable consists of all real numbers.

1. $\exists x \forall y (x > y)$.
2. $\exists x \exists y (((x \geq 0) \wedge (y \geq 0)) \rightarrow (xy \geq 0))$.
3. $\exists x \forall y \exists z (x = y + z)$.

Hint 186 It might help to plug in some actual numbers (“concrete”).

AT आपका उत्तर:

Exercise 187 Let $Q(x, y)$ be the statement “ x asks y a question”, where the domain for both x and y consists of all students in a study group. Express each nested quantification as an English sentence.

1. $\forall x \exists y Q(x, y)$.
2. $\forall x \forall y Q(x, y)$.
3. $\exists x \exists y Q(x, y)$.
4. $\exists x \forall y Q(x, y)$.
5. $\forall y \exists x Q(x, y)$.
6. $\exists y \exists x Q(x, y)$.

Hint 187 Use, every, and some (or, at least one).

say true answers to 3.)

6. Some student asks some student a question. (Another may to student)

7. Every student is asked a question by some (or at least one)

8. Some student asks every student a question.

9. Some student asks at least one student a question.

10. Every student asks every student a question.

ANSWER 1. **NOT**

Exercise 188 Let $Q(x, y)$ be “ x asks y a question”, where the domain for both x and y consists of all people at your school. Express each of these sentences in terms of $Q(x, y)$, quantifiers, and logical connectives. Use the predicates $S(x) = “x \text{ is a student}”$, $T(x) = “x \text{ is a teacher}”$, and $A(x) = “x \text{ is a TA}”$ to distinguish different roles for people.

1. No student has ever asked a teacher a question.
2. There is a student who has asked a teacher a question.
3. Every student has asked a teacher and a TA a question.
4. At least two students have asked a teacher a question.

Hint 188 The last two need three quantifiers.

ANSWER

Exercise 189 Let $T(x, y)$ be “ x has taught y ”, where the domain for x is all teachers teaching in some department, and the domain for y is all classes taught in that department. Express each of these sentences in terms of $T(x, y)$, quantifiers, and logical connectives.

1. No teacher has taught every class.
2. Every teacher has taught every class.
3. Every teacher has taught some class.
4. At least two teachers have taught a class.

variables.

Hint 189 The last one needs three quantifiers binding three different

. $\forall x \forall y T(s, y) \wedge \exists z Q(x, z)$

. $\exists z \forall x T(s, x) \wedge Q(z)$

. $\exists z \forall x T(x, z) \wedge Q(z)$

. $\forall x \forall y T(x, y) \wedge \exists z Q(z)$

ANSWER 1. **NOT**

Exercise 190 Use quantifiers and predicates, some with two variables, to express these statements and their negations.

1. Every student needs a laptop.
2. No student has been in every building on campus.
3. Exactly one student has been in every room of exactly one building on campus.
4. Every student has been in at least one room of every building on campus.

Hint 190 Use the predicates $S(x)$ = “ x is a student”, $N(x, y)$ = “ x needs y ”, $B(x, y)$ = “ x is a building on campus”, $R(x, y)$ = “ x is a room in building y ”, $I(x, y)$ = “ x has been in y ”.

AT A FSA OEL TROWSSA

Exercise 191 Express each of these statements using predicates, quantifiers, logical connectives, and mathematical operators where the domain consists of all integers.

1. The average of two numbers is not greater than those two numbers.
2. The product of a positive number and a negative number is negative.
3. There is no solution in radicals to a quintic equation.
4. All positive numbers are greater than all negative numbers.

Hint 191 Both the second one and the fourth one need an implication. All involves just one after applying De Morgan’s rule for negating quantifiers. All involve two universal quantifiers except the third one, which involves just one after applying De Morgan’s rule for negating quantified statements.

$\forall x \forall y (((x > 0) \vee (y < 0)) \rightarrow x > y)$.
 $\forall x \forall y \forall z ((x > 0) \wedge (y < 0) \wedge (z > 0) \rightarrow x + y < z)$.
 $\forall x \forall y \forall z ((x > 0) \wedge (y < 0) \wedge (z > 0) \rightarrow xy < 0)$.
 $\forall x \forall y \forall z ((x > 0) \wedge (y < 0) \wedge (z > 0) \rightarrow xz < yz)$.

TEL TROWSSA

Exercise 192 Express the statement “no prime number has 3 factors”.

tors” using predicates, quantifiers, logical connectives, and mathematical operators.

Hint 192 This needs four universal quantifiers binding four different variables, and it definitely calls for an application of the *germany principle*.

AT A FSA 461 TAWSA

Exercise 193 If the domain for x and y is \mathbb{Z} , what are the truth values of these statements?

1. $\forall x \forall y ((x \neq 0) \wedge (y \neq 0) \leftrightarrow (xy \neq 0))$.
2. $\exists x \forall y (x + y > y)$.
3. $\exists x \forall y (x + y < y)$.
4. $\exists !x \forall y (x + y = y)$.

Hint 193 Recall that \mathbb{Z} is the set of all integers.

AT A FSA 461 TAWSA

Exercise 194 What are the truth values of the expressions below if $T(x, y)$ is “ $xy < y$ ”, where the domain for both x and y is \mathbb{Z} ?

1. $T(0, 5)$.
2. $T(1, 1)$.
3. $T(7, -3)$.
4. $\exists x \forall y T(x, y)$.
5. $\exists x T(x, 2)$.
6. $\forall y T(9, y)$.
7. $\forall x \forall y T(x, y)$.
8. $\forall x \exists y T(x, y)$.

Hint 194 Write some code that implements the T predicate and tests it.

AT A FSA 461 TAWSA

Exercise 195 Determine the truth value of the statement

$$\forall x \exists y (x - y \geq x)$$

if the domain for x and y is

1. the real numbers.
2. the natural numbers.
3. the negative integers.
4. the rational numbers.

Hint 195 Plug in concrete numbers if necessary.

ANSWER 192 The answers are the same as in Exercise 183.

Exercise 196 Expand to two nested quantifiers the code you explored for one quantifier in Exercise 184. Specifically, implement these four functions, shown side-by-side with their symbolic logic equivalents:

Function Name	In Symbols
forallforall	$\forall x \forall y$
forallforsome	$\forall x \exists y$
forsomeforall	$\exists x \forall y$
forsomeforsome	$\exists x \exists y$

Look at these quantifications in the context of loops and a generic predicate P . These will be nested loops, an outer one wrapping an inner one, because nested quantification is what is being expressed. The predicate (function) call $P(x, y)$ goes in the inner loop, which controls the y , while the outer loop controls the x .

“For all x for all y ” wants to find $P(x, y)$ always true. That’s what it means for the nested quantification to be true, and naturally, this only works if the domains of x and y are finite. Even then, it really only works if these domains are reasonably finite — not too big. Iteration is serial, after all, and time is limited.

So, `forallforall` loops through x ’s domain, and for each x loops through each y in y ’s domain. On each inner-loop iteration it calls $P(x, y)$ and checks the value. If the value is ever false, `forallforall` is false — immediately — no need to check any further. Some y has been found for some x where the predicate is false. If both loops finish with no false evaluation, `forallforall` is ultimately true. There is no x for which, for any y , $P(x, y)$ is false.

The other function with relatively simple logic is `forsomeforsome`. This function loops through x ’s domain, and for each x loops through each y in y ’s domain. On each inner-loop iteration it calls $P(x, y)$ and checks the value. If a true value is found, then `forsomeforsome` is true — immediately — no need to check any further. If both loops finish never having triggered true, `forsomeforsome` is ultimately false. There is no x for which there is some y for which $P(x, y)$ is true.

The other two are trickier: “for all x for some y ” wants $P(x, y)$ to

always be true sometimes, and “for some x for all y ” wants $P(x,y)$ to sometimes be true always.

Implement forallforsome and forsomeforall the best, most elegant way you can. Test your implementations of all four functions using suitable binary (2-ary) predicates, for example ‘>’, and their associated domains.

Hint 196 You will profit by preferring mapping to looping!

AT A GLANCE

Exercise 197 Another highly useful higher-order function is filter, which takes a predicate and a sequence and creates a new sequence filtered through the predicate. Items in the original sequence for which the predicate returns true remain, if false they are filtered out.

For example:

```
print(list(filter(lambda x: x % 2 == 1, range(1, 7))))
```

prints out a list of the first three odds.

Write a function called summer (that has nothing to do with sums or series) and make it so the summer months (June, July, August) are kept and the others discarded when passing summer to filter with the list ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December'].

Hint 197 Make the filter code as efficient as possible.

```
def summer(months):    return months[j] == 'J',  
as first second letter:  
AT A GLANCE  Explain the fact that the summer months off have a J,
```

Exercise 198 In the spirit of nested quantifiers, define a “nested filter” creator function. This is a function that takes two parameters, m and r , and returns a lambda filter with parameter x and expression $(x \% m) == r$. Call this function with three different pairs of values for m and r to create three filters. Apply these filters to the range(0, 100) and print lists of the filtered results. Where have you seen this type of list before?

Hint 198 This foreshadows some number theoretic concepts coming later, but introduced in the next chapter with the TLA CMM.

AT A GLANCE FOR ANSWERS

```
def test_predicate(predicate):
    return forall(predicate, [[ 'Discrete', 8], ['Math', 8],
                             ['Is', 6], ['Awesome', 28]]) and \
           not forall(predicate, [[ 'Oracle', 0], ['Apple', 11],
                                  ['Microsoft', -18], ['Google', 25]])

def my_predicate(data):
    # YOUR ANSWER GOES HERE
    raise NotImplementedError

print(test_predicate(my_predicate))
```

Figure 3.27: A Predicate Test For All Certain Somethings and Not For Others

Exercise 199 Refer to Figure 3.27. Write a predicate for which the call to `test_predicate` returns True.

Hint 199 This will require some thought. But it's simpler than it looks. Put your code (a one-liner) in the place of the line that says "YOUR ANSWER GOES HERE". After doing so, you can remove the `raise NotImplementedError` line, or leave it, it won't matter:

YOUR ANSWER GOES HERE

Exercise 200 The `fn_decrypt` function, defined in and referencing the `first_pull` and `second_pull` poems in Figure 3.28, takes a list and returns one of two different functions, each with the same codomain, depending on if a certain function returns true for all elements of the input. Your task is to unscramble a secret message in a list of lists of numbers by first transforming the data, then mapping the data through the respective results of the `fn_decrypt` function.

```
first_pull = """BECAUSE I could not stop for Death,
He kindly stopped for me;
The carriage held but just ourselves
And Immortality.
```

We slowly drove, he knew no haste,
And I had put away
My labor, and my leisure too,
For his civility.

We passed the school where children played
At wrestling in a ring;
We passed the fields of gazing grain,
We passed the setting sun.

We paused before a house that seemed
A swelling of the ground;
The roof was scarcely visible,
The cornice but a mound.

Since then 't is centuries; but each
Feels shorter than the day
I first surmised the horses' heads
Were toward eternity."""" # Public domain. From <https://www.bartleby.com/113/4027.html>

```
second_pull = """Tiger, tiger, burning bright
In the forests of the night,
What immortal hand or eye
Could frame thy fearful symmetry?
```

In what distant deeps or skies
Burnt the fire of thine eyes?
On what wings dare he aspire?
What the hand dare seize the fire?

And what shoulder and what art
Could twist the sinews of thy heart?
And, when thy heart began to beat,
What dread hand and what dread feet?

What the hammer? what the chain?
In what furnace was thy brain?
What the anvil? what dread grasp
Dare its deadly terrors clasp?

When the stars threw down their spears,
And watered heaven with their tears,
Did He smile His work to see?
Did He who made the lamb make thee?

Tiger, tiger, burning bright
In the forests of the night,
What immortal hand or eye
Dare frame thy fearful symmetry?"""" # Public domain. From <http://www.gutenberg.org/files/1934/1934-h/1934-h.htm>

```

def fn_decrypt(data):
    if forall(lambda x: x % modulus == 0, data):
        return (lambda lst: first_pull[int(''.join(str(d) for d in lst), 8)])
    else:
        return (lambda lst: second_pull[int(''.join(str(d) for d in lst), 8)]))

print(decrypt_all([[3135, 7077, 7001],
[6585, 2700, 980], [5658, 3828, 412],
[2233, 5420, 8072, 2578],
[8085, 570, 4620],
[4689, 2892, 4032, 1537],
[2950, 4660, 4365],
[8161, 1658, 7843, 1432],
[5610, 3261, 2719], [6065, 1510, 2335],
[1353, 3272, 2195, 1986],
[5630, 4030], [4825, 200, 5205, 2925],
[4053, 3961], [4370, 6310],
[553, 5994, 660, 335],
[7155, 7075, 6480],
[4460, 8170], [20, 549, 7470],
[1853, 3346, 7071], [8017, 4692, 1573],
[3218, 5153, 2443],
[4585, 7480, 1755, 4535],
[7730, 5700, 465],
[4377, 1003, 871, 7611],
[1953, 5193, 2134], [6940, 3379, 1661],
[2385, 8144, 2566, 7781],
[7660, 6177, 745],
[4465, 4320, 7130, 405],
[4734, 68, 2470], [6481, 3050, 1526],
[5194, 2432, 6532], [310, 3730, 7220],
[6185, 3320, 3995, 7260],
[4515, 2165, 1575], [5033, 17, 6555, 125],
[5121, 1256, 7225, 350],
[5545, 3265, 1640, 5155],
[5657, 2928, 1054, 101],
[5385, 3425, 5785, 6900],
[8185, 6750, 2280], [7755, 1355, 7120],
[7145, 8075, 2040], [7210, 6145, 7955],
[1430, 1244], [3941, 1170, 6808],
[7990, 4930, 2845], [1655, 2745, 3365],
[1829, 1868, 7825], [4065, 4850, 4780],
[5882, 6010, 8015], [1750, 2215, 4115],
[1310, 7591, 2642], [3361, 1708],
[1731, 5007, 54], [2011, 706, 2711],
[3958, 3876, 2455], [465, 4808, 7274, 5006],
[7393, 4184, 295, 4629], [7980, 7785, 4120],
[3070, 6310, 1740], [4449, 3829],
[5675, 4080, 2220], [7883, 3983, 7626],
[5780, 260, 7720], [86, 499],
[5609, 1187, 8166, 4187], [1790, 1670, 5860]]))

```

Figure 3.28: Decrypt with Poetry

[See Book Cipher for more ideas.](#)

*positive integer less than 8.)
figure out what some modulus is (modulus in fn-decrypt) — some
as simple as $x \leftarrow \text{map}(y \leftarrow \% \text{some_modulus}, x)$. You'll need to
if you make the function required to transform the data somehow
Hint 200 This idea may sound very hard, but it won't be too difficult*

AT A GLANCE

3.15 Summary of Terms and Definitions

A **function** is just an object that takes objects and gives other objects.

Functions have many **representations**, including:

- assignment
- association
- mapping (or map)
- machine
- process
- rule
- formula
- equation
- table
- graph
- code

The inputs to a function are its **arguments** (AKA **parameters**).

A function's **arity** is the number of input arguments it takes.

A **k-ary function** is a function with **k** inputs.

A **unary function** is a function with 1 argument.

A **binary function** is a function with 2 arguments.

A **ternary function** is a function with 3 arguments.

The notation called **infix notation** describes a *binary* function that is written as a symbol placed *between* its two arguments (rather than before or after).

The notation called **prefix notation** describes a *k-ary* function that is written as a symbol placed *before* its arguments, frequently having parentheses surrounding the arguments.

The notation called **postfix notation** describes a function that is written as a symbol placed *after* its arguments.

Image describes the output of a function applied to some input (which in turn is the **preimage** of the output).

Domain is the **set** of all *possible* inputs for a function.

Codomain is the set of all *possible* outputs for a function.

Range is the set of all *actual* outputs for a function.

Onto describes a function whose range is the same as its codomain, that is, an **onto** function produces (for the right inputs) all possible outputs.

Surjective is a synonym for onto.

Surjection is a noun naming a surjective function.

One-to-one describes a function each of whose outputs is generated by only one input.

Injective is a synonym for one-to-one.

Injection is a noun naming an injective function.

Bijective describes a function that is both one-to-one and onto.

Bijection is a noun naming a bijective function.

Recursive describes a function that calls itself.

A **predicate** is a function whose codomain is the set $\{true, false\}$.

Property is another word for predicate.

Decision-maker is another way to say predicate.

Decider is short for decision-maker.

The **floor** of a real number, $\lfloor x \rfloor$, is the *largest integer* less than or equal to that number. AKA the “round down” function.

The **ceiling** of a real number, $\lceil x \rceil$, is the *smallest integer* greater than or equal to that number. AKA the “round up” function.

The **inverse** of a function $f : A \rightarrow B$, denoted f^{-1} , is a function from B to A such that $\forall a \in A, \forall b \in B$, if $f(a) = b$, then $f^{-1}(b) = a$.

The **composition** of two functions f and g , denoted $f \circ g$, is the operation of applying f to the result of applying g to something. That is, $(f \circ g)(x)$ is $f(g(x))$. To form the composition (to compose) f and g , they must be **composable** (be compatible), meaning the codomain of g must be the same as (or a subset of) the domain of f .

A finite **sequence** of elements of a set S is a function from the set $\{1, 2, 3, \dots, n\}$ to S — the number n is the **length** of the sequence. Alternatively, the domain of the sequence function can be $\{0, 1, 2, \dots, n - 1\}$.

A **string** is another term for a finite sequence.

An **arithmetic progression** is a sequence of the form $a, a + d, a + 2d, a + 3d, \dots$. In other words, the sequence starts with a , and the common *difference* between the next term and the current term is d .

A **geometric progression** is a sequence of the form $a, ar, ar^2, ar^3, ar^4, \dots$, i.e., the sequence starts with a , and r is the common *ratio* between the next term and the current term.

A **summation** (or **series**), denoted by the Greek letter Σ , collapses a sum of many related terms into a single term parameterized by a variable (or variables) called the **index** (**indices** plural) of summation that ranges from a lowest to a highest value. With i as the index of summation ranging from 1 to n ,

$$f(a_1) + f(a_2) + \cdots + f(a_n)$$

collapses to

$$\sum_{i=1}^n f(a_i).$$

The **sum of a bounded arithmetic series rule** says

$$\sum_{i=l}^u a + i \cdot d = (u - l + 1)(a + d(u + l)/2).$$

The **sum of the first n positive integers** is

$$\sum_{i=1}^n i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}.$$

The **partial sums of the first n positive integers** form the **triangular numbers**.

A Σ in the case where the upper bound is ∞ is called an **infinite series**, and can either **converge** (produce a finite number) or **diverge** (grow *infinitely large*, or grow *without bound*).

The **convergent sum of a geometric series rule** says

$$\sum_{i=0}^{\infty} ar^i = \frac{a}{1-r}, 0 < r < 1.$$

The **distributivity of multiplication over addition rule** says

$$c \cdot a_1 + c \cdot a_2 + \cdots + c \cdot a_n = \sum_{i=1}^n c \cdot a_i = c \cdot \sum_{i=1}^n a_i.$$

The **sums or differences of two terms are sums of the first term plus or minus sums of the second term rule** says

$$\sum_{i=1}^n (a_i \pm b_i) = \sum_{i=1}^n a_i \pm \sum_{i=1}^n b_i.$$

The **sums can be split rule** says

$$\sum_{i=a}^z s_i = \sum_{i=a}^m s_i + \sum_{i=m+1}^z s_i, \text{ where } a < m < z.$$

The **nested summation interchange rule** says

$$\sum_{j=1}^n \sum_{k=1}^j f(j, k) = \sum_{k=1}^n \sum_{j=k}^n f(j, k).$$

The **harmonic series** is the infinite sum of the reciprocals of the positive integers. Its finite approximation is $H_n = \sum_{i=1}^n \frac{1}{i}$ for any given (positive integer) value of n .

A **universe of discourse** (AKA **domain of discourse**) is the set of all things under consideration.

The **universal quantifier** (\forall) says something is true for *all* members of the universe of discourse.

The **existential quantifier** (\exists) says something is true for *some* member (i.e., at least one) of the universe of discourse.

In formulas, variables can be **bound** or **free**. Being assigned a value is one way to be bound, the other way is being under the control of a quantifier. A variable that is neither quantified nor assigned is free.

4

Relations

4.1 Products and Subsets

“Welcome back!” said Til, adjusting his chair while Abu and Ila slipped easily into theirs. “I hope you’ve enjoyed —” he paused, and for a moment it seemed as if he were about to engage in a tiny bit of small talk — but instead Til resumed with “— your many excursions into function land. Today we’re going to further explore this terrain, starting with the fact that functions have been generalized to something called *relations* — which are essentially functions, but with one big difference — there is no *fan-out* constraint on relations as there is on functions.”

“Excuse me,” said Ila. “What does that mean — ‘no fan-out constraint?’” Abu raised his eyebrows at Til to signal that he too craved an explanation.

“What that means,” Til said, “is that an element of the domain of a relation can map to *any number of elements* — zero, one, or more — in the relation’s codomain. Recall that a function’s mapping’s fan-out is *exactly one*.¹”

Excited, Abu gestured as he said, “I’m seeing myself as an element pointing both hands away from me at 45° angles. My fingers are spread like a fan too, so it’s like I’m pointing — fanning out — to 10 different things — elements in a codomain. Is that like what a relation does, or can do?”

“Yes, good analogy!” said Til. Ila joined in, “And if Abu were pointing with one hand’s index finger to just one thing, that would be like a function, right?”

“Exactly,” said Til, “with the possibility that some other element or elements in the domain could be pointing at the same thing — a multiple *fan-in*¹ from that thing’s point of view.”

Abu and Ila looked at Til with eager anticipation. Til teased them by asking and then answering his own question, “So how do we mathematically model this generalized kind of many-to-many mapping? First, we need a definition: $A \times B$, the **Cartesian product** (or just product) of set

¹ Thus disqualifying the function as an injection.

A with set B is the set of all ordered pairs (2-tuples) (a, b) where $a \in A$ and $b \in B$.² For example, the product of $\{a, b, c\}$ and $\{1, 2, 3\}$ is the set of 2-tuples $\{(a, 1), (a, 2), (a, 3), (b, 1), (b, 2), (b, 3), (c, 1), (c, 2), (c, 3)\}$. We need this definition because we define a relation in terms of it.”

Til displayed some code and then said, “Before I give you the definition of relation, let’s look at a three-set $A \times A \times A$ example. Take our favorite `personality3` function and reimplement it using the Python `product` function.³ So rather than hardwiring those eight `inputs3` 3-tuples we generate them by passing `[0, 1]` assigned to `bits` as the same three arguments to `product`, as in Figure 4.1.”

```
from itertools import product
from operator import concat
from functools import reduce

def personality3(boolfunc3):
    bits = [0, 1]
    inputs3 = list(product(bits, bits, bits))
    # thus inputs3 are generated instead of hardwired as
    # [(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
    outputs = map(boolfunc3, map(lambda x:x[0], inputs3), map(lambda x:x[1], inputs3), map(lambda x:x[2], inputs3))
    return reduce(concat, map(str, outputs))
```

Figure 4.1: The `personality3` function with the `inputs3` 3-tuples created by taking the Cartesian product of `[0, 1]` by itself 3 times. Then by calling `list` on that result, a list is created of those 3-tuples, where each 3-tuple is suitable for passing as the three arguments to `boolfunc3`.

“That’s so cool!” said Ila, and after he studied the code for a moment, Abu said, “I can almost see the `product` function doing that triply-nested loop thing — under the hood.” “More likely some maps happening,” said Ila, surprised but pleased by her growing independence from loopiness. “Well,” said Til, “we know what we call finding out exactly how Python does it, don’t we!”

“A GPAO!” cried Abu and Ila in unison.

Til smiled, and said, “Let’s look at that definition now — a **binary relation** R from a set A to a set B is a *subset* of the product of A and B . So $R \subseteq A \times B$. We can have B be the same set as A , in which case it’s called a **self relation**, so $R \subseteq A \times A$. For an $A \times B$ example, take the sets I used in my first example of a Cartesian product, giving them names:

- $A = \{a, b, c\}$
- $B = \{1, 2, 3\}$
- $R = \{(a, 1), (a, 2), (a, 3), (b, 1), (b, 2), (b, 3), (c, 1), (c, 2), (c, 3)\}$.

And for a self relation ($A \times A$) example:

- $A = B = \{1, 2, 3\}$
- $R = \{(1, 1), (1, 2), (2, 3)\}$.

And these are just abstract relations, of course.”

4.2 Four Main Properties

Abu and Ila were hungry for more — which hunger only deepened their realization that what they learned about universal quantifiers and the logical conditional was about to be put to the test.

“There are many special properties of binary relations,” said Til, “but we’re going to limit our focus to four for now. I’ll give you their definitions after introducing a shorthand notation, and then have you practice recognizing whether or not these definitions fit a few small abstract binary relations. So, given:

- a Universe U ;
- a binary relation R on U (or a subset of U);
- xRy short for ‘ x is related to y ';⁴

R is

- **reflexive** when $\forall x x \in U \rightarrow xRx$;
- **symmetric** when $\forall x \forall y xRy \rightarrow yRx$;
- **antisymmetric** when $\forall x \forall y xRy \wedge yRx \rightarrow x = y$; and
- **transitive** when $\forall x \forall y \forall z xRy \wedge yRz \rightarrow xRz$;

and before I turn you loose to study these definitions on your own for a while, please note that symmetry and antisymmetry are *not* mutually exclusive properties. They can either both be absent, both be present, or one absent and the other present.”

Cautiously nodding their assent, Abu and Ila were in reality puzzled by this not-opposite meaning of *anti*.

Til noticed their puzzled looks but made no comment. “So,” he said, “here are the small abstract examples of binary relations over the set $A = \{1, 2, 3, 4\}$ to have a go with classifying:”

- $R_1 = \{(1, 1)\}$
- $R_2 = \{(1, 2), (2, 3), (3, 2)\}$
- $R_3 = \{(1, 3), (3, 2), (2, 1)\}$
- $R_4 = \{(1, 4), (2, 3)\}$

Abu and Ila studied these four relations, consulted together for several minutes, and then gave Til their answers.⁵ Til was pleased to confirm that their reasoning⁶ was 100% correct, exactly matching his key:

Relation	Reflexive?	Symmetric?	Antisymmetric?	Transitive?
R_1	No	Yes	Yes	Yes
R_2	No	No	No	No
R_3	No	No	Yes	No
R_4	No	No	Yes	Yes

“Good work, you two!” enthused Til. “Now I have a puzzle for you. Find the connection between the Python dictionary $\{1:7, 2:0, 3:2, 4:3\}$ and what you just did!”

“That’ll take a few minutes,” said Abu. Ila nodded her agreement.

⁴ Shorter than saying “the pair (x, y) is in the relation R ” (in symbols, $(x, y) \in R$), and shorter too than “ x is related to y (under the relation R).” The “(under the relation R)” is especially verbose, and when that is the clear context, the succinctness of xRy is compelling.

⁵ They got everything right. Can you?! Please try before looking at the explanations below.

⁶ R_1 is not reflexive because it is missing $(2, 2)$, $(3, 3)$ and $(4, 4)$. It is symmetric because, for all x and for all y (all one of each of them), $xRy \rightarrow yRx$ ($1R1 \rightarrow 1R1$). It is antisymmetric because $1R1 \wedge 1R1 \rightarrow 1 = 1$. It is transitive because $1R1 \wedge 1R1 \rightarrow 1R1$ (1 is the only value x , y and z can be).

R_2 is not reflexive because it is missing all four: $(1, 1), (2, 2), (3, 3)$, and $(4, 4)$. It is not symmetric because it lacks $(2, 1)$, which is the symmetric partner of $(1, 2)$. It is not antisymmetric because $2R3$ and $3R2$ do *not* imply that $2 = 3$. It is not transitive because it lacks $(1, 3)$, needed to satisfy $1R2 \wedge 2R3 \rightarrow 1R3$. It also fails to be transitive for the lack of $(2, 2)$ and $(3, 3)$, needed to satisfy $2R3 \wedge 3R2 \rightarrow 2R2$ and $3R2 \wedge 2R3 \rightarrow 3R3$.

R_3 is not reflexive because it lacks all four: $(1, 1), (2, 2), (3, 3)$ and $(4, 4)$. It is not symmetric because it lacks the symmetric partners of all three pairs. It is antisymmetric — vacuously so — because there are *no* symmetric pairs, and thus the antecedent of the conditional in the definition is always false, meaning the conditional is vacuously true. Recall § 2.11.)

R_3 is not transitive because it lacks $(1, 2)$, needed to satisfy $1R3 \wedge 3R2 \rightarrow 1R2$. (Ditto for the missing $(3, 1)$ and $(2, 3)$, needed to satisfy $3R2 \wedge 2R1 \rightarrow 3R1$ and $2R1 \wedge 1R3 \rightarrow 2R3$.)

R_4 is not reflexive because it lacks all four: $(1, 1), (2, 2), (3, 3)$ and $(4, 4)$. (Lacking any one of them, of course, would be enough to disqualify it.) It is not symmetric because it lacks the symmetric partners of both its pairs. Like R_3 , it is vacuously antisymmetric. For similar reasons, it is vacuously transitive. There are no x , y and z for which xRy and yRz are both true, hence xRz can never be false!

"Take your time," said Til.

With their gersy principle application experience to draw on, putting their heads together Abu and Ila solved the puzzle in just under two minutes.

"We have a theory — and we think it's correct," said Ila. "The dictionary keys are the relation numbers and the values are the binary encoding of the No's and Yes's for the four properties." Showing Til his notes, Abu said, "So R_1 being 'No' for reflexive but 'Yes' for the other three is 7 — 0 for No, 1 for Yes — 0111, or 7. R_2 being all No's is just zero, and so on."

“That is 100% correct! You two make a great team!” Til was happy to give them some well-deserved praise, but quickly continued speaking to make his point. “Now, other than explicitly listing the pairs in a relation there are other quite useful representations⁷ that this puzzle you just solved only hints at. Your exercises will make it clear. So keep studying these concepts, and see you next time!”

⁷ One useful representation is called a “zero-one matrix” (AKA *logical matrix* or connection matrix). This matrix representation allows easy discerning of most properties — 6 out of 7 — yes, there are more than 4 main properties — transitivity being the only property that is a little hard to detect. See Exercise 208 and its companions. Learning more about matrix math, and matrices in general, is, of course, a GPAO.

Exercise 201 From the definition of antisymmetric, logically argue that if xRy and $x \neq y$ then it must be false that yRx .

Hmit 201 The definition is a conditional, so use its contrapositive, which is equivalent.

Exercise 202 Which of the following pairs are in the relation on the set $\{1, 2, 3, 4\}$ given by the description $\{(x, y) \mid x > y + 1\}$?

1. (2, 1)
 2. (2, 2)
 3. (2, 3)
 4. (2, 4)
 5. (3, 1)
 6. (3, 2)
 7. (4, 1)
 8. (4, 2)

9. (4,3)

`lambda x, y: x < y + 1 in a nested loop to verify.`

Hint 202 This is easy to eyeball, but consider calling the function

AT a `for` `break` `else`

Exercise 203 The four properties of reflexivity, symmetry, antisymmetry, and transitivity, in that order, can be encoded as a string of four bits, then converted to a number in the range 0 to 15. Call this process the “profiling” of the relation. Call the resulting number the “profile” of the relation. For example, the profiles of R_1 , R_2 , R_3 , and R_4 are 7, 0, 2, and 3, respectively (converted from the “bitstrings” 0111, 0000, 0010, and 0011).

Profile the relation $\{(1,1),(1,3),(2,2),(3,1)\}$ over the set $\{1,2,3\}$.

Hint 203 This is a very straightforward exercise.

AT a `for` `break` `else`

Exercise 204 Profile the relation $\{(1,1),(2,2),(3,1),(3,3)\}$ over the set $\{1,2,3\}$.

Hint 204 This is a very straightforward exercise.

AT a `for` `break` `else`

Exercise 205 Profile the relation $\{(1,2),(2,1),(3,3)\}$ over the set $\{1,2,3\}$.

Hint 205 This is a very straightforward exercise.

AT a `for` `break` `else`

Exercise 206 Profile the relation $\{(1,3),(2,3)\}$ over the set $\{1,2,3\}$.

Hint 206 This is a very straightforward exercise.

AT a `for` `break` `else`

Exercise 207 How many possible binary relations are there on a set with 3 elements? What about a set with n elements? Write a Python expression as a function of n as your answer, and explain it.

Hint 207 How many subsets are there of a set of size s ?

5
ολα σετ από n ελέμεντας έχει n^2 τοποθεσίες. Η πλήθης: 2^n

Answer 207 Της απάντησης είναι 2^n λόγω της περιπτώσεως ότι η προσθήτη

Exercise 208 A relation R on $A = \{a_1, a_2, \dots, a_n\}$ can be represented by an $n \times n$ zero-one matrix M_R whose $(i, j)^{\text{th}}$ entry is 1 if $(a_i, a_j) \in R$ and 0 otherwise.

For example, R_1 defined as a binary self-relation over the set $\{a, b\}$ consisting of the pairs $\{(a, a), (a, b), (b, b)\}$ is represented by this 2×2 matrix:

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

Implemented by the Python list-of-lists `[[1, 1], [0, 1]]`, this matrix is collapsed into the $2^2 = 4$ bits 1, 1, 0, and 1 — expressed as the number 13 (1101 in binary) — and so encoded as the pair (13, 2).

For another example, R_2 defined over $\{a, b, c, d\}$ as $\{(a, a), (a, b), (b, b), (b, c), (c, c), (d, b), (d, d)\}$ has this 4×4 matrix:

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

Implemented by `[[1, 1, 0, 0], [0, 1, 1, 0], [0, 0, 1, 0], [0, 1, 0, 1]]`, this matrix is collapsed into the $4^2 = 16$ bits 1100011000100101, expressed as 50725, encoded as the pair (50725, 4).

Study the code below carefully — it is deliberately undocumented. Given this code and a relation R encoded as a pair consisting of a decimal number and the size n of R 's matrix M , write a Python function to do (“implement”) a reflexivity check, where R is reflexive if $M_{ii} = 1$ for all i . (The total number of bits in M is $n \times n = n^2$, as it is a square matrix with n rows and n columns.)

```
def int2bits_rec(num, a):
    if num:
        a.insert(0, num % 2)
        int2bits_rec(num // 2, a)
```

```

def int2bits(num, size):
    accum = []
    int2bits_rec(num, accum)
    return [0] * (size - len(accum)) + accum

def int2matrix(encoding, n):
    bits = int2bits(encoding, n * n)
    matrix = []
    start = 0
    for i in range(n):
        matrix.append(bits[start:start + n])
        start += n
    return matrix

def matrix2int_rec(bits, p):
    if not bits:
        return 0
    else:
        return (2 ** p if bits[0] else 0) + \
            matrix2int_rec(bits[1:], p + 1)

def matrix2int(matrix):
    flat = [item for row in matrix for item in row]
    flat.reverse()
    return matrix2int_rec(flat, 0)

for encoding in range(0, 16):
    matrix = int2matrix(encoding, 2)
    print(matrix, '=', matrix2int(matrix))

matrix = int2matrix(500, 3)
print(matrix, '=', matrix2int(matrix))

matrix = int2matrix(51351, 4)
print(matrix, '=', matrix2int(matrix))

```

In other words, M 's “main diagonal” is all 1's.

Hint 208 The condition to be met for reflexivity is $M_{ii} = 1$ for all i .

AT ڈا سا 208 تکہسا

Exercise 209 Implement an “IRreflexivity” check, where R is irreflexive if $M_{ii} = 0$ for all i .

Hint 209 This is like Exercise 208 with one minor difference.

```

return True
else:
    for i in range(n):
        if M[i][i] == 1:
            return False
    return True

```

AT ڈا سا 209 تکہسا

Exercise 210 Implement a “NONreflexivity” check, where R is non-reflexive if $M_{ii} = 1$ for some i ’s, 0 for others.

Hint 210 Combine implementations of reflexive and irreflexive.

AT A GLANCE

Exercise 211 Implement a symmetry check, where R is symmetric if $M = M^T$. The “transpose” of a matrix, M^T , is the matrix M with its rows and columns swapped. For example, $[[1, 2, 3], [4, 5, 6], [7, 8, 9]]$ transposed is $[[1, 4, 7], [2, 5, 8], [3, 6, 9]]$.

Hint 211 You don’t need it, but F^T , Python can transpose a matrix with one line of code using the `zip` function:

```
def is_symmetric(M):
    for i in range(len(M)):
        for j in range(len(M[0])):
            if M[i][j] != M[j][i]:
                return False
    return True
```

AT A GLANCE

Exercise 212 Implement an antisymmetry check, where R is anti-symmetric if $M_{ij} = 0$ or $M_{ji} = 0$ for all $i \neq j$. That is, there are never two 1’s symmetrically found across M ’s main diagonal.

Hint 212 Adapt your symmetry check.

AT A GLANCE

Exercise 213 Implement an “Asymmetry” check, where R is asymmetric if “ x is related to y ” implies “ y is NOT related to x ” for all $x, y \in A$. In other words, R is both antisymmetric and irreflexive.

Hint 213 Combine your antisymmetry and irreflexivity checks with the and operator.

թե՛ս աստիճանաբար (ըստօրինակ՝ և այլ դրայվերներում) առաջանակած է այսպիսի աշխատավորություն (ըստօրինակ՝ և այլ դրայվերներում):

ԱՐԴ ՔՐԵՎՏԱ

Exercise 214 Implement a transitivity check, which will require doing some research!

Hint 214 Not an explanation, but for your unpacking pleasure, M^2 . That is, there is a 1 in M at every location where there is a 1 in M^2 , the “Boolean product” of M with itself is “less than or equal to” is the zero-one matrix of a transitive relation if M ’s “Boolean square” M^2 .

ԱՐԴ ՔՐԵՎՏԱ

Exercise 215 To thoroughly test your seven properties’ checking functions, answer the following question three times — for $n = 2$, $n = 3$, and $n = 4$. What is the breakdown of the frequencies of all seven properties for every possible $n \times n$ matrix? Sort your tabulated results in descending order, i.e., the first row has the most-frequently-occurring property and the last row has the least-frequently-occurring property.

Hint 215 Organize your work, prepare every needful thing.

```

b1tuf(tfw[j], tfw[0])
for tfw in gfsorteq.tfw():
    b1tuf(, /n = , ' n)
        ksl=gfwqba tfw: tfw[j], LEVELS=TRUE}
gfsorteq = {ksl:vejue for ksl, vejue in sorteq(gfotmewtfewas()) /
gfotmew = fapufaf(u)
for n in lnsd(s, 2):

    refun gfsr
        gfsr[,frusfttve,] += frusfttve(m, n)
        gfsr[,gslamweftrc,] += gslamweftrc(m, n)
        gfsr[,gutgslamweftrc,] += gutgslamweftrc(m, n)
        gfsr[,slamweftrc,] += slamweftrc(m, n)
        gfsr[,monrelfexiv,] += monrelfexiv(m, n)
        gfsr[,tilrelfexiv,] += tilrelfexiv(m, n)
        gfsr[,relfexiv,] += relfexiv(m, n)
        m = tufswaftrix(gucoqtd, n)
    for enocqtd in lnsd(s ** n ** s):
        frusfttve,:0}
        gslamweftrc,:0', gutgslamweftrc,:0', gslamweftrc,:0'/
gfsr = {,relfexiv,:0', tilrelfexiv,:0', monrelfexiv,:0'/
def fapufaf(u):

couhffeq ouce
stous ol the hroberly cneqiuq huiuciuq thut fave the zero-one matrix
AUTOMATE This code assmues lon yau createq more effcienc uer-
```

Exercise 216 The matrix representation of a binary self-relation can be applied to binary relations on two different sets. For example, the M for R where

$$\begin{aligned} A &= \{a, b, c\} \\ B &= \{e, f, g, h\} \quad \text{is} \\ R &= \{(a, e), (c, g)\} \end{aligned}$$

Assume the rows are labeled with the elements of A and the columns are labeled with the elements of B . Note that the sizes of A and B can be different, and the order of the elements of A and B matters.

Modify the code from Exercise 208 to handle relations on two different sets.

Would it make sense to modify the classifying code you wrote in previous exercises to handle these relations?

Hint 216 Instead of passing just n you will need to pass two parameters: r , the number of rows in the matrix, and c , the number of columns. The size in bits of M will be $r \times c$.

AT A GLANCE

Exercise 217 Modify the code from Exercise 208 to handle relations on three different sets, which can be of different sizes.

of lists.

Hint 217 A 3-dimensional matrix is representable as a list of lists

արդիտական և գումարայի մաթրիչ բերեստավորություն — որտ ոչ առաջ.

ՏԱՐԱՎԱՆԴԻ ՀԱՅՈՒԹՅՈՒՆ

4.3 Relational Databases

Til was quietly conversing with Tessie when Abu and Ila arrived. Tessie gave them a cheery greeting, then left without further ado. Til was in fine form, barely acknowledging their presence before he began speaking.

“Binary relations can be generalized to n -ary relations quite easily — and advantageously. Just as binary relations are for couples, 5-ary relations are for 5-tuples, 123-ary relations are for 123-tuples, and so on — n -ary relations are for n -tuples. Two definitions apply: an **n -ary relation** on domains A_1, A_2, \dots, A_n is a subset of $A_1 \times A_2 \times \dots \times A_n$. The number n is called the **degree** of the relation.”

“So,” said Ila, “let’s see if I’ve got this right. The A_i domains of the relation are the individual sets that we choose elements from to form tuples? And so, for example, $(a1, b3, c2378, d0, e3)$ could be a five tuple where $a1 \in A, b3 \in B, c2378 \in C, d0 \in D$, and $e3 \in E$?

“Yes and yes,” said Til. “So,” said Abu, “to use the same names as you did, Ila, $A \times B \times C \times D \times E$ is the product that a subset can be taken from to make a relation, where $(a1, b3, c2378, d0, e3)$ is just one of many 5-tuples in the relation.”

“Here,” said Ila, displaying Table 4.1, “I’ve made a more realistic example, letting $A = \text{Name}$, $B = \text{Address}$, $C = \text{City}$, $D = \text{State}$, and $E = \text{Zip}$. I’ve seen enough databases to know that they look like tables.”

“Nice!” said Abu, glancing at Til, who seemed to be enjoying this exchange, and felt no need to jump in. “Even for this tiny of a database it makes it clear what’s what.”

Name	Address	City	State	Zip
Bill	123 S Main	Rexburg	ID	83440
Bob	51 S Center	Rexburg	ID	83440
Sue	4134 E 200 West	Rigby	ID	83442

Table 4.1: A tiny fictitious address book.

Now Til jumped in. “This is nice surface-scratching,” he said, “but I want you to appreciate that *Relational Database Theory* is a huge, sprawling field of study. An amazingly practical field of study. All based on the

mathematics of tuples. So here's what I want you to do. I need to help my wife with something, so I'll give you a little bit of vocabulary, then I'll leave you alone for a few minutes to continue exploring this field. While you're seizing this GPAO, please be on the lookout for three basic mathematical operations that can be performed on n -ary relations. Ila, are you familiar with SQL?"

"Yes," Ila said. "So I can help Abu learn the basics of that." Abu smiled gratefully, silently thanking Ila that she could be an experienced guide on this adventure.

At that, Til rattled off some terms. "**Relational databases** contain relations (**tables**). Tables contain tuples (**records** or **rows**). Records contain elements of domains (**fields**, **attributes**, or **columns**). Records are identified by **keys**. A **primary key** is a field or fields that is/are used to **uniquely** identify each record."

"Goodness, gotta run," said Til. "I'll see you in a little while. Oh, one more thing — primary keys must be unique everywhere and everywhen — for all time and throughout all space!" And with that mysterious comment he left them to their devices.

"Well," said Abu, "where do we start?" Ila, busy searching, raised her eyebrows and said, "Hold on a sec, I've got something here. First of all, do you know anything about SQL?" "Never even heard of it," said Abu. "Ess-cue-ell sounds like some kind of sport league." "Well it's not," said Ila. "It's the Structured Query Language⁸ — what we mostly use at my company — well, under the covers anyway — to talk to relational databases."

"I see," said Abu, who didn't really. "So did you already find those three mathematical operations Til said to be on the lookout for?" "I think so," said Ila. "Here, I'll show you the definitions of the

- Projection,
- Selection, and
- Join

operations, and then some examples using SQL."

Projection is selecting rows containing only specified columns.

The projection P_{i_1, i_2, \dots, i_m} maps an n -tuple to the m -tuple formed by deleting all fields not in the list i_1, i_2, \dots, i_m .

"The way I learned it is that projection is like taking a vertical 'claw slash' through a table, where only some columns get taken — clawed — out. Here," Ila paused, her fingers typing in a blur, "look at Table 4.2."

"Hmm," said Abu, "it looks like you took our simple address book table, and made i_1 be Name, i_3 be City, and i_5 be Zip."

"Right," said Ila, "It's the projection P_{i_1, i_3, i_5} , to use that notation I found. Now there's also **Selection**, which is selecting rows whose columns match some condition. So, for a selection of Rexburg residents only, we need a condition: if the City column matches Rexburg, select it, otherwise leave it out. SQL specifies that condition using a WHERE clause, like this:"

Name	City	Zip
Bill	Rexburg	83440
Bob	Rexburg	83440
Sue	Rigby	83442

Table 4.2: Three 3-tuples from a projection on a 5-tuple table.

```
SELECT * FROM address_book WHERE City = 'Rexburg'
```

Ila pointed to Table 4.3 while Abu said, “I take it `address_book` is the name of the table?”

Name	Address	City	State	Zip
Bill	123 S Main	Rexburg	ID	83440
Bob	51 S Center	Rexburg	ID	83440

Table 4.3: Rexburg only selection.

Ila said, “That’s right. And the star means take every column. We call that a ‘wildcard’ because it matches anything, like when you use a wild card, like a Joker, in card games to substitute for any other card. Oh, by the way, the SQL command for the projection in Table 4.2 is

```
SELECT Name, City, Zip FROM address_book  
so, you see, you spell out the fields you want and just leave off the  
WHERE clause. Or keep it, if you just want some of the rows.”
```

Abu stared for a few seconds, then raised his objection. “I thought you said `SELECT` was for selection, shouldn’t that be `PROJECT ...?`”

Ila was ready for that. “It *should be*, but unfortunately it’s not. SQL uses the same keyword for both selection *and* projection. Don’t ask me why.”

Abu said, “Okay, I’m not going to argue that decision, but check me on this — with `WHERE` restrictions leaving certain rows out, selection is — to use your analogy — like taking a *horizontal* claw slash through a table?”

“Yes, good point!” said Ila. “So, are we clear on selection and projection?” “I think so, yes,” said Abu. “Good,” said Ila, “because **Join** is the hardest operation of the three. It essentially creates a single table from two (or more) tables.”

“The **join** J_p takes a relation R of degree m and a relation S of degree n and produces a relation of degree $m - p + n$, where p is the number of fields R and S have in common, by finding all tuples $(a_1, a_2, \dots, a_{m+n-p})$ such that $(a_1, a_2, \dots, a_m) \in R$ and $(a_{m-p+1}, a_{m-p+2}, \dots, a_{m-p+n}) \in S$.

“Gonna need a good example to unravel that one,” Abu said. “Absolutely,” said Ila. “We’ll need two tables for R and S that have some fields in common. Let’s make R be a *birthday* table, and S something related to that — um, you think of something while I give Bill, Bob and Sue some birthdays. (See Table 4.4.)”

Abu thought for a moment and said, “How about a table that associates the 12 signs of the Zodiac with the month and day they start on?”

“Yeah, that’s perfect!” said Ila, and busied herself creating Table 4.5 after her quick search brought up the data.

It occurred to Abu that Ila, in the nearly two months they had worked together, had never asked him for his sign, nor he hers. Perhaps with her generation horoscopes weren’t a thing. He thought it best not to mention that he *occasionally* looked at his!

“Okay,” said Ila, “to unravel this join business, let’s see what we have.

Name	Year	Month	Day
Bill	1992	Jan	1
Bob	2001	May	13
Sue	2000	Dec	22

Table 4.4: Names and birthdays.

Month	Day	Sign
Jan	20	Aquarius
Feb	19	Pisces
Mar	21	Aries
Apr	20	Taurus
May	21	Gemini
Jun	21	Cancer
Jul	23	Leo
Aug	23	Virgo
Sep	23	Libra
Oct	23	Scorpio
Nov	22	Sagittarius
Dec	22	Capricorn

Table 4.5: Starting dates of the 12 Zodiac signs. Ending dates are inferable.

We have $R = \text{birthday}$ and $S = \text{zodiac}$. The tables R and S have two fields in common — Month and Day. We want to join $a_1 = \text{Name}$, $a_2 = \text{Year}$, $a_3 = \text{Month}$, $a_4 = \text{Day}$, and $a_5 = \text{Sign}$, given that attributes a_1, a_2, a_3, a_4 are in birthday and attributes a_3, a_4, a_5 are in zodiac .”

“So,” broke in Abu, “if I got this, $m = 4, n = 3, p = 2$, and $m - p + n = 5$. Then the join would have 15 tuples, with 12 rows from zodiac ‘joined’ together with 3 rows from birthday .”

“Right!” said Ila. “Give me a minute and I’ll create the joined table — and I’ll sort it by increasing Month, then Day.” Abu was fascinated as he watched Ila think and type at breakneck speed.

“Voilà!” said Ila, and with a flourish showed Abu the final Table 4.2.

Figure 4.2: The birthday and zodiac tables joined on their common Month and Day fields. The ‘-’ indicates a missing (null) field value.

Name	Year	Month	Day	Sign
Bill	1992	Jan	1	-
-	-	Jan	20	Aquarius
-	-	Feb	19	Pisces
-	-	Mar	21	Aries
-	-	Apr	20	Taurus
Bob	2001	May	13	-
-	-	May	21	Gemini
-	-	Jun	21	Cancer
-	-	Jul	23	Leo
-	-	Aug	23	Virgo
-	-	Sep	23	Libra
-	-	Oct	23	Scorpio
-	-	Nov	22	Sagittarius
-	-	Dec	22	Capricorn
Sue	2000	Dec	22	-

“So what this tells me,” said Abu, “is if I disregard any record with a null field value, except the last two, and I crunch those two together because they match on Month and Day,⁹ what I’m left with is the single tuple — in Python style — (`'Sue', 2000, 'Dec', 22, 'Capricorn'`). So it seems that ‘Sue’ is the answer to the ‘who has a birthday on the first day of a Zodiac sign’ question.”

Til returned, sat down, and silently observed Ila and Abu, too keenly concentrating to notice his return. “So look, Abu,” said Ila, “we can also tell from these tables that Bill’s sign is Capricorn and Bob’s is Taurus.”

“Oh yeah, I see that,” said Abu. Now Til cleared his throat, and they jumped, startled. “Til, we didn’t see you come back in,” said Ila. Abu thought “Good thing you didn’t catch us goofing off” but then thought “Wait, we never goof off!”

Til smiled and said, “I’m glad to see the progress you’ve made here. Tell me, Ila, do you have enough SQL experience to ask and answer what Bill’s and Bob’s Zodiac signs are?” “Do you mean,” said Ila, “can I create

⁹ A join that keeps only the tuples that match on their shared field values is called a “natural” join.

the more complex join/select/project SQL query with the necessary *where* conditional logic?"

"Precisely," said Til.

"Not really," confessed Ila. "But I'm sure I could learn!" Abu chimed in, "I'm sure you could too, Ila — but I'm having too much fun with Python to want to learn SQL! Couldn't we do the same logic with Python?"

"Sounds like a good exercise!" Til said, winking at Ila. "Realize, of course, that simple relational modeling only goes so far. The complexity of today's databases and the queries that extract meaning from them will require many years of work to achieve real expertise. So we'll leave it at that."

As Til started to signal they were done — "Wait!" said Abu. "Earlier you said something about primary keys — about needing to be unique through all time and space, or something like that. Could you elaborate on that, please?"

"I could if I had time, but unfortunately I don't right now," said Til. "But let me just say that good keys are *key* — vital, crucial — to a well-designed relational database. So do your best to learn what they do and how they do it — or should do it — and I guarantee you'll be a better logical thinker for it!"

Exercise 218 Add a few more records to the birthday table, and implement that table and the zodiac table as a tuple of tuples in Python. Then answer the query "What is a person's Zodiac sign?" by writing code that does the appropriate projections, selections and joins of these two tables.

Hint 218 Tuple-slicing is good for projections; filter and product, and of course, map and zip are useful too. Start by creating a full zodiac table with a row for each day of the year and its sign. The sign for each day is inferable from the original zodiac table that just gives start days.

AT A GLANCE

Exercise 219 Here is a tiny database of students and courses, implemented as Python lists and dictionaries.

```
students = [1, 2, 3, 4, 5, 6, 7, 8]
courses = [100, 200, 300, 400]
student2course = { \
    1:[100, 200], 2:[100], 3:[100, 200, 300], \
    4:[100, 200, 300, 400], 5:[100], 6:[100], \
    7:[100, 200], 8:[100]}
```

```

7:[100, 200], 8:[100, 200, 300, 400]}
course2student = { \
100:[1, 2, 3, 4, 5, 6, 7, 8], \
200:[1, 3, 4, 7, 8], \
300:[3, 4, 8], 400:[4, 8]}

```

You can tell just by looking, but your task is to write code to

- *find all students who have taken all courses, and*
- *find all courses that have been taken by all students.*

all, and map functions.
use a couple of lambda functions and a single call to the list, filter,
Hint 219 The queries are “duals” of each other. Both of them could

```

# course2student is courses[student][course]
# courses that have been taken by all are student[s]
courses = [course for course in courses if len(course) == len(students)]
# courses that have been taken by some are student[s]
courses = [course for course in courses if len(course) < len(students)]

```

ANSWER Here are the details:

Exercise 220 Investigate primary keys, secondary keys, foreign keys, and “normal forms” for relational databases.

with these topics.
Hint 220 A report on your findings will show your engagement level

.
ANSWER Here are the details:

4.4 Equivalence Relations

Til started right in, with an urgency that took Abu and Ila by surprise.

“A VERY IMPORTANT type of relation is known as an *equivalence relation*. This type captures what it means for something to be somehow “the same as” something else, without necessarily being identical or equal to it. So first of all, I’ll give you the definition:

An **equivalence relation** is a binary relation that is

- **reflexive**,
- **symmetric**, and
- **transitive**.

Note that plain old equality ($=$) is an equivalence relation — in fact, it comes in first place, the gold medal ER — I’ll abbreviate to save five syllables every time I say it!

Does $a = a$? Yes, so = is reflexive.

If $a = b$ does $b = a$? Yes, so = is symmetric.

If $a = b$ and $b = c$, does $a = c$? Yes, so = is transitive too.”

Abu thought of something. “If equality is the gold medal ER, then what about the silver and bronze medalists?”

Til smiled. “Silver medal easily goes to equality operators in programming languages. In fact, Java makes it explicit. For example, the code in Figure 4.3 works like the `java.lang.Object.equals()` method’s documentation says it should:”

```
public boolean equals(Object obj)
```

Indicates whether some other object is “equal to” this one.

The equals method implements an equivalence relation on non-null object references:

- It is reflexive: for any non-null reference value x , $x.equals(x)$ should return true.
- It is symmetric: for any non-null reference values x and y , $x.equals(y)$ should return true if and only if $y.equals(x)$ returns true.
- It is transitive: for any non-null reference values x , y , and z , if $x.equals(y)$ returns true and $y.equals(z)$ returns true, then $x.equals(z)$ should return true.

[...]

The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y , this method returns true if and only if x and y refer to the same object ($x == y$ has the value true).

Ila jumped in. “I found the same kind of wording and examples for the JavaScript `Object.is` operation. So I bet there’s something similar in Python — and other languages too I suppose?”

Til laughed. “Yes, math invades programming quite pervasively — but there’s a GPAO for you — find all the languages whose documentation explicitly acknowledges that fact!”

Abu coughed. “I’m excited to take that GPAO — but I’m dying of curiosity here — what about that bronze medal ER?!”

4.5 Congruences

“It’s coming, Abu, real soon now!” Til was enjoying keeping him in suspense. “Ila, do you recall the recent ‘filters’ exercise¹⁰ whose hint mentioned the three-letter-acronym CMM?”

“Yes,” said Ila. “The hint said this had something to do with some number theory concepts coming later...”

“Right,” said Til. “CMM is *Congruence Mod M*, and yes — it’s the bronze medal ER.”

```
Integer a = new Integer(1);
Integer b = new Integer(1);
System.out.println(a == b);
System.out.println(a.equals(b));
```

Figure 4.3: Java Integers’ equality of structure but not identity. If you wrap this code in a class and main method, and run it, it prints:

```
false
true
```

The `a == b` comparison is false because the two Integer objects are *not* identical. The `a.equals(b)` comparison is true because they are structurally ‘the same’ — they both wrap the integer 1.

¹⁰ Exercise 198 to be precise.

"I'm relieved to hear," said Abu, "that it's not just another building block leading up to that third-place ER!"

Til chuckled at that. "No, but perhaps it's time to at least introduce the number-theoretic concept CMM stems from — or the building block, as you said, that it builds on."

Abu was all ears. Ila was nervous — naturally leery of anything "theoretic" but especially when that anything was numbers.

"A basic fact about numbers — integers —" Til clarified, "that can be divided evenly by other integers is denoted by the vertical bar '| — acting as the verb placed between subject and object in a three-word sentence: $a | b$, pronounced a **divides** b , is true if $\frac{b}{a}$ is an integer."

Abu thought that looked strange — upside-down? He decided to check his understanding. "So let me see if I got this," he said. "3 divides 12 but not 13, right?"

Til paused, noticing that Ila was looking at Abu strangely, and suspected that she was succumbing to her theory-aversion. But she got a grip on whatever it was, and chipped in. "That's right, because $\frac{12}{3}$ is the integer 4, but $\frac{13}{3}$ is not an integer — it's $4\frac{1}{3}$."

Til smiled his most approbative smile. "Excellent!" he said, "you two are grasping the basic idea of *divisibility* — this is the building block for the wonderful edifice of number theory that we'll be exploring in a few weeks."

"But for now, **congruence mod m** says a relationship holds between two integers: a is congruent mod m to b , denoted $a \equiv_m b$ or sometimes $a \equiv b \pmod{m}$, means a and b have the *same remainder* when divided by m ."

"In other words," he continued, " $a \equiv_m b$ if $a \bmod m = b \bmod m$, where 'mod' is the remainder-finding operator. Which is what we'll focus on now. $4\frac{1}{3}$ has a legitimate relationship with the numbers 13 and 3, but we want 4 and 1 to take the spotlight — 4 being the multiplier of 3 that gives us 12, and 1 being the remainder when 13 is divided by 3, because $13 - 12 = 1$. So let's look at this relationship in differently expressed but equivalent ways, and then see how we can use these ways to show that CMM is an ER."

$$a \equiv_m b \text{ if}$$

1. $m | a - b$
2. $\exists c (c \cdot m = a - b)$.
3. $\exists k (a = k \cdot m + b)$.
4. In Python, where % is the mod operator: $a \% m == b \% m$
5. Or, $(a - b) \% m == 0$.

"For showing CMM is *reflexive*, use 1: m divides the difference¹¹ of a and b , where b is a : $a \equiv_m a$ because $m | a - a$. Any number divides 0."

"For showing CMM is *symmetric*, use 2: if $a \equiv_m b$ then $b \equiv_m a$ because if c is such that $c \cdot m = a - b$ then $-c$ is such that $-c \cdot m = b - a$."

¹¹ A nuance of notation for "divides the difference" deserves attention. Note that in $m | a - b$ there is no ambiguity nor need for parentheses — as in $m | (a - b)$. There is no danger of $m | a - b$ being confused with $(m | a) - b$, because the statement $m | a$ (m divides a) is not a number — which is what the '-' operator expects.

"For showing CMM is *transitive*, use 3: if $a \equiv_m b$ (so $a = k \cdot m + b$) and $b \equiv_m c$ (so $b = j \cdot m + c$) then $a \equiv_m c$ because $i = k + j$ is such that $a = i \cdot m + c$. To see this, replace b with its equivalent $j \cdot m + c$ in $a = k \cdot m + b$ to get $a = k \cdot m + j \cdot m + c$; and then replace $k \cdot m + j \cdot m$ with its equivalent $(k + j) \cdot m$ in $a = k \cdot m + j \cdot m + c$ to get $a = (k + j) \cdot m + c$."

4.6 Arithmetic Modularity

"CMM is an example of *modular arithmetic*," said Til, "which is sometimes referred to as *clock arithmetic* — but that really does it a disservice, as clock arithmetic restricts the modulus to be either 12 or 24.¹² Whenever you interact with a clock or a calendar you flirt with modular arithmetic, even if you are unaware of their relationship. Indeed, the seconds, minutes, hours, days, weeks, months, years — the seasons and cycles of our lives are all counted with modular arithmetic. A circular analog clock face suggests the cyclical nature of our timekeeping habits. Replace the 12 at the top with a 0 to get the idea. Zero o'clock, all hands straight up — is it midnight or noon?"¹³

"I'd say it depends on how light it is!" said Abu, a smile playing his lips. Ignoring him, Til continued with an example.

"11:00 am is 4 hours away from 3:00 pm ($11 + 4 = 15$, $15 \% 12 = 3$). 75 hours in the future is 3 days ($75 / 24 = 3$) + 3 hours ($75 \% 24 = 3$ as well) away. To know what time the clock will display in 75 hours, if the time now is 4:00, add 4 to 75 and reduce modulo 24. ($75 + 4 = 79$, $79 \% 24 = 7$). It will display 7:00. Or just add 3 (hours) to 4 (:00) to get 7 (:00)."

Abu thought he saw Ila drifting, but she quickly stirred when she saw him glance at her motionless hands.

"Now," said Til, "How many years is 1000 days? Mod 365 would seem to be the ticket here, but integer division by 365 is what's really called for. $1000 // 365 = 2$; $1000 \% 365 = 270$. By the way, the floating-point division of 1000 by 365 is approximately 2.739726."

Ila, to show Abu she was following this, broke in with, "Are you saying that 2.739726 is the precise answer to that question? 'How many years is 1000 days?'"

"No," said Til, simply. "Because what is 0.739726 times 365? It's 269.99999 — 270 if we take the ceiling. Thus that fraction of a year is 270 days, so 1000 days from today is 2 years and 270 days away. But regardless of how much precision the fraction has, the count is off — because of leap years!"

Abu said, "So we have to change the divisor and modulus to 366 every 4 years?"

Ila objected, "But there's *not* a leap year every 4 years — there are exceptions!"

"Exactly!" said Til. "The point is, calendars in years and days are

¹² Clock arithmetic can also apply to a modulus of 5 (intervals of when the minute hand is pointing directly to the hour marker), 60 (minutes per hour, seconds per minute), or 360 (since the face of an analog clock is a circle).

¹³ 12:00 am (midnight) is called "zero hundred" (0:00) hours in military-time-speak, to distinguish it from 12:00 pm (noon) or 12 hundred hours (as if it were written without the colon as 1200). But military time is still just mod 24, whereas "normal" AM/PM time is mod 12.

messy, and leap years are a big reason why. Weeks work better, where mod 7 is as regular as — clockwork! What day of the week will 1000 days from now be if today is Tuesday? Well, n days from Tuesday is Tuesday again, if n is 7, 14, 21, 28, 35, or $7 \cdot k$ for any integer $k \geq 6$. Any multiple of 7 looks like 0 when you mod it by 7. How many of these 7-day weeks are in 1000 days? It doesn't matter. What fraction of a week remains after cycling through all those whole weeks is what matters. So $1000 \% 7 = 6$ means Tuesday + 6 days = Monday is the answer — and leap years are simply a non-issue.”

Ila and Abu pondered this silently while Til gathered his thoughts.

“We'll delve into modular arithmetic and congruences much more later, but in the meantime, let's take a moment and investigate one very special feature of an ER, illustrated quite nicely by CMM.”

4.7 Equivalence Classes and Partitions

“Take 5 as the modulus and specialize CMM to CM5.” Til paused only for a moment, but this prompted Abu to rein in his wandering attention. “This is just for convenience — we could have used any one of infinitely many *moduli*.¹⁴ When you take an integer i and perform the mod 5 operation on it (`def mod5(i): return i % 5`) you can say you've ‘modded i by 5’. Now note that *every* integer can be put into one of five subsets, namely, those integers that when modded by 5 give a remainder of 0, 1, 2, 3, or 4 (see Table 4.6).”

¹⁴ Plural of modulus. Any integer $m > 1$ would show this property too.

Table 4.6: Subsets of \mathbb{Z} organized by CM5. Which $i, 0 \leq i < 5$, is added to some multiple of 5 is the organizing principle.

Subset	Remainder	Multiples of 5	Examples
A_0	0	plus 0	0, 5, 10, 15, -15, -10, -5
A_1	1	plus 1	1, 6, 11, 16, -14, -9, -4
A_2	2	plus 2	2, 7, 12, 17, -13, -8, -3
A_3	3	plus 3	3, 8, 13, 18, -12, -7, -2
A_4	4	plus 4	4, 9, 14, 19, -11, -6, -1

“For any given set, the subset of all elements related to a particular element forms a universal relation *on that subset*. In other words, it contains all possible pairs. The number of such subsets is called the **rank** of the ER. Each of the subsets is called an **equivalence class** (EC). Square brackets around an element denote the EC in which the element lies. $[x] = \{y \mid (x, y) \in R\}$, or $[x] = \{y \mid xRy\}$. The element in the brackets is called a **representative** of the EC. Any one of them could be chosen. For example, here's another way to look at the organization of Table 4.6, showing that the ECs of CM5 have an infinite number of representatives.”¹⁵

- $A_0 = [0] = [5] = [-5] = [10] = [-10] = \dots$
- $A_1 = [1] = [6] = [-4] = [11] = [-9] = \dots$
- $A_2 = [2] = [7] = [-3] = [12] = [-8] = \dots$
- $A_3 = [3] = [8] = [-2] = [13] = [-7] = \dots$

¹⁵ Arguably, the best representatives are the ones listed first — the ones with the x 's in $[x]$ with the simplest — smallest non-negative — values.

- $A_4 = [4] = [9] = [-1] = [14] = [-6] = \dots$

"Now, let's say $A_0, A_1, A_2, \dots, A_{n-1}$ are n subsets of a generic set S ." Til adjusted his display, looked to make sure Abu and Ila were still with him, and went on. "A **partition** of S is formed by these subsets if they are nonempty, disjoint, and if they *exhaust* S . If these three conditions hold, we say these subsets **partition** S .¹⁶ In symbols, where i and j range from 0 to $n - 1$:"

1. $A_i \neq \emptyset$.
 2. $A_i \cap A_j = \emptyset$ if $i \neq j$.
 3. $\bigcup A_i = S$, short for $A_0 \cup A_1 \cup A_2 \cup \dots \cup A_{n-1} = S$.

"I've tabulated CM5's subsets' members one more time in Table 4.7 to make it clear how CM5 partitions \mathbb{Z} ."

Set	EC	$n < -15$				\downarrow	$n > 19$			
A_0	[0]	...	-15	-10	-5	0	5	10	15	...
A_1	[1]	...	-14	-9	-4	1	6	11	16	...
A_2	[2]	...	-13	-8	-3	2	7	12	17	...
A_3	[3]	...	-12	-7	-2	3	8	13	18	...
A_4	[4]	...	-11	-6	-1	4	9	14	19	...

"We gather the simplest representatives from the CM5 ECs together into one set, and name it \mathbb{Z}_5 . This is a subset of \mathbb{Z} , consisting of just the possible remainders, or *residues*, that you can get when you mod by 5. To generalize, the \mathbb{Z}_n **residue set** is the set of possible remainders that result when modding by n , i.e., $\{0, 1, 2, 3, \dots, n - 1\}$."

"We'll be seeing these residue sets again!" And with that, Til left his twotees to work on exercises for a short time.

¹⁶ The **Partition Theorem** says that the equivalence classes of an equivalence relation R partition a set S into disjoint nonempty subsets whose union is the entire set. This partition is denoted S/R and is called

- the quotient set, or
 - the partition of S induced by R , or
 - S modulo R .

Table 4.7: The $A_i = [i], 0 \leq i < 5$ subsets with their ECs and partial membership listings. The numbers in the column in the middle pointed to by the arrow and **bold-faced** are the representatives of the ECs. None of these subsets is empty, none has any members in common with any other, and no integer is left out — a partition it is!

Exercise 221 How many buses does the army need to transport 1,128 soldiers if each bus holds 36 soldiers?

A test item on a National Assessment of Educational Progress (NAEP) mathematics assessment presented [this] question to eighth grade students as an open-ended prompt demanding a written answer:

Hint 221 This is to see if you are smarter than an eight-grader:

မြန်မာစာတမ်းကို အသေးစိတ် ပေါ်လေ့ရှိခဲ့သူများ မြန်မာစာတမ်း၏ အမြန်ဆုံး ပေါ်လေ့ရှိခဲ့သူများ ဖြစ်ပါသည်။

Exercise 222 Create a table similar to Table 4.7 to make it clear how CM7 partitions \mathbb{Z} ."

Hint 222 CMT will require adding some rows, but not columns.

ATLAS Egg Survey

Exercise 223 Call two bitstrings equivalent if they both end in a 0 (and have length at least one). What CMM equivalence relation does this suggest?

Hint 223 Think parity, or oddness and evenness.

Answers 553 CWS is the entity responsible for setting standards (norms) in the field of quality management.

Exercise 224 Consider the set of all bitstrings having length at least 3. Show that the relation on this set that says two bitstrings are related if they agree in the first three bits is an equivalence relation.

Hint 22A The function `lambda bitsizing:bitsizing[3]:` will partition into 8 ECs all bitsizing having length at least 3. Argue that the three properties defining an ER are satisfied when equiva-lent bitstrings belong to the same EC.

AL DÍA EN LA PRENSA

Exercise 225 Investigate and experiment with the code and concepts found here.

Hint 225 What are some good ERs to create interesting ECs from?

MOLK

ANSWER You must show ample evidence of a serious injury at

4.8 Same Old Same Old

Pleased to see Abu and Ila absorbed in their work, Til waited until they looked up and seemed ready to move on.

“A *key point* that bears repeating,” said Til, “is that whenever you have a relation where something has the same whatever as something else, you have an equivalence relation! Be it

- the same age as
- the same birthday as
- the same name as
- the same parents as
- the same first three bits as
- the same ... as

— whatever it may be — look for the words ‘the same (as)’ — like in CMM, where ‘the same remainder as’ is the key phrase.”

Abu and Ila were contemplative, but still engaged.

“Here,” said Til, “I’ll show you another foolproof way to tell if a relation is an ER. Let A be any *nonempty* set, and let f be a function that has A as its domain. Let R be the relation on A consisting of all ordered pairs (x, y) such that $f(x) = f(y)$. Showing that R is an ER¹⁷ on A is super easy:

- R is reflexive: $f(x) = f(x)$, check.
- R is symmetric: $f(x) = f(y) \rightarrow f(y) = f(x)$, check.
- R is transitive: $f(x) = f(y) \wedge f(y) = f(z) \rightarrow f(x) = f(z)$, check!

See? Foolproof!”

Proof we’re fools, thought Ila — if we don’t understand!

Til was relentless. “For another classic example, let R be the relation on the set of ordered pairs of positive integers such that $(a, b) R (c, d)$ if $ad = bc$. Is R an ER?”

Not waiting even a second, Til answered his own question.

“Yes, by virtue of the function f from the set of pairs of positive integers to the set of positive rational numbers that takes the *pair* (a, b) to the *fraction* $\frac{a}{b}$, since $ad = bc$ if and only if $\frac{a}{b} = \frac{c}{d}$.”

That was a mouthful, thought Ila. She was struggling to keep up. So was Abu, who was furiously scribbling some notes. Ila, whose nightmarish visions of grade school fractions made her visibly tremble, asked, “Til, please, some examples?”

“For example,” said Til, “ $\frac{3}{4} = \frac{6}{8}$ because $3 \cdot 8 = 4 \cdot 6$. $\frac{9}{15} = \frac{3}{5} = \frac{21}{35}$ because $9 \cdot 5 = 15 \cdot 3$, $9 \cdot 35 = 15 \cdot 21$, and $3 \cdot 35 = 5 \cdot 21$.”

“I’ve got one,” said Abu. “ $\frac{19}{141} = \frac{133}{987}$ because $19 \cdot 987 = 141 \cdot 133$.”

“Very good!” said Til. “The beautiful thing about this ER is that multiple fractions are obviously the ‘same’ fraction if they ‘reduce to lowest terms’ to the identical fraction.”

At that Ila’s budding “aha” moment burst into full bloom. “So that’s how reducing to lowest terms shows how fractions are the same! Why didn’t my grade school teachers explain it that way?”

¹⁷ Underlying *any* ER is a function. There is not a single ER missing this feature. *Not a single one!* Identify this function (e.g., mod5) to capture the essence of the equivalence.

Exercise 226 What are the equivalence classes of the ER described as having a “foolproof” way to tell that it’s an ER?

Hint 226 Think about a bijective function and its inverse.

AT&T ASSET MANAGEMENT

Exercise 227 “Characterize” R , the relation that says xRy if x and y are siblings. The domain of R is people. If it is an ER, show it has reflexivity, symmetry and transitivity. If not, explain which of these three properties it lacks.

this exercise.

Hint 227 I'm my own *granada*, is a song that doesn't apply, except maybe to express an absurdity that you should avoid in analyzing

၁၃၆

ANSWER Not all E&B factors reflectivity. You are not likely to find many who are not E&B factors.

Exercise 228 Characterize xRy if x and y have the same parents, with people as R 's domain.

Hint 228 This is really straightforward.

AT&T AS A 5G PROVIDER

Exercise 229 Characterize xRy if x and y share a common parent, with people as R 's domain.

Hint 229 Sharing is caring. Be careful!

barely

გთხოვთ ამ სტრუქტურას დანართოთ და გვიყვანეთ მათ შესაბამის მნიშვნელობები. ამასთან ერთად, გვიყვანეთ მათ შესაბამის მნიშვნელობები, რომელთა განცხადება გვიჩვენ მათ შესაბამის მნიშვნელობების მიზნით.

Exercise 230 Characterize xRy if x and y speak a common language, with people as R 's domain.

Hint 230 Exercise 229 could be helpful.

AT A GLANCE ANSWERS

Exercise 231 Characterize xRy if x shares a vowel with y , with English words as R 's domain.

Hint 231 Exercise 229 could be helpful.

Exercise 231 It's not too hard to shift from English words to letters in order to figure out which vowels share a vowel with another vowel.

Exercise 232 Characterize xRy if $x \leq y$, with numbers as R 's domain.

Hint 232 It's pretty easy to see this one just by plugging in some numbers.

AT A GLANCE ANSWERS

Exercise 233 Characterize xRy if $x^2 + y^2 = 1$, with numbers as R 's domain.

Hint 233 It's pretty easy to see this one just by plugging in some numbers.

Exercise 233 If $x^2 + y^2 = 1$, then $x^2 = 1 - y^2$. This means that x^2 is a nonnegative number, so x must be a nonnegative number between -1 and 1 .

Exercise 234 Characterize xRy if $x + y = 42$, with numbers as R 's domain.

Hint 234 Think about what the solution to the equation $x + x = 42$ tells you.

AT A GLANCE

Exercise 235 Characterize xRy if $[x] = [y]$, with numbers as R 's domain.

Hint 235 It's pretty easy to see this one just by plugging in some numbers.

ANSWER

Not all ERs have reflexive relations. $\{3, 4\} \neq \{3, 4\}$.
For example, $\{1, 2, 3\}$ has a reflexive relation $\{(1, 1), (2, 2), (3, 3)\}$.

Exercise 236 Modify your “tabulation” code from Exercise 215 to determine how many relations of size $n = 2$, $n = 3$, and $n = 4$, are ERs, and how many are of two other special types, namely, the “preorder” and the “partial order” types. GPAOs await to delve into these, but for determining their frequency of occurrence, it will suffice to know that a preorder is a relation that is reflexive and transitive, while a partial order is one that is reflexive, antisymmetric, and transitive.

Hint 236 This is a straightforward exercise, and you can check your answers in the Enumeration table at this web site.

AT A GLANCE

¹⁸ *How Mathematicians Think*, by William Byers, intriguingly subtitled “Using Ambiguity, Contradiction, and Paradox to Create Mathematics” and masterfully exploring the question “Is mathematics algorithmic or creative?”.

```
def four_in_four_out(a, b, c, d):
    return a+b, d//b, d-a, c*d
```

Figure 4.4: A function returning four computations on four numbers.

4.9 How Mathematicians Think

“There’s an especially illuminating book¹⁸ I highly recommend for your consideration that will help you see how mathematicians make sense of the world. There’s one mathematician in particular I’ll mention in a minute, but first I want to return to something we started our investigation of functions with.”

“The function you saw earlier displayed in Figure 3.1 I’ve reproduced here for your convenience in Figure 4.4. It’s been modified slightly, renamed, and also made to just return four numbers, not a list of those four numbers.”

“We’re now in a position to revisit the question — what should the domain and codomain of this function be? Remember?”

“Using what we learned about the Cartesian product, we could make the inputs come from the product $\mathbb{Z} \times \mathbb{Z} \setminus \{0\} \times \mathbb{Z} \times \mathbb{Z}$, and the outputs from

$\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$. But with 4-tuples for inputs, and 4-tuples for outputs, why not just put them together? This function is simply a relation of 8-tuples:

$$\{(a, b, c, d, w, x, y, z) \mid w = a + b, x = d // b, y = d - a, z = c * d\}.$$

So remember, even with only one output, functions *are* relations, and relations in their full glory are amazingly versatile!"

Abu and Ila practically burst into applause, but they managed to curb their enthusiasm to where they just gave Til their heartfelt appreciation with smiles and thumbs up.

Returning their smiles, Til said, "I'd like to wrap up our whirlwind foray into relations with a shout out to Roger Penrose — an extraordinary individual — in addition to being a world-class mathematician he is also a world-class 'mathematical physicist'. I'd like *you* to find something he said about the relations between some pretty heavy topics that speaks volumes about his intellectual humility — something we should all emulate!"

4.10 Summary of Terms and Definitions

The **Cartesian product** (or just product) $A \times B$ of set A with set B is the set of all ordered pairs (2-tuples) (a, b) where $a \in A$ and $b \in B$. Generalizes to $n > 2$ sets in a straightforward way.

A **binary relation** R from a set A to a set B is a subset of $A \times B$.
 $R \subseteq A \times B$.

A **self relation** is a binary relation R on a set A , or a relation from A to A (from the set to itself). $R \subseteq A \times A$.

R is **reflexive** if $\forall x x \in U \rightarrow xRx$.

R is **symmetric** if $\forall x \forall y xRy \rightarrow yRx$.

R is **antisymmetric** if $\forall x \forall y xRy \wedge yRx \rightarrow x = y$

R is **transitive** if $\forall x \forall y \forall z xRy \wedge yRz \rightarrow xRz$.

An **n -ary relation** on domains A_1, A_2, \dots, A_n is a subset of $A_1 \times A_2 \times \dots \times A_n$.

The number n is the **degree** of the relation.

Relational databases contain relations (**tables**).

Tables contain tuples (**records** or **rows**).

Records contain elements of domains (**fields**, **attributes**, or **columns**).

Records are identified by **keys**.

A **primary key** is a field or fields that is/are used to **uniquely** identify each record.

Projection is selecting rows containing only specified columns. The projection P_{i_1, i_2, \dots, i_m} maps an n -tuple to the m -tuple formed by deleting all fields not in the list i_1, i_2, \dots, i_m .

Selection is selecting rows whose columns match a specified condition.

SQL is the Structured Query Language.

In an unfortunate naming collision, SQL uses the keyword **SELECT** for both projection and selection.

The **join** J_p takes a relation R of degree m and a relation S of degree n and produces a relation of degree $m + n - p$ by finding all tuples $(a_1, a_2, \dots, a_{m+n-p})$ such that $(a_1, a_2, \dots, a_m) \in R \wedge (a_{m-p+1}, a_{m-p+2}, \dots, a_{m-p+n}) \in S$. The p is the number of fields R and S have in common.”

An **equivalence relation** is a binary relation that is

- **reflexive**,
- **symmetric**, and
- **transitive**.

$a | b$, pronounced a **divides** b , is true if $\frac{b}{a}$ is an integer.

a is congruent mod m to b , denoted $a \equiv_m b$ or sometimes $a \equiv b \pmod{m}$, means a and b have the *same remainder* when divided by m .

$a \equiv_m b$ if $a \bmod m = b \bmod m$, where ‘mod’ is the remainder-finding operator.

$a \equiv_m b$ means

- $m | a - b$.
- $\exists c(c \cdot m = a - b)$.
- $\exists k(a = k \cdot m + b)$.
- In Python, where `%` is the mod operator: `a % m == b % m`
- Or, $(a - b) \% m == 0$.

For any given set, the subset of all elements related to a particular element forms a universal relation (contains all possible pairs) *on that subset*.

The number of such subsets is called the **rank** of the equivalence relation.

Each of the subsets is called an **equivalence class**.

Square brackets around an element denote the equivalence class in which the element lies.

$[x] = \{y \mid (x, y) \in R\}$, or

$[x] = \{y \mid xRy\}$.

The element in the brackets is called a **representative** of the equivalence class. Any one of them could be chosen.

Let $A_0, A_1, A_2, \dots, A_n$ be subsets of a set S . A **partition** of S is formed by these subsets if they are nonempty, disjoint and exhaust S .

1. $A_i \neq \emptyset$.
2. $A_i \cap A_j = \emptyset$ if $i \neq j$.
3. $\bigcup A_i = S$.

If these three conditions hold, we say the A_i subsets **partition** S .

The **Partition Theorem** says that the equivalence classes of an equivalence relation R partition a set S into disjoint nonempty subsets whose union is the entire set. This partition is denoted S/R and is called

- the quotient set, or
- the partition of S induced by R , or
- S modulo R .

The \mathbb{Z}_n **residue set** is the set of possible remainders when modding by n , i.e., $\{0, 1, 2, 3, \dots, n-1\}$.

Part II: Excursions



What is *consciousness*? Well, I don't know how to define it.

I think this is not the moment to attempt to define consciousness, since we do not know what it is. I believe that it is a physically accessible concept; yet, to define it would probably be to define the wrong thing. I am, however, going to describe it, to some degree. It seems to me that there are at least two different aspects to consciousness. On the one hand, there are *passive* manifestations of consciousness, which involve *awareness*. I use this category to include things like perceptions of colour, of harmonies, the use of memory, and so on. On the other hand, there are its *active* manifestations, which involve concepts like free will and the carrying out of actions under our free will. The use of such terms reflects different aspects of our consciousness.

I shall concentrate here mainly on something else which involves consciousness in an essential way. It is different from both passive and active aspects of consciousness, and perhaps is something somewhere in between. I refer to the use of the term *understanding*, or perhaps *insight*, which is often a better word. I am not going to define these terms either — I don't know what they mean. There are two other words I do not understand — *awareness* and *intelligence*. Well, why am I talking about things when I do not know what they really mean? It is probably because I am a mathematician and mathematicians do not mind so much about that sort of thing. They do not need precise definitions of the things they are talking about, provided they can say something about the *connections* between them. The first key point here is that it seems to me that intelligence is something which requires understanding. To use the term intelligence in a context in which we deny that any understanding is present seems to me to be unreasonable. Likewise, understanding without any awareness is also a bit of nonsense. Understanding requires some sort of awareness. That is the second key point. So, that means that intelligence requires awareness. Although I am not defining any of these terms, it seems to me to be reasonable to insist upon these relations between them. — Roger Penrose, in *The Large, the Small and the Human Mind*.

5

Combinatorics and Probability

5.1 Frequently Subjective

“So, what is the connection between probability and counting?” asked Til. Ila smiled big at Abu, knowing she had won their bet that Til would, yet again, dispense with any hint of small talk when beginning their latest session. With a cheery return smile that made Ila believe he was strangely happy she had won, Abu said “Well, I think it has to do with counting the number of ways something can happen and then confirming that count by experimenting, like flipping a coin or rolling dice.”

“That makes you a frequentist!” said Til. Abu blinked and Ila stared. “As opposed to a subjectivist, whose view of probability is different.”

“Let’s compare and contrast these two schools of thought,” said Til, after a brief pause, while he looked back and forth from Abu to Ila, who were all ears. “To a frequentist, probability is counting the number of times something actually happens, like landing heads to use Abu’s coin flipping experiment, and comparing that frequency with the number of flips. The ‘landing heads’ when flipping a coin is called an event, the ‘landing-either-heads-or-tails (but not neither and certainly not both)’ is called the space of possibilities, or the probability space of this experiment. In other words, the probability space is all possible outcomes, and the event is a subset of that space. The frequentist holds that the frequency of actual outcomes, counting say 51 heads in 100 coin tosses, is an approximation to the exact probability of getting heads on any single toss of a coin.”

“So, if I understand this right,” said Ila, “I can say the probability of getting heads is 51/100, or 498/1000 or whatever it is after repeating the experiment however many times I want.” “Yes, if you’re a frequentist!” said Til. “But I really just want to say it’s 50%, or a 50-50 chance, because of how coins have two sides, and unless there’s something funny about my quarter here, tossing it gives no preference to one side or the other. So either is equally likely, right?” “So right, if you’re a thought-experiment frequentist!” said Til.

“But aren’t there fake coins that are biased to land more often heads than tails, or vice-versa?” said Abu. “Yes, but what’s your point?” said Til. “Well, if I did the experiment of tossing some such biased coin a thousand times, and counted 600 heads,” said Abu, “I’d be more inclined to suspect the coin was unfair than to think the likelihood of getting heads was 60%.” “That’s because you have a high degree of belief, a subjective intuition if you will, about how coins should behave.” said Til.

“Is that what a subjectivist is, someone who treats probability as a subjective belief in how likely something is?” said Ila. “Exactly right!” said Til. “Obviously, the frequentist approach cannot work in all cases, because some ‘experiments’ can’t be done, or if not outright impossible, can’t be repeated, or even tried once without disastrous results.” Abu chimed in, “Like what’s the probability that the world will end and humanity extinguish itself in a nuclear holocaust? I may think it’s not very likely, but someone else might disagree and think the probability is high.”

“While we’re being gloomy, I’ve got one!” said Ila. “What’s the probability that our sun will go supernova tomorrow?”

“I’d give that one a 0% chance, personally,” said Abu. “Same here,” said Til. “But that’s just our belief — mixed of course with scientific knowledge about how stars live and die. When people use pseudoscience to predict the future, or substitute opinion for fact, and ‘estimate’ probabilities to further some hidden agenda — that’s when math abuse, and truth abuse abound.”

“Truth abuse sounds worse than math abuse,” said Ila. “Truth is,” replied Til, “popular culture conspires to disguise truth, and substitute illusion for what’s real.”¹

Til paused, which gave Abu and Ila pause to consider the implications of what he just said. Then Ila had a thought. “Til, are you a frequentist or a subjectivist?” she asked. “Both!” said Til, with a mischievous smile. Then while softly clearing his throat he subvocalized the contradictory addendum, “And neither.”

“At any rate,” continued Til, full voice, “if we’re going to start to learn how to count today, we’d better get to it!”

5.2 Elementary and Basic

“First,” said Til, “I want you to distinguish between EEC and EGC — two TLAs sharing *Elementary* and *Combinatorics* — which is the study of counting, or of how things combine. We’re going to hit EEC — Elementary *Enumerative Combinatorics* — very lightly. Even lighter, for now, I’m just mentioning EGC — Elementary *Generative Combinatorics* — by way of contrast. We’ll delve deeper into that when we get to trees.”

“Counting,” Til said, “involves applying logic and some basic principles to find the number of ways some choice can be made, some task can be

¹A great example of math abuse in the popular culture, A. K. Dewdney describes in his book *200 % of Nothing*, subtitled *From “Percentage Pumping” to “Irrational Ratios” — An Eye-Opening Tour through the Twists and Turns of Math Abuse and Innumeracy*, the many logical errors made by the famous Mister Spock of Star Trek. For example, in the original series Spock was always telling Captain Kirk ‘I estimate our chances (of getting out of such-and-such a predicament) to be 0.0162%’ or some such ridiculously small and precise number. And viewers accepted this ludicrous scenario without question because Hollywood had convinced them that Spock was this logical, mathematical, scientific paragon.

done, or some event can happen. Our goal is to count the number of ways we can combine abstract or concrete objects from the given sets they belong to.”

Til noticed that Ila was studying her fingernails as if preparing to count with her fingers. He smiled and went on. “The first of these general principles, the **Addition Principle** (AKA the **Sum Rule**) applies when we are faced with a task to select just one item from two or more sets. The *number of ways* to make the selection is the sum of the sizes of the sets, *provided* the sets are disjoint.”²

“For example, the number of ways to choose one vowel **or** one consonant from the alphabet is $5 + 21 = 26$. ”

Abu had a thought. “I guess we shouldn’t be surprised that this is the same as the number of ways to choose *one letter* from the whole alphabet.” “Correct,” said Til. “The Addition Principle involves an **either-or** choice, where the choices are mutually exclusive.”

To show she was paying attention, Ila interjected, “Let me see if I can guess what another general principle is, or at least what’s it’s called — how about the **Multiplication Principle** (AKA the **Product Rule**).” “You are right on,” said Til, “this principle applies when we are faced with a task to select one member from one set, **and** one member from another set, **and** (continue with arbitrarily many sets). The *number of ways* to do so is the product of the sizes of the sets. Here again we usually stipulate that the sets are disjoint.”

“To extend our earlier example: The number of ways to choose one vowel **and** one consonant from the alphabet is $5 \times 21 = 105$. The Multiplication Principle involves a **both-and** choice.”

Abu had another thought. “I think I can sum up these two principles and relate them to logic,” he said. Til smiled and said, “Go ahead.” Abu showed Til and Ila what he had written: “Adding is like or-ing, and multiplying is like and-ing, to liken + to \vee and \times to \wedge .”

“Excellent summary,” said Til. “But let’s also relate them to games. You remember my wife, Tessie? She likes to have game nights with some friends, who bring some of their own games. Here are a few exercises so you can test your grasp of these basic counting rules — and then apply them to counting other discrete mathematical things as well! Til we meet again, then . . .”

² Which recall, means the sets have an empty intersection; in other words, they have no members in common.

Exercise 237 Tessie and her friends have between them 15 board games, 12 card games, and one word game. If their game nights feature the word game plus one other game — either a board game or a card game — how many different game nights can they have?

ward. *The either/or is a very obvious clue.*

Hint 237 Avoid overthinking this exercise, it is very straightforward.

Answers to these exercises can be found at the end of the chapter.

Exercise 238 Which basic counting principle applies to Exercise 237?

Hint 238 Rhymes with *some!*

Answers to these exercises can be found at the end of the chapter.

Exercise 239 Now Tessie and her friends have acquired two more word games, and have decided to vary the snacks they munch on while playing games. If a game-night configuration consists of a word game and another game and a snack, and they have 7 different types of snacks, how many different game-night configurations are possible now?

Hint 239 This is only slightly harder.

Answers to these exercises can be found at the end of the chapter.

Exercise 240 Which basic counting principle applies to Exercise 239?

Hint 240 How about the other one?

Answers to these exercises can be found at the end of the chapter.

Exercise 241 How should the Sum Rule be adjusted when the two sets being selected from are not disjoint — which means the opposite of the oft-repeated “the sets are disjoint” — they have a nonempty intersection; in other words, they have some members in common.

some insights.

Hint 241 Concreteize two sets and draw their Venn diagram to get

in this case.

ment amounts to subtracting zero — no overlap from two sets if the intersection is empty (zero members in common) plus one if $|A \cap B| = |A| + |B| - |A \cup B|$.

If the total count of members of A and members of B . In other words: plus in common (intersection) minus the overlap of both from the count of each set (say A and B).
For example:
Explanation:
as that since overlap is counted twice if the sets were not disjoint.
sets that overlap are counted twice so the intersection of the two sets is always zero.

Exercise 242 How many different functions are there from a set with 3 elements to a set with 5 elements?

Hint 242 It may be helpful to review the definition of a function.

AT A GLANCE

Exercise 243 How many different one-to-one functions are there from a set with 3 elements to a set with 5 elements?

function (an injection).

Hint 243 It may be helpful to review the definition of a one-to-one

$$2 \times 4 \times 3 = 24.$$

members' 4 choices for the second, and 3 for the third, for a total of four times. So there are 2 choices for the third of the first of 3 cases. In this case, the same third can not be selected more

Exercise 244 How many different onto functions are there from a set with 3 elements to a set with 5 elements?

tion (a surjection).

Hint 244 It may be helpful to review the definition of an onto function.

AT A GLANCE

5.3 Perms and Combs

Tessie greeted Abu and Ila warmly, and chatted briefly as she escorted them to the study. “You just relax for a minute,” she said, “Til will be here shortly — he’s just been helping me prepare for game night tonight!”

Ila smiled, “We know something about your game nights, right Abu?” Abu nodded, with a pleasant grin. He said, “What is the configuration for tonight?”

“Well,” Tessie said, “you know, we haven’t played a dice game in years, so tonight we’re just going to play Yahtzee — to refamiliarize ourselves with it. Til’s refreshing me on the rules and some strategies, so I’ll be well prepared!”

“Sounds like you’ll have the advantage, being coached by your husband!” said Ila.

Tessie smiled and said, “Yes, indeed, we do like to win, you know! So, what are you learning about today?”

Abu said, “Til said we’d be studying ‘perms’ and ‘combs’ — which are quite the abbreviations, I think!”

“Ah yes, the ‘hair salon’ lesson,” chuckled Tessie, and Ila laughed out loud. Then, suddenly serious, she asked, “Tessie, why doesn’t Til like small talk?”

Tessie said, “You know, I think it’s because he’s really not much of a talker, period — except when he’s teaching or tutoring, of course! Mostly he lives in his head, which is most of the time in the clouds!” Or the stars, she thought.

“Interesting,” said Ila. “Abu and I have a running bet that —” Abu interrupted, “Hey, don’t tell her! We don’t want to risk Tessie tipping our hand!”

Tessie laughed. “My lips are sealed, but you don’t have to tell me — I think I can read your minds!” She winked and waved goodbye, as Til was just coming in, so it was time to get on with it. And off they went.

“Mixing the Addition and Multiplication principles in various ways leads to basic counting techniques applicable to many, many counting tasks.” Til paused for a moment, then said, “I want to stress that there’s a hard part and an easy part to this whole counting business. The hard part of the solution to any counting problem is *choosing* the appropriate model and technique. After you do that, *applying* the appropriate model or technique or formula is easy!”

“There are many steps,³ but step one on the path to mastering these techniques is to learn two basic definitions.”

“A **permutation** is an arrangement, where *order matters* — just like with nested quantifiers.”

“A **combination** is a subset, where *order does not matter*, like with sets in general.”

³For example, Twelve seems like a good number!

"Now," continued Til, "whether or not order matters is just one aspect of a counting problem. Another aspect is whether or not you allow *repetition* of elements."⁴

Ila and Abu were both furiously taking notes. They looked up as Til paused, as he gave the distinct impression that he had lost his train of thought — momentarily.

"So," resumed Til, "given a set of n objects, say you want to select r of them. After choosing one object from the set —

- you can either choose it again — in this case *allowing repetition* —
- or not — in that case *disallowing repetition*."

Ila had to break in here, "Let me see if I understand — choosing the right technique means that we have to decide:

1. Does the order of the objects we're selecting matter or not? And
2. Do we allow repeat selections of the objects or not?"

"Exactly!" said Til. "Let's look at the easier case first — order matters, without repetition — disallowing it. First note that in general, how many different possible sequences — arrangements — of objects there are depends on the size — the length — of the sequence. Selection without repetition of r objects from a set of n objects naturally constrains r to be at most n . If more than one object is selected ($r \geq 2$), two different orderings or arrangements of the objects constitute different permutations. Let's break it down into steps:"

1. Choose the first object n ways, and
2. Choose the second object $(n - 1)$ ways — without repetition means there is one fewer object to choose from — and
3. Choose the third object $(n - 2)$ ways — again without repetition means two fewer objects to choose from — and so on —

"Finally choose the r^{th} object $(n - (r - 1))$ ways. By the Multiplication Principle, the number of r -permutations of n things, or said another way, the number of permutations of n things taken r at a time is

$$P(n, r) = n(n - 1)(n - 2) \cdots (n - (r - 1)) = \frac{n!}{(n - r)!}.$$

For example, how many permutations of the letters 'ABCDELMNO' are there?"

"I've got this," said Abu. "There are 9 letters — so $n = 9$ — and we're choosing all 9 — so $r = 9$ also — so it's $P(9, 9) = 9!/(9 - 9)! = 9!/0! = 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1/1 = 362880$ permutations, if I calculated correctly."

"Wait, what?" Ila said. "How is 0 factorial equal to 1?" Til waited, so Abu said, "I remember learning this math factoid way back when, and it just came back to me — but I'm not sure how to explain it!"

"Sounds like a GPAO to me!" said Til. "Work through these exercises, and then we'll talk about scenarios where order doesn't matter."

⁴ Generally speaking, these considerations are germane to either permutations or combinations, but there are subtle variations of permutations and combinations to take into account as well.

Exercise 245 Concretize the equality

$$n(n-1)(n-2)\cdots(n-(r-1)) = n!/(n-r)!.$$

and convince yourself that it really is an equality.

any two positive integer values with $n \geq r$ (e.g., 8 and 5) will do.

Hint 245 Concrete by plugging in actual numbers for n and r —

AT A CONCRETE EXAMPLE

1 · 2 · 3 · 4 · 5 · 6 · 7 · 8 = 48 = 8 · 7 · 6 · 5 · 4 · 3 · 2 · 1

Exercise 246 The English language predominantly uses the Subject-Verb-Object word type ordering; for example, “Gary saw Alice”. Other languages might use Subject-Object-Verb or Object-Verb-Subject, etc. How many different word orderings are possible, not depending on specific words, just their type?

Hint 246 This is a very straightforward exercise in permutations.

AT A CONCRETE EXAMPLE

Exercise 247 Recalling the example of how many are the permutations of the letters ‘ABCDELMNO’, investigate and explain why $0!$ is defined to be equal to 1.

Hint 247 How many ways can you arrange zero objects? Really?

It is important to know that if there are no objects — no arrangement — there is exactly one way to do it: leave them as they are. There are no arrangements if there are no objects at all.

Exercise 248 Of the $9!$ arrangements of the letters ‘ABCDELMNO’, how many contain the string ‘ELM’?

Hint 248 How many arrangements of ABCDENO are there?

AT A GLANCE

Exercise 249 How many TLAs (arrangements of 3 alphabetic letters) are there with repetition allowed?

product being the same number.

Hint 249 Use the Multiplication Principle with each term in the

number repeating is allowed.

number of ways to choose the first letter, times the number of ways to choose the second letter, times the number of ways to choose the third letter. This is called the Product Rule.

Exercise 250 How many different sequences of initials are there for people who either have two initials (first and last) or three initials (first, middle and last)?

the Product Rule by itself.

Hint 250 Use the Sum Rule together with the Product Rule, or else

AT A GLANCE

Exercise 251 How many different sequences can be formed using either two or three letters (A-Z) followed by either two or three digits (0-9)?

For example:

- AB01
- ABC02
- XYZ123
- ...

two numbers.

Hint 251 There are four terms in the sum, each term the product of

$$.00077002 = 2 \cdot 1 \cdot 2 \cdot 3 + 2 \cdot 1 \cdot 2 \cdot 3 + 2 \cdot 1 \cdot 2 \cdot 3 + 2 \cdot 1 \cdot 2 \cdot 3$$

Exercise 252 How many different “words” of the form ccvcv are there where c represents one of the 21 consonants and v represents one of the 5 vowels in the English alphabet? Assume no letter can be repeated, just count words like “tribe” or “blizo” — a non-word but count it anyway — but not something like “trite” or “gligi”, which have repeated letters.

Hint 252 Compute the permutations of vowels and consonants separately, then apply the Product Rule.

AT A GLANCE

5.4 Choosing

Til was pleased at how quickly Ila and Abu finished their exercises.

“Good job! Now to press on. First, we’ll look at choosing, and second, multichoosing, so-called. Choosing is when we’re selecting without repetition and order does not matter, so this is the same as selecting subsets of size k from a set of size n .⁵

“Wait,” said Ila, “don’t you mean subsets of size r ? ”

“No,” Til said, “not only does order not matter, but neither does our choice of variable names! So, for variety, I chose k . As with arrangements or permutations, $0 \leq k \leq n$, and what needs to happen to disregard order is a simple adjustment. Just divide out the number of permutations of the k objects from the number of k -permutations of n objects. Since each k -combination can be arranged $k!$ ways, we have $k!$ permutations for each k -combination. Hence, the number of k -combinations of n things, or, the number of combinations of n things taken k at a time is

$$C(n, k) = \frac{P(n, k)}{P(k, k)} = \frac{n!}{(n - k)!k!} = \binom{n}{k}.$$

The n - k -stacked-vertically-in-parentheses is pronounced *n choose k*.⁵

Abu was excited. “May I offer an example of this?” “Certainly,” said Til, and Abu fairly tripped over his words, “How many ways can we choose a three-letter subset of the nine letters ‘ABCDELMNO’?”

Ila perked up. “You have the solution too, right?” “Yes,” said Abu, “we’re choosing without repetition, and because we want a subset (not a sequence) the order does not matter — so, for example, CDE = CED = DCE = DEC = ECD = EDC — they’re all the same. So $n = 9$ and $k = 3$:”

$$C(9, 3) = \frac{9!}{(9 - 3)!3!} = \frac{9!}{6!3!} = \frac{362880}{720 \cdot 6} = 84.$$

“Great!” said Til. “I think you’re ready to take on some challenging

⁵ The numbers $\binom{n}{k}$ for valid values of n and k are known as the *binomial coefficients*. Learning about these combinatorial numbers is a mind-stretching GPAO. Please take it! When you do, you will find these numbers in the amazing mathematical playground known as *Pascal’s Triangle* — a playground that is chock full of intriguing and challenging ideas.

exercises about choosing, starting with one about a *recurrence relation*⁶. Afterwards, we'll take up *multichoosing*."

Exercise 253 The recurrence relation

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

is normally defined for nonnegative n and k ($0 \leq k \leq n$), and to implement it as a recursive function of two parameters we need to supply some "escape hatch" base cases:

$$\binom{n}{k} = 1 \text{ whenever } k = 0 \text{ or } k = n.$$

A straightforward implementation that expects $n \geq k \geq 0$ falls right out of this definition:

```
def nCk(n, k):
    return 1 if k == 0 or k == n else \
               nCk(n - 1, k) + nCk(n - 1, k - 1)
```

Note that this code will fail with a `RecursionError` if you give it an n that is smaller than k . In addition to fixing this problem, there are good reasons to define other boundary conditions and let the parameters of this basic combinatorial function range over all the integers: $\binom{n}{k} = 0$ whenever $k < 0$ or $k > n$. Explore these reasons, and reimplement `nCk` to handle these other conditions.

⁶ Recursion and recurrence relations go hand in hand.

Given a sequence $\{a_{g(0)}, a_{g(1)}, a_{g(2)}, \dots\}$, a **recurrence relation** (sometimes called a **difference equation**) is an equation that defines the n^{th} term in the sequence as a function of the previous terms: $a_{g(n)} = f(a_{g(0)}, a_{g(1)}, \dots a_{g(n-1)})$.

For example, the Fibonacci (or Lucas) sequence has the "index function" $g(n) = n$: $a_n = a_{n-1} + a_{n-2}$.

Another example: If $g(n) = m^n$, the recurrence $a_n = c \cdot a_{n/m} + b$ describes the running-time of many divide-and-conquer algorithms.

And of course, the two-variable recurrence equation that is the subject of Exercise 253:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

Or presented slightly differently (can you quickly spot the difference?):

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}.$$

Normally, there are infinitely many sequences that satisfy these recurrences. We distinguish them by the *initial conditions* — the values of $a_{g(0)}, a_{g(1)}, a_{g(2)}, \dots$, however many are needed to uniquely identify the sequence. Thus, we specify:

- a_0 and a_1 in the Fibonacci (or Lucas) recurrence;
- $a_{m^0} = a_1$ in the divide-and-conquer recurrence; and
- $\binom{1}{0}$ and $\binom{1}{1}$ in the $\binom{n}{k}$ recurrence.

Hint 253 Start with the links in the side notes

```
nCk(n - 1, k) + nCk(n - 1, k - 1)
if k == 0 or k == n:
    return 1
else:
    return nCk(n - 1, k) + nCk(n - 1, k - 1)
```

Answer 253 Try this:

Exercise 254 Thinking in terms of subsets of size k from a set of size n , convince yourself of the validity of this summation:

$$\sum_{k=0}^n \binom{n}{k} = 2^n.$$

sums

Hint 254 Plug in a few values for n and grind out the left-hand

AT A GLANCE

Exercise 255 Suppose you flip a fair coin 10 times. How many different ways can you get

1. no heads?
2. exactly one head?
3. exactly two heads?
4. exactly r heads?
5. at least two heads?

Hint 255 The last one is the only tricky one.

$$S_{10} = C(10,0) + C(10,1) + C(10,2) + \dots + C(10,10) = 1024 - 1 = 1023.$$

It easier to compute the last one is:

$$1024 - 1 = 1023.$$

$$2. C(10,2) + C(10,3) + \dots + C(10,10) = 49 + 102 + 120 + 120 +$$

$$4. C(10,1) = \frac{10!}{9!1!}.$$

$$3. C(10,2) = 49.$$

$$5. C(10,1) = 10.$$

without repetition.

Answers This is a straightforward application of combinations

Exercise 256 How many bit strings of length 10 contain at least two 1s?

Hint 256 Think of heads as 1s, and tails as 0s.

AT A GLANCE

Exercise 257 There are many kinds of fruit: raspberries, strawberries, blueberries, apples, oranges, bananas, kiwi, papaya, mango, just to name a few. Just choosing five from that list of nine, how many different combinations of fruit salad can you make?

Hint 257 Nine choose five is easy.

$$\text{Answers} \quad \binom{9}{5} = \frac{9!}{5!4!} = \frac{9 \cdot 8 \cdot 7 \cdot 6}{4 \cdot 3 \cdot 2 \cdot 1} = 9 \cdot 7 \cdot 3 = 126.$$

Exercise 258 In the game of Five Crowns there are 116 cards in two combined decks, and in each of 11 hands the number of cards dealt each player increases by one, starting at 3, ending at 13. This implies 11 different numbers of total possible hands from randomly shuffled decks. How many total possible hands are there?

11 numbers.

Hint 258 Treat each card as unique, and the answer is the sum of

AT A GLANCE

Exercise 259 A standard 52-card deck has 4 suits: ♠, ♡, ♢, and ♣, of 13 cards each. A 5-card hand can be chosen in any of $\binom{52}{5}$ ways. Bridge is a game where each of 4 players is dealt 13 cards, which means the whole deck is dealt. A hand has a distribution of suits, which is how many of each suit make up the 13 total. For example, [2, 2, 3, 6] is the suit distribution of the hand [Ace♠, 3♦, 2♣, 9♦, 4♣, 2♣, 7♣, Jack♦, 9♣, 5♣, Queen♦, 10♣, 6♣] — so 2 hearts, 2 diamonds, 3 clubs, and 6 spades.

How many different hands with a [4, 4, 3, 2] suit distribution can be dealt?

How many different suit distributions are there? ([4, 4, 3, 2] is not different than [2, 3, 4, 4], because the order of the suits does not matter.)

$\binom{n}{k}$ formula. Count carefully.

A good approach would be to use the product rule together with the

just that the total number of all 4 suits is 13?

last suit, where what the order of the suits is does not matter,

ber of another, a number of yet another, and a number of the

2. How many choices are there for a number of one suit, a num-

2 of the last suit?

cards to have 4 of one suit, 4 of another, 3 of yet another, and

1. How many choices of suit distribution are there for 13 dealt

counting ways of choosing cards from a standard 52-card deck:

Hint 259 Rephrased, there are two questions to answer, both about

carqş tñmñ fñre J3 olç eacqş smit: $J3 \cdot C(J3' \cdot 4) \cdot C(J3' \cdot 4) \cdot C(J3' \cdot 3) \cdot C(J3' \cdot 3) =$
 $J3$ anq yom mñanlı tñmlı fñreler are olç çwoosinib fñre sñbecipic mñmärer olç
 Δ qeaff carqş. Hencce fñre mñmärer olç mñanqş is fñre bñroqñc olç. $\theta \cdot \Delta \cdot J =$
 carqş anq ouyl $\binom{J}{J} = J$ tñml şelb fo çwoosse fñre tematimib smit lori fñre
 Δ seft olç Δ qeaff carqş $\binom{J}{3} = 3$ tñmlı olç çwoosinib fñre smit lori fñre 3
AñsweR 228 Mñreler are $\binom{J}{3} = 6$ tñmlı olç çwoosinib fñre 6 smit lori fñre

5.5 Multichoosing

“So moving on,” said Til, allowing no time for Abu and Ila to relax, “when repetition is allowed in sets, we no longer have sets, but what we call **multisets**. Since elements can appear more than once in a multiset⁷ we pair up each unique element with a number called its **multiplicity** — how many times it appears. For example, $\{3 \cdot a, 5 \cdot b, 2 \cdot c\}$ represents the multiset $\{a, a, a, b, b, b, b, b, c, c\}$.

Multichoosing is choosing multisets — subsets, combinations — with repetition allowed. And the formula for multichoosing — for counting with repetition allowed — choosing r again, for repetition — the number of r -combinations of n things, or, the number of combinations of n things taken r at a time, is

$$\binom{n}{r} = C(n - 1 + r, r) = \binom{n - 1 + r}{r} = \frac{(n - 1 + r)!}{(n - 1)!r!}.$$

The n - r -in-double-parentheses — that’s pronounced n *multichoose* r .”

Neither Abu nor Ila felt the urge to suggest an example of this unfamiliar beast.

“Let’s take an appetizing example,” said Til. “Suppose we have many different kinds of ice cream to choose from. How many different ways can we choose six scoops of four different kinds? It only matters what *kind* or *type* or *flavor* we choose (e.g., vanilla, chocolate, strawberry, butterscotch) — *not* the individual scoops, nor the order in which we choose them. In other words, given 4 types of ice cream, choose 6 with repetition —”

“So wait,” said Abu, “does that mean I can choose as many scoops of chocolate ice cream as I want?!”

“Or in my case, vanilla!” added Ila.

Til smiled, “Incredible, isn’t it! With repetition means you *can* choose the same flavor over and over again without worrying about running out! We *are* only choosing six total scoops, of course, but obviously we could envision deliciously decadent situations where the number might be 20, or 50, or any number — hypothetically!⁸ But still with only the same four flavors.”

Abu was licking his lips, which Ila found amusing. “Abu,” she said, “did you forget to eat lunch today?!”

“I’m just hoping,” said Abu, “that this 4-flavors-6-scoops example is not just a hypothetical situation!”

Til just smiled. “Sorry, today you’re going to have to settle for some stars and bars!⁹ This is a visualization that applies not just to ice cream but to any objects with different types — cookies, donuts, candy bars — okay, stop drooling you two! Coins or bills of different denominations, books of different genres, etc. Back to our example, given 4 types of objects, we can choose 6 with repetition $C(4 - 1 + 6, 6) = 9!/6!3! = 84$ ways,

⁷ Also called a **bag**.

⁸ And exhausting Tessie’s supply very quickly!

⁹ Not to be confused with stars and stripes, this clever visual aid has *its own Wikipedia entry*.

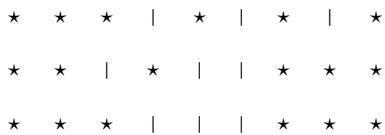


Figure 5.1: Three of 84 ways to choose 6 scoops of 4 different flavors: 3 scoops vanilla, 1 scoop chocolate, 1 scoop strawberry, 1 scoop butterscotch; 2 scoops vanilla, 1 scoop chocolate, 0 scoops strawberry, 3 scoops butterscotch; and 3 scoops vanilla, 0 scoops chocolate, 0 scoops strawberry, 3 scoops butterscotch.

three of which I've depicted in Figure 5.1. There the stars represent the objects, and the bars represent *divisions* between the four spots — the spots are just places to put stars — and it's the *position* of the star — in other words, which spot it's in — that indicates the type of the object.”

Abu was wearing a puzzled look. Ila had a faraway look in her eyes, as if trying to remember ever seeing this type of problem before. “Til,” Abu said, “I'm not quite sure I understand this stars and bars picture, and how the formula works.”

“Let's break it down,” said Til. “Ila, looking at just the first row, how many stars come before the first bar?” “Three,” said Ila, “and I get that's how many scoops of vanilla we're choosing. But I'm with Abu, how does the formula relate to the number of stars and bars.” “I'm getting to that,” said Til. “But first let's be clear on how the placement of the $n - 1$ bars dictates where the stars *can* go:”

- The first spot stars can go is before the first bar;
- the second spot is between the first and second bars;
- the third spot is between the second and third bars;
- ...
- the $(n - 1)^{th}$ spot is between bar number $n - 2$ and bar number $n - 1$.
- the n^{th} (last) spot is after bar number $n - 1$.

“So,” continued Til, “do you see how the total number of bars, $n - 1$, is the right number to specify n spots?” Abu and Ila were both hesitant, but — the light bulb turned on for Ila first. “I see it now! I thought I had seen this before, and I do remember reading about this in some math blog a while back.”

“Nice, Ila” said Til, “Abu, how about you? Do you see it yet?” Ila said, “Let me explain it!” Til eagerly agreed.

“So,” Ila said, “it's $n - 1 + r$ things we're choosing r things from, because we need $n - 1$ bars to show the borders between the n types of things that we're choosing r of. So think of slots — not spots, *slots* — all lined up in a row — 9 of them total, which is $4 - 1 + 6$ in our example. We fill the 9 slots with 3 bars and 6 stars. Once you choose which 6 slots to put the stars in, you only have 3 left — and that's where you put the bars.”

“I see it too!” Abu exclaimed. “But couldn't you also choose the 3 slots to put the bars in first, and then, the 6 stars would have to go in the 6 remaining slots?”

“Yes, indeed!” Til said, “it's symmetrical¹⁰ that way. Here, let's finish off this example with a visualization of all 84 ways to choose our ice cream scoops. See Figure 5.2.”

¹⁰ Seeing the symmetry in the equation $\binom{n}{k} = \binom{n}{n-k}$ and understanding the clever counting argument for its validity will give you a peek into the fascinating realm of *Combinatorial Proof*.

"Did somebody say ice cream?" Tessie said cheerfully, as she glided into the room carrying a well-laden tray. "I overhead Til apologize for his lack of something real to offer you — he didn't know I had an extra stash!"

Til smiled broadly while Abu and Ila thanked Tessie profusely and eagerly helped themselves from the choices she set before them. Tessie exchanged winks with her husband, then said "I know you two now know how to count how many different 3-scoop combinations you could have if you were choosing from, oh, 31 different flavors. Limited to three flavors today, at least you can be glad for the easier counting exercise?!"

"Speaking of exercises," said Til, "enjoy these while you enjoy your sweet treat. I've got to go help Tessie some more, but heads up — next time we're going to tackle BPT — Basic Probability Theory!"

Exercise 260 How many different 3-scoop cups of ice cream are there with just 3 flavors to choose from?

scoops?

Hint 260 weren't we just talking about multichooseing ice cream

AT A GLANCE ANSWER

Exercise 261 How many different 3-scoop cups of ice cream are there with 31 flavors to choose from?

Hint 261 Do you really need a hint here?

$$\binom{3}{3} = C(31 - 1 + 3, 3) = \binom{33}{3} = \frac{30!3!}{33!} = 2430.$$

ANSWER AT A GLANCE

Exercise 262 How many different ways are there to choose ten cookies from the three types: chocolate chip, oatmeal, and peanut butter?

Hint 262 The word 'types' is a dead giveaway.

AT A GLANCE ANSWER

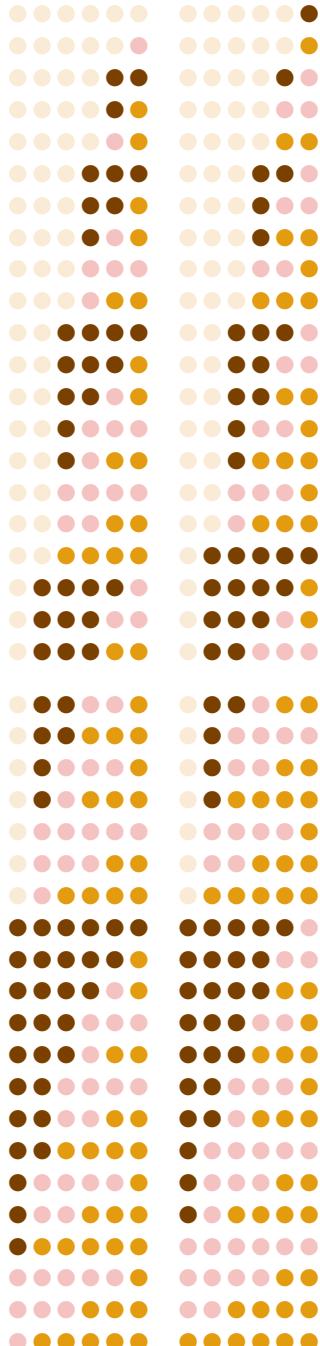


Figure 5.2: $4 \times 21 = 84$ ways to choose 6 scoops of vanilla, chocolate, strawberry, or butterscotch ice cream.

Exercise 263 Find the connection between multichoosing and counting the number of solutions to equations of the form $x_1 + x_2 + \dots + x_m = q$, where q is a positive integer, and all of the x_i variables have non-negative integer values.

Hint 263 Substitute n for m and r for q .

of some flaws)

nonnegative integers. (Zero is itself a nonnegative integer.) Now can you see why this is so? — so it makes sense to use numbers of possible solutions in problems like counting the number of ways to choose r -combinations from n objects in a combinatorial situation. If we substitute n for m and r for q , we get the following result:

Answer 263 Because there are n^r ways to choose r objects from n objects, the number of nonnegative integer solutions to the equation $x_1 + x_2 + \dots + x_n = r$ is n^r .

Exercise 264 Refer to Exercise 259. What if order matters? That is, what if we want to know which number goes with which suit? The sets in the answer to Exercise 259 are reproduced below in table form, with an additional fifth column. What do the numbers in this fifth column count?

0	0	0	13	4
0	0	1	12	12
0	0	2	11	12
0	0	3	10	12
0	0	4	9	12
0	0	5	8	12
0	0	6	7	12
0	1	1	11	12
0	1	2	10	24
0	1	3	9	24
0	1	4	8	24
0	1	5	7	24
0	1	6	6	12
0	2	2	9	12
0	2	3	8	24
0	2	4	7	24
0	2	5	6	24
0	3	3	7	12
0	3	4	6	24
0	3	5	5	12
0	4	4	5	12
1	1	1	10	4
1	1	2	9	12
1	1	3	8	12
1	1	4	7	12
1	1	5	6	12
1	2	2	8	12
1	2	3	7	24
1	2	4	6	24
1	2	5	5	12
1	3	3	6	12
1	3	4	5	24
1	4	4	4	4
2	2	2	7	4
2	2	3	6	12
2	2	4	5	12
2	3	3	5	12
2	3	4	4	12
3	3	3	4	4

Hint 264 Another way to ask the question is, how many solutions in nonnegative integers are there to the equation $h + s + d + c = 13$?

AT A GLANCE

Exercise 265 What is the number of different strings that can be formed by reordering the characters in the word "SUCCESS"?

These are all indistinguishable if you remove the subscripts.

S²UCCES³S¹ S³UCCES¹S² S³UCCES²S¹

S₁UCCES₂S₃ S₁UCCES₃S₂ S₂UCCES₁S₃

subscripts are removed:

Hint 265 The position of the individual S_i 's doesn't matter—only where there are S_i 's does. One way to see this is to label the individual letters with subscripts — S_1, S_2 , and S_3 — and note that different permutations of the letters lead to the same string when the different permutations of the letters lead to the same string when the

Answer 50 The answer is $11111111 = 1 \times 8 \times 2 \times 5 = 40$.

Exercise 266 Make a connection between permutations with indistinguishable objects and combinations with repetitions.

Hint 266 Study the patterns in Exercise 264.

ATLAS see **see also** ANSWER

5.6 Basic Probability Theory

Til started right in. “Do you recall what the frequentist position is with respect to Basic Probability Theory?”¹¹

"Yes!" replied Ila and Abu, in unison.

"Good," said Til. "That's the usual approach, and the one we'll take in this very, very light treatment of this very, very big subject area."

“So a couple of definitions to know,” Til began, “are one, a **probability space** is a finite set of points,¹² each of which represents one possible outcome of an experiment. In addition:

- Each point x is associated with a real number between 0 and 1 called the probability of x .
 - The sum of all the points' probabilities is 1.
 - Given n points, if all are equally likely (a typical assumption), then $1/n$ is the probability of each point.

“And two, an **event** is a *subset* of the points in a probability space. We denote the probability of the event **E** a few different ways — **P(E)**, or **Pr(E)**, or sometimes even **p(E)**. If the points are equally likely, then **P(E)**

¹¹ A good GPAO is to compare basic probability theory with (the “modern understanding of”) Quantum Mechanics’s *version of it*. Watch closely starting at about ten and a half minutes into this fifteen minute talk. Beware that watching the whole video may expose your mind to ideas it has never before considered!

¹² This is the discrete definition, as opposed to the continuous one with infinite points.

is the total number of points in \mathbf{E} divided by the total number of points in the entire probability space.”

“For example,” Til sped along, “the possible outcomes of throwing a single die are {1, 2, 3, 4, 5, 6}, so that set is the probability space of outcomes. Assuming that it’s a *fair* die, each number 1 – 6 is equally likely, so 1/6 is the probability of rolling each number.”

“I know we talked about this,” said Abu, “but let me ask again — is it always the frequentist’s position that *fairness* is inherent and not infused into objects like dice or coins?” Ila joined in, “Yes, I wondered that too, and I also think that the possibility of being bamboozled by tricksters who ‘infuse’ — to use Abu’s word — *unfairness* into dice or coins should change how we look at ‘harmless’ activities like, oh, I don’t know — *gambling!*”

“Excellent points!” said Til. “Gambling is almost certainly a bad idea. I mean, the odds favor the house ever so slightly — by design¹³ — and so if people get lucky often enough to keep at this — yes, I’ll say it — *addictive* behavior, more often than not it lands them into deep trouble.”

Abu was glad he had never felt the need or desire to *gamble*, and from Ila’s comment, and the way she said it, and the shine in her eyes, he could tell she hadn’t either. Til probably too, as he continued without further ado.

“Here’s a coin example. What are the possible outcomes of flipping two *fair* coins? Let’s say our event is seeing one head (H) and one tail (T) — the middle two of our four possibilities, HH, HT, TH, or TT. Since the probability space has 4 outcomes, 1/4 is the probability of each outcome, and the probability of our event one-head-one-tail is thus 2/4 = 1/2.”

Never one to leave it at two, Til went on. “Now let’s ramp up to four. When flipping four coins is the experiment, the probability space quadruples — go ahead, pick out the points and compute the probability of the event ‘seeing two heads and two tails’ by looking at Figure 5.3.”

Abu jumped in first. “I count 6 — Ila, do you see them too?” Ila said, “Yes, so that makes 6/16 = 3/8 the probability of this event, right?” Big smiles from Abu and Til gave her her answer.

“And with that, my twotees, you are ready to tackle some exercises — starting with one dealing with the very definition of probability!”

¹³ No need to unfairly ‘bias’ dice to make the calculations tip towards an advantage.

HHHH	HHHT	HHTH	HHTT
HTHH	HTHT	HTTH	HTTT
THHH	THHT	THTH	THTT
TTHH	TTHT	TTTH	TTTT

Figure 5.3: The 16 possible outcomes when flipping four coins, each having 1/16 probability.

Exercise 267 What improvements or fixes does this probability function need? Replace “Your code goes here” with what should go there.

```
def probability(event, space):
    """A function that takes an event and a space
    and returns a number between 0 and 1.
    """
    result = len(event) / len(space)
    # Your code goes here
    return result
```

Hint 267 Try it with various concrete sets for event and space.

תְּוֹרֶה לְעָגֵל וְלִשְׁבָּרָא :
אַתְּ מִמְּנָא ?

Exercise 268 Generalize the two- and four-fair-coin-flipping experiment to deal with the experiment of flipping $2n$ coins. Find the probability of the event that n will be heads, and n will be tails.

Hint 268 The answer will be a function of n .

אַתְּ מִמְּנָא ?

Exercise 269 Ruth and Ed flip a coin to decide who's going to do the chores. One day Ed complains — he thinks he loses too often. So Ruth says, "OK, this time you flip two coins, and I flip one. If you get more heads than I do, I'll clean the toilets." Ed likes Ruth's display of generosity and says, "You're on!"

What are his chances?

Hint 269 This was adapted from a problem posed by Marilyn vos Savant in her "Ask Marilyn" Parade Magazine column.

L	LL	M ⁰
L	LH	M ¹
L	HL	M ¹
L	HH	M ²
H	LL	M ⁰
H	LH	M ¹
H	HL	M ¹
H	HH	M ²

בְּנִינְתְּךָ אֲתָּה מִמְּנָא ?
בְּנִינְתְּךָ אֲתָּה מִמְּנָא ?
בְּנִינְתְּךָ אֲתָּה מִמְּנָא ?
בְּנִינְתְּךָ אֲתָּה מִמְּנָא ?

Exercise 270 What is the probability of rolling an even number with one roll of a fair die?

Hint 270 This is really easy.

AT A GLANCE

Exercise 271 When you roll two dice, what is the probability that both of them will be sixes?

Hint 271 This is really easy too.

AT A GLANCE

Exercise 272 When you roll two dice, what is the probability that the sum of the numbers (1-6) on their faces will be a multiple of 3 — i.e., 3, 6, 9 or 12?

that order doesn't matter

Hint 272 Assume order matters. Then try it with the assumption

AT A GLANCE

Exercise 273 What is the probability (“Calcprob”) that the event “The first 13 letters will be in alphabetical order” will occur when randomly selecting a permutation of the 26 letters (assumed all uppercase) of the alphabet?

Hint 273 For this you don't need a probability space size of $26!$.

AT A GLANCE

Exercise 274 Calcprob “The first and last letters will be B and Y.”

Hint 274 The number $24!$ will figure into this.

AT A GLANCE

Exercise 275 Calcprob “The letter I will come before both J and K.”

Hint 275 Just consider the relative (not absolute) placements of I , J , and K .

Exercise 276 Calcprob “The letters *O* and *P* will be next to each other.”

Hint 276 A permutation of 25 things is featured here.

ATLAS ATLAS ATLAS ATLAS

Exercise 277 Calcprob “The letters U and V will be separated by at least 23 letters.”

Hint 2.11 There are six different ways this can happen.

Exercise 278 What is the probability that a randomly chosen 3-digit number unambiguously designates a date (month number plus day number paired without punctuation)? E.g., 111 is ambiguous, it could mean January 11th (month 1, day 11) or November 1st (month 11, day 1). Don't count numbers with zeros in the second or third position (why not the first position?) as ambiguous, given the rule that days are never written with a leading zero. E.g., 101 can only mean October 1st, not January 1st. Eliminate from the event space

any invalid numbers, e.g. 345, as neither 3/45 nor 34/5 is a valid date.

Hint 278 Concreteize! List all the 3-digit numbers from 100 to 999 (which means unambiguous) date-numbers by the total ($999 - 100 + 1 = 900$). Carefully count the members of each, and divide the number of valid (that's all of them) and divide them into two sets, valid and invalid.

AT A GLANCE

Exercise 279 A dark room contains two barrels. The first barrel is filled with green marbles, the second is filled with a half-and-half mixture of green and blue marbles. So there's a 100% chance of choosing a green marble from the first barrel, and a 50% chance of choosing either color in the second barrel. You reach into one of the barrels (it's dark so you don't know which one) and select a marble at random. It's green. You select another. It's green too. You select a third, a fourth, a fifth, etc. Green each time. What is the minimum number of marbles you need to select to exceed a probability of 99% that you are picking them out of the all-green barrel? (Note that there are enough marbles so that the answer does not depend on how many marbles are in the second barrel.)

Hint 279 Assume the probability of choosing a green marble in a sure that you are picking marbles out of the all-green barrel? other words, after how many green marbles in a row are you 99+% 1 - $(1/2)^n$? That probability first exceeds 99% when n equals what? In turn probability — that of choosing at least one blue marble — to be row from the half-and-half barrel is $(1/2)^n$. Compute the complemen-

ty probability and subtract it from 1. This will give you the probability of getting at least one blue marble in a row of n picks. Set this equal to 0.01 and solve for n .

5.7 A Dicey Analysis

Tessie was just leaving when Til entered. They did a nonchalant elbow bump, noticed by both Abu and Ila. “We were having a nice chat with your wife about her game night where they played Yahtzee,” said Ila. “Yeah,” said Abu, “she said you helped her figure out the probabilities of getting different rolls of the dice, which I thought was very cool!” Ila

nodded in agreement, and said, “She talked so excitedly about the math principles involved in those calculations, I never would have suspected ...” Ila trailed off, and there was an awkward silence for just a couple of seconds, which Til broke by saying “Tessie loves what I love. And because we love each other, I love what she loves.”

“Sounds very set intersectional and equivalence relational,” said Abu, but he was thinking, Where can I find a wife like Til’s? Why can’t my husband be more like Til?, thought Ila.

Til smiled, and went right on. “We made some simplifying assumptions in our analysis. Instead of three times, we looked at rolling the five dice one time only. Of course we stipulated that each number was equally likely to occur. And we made no distinction between a large straight and a small straight.”

“I take it Yahtzee is what we’re going to talk about today,” said Abu. “Yes,” said Til, “it will be a good exercise in combinatorics and probability. Start by stipulating that there are 7,776 different — unique — results of rolling five dice. That’s six to the fifth power, and that will be the size of the probability space, and thus the denominator of all our fractions. Look at Figure 5.4, which shows the eight possible results of a single roll. I’ll walk you through the first two, then you can study the others on your own for a few minutes.”

“The first is the most rare, hence garners the most scoring points, the event of getting five of a kind. We can easily enumerate the six members of this event: 11111, 22222, 33333, 44444, 55555, and 66666, so $6/7776 = 0.00077$ is the very small probability of getting a Yahtzee. For four of a kind, we could just enumerate all the possibilities, for example, 22221, 33343, 55655, etc., but a better approach would be to choose where the one different die goes — $\binom{5}{1} = 5$ ways — and use the product rule: 5 patterns — aaaab, aaaba, aabaa, abaaa, and baaaa — times 6 choices for a times 5 choices for b equals $5 \cdot 6 \cdot 5 = 150$, and $150/7776 = 0.01929$. Less rare — more probable — hence not worth as much.”

Abu was being nagged by a thought that he couldn’t quite form into words. Ila looked puzzled — perhaps she had the same thought?

Til, giving no indication that he noticed either Abu’s or Ila’s confused looks, just said, “Study Tables 5.1, 5.2, and 5.3, and I’ll be right back.”

1. Five of a kind (a “Yahtzee”)
 2. Four of a kind
 3. Straight
 4. Full house
 5. Three of a kind
 6. Two pair
 7. One pair
 8. Junk
- Figure 5.4: Eight possible results of one five-dice roll.

Table 5.1: Six of eight events

Roll	Examples	Model	Patterns	Event Size
Five of a kind	11111, 66666	Enumerate	aaaaa	Total $= 6$
Four of a kind	11112, 56666	$\exists \binom{5}{1}$ places where the one different die can go — use Product Rule	aaaab aaaba aabaa abaaa baaaa	5 patterns $\times 6$ choices for a $\times 5$ choices for b = 150
Straight	12345, 23456	Sum Rule with 5! permutations of each pattern	abcde bcdef	$120 + 120 = 240$
Full house	11444, 33366	Choose where the pair goes = $\binom{5}{2} = 5!/2!3!$ $= 10$ patterns and use Product Rule	aaabb aabab abaab baaab baaba babaa aabba ababa abbaa	10 patterns $\times 6$ choices for a $\times 5$ choices for b $= 300$
Three of a kind	11123, 45644	Choose where the triple goes = $\binom{5}{3} = 5!/3!2!$ $= 10$ patterns and use Product Rule	aaabc aabac abaac baaac bacaa bcaaa aabca abaca abcaa	10 patterns $\times 6$ choices for a $\times 5$ choices for b $\times 4$ choices for c $= 1200$
Two pair	11223, 45566	Choose where the two pairs go = $5!/2!2!/2 = 15$ patterns and use Product Rule	aabbc ababc abbac aabcb abacb abbca aacbb abcab abcba acabb acbap acbba caabb cabab cabba	15 patterns $\times 6$ choices for a $\times 5$ choices for b $\times 4$ choices for c $= 1800$

Table 5.2: The other two of eight events

Roll	Examples	Model	Patterns	Event Size
One pair	11234, 13566	Two the same and the other three different $= \binom{5}{2}$ $= 5!/2!3!$ $= 10$ patterns and use Product Rule	aabcd abacd abcad abcda baacd bacad bacda bcaad bcada bcdaa	10 patterns $\times 6$ choices for a $\times 5$ choices for b $\times 4$ choices for c $\times 3$ choices for d = 3600
Junk	12346, 12356, 12456, and 13456	Enumerate — no duplicates, not a straight	abcdef abcef abdef acdef where each pattern has 5! permutations	Total = 4 $\times 120$ = 480

Abu was stuck on the sixth event. He said, “Ila, for two pair, I’m not sure they chose the right model. If I calculate the number of permutations of five letters, with two of them indistinguishable, and another two of them indistinguishable, shouldn’t that be $5!/2!2! = 30$ patterns?¹⁴ That is, add these patterns to what’s already there: bbaac, babac, baabc, bbaca, babca, baacb, bbcaa, bacba, bacab, bcbaa, bcaba, bcaab, cbbaa, cbaba, and cbaab?

Ila thought a moment, and said, “I think the reason that’s not correct is because of symmetry. Because 11223 is the same two pair as 22113, etc., 30 is twice too many!”

Abu wasn’t convinced. “But I thought — and this has been bugging me ever since Til started explaining this — that order doesn’t matter when rolling dice.”

Ila said, “But remember doing Exercise 272, where we got the same answer whether or not we assumed order mattered?”

Abu said, “Yes, I remember that. But I still —”

“Besides,” Ila said, “if we add 1800 more members to the ‘two pair’ event, the total won’t add up to 7776.”

“True,” said Abu, “— right now everything adds up nicely to the probability space size.” Then it hit him. “But what if that number is wrong?! What if it’s **not** really supposed to be six to the fifth?”

5.8 A Matter of Order

“Excellent, you two!” Til boomed as he slid into his chair. Ila and Abu, being too absorbed to notice his stealthy return, visibly jumped at that.

¹⁴ Recall Exercise 265?

Roll	Fraction	Float
Yahtzee	6/7776	0.00077
Four of a kind	150/7776	0.01929
Straight	240/7776	0.03086
Full house	300/7776	0.03858
Three of a kind	1200/7776	0.15432
Two pair	1800/7776	0.23148
One pair	3600/7776	0.46296
Junk	480/7776	0.06172
Total	7776/7776	0.99998

Table 5.3: Probability tabulation

"You have noticed the flaw in our analysis, and now you want to correct it?!"

Abu, recovering from being so startled, said "Indeed we did, and indeed we do! Order *does not matter* when rolling these dice, and — I suspected this all along — this game has a far smaller probability space."

"I see it now too," said Ila. "because combinations with repetition are less numerous than permutations with repetition, which is what you were using."

"So what should it be?" asked Til. "Rather than 6^5 ," said Abu, jumping in again, "how about $6 \text{ multichoose } 5$, which is 10 choose 5, which is 252 — double check me on this."

"Bingo!" said Ila. "252 is a lot fewer ways to roll 5 dice than 7,776 — and I'll bet the probability calculations come out differently too!"¹⁵

"Okay, 252 it is," said Til, "and you can start a correct analysis by looking at Tables 5.4 and 5.5 and then doing some more exercises!"

Exercise 280 Find the connection between the strings in Table 5.4 and those in Table 5.5.

¹⁵ From a GPAO they took later, Ila and Abu discovered some web sites purporting to analyze Yahtzee, the authors of which used the same bad models and hence got the probabilities wrong. Apparently, the creators of Yahtzee fell victim to bad reasoning as well.

Hint 280 Remember the multichooseing visual representation?

AT A GLANCE

Exercise 281 Find the connection between the strings in Table 5.5 and the corresponding dice rolls, the first nine of which are: 66666, 56666, 55666, 55566, 55556, 55555, 46666, 45666, and 45566.

Hint 281 These nine rolls are symbolized, respectively, by Yahtzee and TwoPair:

66666 is {6·6·6·6·6} and TwoPair is {2·2·2·2·6}. 56666 is {5·6·6·6·6} and FullHouse is {3·3·3·3·6}. 55666 is {5·5·6·6·6} and FourOfAKind is {4·4·4·4·6}. 55566 is {5·5·5·6·6} and ThreeOfAKind is {3·3·3·5·6}. 55556 is {5·5·5·5·6} and Yahtzee is {5·5·5·5·5}. 55555 is {5·5·5·5·5} and FiveOfAKind is {5·5·5·5·5}.

Exercise 282 Calculate the true probability ("calctrue") of getting a Yahtzee (five of a kind).

56789	46789	45789	45689	45679	45678	36789	35789
35689	35679	35678	34789	34689	34679	34678	34589
34579	34578	34569	34568	34567	26789	25789	25689
25679	25678	24789	24689	24679	24678	24589	24579
24578	24569	24568	24567	23789	23689	23679	23678
23589	23579	23578	23569	23568	23567	23489	23479
23478	23469	23468	23467	23459	23458	23457	23456
16789	15789	15689	15679	15678	14789	14689	14679
14678	14589	14579	14578	14569	14568	14567	13789
13689	13679	13678	13589	13579	13578	13569	13568
13567	13489	13479	13478	13469	13468	13467	13459
13458	13457	13456	12789	12689	12679	12678	12589
12579	12578	12569	12568	12567	12489	12479	12478
12469	12468	12467	12459	12458	12457	12456	12389
12379	12378	12369	12368	12367	12359	12358	12357
12356	12349	12348	12347	12346	12345	06789	05789
05689	05679	05678	04789	04689	04679	04678	04589
04579	04578	04569	04568	04567	03789	03689	03679
03678	03589	03579	03578	03569	03568	03567	03489
03479	03478	03469	03468	03467	03459	03458	03457
03456	02789	02689	02679	02678	02589	02579	02578
02569	02568	02567	02489	02479	02478	02469	02468
02467	02459	02458	02457	02456	02389	02379	02378
02369	02368	02367	02359	02358	02357	02356	02349
02348	02347	02346	02345	01789	01689	01679	01678
01589	01579	01578	01569	01568	01567	01489	01479
01478	01469	01468	01467	01459	01458	01457	01456
01389	01379	01378	01369	01368	01367	01359	01358
01357	01356	01349	01348	01347	01346	01345	01289
01279	01278	01269	01268	01267	01259	01258	01257
01256	01249	01248	01247	01246	01245	01239	01238
01237	01236	01235	01234				

Table 5.4: String representations of subsets of “set(range(10))” with all but those of size five filtered out.

000005	000014	000023	000032	000041	000050	000104	000113
000122	000131	000140	000203	000212	000221	000230	000302
000311	000320	000401	000410	000500	001004	001013	001022
001031	001040	001103	001112	001121	001130	001202	001211
001220	001301	001310	001400	002003	002012	002021	002030
002102	002111	002120	002201	002210	002300	003002	003011
003020	003101	003110	003200	004001	004010	004100	005000
010004	010013	010022	010031	010040	010103	010112	010121
010130	010202	010211	010220	010301	010310	010400	011003
011012	011021	011030	011102	011111	011120	011201	011210
011300	012002	012011	012020	012101	012110	012200	013001
013010	013100	014000	020003	020012	020021	020030	020102
020111	020120	020201	020210	020300	021002	021011	021020
021101	021110	021200	022001	022010	022100	023000	030002
030011	030020	030101	030110	030200	031001	031010	031100
032000	040001	040010	040100	041000	050000	100004	100013
100022	100031	100040	100103	100112	100121	100130	100202
100211	100220	100301	100310	100400	101003	101012	101021
101030	101102	101111	101120	101201	101210	101300	102002
102011	102020	102101	102110	102200	103001	103010	103100
104000	110003	110012	110021	110030	110102	110111	110120
110201	110210	110300	111002	111011	111020	111101	111110
111200	112001	112010	112100	113000	120002	120011	120020
120101	120110	120200	121001	121010	121100	122000	130001
130010	130100	131000	140000	200003	200012	200021	200030
200102	200111	200120	200201	200210	200300	201002	201011
201020	201101	201110	201200	202001	202010	202100	203000
210002	210011	210020	210101	210110	210200	211001	211010
211100	212000	220001	220010	220100	221000	230000	300002
300011	300020	300101	300110	300200	301001	301010	301100
302000	310001	310010	310100	311000	320000	400001	400010
400100	401000	410000	500000				

Table 5.5: Alternate representations of the strings in Table 5.4.

Hint 282 For this and the next seven exercises, count the number of rolls of each of the eight types carefully.

AT [aasa 585 rawsua](#)

Exercise 283 Calctrue four of a kind.

Hint 283 Perhaps writing some code would help?

AT [aasa 585 rawsua](#)

Exercise 284 Calctrue a straight.

Hint 284 Are you starting to get the hang of this?

AT [aasa 585 rawsua](#)

Exercise 285 Calctrue a full house.

Hint 285 It's the same as four of a kind. Verify this.

AT [aasa 585 rawsua](#)

Exercise 286 Calctrue three of a kind.

Hint 286 Carefully count with code.

AT [aasa 585 rawsua](#)

Exercise 287 Calctrue two pair.

Hint 287 It's the same as three of a kind.

AT [aasa 585 rawsua](#)

Exercise 288 Calculate one pair.

Hint 288 It's the same as two pair and three of a kind.

AT A GLANCE ANSWERS

Exercise 289 Calculate junk.

Hint 289 Even more rare than five of a kind!

AT A GLANCE ANSWERS

Exercise 290 Tabulate all eight probabilities and make sure they add up to 1. Based on this, which type of roll should have the highest scoring value?

Hint 290 Which roll has the smallest probability?

AT A GLANCE ANSWERS

5.9 Summary of Terms and Definitions

The **Addition Principle** (AKA the **Sum Rule**): faced with a task to select just one item from two or more sets, the *number of ways* to make the selection is the sum of the sizes of the sets, *provided* the sets are disjoint.

The **Multiplication Principle** (AKA the **Product Rule**): faced with a task to select one member from one set, **and** one member from another set, **and** (continue with arbitrarily many sets), the *number of ways* to do so is the product of the sizes of the sets, *provided* the sets are disjoint.

A **permutation** is an arrangement, where *order matters*. The number of r -permutations of n things is $P(n, r) = n(n - 1)(n - 2)\cdots(n - (r - 1)) = n!/(n - r)!$ — *without repetition*. With repetition allowed, the number of r -permutations of n things is n^r .

A **combination** is a subset, where *order does not matter*. The number of k -combinations of n things is $C(n, k) = \binom{n}{k} = n!/k!(n - k)!$ when *repetition is disallowed*. When *repetition is allowed*, the number of r -combinations of n things is $C(n - 1 + r, r) = (n - 1 + r)!/r!(n - 1)!$

Multisets are unordered collections of elements where an element can be a member more than once. $\{m_1 \cdot x_1, m_2 \cdot x_2, \dots, m_r \cdot x_r\}$ denotes the multiset with element x_1 occurring m_1 times, element x_2 occurring m_2 times, and so on. The m_i values are the elements' **multiplicities**.

Given a sequence $\{a_{g(0)}, a_{g(1)}, a_{g(2)}, \dots\}$, a **recurrence relation** (or **difference equation**) is an equation that defines the n^{th} term in the sequence as a function of the previous terms: $a_{g(n)} = f(a_{g(0)}, a_{g(1)}, \dots, a_{g(n-1)})$. $g(n)$ is the **index** function.

A **probability space** is a finite set of points, each of which represents one possible outcome of an experiment.

- Each point x is associated with a real number between 0 and 1 called the probability of x .
- The sum of all the points' probabilities is 1.
- Given n points, if all are equally likely, then $1/n$ is the probability of each point.

An **event** is a *subset* of the points in a probability space.

- **P(E)** (or **Pr(E)**, or sometimes even **p(E)**) denotes the probability of the event **E**.
- If the points are equally likely, then **P(E)** is the number of points in **E** divided by the number of points in the entire probability space.

6

Number Theory and Practice

6.1 Infinitely Many

“Euclid proved that there are infinitely many primes.” Til said this nonchalantly as he looked up to see his twotees arrive. “Exactly how he did it is something I want you to investigate,¹ but to prepare you for this GPAO, let me introduce the classic method he used to accomplish this amazing feat. Use this as a reference example of **proof by contradiction**, which is that classic method. Do you remember when you looked at the unboundedness of the harmonic series, the sum of the reciprocals of the positive integers?”

“I think so,” said Abu, “but remind me if you don’t mind.” Ila said, “I’ve got it here,” and displayed the formula in Figure 6.1.

“So,” continued Til, “contrary to what we want to prove, suppose that as n gets bigger and bigger, the H_n partial sums converge to S . Let’s see if this supposition gets us into trouble, because if it does, we know that the sum must diverge. H_n converging to S would mean that:

$$\begin{aligned} S &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \dots \\ &= \left(1 + \frac{1}{2}\right) + \left(\frac{1}{3} + \frac{1}{4}\right) + \left(\frac{1}{5} + \frac{1}{6}\right) + \left(\frac{1}{7} + \frac{1}{8}\right) + \dots \\ &> \left(\frac{1}{2} + \frac{1}{2}\right) + \left(\frac{1}{4} + \frac{1}{4}\right) + \left(\frac{1}{6} + \frac{1}{6}\right) + \left(\frac{1}{8} + \frac{1}{8}\right) + \dots \\ &= \frac{2}{2} + \frac{2}{4} + \frac{2}{6} + \frac{2}{8} + \frac{2}{10} + \frac{2}{12} + \frac{2}{14} + \frac{2}{16} + \dots \\ &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \dots \\ &= S. \end{aligned}$$

So $S > S$. Contradiction!”

“Cool,” said Ila. “What good is knowing that particular fact?” Abu almost said something, but noticed the barely perceptible smile on Ila’s face, and he knew she was just goading Til — who gave no indication that he found Ila’s question provocative. “That particular fact,” he said, “is not super important to know, but the technique for proving it is. So compare and contrast this proof with Euclid’s proof — when you have a look at that.”

¹ See Appendix A.

$$H_n = \sum_{i=1}^n \frac{1}{i}.$$

Figure 6.1: The harmonic series partial sum for any given (positive integer) value of n . Diverges to infinity as n tends to infinity.

² In his book *Elementary Number Theory*, mathematician Underwood Dudley wrote, “Prime numbers have always fascinated mathematicians. They appear among the integers seemingly at random, and yet not quite: There seems to be some order or pattern, just a little below the surface, just a little out of reach.”

³ See Appendix B.

⁴ Basic algebra, that is. There’s really a third branch called Algebraic Number Theory (ANT as well) which is definitely in the Advanced realm, using advanced, abstract algebra to study numbers. A great GPAO on ANT is finding *Pi hiding in prime regularities*.

⁵ *Divide* = verb for the process; *division* = noun for the process; *divisible* = able to be divided; *divisibility* = noun for the process-ability.

⁶ Congruence Modulo M as an Equivalence Relation.

“Now on the other hand, the fact that there are infinitely many primes,” Til went on, “means we’ll never run out. Finding a finite list of primes² — up to a given maximum — is a well-studied problem, and there are various types of ‘sieves’³ that will do it when needed.”

6.2 Divisibility Redux

“Before we go into primes, however,” said Til, “I want to distinguish Elementary Number Theory (ENT) from Advanced (or Analytic) Number Theory (ANT). ANT uses the techniques of calculus to analyze whole numbers, whereas ENT uses only arithmetic and algebra.⁴ Recall, we dipped our toes into the waters of what separates the primes from the nonprimes, the notion of divisibility⁵ as it relates to CMM as an ER.”⁶

Abu and Ila glanced at each other. Their eyes handed them the mutual verdict — guilty of recalling these more recent ideas only vaguely.

“Divide and conquer!” Til was enjoying himself. “So the saying goes. The saying has become a popular and powerful algorithm design technique. Equally well-known (but less powerful) is the technique called brute force. We will employ both of these techniques in a study of the integers and their properties — as well as all the mind tools you have acquired through your study of the foundations — sets and logic, functions and relations, even combinatorics and probability — everything you’ve learned can be brought to bear on this study.”

Now Abu and Ila had dagger-eyes for each other — expressing annoyance at allowing themselves to slack off reviewing the concepts they had worked so hard to learn.

Til continued. “It’s pretty easy to see that integers can be added, subtracted and multiplied together, and the result is always an integer. This rather obvious fact is called **closure**. We say that the integers are **closed** under addition, subtraction, and multiplication. But because one integer *divided* by another may or may not result in an integer, the integers are **not** closed under division. Which leads to this definition.”

A **rational number** is the ratio of two integers; i.e., $\frac{p}{q}$ where p and q are integers, with q not zero.

“Please understand,” said Til, “that the rational numbers are worthy of study⁷ in their own right, but now is not the time. So — let our domain of discourse be integers. Unless I explicitly state otherwise hereafter, when I say ‘number’ I mean ‘integer’.”

⁷ As are the real numbers. Both of these sets have a property the integers lack, namely, they are everywhere *dense*. Loosely speaking, that means that between any two rationals or reals you can always find another one. There are surprising differences despite this similarity, which is the subject for another time. The set of rational numbers is named \mathbb{Q} (for Quotient), and the set of real numbers is named \mathbb{R} .

"Here's a quick review," said Til. "Classify the divisibility claims in Figure 6.2 as either true or false — I'll wait!"

Abu and Ila focused, and after a few seconds, reached the same right conclusions.⁸

"Good job!" Til said, but allowed no time for glory-basking. "In the close-but-no-cigar cases, the attempted division fails to produce an integer, being off by one or two. This being off leads to another basic but super, super important fact about divisibility."

If n and d are integers, then there exist integers q and r such that $n = d \cdot q + r, 0 \leq r < d$.

"This fact is called **the Division Theorem**.⁹ Note that q — the quotient — and r — the remainder — are completely determined by n and d . Also note that r can be zero, which is the case when d evenly divides n ."¹⁰

"I'll leave you to ponder this while doing a few exercises," said Til, "and when I come back it'll be **prime** time!"

Exercise 291 There is a redundant word in the statement of the Division Theorem in this Chapter's Summary of Terms, Definitions, and Theorems. What is it, and why is it redundant?

Hint 291 Compare it with the Division Theorem's statement in this section.

differs from $0 \leq r < d$ because d is greater than zero.
because it is not valid if d is zero.
because it is not valid if d is zero.
ANSWER

Exercise 292 There are shortcuts for determining divisibility by 2, 3, 4, 5, 6, 7, 8, 9, 10 and 11. Of course, using the mod function is the easy way, but in case this function were somehow off limits, or only worked for numbers that aren't too big (3 digits max), these shortcuts are helpful.

In the following list N (preceding the colon) is short for A number (a positive integer) is divisible by N if:

- 2: the units digit is 0, 2, 4, 6, or 8.
- 3: the "iterated sum" of the digits is 3, 6, or 9.
- 4: the number formed by the 2 rightmost digits is divisible by 4.
- 5: the units digit is 0 or 5.

⁸ How about you?!

1. $3 | 12$
2. $3 | 13$
3. $13 | 52$
4. $14 | 58$
5. $1 | n$
6. $n | n$
7. $n | 0$
8. $0 | n$

Figure 6.2: Divisibility claims to be classified.

⁹ In an unfortunate misuse of language, it is also called **the Division Algorithm**.

¹⁰ In Python, an easy way to tell when d evenly divides n is to apply "`lambda n,d: n % d == 0`" to them.

- 6: it passes the shortcuts for 2 and 3.
- 7: take the last digit, double it, and subtract it from the rest of the number. If that results in a number divisible by 7 (including 0), then the original number is also divisible by 7. If you find the new number in a list of “small” multiples of 7 (e.g., `range(-98, 100, 7)`) you’re done, otherwise apply the rule again to the new number.
- 8: the number formed by the 3 rightmost digits is divisible by 8.
- 9: the iterated sum of the digits is 9.
- 10: the units digit is a zero. (It must pass shortcuts for 2 and 5).
- 11: add up every other digit into sum1, add up the rest that are not already summed into sum2, and if the difference between sum1 and sum2 is divisible by 11, the original number is.

Write Python functions implementing these divisibility-determining predicates and test them on several large (30-digit or so) numbers.

Hint 292 The rule for 7 is the hardest, potentially most time-consuming one to implement. Also, “iterated sum” means continually adding digits over and over until you reach a single-digit number — for example, given a function `“sumd”` that sums digits once, `sumd(766893741909388135)` yields 97, `sumd(97)` yields 16, and `sumd(16)` yields 7.

AT A GLANCE

Exercise 293 How many divisors does a given number have? Take 36. By trial division (trying successive divisors, starting with 1, which divides everything) we see that 36 is

1 times 36 (that’s 2),
 2 times 18 (another 2),
 3 times 12 (2 more),
 4 times 9 (2 more),
 6 times 6 (1 more — no double counting!)

The total is 9 divisors. What is a general method for finding this total divisor count for any positive number?

Hint 293 Do as the example suggests. I and the number itself make 2 divisors. Then see if 2 divides the number; then 3, then 4, etc., counting 2 divisors for each pair you find. Stop when you get to the square root of the number (else you will double-count divisors). This will find all the divisors as well as their total count. Can just the count be computed without finding all the divisors?

There are lots of ways to see if a number is prime. It's not just prime factorizations of the number.

6.3 Building Blocks

Til came back just in time to catch Ila and Abu discussing their shortcut for divisibility by 11. He saw what they did, made some suggestions, then abruptly changed the subject.

“As we’re still just getting our feet wet with ENT, there is no more *pri-mordial* concept to continue our excursion with than those *multiplicative* building blocks¹¹ — the **atoms** of numbers, as they are aptly called.”

Abu took a turn. “You’re speaking, of course, of the prime numbers, right!”

“Right,” said Til, “and so without further delay, let’s get down to business and look at several ways to say the same thing:”

1. A **prime** is a number (integer) greater than 1 that is divisible *only* by 1 and itself.
2. A **prime** is a positive integer > 1 with exactly 2 divisors, those being 1 and itself.¹²
3. A **nonprime** or **composite** number is a positive integer > 1 that is *not* prime.¹³
4. In first-order logic, $\text{prime}(n) \leftrightarrow \forall x \forall y [(x > 1 \wedge y > 1) \rightarrow xy \neq n]$.¹⁴

Til paused to give Abu and Ila time to digest these definitions. But not for long.

“An interesting fact,” Til said, “that to prove will test your grasp of *reductio ad absurdum*, is that when factoring an integer n , you only need to do trial divisions by integers up to the square root of n .¹⁵

Abu interjected, “Excuse me, I know we just did Exercise 293, but would you please explain what you mean by trial divisions?”

Ila said, “I know, it’s *trying* one divisor after another, until you find one that evenly divides the number. Then you can factor the quotient you get by that division using the same approach. Rinse and repeat. But I’m wondering — could you do it with mapping instead of this very iterative way?”

“You could,” said Til, “but just because you can ...”

“... doesn’t mean you should!” recited Abu and Ila in unison.

“A good GPAO!” said Til. “Research how factoring is done in practice. First, though, master the fundamentals — the foundation supporting any serious practice of number theory. I’ll list them, and then turn you loose to study them with the help of a few notes and exercises I’ve just sent you. This should keep you busy until our next meeting!”

¹¹ Compare these building blocks with the *additive* building blocks, the powers of 2. Sums of 1, 2, 4, 8, etc., in all their combinations, suffice to generate all the non-negative integers. (Zero, of course, is the combination that takes *none* of these powers of 2.)

¹² A noteworthy fact: 2 is the only even prime.

¹³ Hence, it is a number with 3 or more divisors, because 3 is the first number greater than 2, and a number greater than 1 cannot have fewer than two divisors. Note that 1 is neither prime nor composite, but is called a **unit**, as it has but one divisor — itself. See *reasons why* 1 is indeed the *loneliest* number.

¹⁴ Can you write the *negation* of $\text{prime}(n)$ in first-order logic?

¹⁵ Try to convince yourself why this must be the case before reading the rest of this note. If $uv = n$, then u and v cannot both exceed \sqrt{n} . Otherwise $u > \sqrt{n}$ and $v > \sqrt{n}$ would both have to be true. Multiplying these two inequalities produces the contradictory inequality $uv > n$. The upshot is that any composite n is divisible by a prime p that is not greater than \sqrt{n} — a timesaver when you’re trying to factor n .

- The Division Theorem (bedrock solid)
- The Fundamental Theorem of Arithmetic
- Prime Lore (excursions galore)
- The Prime Number Theorem
- The Euclidean GCD Algorithm
- Alternate Base Representation
- Modular Arithmetic (recall CMM?)
- Modular Multiplicative Inverse
- Modular Exponentiation

6.4 Fundamental and Standard

What's so important about primes? Well, for one, the **Fundamental Theorem of Arithmetic** says that every positive integer can be written as the product of prime numbers in essentially one way.

So, $2 \cdot 2 \cdot 2 \cdot 3 \cdot 5 \cdot 7 = 840 = 7 \cdot 5 \cdot 3 \cdot 2 \cdot 2 \cdot 2$ are two ways that we collapse to one — by saying that the order of the factors *does not* matter.

Sometimes uniqueness is guaranteed by using the phrasing “every integer greater than 1 can be written uniquely as a prime or as the product of two or more primes where the prime factors are written in order of nondecreasing size.”¹⁶

For another reason, determining if a given integer is prime is a *key* component to modern cryptology, the study of secret messages — and *crucial* to the current success of the cryptology enterprise is the immense difficulty of factoring large integers into their prime factors.¹⁷ But more on that later.

More prime lore can be perused and pursued¹⁸ if you're interested, but here's a playful smattering:

1. What's the smallest three-digit number that remains prime no matter how its digits are shuffled?
2. Why are the primes 2, 5, 71, 369119, and 415074643 so special?
3. Cicadas of the genus *Magicicada* appear once every 7, 13, or 17 years; whether it's a coincidence or not that these are prime numbers is unknown.
4. Sherk's conjecture. The n^{th} prime, if n is even, can be represented by the addition and subtraction of 1 with all the smaller primes. For example, $13 = 1 + 2 - 3 - 5 + 7 + 11$.
5. You look up one on your own so sexy can be 6 (because the Latin for *six* is *sex*).
6. **Sexy primes** are such that n and $n + 6$ are both prime. The first few such pairs are $\{5, 11\}$, $\{11, 17\}$, and $\{13, 19\}$.

¹⁶ But what about 1? The first phrasing didn't exclude it, why did the second?

¹⁷ “The problem of distinguishing prime numbers from composite numbers and of resolving the latter into their prime factors is known to be one of the most important and useful in arithmetic.

It has engaged the industry and wisdom of ancient and modern geometers to such an extent that it would be superfluous to discuss the problem at length.

Further, the dignity of the science itself seems to require solution of a problem so elegant and so celebrated.”

—Karl Friedrich Gauss (1777-1855)

¹⁸ These factoids come from the book *Prime Numbers, The Most Mysterious Figures in Math* by David Wells.

A prime factorization is in **standard** (or **canonical**) form if its factors are listed in nondecreasing order, with none repeated but instead raised to the appropriate power (at least 1).

For example, the number 1725's factorization¹⁹ is $5 \cdot 23 \cdot 3 \cdot 5$ (**nonstandard**), or $3 \cdot 5^2 \cdot 23$ (**standard**).

A prime factorization is in **complete standard form** (CSF) if its factors are listed as powers (including 0 and 1) of all the primes in order from the smallest (2) up to its largest prime factor. The CSF for 1725 is $2^0 \cdot 3^1 \cdot 5^2 \cdot 7^0 \cdot 11^0 \cdot 13^0 \cdot 17^0 \cdot 19^0 \cdot 23^1$.

Rendering a number's CSF can be done in various ways. One way is to list only the powers of the primes (in spirit all there in order), but not the primes themselves. For example, 1725's CSF is rendered [0, 1, 2, 0, 0, 0, 0, 0, 1]. Jump to 129857981739587, for another example:²⁰

This unwieldy rendition motivates a technique for data compression that is simple and effective.

A run of data is a sequence of consecutive occurrences of the same data value.

A run-length encoding (RLE) replaces runs with a *pair* consisting of the single recurring value together with a count of its occurrences.

For example, the above CSF for 129857981739587 when run-length encoded looks like

`[[6,0], 1, 1, 1, 0, 0, 1, 1, [19,0], 1, 0, 0, 1, [65,0], 1]`
where the 2-element sublists are pairs of counts (first) and values (second). Do you see why there would be no real advantage to encoding a run of only two or three zeros?

If the prime power list is not too long (how long?), the function in Figure 6.3 can be used to reconstruct the number it represents.

Exercise 294 Come up with at least two improvements to the `ppl2n` function of Figure 6.3.

Hmit 294 Do some research into Python modules dealing with primes.

ATLAS AEGIRES

¹⁹ Found thanks to *this great resource*. Try another web site that shows factorizations of relatively small numbers in table form. You can also enjoy visualizing 2-dimensionally and animatedly the factorizations of small numbers, courtesy *this web site*.

²⁰ Found for this much larger number via factordb.com — append a 1 to that number and see the marked difference!

```
def ppl2n(ppl):
    """ppl2n = prime power list to
    number takes ppl, an ordered
    list of prime powers, and
    computes the number by
    raising each prime in
    ascending order to the
    associated power, and
    multiplying these
    exponentiations together.
    """
    primes = [2, 3, 5, 7, 11, 13]
    number = 1
    for prime,pow in zip(primes,ppl):
        number *= prime ** pow
    return number

print(ppl2n([6, 5, 4, 3, 2, 1]))
# prints 5244319080000
```

Figure 6.3: A function to reconstruct a number from its prime power list

Exercise 295 The FTA supports the mapping of the sequence [2, 0, 0, 0, 0, 1] to a two-digit number. What is that number?

Hint 295 Use the ppl2n function.

Answer 295 `ppl2n([2, 0, 0, 0, 0, 1])`

Exercise 296 Write a Python function that will take an RLE like the one for the prime power list of 129857981739587 and revert it to its uncompressed form, testing the result with your enhanced ppl2n function.

Hint 296 This will require some ingenuity, especially if you try the functional rather than the iterative approach.

Answer 296 AT A GLANCE

Exercise 297 Here is another way to RLE:

[1, 1, 0, 1, 0, 1, 3, 1, 0, 1, 3, 1, 0, 1, 3, 1, 5, 1, 0, 1, 5, 1, 3, 1, 0, 1, 3, 1, 5, 1, 5, 1, 0, 1, 5, 1, 3, 1, 0, 1, 5, 1, 3, 1, 5, 1, 7, 1, 3]

Decode this RLE.

Hint 297 The numbers greater than 1 are the zero counts. A one represents one kind of number, a zero represents another kind of number.

Answer 297 `sum([x * y for x, y in zip([1, 0, 1, 0, 1, 3, 1, 0, 1, 3, 1, 0, 1, 3, 1, 5, 1, 0, 1, 5, 1, 3, 1, 0, 1, 3, 1, 5, 1, 5, 1, 0, 1, 5, 1, 3, 1, 0, 1, 5, 1, 3, 1, 5, 1, 7, 1, 3], [1, 1, 0, 1, 0, 1, 3, 1, 0, 1, 3, 1, 0, 1, 3, 1, 5, 1, 0, 1, 5, 1, 3, 1, 0, 1, 3, 1, 5, 1, 5, 1, 0, 1, 5, 1, 3, 1, 0, 1, 5, 1, 5, 1, 0, 1, 5, 1, 3, 1, 0, 1, 5, 1, 3, 1, 5, 1, 7, 1, 3]))`

6.5 Knowns and Unknowns

There are many more prime puzzles for your pondering pleasure provided also courtesy David Wells. A random sample will suffice for now.

Is there a prime between n and $2n$ for every integer $n > 1$? Yes, probably so.

Is there a prime between n^2 and $(n+1)^2$ for every $n > 0$? No one knows.

Many primes are of the form $n^2 + 1$ (e.g., 2, 5, 17, 37, 101, ...). Are there a finite or an infinite number of primes of this form? No one knows.²¹

Is there an even number greater than 2 that is *not* the sum of two primes? No one knows. (That there is not is Goldbach's Conjecture.)

²¹ But mathematicians love banging their heads against these kinds of problems!

Is there a polynomial with integer coefficients that gives only prime values at the integers? The next exercise invites you explore this question.

Exercise 298 Euler's polynomial, x^2+x+41 , computes a prime number for every integer x in the range $-40 \leq x \leq 39$, which is 80 primes total. So, in probability terms, the chances that you will generate a prime by plugging in a random integer in that range are 100%. So no chance about it, it's certain!

Other polynomials of the form $f(x) = x^2 + x + C$, where C is some integer constant, can also generate primes, though maybe not as reliably as Euler's (which is the special case where $C = 41$). And different ranges and constants will vary in the number of primes generated.

Define the sweetness of a given range+constant pair to be a quantity such that the more integers in the range that generate primes through this constant plugged into the polynomial (and taking the absolute value, $|f(x)|$), the higher the sweetness. So the first example, with range+constant $-40 \leq x \leq 39, C = 41$, is seemingly pretty sweet, 80 out of 80, 100%, as already mentioned. But how many of these 80 are duplicates? Looking closer, it's half. Only 40 unique primes are generated. Not so sweet after all.

Find a range and constant that have a sweetness value as high as possible, subject to the arbitrary constraints that the constant will be between negative four million and positive four million. To fix another degree of freedom, which will also save some time, use the constant value of ten thousand for the size of the range (so only its starting value will vary).

Hint 298 Two observations will help you shorten this potentially time-consuming search task. First observation: $x^2 + x$ is always even, so for adding C and having some hope of being prime, C can only be odd. That cuts the possible values for C in half! Second observation: If you limit your search to the best constant for the range 0 to 9999, you can assume you're not going to do much better with a different constant while varying the range. So find the best constant first, then see if varying the range can improve the sweetness. You will meet with success if you follow these hints, and divide and conquer: divide the search for C into non-overlapping intervals; give each member of a large group one interval to test; and finally consolidate the results from all participants.

AT A GLANCE

A famous conjecture (not theorem) states that every even number appears infinitely often as a gap between consecutive primes. This statement has not been proved — nor has it been disproved. Even proving that there are infinitely many twin primes (gap of 2) remains elusive. Indeed, the TPC²² is an active area of mathematical research, both in finding twins and in attempting to prove (or disprove) that they never run out.

As of September 2016 the pair $2996863034895 \cdot 2^{1290000} \pm 1$ holds the twin-prime record. Each is a prime with 388,342 digits!

Exercise 299 Here is a statement in first-order (predicate) logic that there are infinitely many primes:

$$\forall q \exists p \forall x \forall y [p > q \wedge ((x > 1 \wedge y > 1) \rightarrow xy \neq p)].$$

Add to this statement no more than ten symbols to change it to a statement of the TPC, namely, that there is no largest p such that p and $p + 2$ are both prime.

straitjacket forward, and easier than you think.
Hint 299 Time for another refresher on predicate logic? This is very

S).

AT A GLANCE

Exercise 300 Find the smallest primes such that $n^2 < p < (n + 1)^2$ for the first 100 pairs of consecutive squares. That is, find the smallest primes in the ranges (1, 4), (4, 9), (9, 16), etc.

Hint 300 Find them all with the help of a small amount of code.

AT A GLANCE

Exercise 301 How many 7-digit primes have the form $n^2 + 1$? What are they?

enough primes, plus a little code, will reveal the answer.
Hint 301 A working prime sieve or an already-generated list of big

IBM 1108001 utility software for the 3800. AOS is released under IBM 1108001A.

One very special kind of prime is the so-called **Mersenne prime**, which is one of the form $2^p - 1$ where p is prime. Here are a few examples:

- $2^2 - 1 = 3$
 - $2^3 - 1 = 7$
 - $2^5 - 1 = 31$
 - $2^7 - 1 = 127$
 - $2^{11} - 1 = 2047$ (Which is $23 \cdot 89$ — therefore *not* prime.)
 - ...
 - $2^{31} - 1 = 2147483647$ (Prime or not? How do you tell, easily?)
 - ...

The Great Internet Mersenne Prime Search²³ gets credit for discovering the 51st known Mersenne prime, a number with over 24 million digits²⁴ that was discovered in 2018, and that as of this writing holds the record for the largest known prime.²⁵

6.6 Primes Up To N

To reiterate: How many primes are there? Infinity. Not a very useful question/answer. Much more useful would be: How many primes are there up to a given number n ? Revealing in a sense how the primes are distributed among the composites:

The **Prime Number Theorem** states that there is a function that measures the “average” distribution of prime numbers, mysteriously given the name π :

$\pi(n)$ = the number of primes not exceeding (less than or equal to) n .
 $\pi(n) \approx$ the ratio of n and the logarithm of n .

²³ See (and participate at) the *GIMPS* website.

²⁴ Do not even *think* about looking at them all!

²⁵ See also the fascinating topic: *Large Gaps Between Primes.*

Table 6.1: $\pi(n)$ and its approximation.

n	$\pi(n)$	$n/\pi(n)$	$[n/\log n]$
10	4	2.5	5
100	25	4.0	22
1000	168	6.0	145
10,000	1,229	8.1	1,086
100,000	9,592	10.4	8,686
1,000,000	78,498	12.7	72,383
10,000,000	664,579	15.0	620,421
100,000,000	5,761,455	17.4	5,428,682
1,000,000,000	50,847,534	19.7	48,254,943
10,000,000,000	455,052,512	22.0	434,294,482

²⁶ Many interesting things have been said about the PNT. For example: “The discovery of [this theorem] can be traced as far back as Gauss, at age fifteen (around 1792), but the rigorous mathematical proof dates from 1896 and the independent work of C. de la Vallée Poussin and Jacques Hadamard. Here is order extracted from confusion, providing a moral lesson on how individual eccentricities can exist side by side with law and order.” — from *The Mathematical Experience* by Philip J. Davis and Reuben Hersh.

Check it out in Table 6.1, where the exact values of $\pi(n)$ have all been derived by careful counting, and give evidence that $\pi(n) \approx \frac{n}{\log n}$.

More formally, the PNT says²⁶

$$\lim_{n \rightarrow \infty} \frac{\pi(n)}{n / \log n} = 1.$$

So $n / \log n$ is a fairly simple approximation for $\pi(n)$, but it is not especially close. Much closer is an approximation that uses the renowned Riemann zeta function:

$$\zeta(z) = 1 + \frac{1}{2^z} + \frac{1}{3^z} + \frac{1}{4^z} + \dots; \text{ and}$$

$$R(n) = 1 + \sum_{k=1}^{\infty} \frac{1}{k \zeta(k+1)} \frac{(\log n)^k}{k!}.$$

Table 6.2 shows how much better $R(n)$ approximates $\pi(n)$:

Table 6.2: $\pi(n)$ and $R(n)$ compared.

n	$\pi(n)$	$R(n)$
100,000,000	5,761,455	5,761,552
200,000,000	11,078,937	11,079,090
300,000,000	16,252,325	16,252,355
400,000,000	21,336,326	21,336,185
500,000,000	26,355,867	26,355,517
600,000,000	31,324,703	31,324,622
700,000,000	36,252,931	36,252,719
800,000,000	41,146,179	41,146,248
900,000,000	46,009,215	46,009,949
1,000,000,000	50,847,534	50,847,455

6.7 A Dearth of Primes

²⁷ Here’s how *Hardy and Wright* put it:

“The ‘average’ distribution of the primes is very regular; its density shows a steady but slow decrease. The numbers of primes in the first five blocks of 1,000 numbers are

168, 135, 127, 120, 119,

and those in the last five blocks of 1,000 below 10,000,000 are

62, 58, 67, 64, 53.

The last 53 primes are divided into sets of

5, 4, 7, 4, 6, 3, 6, 4, 5, 9

in the ten hundreds of the thousand.

On the other hand the distribution of the primes in detail is extremely irregular.”

The long-haul average distribution of the primes²⁷ belies how chaotic their appearances are in any short range we may scrutinize. The variegated gaps between consecutive primes are evidence of this chaos. Recall the TPC and its generalization, which suggest that there are arbitrarily large gaps between successive primes. Or in other words, for any natural number r , there should exist a sequence of (at least) r consecutive composite numbers. Is this so? (Don’t confuse this with de Polignac’s conjecture — the TPC generalized — which says that *every possible* gap-size (except 1) appears infinitely often.) The following exercises invite you to explore this CCS (Consecutive Composite Sequence) phenomenon.

Exercise 302 What is the first CCS of length 2?

Hints 302 queences (think about it).

Hint 302 Read this as "at least 2". There are no even-length se-

AT a sA 302 TAWsNA

Exercise 303 What is the first CCS of length 3?

Hint 303 It's not hard to find.

Hint 303 All pairs, e.g. if 3 digits must be CCS first 10.

Exercise 304 What is the first CCS of length 4?

Hint 304 Still pretty easy.

AT a sA 402 TAWsNA

Exercise 305 What is the first CCS of length 10?

far:

Hint 305 If your search enters the 3-digit numbers, you've gone too

Hint 305 This problem is similar and requires you to search for all 3-digit numbers.

Exercise 306 Write a Python function that takes a positive integer n and returns a list that is a CCS of length n .

Hint 306 This is asking for some list, not the first one, nor even the

only one. Perhaps a factorial function can help?

AT a sA 306 TAWsNA

Exercise 307 Experiment to see that there are gaps of many (with one exception, even-numbered) sizes between consecutive primes. A gap is computed as the difference between a given prime and the previous prime. E.g., the gap between 11 and 7 is 4, even though there are only 3 nonprimes filling the gap.

Write some Python code to facilitate your experiment. For example, create something like a frequency table suitable for computing some statistics, e.g., the average gap size in the first million primes.

Hint 307 The function `primerrange` in the `sympy` module is your friend. Use a dictionary for the frequency table.

ANSWER 301 See this notebook if you are asked to make a suggestion to

It is an **astounding** fact that there exist *arbitrarily long* sequences of *highly composite*²⁸ consecutive integers! (That means having as many prime factors as we want.) In other words:

Let r and n be positive integers. There exist r consecutive numbers, each divisible by AT LEAST n distinct primes.

You'll need a little more knowledge before being able to fully understand this amazing (if you think about it) theorem. But as persistent investigation will reveal, what this theorem and its proof demonstrate, as do many others, are that these prime building blocks are chock full of surprises!

6.8 Greatest and Least

Switching gears, let's leave primes for a while, and examine a couple of concepts known to ancient Greek mathematicians, and no doubt before then too: Greatest Common Divisors (GCD), and Least Common Multiples (LCM). These, it turns out, have an interesting interplay.

The Greatest Common Divisor, $\text{gcd}(a, b)$, is the largest integer that divides both a and b .

Its counterpart, the **Least Common Multiple**, $\text{lcm}(a, b)$, is the smallest positive integer that is divisible by both a and b .

Integers whose gcd is 1 are called **coprime** (or **relatively prime**, meaning *prime relative to each other*), while **pairwise coprime** (or **pairwise relatively prime**) describes a set of integers every pair of which is **coprime**.

The **Euclidean GCD Algorithm** to find the GCD of two numbers (positive integers) goes like this: Divide the larger number by the smaller, replace the larger by the smaller and the smaller by the remainder of this division, and repeat this process until the remainder is 0. At that point, the smaller number is the greatest common divisor.

This algorithm is implemented by the built-in `gcd` function from the `math` module. I'll walk you through the steps of this algorithm for a simple example, and then have you do it with other pairs of numbers. For each division, we'll keep track of the quotient and remainder. We don't *need* the quotient, but we do need the remainder for the next iteration. See Figure 6.4.

When r is 0 we stop; and as promised, the smaller number ($b = 1$) is the greatest common divisor of our original 97 and 11. Note this is the same as r from the previous step, so we could also say the GCD is the last non-zero remainder of this process. Note too the guaranteed termination. At each step the remainder is necessarily getting smaller, so eventually it must reach 0. We'll see a variation on this theme a little later when we investigate GCDs in another context.

Exercise 308 Trace through the steps of computing ("Trace") $\text{gcd}(57, 43)$.

solidate all steps into one table.

Hint 308 Actually make a table like the ones in Figure 6.4, only con-

AT A GLANCE 308 ANSWERS

Exercise 309 Trace $\text{gcd}(501, 39)$.

Hint 309 Write code to trace the steps.

AT A GLANCE 309 ANSWERS

$$\begin{array}{rcl}
 q & = & 3 \quad (S) \quad + \quad 0 \\
 33 & = & q \quad (Q) \quad + \quad 3 \\
 33 & = & 33 \quad (I) \quad + \quad 0 \\
 201 & = & 33 \quad (IS) \quad + \quad 33 \\
 \hline
 a & = & p \quad (d) \quad + \quad r
 \end{array}$$

ANSWERS 309

Step 1, with $a = 97$ and $b = 11$; the larger and the smaller numbers, respectively:

$$\begin{array}{rcl}
 a & = & b \quad (q) \quad + \quad r \\
 97 & = & 11 \quad (8) \quad + \quad 9
 \end{array}$$

Step 2, b becomes the new a (replace the larger (97) by the smaller (11)), and r becomes the new b (replace the smaller (11) by the remainder (9)):

$$\begin{array}{rcl}
 a & = & b \quad (q) \quad + \quad r \\
 97 & = & 11 \quad (8) \quad + \quad 9 \\
 11 & = & 9 \quad (1) \quad + \quad 2
 \end{array}$$

Step 3, b becomes a , r becomes b :

$$\begin{array}{rcl}
 a & = & b \quad (q) \quad + \quad r \\
 97 & = & 11 \quad (8) \quad + \quad 9 \\
 11 & = & 9 \quad (1) \quad + \quad 2 \\
 9 & = & 2 \quad (4) \quad + \quad 1
 \end{array}$$

Step 4, ditto:

$$\begin{array}{rcl}
 a & = & b \quad (q) \quad + \quad r \\
 97 & = & 11 \quad (8) \quad + \quad 9 \\
 11 & = & 9 \quad (1) \quad + \quad 2 \\
 9 & = & 2 \quad (4) \quad + \quad 1 \\
 2 & = & 1 \quad (2) \quad + \quad 0
 \end{array}$$

Figure 6.4: Stepping through the GCD algorithm.

Exercise 310 Trace $\text{gcd}(765, 110)$.

Hint 310 Did you write some code?

AT A GLANCE

Exercise 311 Trace $\text{gcd}(899, 493)$.

Hint 311 You got this!

AT A GLANCE

$$\begin{array}{rcl}
 28 & = & 89 - 2 \\
 89 & = & 28 + 2 \\
 409 & = & 89 - 4 \\
 463 & = & 409 - 4 \\
 866 & = & 463 - 409 \\
 \hline
 0 & = & 0 + 0
 \end{array}$$

AT A GLANCE

Exercise 312 Predict and then verify, and if you were surprised by what the following code returns, ponder why you were surprised:

```
all([gcd(56,8)==8,gcd(65,15)==5,lcm(5,7)==35,lcm(4,6)==12])
```

Hint 312 Don't forget to import what you can from the math module.

AT A GLANCE

Exercise 313 Verify that for many pairs of positive integers a and b :

```
gcd(a, b) * lcm(a, b) == a * b
```

Does this equality necessarily hold for all pairs of positive integers? Why or why not?

Hint 313 Use assert in a nested loop.

AT A GLANCE

Exercise 314 Given that (a slight expansion of) the CSF of a is $2^3 \cdot 3^2 \cdot 5^1 \cdot 7^0$ and the CSF of b is $2^2 \cdot 3^3 \cdot 5^0 \cdot 7^1$, give the CSFs of the GCD, the LCM, and the product of a and b . Write a Python function that will compute these for any CSFs for a and b , where the representation of the CSFs is a list of (prime, power) tuples, e.g., the CSF for 360 is $[(2, 3), (3, 2), (5, 1), (7, 0)]$.

Appendix A might shed some light.

Exercise 313. If not, the $\text{GCD}(a, b) \cdot \text{LCM}(a, b) = ab$ theorem/proof in Hint 314 Doing this exercise should help you see the answer to Ex-

AT A GLANCE ANSWERS

6.9 Infinite Alternatives

“How ya doin’, Abu?” Ila knew he was in high gear — but they had been at this for several hours, and she was exhausted. Abu studied her for a moment, and said, “I think we both need a break. This is heady stuff, and my head is stuffy. Let’s clear the cobwebs and go at it again later.” “Yes, let’s,” sighed Ila, relieved. Abu hoped he was being sufficiently compassionate, and sensitive to Ila’s fatigue. He was really enjoying learning these topics, and thought he could keep on going and going, but …oh, wow, what a crazy idea Til’s notes greeted them with at their next session!

Infinite Representational Fluency. Sounds crazy, no?²⁹ But representations of integers are endless, because the choice of **base** to use is infinite.

An **Alternate Base Representation** (ABR) for a number n , denoted

$$(a_k a_{k-1} \dots a_1 a_0)_b,$$

is a sequence of “digits” (the numbers a_k down to a_0), where each digit is between 0 (inclusive) and b (exclusive) in magnitude, i.e., $0 \leq a_i < b$. The number b is called the **base** or **radix**. The value or magnitude of n is

$$a_k \cdot b^k + a_{k-1} \cdot b^{k-1} + \dots + a_0 \cdot b^0.$$

Examples:

b	Name for ABR	Example $n =$ (expanding above form making b explicit)
2	binary	$a_6 \cdot 2^6 + a_5 \cdot 2^5 + a_4 \cdot 2^4 + a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0$
3	ternary	$a_5 \cdot 3^5 + a_4 \cdot 3^4 + a_3 \cdot 3^3 + a_2 \cdot 3^2 + a_1 \cdot 3^1 + a_0 \cdot 3^0$
8	octal	$a_4 \cdot 8^4 + a_3 \cdot 8^3 + a_2 \cdot 8^2 + a_1 \cdot 8^1 + a_0 \cdot 8^0$
16	hexadecimal	$a_2 \cdot 16^2 + a_1 \cdot 16^1 + a_0 \cdot 16^0$
64	base-64 (imagine that)	$a_2 \cdot 64^2 + a_1 \cdot 64^1 + a_0 \cdot 64^0$

²⁹ See § 3.7 if you’ve forgotten what *representational fluency* is.

And so forth, ad infinitum.

Here's an **Algorithm for Alternate Base Representations:**

Repeat the three steps until zero appears on the left side of the equation

$$n = \dots + a_6 \cdot b^6 + a_5 \cdot b^5 + a_4 \cdot b^4 + a_3 \cdot b^3 + a_2 \cdot b^2 + a_1 \cdot b^1 + a_0 \cdot b^0.$$

1. The right-most a_i is the remainder of the division by b .
2. Subtract this remainder from n .
3. Divide both sides of the equation by b .

Each iteration yields one a_i coefficient.

Simple algebra guarantees the correctness of this algorithm. For example, take $n = 97$ and $b = 3$.

$$\begin{aligned}
 97 &= a_4 \cdot 3^4 + a_3 \cdot 3^3 + a_2 \cdot 3^2 + a_1 \cdot 3^1 + a_0 \cdot 3^0 \\
 1) \quad 97/3 &= 32, \text{ remainder } = 1 = a_0 \\
 2) \quad 96 &= a_4 \cdot 3^4 + a_3 \cdot 3^3 + a_2 \cdot 3^2 + a_1 \cdot 3^1 \\
 3) \quad 32 &= a_4 \cdot 3^3 + a_3 \cdot 3^2 + a_2 \cdot 3^1 + a_1 \cdot 3^0 \\
 1) \quad 32/3 &= 10, \text{ remainder } = 2 = a_1 \\
 2) \quad 30 &= a_4 \cdot 3^3 + a_3 \cdot 3^2 + a_2 \cdot 3^1 \\
 3) \quad 10 &= a_4 \cdot 3^2 + a_3 \cdot 3^1 + a_2 \cdot 3^0 \\
 1) \quad 10/3 &= 3, \text{ remainder } = 1 = a_2 \\
 2) \quad 9 &= a_4 \cdot 3^2 + a_3 \cdot 3^1 \\
 3) \quad 3 &= a_4 \cdot 3^1 + a_3 \cdot 3^0 \\
 1) \quad 3/3 &= 1, \text{ remainder } = 0 = a_3 \\
 2,3) \quad 3 &= a_4 \cdot 3^1, 1 = a_4 \cdot 3^0 \\
 1) \quad 1/3 &= 0, \text{ remainder } = 1 = a_4 \\
 2,3) \quad 0 &= 0, \text{ so} \\
 97 &= 1 \cdot 3^4 + 0 \cdot 3^3 + 1 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0 \\
 &= 81 + 9 + 6 + 1 = 97, \therefore 97_{10} = 10121_3.
 \end{aligned}$$

For another example, base 27 can be useful in converting "messages" into base 10 numbers. Normally for base x , where $x = 11$ through 36, the digits 0–9 are used and then the alphabetic letters, as many as needed. Base 16 (hexadecimal) needs 6, the letters A through F in this mapping (hex-digit = decimal value):

$$\begin{aligned}
 0 &= 0 & 8 &= 8 \\
 1 &= 1 & 9 &= 9 \\
 2 &= 2 & A &= 10 \\
 3 &= 3 & B &= 11 \\
 4 &= 4 & C &= 12 \\
 5 &= 5 & D &= 13 \\
 6 &= 6 & E &= 14 \\
 7 &= 7 & F &= 15
 \end{aligned}$$

If we followed this scheme for base 27, we'd just continue through Q:

0 = 0	9 = 9	I = 18
1 = 1	A = 10	J = 19
2 = 2	B = 11	K = 20
3 = 3	C = 12	L = 21
4 = 4	D = 13	M = 22
5 = 5	E = 14	N = 23
6 = 6	F = 15	O = 24
7 = 7	G = 16	P = 25
8 = 8	H = 17	Q = 26

However, we could instead use only uppercase alphabetic letters and the @ character (standing in for the space character) in this mapping:

@ = 0	I = 9	R = 18
A = 1	J = 10	S = 19
B = 2	K = 11	T = 20
C = 3	L = 12	U = 21
D = 4	M = 13	V = 22
E = 5	N = 14	W = 23
F = 6	O = 15	X = 24
G = 7	P = 16	Y = 25
H = 8	Q = 17	Z = 26

The advantage of this approach is that strings with only A-Z and @ can represent simple messages to be encoded as numbers and decoded back to strings. Let's do an example. If I want to encode your names as a number I can treat the string "ABU AND ILA" as a base 27 number, with @ replacing the spaces making it have 11 characters total. Then using the above mapping the conversion to base 10 from "ABU@AND@ILA" is $1 \cdot 27^{10} + 2 \cdot 27^9 + 21 \cdot 27^8 + 0 \cdot 27^7 + 1 \cdot 27^6 + 14 \cdot 27^5 + 4 \cdot 27^4 + 0 \cdot 27^3 + 9 \cdot 27^2 + 12 \cdot 27^1 + 1 \cdot 27^0$, which equals $205891132094649 + 15251194969974 + 5931020266101 + 0 + 387420489 + 200884698 + 2125764 + 0 + 6561 + 324 + 1$, which (do the math) equals 227073937768561, which has 15 digits — 4 more than the base 27 representation. Life's tough that way.

Exercise 315 Implement in Python a recursive version of the iterative algorithm sketched above for finding an ABR.

clise 208. Its name is int2bits.

Hint 315 You've already seen a function like this for base 2 in Exercise 208. Its name is int2bits.

commaatae hif e exameyam amsan yrtajdum tifitw.
to sotutitio n a soane a ydne a siidt ee S 212 revewsa

Exercise 316 Try your `abr` function with several values for n and b , e.g.:

```
print(abr(987654321, 127))
# prints
# [3, 101, 20, 87, 86]
```

Why are some ABRs more useful than others?

Hint 316 Think about the difference in magnitude between the first element of the `abr` given above, and the other elements.

AT A GLANCE

Exercise 317 Convert your name (first and last with @ between) as a base 27 number to decimal. Write two Python functions `from27to10` and `to27from10` to do in general the round-trip conversion from base 27 to decimal and back to base 27 again.

Hint 317 Converting `from27to10` can profit from a helper function that reconstructs a number from its abr list.

AT A GLANCE

Exercise 318 **Squareful** numbers are positive integers with some factor besides 1 that is a square (e.g., 4, 9, 16). **Squarefree** numbers are positive integers with **no** factors that are squares. (The one exception is 1 — by convention, 1 is considered squarefree.) For example, 12 is squareful because it has 4 as a factor. 15 is squarefree because none of its factors — 3 and 5 and 15 (remember, don't count 1) — are squares.

Explain how the mapping below of squarefree number to nonnegative integer works:

<i>Squarefree Number</i>	<i>Nonnegative Integer</i>
1	0
2	1
3	2
5	4
6	3
7	8
10	5
11	16
13	32
14	9
15	6
17	64
19	128
21	10
22	17
23	256
26	33
29	512
30	7
31	1024

What is the integer that the squarefree number 42 would map to under this mapping?

Hint 318 Think alternate base representation, base 2, specifically.

AT A GLANCE

6.10 A Combination of Theorems

In addition to what we've seen so far, there are four fundamental theorems of ENT that I'll just state³⁰ and then we'll see how they apply.

³⁰ Again, see Appendix A for proofs.

The Linear Combination Theorem: If $d \mid a$ and $d \mid b$ then $d \mid ax + by$ for any integers x and y .³¹

³¹ $ax + by$ is a *linear* combination because no higher power than 1 is used as an exponent of either x or y (or a or b for that matter).

Bézout's Theorem: If a and b are coprime, then there exist integers x and y such that $ax + by = 1$.

Bézout's Theorem Generalized: If $\gcd(a, b) = g$, then there exist x and y such that $ax + by = g$.

Euclid's First Theorem: If p is prime and $p \mid ab$ then $p \mid a$ or $p \mid b$.

³² According to *Wolfram's MathWorld* his first theorem is called Euclid's Lemma, first appearing in Euclid's Elements, Book VII, as Proposition 30: If for any two integers a and b , $d \mid ab$, where d is relatively prime to a , then $d \mid b$. But this lemma also generalizes as follows: If p is prime and $p \mid abcd\dots$ then $p \mid a$ or $p \mid b$ or $p \mid c$ or $p \mid d$ or $p \mid \dots$ — and this generalization is of prime importance, because the FTA (Fundamental Theorem of Arithmetic) follows from it.

It was his second theorem³² that proved the primes are infinite.

For a few applications, do these exercises:

Exercise 319 You are given the fact that $33(-5) + 28(11) = 143$. Using the rinse and repeat method (demonstrated in the proof of which of the four fundamental theorems?), find a pair of integers x and y such that $33x + 28y = 1$.

Hint 319 You may find helpful a function that computes the linear combination $ax + by$ given those four parameters.

AT A GLANCE

Exercise 320 Find if there is an integer-pair solution to $91x + 11y = 1$ in the range from -49 to 50 .

```
def forsomeforsome(predicate, domainX, domainY):
    return any(map(lambda x: any(map(lambda y: predicate(x, y),
                                      domainY)), domainX))

predicate = # your predicate goes here
domain = range(-49, 51)
print(forsomeforsome(predicate, domain, domain))
```

Hint 320 Think about how a straightforward translation of the equation $91x + 11y = 1$ could be written as a lambda predicate suitable for passing to forsomeforsome in the given code.

AT A GLANCE

Exercise 321 What is the largest unfillable order of chicken nuggets that only come 5 to a box or 7 to a box, with no partial boxes allowed? Note that negative boxes aren't allowed either, but zero boxes are, hence every multiple of 5 or 7 is a fillable number.

Hint 321 Get your hands dirty. Try lots of numbers and look for patterns. Functions from (or like those in) the previous two exercises might be helpful.

AT A GLANCE

6.11 Theorems of Congruence

We saw when looking at equivalence relations the importance of congruence modulo some modulus m (CMM). Congruences take center stage in the play of modular arithmetic, because of how closely they resemble *equality* (which, recall, is the gold-medal ER). Just as equals added to (or subtracted from) equals are equal, equals multiplied by equals are equal, and equals exponentiated are equal, so likewise:

If $a \equiv_m b$ and $c \equiv_m d$, then $a \pm c \equiv_m b \pm d$.

If $a \equiv_m b$ and $c \equiv_m d$, then $a \cdot c \equiv_m b \cdot d$.

If $a \equiv_m b$ then $a^n \equiv_m b^n$ for $n \geq 0$.

These facts follow from the definitions of \equiv_m . Try some concrete examples to get a feel for them, and we'll put them to use when we meet again.

Note that exponentiation follows immediately from multiplication — just multiply the congruence $a \equiv_m b$ by itself n times.

So congruences can be added and subtracted, multiplied and exponentiated. Can they be divided? Like with equality, can a divisor be “canceled” from both sides of a congruence and the result still be congruent?

The answer is — it depends!

The **Congruence Cancellation Theorem** (CCT) says that if $ax \equiv_m ay$ and **if** a and m are coprime, then $x \equiv_m y$.

Exercise 322 Two notable applications of CMM are “casting out nines” (using CM9) and “ISBN error detection” (using CM11). Investigate and report on both.

Hint 322 Where have you seen CM11 before?

AT A **SEE THIS**

6.12 Modular Inverse

Next we'll examine the mouthful that is the **Modular Multiplicative Inverse (MMI)**.

The number a is said to have a modulo m modular multiplicative inverse if there is some integer i such that $a \cdot i \equiv_m 1$.

For example, 9 is an MMI mod 17 of 2 because $2 \cdot 9 = 18 \equiv_{17} 1$.

The **Modular Multiplicative Inverse Existence Theorem** says that i , the mod m MMI of a , exists if and only if a and m are coprime.

Often we want to narrow our focus to just one MMI, the smallest positive one.

The Unique Modular Multiplicative Inverse (TUMMI) is the smallest positive MMI, which puts it between one and the modulus less one.

With MMI knowhow we are able to solve linear congruences of the form $ax \equiv_m b$, for example:

$$3x \equiv_7 4.$$

There are in fact infinitely many solutions: $x = 6 + 7k, k \in \mathbb{Z}$. Each solution is six more than a multiple of seven, but so how do we determine that 6 is a basic ($k = 0$) solution?

With numbers this small we can easily eyeball it. What multiple of 3 is 4 more than a multiple of 7? A brute force search technique is to tabulate the first few multiples of 3 and 7 and look back and forth between the columns:

3	7
6	14
9	21
12	28
15	35
18	42

Sure enough, 18, the 6th multiple of 3, is 4 more than 14, the 2nd multiple of 7. This approach, while valid, will never do for larger values of a , m or b .

Suppose instead of a linear congruence we had a linear equation:

$$3x = 4.$$

Every beginning algebra student knows how to solve for x , by dividing both sides of the equation by 3 to get $x = 4/3$. Not so fast dividing both sides of a congruence by the same number, because we have to stay in the integer domain. 3 is certainly coprime with 7 (every prime is relatively prime to every other prime!), so one condition of the CCT is satisfied, but not so the part about $3x \equiv_7 3y$. What is the equivalent operation to dividing both sides by the same number? Multiplying by the multiplicative inverse, of course, which for 3 is $1/3$, or $1/x$ for any x we might have.³³

³³ For the equation $3x = 4$, dividing by 3 is equivalent to multiplying by $1/3$. $(1/3)3x = (1/3)4$, whence $(3/3)x = 4/3$, whence $(1)x = 4/3$.

For the congruence, the mod 7 MMI of 3 is what we need to multiply both sides by. What is that MMI? What multiple of 3 is 1 more than a multiple of 7? The same table we used before reveals the answer:

3	7
6	14
9	21
12	28
15	35
18	42

$5 \cdot 3 = 2 \cdot 7 + 1$, hence 5 is 3's mod 7 MMI. TUMMI in fact. Likewise, 3 is TUMMI of 5 (mod 7), but we don't care about that. Just use 5 as the multiplier of both sides to get:

(5) $3x \equiv_7 4$, whence (1) $x \equiv_7 20$ (because $(5)3 = 15$ is congruent mod 7 to 1), whence $x \equiv_7 6$ (because 20 is congruent mod 7 to 6).

So finding TUMMI mod m to a is key to solving for x in the generic linear congruence $ax \equiv_m b$. What is a workable method for doing so when eyeballing is too hard?

Bézout's theorem guarantees TUMMI exists when conditions are right,³⁴ but for finding MMIs, what fills the bill is the Extended Euclidean GCD algorithm — and a very rewarding GPAO awaits you to discover what that is and how it works!

Exercise 323 Using the “eyeball it” method described in this section, find a mod-11 MMI of 2. Express the infinitely many of these MMIs in the form $i + mk, k \in \mathbb{Z}$ by filling in the correct values for i and m . Which one of the infinite set of MMIs is TUMMI?

³⁴ Recall $ax + my = 1$? Note, if a and m are *not* coprime, it is a total bust; for example, can $3x + 6y = 1$ be solved for integers x and y ? Never! In rationals, yes ($x = 1/4, y = 1/24$ is one of infinitely many solutions), but not in integers. To convince yourself, divide everything by 3 (the GCD of 3 and 6) and contemplate the impossibility of finding integers x and y such that $x + 2y = 1/3$. (Any linear combination of integers must be an integer.)

Hint 323 The variable names used suggest their values.

ANSWER 333 One answer is six since from times six is six less than nine.

Exercise 324 Show that the positive integers less than 11, except 1 and 10, can be grouped into pairs of integers such that each pair consists of integers that are mod-11 MMIs of each other.

Hint 324 Start by recalling Exercise 323.

ARMED SERVICES **ARMED SERVICES** **ARMED SERVICES**

6.13 Modular Exponentiation

Last, I want to briefly sketch an algorithm that, as algorithms go, is as powerful as it is simple.

Say we want to compute, for example, $893^{2753} \% 4721$. (Why we would want to do something like this will become clear later.)

Doing this the straightforward way — raising 893 to the power 2753 and then reducing that huge number modulo 4721 — is difficult and slow to do by hand. An easy, fast way to do it (by hand and even faster by computer) first expands the exponent in base 2:

$$2753 = 101011000001_2 = 1 + 2^6 + 2^7 + 2^9 + 2^{11} = 1 + 64 + 128 + 512 + 2048.$$

In this table of powers of 2 powers of 893, all intermediate values are reduced modulo 4721.

893^1	\equiv_{4721}	893		
893^2	\equiv_{4721}	4321		
893^4	\equiv_{4721}	4321^2	\equiv_{4721}	4207
893^8	\equiv_{4721}	4207^2	\equiv_{4721}	4541
893^{16}	\equiv_{4721}	4541^2	\equiv_{4721}	4074
893^{32}	\equiv_{4721}	4074^2	\equiv_{4721}	3161
893^{64}	\equiv_{4721}	3161^2	\equiv_{4721}	2285
893^{128}	\equiv_{4721}	2285^2	\equiv_{4721}	4520
893^{256}	\equiv_{4721}	4520^2	\equiv_{4721}	2633
893^{512}	\equiv_{4721}	2633^2	\equiv_{4721}	2261
893^{1024}	\equiv_{4721}	2261^2	\equiv_{4721}	3999
893^{2048}	\equiv_{4721}	3999^2	\equiv_{4721}	1974

Now we can use this table to quickly compute $893^{2753} \% 4721$:

$$\begin{aligned} &\equiv_{4721} 893^{1+64+128+512+2048} \\ &\equiv_{4721} 893^{(1+64+128+512+2048)} \\ &\equiv_{4721} 893^1 \cdot 893^{64} \cdot 893^{128} \cdot 893^{512} \cdot 893^{2048} \\ &\equiv_{4721} 893 \cdot 2285 \cdot 4520 \cdot 2261 \cdot 1974 \\ &\equiv_{4721} 1033 \cdot 4520 \cdot 2261 \cdot 1974 \\ &\equiv_{4721} 91 \cdot 2261 \cdot 1974 \\ &\equiv_{4721} 2748 \cdot 1974 \\ &\equiv_{4721} 123 \\ &= 123 \end{aligned}$$

Exercise 325 Use the demonstrated table method to find $893^{1357} \% 4721$.

Hint 325 You can reuse the table of powers of 2 powers of 893.

AT A GLANCE 326

Exercise 326 This algorithm is implemented in Python’s `pow` function (the arity-3 version). The documentation for this function says: “Some types, such as `ints`, are able to use a more efficient algorithm when invoked using the three argument form.” Investigate by finding the source code (written in C) to this function and compare it with the pseudocode for modular exponentiation that you can also find online.

number-theoretic algorithms.

Hint 326 This is a good exercise in reverse engineering implementations of

AT A GLANCE 326

6.14 Solving Simultaneously

Ila was nervous. She felt she understood most of what they had studied, but just didn't see the point of it all. Abu sat quietly, smiling, and looking deep in thought. Til normally didn't keep them waiting this long — not without having Tessie there to chit-chat with them.

Abu was excited. He knew because she told him that Ila was amazed at not only how quickly he hoovered up number theory, but at how unconcerned he was with any practical use of all this knowledge.

"Today," began Til, even before he sat down, let alone greeting his twotees, "what we're going to tackle is the problem of solving a system of simultaneous congruences."

Ila couldn't help herself. "Hello to you too, Til!" Abu elbowed her in the ribs.

"Hello Ila," said Til, "and feel free to jab Abu back!" Which she promptly did.

"Hey," said Abu, "aren't you going to say hello to me too?!"

"Hello Abu," said Til. "Now can we get on with it?"

"I've sent you this system of three congruences, which I've numbered 1–3. This seeds our knowledge base with facts about an unknown quantity x that we want to solve for. We'll use helper variables t , u , and v ; some basic facts about congruences modulo m (that you two will verify); and some eyeballing of modular inverses — altogether, 31 facts of arithmetic and algebra. We'll record how we arrive at each fact that follows from previous facts."

#	Fact Description	How Arrived At?
1.	$x \equiv_7 4$ (equivalently, $x \% 7 = 4$).	Given congruence.
2.	$x \equiv_{11} 2$ (equivalently, $x \% 11 = 2$).	Given congruence.
3.	$x \equiv_{13} 9$ (equivalently, $x \% 13 = 9$).	Given congruence.
4.	$a \equiv_m b \rightarrow a = mk + b$ for some k .	By definition of \equiv_m .
5.	$a \equiv_m b \rightarrow a + c \equiv_m b + c$.	Easily verified by Ila.
6.	$a \equiv_m b \rightarrow a - c \equiv_m b - c$.	Easily verified by Ila.
7.	$a \equiv_m b \rightarrow ac \equiv_m bc$.	Easily verified by Ila.
8.	$a \equiv_m b \rightarrow a \equiv_m b \% m \wedge a \% m \equiv_m b$.	Easily verified by Abu.
9.	$a \equiv_m b \rightarrow a \equiv_m b + m$.	Easily verified by Abu.
10.	$x = 7t + 4$	Fact 1 used with Fact 4.
11.	$7t + 4 \equiv_{11} 2$	Fact 10 substituted into Fact 2.
12.	$7t \equiv_{11} -2$	Fact 6 used to subtract 4 from both sides of Fact 11.
13.	$7t \equiv_{11} 9$	Fact 9 used to add 11 to the RHS of Fact 12.
14.	Need to find an MMI mod 11 of 7.	Looked for a multiple of 7 that is 1 more than a multiple of 11.
15.	$8 \cdot 7 = 56 = 1 + 5 \cdot 11$.	Eye-balled it. So 8 is TUMMI mod 11 of 7.
16.	$56t \equiv_{11} 72$	Used Fact 7 to multiply both sides of Fact 13 by 8 from Fact 15.
17.	$t \equiv_{11} 6$	Used Fact 8 twice with $m = 11; 56 \% 11 = 1; 72 \% 11 = 6$.
18.	$t = 11u + 6$	Used Fact 4 applied to Fact 17.
19.	$x = 7(11u + 6) + 4 = 77u + 46$	Substituted Fact 18 into Fact 10 and simplified.
20.	$77u + 46 \equiv_{13} 9$	Substituted Fact 19 into Fact 3.
21.	$12u + 7 \equiv_{13} 9$	Used Fact 8 with $m = 13; 77 \% 13 = 12, 46 \% 13 = 7$.
22.	$12u \equiv_{13} 2$	Used Fact 6 to subtract 7 from both sides of Fact 21.
23.	Need to find an MMI mod 13 of 12	Looked for a multiple of 12 that is 1 more than a multiple of 13.
24.	$12 \cdot 12 = 144 = 1 + 11 \cdot 13$	Eye-balled it. So 12 is TUMMI mod 13 of 12.
25.	$u \equiv_{13} 24$	Used Fact 7 to multiply both sides of Fact 22 by 12 from Fact 24.
26.	$u \equiv_{13} 11$	Used Fact 8 with $m = 13; 24 \% 13 = 11$.
27.	$u = 13v + 11$	Used Fact 4 applied to Fact 26.
28.	$x = 77(13v + 11) + 46$	Substituted Fact 27 into Fact 19.
29.	$x = 1001v + 893$	Simplified Fact 28.
30.	$x \equiv_{1001} 893$	Reverse-applied Fact 8 to Fact 29.
31.	$x = 893$ is the simultaneous solution.	Double-checked: $893 \% 7 = 4, 893 \% 11 = 2, 893 \% 13 = 9$.

N	N	N	N	N	N
%	%	%	%	%	%
3	5	8	4	6	
0	0	0	0	0	0
1	1	1	1	1	1
2	2	2	2	2	2
3	0	3	3	3	3
4	1	4	4	0	4
5	2	0	5	1	5
6	0	1	6	2	0
7	1	2	7	3	1
8	2	3	0	0	2
9	0	4	1	1	3
10	1	0	2	2	4
11	2	1	3	3	5
12	0	2	4	0	0
13	1	3	5	1	1
14	2	4	6	2	2
15	0	0	7	3	3
16	1	1	0	0	4
17	2	2	1	1	5
18	0	3	2	2	0
19	1	4	3	3	1
20	2	0	4	0	2
21	0	1	5	1	3
22	1	2	6	2	4
23	2	3	7	3	5
24	0	4	0	0	0

Table 6.3: Repeating Residues

“Is 893 necessarily the only solution?” asked Abu. Til said, “Study Table 6.3 and arrive at your own conclusions. Sorry, but we’re out of time today.”

6.15 Bijectivity Guaranteed?

Ila and Abu were excited. “I see it!” “Me too!”

“Let’s write some code,” said Ila, “and see if Til likes it. How about a function to create Table 6.3, or at least the body of it?”

“Yes!” said Abu. “And make it take any number of numbers to do residues with.”

“Okay,” said Ila, “but ‘doing residues’ is an odd way to put it.”

“You know what I mean,” said Abu, smiling.

Later when they were together, Ila proudly displayed their creation.

“We’ve got some code for you to look at,” said Ila. “Our `repres` function (see Figure 6.5) reproduces Table 6.3, or at least its body.”

“We believe,” said Abu, “that our function shows that a bijection *necessarily exists* between 0 and 14 and *pairs* of numbers.”

“Not just any pairs, right?” said Til. “No,” said Abu. “The pairs are made from modding the number by 3 and by 5, just like the second and third columns of Table 6.3 show.”

“We also noticed,” said Ila, “that those pairs repeat starting at 15 — not coincidentally 3 times 5!”

“But not so for the last two columns,” said Abu. He was excited because he noticed this first. “Those pairs, the residues from modding by 4 and 6, repeat starting at 12 — not the product of 4 and 6, only half that.”

“So we concluded,” said Ila, “that the cyclic patterns of remainders running down these columns repeat at the *least common multiple* of 4 and 6, which is 12. And the second and fourth columns show that using 3 and 8, whose LCM *is* their product, have the pattern not start over until 24.”

“And the difference,” said Til, “between 4 and 6, and 3 and 8 is?”

“It’s that 3 and 8 are coprime,” Abu said, “whereas 4 and 6 have a GCD of 2 — so not coprime.”

```
def repres(rows, *moduli):
    """Repeating residues made
    by modding range(rows+1)
    by each of an arbitrary
    number of moduli, one
    tuple of residues per row.
    """
    for n in range(rows + 1):
        print(n, end=' ')
        residues=list(map(lambda m:\n            n % m, moduli))
        print(*residues)

repres(24, 3, 5, 8, 4, 6)
```

Figure 6.5: A function to print repeating residues.

"We think our `represBG` function," said Ila, "and by the way, it was Abu who suggested these improvements, shows the guaranteed bijectivity more explicitly. See Figure 6.6."

"Very good, you two!" said Til. "So, in math speak we would say $\mathbb{Z}_{pq} \leftrightarrow \mathbb{Z}_p \times \mathbb{Z}_q$. If p and q are coprime positive integers, that is."

"And," said Abu, nearly tripping over his tongue, "you wouldn't even need the 'if' if you replaced pq with $\text{lcm}(p, q)$, right?"

"Yes!" said Til. "Are you seeing now the answer to your question, Abu, that I left you hanging with last time?"

Abu nodded vigorously, and gestured to Ila, who jumped in. "Yes, we are! Here's part of the table we printed using 7, 11, and 13 as moduli (see Table 6.4). We used the original version of our function, and sure enough, 893 corresponds to the residues 4, 2, and 9, just like we saw last time!"

Exercise 327 Reimplement `represBG` more in the functional programming style.

Hint 327 Use mapping instead of looping.

comədətəs nəfəs türkəmənək dərisiñər.

Answer 327 See this notebook for how to implement a solution to

```
from math import gcd, prod

def lcm(*numbers):
    return prod(numbers) // \
        gcd(*numbers)

def represBG(*moduli):
    """Repeating residues with
    bijectivity guaranteed
    made by modding
    range(lcm(*moduli))
    by each of an arbitrary
    number of moduli, one
    tuple of residues per row.

    """
    for n in range(lcm(*moduli)):
        print(n, end=' <-> (')
        residues=list(map(lambda m:\n            n % m, moduli))
        print(*residues, sep=', ', \
            end=')\n')
```

```
represBG(3, 5)
# prints
# 0 <-> (0, 0)
# 1 <-> (1, 1)
# 2 <-> (2, 2)
# 3 <-> (0, 3)
# 4 <-> (1, 4)
# 5 <-> (2, 0)
# 6 <-> (0, 1)
# 7 <-> (1, 2)
# 8 <-> (2, 3)
# 9 <-> (0, 4)
# 10 <-> (1, 0)
# 11 <-> (2, 1)
# 12 <-> (0, 2)
# 13 <-> (1, 3)
# 14 <-> (2, 4)
```

Figure 6.6: Showing an explicit bijection.

6.16 A Most Beguiling Theorem

³⁵ See Appendix C for more evidence of this, which is also to follow up on Til's mention of this **astounding** fact at the end of § 6.7.

³⁶ If the divisors are pairwise coprime, their LCM is their product, which is slightly easier to compute, which is why the CRT is usually presented in this more restrictive less arbitrary form:
Given a system of simultaneous congruences

$$\begin{aligned} x &\equiv_{m_1} a_1 \\ x &\equiv_{m_2} a_2 \\ &\vdots \\ x &\equiv_{m_n} a_n \end{aligned}$$

if the moduli m_k are pairwise relatively prime positive integers, then there exists a unique solution x modulo $m = m_1 m_2 \dots m_n$.

A proof by construction is typical — show how to find the unique solution, and prove the construction is correct.

N	N	N	N
%	%	%	%
7	11	13	
0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
6	6	6	6
7	0	7	7
8	1	8	8
9	2	9	9
.	.	.	.
.	.	.	.
.	.	.	.
893	4	2	9
.	.	.	.
.	.	.	.
991	4	1	3
992	5	2	4
993	6	3	5
994	0	4	6
995	1	5	7
996	2	6	8
997	3	7	9
998	4	8	10
999	5	9	11
1000	6	10	12

Table 6.4: Slices of a much bigger table.

Til was enjoying Abu's and Ila's excitement. But it was high time to reveal the theorem that intrigued him no end.³⁵ He said, "We've been investigating a phenomenon of modular arithmetic known to the first century Chinese, and probably before that too. But the name has stuck to them: the Chinese Remainder Theorem. In a nutshell:"

"The **Chinese Remainder Theorem**, or CRT, says that given the remainders from dividing an integer n by an *arbitrary* set of divisors,³⁶ the remainder when n is divided by the *least common multiple* of those divisors is *uniquely* determinable."

"Let's see how this works with our favorite example, which I'll reproduce here, in both congruence form and the equivalent modulus form:"

$$\begin{aligned} x &\equiv_7 4 & \leftrightarrow & x \% 7 = 4 \\ x &\equiv_{11} 2 & \leftrightarrow & x \% 11 = 2 \\ x &\equiv_{13} 9 & \leftrightarrow & x \% 13 = 9 \end{aligned}$$

"Now hold on tight, because next I'm collapsing a lot of information that we carefully uncovered last time — for example, the MMIs y_1, y_2 , and y_3 ."

$$\begin{aligned} 143y_1 &\equiv_7 1 \rightarrow y_1 = -2 \\ 91y_2 &\equiv_{11} 1 \rightarrow y_2 = 4 \\ 77y_3 &\equiv_{13} 1 \rightarrow y_3 = -1 \end{aligned}$$

"So," continued Til, "here's what we end up with:"

$$\begin{aligned} x &= 4(143)(-2) + 2(91)(4) + 9(77)(-1) \\ &= 4(-286) + 2(364) + 9(-77) \\ &= -1144 + 728 + -693 \\ &= -1109 \\ &\equiv 893 \end{aligned}$$

"And of course, that last \equiv should be \equiv_{1001} , you see?" said Til.

"Whoa, back up," said Abu. "Where did those numbers 143, 91, and 77 come from?"

6.17 Linear Systems

“That’s a great question!” said Til. “You two will discover the answer through doing some exercises, which should convince you that this construction works. As a bonus, if you’ve studied any linear algebra, the embedded zero-one matrix I’m showing in Figure 6.7 may ring a bell.”

“After doing these exercises, you’ll be ready for action — putting all this theory into practice!” said Til. “There are two main applications — computer arithmetic with large numbers, and the RSA Cryptosystem — both worth exploring. I hope you agree when you return from your brief excursions into this vast terrain!”

Exercise 328 Replace the “# . . .” line with three lines, setting the values of y_1 , y_2 and y_3 . Write a helper function to compute these values.

```
r1 = 4
r2 = 2
r3 = 9
m1 = 7
m2 = 11
m3 = 13
m = m1 * m2 * m3
o1 = m // m1
o2 = m // m2
o3 = m // m3
# ...
roys = (r1 * o1 * y1) + (r2 * o2 * y2) + (r3 * o3 * y3)
solution = roys % m
print(solution)
```

$$\begin{array}{ccccccc} -286 & \equiv_7 & 1 & \equiv_{11} & 0 & \equiv_{13} & 0 \\ 364 & \equiv_7 & 0 & \equiv_{11} & 1 & \equiv_{13} & 0 \\ -77 & \equiv_7 & 0 & \equiv_{11} & 0 & \equiv_{13} & 1 \end{array}$$

Figure 6.7: Embedded “identity matrix” in congruence form.

Hint 328 The y_i 's are the mod m_i MIs of the o_i 's.

ANSWER

Exercise 329 What is a 3-digit positive simultaneous solution to the following system of linear congruences?

$$\begin{aligned} x &\equiv_7 3 \\ x &\equiv_{11} 5 \\ x &\equiv_{13} 4 \end{aligned}$$

Hint 329 Did you do Exercise 328? If you make a general-purpose “Simultaneous System of Congruences solver function” that passes instead of hardwiring the r_1, r_2, r_3 , and the m_1, m_2 , and m_3 values, this and the following three Exercises will be a cinch.

AT A GLANCE ANSWER

Exercise 330 What is a 3-digit positive simultaneous solution to the following system of linear congruences?

$$\begin{aligned}x &\equiv_7 3 \\x &\equiv_{11} 2 \\x &\equiv_{13} 1\end{aligned}$$

Hint 330 See the hint for Exercise 329.

AT A GLANCE ANSWER

Exercise 331 What is a 3-digit positive simultaneous solution to the following system of linear congruences?

$$\begin{aligned}x &\equiv_7 2 \\x &\equiv_{11} 3 \\x &\equiv_{13} 4\end{aligned}$$

Hint 331 See the hint for Exercise 329.

AT A GLANCE ANSWER

Exercise 332 What is a 3-digit positive simultaneous solution to the following system of linear congruences?

$$\begin{aligned}x &\equiv_7 0 \\x &\equiv_{11} 8 \\x &\equiv_{13} 12\end{aligned}$$

Hint 332 See the hint for Exercise 329.

AT A GLANCE ANSWER

Exercise 333 Are there any 3-digit ($d_1d_2d_3$) positive simultaneous solutions to a system of linear congruences

$$\begin{aligned}x &\equiv_7 r_1 \\x &\equiv_{11} r_2 \\x &\equiv_{13} r_3\end{aligned}$$

with the property that $d_1 = r_1, d_2 = r_2$, and $d_3 = r_3$?

digit numbers) will reveal the answer.

Hint 333 A simple search through the range (100, 1000) (all 3-

ANSWER 333 No, there are no solutions.

Exercise 334 These eight small-number triples are not random:

- (1, 1, 3), (1, 1, 4),
- (1, 4, 3), (1, 4, 4),
- (2, 1, 3), (2, 1, 4),
- (2, 4, 3), (2, 4, 4).

They have something to do with the product of the first three odd primes and the fourth power of two.

Find the connection.

13 to ones more appropriate for this problem.

would be helpful here, as long as you change the moduli from 7, 11,

Use of the solver function mentioned in the hint for Exercise 329.

Hint 334 This problem involves the Chinese Remainder Theorem.

AT A GLANCE

6.18 Residue Number Systems

The Chinese Remainder Theorem makes possible the use of a *residue number system* to do computer arithmetic with “large” integers, where largeness is relative. Large in practice means numbers with hundreds or thousands of digits. For now, large means no greater than 1000. Normally, arithmetic with numbers in this range would be trivial, but it’s instructive to view the normal arithmetic operations of addition, subtraction, and multiplication as off limits because they’re too expensive. Your challenge is to find ways to avoid using them.³⁷

³⁷ Exponentially more expensive are the division and modulus operations — especially avoid these!

Exercise 335 Using the code provided in this notebook as a guide and a starting point, finish implementing the `toRNS`, `fromRNS`, `add2` and `mul2` functions. Test your code to see if it can correctly add and multiply one-, two-, and three-digit nonnegative integers, converting to `rns`, doing the `rns` operations, then converting back from `rns`.

Avoid at all costs the use of `+`, `-`, `*`, `/`, or `%`. No fudging by implementing `+`, say, by more primitive bit operations either.

Hint 335 You will benefit greatly if you write helper functions and use good functional programming style throughout.

contribute with the exercises and answers here!

Answer 332 See this notebook if you already have a solution to

6.19 The RSA Cryptosystem

³⁸ This description of RSA claims to be a completely self-contained exposition of the number theory behind RSA. From the authors' abstract: "This is our proof of the RSA algorithm. There are probably more elegant and succinct ways of doing this. We have tried to explain every step in terms of elementary number theory and avoid the 'clearly it follows...' technique favoured by many text books." Though it's not mentioned by name, a consequence of the Chinese Remainder Theorem is used in their proof. Finding where is a test of your understanding.

³⁹ Not to be confused with *Fermat's Last Theorem*, which is a theorem that is easier to state and understand what it's saying than Fermat's *Little Theorem*, but much, much harder to prove!

"I did a little research on *public-key cryptography*," Ila began, "and I've got some hint of a handle on RSA."³⁸

Abu said, "The initials honoring this cryptosystem's brilliant creators — let's see — R = Rivest (Ron), S = Shamir (Adi), and A = Adleman (Leonard)."

"Who cares?" Ila said. "I don't have time for a lesson on who, where or when, I just want what and how." Abu countered, "What about why? — Don't you want to know *why* RSA works, not just *how* it works?"

"From what I gather, to understand *why* RSA works we need to understand both the CRT and the FLT," Ila replied. "I think I understand the Chinese Remainder Theorem pretty well, but Fermat's *Little Theorem*³⁹ is another story." Abu agreed.

"Let's come back to that question," said Ila. "For right now — as I understand it, RSA exponentiates a number (the 'message' — which is the base) to a power and then reduces it modulo the product of two large primes. And thankfully, we can use the arity-3 `pow` function for that. The number we get is a 'scrambling' of the original number."

"That's how I understand it too," said Abu. "And the scrambled number can only be unscrambled and the original number recovered by modular exponentiation using the modular inverse of the power."

"And what boggles my mind," Ila continued, "is that the whole security of this scheme is that the encryption key's inverse needed to decrypt encrypted messages is determined by the two large primes, which have to be kept secret. Without those two primes, it's a very huge haystack to search for a very tiny needle. Searching for the inverse, that is. I suppose you could instead look for the two primes by attempting to factor the

modulus, but — good luck!”

Abu said, “The modulus being the product of those primes is one part of the public key that is published to the world. The encryption exponent being the other part. And the decryption exponent is the deep dark secret key!”

“Right!” said Ila. “Remember the question, have you ever met an employee of your bank in a dark alley to exchange secret keys?”

“Yes,” said Abu, “it would be insane if you had to do that for every company you wanted to do business on the web with.”

“I know!” Ila gleamed. “But that’s when I first caught a glimpse of the awesomeness of RSA. By the way, I made this table so I could keep all the RSA players and their roles and relationships straight.”

Who	What	Roles/Relationships
m	message (plaintext)	decrypted $c : m = c^d \% n$
c	ciphertext	encrypted $m : c = m^e \% n$
p	a large prime	where “large” is relative
q	another large prime	$\neq p$, but with the same # of digits (more or less)
n	the modulus	the product of p and q : must be bigger than m
t	the totient	the product of $p - 1$ and $q - 1$
e	the encryption exponent	must be coprime to t , typically $65537 = 2^{16} + 1$
d	the decryption exponent	TUMMI (mod t) of e

“Ila, this is cool!” exclaimed Abu. “Thanks for sharing it!” he said to Ila’s delight. They both silently studied the table for several seconds.

“Abu, about the totient —” Ila ventured hesitantly — she knew Abu would be pleased if she showed she cared about the *why*, not just the *what*, but she didn’t want him to think she was insincere. “Why is it p minus one times q minus one?” she finally said.

“Well,” said Abu, “I have it here in my notes that the totient is a function that counts the number of totatives a given number has.”

“And what, pray tell, is a totative?” Ila sighed. She had no recollection of seeing an explanation of any such thing in her research.

Abu chuckled. “A totative of a number n is a number smaller than n that is coprime to n . For example, the numbers 1, 3, 7, and 9 are coprime to 10, but 2, 4, 5, 6, and 8 aren’t, making four its total totative count. So the totient of 10 is 4.”

“Oh right,” said Ila, “and since every number smaller than a prime is one of its totatives, the totient of a prime is just that prime minus one.”

“Don’t forget the linking piece,” Abu said. “When the factors are coprime, the totient is a multiplicative function, meaning that the totient of a product of two factors, say $\text{totient}(p * q)$ is the product of the totients of the factors: $\text{totient}(p) * \text{totient}(q)$. Which when p and q are prime is just $(p - 1) * (q - 1)$.”

Ila said, “Okay! — here goes — my very first simple RSA experiment.”

```

m = 2
p = 3
q = 11
n = p * q
t = (p - 1) * (q - 1)
e = 3
d = 7
c = pow(m, e, n)
m_again = pow(c, d, n)
# print(m_again, m == m_again)
# 2 True

```

Abu took it and ran with it. “Here’s mine. I chose different primes, and my message is bigger (by one!) than yours, but isn’t that weird how my e and d just happened to turn out to be the same number?”

```

m = 3
p = 5
q = 7
n = p * q
t = (p - 1) * (q - 1)
e = 5
d = 5
c = pow(m, e, n)
m_again = pow(c, d, n)
# print(m_again, m == m_again)
# 3 True

```

“Well,” said Ila, “I suppose there’s nothing that says they can’t be the same. Except that would defeat the whole idea of keeping one private and making the other public!”

“These are just toy examples anyway,” said Abu. “But they’re enough to get a feel for it!”

Exercise 336 Experiment with the RSA Cryptosystem using the code found in [this notebook](#). How “big” a message can you successfully round-trip?

Hint 336 Pay attention to the difference between encoding and decoding.

AT A RSA 336 ANSWER

6.20 Summary of Terms, Definitions, and Theorems

A **rational number** is the ratio of two integers; i.e., $\frac{p}{q}$ where p and q are integers, with q not zero.

The **Division Theorem** says that if n is an integer and d is a positive integer, then $\exists q \exists r$ such that $n = d \cdot q + r, 0 \leq r < d$. Often wrongly referred to as the **Division Algorithm**.

A **prime** is an integer greater than 1 that is divisible *only* by 1 and itself.

A **prime** is a positive integer > 1 with exactly 2 divisors, those being 1 and itself.

A **nonprime** or **composite** number is a positive integer > 1 that is *not* prime.

In first-order logic, $\text{prime}(n) \leftrightarrow \forall x \forall y [(x > 1 \wedge y > 1) \rightarrow xy \neq n]$.

The **Fundamental Theorem of Arithmetic**: every positive integer (including 1) can be written as the product of primes in essentially one way.

A prime factorization is in **standard** (or **canonical**) **form** if its factors are listed in nondecreasing order, with none repeated but instead raised to the appropriate power (at least 1).

A prime factorization is in **complete standard form (CSF)** if its factors are listed as powers (including 0 and 1) of all the primes in order from the smallest (2) up to its largest prime factor.

A **run of data** is a sequence of consecutive occurrences of the same data value.

A **run-length encoding** (RLE) replaces runs with a *pair* consisting of the single recurring value together with a count of its occurrences.

The **Prime Number Theorem** states that there is a function that measures the “average” distribution of prime numbers:

$\pi(n)$ = the number of primes not exceeding (less than or equal to) n . $\pi(n) \approx \frac{n}{\log n}$.

The **Greatest Common Divisor**, $\gcd(a, b)$, is the largest integer that *divides* both a and b .

The **Least Common Multiple**, $\text{lcm}(a, b)$, is the smallest positive integer that *is divisible by* both a and b .

Integers whose \gcd is 1 are called **coprime** (or **relatively prime**, meaning *prime relative to each other*).

Integers are **pairwise coprime** (or **pairwise relatively prime**) if every pair of them is **coprime**.

The **Euclidean GCD Algorithm** finds the GCD of two numbers (positive integers). Divide the larger number by the smaller, replace the larger by the smaller and the smaller by the remainder of this division, and repeat this process until the remainder is 0. At that point, the smaller number is the greatest common divisor.

An **Alternate Base Representation** (ABR) for a number n , denoted

$$(a_k a_{k-1} \dots a_1 a_0)_b,$$

is a sequence of “digits” (the numbers a_k down to a_0), where each digit is between 0 (inclusive) and b (exclusive) in magnitude, i.e., $0 \leq a_i < b$. The number b is called the **base** or **radix**. The value or magnitude of n is

$$a_k \cdot b^k + a_{k-1} \cdot b^{k-1} + \dots + a_0 \cdot b^0.$$

An Algorithm for Alternate Base Representations:

Repeat the three steps until zero appears on the left side of the equation

$$n = \dots + a_6 \cdot b^6 + a_5 \cdot b^5 + a_4 \cdot b^4 + a_3 \cdot b^3 + a_2 \cdot b^2 + a_1 \cdot b^1 + a_0 \cdot b^0.$$

1. The right-most a_i is the remainder of the division by b .
 2. Subtract this remainder from n .
 3. Divide both sides of the equation by b .
- Each iteration yields one a_i coefficient.

If $a \equiv_m b$ and $c \equiv_m d$, then $a \pm c \equiv_m b \pm d$.

If $a \equiv_m b$ and $c \equiv_m d$, then $a \cdot c \equiv_m b \cdot d$.

If $a \equiv_m b$ then $a^n \equiv_m b^n$ for $n \geq 0$.

The **Congruence Cancellation Theorem** (CCT): if $ax \equiv_m ay$ and if a and m are coprime, then $x \equiv_m y$.

The **Modular Multiplicative Inverse** (MMI). An integer i is the modulo m MMI of a number a if $a \cdot i \equiv_m 1$.

The **Modular Multiplicative Inverse Existence Theorem**: i , the mod m MMI of a , exists if and only if a and m are coprime.

The **Unique Modular Multiplicative Inverse** (TUMMI) is the smallest positive mod m MMI, the one and only i such that $0 < i < m$.

The **Chinese Remainder Theorem**: given the remainders from dividing an integer n by an *arbitrary* set of divisors, the remainder when n is divided by the *least common multiple* of those divisors is *uniquely* determinable.

Part III: Connections



Consider a tree. A tree, in its semi-“frozen” state, is a computer powered by sunlight. A tree in New England reacts to the length of the day, running a different program in the summer than in the winter. It figures out when to shed its leaves and when to sprout new ones. A tree processes the information that is available in its environment. Its proteins, organized in signaling pathways, help the tree figure out how to grow its roots toward the water it needs, how to activate an immune response when it is threatened by pathogens, and how to push its leaves toward the sun it craves. A tree does not have the consciousness or language that we have, but it shares with us a general ability to process information. A tree has knowhow, even though the way in which it processes information is unlike our mental abilities and more similar to the processes our own bodies do without knowing how: digestion, immunity, hormonal regulation, and so on.

While a tree is technically a computer, its power source is not an electrical outlet but the sun. A tree is a computer that, just like us, cannot run MATLAB, but unlike computers and us, it has the knowhow to run photosynthesis. Trees process information, and [embody knowhow], which they use to survive. — César Hidalgo in *Why Information Grows: The Evolution of Order, from Atoms to Economies*.

7

Trees

7.1 Growing Trees Left and Right

Til, Abu, and Ila had many fruitful discussions about trees. With access to a record of their sessions, we learn that in their first one, Til said it was time to switch and start drawing trees in the more traditional top-to-bottom way, rather than the left-to-right way they'd been doing. In response, Abu quoted from a book Til had recommended to him by Steven Pinker, who had mentioned the fact that growing trees from the root down was 'botanically improbable' — which Abu thought was pretty funny. Ila just shook her head.

Til went on to explain that, because of their superlative beauty and utility, the two main applications of trees that he wanted them to learn about were

1. fast searching; and
2. data compression.

To learn these applications, they started by growing — building — simple trees from scratch. Constructing them from cells. Not biological cells, that takes too long. Memory cells. Storage cells in computer memory take many forms, Figure 7.1 shows one called a *cons* cell. So called because it's a *construct* — a structure (sometimes abbreviated 'struct') — of two things (the dots are just placeholders — null pointers, perhaps). It could also be called a dyad, a pair, a *ConsPair*,¹ or a 2-tuple. The name is unimportant. What's important is that it's really just a place to store two pointers to other things — including other cons cells.

See Figures 7.2 and 7.3. If the list in Figure 7.3 looks like a tree, it's because it is — a completely lopsided tree — totally "right heavy" but a tree nonetheless. Note that the cons cells in a list are usually drawn lined up horizontally, but it's instructive to look at a list as an unbalanced tree. Other combinations of cons cells build different, this time balanced, trees. See Figures 7.4 and 7.5. Once we know how trees look as cons cells consed together, we can render them without the clutter of the containers and the arrowheads, as shown in Figure 7.6.

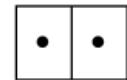


Figure 7.1: A cons cell.

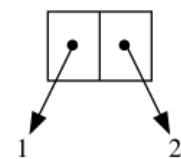


Figure 7.2: A cons cell with pointers pointing to pointees.

¹ See Figure 7.4, for example.

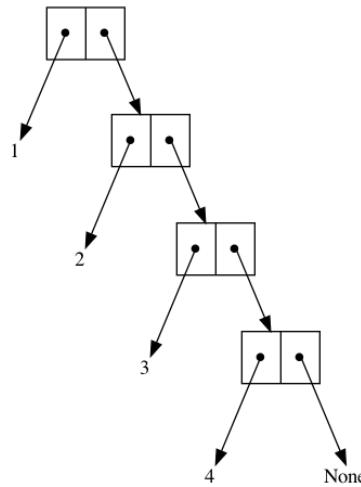


Figure 7.3: A list as a chain of cons cells.

```

from cons import *
print(cons(1, cons(2, cons(3, cons(4, None)))))

# prints [1, 2, 3, 4]
  
```

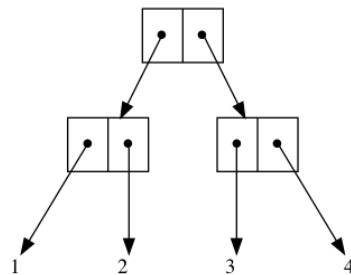


Figure 7.4: A tree with three cons cells.

```

from cons import *
print(cons(cons(1,2), cons(3,4)))

# prints
# ConsPair(ConsPair(1, 2), ConsPair(3, 4))
  
```

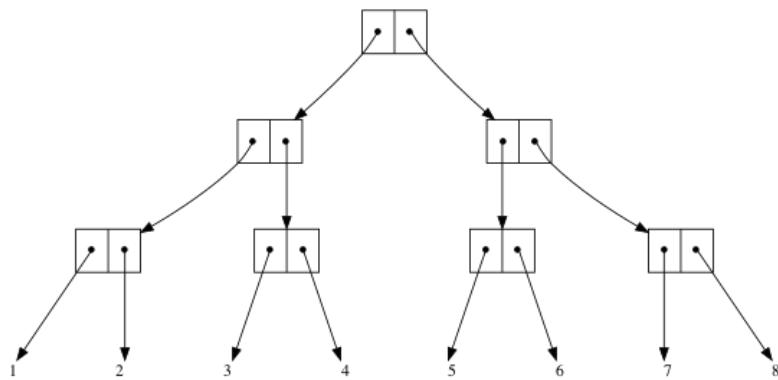
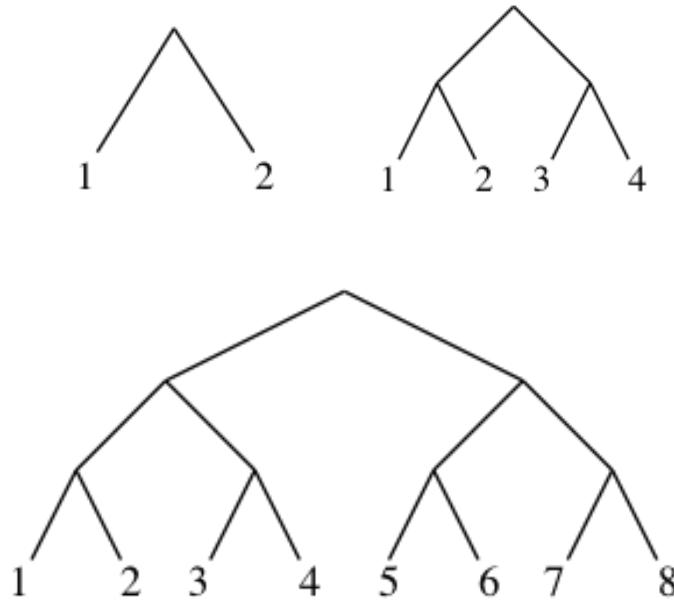


Figure 7.5: A tree with seven cons cells.

```

from cons import *
print(cons(cons(cons(1, 2), cons(3, 4)), cons(cons(5, 6), cons(7, 8))))
# prints (all in one line)
# ConsPair(ConsPair(ConsPair(1, 2), ConsPair(3, 4)),\n
#           ConsPair(ConsPair(5, 6), ConsPair(7, 8)))
  
```

Figure 7.6: Uncluttered trees.



This growth pattern can continue to any depth, but this is enough for now. Before drawing more trees, let's lay out the common nomenclature for things *knobby* and *smooth*.

7.2 Nomenclature

A **node** is the knobby thing, also called a **vertex** (plural **vertices**). It is typically drawn as a two-dimensional box or circle or some other shape, including the null (empty) shape.

A **link** is the smooth thing (drawn as a one-dimensional line), also called an **edge** or an **arc**. As shown in Figure 7.6, a null-shaped node can be inferred as the point where link-lines intersect.

A **tree** is a collection of nodes and links, the links connecting nodes to other nodes such that there is *only one path* from one node to any other node. If there is more than one path, it's not a tree, it's a graph.²

A **path** is a sequence of links from node to node. A proper link sequence has shared nodes between successive links. A link can be represented as an ordered pair (a cons) of nodes. To be a sequence, the first node of a link must be the second node of its preceding link; likewise, the second node of a link must be the first node of its following link. Writing nodes as 1-digit numbers and links as 2-digit numbers, [12, 23, 34] is a valid path sequence, but [12, 32, 34] is not.

A tree always has one more node than it has links. Always. This can be seen in Figure 7.6, but let's make it even more clear by rendering nodes as solid knobs and links as straight lines. Visualize starting with a *seedling* that has only one node (sometimes called a **degenerate** tree). This singleton node has zero links. One is one more than zero.

In succession, as shown in Figure 7.7, we add one link and a node as the other end of our singleton node (for two nodes and one link), then we add another link (with its other-end node) to the first node, and we end up with three nodes and two links. No matter what we try, we can never change the property of a tree having one more node than links. It is **invariant**.

Figure 7.7: A succession of n nodes with $n - 1$ links, $n = 1, 2, 3$.

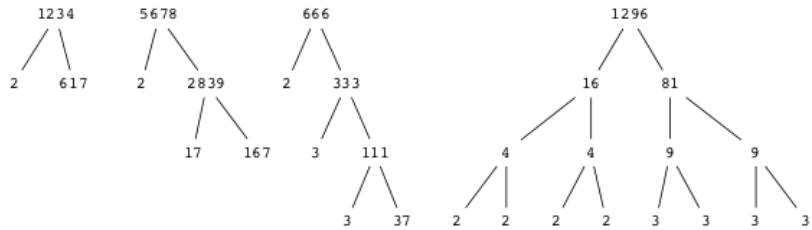


7.3 Branching Out

A **binary** tree (the only kind depicted so far) has *two* links emanating from each node. *Factor* trees are often binary, as shown in Figure 7.8.

This notion can be generalized. An **m-ary** tree has nodes with **m** links, or rather *up to m* links, but to be a **regular** tree, all its nodes must have

Figure 7.8: A forest of factor trees.



exactly **m** links. The number **m** is called the **branching factor**.

Returning to a counting theme, the connection between trees and EGC — Elementary Generative Combinatorics — mentioned in §5.2 — is illuminating. Generating lists of combinations of things allows counting these combinations by simply getting the lengths of the generated lists. Factor trees generate prime factors, which are easy to count by looking at the fringes of the tree, but generating *all* factors (not just the prime ones) may require making **m** *different* at each level of the tree. For example, all factors of 360, whose prime factorization is $2^3 \cdot 3^2 \cdot 5^1$, are found at the bottom of the tree in Figure 7.9. Building this tree is described in Table 7.1.

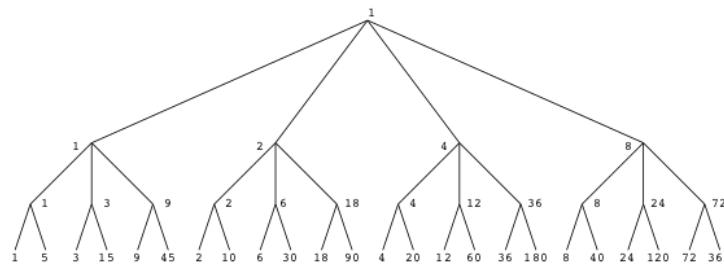


Figure 7.9: A complete factor tree.

Level	Branching Factor	Action to Create Next Level
0	4	Start the branching with four powers of two: $2^0, 2^1, 2^2$ and 2^3 .
1	3	Multiply everything branching off level 0 by three powers of three: $3^0, 3^1$ and 3^2 .
2	2	Multiply everything branching off level 1 by two powers of five: 5^0 and 5^1 .

Table 7.1: Building a complete factor tree.

Multiplying together the three branching factors, $4 \cdot 3 \cdot 2$, gives the total factor count. The code in Figure 7.10 shows the nested-loop nature of enumerating all 24 factors of 360, and then reconstituting each factor in a functional style using the paths through the tree.

Figure 7.10: Enumerating all 24 factors.

```

def print_one_factor(i, j, k, x, y, z):
    print(f'2^{i} * 3^{j} * 5^{k} = {x} * {y} * {z} = {x * y * z:3d}')

def show_all_factors_of_360():
    for i in range(4):
        for j in range(3):
            for k in range(2):
                print_one_factor(i, j, k, 2 ** i, 3 ** j, 5 ** k)

show_all_factors_of_360()
# prints
#  $2^0 \cdot 3^0 \cdot 5^0 = 1 \cdot 1 \cdot 1 = 1$ 
#  $2^0 \cdot 3^0 \cdot 5^1 = 1 \cdot 1 \cdot 5 = 5$ 
#  $2^0 \cdot 3^1 \cdot 5^0 = 1 \cdot 3 \cdot 1 = 3$ 
#  $2^0 \cdot 3^1 \cdot 5^1 = 1 \cdot 3 \cdot 5 = 15$ 
#  $2^0 \cdot 3^2 \cdot 5^0 = 1 \cdot 9 \cdot 1 = 9$ 
#  $2^0 \cdot 3^2 \cdot 5^1 = 1 \cdot 9 \cdot 5 = 45$ 
#  $2^1 \cdot 3^0 \cdot 5^0 = 2 \cdot 1 \cdot 1 = 2$ 
#  $2^1 \cdot 3^0 \cdot 5^1 = 2 \cdot 1 \cdot 5 = 10$ 
#  $2^1 \cdot 3^1 \cdot 5^0 = 2 \cdot 3 \cdot 1 = 6$ 
#  $2^1 \cdot 3^1 \cdot 5^1 = 2 \cdot 3 \cdot 5 = 30$ 
#  $2^1 \cdot 3^2 \cdot 5^0 = 2 \cdot 9 \cdot 1 = 18$ 
#  $2^1 \cdot 3^2 \cdot 5^1 = 2 \cdot 9 \cdot 5 = 90$ 
#  $2^2 \cdot 3^0 \cdot 5^0 = 4 \cdot 1 \cdot 1 = 4$ 
#  $2^2 \cdot 3^0 \cdot 5^1 = 4 \cdot 1 \cdot 5 = 20$ 
#  $2^2 \cdot 3^1 \cdot 5^0 = 4 \cdot 3 \cdot 1 = 12$ 
#  $2^2 \cdot 3^1 \cdot 5^1 = 4 \cdot 3 \cdot 5 = 60$ 
#  $2^2 \cdot 3^2 \cdot 5^0 = 4 \cdot 9 \cdot 1 = 36$ 
#  $2^2 \cdot 3^2 \cdot 5^1 = 4 \cdot 9 \cdot 5 = 180$ 
#  $2^3 \cdot 3^0 \cdot 5^0 = 8 \cdot 1 \cdot 1 = 8$ 
#  $2^3 \cdot 3^0 \cdot 5^1 = 8 \cdot 1 \cdot 5 = 40$ 
#  $2^3 \cdot 3^1 \cdot 5^0 = 8 \cdot 3 \cdot 1 = 24$ 
#  $2^3 \cdot 3^1 \cdot 5^1 = 8 \cdot 3 \cdot 5 = 120$ 
#  $2^3 \cdot 3^2 \cdot 5^0 = 8 \cdot 9 \cdot 1 = 72$ 
#  $2^3 \cdot 3^2 \cdot 5^1 = 8 \cdot 9 \cdot 5 = 360$ 

paths_in_tree = [[1,1,1], [1,1,5], [1,3,1], [1,3,5],
                 [1,9,1], [1,9,5], [2,1,1], [2,1,5],
                 [2,3,1], [2,3,5], [2,9,1], [2,9,5],
                 [4,1,1], [4,1,5], [4,3,1], [4,3,5],
                 [4,9,1], [4,9,5], [8,1,1], [8,1,5],
                 [8,3,1], [8,3,5], [8,9,1], [8,9,5]]

from math import prod
print(list(map(prod, paths_in_tree)))
# prints (on one line)
# [1, 5, 3, 15, 9, 45, 2, 10, 6, 30, 18, 90,
#  4, 20, 12, 60, 36, 180, 8, 40, 24, 120, 72, 360]

```

Another combinatorial **m**-ary tree example is a *permutation tree*. For example, to generate all 24 permutations of [1, 2, 3, 4], *m* goes from 4 to 3 to 2 to 1 as the tree grows. We revert to left-to-right drawing, to save horizontal space, per Figure 7.11.

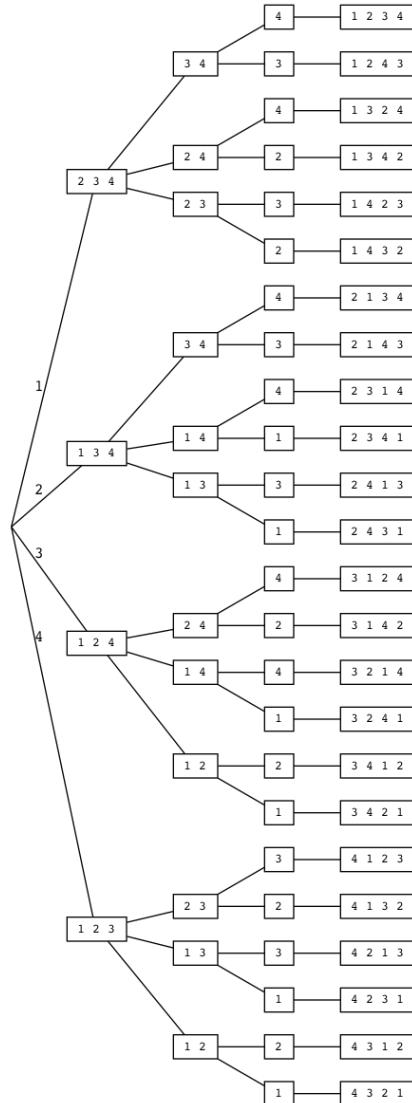


Figure 7.11: $4!$ permutation tree.

In a binary tree a node is also allowed to have only one link; if so, we say it lacks **completeness** or **fullness**. Some nodes legitimately have zero links. These are the **leaf** nodes, or the **leaves** of the tree. Mixing the botanical and the genealogical metaphors, a leaf node is also called a **childless** node. The **children** of non-leaf nodes are distinguishable as either the **left child** or the **right child**.

The node whose children are u and v is naturally called the **parent** of u and v . That makes u and v **siblings** in this family tree. The parent of the parent of u is its **grandparent**, u is its grandparent's **grandchild**, and so on, with **ancestor** and **descendant** likewise defined genealogically.

7.4 Rooted or Not

A **rooted** tree differs from an **unrooted** (or **free**) tree by having a single node designated as its root. Family tree relationships hold only in rooted trees, which break the symmetry of “is-related-to” (a node is related to another node by a line connecting them) into a child-to-parent or a parent-to-child direction. Downwards away from the root goes parent-to-child, upwards towards the root goes child-to-parent. As Figure 7.9 has already revealed, the **level** of a node is the number of links between it and the root. The root node has level 0. The maximum level over all leaf nodes is the **height** (or **depth**) of the tree.

Two trees can look different visually, yet be structurally the same. See examples in Figure 7.12 for two unrooted trees, and Figure 7.13 for two rooted trees.

Back to building, with tree roots explicitly designated as the one at the top, we create the rooted tree of Figure 7.14 with 5 nodes and 4 links. The level of leaf node **a** is 1, the level of both **b** and **c** is 2, so the height of the tree is $\max(1, 2, 2)$ which is 2.

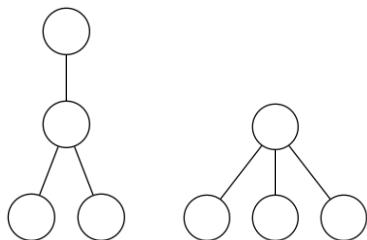


Figure 7.12: Two unrooted trees, structurally equivalent, visually different.

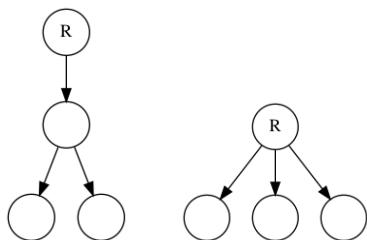


Figure 7.13: Two rooted trees, visually and structurally different. The root node is designated by the ‘R’ label.

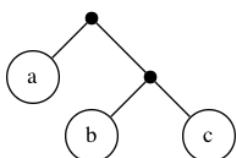


Figure 7.14: 5 nodes, 4 links, 1 root.
cons('a', cons('b', 'c'))

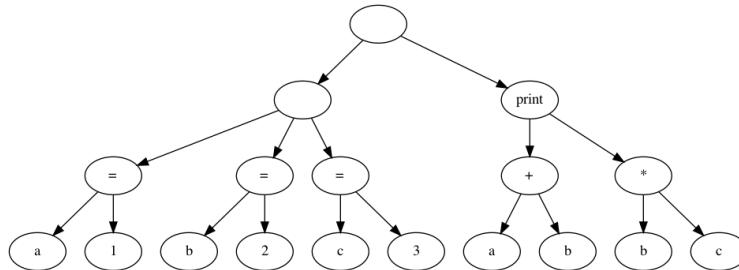
Now we'll build a taller tree, almost like one in Figure 7.3, but rendered slightly differently in Figure 7.15. The height of this tree is 4.

In these simple examples, internal (non-leaf) nodes are different than leaf nodes, in that the leaf nodes are the only ones whose left and right pointees can be actual (non-cons) data. So building trees that store arbitrary data items in their nodes, in addition to left-child and right-child links, requires a method different than using one cons per internal node.

7.5 Expressions

One simple technique for building trees with record-like nodes is to use a list instead of a single cons cell at each node. The elements of the list provide the fields in the record. The idea is to reserve the last m elements for the m child pointers. The pointers are actually to other (nested) lists or non-list data if at a leaf. For example, the list

`[[[['=','a',1],[['=','b',2],[['=','c',3]],['print',[['+','a','b'],[['*','b','c']]]]]]`
represents the tree shown in Figure 7.16.

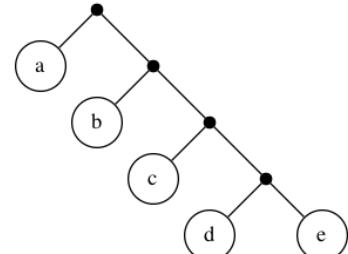


More expressions-as-trees will be explored in Chapter 9 when we study abstract syntax trees and parse trees.

7.6 Fast and Glorious

For our purposes, **fast searching** is looking for some item of data in a data structure that is designed for speed, glorious speed!

In searching for an item in a list of *orderable* items, there is a slow (less efficient) way and a fast (more efficient) way. The less efficient way is **linear search** — or sequential search — in which the searched-for item (called a key) is compared with each item in the list (or array/vector — any sequence type) in a linear left-to-right front-to-back order. The items in the list can be in any order, but orderable items are ones that *can* be sorted. First putting them in sort order (typically ascending — from “smallest” to “largest”) leads to a more efficient way, called **binary search**. The non-tree binary search algorithm uses an indexable array of data. We'll skip taking a closer look at that and just focus on the tree version.



Node	Level
a	1
b	2
c	3
d	4
e	4

Figure 7.15: 9 nodes, 8 links, 1 root.
`cons('a', cons('b', cons('c', cons('d', 'e'))))`

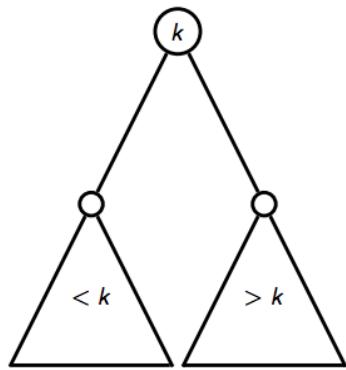
Figure 7.16: Expression tree.
“Evaluating” this tree is equivalent to running the following code:
`a = 1
b = 2
c = 3
print(a + b, b * c)`

```

items = [1, 2, 3, 4]
key = 3
for item in items:
    print('<' if key < item else\
          '=' if key == item else\
          '>')
# prints
# >
# =
# =
# <

```

Figure 7.17: Orderable numbers as keys.

Figure 7.18: An invariant of binary search trees. For any node with k as key, the nodes in its left subtree all have keys less than k , and the nodes in its right subtree all have keys greater than k .

³This irrational number is the log base ϕ of 2, where ϕ is the golden ratio, $\frac{1+\sqrt{5}}{2} \approx 1.618$. Finding the connection between the golden ratio (via the Fibonacci numbers) and BSTs is a good GPAO.

⁴One of four types of tree traversals: in-order, preorder, postorder, and level order.

⁵Many an enterprising individual has created a web site to help visualize building and manipulating binary trees, and especially binary search trees. [This is one of the better ones.](#)

Key comparisons are usually three-way, but it takes a nested “if” to discern which is the case:

1. $\text{key1} < \text{key2}$
2. $\text{key1} = \text{key2}$
3. $\text{key1} > \text{key2}$

Figure 7.17 illustrates this with orderable numbers as keys.

In **binary search**, after each step the search time is cut in half, so a list of size 16 is reduced to a list of size 1 in 4 steps (counting arrows in $16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$). A list of size one is easy to search, just check whether or not its single element compares as equal to the search key.

A **binary search tree** (BST) is a binary tree with a key at each node such that all keys in the node’s left descendants (e.g., **subtree**) are *less than* the node’s key, and all keys in the node’s right descendants (i.e., **subtree**) are *greater than* the node’s key. And of course, all keys at the node itself (all one of them) are equal to that one key. See Figure 7.18.

Note that BSTs are one of the applications of trees that requires ‘beefier’ nodes than cons cells can easily provide.

Here is a list of some other cool characteristics of BSTs, where n is the number of nodes (keys), and h is the height of the tree:

- Search time is a logarithmic function of n (assuming a *balanced* tree — see §7.7).
- Insertion of a key is equally fast: search for the key, insert it at the leaf where the search terminates.
- Deletion is sometimes but not always as fast as insertion.
- For each search, insert, or delete operation, the worst case number of key comparisons is $h + 1$.
- Height is bounded, below and above: $\lfloor \lg n \rfloor \leq h \leq n - 1$
- With average (random) data: $h < 1.44 \lg n$ — actually, the constant is closer to 1.4404200904125564 .³
- With a poorly built (unbalanced) tree, in the worst case binary search degenerates to a sequential search.
- Inorder⁴ traversal produces a sorted list (the basis of the *treesort* algorithm).

The process of **building a BST** iterates through a sequence of keys, in whatever order they may be, and creates a node for each key. It links each node after the first (which has no parent, it being the root) to its parent as either its left child (if the key is less than its parent’s key) or its right child (if the key is greater than its parent’s key). The parent might already have that child; if so, that child becomes the candidate parent to link the new node to.

To build a BST⁵ for a set of words, for example — trees, are, cool, and, the, binary, search, kind, wonderfully, and so — use alphabetical (dictionary) order for ordering the words, which are the keys. See Figures 7.19, 7.20, 7.21, 7.22, and 7.23.

Figure 7.19: Steps 1 and 2 of building a BST.

Step 1, insert `trees` as the root of the BST.

`trees`

Step 2, insert `are` as the left child of `trees` since `are < trees`.

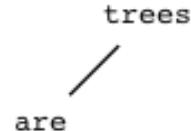
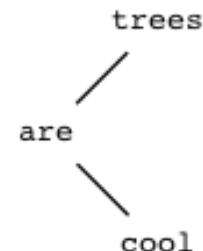


Figure 7.20: Steps 3 and 4 of building a BST.

Step 3, insert `cool` as the right child of `are` since `cool < trees` and `cool > are`.



Step 4, insert `and` as the left child of `are` since `and < trees` and `and < are`.

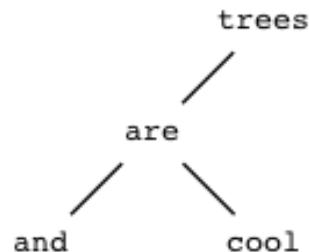
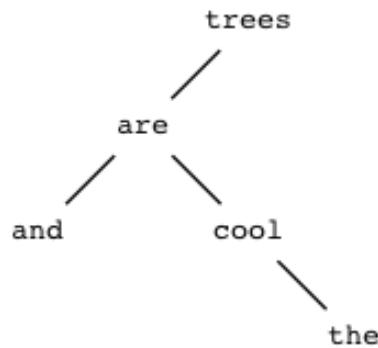


Figure 7.21: Steps 5 and 6 of building a BST.

Step 5, insert the as the right child of cool since the $<$ trees and the $>$ are and the $>$ cool.



Step 6, insert binary as the left child of cool since binary $<$ trees and binary $>$ are and binary $<$ cool.

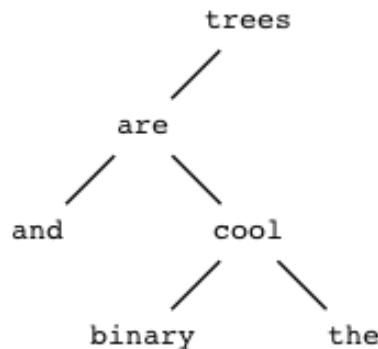
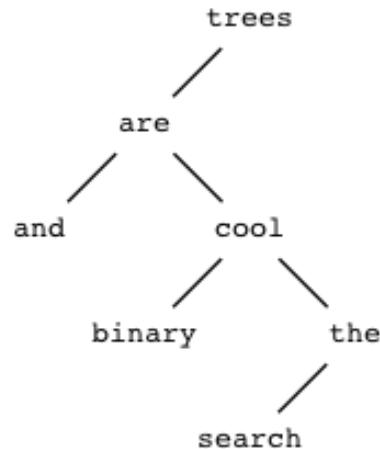


Figure 7.22: Steps 7 and 8 of building a BST.

Step 7, insert search as the left child of the since search < trees and search > cool and search < the.



Step 8, insert kind as the left child of search since that *is* where it goes.

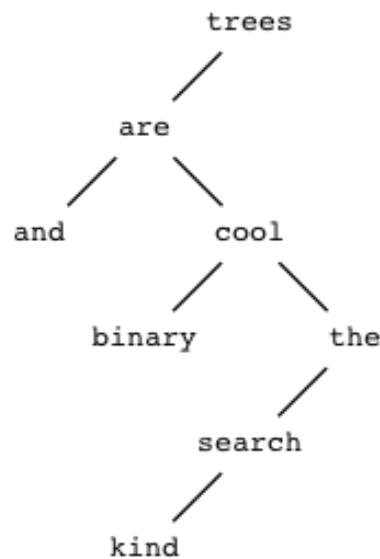
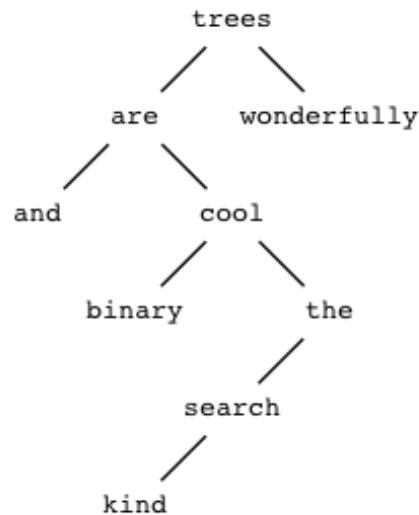
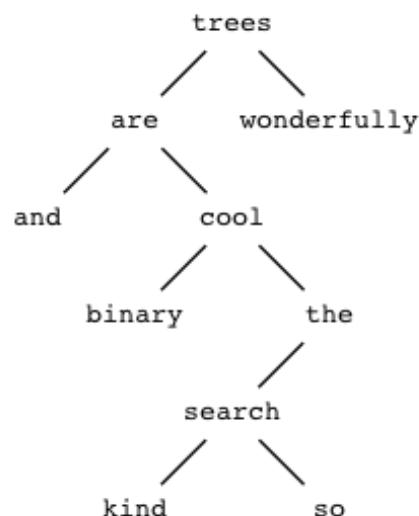


Figure 7.23: Steps 9 and 10 of building a BST.

Step 9, insert wonderfully where it goes. (Finally, a right child for the root!)



Step 10, insert so where it goes, and the tree is finished.



The process of **post-construction binary search tree insertion** is what we call inserting a new key *not* in the set of keys from which the BST was originally built.

To insert a new word, *very*, into this BST, we must compare it to *trees*, then *wonderfully*. At this point the search terminates, because *wonderfully* has no children. Two comparisons were needed to find the insertion site for *very* as the left child of *wonderfully*. See Figure 7.24.

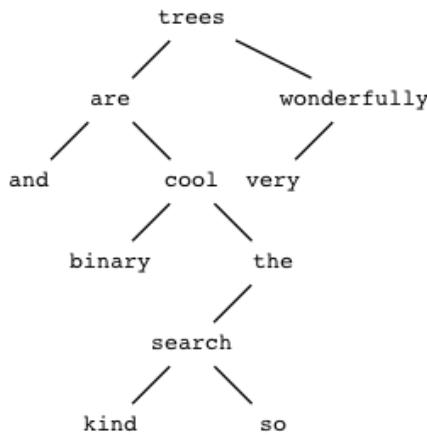


Figure 7.24: Post construction BST insertion.

If a key is already in the BST, inserting that key has no effect. It still costs some comparisons, however. Inserting (*futilely*) *cool* costs 3 comparisons (one plus the level of the *cool* node), and likewise, *kind* costs 6 comparisons.

The **expected search cost** is the average cost of finding the keys in a BST. It is computed as the average over all keys of the cost of finding each key, which is the number of key comparisons necessary to find that key. The key at the root costs one comparison to find, its child keys two comparisons, its grandchildren three, and so on.

7.7 Balancing Act

Table 7.2 shows a comparison of the search costs for the finished tree in Figure 7.24 and two balanced versions of it (Figures 7.25 and 7.26).

Version 2 is “left-heavy” — and it’s more typical to have the last level filled from the left instead of the right (as in version 1).

In a **balanced tree**, all leaves are on the same level, or within one level of each other — so neither left-heavy nor right-heavy — on the level. Other terms describing binary trees are **full**, **complete**, and **perfect**.

- **Full:** Every node has two links or zero links. The nodes with zero links are the leaves. The tree in Figure 7.3 is full, but neither complete nor balanced.

Key	Cost in 7.24	Cost in 7.25	Cost in 7.26
and	3	3	4
are	2	2	3
binary	4	3	4
cool	3	1	2
kind	6	4	4
search	5	3	3
so	6	2	1
the	4	4	3
trees	1	3	2
very	3	3	4
wonderfully	2	4	3
Total	39	32	33
Average	3.55	2.91	3.0

Table 7.2: Unbalanced and balanced search costs. Average cost: 3.55 comparisons (in an unbalanced BST) versus 2.91 or 3.0 comparisons (in balanced BSTs).

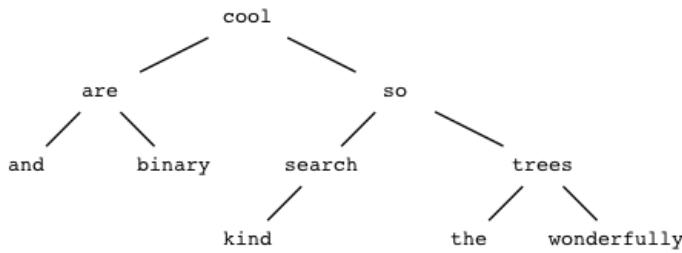


Figure 7.25: Balanced BST version 1.

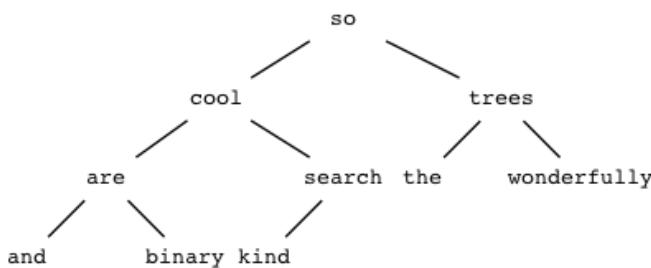


Figure 7.26: Balanced BST version 2.

- **Complete:** Nodes with one leaf have that leaf to their left, with the last level having all nodes as far to the left as possible. The last level is possibly unfilled, but every other level is completely filled. A complete tree is necessarily balanced, and, fun fact, an array can be used to efficiently represent it.
- **Perfect:** Every node has two children except the leaf nodes, and all leaves are at the same level. Perfect implies balanced, full and complete.⁶ The trees in Figures 7.4, 7.5, and Figure 7.6 are all perfect.

⁶ The Wikipedia “Binary Tree” page gives the example of a perfect binary tree of “the (non-incestuous) ancestry chart of a person to a given depth, as each person has exactly two biological parents (one mother and one father).”

Of the set of all binary trees, neither of the subsets “Complete” and “Full” is contained in the other. They have a non-empty intersection — some trees are both, but a tree can be full but not complete, or complete but not full, as per Figures 7.27 and 7.28.

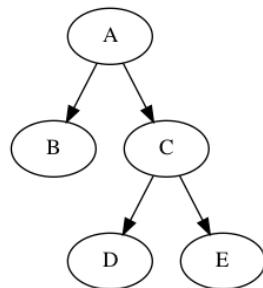


Figure 7.27: Full but not complete.

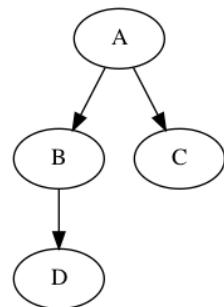


Figure 7.28: Complete but not full.

⁷ To quote the [Binary Tree page](#) again, “A balanced binary tree [is one] in which the left and right subtrees of every node differ in height by no more than 1. One may also consider binary trees where no leaf is much farther away from the root than any other leaf. (Different balancing schemes allow different definitions of ‘much farther’.)” All of these different balancing schemes deserve a closer look, but now is not the time. For now, a summary will suffice. So, for insertion on-the-fly, one general strategy is to be sure to only insert items in a sorted manner, thus keeping the tree sorted. More specifically, use

- a *red-black* tree, which keeps the tree somewhat balanced by mapping a binary tree to a 2-3-4 tree.
- an *AA* tree, which is also sometimes referred to as a red-black tree, and keeps itself balanced by mapping a binary tree to a 2-3 tree.
- an *AVL* tree, which keeps the tree balanced by storing at each node a “balance factor” (essentially the height of the tree at each node), and rotating portions of the tree whenever one side gets too much taller than the other.
- a *scapegoat* tree is just like a regular binary tree except that it occasionally rebalances a large portion of it whenever it detects that it is becoming too unbalanced.
- a *treep* assigns a random weight to each node and ensures that the weights of child nodes are always less than the weights of parent nodes. This keeps the tree fairly balanced.

For at-the-end insertion, find which node would make the tree the most balanced if it were the root. Then, rotate the tree, making that node the root and all other nodes its descendants.

⁸ There are three cases to consider. See a description of these cases and how each is dealt with at [this web site](#). Deletion is in the same efficiency class as insertion ($\mathcal{O}(\log n)$), but is slightly slower because of the logic and possible tree manipulation required to deal with the three cases.

Various techniques have been devised to rebuild unbalanced BSTs to make them balanced. This rebuilding can happen on-the-fly (as items are inserted) or at-the-end (after all initial items are inserted).⁷

Also warranting investigation is how to delete a key from a BST, and answer the question: Is deletion always as fast as insertion? As briefly mentioned at the beginning of §7.6, in the list of cool characteristics of BSTs, deletion is sometimes but not always as efficient as insertion. To delete a key from a BST, a search must first be performed to find where the key is in the binary tree. Then, if it is found, the key is deleted and its children (if any) reparented.⁸ The tree may need to be rebalanced as well.

7.8 Optimal Search

Minimizing the expected search cost for a static (unchanging) tree and a set of keys whose access frequencies are known in advance (think English words as might be used by a spell-checker) is the *Optimal BST problem*. In this problem keys are weighted based on how likely they are to be looked up. Estimates of these likelihoods are usually good enough if they can't be computed exactly. Of course, the more accurate the estimates, the speedier the average lookup. So an optimal BST would have a common word like *is* at or near the root, and a very uncommon word like *syzygy* in the leaves.

For a simple example of the difference it can make, let's consider 4-node BSTs with the keys 1, 2, 3, and 4. There are only so many of these, and for a given set of weights, in terms of expected search cost one BST may be better than another, or indeed all others. Figures 7.29 and 7.30 show two of fourteen possibilities.

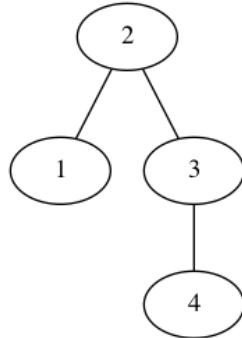


Figure 7.29: A BST with weights {1 : 0.4, 2 : 0.3, 3 : 0.2, 4 : 0.1}, and expected search cost $2(0.4) + 1(0.3) + 2(0.2) + 3(0.1) = 1.8$.

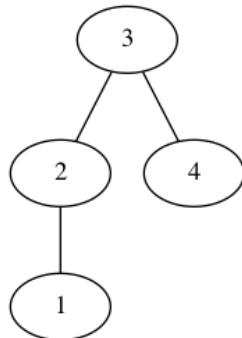


Figure 7.30: Another BST with weights {1 : 0.4, 2 : 0.3, 3 : 0.2, 4 : 0.1}, but expected search cost $3(0.4) + 2(0.3) + 1(0.2) + 2(0.1) = 2.2$.

The tree in Figure 7.29 is better, as it has a lower expected search cost, 1.8 versus 2.2 for the tree in Figure 7.30.

7.9 Compression Confession

⁹ Til's favorite kind of binary tree, he confessed to Abu and Ila.

¹⁰ A *fascinating individual* to get to know more about.

A **Huffman tree**⁹ is *like* an optimal BST, but with a twist. It optimizes space, not time.

The idea of **data compression** is to change how data is represented in order to save storage space by encoding the data in a space-efficient way. Pioneering computer scientist David Huffman¹⁰ figured out how to encode certain information efficiently — meaning, specifically, minimizing the average number of bits per symbol (information atom). A Huffman tree does this by storing more frequently seen symbols closer to the root and less frequently seen symbols further from the root. Bits (1s and 0s) can label the links on each path from root to leaf, giving a unique bit string encoding for each symbol.

Unlike a BST, a Huffman tree does not care about key order, and it doesn't (have to) use internal (non-leaf) nodes to store the data (neither keys nor bits). The only necessary data items are the symbols, stored at the leaves. The symbols' encodings are implicitly stored in the structure of the tree. For example, consider the symbols A through F together with their frequencies shown in Table 7.3. These frequencies could have come from some random text like

BDFCFFDFFBFECEFFDEFBAFEFFECBBDFCFFFFCBDFCEFFCB
DAFFACEDFADEEFFAEDEBDADDFEEFAFACFDFFDBECEEBDFCCFE

which has 100 characters in it. Table 7.4 shows counts of how many of each symbol there are in this random text. The difference is that counts are whole numbers. The sum of the counts is the denominator of the fraction having each count as the numerator to obtain the frequency. It should come as no surprise that the sum of the frequencies is 1. Starting with counts and ending with frequencies is a **normalization** operation, where 1 is the norm. Computing this normalization to obtain frequencies is optional as far as Huffman is concerned, as the counts could serve equally well.

The compression process begins with an initial **forest** of trees (Abu calls them *seedlings*). These are conveniently stored in a *priority-queue* — a data structure where high-priority items are always maintained at the front of the queue. The nodes in this collection are ordered by lowest-to-highest frequencies. (Again, we could also have used counts.)

Shown in Figure 7.31, the algorithm for building a Huffman tree creates an initial queue Q of single-node trees from some pairing of n symbols and n numbers, which can be passed as pairs or as parallel lists. We'll call the numbers *weights*. They are the frequencies, or the counts — how often the symbols (e.g., characters) occur in some information (e.g., a body of text), as shown above. These weights define the priorities of the symbols, reverse ordered, so the lower the weight, the higher the priority. Once the tree is built, the weights can be forgotten, all that matters is where the symbols are found in the tree.

Algorithm Huffman tree (inputs and initialization of Q as above)

```

while  $Q$  has more than one element do
     $T_L \leftarrow$  the minimum-weight tree in  $Q$ 
    Delete the minimum-weight tree in  $Q$ 
     $T_R \leftarrow$  the minimum-weight tree in  $Q$ 
    Delete the minimum-weight tree in  $Q$ 
    Create a new tree  $T$  with  $T_L$  and  $T_R$  as its left and right subtrees,
    and weight equal to the sum of  $T_L$ 's weight and  $T_R$ 's weight.
    Insert  $T$  into  $Q$ .
return  $T$ 
```

The steps in the body of the **while** loop consolidate T_L and T_R into a parent node (with weight the sum of the weights of T_L and T_R) and insert it after x where it goes in the priority queue. With s standing for symbol, w standing for weight, and the '+' operator defined for symbols as normal string concatenation, we'll use “*insert* (s_1, w_1) + (s_2, w_2) = ($s_1 + s_2, w_1 + w_2$) after x ” as shorthand for these loop steps and show step-by-step how the algorithm proceeds with the given example, using record-type tree nodes

Symbol	Frequency
A	0.08
B	0.10
C	0.12
D	0.15
E	0.20
F	0.35

Table 7.3: Six symbols and their occurrence frequencies.

Symbol	Count
A	8
B	10
C	12
D	15
E	20
F	35

Table 7.4: Six symbols and their occurrence counts.

Figure 7.31: Algorithm for building a Huffman tree.

having two fields, symbol and weight, starting with our initial queue of singleton trees (Figure 7.32). After that, the current contents of the queue will always be displayed as the top row of Figures 7.33, 7.34, 7.35, 7.36, and 7.37.

Figure 7.32: Initial queue of singleton trees.

A 0.08	B 0.10	C 0.12	D 0.15	E 0.20	F 0.35
----------	----------	----------	----------	----------	----------

Figure 7.33: Iteration 1, insert (A,0.08) + (B,0.10) = (AB,0.18) after (D,0.15).

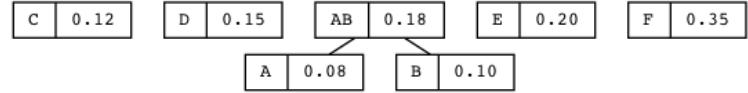


Figure 7.34: Iteration 2, insert (C,0.12) + (D,0.15) = (CD,0.27) after (E,0.20).

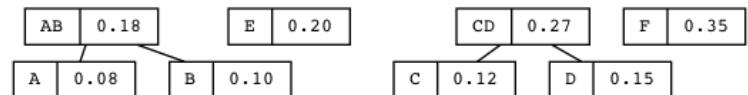


Figure 7.35: Iteration 3, insert (AB,0.18) + (E,0.20) = (ABE,0.38) after (F,0.35).

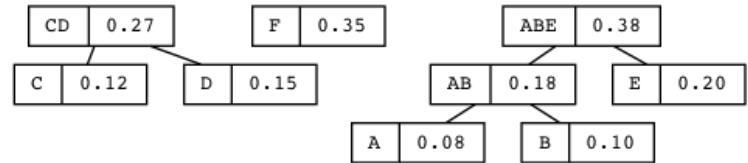


Figure 7.36: Iteration 4, insert (CD,0.27) + (F,0.35) = (CDF,0.62) after (ABE,0.38).

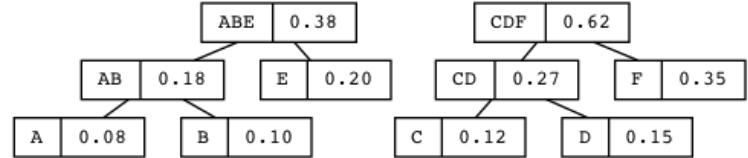
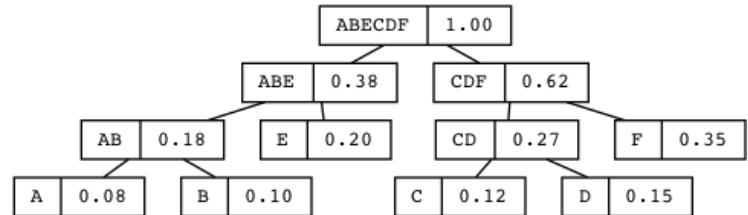


Figure 7.37: Iteration 5, just insert (ABE,0.38) + (CDF,0.62) = (ABECDF,1.00) back into the queue.



The queue no longer has more than one element, so exit the **while** loop and return the tree, whose root node is (ABECDF, 1.00).

A preview of the encoding embedded in this tree is shown in Table 7.5, with more explanation to come in §7.10.

A	000
B	001
C	100
D	101
E	01
F	11

Table 7.5: Encoding preview.

By virtue of its construction as a tree, this encoding is **prefix-free**, meaning no symbol's bit string is a prefix (initial substring) of any other symbol's bit string. The compression is not great, but serves to illustrate the point of all this work. A quick measure of this encoding's success is the average number of bits ($3 \cdot 0.08 + 3 \cdot 0.10 + 3 \cdot 0.12 + 3 \cdot 0.15 + 2 \cdot 0.20 + 2 \cdot 0.35 = 2.45$) used to encode a character, again, more on which in §7.10.

7.10 Encoding and Decoding

Huffman Encoding is the process of creating a code for each symbol in some text, or simply, each character in a string, using the following algorithm on the Huffman tree built for that string:

1. Traverse the binary tree and generate a bit string for each node of the tree as follows:
 - Start with the empty bit string for the root;
 - append 0 to the node's string when entering the node's left subtree.
 - append 1 to the node's string when entering the node's right subtree.
2. When entering a leaf node, print out the current bit string as the leaf's encoding.

Rather than output the encoding, an alternative is to store it as another field of the leaf nodes (and/or in a separate lookup table, like the one in Table 7.5, which can easily be turned into a Python dictionary). As mentioned earlier, the links can be labeled 0 or 1, as they are in the rendition of the Huffman tree of Figure 7.38.

A Huffman tree with n leaves has a total of $2n - 1$ nodes, each of which must be visited to generate the leaf nodes' strings. But since each node has a constant height, the efficiency of this algorithm is very good — linear in the number of nodes.

Given the Huffman encoding in Table 7.5, the string BABEFACE is encoded as 001000001011100010001, a total of 21 bits. This is slightly bet-

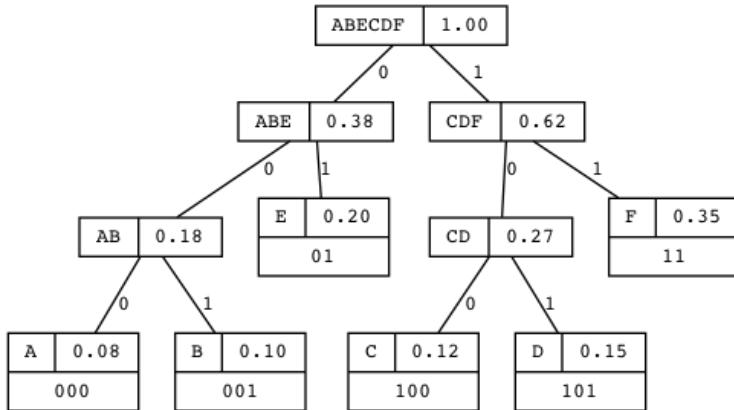


Figure 7.38: Huffman tree encoding stored at its leaves.

ter than the total of 24 bits of 010000101100001000001100 for this 8-character string that is required by the standard fixed-length (3 bits per character) encoding scheme shown in Table 7.6.

Table 7.6: Fixed-length encoding.

A	000
B	001
C	010
D	011
E	100
F	101

Huffman Decoding is the process of decoding a string encoded using a Huffman encoding. A bit string (e.g., 0010001011000010101) is decoded by considering each bit from left to right, and starting at the root of the Huffman tree, traversing left if the bit is 0 and traversing right if the bit is 1, stopping when a leaf is reached. Output the symbol at the leaf, and continue with the next bit. The result for the sample string is shown in Table 7.7.

Table 7.7: Example of decoding.

001	000	101	11	000	101	01
B	A	D	F	A	D	E

7.11 Walking It Through

In this longer example, we start with a string of text and Huffman encode it, walking through each step, but without drawing trees. Recall that in the A through F example of §7.9, the frequencies are given as fractions whose total is 1. When starting with a string to encode, the frequencies of each character are more easily found as occurrence counts, and then

either normalized to total 1, or just left as counts. We opt for the latter approach for this cryptic string.¹¹

THE INTERSECTION OF STARVIEW DRIVE AND CENTENNIAL LOOP

This string has 54 total characters including the 7 spaces between the 8 words. Using ‘@’ for the space character, the occurrence counts are as given in Table 7.8.

Character	Count
@	7
A	3
C	2
D	2
E	7
F	1
H	1
I	5
L	2
N	6
O	4
P	1
R	3
S	2
T	5
V	2
W	1

¹¹ Perhaps it’s a message giving the location of some event!

Table 7.8: Message character counts.

Sorting by lowest to highest count, we initialize our queue of character-count pairs. Note that in this example we have ties — counts that are equal: F1 H1 P1 W1 C2 D2 L2 S2 V2 A3 R3 O4 I5 T5 N6 @7 E7

Compressing (!) the Huffman tree construction steps, we show only the abbreviated root of each tree in the ever-shrinking queue in Table 7.9. The simple rule we use to break ties¹² is to give “deference to our elders”—put *younger* nodes (those more recently created) after *older* nodes (those less recently created).

The tree can be drawn¹³ by carefully working backwards through the 16 construction steps. We highlight the distinction between internal and leaf nodes by showing internal nodes as ellipses labeled only with the combined count (weight) at that node. Note that in Figure 7.39 the subtree under node ‘9’ is flopped left-to-right to save horizontal drawing space. See also Table 7.10, which tabulates the data from the tree.

¹² Does it matter what tie-breaking rule you use, including a random, toss-a-coin-to-break-a-tie rule? Why or why not?

¹³ After being generated with the help of an online [Huffman Tree generator tool](#).

Step	Queue
0	F1 H1 P1 W1 C2 D2 L2 S2 V2 A3 R3 04 I5 T5 N6 @7 E7
1	P1 W1 C2 D2 L2 S2 V2 FH2 A3 R3 04 I5 T5 N6 @7 E7
2	C2 D2 L2 S2 V2 FH2 PW2 A3 R3 04 I5 T5 N6 @7 E7
3	L2 S2 V2 FH2 PW2 A3 R3 04 CD4 I5 T5 N6 @7 E7
4	V2 FH2 PW2 A3 R3 04 CD4 LS4 I5 T5 N6 @7 E7
5	PW2 A3 R3 04 CD4 LS4 VFH4 I5 T5 N6 @7 E7
6	R3 04 CD4 LS4 VFH4 I5 T5 PWA5 N6 @7 E7
7	CD4 LS4 VFH4 I5 T5 PWA5 N6 @7 E7 R07
8	VFH4 I5 T5 PWA5 N6 @7 E7 R07 CDLS8
9	T5 PWA5 N6 @7 E7 R07 CDLS8 VFHI9
10	N6 @7 E7 R07 CDLS8 VFHI9 TPWA10
11	E7 R07 CDLS8 VFHI9 TPWA10 N@13
12	CDLS8 VFHI9 TPWA10 N@13 ER014
13	TPWA10 N@13 ER014 CDLSVFHI17
14	ER014 CDLSVFHI17 TPWAN@23
15	TPWAN@23 ER0CDLSVFHI31
16	TPWAN@ER0CDLSVFHI54

Table 7.9: Walk through with ever shrinking queue.

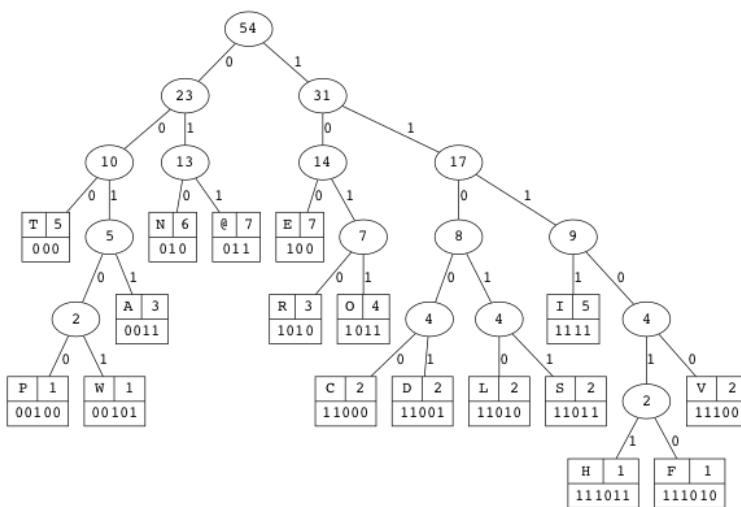


Figure 7.39: Huffman tree for longer message.

Character	Code	# of Bits in Code	Frequency Count	# of Bits × Count
@	011	3	7	21
A	0011	4	3	12
C	11000	5	2	10
D	11001	5	2	10
E	100	3	7	21
F	111010	6	1	6
H	111011	6	1	6
I	1111	4	5	20
L	11010	5	2	10
N	010	3	6	18
O	1011	4	4	16
P	00100	5	1	5
R	1010	4	3	12
S	11011	5	2	10
T	000	3	5	15
V	11100	5	2	10
W	00101	5	1	5
Totals		75	54	207

Table 7.10: Data extracted from Huffman tree for longer message, for an average number of bits per character = $207/54 \approx 3.83$.

With 17 different characters we need 5 bits per character for a fixed-length encoding.¹⁴ Thus $54 \cdot 5 = 270$, so $270 - 207 = 63$ bits are saved by using Huffman encoding. It is customary to use a ratio to express this savings. The **compression ratio** of an encoding is the percentage $\frac{(f-v)}{f} \cdot 100$, where f is the number of bits per symbol of the smallest fixed-length encoding, and v is the average number of bits per symbol with the variable-length (e.g. Huffman) encoding.

So, for this example with $f = 5$ and $v = 3.83$, we calculate the compression ratio as $\frac{5-3.83}{5} \cdot 100 \approx 23\%$.

¹⁴ Why is that? In general, what would be required for n different characters?

7.12 Summary of Terms and Definitions

A **node** is a structural knob or knot, also called a **vertex** (plural **vertices**). Typically drawn as a two-dimensional box or circle or some other shape, including the null (empty) shape.

A **link** is a smooth connector (drawn as a one-dimensional line), also called an **edge** or an **arc**. A node is found where link-lines intersect.

A **tree** is a collection of nodes and links, the links connecting nodes to other nodes such that there is *only one path* from one node to any other node.

A **path** is a sequence of links from node to node. A proper link sequence has shared nodes between successive links. A link can be represented as an ordered pair (a cons) of nodes. To be a sequence, the first node of a link must be the second node of its preceding link; likewise, the second node of a link must be the first node of its following link. For example, with nodes represented as 1-digit numbers and links as 2-digit numbers, [12, 23, 34] is a valid path sequence, but [12, 32, 34] is not.

A **degenerate** tree is a tree with one node.

An **invariant** is some property (e.g., of a tree) that never changes, but always holds.

An **m-ary** tree has nodes with up to **m** links.

A **regular** tree has nodes with exactly **m** links.

The **branching factor** is the number **m** in **m-ary**.

A **binary** tree has a branching factor of 2.

An **invariant** is a property that never changes. One tree invariant is that the number of nodes is always one more than the number of links.

In a binary tree a node can have two, one, or zero links. A **complete** or **full** tree has nodes with mostly two links apiece.

The **leaf** (or **childless**) nodes, or the **leaves** of the tree, have zero links.

The **children** of non-leaf (**internal**) nodes are distinguishable as either the **left child** or the **right child** in a binary tree.

The node whose children are u and v is called the **parent** of u and v , making u and v **siblings**.

The parent of the parent of u is its **grandparent**, u is its grandparent's **grandchild**, and so on, with **ancestor** and **descendant** likewise defined genealogically.

A **rooted** tree differs from an **unrooted** (or **free**) tree by having a single node designated as its root. Family tree relationships hold only in rooted trees, which break the symmetry of “is-related-to” (a node is related to another node by a line connecting them) into a child-to-parent or a parent-to-child direction. Downwards away from the root goes parent-to-child, upwards towards the root goes child-to-parent.

The **level** of a node is the number of links between it and the root. The root node has level 0. The maximum level over all leaf nodes is the **height** (or **depth**) of the tree.

Looking for some data item in a data structure that is designed for speed is called **fast searching**.

Called **linear search** — or sequential search — the searched-for item (called a **key**) is compared with each item in a sequence in a linear left-to-right front-to-back order.

Items are **orderable** if they *can* be sorted based on some attribute, magnitude for example.

Called **binary search** the searched-for item is compared with the middle of some list or sublist or the root of some tree or subtree, going to the left for a ‘<’ comparison, or to the right for a ‘>’ comparison. After each comparison the search time is cut in half.

A **subtree** of any tree, i.e., starting at a given node, is a tree rooted at one of the descendants of the node.

A **binary search tree** (BST) is a binary tree with a key at each node with the invariant property that all keys in the node's left subtree are *less than* the node's key, and all keys in the node's right subtree are *greater than* the node's key. The node's key is equal to

itself.

Called **post-construction binary search tree insertion**, it is the process of inserting a new key *not* in the set of keys from which the BST was originally built.

The **expected search cost** is the average cost of finding the keys in a BST, which is the average over all keys of the cost of finding each key.

In a **balanced tree**, all leaves are on the same level, or within one level of each other.

Changing how data is represented in order to save storage space by encoding the data in a space-efficient way is known as **data compression**.

8

Graphs

8.1 Applications Abound

Til, according to the record, gave Abu and Ila a long list of applications of graphs to investigate. Here's just the start of it:

- Networking
- Scheduling
- Flow optimization
- Circuit design
- Path planning

When they met, he asked them what applications they investigated they found most interesting. Ila said she liked graphs with social implications, like social networks and professional networks that link friends, acquaintances, colleagues, mentors, influencers, etc. Abu asked her to describe some of these. Ila obliged by describing a group of people, where some of them know each other, and some don't. Link the ones that do and you have an acquaintanceship graph. Abu then described one of his favorites, a collaboration graph, which is also about people, but to be connected you have to have collaborated somehow,¹ not just be acquainted. He added, "Like if you were to co-author a paper, or co-star in a movie, or something."

8.2 Origins

Ila responded, "It was a social activity that sparked the idea of graphs in the first place. I love this story about none other than Euler. He came up with the answer to a question the people of Königsberg wondered about as they enjoyed long Sunday walks across the bridges in their city. Seven bridges² to be exact, four of which crossed the river running through the city to land on an island in the river, two others crossed to land on an adjacent island, and the seventh one connected the two islands. Refer to the picture at the link."

Ila continued, "So these Sunday strollers wondered as they walked, is

¹ Two such collaboration graphs have as subjects [Paul Erdős and Kevin Bacon](#), even though those two never collaborated.

² Famously of Königsberg, described as a [historically notable problem in mathematics](#) — the picture Ila was referring to is found [here](#) — and in the Wikipedia article at the top.

it possible to cross each bridge exactly once and come back to the place we started? Euler took on the challenge, and soon settled the question in the negative, giving birth to graph theory in the process.”

Til said, “And what a robust theory has grown from that inauspicious beginning! Consider that just as there are different ways things are related, there are different ways graphs can be constructed, and a whole raft of different kinds of graphs. For a broad example, we can define graphs as *homogeneous* or *heterogeneous*. If all the connections are of the same type — *knows*, *is friends with*, *is adjacent to* — we have the former. If we mix types in the same graph we have the latter.”

Changing the subject a little, Til continued, “We can profit from studying all kinds of relationships in abstract but visual and spatial ways, in order to gain more insight into how things are related. Do you agree we are better served if proximity in a graph means proximity in real life, however we define proximity or closeness?” Abu nodded and said, “I suppose a transportation network would be confusing if geometrically New York and Los Angeles were drawn right next to each other, while Houston and Dallas were far apart.” Til said, “More abstractly, words can be linked in a graph where words with similar meanings are close, and words with different meanings are distant from each other.”

Ila jumped in, “Maps are like graphs, right? A visual way to grasp a large body of data, what’s where, a territory, with many details left out.” Abu said, “Lightening our cognitive load?” Til said, “Yes! Also, concept maps link related words or concepts, and help to organize a tangle of ideas. So, do you like genealogy? Analyzing family relationships is a fruitful application of graphs and trees. How about computer games? These link virtual worlds with players and objects in graphs.”

Abu ventured, “Not computer games, but games and tournaments featured in a family reunion I went to last month. We had a ping-pong tournament where everybody who signed up played one game with everybody else. So in other words, it was a *Round-Robin tournament*. We actually graphed it to see who the ultimate winner was — whoever won the most games.” Ila asked, “Was it you?” “No,” said Abu, “it was one of my young athletic nephews who was the champion. The graph clearly showed he had the most arrows pointing away from him!” Ila said, “They pointed to all the people he beat, right?” Abu nodded. Ila nodded back, and said, “So how many pointed to you? More than pointed away?” Abu said, “The same, actually. I won exactly as many games as I lost. Mediocrity at its finest!” Ila groaned, and said, “You’re a funny, funny man, Abu.” “Hey!” Abu said, with a wink, “Do you mean funny ‘haha’ or funny ‘strange’?!”

Interrupting, Til said, “Let’s stay on target. So can we **truthfully** say that *anything* and *everything* can be a potential application of graphs?” Ila and Abu glanced at each other, but neither spoke. Had Til posed a stumper? “Remember, graphs represent relations, and relations can de-

scribe the extension of *any* predicate, so *anything* describable by relations is representable by graphs!" Ila said, "Wait, what do you mean by *a predicate's extension*?" "I think I know," Abu said. "It's the set of all true facts that predicate describes. But can't that set be infinite? How can a graph represent something infinite?" Ila said, "I think *representable* doesn't mean *drawable*, if that's what you're thinking."

8.3 Visually Appealing

"Ah, but a drawing *is* one representation of a graph. Clipped very small, what this infinite graph³ looks like is mostly left to our imagination," Til said, as he shared the link and went on. "But for finite (and relatively small) graphs, there's an active area of research at the intersection of geometry and graphs that studies how to draw graphs in a visually pleasing way,⁴ with straight lines, few or no line crossings, symmetry, balance, you get the idea. Take the RSA graph, for example." (Til had already shared this graph, shown in Figure 8.1, when they were studying RSA.)

"But that's a tiny graph," said Til. "When they get big is when drawing them gets *really* hard. When they get big as in *real life big* — well — what's the largest living organism on Planet Earth? Not the Blue Whale, as you might be thinking — there's something even bigger."

Said Abu, "I remember reading somewhere about some Aspen trees in Colorado or Utah that are really one big supertree. Their roots are all connected, and the DNA samples from each tree match. So while above ground they look like a forest of distinct trees, it's really a graph of interconnected trees."

Said Ila, "I think as an interconnected life form *the humongous fungus*, as I believe it's called, has that beat!"

Said Til, "Yes, and representing *that* as a graph is a non-starter. But the Aspen supertree, maybe! Not an exact representation, mind you, just a reasonable simulacrum! That would be a good exercise at the intersection of biology, computer science and discrete mathematics."

At this point the record abruptly shifts gears. And so must we.

³ Per [Randall Munroe](#).

⁴ See this popular software suite specializing in [graph visualization](#).

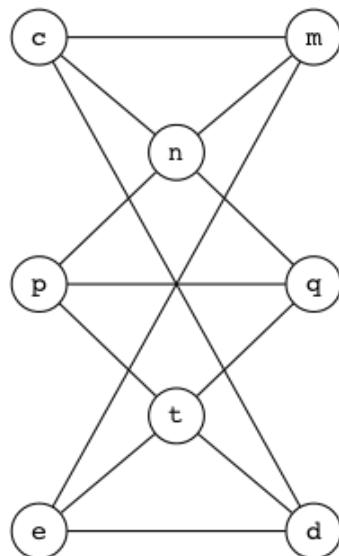


Figure 8.1: The RSA graph.

8.4 Cycles, Links, Paths

Just like trees, graphs have this intuitive knobs-and-lines visual representation (see Figure 8.2). The difference is that graphs allow **cycles**, a path from a node back to itself, which also implies the existence of more than one path from one node to another (recall that trees have exactly one). In this way graphs generalize trees analogous to the way relations generalize functions.

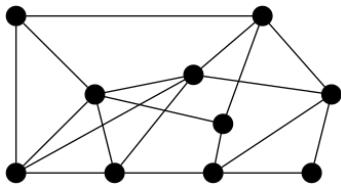


Figure 8.2: Knobs and lines.

Simple Graphs

- Correspond to **symmetric** binary relations.
- Have N , a set of *Nodes* or *Vertices* (singular *Vertex*), corresponding to the universe of some relation R .
- Have L , a set of *Links* or *Edges* or *Arches*, unordered pairs of usually distinct, possibly the same elements $u, v \in N$ such that uRv .
- Are compactly described as a pair $G = (N, L)$ (with R implied).

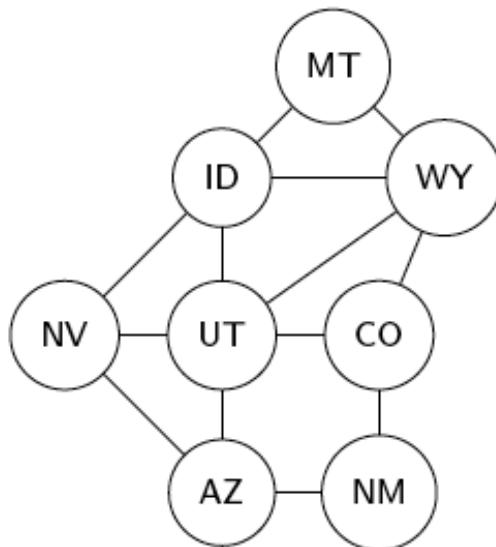
Figure 8.3 shows an example of a simple graph whose nodes represent the states in the Intermountain West.⁵

- $N = \{\text{ID}, \text{MT}, \text{WY}, \text{UT}, \text{CO}, \text{NV}, \text{AZ}, \text{NM}\}$
- $L = [(u, v) \mid u \text{ adjoins } v] = [(\text{ID}, \text{MT}), (\text{ID}, \text{WY}), (\text{ID}, \text{NV}), (\text{ID}, \text{UT}), (\text{MT}, \text{WY}), (\text{WY}, \text{UT}), (\text{WY}, \text{CO}), (\text{UT}, \text{CO}), (\text{UT}, \text{NV}), (\text{UT}, \text{AZ}), (\text{CO}, \text{NM}), (\text{AZ}, \text{NM}), (\text{NV}, \text{AZ})]$

⁵ To reduce clutter, 2-uppercase-character strings identifying states are displayed without the surrounding quotes. Entering in a Python REPL:
`ID,UT,CO = 'ID','UT','CO'`
for example, followed by
`[(ID, UT), (UT, CO)]`
yields `[('ID', 'UT'), ('UT', 'CO')]`.



Figure 8.3: Intermountain West adjoining states.



Path

In a graph with at least two nodes, a path from node s to node t is a sequence of *connected* links, $l_1, l_2, l_3, \dots, l_p$, where $p \geq 1$, and s is the first node of l_1 , t is the second node of l_p , and for $1 \leq i < p$, the second node of

l_i is the same as the first node of l_{i+1} . A path can also be represented as a sequence of nodes, $n_1, n_2, n_3, \dots, n_p$, where $p \geq 2$, which are the links' nodes listed without duplicates, so $l_i = (n_i, n_{i+1})$.

Path Length

The length of a path is the number of links (or nodes) in it. Among several paths, the **shortest path** is the one with the smallest number of links (or nodes) in it, and the **longest path** is likewise the one with the largest number.

Cycle

More formally, in a graph with at least 2 nodes, a cycle is a path from a node back to itself. In other words, the designated node is both the first node of l_1 and the second node of l_p .

Simple Path

In a graph with at least 2 nodes, a simple path is a *cycle-free* path, that is, a path with no cycles in it.

For example, in the graph in Figure 8.3, five paths from ID to AZ are:

1. [(ID, MT), (MT, WY), (WY, UT), (UT, AZ)]
2. [(ID, NV), (NV, AZ)]
3. [(ID, WY), (WY, UT), (UT, AZ)]
4. [(ID, UT), (UT, CO), (CO, NM), (NM, AZ)]
5. [(ID, UT), (UT, CO), (CO, WY), (WY, UT), (UT, AZ)]

Note that in the fourth path the symmetry (or unorderedness) of the link [AZ, NM] was exploited to make the equivalent [NM, AZ] the last link in the path. Note too that the fifth path has a cycle in it, so whereas the first four are simple paths, the fifth is *not* a simple path.⁶

Here are the alternate representations of these five paths:

1. [ID, MT, WY, UT, AZ]
2. [ID, NV, AZ]
3. [ID, WY, UT, AZ]
4. [ID, UT, CO, NM, AZ]
5. [ID, UT, CO, WY, UT, AZ]

In this alternate representation, the existence of a cycle in the fifth path is easily seen by UT's duplicate appearance. Here are both representations of a cycle starting and ending with UT (again exploiting symmetry, this time of [AZ, UT] and [UT, AZ]):

1. [(UT, CO), (CO, NM), (NM, AZ), (AZ, UT)]
2. [UT, CO, NM, AZ, UT]

⁶ How does allowing cycles imply the existence of more than one path from one graph node to another?

Exercise 337 List 7 of the simple paths (in the sequence-of-links representation) from MT to NM, including the shortest path(s) and the longest path(s).

Hint 337 These can be seen in Figure 8.3.

(TU, CO), (CO, MN), (MN, TM)
 (TU, SA), (SA, VN), (VN, ID), (ID, YW), (YW, TM)
 (MN, SA), (SA, VN)
 (VN, ID), (ID, TU), (TU, CO), (CO, YW), (YW, TM)
 (VN, CO), (CO, MN)
 (YW, TU), (TU, SA), (SA, VN), (VN, ID), (ID, MN, SA), (SA, VN)
 (VN, TU), (TU, CO), (CO, YW), (YW, TM)
 (MN, SA), (SA, VN), (VN, ID), (ID, YW), (YW, TM)
 (MN, SA), (SA, TU), (TU, ID), (ID, MN)
 (VN, CO), (CO, MN)

for most shortest paths:

337 ANSWER

Exercise 338 List 7 of the simple paths (in the sequence-of-nodes representation) from MT to NM, including the shortest path(s) and the longest path(s).

Hint 338 Start with your paths from the previous exercise.

ANSWER 338

8.5 Simplicity Abandoned

Multigraphs

- Are like simple graphs, but there may be *more than one* link connecting two given nodes.
- Have N , a set of *Nodes*.
- Have L , a set of *Links* (as *primitive* objects).⁷
- Have a function $f : L \rightarrow \{(u, v) \mid u, v \in N \wedge u \neq v\}$.
- Are compactly described as a triple: $G = (N, L, f)$.

For example, a transportation network whose nodes are cities, and whose links are segments of major highways is a multigraph.

Pseudographs

- Are like multigraphs, but links connecting a node to itself are allowed.
- The function $f : L \rightarrow \{(u, v) \mid u, v \in N\}$.
- Link $l \in L$ is a *loop* if $f(l) = (u, u) = \{u\}$.

For example, Figure 8.4 could be the graph of a campground with three campsites (nodes) linked by trails between them, and also having loops from each campsite back to itself.

8.6 Add Direction

Directed Graphs

- Correspond to arbitrary binary relations R , which need not be symmetric.
- Have N, L , and a binary relation R on N .

For example, $N = \text{people}$, $R = \{(x, y) \mid x \text{ loves } y\}$. For another example, $N = \text{players}$, $R = \{(x, y) \mid x \text{ beats } y\}$ (a tournament graph, as in Figure 8.5).

Exercise 339 Which node in the directed graph in Figure 8.5 represents Abu in the ping-pong tournament he was in?

Hint 339 See §8.2.

јом манъл юе тон' 3 сағп.

Answer 339 Node D because it shows Abu lost an edict number of

⁷ In object-oriented programming, a primitive object can be an instance of a class, allowing that object to maintain its separate identity from another instance, even though both instances may contain the same compound (i.e., *not primitive*) object (e.g., a link as a pair of nodes). For example:

```
link1 = Link(pair = (ID, UT))
link2 = Link(pair = (ID, UT))
link1.pair == link2.pair # True
link1 == link2 # False
```

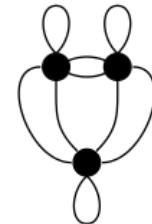


Figure 8.4: Pseudograph representing three campsites in a campground.

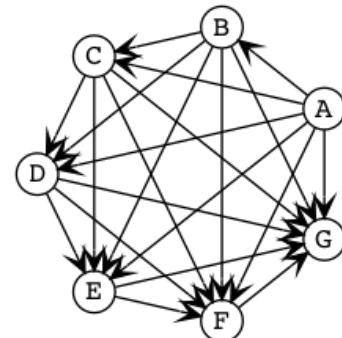


Figure 8.5: A tournament graph is a directed graph.

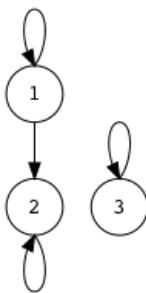


Figure 8.6: Digraph of a binary relation.

Directed graphs (**digraphs** for short) can represent binary relations. It is helpful to draw a graph of a relation that is described more abstractly. For an example of an abstract description, here is the relation on the set $\{1, 2, 3\}$ given by this list of ordered pairs: $[(1, 1), (1, 2), (2, 2), (3, 3)]$. Figure 8.6 shows the three-node four-link graph of that relation.

Exercise 340 Make connections between the four relation types and the existence or nonexistence of certain features in the graph of the relation:

- *Reflexive* = every node has a link that is a loop.
- *Symmetric* = there are no single link arrows (one-way trips) — every link arrow is a double one (roundtrip).
- *Antisymmetric* = there is no double arrow (going both ways) unless it is a loop.
- *Transitive* = there is no two-leg journey (think flight with a layover) without a shortcut (non-stop flight).

concrete in your mind.

Hint 340 Use the digraph of Figure 8.6 to make these connections

AT A GLANCE

Directed Multigraphs

- Are like directed graphs, but there may be more than one link from one node to another.
- Have N, L , and $f : L \rightarrow N \times N$.

Exercise 341 What is a familiar example, used every day, of a directed multigraph?

Hint 341 An activity called surfing comes to mind.

How about surfing? How about surfing? How about surfing? How about surfing?

8.7 Adjacency Counts

To study graphs in any meaningful way, you must learn the extensive vocabulary that has grown up around them. The following terminology is only a starting point:

Adjacency

Let G be an undirected graph with link set L . Let $k \in L$ be (or map to) the node-pair (u, v) .

- u and v are **adjacent**.
- u and v are **neighbors**.
- u and v are **connected**.
- Link k is **incident with** nodes u and v .
- Link k **connects** u and v .
- Nodes u and v are the **endpoints** of link k .

Degree

Let G be an undirected graph, $n \in N$ a node of G .

- The **degree** of n , denoted $\deg(n)$, is the number of links incident to it. Self-loops are counted twice.⁸
- A node with degree 0 is called **isolated**.
- A node with degree 1 is called **pendant**.⁹

⁸ For example, for the graph in Figure 8.6, $\deg(1) = 3$, $\deg(2) = 3$, and $\deg(3) = 2$.

⁹ Tree leaves are pendant nodes.

Degree Sequence

A sequence of the degrees of every node in a graph, listed in *nonincreasing* order (highest to lowest), is the graph's **degree sequence**.

For example, the degree sequence of the graph in Figure 8.2 is:

[5, 5, 4, 4, 4, 4, 3, 3, 2].

Exercise 342 What is the degree sequence of a **perfect** (full, complete) binary tree with 3 levels?

Hint 342 Draw it (or see Figure 7.6 in §7.1) and count carefully.

AT A GLANCE

Simple graphs with fewer than 5 or so nodes are easy to draw given their degree sequences.¹⁰ When there are more than 6 or 7 nodes it becomes more challenging. Try this one: [5, 3, 3, 3].

Drawing that graph with balance and symmetry is tricky, but suppose the degree sequence were given in 2-dimensional (2d) form? Then the x-y position of each number tells where that node is relative to the others, and drawing the graph is a breeze. For example, here's the 2d degree sequence for the graph shown in Figure 8.7.

¹⁰ E.g., [2, 1, 1] and [2, 2, 2] are easy — convince yourself by visualizing them in your mind before trying to draw them.

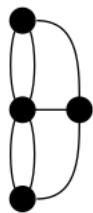


Figure 8.7: Four nodes, seven links.

Try drawing this bigger graph with degree sequence: [4, 4, 4, 4, 3, 3, 2, 2]. Here it is in 2d form:

4	4
3	
2	2
3	3
4	4

Exercise 343 In another multigraph example, what is the connection between the graph shown in Figure 8.7 and the “birth of graph theory”?

Hint 343 What do its nodes and links represent? What is its degree sequence?

3, 3].
لأنه. إنما ينبع المفهوم من المنهج. إنه ينبع من المنهج. إنه ينبع من المنهج.
Answer 343 ليس في المنهج أي اتجاه، وإنما هو مفهوم ينبع من المنهج.

Exercise 344 Does the graph shown in Figure 8.7 have the property “Eulerian”? (Look it up.)

For a graph to be Eulerian, there are necessary and sufficient conditions. Describe these conditions.

Hint 344 This is a good concept that stems from the very origins of graph theory (the reason it's named after Euler).

AT A GLANCE

Exercise 345 Explore the question: Does a graph’s degree sequence alone contain enough information to reconstruct the graph?

ramifications of the answer.

Hint 345 Explore means answer the question and investigate the

3) (1, 5), (1, 2), (3, 2), (5, 4), (5, 6).
 $[(1, 3), (1, 4), (1, 2), (5, 3), (5, 4), (5, 6)]$ աղ
 տակ: Առե ար ես նոստիլիք բարերերերունք (բարերերերունք):
 դպրություն [3, 3, 5, 5, 1, 1] կամ քե բարերերերունք իւ մուռ դպրություն
ԱՏ ՁԵՐ ԹԵՎՄԱՆ

Exercise 346 Draw the graphs of these ten 2d degree sequences:

$$\begin{array}{ccccc} 1 & & 2 & 2 & \\ & & & & 2 & 2 \\ & 2 & 1 & & & 1 & 1 \\ & & 2 & 2 & & & \\ & & & & 2 & 2 & \end{array}$$

$$\begin{array}{ccccc} 2 & 2 & & 1 & 3 & 1 \\ 3 & 3 & & & & \\ 1 & 1 & & 1 & & \end{array}$$

$$\begin{array}{ccccc} 1 & 1 & & 2 & 2 & \\ & 3 & & & & 1 \\ 1 & & 2 & 2 & & 1 \\ & & & & 2 & 2 & 0 \end{array}$$

Hint 346 Treat the ten clusters as separate degree sequences.

ԱՏ ՁԵՐ ԹԵՎՄԱՆ

8.8 Handshaking

Handshaking Theorem

Let G be an undirected (simple, multi-, or pseudo-) graph with node set N and link set L . Then the sum of the degrees of the nodes in N is twice the size of L :¹¹

$$\sum_{n \in N} \deg(n) = 2|L|.$$

A Handshaking Theorem Corollary¹²

Any undirected graph has an even number of nodes of odd degree.

¹¹ A good test of your understanding of this theorem is to write a succinct sentence explaining why the degrees of all the nodes should add up to twice the number of links.

¹² Immediately inferable companion theorem to the main theorem.

Exercise 347 Is there, i.e., can there possibly exist, a simple graph with degree sequence [3, 2, 1, 0]? Why or why not?

Hint 347 This is a real test of your understanding.

Sequence exists

8.9 Get Directed

Directed Adjacency

Let G be a directed (possibly multi-) graph and let k be a link of G that is (or maps to) (u, v) .

- u is **adjacent to** v .
 - v is **adjacent from** u .
 - k **comes from** u .
 - k **goes to** v .
 - k **connects** u to v .
 - k **goes from** u to v .
 - u is the **initial node** of k .
 - v is the **terminal node** of k .

Exercise 348 Choose a link in the directed tournament graph of Figure 8.5. Use all of the directed adjacency vocabulary to describe it.

Hint 348 Be succinct.

ALDASA & GUTIÉRREZ

Directed Path/Cycle

In a directed graph, a **directed path** respects the direction of the arrows, or in other words, it follows the one-way-ness of the links. So if $n_1, n_2, n_3, \dots, n_p$ is the sequence of nodes, links always go from n_i to n_{i+1} — where $1 \leq i < p$ — unless it is a **directed cycle**, in which case

there is also a link that goes from n_p to n_1 .

Exercise 349 Using the list-of-nodes representation, list all the directed paths in the tournament graph of Figure 8.5 from node A to F. How many of these paths include node G?

Hint 349 Be systematic about listing these.

C' It's important to notice that there is a link from n_p to n_1 .
None of them include node C'. You can't get anywhere from C' only to

- [A' E' E]
- [A' D' E' E]
- [A' D' E]
- [A' C' E' E]
- [A' C' D' E' E]
- [A' C' D' E]
- [A' C' E]
- [A' B' C' D' E]
- [A' B' C' E]
- [A' B' C' E' E]
- [A' B' C' D' E' E]
- [A' B' D' E' E]
- [A' B' E' E]
- [A' B' D' E]
- [A' B' E]
- [A' E]

ANSWER There are 16 directed paths from A to E:

Directed Degree

Let G be a directed graph, n a node of G .

- The **in-degree** of n , $\deg^-(n)$, is the number of links **going to** n .
- The **out-degree** of n , $\deg^+(n)$, is the count of links **coming from** n .
- The **degree** of n is just the sum of n 's in-degree and out-degree:

$$\deg(n) = \deg^-(n) + \deg^+(n).$$

Directed Handshaking Theorem

Let G be a directed (possibly multi-) graph with node set N and link set L . Then the sum of the in-degrees is equal to the sum of the out-degrees, each of which is equal to half the sum of the degrees, or the size of L :

$$\sum_{n \in N} \deg^-(n) = \sum_{n \in N} \deg^+(n) = \frac{1}{2} \sum_{n \in N} \deg(n) = |L|.$$

Note that the degree of a node is unchanged whether its links are directed or undirected.

8.10 Special Graphs

The following special types of simple undirected graphs are study-worthy:

- Complete graphs K_n
- Cycles C_n
- Wheels W_n
- n -Cubes Q_n
- Bipartite graphs
- Complete bipartite graphs $K_{m,n}$

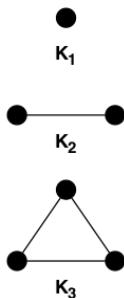


Figure 8.8: The first three complete graphs. Or are they? What about K_0 ? What is there to say about that graph?

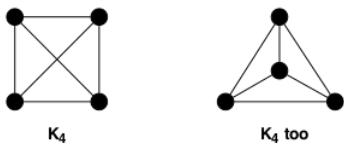


Figure 8.9: The graph K_4 and its planar version are structurally equivalent, but not visually equivalent. Recall the distinction first mentioned in §7.4.

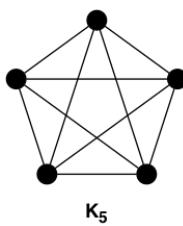


Figure 8.10: The fifth complete graph. Note that K_n graphs have n nodes, and a combinatorial number for how many links:

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \binom{n}{2}.$$

Complete Graphs

For any $n \in \mathbb{N}$, a **complete graph** on n nodes, denoted K_n , is a simple graph with n nodes in which every node is adjacent to every other node. Formally, given node set N and link set L : $\forall u, v \in N : u \neq v \rightarrow (u, v) \in L$.

Figure 8.8 shows K_1 , K_2 and K_3 . Before discussing the next one, K_4 , there is a graph property that deserves mention.

Planar Graphs

A **planar graph** is a graph that can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its links intersect only at their endpoints. In other words, it can be drawn in such a way that no link lines cross each other. Figure 8.9 shows a K_4 with line crossings and an alternate version with no line crossings.

The first non-planar complete graph is K_5 , shown in Figure 8.10. It can be described as a pentagram inscribed in a pentagon, and is the last complete graph we'll look at, but is one of two graphs that have a special property we'll look at later.

Planar graphs have many nice properties. Importantly, they model real-life routing situations where the lines cannot cross without causing problems. For example, in circuit board or VLSI design, where wires are the links between circuit components, if the wires are on the same layer they must not touch each other, or you get a short circuit. Pipes or cables in underground water or electrical networks work similarly. Airline routes have planes flying at certain altitudes, which are planar layers, and must avoid crossing to reduce chances of collisions.

Cycles

For any $n \geq 3$, a **cycle** on n nodes, denoted C_n , is a simple graph where

- $N = \{n_1, n_2, \dots, n_p\}$ and
- $L = \{(n_1, n_2), (n_2, n_3), \dots, (n_{p-1}, n_p), (n_p, n_1)\}$.

Note that C_3 is the same graph as K_3 . Figure 8.11 shows some others.

Exercise 350 As a function of n , how many nodes and links does a C_n have?

Hint 350 This is straightforward.

AT A GLANCE ANSWER

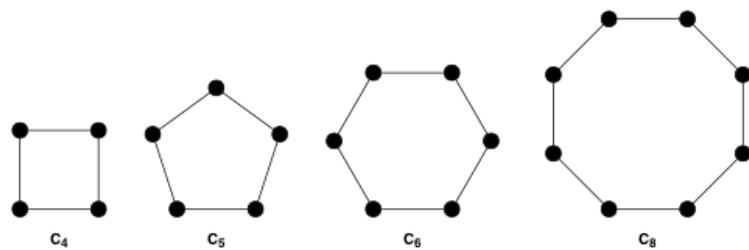


Figure 8.11: A few cycle graphs.

Wheels

For any $n \geq 3$, a **wheel**, denoted W_n , is a simple graph obtained by taking the cycle C_n and adding one extra node n_{hub} and p extra links: $\{(n_{hub}, n_1), (n_{hub}, n_2), \dots, (n_{hub}, n_p)\}$.

Figure 8.12 shows a few wheel graphs.

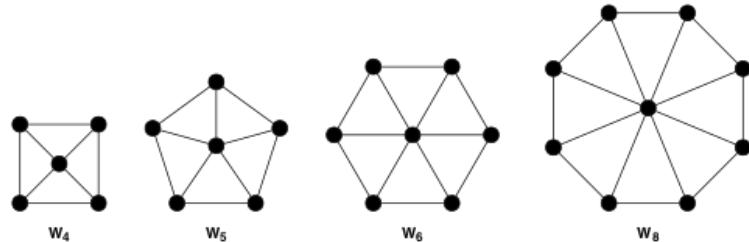


Figure 8.12: A few wheel graphs.

Exercise 351 As a function of n , how many nodes and links does a W_n graph have?

Hint 351 A wheel graph has one more node than a cycle graph, and a few more links.

AT A GLANCE ANSWER

Exercise 352 W_3 is the same as which complete graph?

Hint 352 Think alternatively.

AT A GLANCE

Exercise 353 What is the degree sequence for the wheel graph W_5 ? What is the degree sequence for the wheel graph W_n , in general?

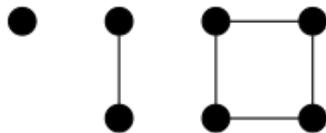
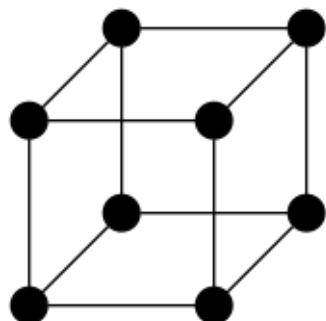
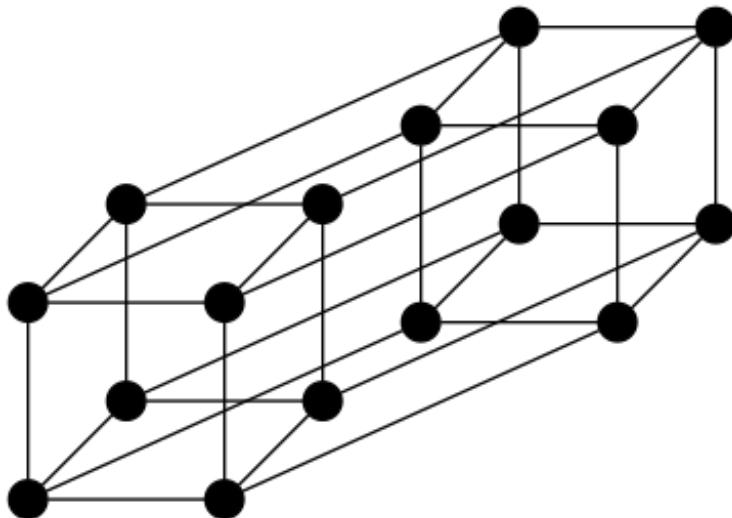
Hint 353 Remember to go from high degree to low degree.

The degree sequence for W_5 is $[5, 3, 3, 3, 3]$ (read from left to right).
Answer 353 The degree sequence for W_n is $[n, 3, 3, 3, \dots, 3]$.

***n*-Cubes**

For any $n \in \mathbb{N}$, the ***n*-cube** (or **hypercube**), denoted Q_n , is a simple graph consisting of two copies of Q_{n-1} connected together at corresponding nodes. Q_0 , the base case of this recursive definition, consists of a single node and no links.

Figures 8.13, 8.14, and 8.15 show Q_0 , Q_1 and Q_2 , followed by Q_3 and Q_4 . Q_n for $n \geq 5$ are exceedingly messy to draw.

Figure 8.13: Q_0, Q_1, Q_2 Figure 8.14: Q_3 Figure 8.15: Q_4

Exercise 354 Q_n graphs have 2^n nodes, and how many links?

Hint 354 The answer is a non-trivial function of n .

AT A GLANCE

Bipartite Graphs

For any $m, n \in \mathbb{Z}^+$, a **bipartite graph** is a simple undirected graph whose nodes can be divided (partitioned) into two (bi-) sets (parts), the first of size m , the second of size n , such that its links only go between (inter-) nodes in either set, never among (intra-) the nodes in one or the other set.

Figures 8.16 and 8.17 show two bipartite graphs.

Figure 8.16: A Bipartite Graph.

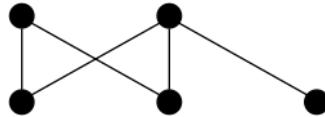
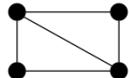
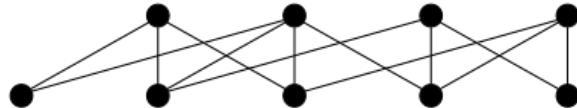


Figure 8.17: Another Bipartite Graph.



By contrast, the graphs shown in Figures 8.18 and 8.19 are *not* bipartite graphs.

Figure 8.18: Not a Bipartite Graph.

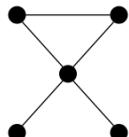


Figure 8.19: Also Not a Bipartite Graph.

Complete Bipartite Graphs

For any $m, n \in \mathbb{Z}^+$, a **complete bipartite graph** is a bipartite graph $K_{m,n}$ having all allowed links.¹³

Figures 8.20, 8.21, and 8.22 show three examples.

Exercise 355 What are m and n for the three $K_{m,n}$ graphs shown in Figures 8.20, 8.21, and 8.22?

Hint 355 Count m nodes in the top set, n nodes in the bottom set.

EE.8	Left A	Count the Complete Bipartite Graph	3	4
8.8.1	Top A	Count the Complete Bipartite Graph	3	3
8.8.2	A	Count the Complete Bipartite Graph	1	2
Exercise 355			m	n

AT A GLANCE ANSWER

Exercise 356 In what sense is a $K_{3,3}$ like a K_5 ?

"later" in the context of planar graphs.

Hint 356 This ties back to the special property we said "we'll look at

AT A GLANCE ANSWER

¹³ A good GPAO is to investigate bipartite graphs and complete bipartite graphs. What kinds of applications fit these types of graphs? What would a *tri*-partite graph look like? An n -partite graph?

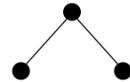


Figure 8.20: A Complete Bipartite Graph.

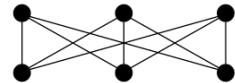


Figure 8.21: Another Complete Bipartite Graph.

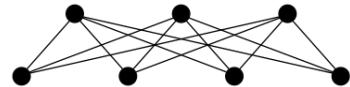


Figure 8.22: Yet Another Complete Bipartite Graph.

8.11 Graph Representation

Other than as drawings, graphs can be represented abstractly as various data structures:

- **Adjacency List** — a table with one row per node, listing each node's adjacent nodes.
- **Directed Adjacency List** — a table with one row per node, listing the terminal nodes of each link incident from that node.
- **Adjacency Matrix** — a matrix $A = [a_{ij}]$, where $a_{ij} = 1$ if (n_i, n_j) is a link, 0 otherwise.
- **Incidence Matrix** — a matrix $M = [m_{ij}]$, where $m_{ij} = 1$ if link l_j is incident with node n_i , 0 otherwise.

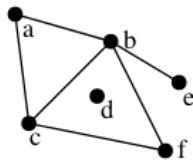


Figure 8.23: Graph with six nodes, six links.

Node	Adjacent Nodes
a	b c
b	a c e f
c	a b f
d	
e	b
f	c b

Table 8.1: Adjacency list of graph in Figure 8.23.

Examples of list representations are shown in Figures 8.23 and 8.24, and Tables 8.1 and 8.2,

A graph can be represented solely as a list of its links, from which a list of its nodes can be extracted as long as there are no isolated (degree 0) nodes. For example, Figure 8.25 and Tables 8.3, 8.4, 8.5, and 8.6 show the drawing of a small graph with 9 nodes and 21 links, and three different representations that can be extracted from its list of links, which rendered forwards (lower-numbered nodes come first, higher-numbered nodes are second) is:

```
[(1, 2), (1, 9), (2, 8),
 (2, 4), (2, 3), (2, 5),
 (2, 9), (3, 4), (3, 8),
 (3, 9), (4, 8), (4, 9),
 (4, 5), (5, 9), (5, 6),
 (5, 7), (6, 7), (6, 9),
 (7, 8), (7, 9), (8, 9)]
```

Reversing the order of the nodes in its links yields:

```
[(2, 1), (9, 1), (8, 2),
 (4, 2), (3, 2), (5, 2),
 (9, 2), (4, 3), (8, 3),
 (9, 3), (8, 4), (9, 4),
 (5, 4), (9, 5), (6, 5),
 (7, 5), (7, 6), (9, 6),
 (8, 7), (9, 7), (9, 8)]
```

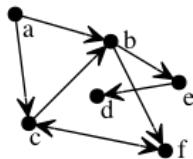


Figure 8.24: Directed graph with six nodes, seven links.

Initial Node	Terminal Nodes
a	b c
b	e f
c	b f
d	
e	d
f	c

Table 8.2: Directed adjacency list of graph in Figure 8.24.

Figure 8.25: A graph with 9 nodes and 21 links.

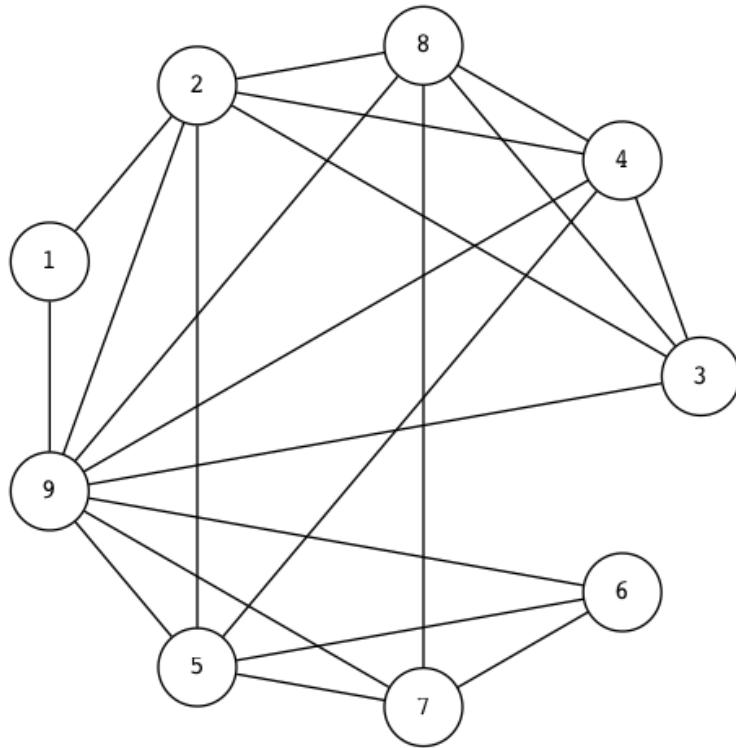


Table 8.3: Adjacency lists, merged and sorted to give unique node list [1, 2, 3, 4, 5, 6, 7, 8, 9].

Node	Adjacency List	Adjacency List From Reversed Links	Merged and Sorted
1	2 9		2 9
2	8 4 3 5 9	1	1 3 4 5 8 9
3	4 8 9	2	2 4 8 9
4	8 9 5	2 3	2 3 5 8 9
5	9 6 7	2 4	2 4 6 7 9
6	7 9	5	5 7 9
7	8 9	5 6	5 6 8 9
8	9	2 3 4 7	2 3 4 7 9
9		1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8

Table 8.4: Adjacency matrix for 9 node, 21 link graph.

	1	2	3	4	5	6	7	8	9
1	0	1	0	0	0	0	0	0	1
2	1	0	1	1	1	0	0	1	1
3	0	1	0	1	0	0	0	1	1
4	0	1	1	0	1	0	0	1	1
5	0	1	0	1	0	1	1	0	1
6	0	0	0	0	1	0	1	0	1
7	0	0	0	0	1	1	0	1	1
8	0	1	1	1	0	0	1	0	1
9	1	1	1	1	1	1	1	1	0

Table 8.5: Incidence matrix for 9 node, 21 link graph, part 1. Links label the columns of the table vertically.

	1	1	2	2	2	2	2	3	3	3
	2	9	8	3	4	5	9	4	8	9
1	1	1	0	0	0	0	0	0	0	0
2	1	0	1	1	1	1	1	0	0	0
3	0	0	0	1	0	0	0	1	1	1
4	0	0	0	0	1	0	0	1	0	0
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	1	0	0	0	0	0	1	0
9	0	1	0	0	0	0	1	0	0	1

Table 8.6: Incidence matrix for 9 node, 21 link graph, part 2. The top row of the table header is the first node of the link, the bottom row is the second node.

	4	4	4	5	5	5	6	6	7	7	8
	8	5	9	6	7	9	7	9	8	9	9
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0
0	1	0	1	1	1	0	0	0	0	0	0
0	0	0	1	0	0	1	1	0	0	0	0
0	0	0	0	1	0	1	0	1	1	0	0
1	0	0	0	0	0	0	0	1	0	1	0
0	0	1	0	0	1	0	1	0	1	1	1

Exercise 357 Draw this small graph having 8 nodes and the following 13 links (for less clutter, rendered without the quote marks):

```
[(p, q), (p, n), (q, n),
 (p, t), (q, t), (e, t),
 (d, t), (d, e), (m, e),
 (m, n), (m, c), (c, d),
 (c, n)]
```

Hint 357 You have seen this graph before.

ANSWER 357 See Figure 3.8.

Exercise 358 Give adjacency list, adjacency matrix, and incidence matrix representations of the graph of Exercise 357.

by hand.

Hint 358 Follow the examples shown, and this is easy enough to do

ANSWER 358 See Figure 3.9.

Exercise 359 Construct a graph using the following list of eight words as node labels:

dine, done, gone, tine, tone, wind, wine, wins

Link two nodes if they differ in only one letter in the same position (e.g., wind and wins differ in the fourth letter only, so that defines a link. But dine and tone differ in the first and second letters, so no link there).

Hint 359 It might help to line up pairs of words vertically.

is differenting atobd ait yomwglasse. Draw it as a triangular staircase with a square
ftiunq, wtiue, tuiue, (wtiue, wtiue), (wtiue, tuiue), (wtiue, (wtiue, (wtiue,
(wtiue, (wtiue, (wtiue, (wtiue, (wtiue, (wtiue, (wtiue, (wtiue, (wtiue, (wtiue,
ftiunq, done, done,

ANSWER 359 The links of this graph are [(dine, done), (dine, done),

Exercise 360 Give adjacency list, adjacency matrix, and incidence matrix representations of the graph of Exercise 359.

ous one.

Hint 360 This should be only slightly more difficult than the previous one.

AT A GLANCE

Exercise 361 Make a connection between graph theory and probability theory. Randomly select two nodes of the graph of Exercise 359. What is the probability that there is a link between those two nodes?

Hint 361 What does the graph of the probability space look like?

0.828145282145282 ≈ 36%

It is a graph with 10 nodes and 18 edges. It is fully connected, meaning every node is connected to every other node.

8.12 Graphs and Set Operations

Since graphs are defined as pairs of sets, naturally we can perform set operations on them. These operations include subset, union, complement, and intersection. Mostly it makes sense to do these operations on both sets in the pair, the set of nodes *and* the set of links. In the case of complement, however, it makes more sense to operate on just the set of links.

A **subgraph** of a graph $G = (N, L)$ is any graph $H = (O, P)$ where $O \subseteq N$ and $P \subseteq L$. For example, if G is the 10-node-19-link “knobs-and-lines” graph first shown in Figure 8.2 and reproduced in Figure 8.26, then H shown in Figure 8.27 is one of its subgraphs.

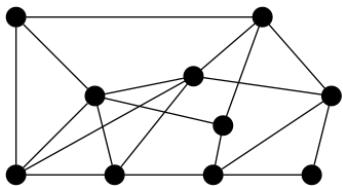


Figure 8.26: A graph G ...

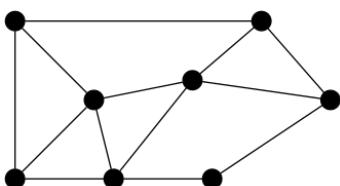


Figure 8.27: ... with subgraph H .

Exercise 362 Describe H in terms of subsets of G represented as:

```
([1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
 [(1, 2), (1, 3), (1, 7), (2, 4), (2, 6), (2, 5), (3, 4),
 (3, 6), (3, 7), (3, 8), (4, 5), (4, 7), (4, 8), (5, 9),
 (5, 10), (6, 9), (7, 8), (8, 9), (9, 10)]).
```

order:

Hint 362 Label G 's nodes left-to-right, top-to-bottom in increasing

AT A GLANCE

The **union** $G_1 \cup G_2$ of two simple graphs $G_1 = (N_1, L_1)$ and $G_2 = (N_2, L_2)$ is the simple graph $G = (N_1 \cup N_2, L_1 \cup L_2)$.

For example, let $N_1 =$

```
[‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’, ‘G’, ‘H’, ‘I’, ‘J’]
```

and $L_1 =$

```
[('A', 'B'), ('A', 'C'), ('A', 'F'), ('B', 'H'), ('C', 'D'),  
 ('C', 'I'), ('D', 'E'), ('E', 'F'), ('E', 'G'), ('F', 'J'),  
 ('G', 'I'), ('H', 'J'), ('I', 'J)]
```

as shown in Figure 8.28. Let $N_2 =$

['B', 'D', 'E', 'G', 'H']

and $L_2 =$

```
[('B', 'G'), ('B', 'H'), ('D', 'E'), ('D', 'H'), ('E', 'G')]
```

as shown in Figure 8.29. Then $G_1 \cup G_2$ can be drawn as a pentagram inside a pentagon connected by five spokes, as shown in Figure 8.30.

Exercise 363 The graph in Figure 8.30 is well-known in graph theory. What is its name? What are some of its attributes?

Hint 363 It's not hard to find this on the web.

թիւ թարմի ուժի ո բրաչ-զինունք՝ „Տես ին առիջը կօ առօղ.” Եթերսը՝ որո և 1888 շաբաթունքը ի դե ին ամսով բրգնելը ըստ ամեյց կօ առաջ երթուաց իւ թարմի բրուն. [If] իս առաջ պիտի լոյնուց այս թարմի բրաթ տեսուց այս ուժելի չամեյց առզ օստիւած- ունդունքը թարմի ուժի 10 առօղ առզ 12 լինք. Առօրդունք իւ Ուկրաինա **Անվան 303.** Լոյն թարմի իս առաջ ին Եթերսը թարմի ա շամեյց՝

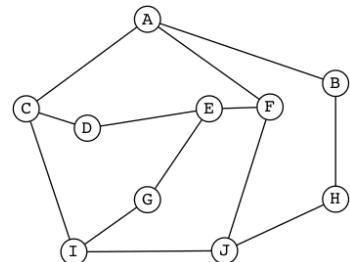


Figure 8.28: G_1 for graph union.

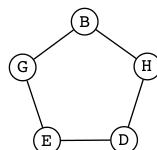


Figure 8.29: G_2 for graph union.

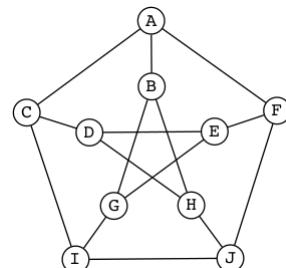


Figure 8.30: Graph union $G_1 \cup G_2$

The **complement** of a simple graph G , denoted \overline{G} , is the graph with the same node set (N) as G , but with none of the same links. Instead, the links of \overline{G} are all the links of the complete graph K_n , where $n = |N|$, except for those that are also links of G .

To form the complement of a graph G , take all its nodes, and take the complement of the link set of G with respect to the universe, which is the set of *all* links in a complete graph formed from G 's nodes. As in the following example, where G is shown in Figure 8.31, the graph in Figure 8.32 shows in red the complementary links, and the graph in Figure 8.33 is the end result of complementing G .

The **intersection** of two graphs can be defined in terms of the intersection of both of their node sets and link sets. For example, with $G_1 = (N_1, L_1)$ and $G_2 = (N_2, L_2)$, $G_1 \cap G_2 = (N_1 \cap N_2, L_1 \cap L_2)$.

Recall that the intersection of two sets can be defined in terms of the complement of the union of the complements of the two sets, by DeMorgan's law.

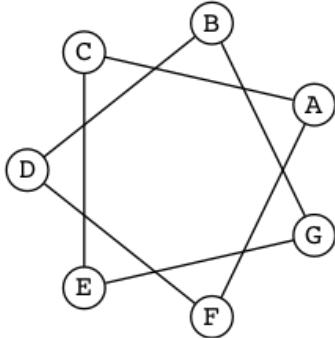


Figure 8.31: Graph to complement.

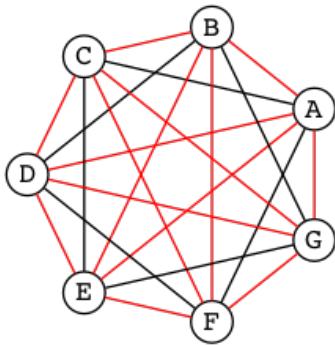


Figure 8.32: Complementary links in red.

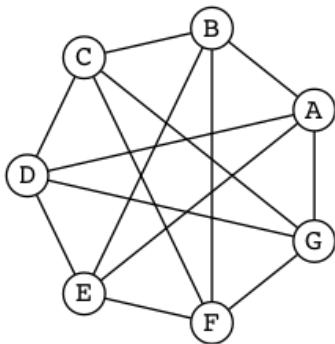


Figure 8.33: Graph complemented.

Exercise 364 Create two graphs with node sets and link sets that have nonempty intersections. Do the graph intersection of these two graphs two ways, the way described by the definition, and the alternate way using DeMorgan's law.

Hint 364 Make the two graphs small, but not too small.

AT A GLANCE

Exercise 365 Not to be confused with **graph intersection**, the **intersection graph** of a collection of sets A_1, A_2, \dots, A_n is the graph that has a node for each of these sets and has a link connecting the nodes representing two sets if these sets have a nonempty intersection.

Construct the intersection graph of this collection of five sets:

Set Name	Set Elements
A_1	$\{-4, -3, -2, -1, 0\}$
A_2	$\{-2, -1, 0, 1, 2\}$
A_3	$\{-6, -4, -2, 0, 2, 4, 6\}$
A_4	$\{-5, -3, -1, 1, 3, 5\}$
A_5	$\{-6, -3, 0, 3, 6\}$

What is the degree sequence of this graph?

Hint 365 Draw the graph to make it easier to see the answer.

AT A GLANCE SEE ANSWER

Exercise 366 Here is an adjacency-list representation of a graph implemented as a Python dictionary mapping nodes (represented as numbers) to lists of adjacent nodes. What does this graph model?

```
mystery_graph = {  
    1:[22,13],2:[21,23,14],3:[11,22,24,15],4:[12,23,25,16],  
    5:[13,24,26,17],6:[14,25,27,18],7:[15,26,28],8:[16,27],  
    11:[32,23],12:[31,33,24],13:[21,32,34,25],14:[22,33,35,26],  
    15:[23,34,36,27],16:[24,35,37,28],17:[25,36,38],18:[26,37],  
    21:[42,33],22:[41,43,34],23:[31,42,44,35],24:[32,43,45,36],  
    25:[33,44,46,37],26:[34,45,47,38],27:[35,46,48],28:[36,47],  
    31:[52,43],32:[51,53,44],33:[41,52,54,45],34:[42,53,55,46],  
    35:[43,54,56,47],36:[44,55,57,48],37:[45,56,58],38:[46,57],  
    41:[62,53],42:[61,63,54],43:[51,62,64,55],44:[52,63,65,56],  
    45:[53,64,66,57],46:[54,65,67,58],47:[55,66,68],48:[56,67],  
    51:[72,63],52:[71,73,64],53:[61,72,74,65],54:[62,73,75,66],  
    55:[63,74,76,67],56:[64,75,77,68],57:[65,76,78],58:[66,77],  
    61:[73],62:[74],63:[71,75],64:[72,76],65:[73,77],66:[74,78],  
    67:[75],68:[76]}
```

Hint 366 Click here to see the mystery graph.

AT A GLANCE SEE ANSWER

8.13 Exploration

¹⁴ Ramsey Theory, according to the [linked article](#), is named after the British mathematician and philosopher Frank P. Ramsey, [and] is a branch of mathematics that studies the conditions under which order must appear. Problems in Ramsey theory typically ask a question of the form: ‘how many elements of some structure must there be to guarantee that a particular property will hold?’”

We have a record of Ila’s and Abu’s very first exploration into Ramsey Theory¹⁴ territory, as Til had suggested it would be a grand GPAO, allowing them to expand on their knowledge of complete graphs, and also allowing them to hone their logical thinking skills.

Having first asked Ila what her top two favorite colors were, they went with green (her eye color) and purple (the color of royalty, as she had royals in her bloodline). They decided that green means friends, purple strangers. Why? They were looking at a complete graph K_n as modeling n people connected by a friendship — or acquaintanceship — relation, and its opposite — nonacquaintanceship. But for simplicity (fewer syllables to say out loud), they decided to treat any two people as either friends or strangers.

Ila said, “It’s like it’s a heterogeneous graph, but almost not.” “Right,” said Abu. “There are two different types of links, but the second type is really just the absence of the first type. At any rate, there’s no third option, like semi-acquainted.” Ila said, “That’s why two colors are all we need.”

“Right!” Abu said, “So let’s get concrete, and say we have five people — call them A, B, C, D, and E, to make short node names.”¹⁵

With unusual speed and dexterity, Ila used her graph-drawing tools to produce the graphs in Figures 8.34 and 8.35 (and the rest), two random K_5 graphs, with links colored.

“Let’s see,” said Abu, “in this first graph (Figure 8.34), we have AB, AC, BC, CD, CE, and DE colored green, and AD, AE, BD, and BE colored purple.”

Ila said, “I see something interesting in both graphs. If you have any same-colored triangles — what did Til call them?” Abu said, “It was a mouthful.” “I recorded it,” Ila said, searching through her notes. “Ah, here it is: *embedded monochromatic K_3 subgraphs!*” “I like triangles better,” said Abu. “So do I,” said Ila, and continued, “So if you have any of these triangles at least one side has to be an outer link — outer meaning around the periphery of the pentagon.”

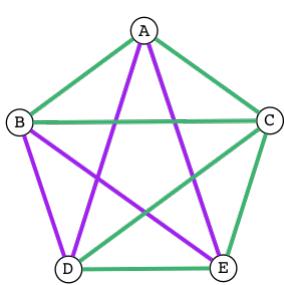


Figure 8.34: Randomly Colored K_5 .

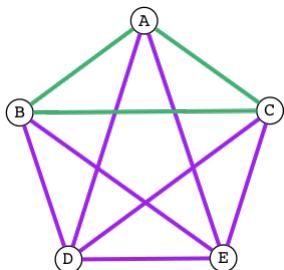


Figure 8.35: Another Randomly Colored K_5 .

"Yes, I see it too," said Abu. Ila said, "So what if we colored *all* the outer links green and all the inner ones purple?" (See Figure 8.36.)

Ila said, "Now we have many *pairs* of friends, and likewise, many *pairs* of strangers, but no triples — *triangles* — of either."

"So how did the Ramsey question go?" said Abu, looking at his notes. "If you want to guarantee that at least three mutual friends or three mutual strangers exist in a group of people, what is the minimum number of people you need? It has to be more than five, as we just saw with only five people there is a way to avoid this scenario." Ila said, "So we know it's at least six." "Right," said Abu, "so can we argue, from logical necessity, that six it is — no more, no less? That there *must exist* in a K_6 graph, whose links are colored either green or purple, at least one green or one purple triangle? Meaning that it can't be avoided, no matter how you color the graph?" Ila said, "Yes, well, according to Ramsey, it must be *absolutely* unavoidable. Avoiding it would have to be logically impossible."

Abu mused, "So how do we make that argument? Tell you what. Let's *not* do it by looking at and eliminating every possible coloring, since a K_6 has 15 links, that would make $2^{15} = 32768$ different colorings!" Ila said, "Remind me how you got that?" Abu replied, "You have two color choices to make for each link, green or purple." "Oh yeah," said Ila, "that 2^n thing again: $2 \cdot 2 = 2^{15}$, 15 links, 2 choices for each, 32768 total colorings, got it!"

Abu said, "So we have to reason without using this silly, brute-force kind of thinking." Both Abu and Ila went silent for several seconds. Abu broke the silence with a sudden clearing of his throat. "I have an idea," he said. "Suppose we use one node (say C) as representative of all of them. C has five links, and with just two colors at our disposal, at least three of the links *must be* the same color!" "Wait," said Ila, "you can color green any of the five links — 0, 1, 2, 3, 4, or all 5 of them — so how can you say that?"

"Well," said Abu, "if 0, 1 or 2 (less than 3) are green, then 5, 4, or 3 (at least 3) will have to be purple, right?" "Oh, right," said Ila, "I wasn't thinking generally." "I know," said Abu, "it's like we need to pick one color to focus on, but the problem doesn't say at least three friends, it doesn't say at least three strangers, it says at least three mutual friends *or* three mutual strangers. It's got this *yin-yang* feel to it!"

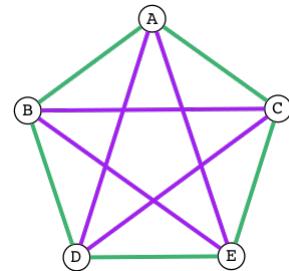


Figure 8.36: Outer Green Inner Purple K_5 .

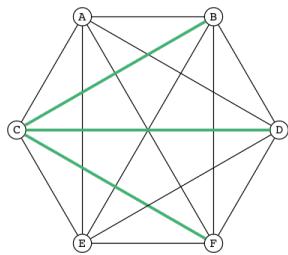


Figure 8.37: Three Green Links

"So generally speaking," said Ila, "we could take our green-primary, purple-secondary scenario – *yin*, swap green and purple and have the same *yang* argument!" "That cuts our work in half — not bad," Abu said, exuberantly. "So now draw *yin*, and just for grins make the middle three links *grin* — er, I mean green!" (See Figure 8.37.)

"Now look at links BD, BF, and DF," said Abu. "Coloring any of them green would give us a green triangle — BCD, BCF, or CDF." (See Figures 8.38, 8.39, and 8.40.)

Ila could barely contain her excitement. "I see it!" she said. "If we can't color any of them green, they must all be purple!"

Abu grinned his biggest grin. "BDF is forced to be a purple triangle — 'tis Un. Avoid. Able!" (See Figure 8.41.)

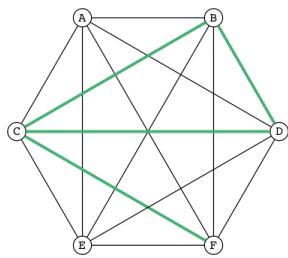


Figure 8.38: Green Triangle BCD.

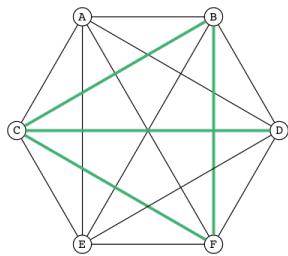


Figure 8.39: Green Triangle BCF.

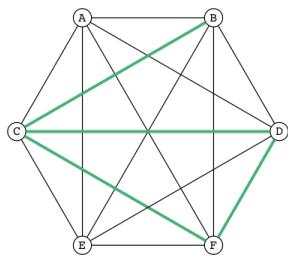


Figure 8.40: Green Triangle CDF.

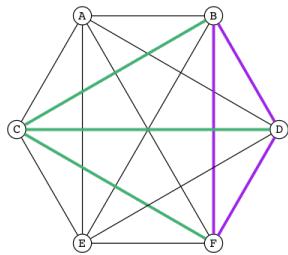


Figure 8.41: Purple Triangle BDF.

Awestruck, Ila said, “So there *must be* three mutual friends, or three mutual strangers — having no three be mutual friends *and* simultaneously no three be mutual strangers is impossible.”¹⁶

Still grinning from ear to ear, Abu said, “That, my friend, is an airtight argument — congratulations!” Ila beamed as well, “Kudos to both of us. Til will be proud!”

8.14 Summary of Terms, Definitions, and Theorems

A **simple graph** corresponds to a **symmetric** binary relation; it has N , a set of *Nodes* or *Vertices* (singular *Vertex*), corresponding to the universe of some relation R ; it also has L , a set of *Links* or *Edges* or *Arcs*, unordered pairs of usually distinct, possibly the same elements $u, v \in N$ such that uRv . It can be compactly described as a pair $G = (N, L)$ (with R implied).

In a graph with at least two nodes, a **path** from node s to node t is a sequence of *connected* links, $l_1, l_2, l_3, \dots, l_p$, where $p \geq 1$, and s is the first node of l_1 , t is the second node of l_p , and for $1 \leq i < p$, the second node of l_i is the same as the first node of l_{i+1} . A path can also be represented as a sequence of nodes, $n_1, n_2, n_3, \dots, n_p$, where $p \geq 2$, which are the links’ nodes listed without duplicates, so $l_i = (n_i, n_{i+1})$.

The **path length** is the number of links (or nodes) in the path. Among several paths, the **shortest path** is the one with the smallest number of links (or nodes) in it, and the **longest path** is likewise the one with the largest number.

In a graph with at least 2 nodes, a **cycle** is a path from a node back to itself.

In a graph with at least 2 nodes, a **simple path** is a *cycle-free* path, that is, a path with no cycles in it.

A **multigraph** is like a simple graph, but there may be *more than one* link connecting two given nodes. It has N , a set of *Nodes*, L , a set of *Links* (as *primitive objects*), and a function $f : L \rightarrow \{(u, v) \mid u, v \in N \wedge u \neq v\}$. It is compactly described as a triple: $G = (N, L, f)$.

¹⁶Generalizing, the Ramsey function R with two parameters, $R(i, j)$, returns the minimum n for which in a K_n graph where the links are colored with two colors, there must exist either an embedded K_i subgraph whose links are all of the first color, or an embedded K_j subgraph whose links are all of the second color. As Abu and Ila showed, $R(3, 3) = 6$. Many others of these so-called *Ramsey numbers* are known and are provable, e.g., $R(4, 4) = 18$ and $R(4, 5) = 25$, but most are not. For example, it is known that $R(5, 5)$ is at least 43 and at most 48, but which number between 43 and 48 is the exact value is unknown. ($R(5, 5)$ is at least 43, because, as *Geoff Exoo* (and probably others) found, there is a K_{42} that is link-2-colorable but has no same-colored triangles.)

There is no efficient algorithm for calculating Ramsey numbers. There have been some algorithms that help calculate the lower and upper bounds, but even looking at $R(5, 5)$ shows how difficult it is to calculate. If we assume that $R(5, 5) = 43$, than for a K_{43} graph colored with 2 colors, we would have to show that it holds true for $\binom{43}{2} = 903$ edges, which equals 2^{903} possible graphs. That number is unimaginably larger than the total number of atoms in the universe.

Here’s a great quote from Erdős via Joel Spencer on this matter (quoted in the [Wikipedia article on Ramsey’s Theorem](#)):

“Erdős asks us to imagine an alien force, vastly more powerful than us, landing on Earth and demanding the value of $R(5, 5)$ or they will destroy our planet. In that case, he claims, we should marshal all our computers and all our mathematicians and attempt to find the value. But suppose, instead, that they ask for $R(6, 6)$. In that case, he believes, we should attempt to destroy the aliens.”

A **pseudograph** is like a multigraph, but links connecting a node to itself are allowed. It has a function $f : L \rightarrow \{(u, v) \mid u, v \in N\}$, where link $l \in L$ is a *loop* if $f(l) = (u, u) = \{u\}$.

A **directed graph**, or **digraph**, corresponds to an arbitrary binary, not necessarily symmetric, relation R . It needs N, L and the binary relation R on N to describe it fully.

A **directed multigraph** is like a directed graph, but there may be more than one link from one node to another. It has N, L , and $f : L \rightarrow N \times N$.

Let G be an undirected graph with link set L . Let $k \in L$ be (or map to) the node-pair (u, v) .

- u and v are **adjacent**.
- u and v are **neighbors**.
- u and v are **connected**.
- Link k is **incident with** nodes u and v .
- Link k **connects** u and v .
- Nodes u and v are the **endpoints** of link k .

Let G be an undirected graph, $n \in N$ a node of G .

- The **degree** of n , denoted $\deg(n)$, is the number of links incident to it. Self-loops are counted twice.
- A node with degree 0 is called **isolated**.
- A node with degree 1 is called **pendant**.

A graph's **degree sequence** is a listing of the graph's nodes' degrees in *nonincreasing* order (highest to lowest).

The **Handshaking Theorem** says if G is an undirected (simple, multi-, or pseudo-) graph with node set N and link set L , then the sum of the degrees of the nodes in N is twice the size of L :

$$\sum_{n \in N} \deg(n) = 2|L|.$$

A **Handshaking Theorem Corollary** says that any undirected graph has an even number of nodes of odd degree.

Let G be a directed (possibly multi-) graph and let k be a link of G that is (or maps to) (u, v) .

- u is **adjacent to** v .
- v is **adjacent from** u .
- k **comes from** u .
- k **goes to** v .
- k **connects** u to v .
- k **goes from** u to v .
- u is the **initial node** of k .
- v is the **terminal node** of k .

In a directed graph, a **directed path** respects the direction of the arrows, or in other words, it follows the one-way-ness of the links. So if $n_1, n_2, n_3, \dots, n_p$ is the sequence of nodes, links always go from n_i to n_{i+1} — where $1 \leq i < p$ — unless it is a **directed cycle**, in which case there is also a link that goes from n_p to n_1 .

Let G be a directed graph, n a node of G .

- The **in-degree** of n , $\deg^-(n)$, is the number of links **going to** n .
- The **out-degree** of n , $\deg^+(n)$, is the count of links **coming from** n .
- The **degree** of n is just the sum of n 's in-degree and out-degree:

$$\deg(n) = \deg^-(n) + \deg^+(n).$$

The **Directed Handshaking Theorem** says that if G is a directed (possibly multi-) graph with node set N and link set L , then the sum of the in-degrees is equal to the sum of the out-degrees, each of which is equal to half the sum of the degrees, or the size of L :

$$\sum_{n \in N} \deg^-(n) = \sum_{n \in N} \deg^+(n) = \frac{1}{2} \sum_{n \in N} \deg(n) = |L|.$$

For any $n \in \mathbb{N}$, a **complete graph** on n nodes, denoted K_n , is a simple graph with n nodes in which every node is adjacent to every other node. Formally, given node set N and link set L : $\forall u, v \in N : u \neq v \rightarrow (u, v) \in L$.

A **planar graph** is a graph that can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its links intersect only at their endpoints. In other words, it can be drawn in such a

way that no link lines cross each other.

For any $n \geq 3$, a **cycle** on n nodes, denoted C_n , is a simple graph where

- $N = \{n_1, n_2, \dots, n_p\}$ and
- $L = \{(n_1, n_2), (n_2, n_3), \dots, (n_{p-1}, n_p), (n_p, n_1)\}$.

For any $n \geq 3$, a **wheel**, denoted W_n , is a simple graph obtained by taking the cycle C_n and adding one extra node n_{hub} and p extra links: $\{(n_{hub}, n_1), (n_{hub}, n_2), \dots, (n_{hub}, n_p)\}$.

For any $n \in \mathbb{N}$, the **n -cube** (or **hypercube**), denoted Q_n , is a simple graph consisting of two copies of Q_{n-1} connected together at corresponding nodes. Q_0 , the base case of this recursive definition, consists of a single node and no links.

For any $m, n \in \mathbb{Z}^+$, a **bipartite graph** is a simple undirected graph whose nodes can be divided (partitioned) into two (bi-) sets (parts), the first of size m , the second of size n , such that its links only go between (inter-) nodes in either set, never among (intra-) the nodes in one or the other set.

For any $m, n \in \mathbb{Z}^+$, a **complete bipartite graph** is a bipartite graph $K_{m,n}$ having all allowed links.

Some graph representations are:

- **Adjacency List** — a table with one row per node, listing each node's adjacent nodes.
- **Directed Adjacency List** — a table with one row per node, listing the terminal nodes of each link incident from that node.
- **Adjacency Matrix** — a matrix $A = [a_{ij}]$, where $a_{ij} = 1$ if (n_i, n_j) is a link, 0 otherwise.
- **Incidence Matrix** — a matrix $M = [m_{ij}]$, where $m_{ij} = 1$ if link l_j is incident with node n_i , 0 otherwise.

A **subgraph** of a graph $G = (N, L)$ is any graph $H = (O, P)$ where $O \subseteq N$ and $P \subseteq L$.

The **union** $G_1 \cup G_2$ of two simple graphs $G_1 = (N_1, L_1)$ and $G_2 = (N_2, L_2)$ is the simple graph $G = (N_1 \cup N_2, L_1 \cup L_2)$.

The **complement** of a simple graph G , denoted \bar{G} , is the graph with the same node set (N) as G , but with none of the same links. Instead, the links of \bar{G} are all the links of the complete graph K_n , where $n = |N|$, *except* for those that are also links of G .

The **intersection** of two graphs can be defined in terms of the intersection of both of their node sets and link sets. $G_1 = (N_1, L_1) \wedge G_2 = (N_2, L_2) \rightarrow G_1 \cap G_2 = (N_1 \cap N_2, L_1 \cap L_2)$.

9

Languages and Grammars

9.1 Help or Hurt

Til had asked Ila and Abu to come prepared to discuss how languages either facilitate or impede communication. They both came and shared two quotes they had found by mathematicians¹ and a scientist² on languages (including the language of mathematics) as both boon and bane.

Then Ila had a thought. “I know programming languages facilitate communication with computers, but I don’t see the impediment.”

“I do,” said Abu. “Not knowing any programming languages, except a little Python, impedes my communication with you when you’re talking programming.” “Yes,” said Ila, “but your ignorance is not the fault of the languages. So, Til, did you mean *ignorance of languages* is the impediment?”

“No — not necessarily,” said Til. “Sometimes a perfectly natural and known language like English is used, or I should say *misused*, to obfuscate rather than communicate — to hide rather than reveal meaning. But I’ll come back to this point later.”

“For now, the question ‘What is a language?’ with its obvious answer — a language is something used to communicate — is too general in one sense, and too specific in another. Because as we’ll see, languages *can* be generalized to be sets of strings, but made more specific by *how* those sets are defined.”

“Phrase-Structure Grammars — PSGs — are what define *which* strings are valid members of the languages *associated with* PSGs. Now let me be clear what that association is. We say PSGs *generate* languages — or languages are *generated* (built) by PSGs.”

“In English (or any natural, human language) you follow the rules when you adhere to the grammar. A main component of a formally defined grammar is the set of *productions* — the rules that *produce* valid strings in the language. In essence, applying these rules is how different strings of words (phrases, sentences) are generated. Deliberately, or even ignorantly violating these rules leads to confusion and failed com-

¹ More precisely, math educators. “What we present may not resemble math, because we avoid the cryptic equations, formulas, and graphs that many people have come to know and fear as mathematics. Indeed, those symbols are the memorable icons of an often-forbidding foreign language of mathematical jargon, but it’s not the only language of mathematics and it does not reside at the center of the subject. The deepest and richest realms of mathematics are often devoid of the cryptic symbols that have baffled students through the generations. Ideas — intriguing, surprising, fascinating, and beautiful — are truly at the heart of mathematics.” — Edward B. Burger and Michael Starbird, from their book *Coincidences, Chaos, and All That Math Jazz — Making Light of Weighty Ideas*.

² “To many people who are not physicists, modern physics seems to have left the solid world of understandable here-and-now to enter a weird realm of uncertainties and strange, ephemeral particles that have whimsical names and dubious existence. What has actually happened is that physics has gone far beyond the point where ordinary, everyday experiences can provide a kind of human analogy to the things that the physicists are studying. It is a problem of language.”

“The vocabulary and syntax of human language evolved to describe the workings of the everyday world, the world we can see, hear, touch, taste and smell. Words were simply not intended to describe things unimaginably great or incredibly small, far beyond the range of our unaided senses, where the rules of the game are changed. The true language of physics is mathematics.” — Nobel laureate physicist Sheldon L. Glashow, in his book *Interactions: A Journey Through the Mind of a Particle Physicist and the Matter of This World*.

munication. Because the grammatical rules of English are so complex, ambiguity and misunderstanding frequently lurk beneath the surface.”

“Take what you said earlier, Ila. ‘I know programming languages facilitate X.’ (I’m abbreviating.) One sentence, or two?”

I know programming. Languages facilitate X.

Abu said, “It seems pretty clear from the context that it’s one sentence.” Ila said, “I agree. I don’t see the problem.”

Til said, “It’s *not* a problem — for *you*. Missing the context, and minus the fluidity and rapidity with which you uttered that one sentence, however, it could be misconstrued — legally but erroneously parsed — into two. Take this three-word saying: *All Things Link*.³ You *could* make sense of each of the other five permutations, but only by giving a lot of context:”

1. All Link Things.
2. Link All Things.
3. Link Things All.
4. Things All Link.
5. Things Link All.

“So,” said Ila, “are you saying word order causes communication problems?”

9.2 Order Matters

“Here’s a thought,” said Abu. “Have you ever tried to make sense of some kinds of poetry? Poets are always scrambling word order to fit some rhyme or meter scheme. Like in a poem by Wordsworth⁴ that I recently read:”

*Bliss was it in that dawn to be alive.
But to be young was very Heaven!*

Said Til, “Or from a more familiar poem: ‘Quoth the raven’ instead of ‘The raven said.’ Still perfectly correct. In fact, saying ‘The raven quoth’ is *not* grammatical, by the very special rule attached to that very archaic word.”

“So, not so much word order as word choice. Remember I said I would come back to this point later — about the misuse of English to obfuscate? **Jargon** is what I’m talking about. This misuse of language erects barriers in the guise of facilitators. We justify ourselves in the use of jargon because of the benefits — and there are benefits — but do we ever weigh them against the costs?”

“What do you consider the benefits?” Abu said, and Ila nodded — she wanted to know too.

³ Or, as Abu discovered in his poetry collection, this short poem by Francis Thompson: All things [...] linked are [...] thou canst not stir a flower without troubling of a star.

⁴ The French Revolution as It Appeared to Enthusiasts at Its Commencement.

"Benefit-wise," said Til, "jargon saves time and space by compressing information that needs sharing, but somewhat-intangible-cost-wise that very efficiency impedes understanding by outsiders of what insiders are saying. Hiding meaning from the uninitiated is such a powerful urge, and human vanity is an imp that eggs us on."

"For example, what does *partial template specialization* mean? And what is the cardinality of the power set?"

Said Ila, "I can tell you the answer to your second question, but I have no clue about the first."

Said Abu, "I agree, but regarding your second question, do you think it's better to put it more simply, like, how many subsets does a set of size **n** have?"

"I do," said Til. "Math is a language that desperately needs less jargon, more clarity. And not to keep you in suspense, well, not to keep you from the adventure of discovery, either, so for a little hint, partial template specialization is a very obscure 'feature' of the C++ language."

Ila said, "I'll check it out, but just from the sound of it I'm glad C++ is not the language my business uses!" Abu added, "I'll be sure to avoid it, if I ever need to hire a programmer for my business!"

"Speaking of business," said Til, "you've no doubt heard the stories about when a business consultant, tongue firmly in cheek — or not — randomly chooses three words from three different lists to create for client consideration impressive-sounding, meaningless phrases, like

customer value trajectory, or *stratified business intelligence*, or *hypercubic mission criticality*."

"Wow, did you just make those up!?" said Ila. "I hear stuff like that all the time from the consultants my company hires. It's worse than nonsense, if you ask me."

"But not all of it is so obviously bad," said Til. "Let me put it this way. Proclivities — what a nice word! Many people have proclivities, inclinations, predispositions to use more words, or bigger words, or *shinier* words than necessary to get what they want, or what they think they want. Flattery is full of this abuse of language."

Abu rose to the challenge: "Your mellifluous speech shows a penchant for pulchritudinous word marshalling."

Ila snorted. "You mean marshmallowing — sickly sweet, with no nutritional value!"

Abu admired, "Nice work, Ila! You're becoming quite the punster!" "Learned from the best, Abu!" teased Ila.

Said Til, "Let's go back to talking about more manageable problems — those we tackle in mathematics. Mathematical language, unlike natural language, is precise and unambiguous. Equations — tautologies — always true. Never a doubt. Pure syntax without the clouding confusion of semantics."

9.3 Logic, Beads, and Strings

"That's the official story," Til went on. "Now let me qualify that a bit. Kurt Gödel, one of the, if not *the* most brilliant mathematical logicians of all time, once said:"

The more I think about language, the more it amazes me that people ever understand each other.

"What amazes me about mathematicians, who are people too, is that they are such poor writers — when writing mathematics, at least. I alluded to this a little bit ago. Math writing is notorious for its lack of clarity,⁵ despite its claim of delivering unadulterated truth."

Til sent them a link, then said. "Think about what you know about the language of logic. It had something of a learning curve when you first encountered it, right? Formal logic is a formidable but foundational system of thought, a way to give precision to thought and reasoning, that can nonetheless trip up the unwary. Since I just mentioned Kurt Gödel, let me give you a description of formal systems,⁶ or at least, the rules of formal systems, from a book written about him. This passage pauses while expressing the point of view that 'mathematics is merely syntactic;' and goes on:"

its truth derives from the rules of formal systems, which are of three basic sorts: the rules that specify what the symbols of the system are (its "alphabet"); the rules that specify how the symbols can be put together into what are called well-formed formulas, standardly abbreviated "wff" and pronounced "woof"; and the rules of inference that specify which wffs can be derived from which.

Said Abu, "Now that you mention it, I recall some of the rules we learned for the wffs of propositional logic, like ' $p \rightarrow (q \vee r)$ ' is one but ' $\rightarrow p(\vee qr)$ ' is not."

Said Ila, "Or even more 'unwffified', a scrambling like ' $pq \vee (r \rightarrow)$ '."

Said Til, "Right! But consider that, in English grammar, where the rules are many and varied, and sometimes downright mysterious, lots of scramblings sort of make sense. However, some simple wff like 'he went to the store' is quite obviously *not* well formed if you shuffle it to be 'to he store the went'."

Said Abu, "So, if I recall correctly, there were only four rules for composing propositions."

Said Til, "Correct. However, that simple grammar didn't take into account parentheses for grouping — even though I noticed you just used them — so it was incomplete. But we'll fix that later.⁷ There's one more thing I want to mention about human communication before shifting our focus to problems in computer science that are mathematical — indeed

⁵ Adapted from a document titled *Mathematical Writing* by Donald E. Knuth, Tracy Larrabee, and Paul M. Roberts, based on a course given at Stanford University, here's an example of bad math writing:

Let P = the set of prime numbers, N = the set of natural (nonnegative) numbers, and Z^+ = the set of positive integers. If $L^+(P, N)$ is the set of functions $f : P \rightarrow N$ with the property that

$\exists n \in N \ \forall p \in P [p \geq n \Rightarrow f(p) = 0]$,

then there exists a bijection $Z^+ \rightarrow L^+(P, N)$ such that if $z \rightarrow f$ then

$$z = \prod_{p \in P} p^{f(p)}.$$

And here's Knuth, writing with **clarity** what the above is trying to say:

According to the Fundamental Theorem of Arithmetic, each positive integer n can be expressed in the form

$$n = 2^{e_2} 3^{e_3} 5^{e_5} 7^{e_7} 11^{e_{11}} \dots = \prod_{p \text{ prime}} p^{e_p},$$

where the exponents e_2, e_3, \dots are uniquely determined nonnegative integers, and where all but a finite number of the exponents are zero.

⁶ From the book by Rebecca Goldstein entitled *Incompleteness: The Proof and Paradox of Kurt Gödel*.

⁷ See §9.8.

discrete mathematical — in nature. That thing is *linearity* — or *sequentiality* — a bug, or maybe a feature, of language. This is the analogy of sentences as *beads on a string* used by Steven Pinker⁸ and others. That is, words must be written out or spoken and then read or heard in sequence, over time, one at a time. Consider what our communication would be like if instead we could apprehend words *all at once* — *in toto!*

(Let's just say that this led Abu and Ila to converse thoughtfully and at length, and ultimately to conclude that this way of communicating would be truer and much less error-prone than the way it is.)

9.4 Common Mold

"Computer scientists," said Til, "especially those into theoretical computer science, like to cast problems in the common mold of languages. They do this for technical reasons, more thoroughly delved into in a course on computational theory. But here is a simple, favorite example: Is 23 prime? This is a decision problem whose answer is yes, easily verified by simply trying to divide 23 by 2 and 3, and failing on both counts, of course.⁹ This decision could also be made by sequentially searching for and finding the string "23" in the set of strings {"2", "3", "5", "7", "11", "13", "17", "19", "23", ...}."

Til continued. "This set of strings is a language, and if you allow that the '...' stands for an infinity of bigger and bigger strings of this rather well-known kind, it is the language of PRIMES. It is given the name PRIMES, at any rate."

"So, does PRIMES contain the string "232323232323232323"? is another way to ask, is 2323232323232323 prime? The answer is no — it's a composite number with seven prime factors — including 23 — but the computational solution to that set membership determination problem is significantly harder than the one for 23. It's not done by simply searching in a static list. While many lists of primes exist, no one creates lists with every prime in it up to some huge limit. True, programs exist that can do that, using some variation of the classic Sieve of Eratosthenes,¹⁰ which goes way back, showing how old this problem is. But the point is, to solve a language membership problem you need computational strategies and tactics and resources. Simply put, we can model computation most generally in terms of machinery that can input a string, and output a 'yes' or a 'no' — 'in the language', or 'not'."

Ila said, "But not every problem has a yes-or-no answer!" to which Abu, agreeing, said "Like sorting, which, if I'm not mistaken, is a typical problem for computers, is it not?"

"Ah, my young friends," Til chuckled. "It so happens you are right, but computer scientists are clever people who've figured out a way to model a very large number of problems as decision problems, or as a series of de-

⁸ Found in Pinker's book, a companion to the one mentioned in the Preface, *The Stuff of Thought: Language as a Window into Human Nature*.

This analogy is also used (less successfully) to describe chromosomes as being made up of genes (the beads).

⁹ Why do we not need to also do trial division of 23 by 5, 7, 11, etc., to clinch its primeness?

¹⁰ As Abu and Ila recalled Til mentioning when they were studying number theory. See Appendix B.

```

unsorted = [13, 2, 26]
a = unsorted[0]
b = unsorted[1]
c = unsorted[2]
if a < b:
    if a < c:
        if b < c:
            print([a, b, c])
        else:
            print([a, c, b])
    else:
        print([c, a, b])
else:
    if b < c:
        if a < c:
            print([b, a, c])
        else:
            print([b, c, a])
    else:
        print([c, b, a])
# prints [2, 13, 26]

```

Figure 9.1: Sorting by comparing.

cision problems. Your very example of sorting, Abu, is one of the easiest.”

“How so?” said Abu, exchanging a puzzled look with Ila.

“Look at a simple example,” Til said. “Sorting [13, 2, 26] (or some other permutation) in ascending order is a matter of answering three (to five) yes-or-no questions: Is 13 less than 2? — no, so swap them. Is 2 less than 26? — yes, so don’t swap them. Is 13 less than 26? — yes, so leave them where they are as well. The result? [2, 13, 26] — see the code in Figure 9.1.”

Ila was still puzzled. “How does that relate to a set membership decision problem?”

Said Til, “Suppose we take the *language* of numbers

{"1", "2", "3", "4", "5", "6", ...}

and slice it into overlapping subsets:

less_than_2: {"1"}

less_than_3: {"1", "2"}

less_than_4: {"1", "2", "3"}

and so on, as many as we like. Then to answer the question, is a less than b, we just ask is a in the subset less_than_b?”

Ila frowned. “But isn’t that a way, way inefficient way to compare two numbers?” Til said, “Yes, it is, but if we’re not concerned with efficiency, that approach certainly works.”

“Now,” continued Til, “consider a big advantage of treating numbers as strings of digits. As you know, when the numbers get big we need special procedures if we want to do arithmetic with them. Let’s lump the relational operations with the arithmetic ones, and ask, how would one answer a simple a < b question, given:”

a = "361070123498760381765950923497698325576139879587987251757151"

b = "36107058266725245759262937693558834387849309867353286761847615132"

“That’s easy,” said Abu. “b is bigger because it has more digits.”¹¹ Said Ila, “And even if the strings were the same length, a digit-by-digit comparison would soon reveal the answer.”

Abu nodded. “Even if the strings were unsorted, as long as all those of the same length are grouped together we can do the comparisons, stopping as soon as any digit in a is different than the corresponding digit in b.”

The discussion of numbers as strings, comparing and sorting, and so on, went on a little while longer. We cut it off here in order to catch our breath before diving into the formalities.

¹¹ Unless the first five digits of b were zeros.

9.5 Language Language

In normal usage, a language is something we use to communicate, in speaking or writing. In theoretical computer science, a language is no more and no less than some subset of a set of all strings over some alphabet. Related formal definitions follow:

An **alphabet** is any non-empty, finite set (typically abbreviated Σ).

A **symbol** is a member or element of an alphabet.¹²

A **string** is a finite sequence of symbols from a given alphabet.

The **length** of a string is the number of symbols contained in the string. The length of w is denoted by $|w|$ in yet another overloading of the vertical bars. The **empty string** (abbreviated λ or ϵ) is a string with zero length.

The process of appending the symbols of one string to the end of another string, in the same order, is called **concatenation**.¹³

When the symbols in an alphabet all have length one, strings are commonly written as symbols placed side-by-side without the usual sequence trappings of braces or brackets, commas or spaces. Thus, we write abab rather than {a, b, a, b} or ['a', 'b', 'a', 'b'].

A method of listing strings called **numeric ordering** differs from so-called **dictionary** (or **lexicographical**) **ordering** in one essential way. Numeric ordering sorts strings *first* by increasing length (so shorter strings come before longer ones) and *then* by the predefined (dictionary or lexicographical) order of the symbols as given in association with the strings' alphabet.

For example, in numeric ordering the string baa would come before abab because it is shorter by one symbol. In plain old dictionary ordering lengths are ignored, so the string abab would come before baa, because in the alphabet, a (the first symbol of abab) comes before b (the first symbol of baa).¹⁴

To reiterate, a language is a subset of a set of strings. But which ones? That's where grammars come into play.

¹² Calling symbols *letters* would be too specific, as it would suggest that Σ is **the** alphabet we use in everyday writing.

¹³ In almost all programming languages strings are surrounded by single- or double-quotes, and these languages have at least one operation to perform string concatenation, which in Python is the (overloaded) + operator.

¹⁴ Why numeric ordering is important will become clear when the * (star) operation is discussed in §9.11.

9.6 Grammar Composition

A **Phrase-Structure Grammar** (PSG) is a four-tuple:

(N, T, S, P) where

- N is a set of **Nonterminals** (also called **Variables**)
- T is a set of **Terminals** ($N \cap T = \emptyset$)
- S is the **Start** Nonterminal ($S \in N$)
- P is a set of **Productions** (AKA **Rules**), each one mapping Nonterminals to sequences of Nonterminals and Terminals.

To start with something familiar, here are the components of a sample PSG's four-tuple for a (super small) subset of the English language:

```

N = {SENTENCE, NOUN_PHRASE, VERB_PHRASE,
      ARTICLE, ADJECTIVE, NOUN, VERB, ADVERB}

T = {a, the, hungry, sleepy, cat, dog, chases, runs, slowly, quickly}

S = SENTENCE
  
```

Quotes surrounding the words are missing on purpose, to reduce clutter. The terminals are just concrete words (lowercase by convention), and the Nonterminals are more abstract word *types* (uppercase by convention). NOUN, for example, is normally used to talk *about* and classify words, even though it can also be a terminal in everyday use. For the English language (as for any natural language) these terminal words form the set of symbols comprising the *alphabet* in the sense defined in §9.5.

The rules for this PSG are shown in Table 9.1.

Table 9.1: Rules for a simple PSG. The vertical bar is a “meta-symbol” meaning OR. For example, the NOUN rule produces either cat or dog (exclusive or). With the uppercase, lowercase naming conventions, and also stipulating that, in the absence of any statement to the contrary (so by default), the first symbol of the first production is the start symbol, it is possible to reconstruct the PSG four-tuple given just the set of productions.

$P = \{$	
SENTENCE	\rightarrow NOUN_PHRASE VERB_PHRASE NOUN_PHRASE
SENTENCE	\rightarrow NOUN_PHRASE VERB_PHRASE
NOUN_PHRASE	\rightarrow ARTICLE ADJECTIVE NOUN
NOUN_PHRASE	\rightarrow ARTICLE NOUN
VERB_PHRASE	\rightarrow VERB ADVERB
VERB_PHRASE	\rightarrow VERB
ARTICLE	\rightarrow a the
ADJECTIVE	\rightarrow hungry sleepy
NOUN	\rightarrow cat dog
VERB	\rightarrow chases runs
ADVERB	\rightarrow slowly quickly
}	

Read the right arrow as “produces” or “yields”. The verb **derive** and its noun counterpart **derivation** are also in play. We apply a single rule to *derive* one string R from another string L if the rule has L on the left and R on the right of the arrow. Iterate this process of producing a sequence of terminals from the Start Nonterminal by replacing Nonterminals one at a time by applying some rule — *derivation* is the name of this iterative process. See Figure 9.2.

Figure 9.2: Finding a random derivation.
For example, one call printed:

```
SENTENCE
NOUN_PHRASE VERB_PHRASE
ARTICLE ADJECTIVE NOUN VERB_PHRASE
the ADJECTIVE NOUN VERB_PHRASE
the sleepy NOUN VERB_PHRASE
the sleepy dog VERB_PHRASE
the sleepy dog VERB ADVERB
the sleepy dog runs ADVERB
the sleepy dog runs quickly
```

The derivation and final sentence printed will be different each time, most likely.

```
from random import choice

terminals = {'a', 'the', 'hungry', 'sleepy', 'cat', 'dog',
             'chases', 'runs', 'slowly', 'quickly'}

productions = {
    'SENTENCE': [[['NOUN_PHRASE', 'VERB_PHRASE', 'NOUN_PHRASE'],
                  ['NOUN_PHRASE', 'VERB_PHRASE']],
                 ['NOUN_PHRASE', 'VERB_PHRASE']],
    'NOUN_PHRASE': [[['ARTICLE', 'ADJECTIVE', 'NOUN'],
                     ['ARTICLE', 'NOUN']],
                    ['ADJECTIVE', ['hungry', 'sleepy']],
                    ['NOUN', ['cat', 'dog']],
                    ['VERB', ['chases', 'runs']],
                    ['ADVERB', ['slowly', 'quickly']]]

nonterminals = set(productions.keys())

G = {'nonterminals': nonterminals,
      'terminals': terminals,
      'start_symbol': 'SENTENCE',
      'productions': productions}

def nonterminals_remain(grammar, segment):
    nonterminals = grammar['nonterminals']
    return any(map(lambda s: s in nonterminals, segment))

def derive(grammar, LHS):
    RHS = grammar['productions'][LHS]
    if isinstance(RHS[0], list):
        return choice(RHS)
    else:
        return [choice(RHS)]

def find_derivation(grammar):
    derivation = [grammar['start_symbol']]
    terminals = grammar['terminals']
    while nonterminals_remain(grammar, derivation):
        print(' '.join(derivation))
        derive_one = []
        for i in range(len(derivation)):
            symbol = derivation[i]
            if symbol in terminals:
                derive_one.append(symbol)
            else:
                derive_one.extend(derive(grammar, symbol))
                derive_one.extend(derivation[i+1:])
                break
        derivation = derive_one
    return ' '.join(derivation)

print(find_derivation(G))
```

Exercise 367 Run the code in Figure 9.2 several times. How many times does it take you to derive five different sentences?

Hint 367 The code can be found [here](#).

AT a fSA 862 7ewsnA

Exercise 368 Combinatorially speaking, how many different sentences can eventually be derived by repeated calls to `find_derivation(G)`?

Hint 368 Count how many choices there are for each Nonterminal and for each Terminal. Use the sum rule and the product rule.

AT a fSA 862 7ewsnA

Exercise 369 With this simple grammar is there a derivation for the following sentence? the hungry sleepy dog runs

Hint 369 Look at the ADJECTIVE rule closely.

AT a fSA 862 7ewsnA

α | AVITELCA AVITELCA → AVITELCA
α: αβιτελκα βιτελκα γιτελκα διτελκα ειτελκα ζιτελκα
αβιτελκας' διτελκας' γιτελκας' ειτελκας' ζιτελκας'
αβιτελκασ' διτελκασ' γιτελκασ' ειτελκασ' ζιτελκασ'
αβιτελκασσ' διτελκασσ' γιτελκασσ' ειτελκασσ' ζιτελκασσ'
αβιτελκασσα' διτελκασσα' γιτελκασσα' ειτελκασσα' ζιτελκασσα'
αβιτελκασσασ' διτελκασσασ' γιτελκασσασ' ειτελκασσασ' ζιτελκασσασ'
αβιτελκασσασα' διτελκασσασα' γιτελκασσασα' ειτελκασσασα' ζιτελκασσασα'
αβιτελκασσασασ' διτελκασσασασ' γιτελκασσασασ' ειτελκασσασασ' ζιτελκασσασασ'
αβιτελκασσασασα' διτελκασσασασα' γιτελκασσασασα' ειτελκασσασασα' ζιτελκασσασασα'
αβιτελκασσασασασ' διτελκασσασασασ' γιτελκασσασασασ' ειτελκασσασασασ' ζιτελκασσασασασ'

Exercise 370 Moving towards a more sophisticated grammar, what rules would need to be changed or added to generate this sentence?

the quick brown fox jumps over the lazy dog

Hint 370 What type of word is "over"? You will need to add three rules, and change the ADJECTIVE rule like you had to do for Exercise 369.

AT a fSA 862 7ewsnA

9.7 Parsing Up a Tree

The derivation of a syntactically valid structured phrase from the top down can be visualized as the reverse of the process of building, from the bottom up, i.e., from leaves to root, a **syntax tree** (AKA a **parse tree**).

For example, a valid sentence forms the leaves of a parse tree. Each terminal is given a parent, and as an alternate representation, each parent-child is rendered in tree form as a two-element list. Continue on up the tree, building a nested list in the process. Finish with SENTENCE as the root of the tree. See Figures 9.3, 9.4, 9.5, and 9.6.

Exercise 371 Courtesy Peter Norvig, *there is a better way to implement a PSG — code also found here.*

Generate three sentences and three trees with this grammar.

Now let S be short for SENTENCE, NP for NOUN_PHRASE, VP for VERB_PHRASE, ART for ARTICLE, ADJ for ADJECTIVE, N for NOUN, V for VERB, and ADV for ADVERB.

Use the code

grammar = Grammar(S = 'NP VP NP | NP VP',)

as a starting point. Extend it to match the grammar found in §9.6.

Generate three more sentences and three more trees with this grammar.

Hint 371 This is straightforward, but you should make sure you understand how Norvig's code works.

ANSWER 371

The generated sentences and trees will be different for

Exercise 372 To help with the syntax-tree building process, please use [this tool](#), whose so-called labelled bracket notation is just the tree-as-nested-list representation, only without the commas.

Build a parse tree for

the quick brown fox jumps over the lazy dog.

Include a picture and the list representation as your answer.

Here is an example of a picture of a parse tree created by this tool.

Hint 372 Use the extended PSG you created in Exercise 370.

ANSWER 372



Figure 9.3: Leaves of the parse tree for a simple sentence.

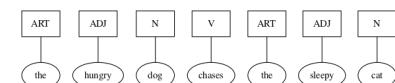


Figure 9.4: Leaves and parents of the parse tree for a simple sentence. Abbreviating ADJECTIVE as ADJ, ADVERB as ADV, ARTICLE as ART, NOUN as N, and VERB as V, the list representation is [[ART, the], [ADJ, hungry], [N, dog], [V, chases], [ART, the], [ADJ, sleepy], [N, cat]].

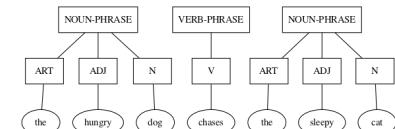


Figure 9.5: Almost complete parse tree. Abbreviating NOUN_PHRASE as NP, and VERB_PHRASE as VP, the list representation is [[NP, [ART, the], [ADJ, hungry], [N, dog]], [VP, [V, chases]], [NP, [ART, the], [ADJ, sleepy], [N, cat]]].

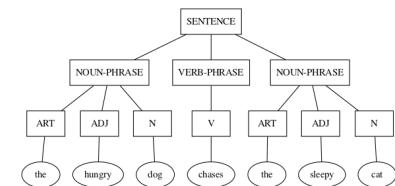


Figure 9.6: Complete parse tree. Abbreviating SENTENCE as S, the list representation is [S, [[NP, [ART, the], [ADJ, hungry], [N, dog]], [VP, [V, chases]], [NP, [ART, the], [ADJ, sleepy], [N, cat]]]].

Exercise 373 Build parse trees for the following sentences:

1. a cat chases a hungry dog.
2. the dog runs quickly.
3. the sleepy dog chases quickly a hungry cat.

Hint 373 Use the same approach you did for Exercise 372.

[N cat][]
[NP [N the] [VP [V chases] [NP [N a] [NP [N hungry] [NP [N dog] [VP [V runs] [ADJ [A quick]]]]]]]]
3. [S [NP [N the] [NP [N dog] [VP [V runs] [ADJ [A quickly]]]]] [VP [V chases] [NP [N a] [NP [N hungry] [NP [N cat] [VP [V sleeps] [ADJ [A sleepy]]]]]]]]]
S. [S [NP [N the] [NP [N dog] [VP [V runs] [ADJ [A quickly]]]]] [VP [V chases] [NP [N a] [NP [N hungry] [NP [N cat] [VP [V sleeps] [ADJ [A sleepy]]]]]]]]]
A 373 ANSWER

9.8 Moving Closer to Useful

Next up for examination is a simple grammar moving in the direction of programming languages. This grammar produces well-formed parenthesized expressions, which could be extended to derive well-formed arithmetical or logical expressions. The idea is to use a *skeleton* for matching opening and closing parentheses. The five productions of this grammar are shown in Figure 9.7. A sample derivation is shown in Table 9.2.

Exercise 374 Implement the PSG you get by combining the four rules for building propositions seen in §2.5 with the five rules in the grammar for matching parentheses. You may end up with more than nine rules.

Hint 374 Use the Norgig code and generate lots of different logical propositions that use parentheses for grouping.

AT A SAA FIZI TAWSA

1. SKEL → OP SKEL CP
2. SKEL → SKEL SKEL
3. SKEL → λ
4. OP → (
5. CP →)

Figure 9.7: Matching parentheses. Rule 1 is a recursive rule for enclosing in parentheses. Rule 2 is for expanding the length of the derivation. Rule 3 is for terminating the recursion. Rules 4 and 5 produce the terminal opening and closing parentheses.

Start	Rule	Next Derivation Step
SKEL	→ 2	SKEL SKEL
	→ 1	OP SKEL CP SKEL
	→ 1	OP OP SKEL CP CP SKEL
	→ 3	OP OP λ CP CP SKEL
	→ 3	OP OP λ CP CP λ
	→ 4	(OP λ CP CP λ
	→ 4	((λ CP CP λ
	→ 5	((λ) CP λ
	→ 5	((λ)) λ
	→	(())

Table 9.2: A sample derivation for a grammar to match parentheses. The last step just removes the invisible λ strings.

9.9 Irregular

¹⁵ *English Is Not a Context-Free Language*,
James Higginbotham, Linguistic Inquiry
Vol. 15, No. 2 (Spring, 1984), pages
225-234.

The grammar for the English language was long thought to be context free, but a clever argument shows that it is not.¹⁵ The simple subset-of-English grammar we have been exploring is certainly not constrained by context. To clarify, a grammar is **context free** if every production has exactly one Nonterminal to the left of its arrow. All of the grammars we have seen so far have been context free. By way of contrast, here's an example of two productions in a NON-context-free grammar:

$$cAb \rightarrow ccpb$$

$$cAt \rightarrow cpct$$

Note that A expands differently when it's surrounded by c and b than when it's surrounded by c and t. That means the expansion of A has context "sensitivity". A grammar/language with this feature is called **context sensitive**. Another example is the grammar that generates the language consisting of any string followed by *an exact copy* of that string, e.g., bitstrings 00, 101101, 010010001010010001, etc.

Moving up to an even more sophisticated type, a **recursively enumerable**¹⁶ grammar/language has the most computational power, the language of PRIMES being just one example.

¹⁶ See §9.16.

9.10 Regular

Moving down to the simplest type, a language is **regular** if it can be generated from its alphabet using the three **regular operations**, union, concatenation, and star. How these work can be crudely illustrated using a type of graph:¹⁷

1. \cup (**union** — Figure 9.8)
2. \circ (**concatenation** — Figure 9.9)
3. $*$ (**star** — Figure 9.10)

For now we'll forgo a complete description of the procedures and rules for how these regular operations combine and compose into one graph, as they can get somewhat complicated and are best left for a more advanced setting. But for the record, a *very important rule* for these graph compositions is:

Every node must have one outgoing link for each symbol in the alphabet.

Regular operations are used to build **regular expressions** which in turn describe **regular languages**.¹⁸ Figure 9.11 shows a recursive definition.

¹⁷ More precisely, a *pseudograph*, as loops are allowed. This is a *directed* graph with labels on the links (and the nodes, as we'll see).

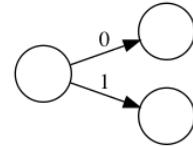


Figure 9.8: $0 \cup 1$. A node with a link to another node for each *disjunct* (0 or 1) — so either path may be taken from the left-most node.



Figure 9.9: $0 \circ 1$. A starting node and two other nodes (one for each symbol) and a link for each *conjunct* (0 and 1) in sequence.



Figure 9.10: 0^* . A node with a loop-link labeled with the symbol preceding the star.

¹⁸ A language is regular *iff* some regular expression describes it.

R is a **regular expression** (an **RE** for short) if R is any of:

- \emptyset
- $\{\lambda\}$
- $\{a\}$ for some $a \in \Sigma$
- $R_1 \cup R_2$, where R_1 and R_2 are RE's
- $R_1 \circ R_2$, where R_1 and R_2 are RE's
- R^* , where R is an RE

Some shorthand and other equivalences:

- $a \equiv \{a\}$
- $\lambda \equiv \{\lambda\}$
- $R^+ \equiv R \circ R^*$
- $R^* \equiv R^+ \cup \{\lambda\}$
- $R^k \equiv R \circ R \circ R \circ \dots \circ R$ (k times)

Figure 9.11: Regular expressions defined. Note that \circ is analogous to multiplication, so $R \circ R$ is usually written RR , with the \circ operator only implied.

9.11 Star Power

¹⁹ Named after Stephen Cole Kleene, the mathematician who defined this powerful operation to work on either a set of symbols or a set of strings. The article referenced defines V^* “as the set of finite-length strings that can be generated by concatenating arbitrary elements of V , allowing the use of the same element multiple times.”

²⁰ For illustration’s sake, and analogous to the set of all bitstrings, take the alphabet $\{‘a’, ‘b’\}$, where the Kleene star produces the following set: $\{“, ‘a’, ‘b’, ‘aa’, ‘ab’, ‘ba’, ‘bb’, ‘aaa’, ‘aab’, ‘aba’, ‘abb’, ‘baa’, ‘bab’, ‘bba’, ‘bbb’, ‘aaaa’, …\}$.

Note that if we used standard dictionary ordering to list this set, it would start like $\{“, ‘a’, ‘aa’, ‘aaa’, ‘aaaa’, …\}$ and never get to ‘b’, etc. That’s the reason for preferring numeric ordering that considers length first.

A complicating feature of these operators is that they can be applied to more than just one symbol, e.g., $(0 \cup 1)^*$, the star (AKA **Kleene star**¹⁹) operator yielding any number of repetitions (including zero) of a 0 or a 1, mixed and matched arbitrarily, which eventually yields all possible strings over the alphabet $\{0, 1\}$ (AKA bitstrings).²⁰

Mitigating some of the complexity, nodes can be split or merged (shared) and thus yield a simpler graph modeling the same language. For example, the graph in Figure 9.12 models the language of all bitstrings that end in 1; equivalently, the regular language $(0 \cup 1)^*1$.

9.12 State Machinery

Regular grammars generate regular languages, and thus are amenable to this kind of graph modeling. In this representation of grammar as graph, nodes correspond to the Nonterminals, and links to Terminals, similar to how nodes and links worked in the examples seen so far. But now let’s shift our focus and change the way we traverse these graphs. Instead of tracing paths to see what strings can be generated, let’s present some string to the graph and ask *it* to try to thread the string through itself.

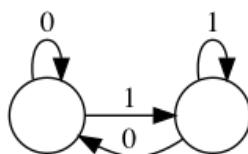


Figure 9.12: Bitstrings ending in 1. The node corresponding to the $*$ in the regular expression $(0 \cup 1)^*1$ has been split in two, one loop from that split node labeled 0 and the other labeled 1, while the link for the 0 in the \cup is shared with the loop-link for the $*$. Tracing different paths starting from the leftmost node and ending at the rightmost node, traversing links more than once being allowed, it is possible to generate the entire language — every bitstring that ends in 1. The method is to concatenate onto a growing string (initially empty) a 0 as a link labeled 0 is traversed, and a 1 as a link labeled 1 is traversed. So, labeling the two nodes L and R, the path L-R yields 1, the path L-R-L-R yields 101, the path L-R-L-L-R-R yields 10011, and so forth.

This recasts the graph as an actor — an agent making decisions. In this guise it is a machine — a **state machine**.²¹ The static picture of the graph is a **state diagram**,²² the nodes are **states**, the links are **transitions** between states. Let's revisit string processing from this new point of view, this time giving nodes labels as well as links.

By convention, the **start state** (node)²³ — corresponding to the grammar's Start symbol — is the node named S, as in Figure 9.14, or with some other symbol, sometimes followed by one or more digits, like subscripts.

When the machine is “turned on” it comes up “in” its start state. It then starts reading symbols, one by one, from its “input” — the string being presented for processing — and it “moves” to other states according as the current symbol directs it to make one transition or another (the one labeled with that symbol). It does three things when making a transition:

1. it updates its current state to be the one at the other end of the transition link;
2. it “consumes” its currently read symbol; and
3. it then turns its attention to the next symbol in the input, making it the current symbol.

Figure 9.15 shows a sample state transition on a 0.

Figure 9.16 shows a sample state transition on both 0 and 1 inputs. This is shorthand notation for two transitions, one for 0, and the other for 1.

Figure 9.17 shows an **accept state**.²⁴

²¹ Or **finite-state machine** (or **finite-state automaton** (plural **automata**)). (We will avoid **infinite-state automata** like the plague!)

²² Or **state-transition diagram**.

²³ More conventionally, the start state is identified by an incoming arrow pointing to it — but coming from nowhere — see Figure 9.13. Or, in another form of annotation, a triangle pointing to it is sometimes used to mark the start state.

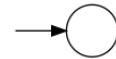


Figure 9.13: Start state with incoming arrow.



Figure 9.14: Start state with label only.

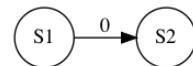


Figure 9.15: Single state-to-state transition on a 0.

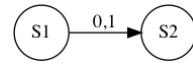


Figure 9.16: Double state-to-state transition on both 0 and 1.

²⁴ AKA a **final state**. There can be more than one of these in a state diagram.



Figure 9.17: Accept state S4.

9.13 Recognition

Accept states serve as language membership detectors. If a candidate input string is exhausted — entirely consumed by the transitions from state to state — at the exact transition an accept state is reached, the string is accepted as part of the language. A string exhausted in a non-accepting state (one without a double circle) is rejected — it is *not* part of the language. If a machine accepts *all* strings that belong to a specific language, and rejects *all* those that *do not* belong to the language, then the machine is said to **recognize** the language. This makes the machine a **language recognizer**.

For example, the machine in Figure 9.18 recognizes the tiny language whose set of a mere three strings is {01, 011, 0111}.

More correctly — following the *very important rule*²⁵ — there should be transitions for each input symbol (0 and 1) out of each state, as shown in Figure 9.19. This six-state machine realizes the five-rule PSG shown in Figure 9.20.

Recursive rules create loops, just like the star operation. For example, the rule $A \rightarrow 0A$ yields the state+transition shown in Figure 9.21.

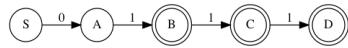


Figure 9.18: Three string machine.

²⁵ This rule must be adhered to for the machine to be a valid **deterministic** finite automaton (**DFA**). The rule can be relaxed when dealing with the related **nondeterministic** finite automata (**NFA**).

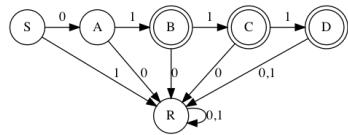


Figure 9.19: Complete three string machine.

$$\begin{array}{lcl}
 S & \rightarrow & 0A \\
 A & \rightarrow & 1B \\
 B & \rightarrow & 1C \mid \lambda \\
 C & \rightarrow & 1D \mid \lambda \\
 D & \rightarrow & \lambda
 \end{array}$$

Figure 9.20: PSG realized by the machine in Figure 9.19. Only the rules are needed, the other components are inferable.



Figure 9.21: $A \rightarrow 0A$. At node A, leave on a 0 and go back to A, as if the 0 in the rule were pulled to the left to label the arrow, and the A on the right were moved over and merged with the A on the left, in the process bending the arrow around into a loop.

For another example, the grammar in Figure 9.22 is represented by the machine in Figure 9.23.

Exercise 375 Add a state and the necessary transitions to make the machine in Figure 9.23 a valid state machine.

Hint 375 Remember the very important rule?

ANSWER 312 **A**pply a state **B** (for **Beliefs**). Transitions from state **C** to a state **D** to itself on both a **θ** and a **T**. Once in **B**, stay there. In other words, beliefs remain stable over time.

Exercise 376 Compare and contrast the $1(0 \cup 1)^*$ machine with the $(0 \cup 1)^*1$ machine.

Hint 376 Make at least two points of comparison (how they are the same) and two points of contrast (how they are different).

ATLAS THE ANSWER

Exercise 377 Figure 9.24 shows a machine that recognizes the language of all bitstrings whose second-to-last bit is a 0. Using this as a pattern, create (and draw) a recognizer of bitstrings whose third-to-last bit is a 0.

Hint 377 This machine needs eight states — four accept states plus four reject (non-accept) states.

$$\begin{array}{rcl} S & \rightarrow & 1A \\ A & \rightarrow & 0A \mid 1A \mid \lambda \end{array}$$

Figure 9.22: PSG realized by the machine in Figure 9.23.

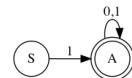


Figure 9.23: Recognizer for the language of bitstrings that *start* with $1 \cdot 1(0 \cup 1)^*$ is its equivalent regular expression.

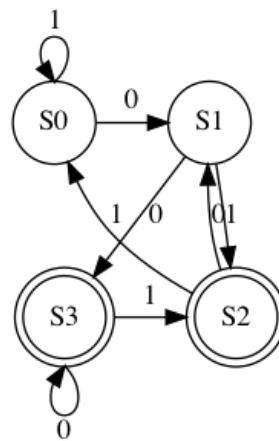


Figure 9.24: Recognizer for the language of bitstrings whose second-to-last bit is a 0.

Exercise 378 This three-production grammar

$$\begin{array}{lcl} S & \rightarrow & A1 \\ A & \rightarrow & A\emptyset \\ A & \rightarrow & \lambda \end{array}$$

generates the simple language consisting of any number of \emptyset s (including zero \emptyset s) followed by a single 1. Build a state machine to model it.



Hint 378 Your machine should have exactly three states.

AT A GLANCE

Figure 9.25: Single state with transitions on \emptyset and 1.

Exercise 379 The machine in Figure 9.25 recognizes which language?

Hint 379 It's a very small language.

еневъл струнък бърсемтедж то ѝ.
реесиусе то маа то донорът сирце то твдигате ассеебатанс' маафес то твдигате
енаве оѣ али. Твдигате реесиусе твдигате маафуме маа то инил а старт стате' твдигате'
AT A GLANCE In fact, it's the empty language. \emptyset , the smallest language.

9.14 Tiny Computer

The language recognizers we have been examining are not the only interpretation we can give these kinds of state machines. You may have seen others in a course on digital logic and circuits. Figure 9.26 shows one of the simplest possible examples.

This is a tiny, 1-bit computer controlling some lights in a room equipped with motion sensors. The lights are either off or on. The state of the lights can thus be remembered with just 1 bit of memory — 0 for off and 1 for on. The lights are controlled — toggled on and off — based on motion (or lack thereof) detected by the motion sensors, which are also connected to a timer.

The lights are initially off, so the computer starts in the OFF state. In this state, only the MOTION input causes it to move to the ON state, which causes the lights to go on. In the ON state, a MOTION input causes it to remain in the ON state (the lights stay on), and also resets the timer. After a configured time duration passes with no intervening MOTION inputs, the timer elapses, triggering the TIMER-ELAPSED input. This input causes it to move to the OFF state, which turns the lights off.

Exercise 380 Find another example of a small computer, one with fewer than 10 states.



Figure 9.26: One-bit computer.

Hint 380 There are approximately five billion out there to find.

AT A GLANCE

9.15 Back and Forth

There are two related questions that we have the preliminary knowledge to try to answer:

1. What language is generated by a given grammar?
2. What grammar generates a given language?

Exercise 381 In this and the following four exercises, $N = \{S, A, B\}$, $T = \{0, 1\}$, and $S = S$. Find the language generated by the (N, T, S, P) grammar when the set P of productions is

$$\begin{array}{lcl} S & \rightarrow & AB \\ A & \rightarrow & 01 \\ B & \rightarrow & 11 \end{array}$$

Hint 381 If just staring at it fails to reveal the answer, use the code found here that you've already seen to implement this grammar and see how many strings it will generate.

0JJ

एति॒ रा॒मानु॒जे॒ शेवरात्रेऽ॒ य॒ त्य॒ ग॒ ग्रंथ॒म्। स॒क्षिप्त॒ लो॒ ओ॒ ए॒ श्वर॒ः॥
त्वं॒ रा॒मानु॒जे॒ शेवरात्रेऽ॒ य॒ त्य॒ ग॒ ग्रंथ॒म्। स॒क्षिप्त॒ लो॒ ओ॒ ए॒ श्वर॒ः॥

Exercise 382 Find the language generated by

$$\begin{array}{lcl} S & \rightarrow & AB \\ S & \rightarrow & AA \\ A & \rightarrow & \theta B \\ A & \rightarrow & \theta 1 \\ B & \rightarrow & 1 \end{array}$$

Hint 382 Same hint as for Exercise 381.

AT ा फ्सA ग्ग॒ रामानु॒जे॒

Exercise 383 Find the language generated by

$$\begin{array}{lcl} S & \rightarrow & AA \\ S & \rightarrow & B \\ A & \rightarrow & \theta\theta A \\ A & \rightarrow & \theta\theta \\ B & \rightarrow & 1B \\ B & \rightarrow & 1 \end{array}$$

Hint 383 Same hint as for Exercise 381.

एति॒ रा॒मानु॒जे॒ शेवरात्रेऽ॒ य॒ त्य॒ ग॒ ग्रंथ॒म्: {θ_n | n ≥ 2} ∪ {J_n | n ≥ 1}
त्वं॒ रा॒मानु॒जे॒ शेवरात्रेऽ॒ य॒ त्य॒ ग॒ ग्रंथ॒म्: त्वं॒ रा॒मानु॒जे॒ शेवरात्रेऽ॒ य॒ त्य॒ ग॒ ग्रंथ॒म्: {θ_n | n ≥ 2} ∪ {J_n | n ≥ 1}

Exercise 384 Find the language generated by

$$\begin{array}{lcl} S & \rightarrow & AB \\ A & \rightarrow & \theta A 1 \\ B & \rightarrow & 1 B \theta \\ A & \rightarrow & \lambda \\ B & \rightarrow & \lambda \end{array}$$

Hint 384 Same hint as for Exercise 381.

AT ڈسا 482 تے وسٹا

Exercise 385 Construct a PSG to generate the language

$$\{0^{2n} \mid n \geq 0\}.$$

but it might be easier to try it with three productions.
being a recursive production, and the other being a *l* production,
Hint 385 You can do this with two productions (with one of them

प्रतिक्रिया समान है: {S → 00, S → 000, S → 0000...}
यह एक विस्तृत उत्पादन है: A → a, A → AA, A → A00
यह एक विस्तृत उत्पादन है: A → a, A → AA, A → A00

Exercise 386 Construct a PSG to generate $\{0^n 1^{2n} \mid n \geq 0\}$.

Hint 386 For this one, only two productions are needed.

AT ڈسا 482 تے وسٹا

Exercise 387 Construct a PSG to generate $\{0^n 1^m 0^n \mid m \geq 0, n \geq 0\}$.

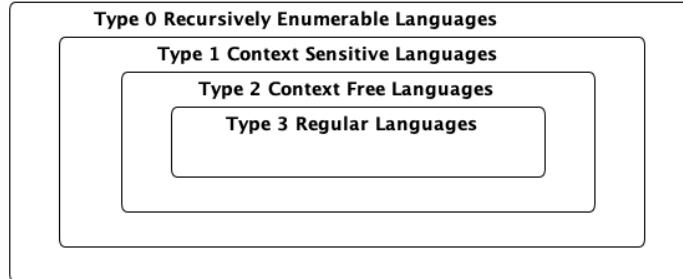
Hint 387 Only four productions are needed for this one.

एक विस्तृत उत्पादन है: A → a, A → AA, A → A00, A → 0A
एक विस्तृत उत्पादन है: A → a, A → AA, A → A00, A → 0A

9.16 Taxonomy

Noam Chomsky is a linguist who first proposed the hierarchical language classification scheme that now bears his name. In Figure 9.27, the nested sets are embedded in a *Universal Set of All Languages* (the superset of Types 0-3).

Figure 9.27: Chomsky hierarchy of languages.



Turning this figure into a table, Table 9.3 maps the languages classes (types) with the types of grammars that can generate those languages. The restrictions on what types of productions are allowed distinguish what's what.

Exercise 388 Find an example of each of the four types of grammars.

Hint 388 Look at the examples in this chapter, or online for lots of others.

AT A GLANCE 388 ANSWERS

Table 9.3: A taxonomy of language and grammar types. N = Nonterminal, tl = terminal, LHS = Left-Hand-Side, and RHS = Right-Hand-Side.

Language class	Type	Restrictions on grammar productions
Recursively Enumerable	0	No restrictions (length of LHS may exceed length of RHS).
Context Sensitive	1	LHS may have more than one Nonterminal, but the length of the LHS must be at most the length of the RHS (except for $S \rightarrow \lambda$ productions).
Context Free	2	Each LHS must have only one Nonterminal.
Regular	3	Left-linear or Right-linear (each RHS must be either a tl or λ , or else have a single Nonterminal and be all like Ntl (Left-linear) or be all like tlN (Right-linear)).

9.17 Conclusion

From your current perspective it may seem that formal languages and grammars are fairly different from most other topics in discrete mathematics, such as number theory.

However, as we've seen, there are two topics that team with connections. Trees show how grammars generate languages, and we see this by building parse trees top-down or bottom-up. Graphs show how to model languages, with labeled, directed graphs describing the finite-state machines that recognize languages.

When all is said and done, if you want reassurance that learning more about discrete math, and especially languages, grammars, and other models of computation, has practical value for your career, consider this summary by the second author, to whom we give the last word:

[...] despite the explosion of input and output devices, computer science continues to rely on text scanning and parsing to support the vast number of programming languages, scripting languages and data handling languages. Text still rules supreme, and correct plus efficient text handling for very large alphabets continues to be cutting-edge CS research that also immediately ties into practice.²⁶

²⁶ In *Automata and Computability: A Programmer's Perspective*, a quote from which also appeared in the Preface.

9.18 Summary of Terms and Definitions

The term **jargon** refers to any specialized, not-well-publicized language that empowers insiders with more efficient communication, at the same time excluding outsiders who don't know the lingo.

An **alphabet** is any non-empty, finite set (typically abbreviated Σ).

A **symbol** is a member or element of an alphabet.

A **string** is a finite sequence of symbols from a given alphabet.

The **length** of a string w , denoted $|w|$, is the number of symbols in the string.

The **empty string** (abbreviated λ or ϵ) is a string with zero length.

The **concatenation** of any strings a and b is the process of joining the symbols of string b to the end of string a , in that order.

A **numeric ordering** is a listing of strings in order by length first, then by their **dictionary ordering**.

A **dictionary ordering** lists strings in an order according to their symbols' natural or predefined order.

Lexicographical ordering is a synonym for **dictionary ordering**.

A **Phrase-Structure Grammar** (PSG) is a four-tuple:

(N, T, S, P) where

- N is a set of **Nonterminals** (also called **Variables**)
- T is a set of **Terminals** ($N \cap T = \emptyset$)
- S is the **Start Nonterminal** ($S \in N$)
- P is a set of **Productions (AKA Rules)**, each one mapping Nonterminals to sequences of Nonterminals and Terminals.

To **derive** string a from string b means to replace b (appearing on the left-hand-side of a rule) with a (appearing on the right-hand-side of a rule.)

A **derivation** is the result of the iterative process of deriving one string from another until what's left is a sequence of terminals.

A **syntax tree** has a grammar's start symbol as root and the grammar's terminals as leaves. It captures the derivation of a syntactically valid structured phrase from the top down. It can also be built from the bottom up in a reverse derivation.

A synonym for syntax tree is **parse tree**.

The **context** of a grammar's nonterminals are the terminals that surround it in the left-hand-side of a rule.

A type of grammar/language that is not sensitive to (does not depend on) context is called **context free**.

A type of grammar/language that *is* sensitive to context is called **context sensitive**.

A **recursively enumerable** grammar/language is the most general, least restrictive type.

A **regular** grammar/language is the least general, most restrictive, and simplest type. It can be generated from its alphabet using the **regular operations**.

The **regular operations** are:

- \cup (**union**) — same as set union.
- \circ (**concatenation**) — string concatenation.
- $*$ (**star AKA Kleene star**) — all strings of all lengths and combinations from an alphabet.

R is a **regular expression** (an **re** for short) if R is any of:

- \emptyset
- $\{\lambda\}$
- $\{a\}$ for some $a \in \Sigma$
- $R_1 \cup R_2$, where R_1 and R_2 are **re**'s
- $R_1 \circ R_2$, where R_1 and R_2 are **re**'s
- R^* , where R is an **re**

Shorthand and other equivalences:

- $a \equiv \{a\}$
- $\lambda \equiv \{\lambda\}$
- $R^+ \equiv R \circ R^*$
- $R^* \equiv R^+ \cup \{\lambda\}$
- $R^k \equiv R \circ R \circ \dots \circ R$ (k times)

The \circ operation is analogous to multiplication, thus $R \circ R$ is usually written RR , without the \circ .

The **Kleene star** operation produces the set of finite-length strings that can be generated by concatenating arbitrary elements of an alphabet, allowing the use of the same element multiple times.

A **regular language** is one that can be described by a regular expression, in fact, a language is regular *iff* some regular expression describes it.

A **state machine** models the generation or recognition of a language.

A **finite-state machine** has a finite number of states.

A **finite-state automaton** is another term for a finite-state machine.

The plural of finite-state automaton is **finite-state automata**

A **state diagram** is a static picture of a graph representing the state machine.

A **state transition diagram** is a more accurate name for a state diagram.

A **state** is a node in the graph visualized by the state diagram.

A **transition** is a link in the graph visualized by the state diagram.

A **start state** is a state the state machine starts in when it is activated. It is marked in a state diagram by a special label and/or an input arrow that comes from nowhere.

An **accept state** is a possible state the state machine ends in when its input is fully consumed. If so, the machine **accepts** the input string. It is marked in a state diagram by a double circle.

A **final state** is a synonym for an accept state.

A **reject state** is a possible state the state machine ends in when its input is fully consumed. If so, the machine **rejects** the input string.

A machine is said to **recognize** a language if it accepts all strings that belong to the language and rejects all strings that do not belong to the language.

A **language recognizer** is a machine that recognizes a languages.

The *very important rule* is that *every node (state) must have one outgoing link (transition) for each symbol in the alphabet.*

A **deterministic** type of automaton adheres to the very important rule, i.e., it defines a transition from each state for each symbol in the alphabet.

A **DFA** is a deterministic finite-state automaton.

A **nondeterministic** type of automaton allows the very important rule to be ignored, i.e., it need not define a transition from each state for each symbol in the alphabet, and indeed, for a given symbol it can have *multiple* transitions from a state to other states. Computational Theory (AKA Automata and Computability) studies these important machine types in depth.

An **NFA** is a nondeterministic finite-state automaton.

A **Type 0** language/grammar is recursively enumerable.

A **Type 1** language/grammar is context sensitive.

A **Type 2** language/grammar is context free.

A **Type 3** language/grammar is regular.

Appendices

A

Proofs of Some Theorems of Number Theory

Theorem A.1 (Chapter 6) (*Euclid*): *There are infinitely many primes.*¹

*Another way to say there are infinitely many primes is to say there is no largest prime. Suppose not. In other words, suppose, in contradiction to what we seek to establish, that there is a largest prime. Call it x.*²

1. *Form the product of all primes less than or equal to x, and then add 1 to the product.*
2. *This yields a new number y. For example, $y = (2 \cdot 3 \cdot 5 \cdot 7 \cdot \dots \cdot x) + 1$.*
3. *If y is itself a prime, then x is not the largest prime, for y is greater than x — by virtue of how it's constructed.*
4. *If y is composite (i.e., not a prime), then again x is not the largest prime, because being composite means y must have a prime divisor z, and z must be different³ from each of the primes 2, 3, 5, 7, ..., up to x. Hence z must be a prime greater than x.*
5. *But y is either prime or composite, there's no third option.*
6. *Hence x is not the largest prime.*
7. *Hence there is no largest prime.*

To back up, and point out the all-too-human tendency to leap to conclusions with inductive assertions, when is y ever composite?

- $(2) + 1 = 3$
- $(2 \cdot 3) + 1 = 7$
- $(2 \cdot 3 \cdot 5) + 1 = 31$
- $(2 \cdot 3 \cdot 5 \cdot 7) + 1 = 211$
- $(2 \cdot 3 \cdot 5 \cdot 7 \cdot 11) + 1 = 2311$

All of these, 3, 7, 31, 211, and 2311, are prime, so if you stop there and leap to conclusions, you'll never see it. However, if you look at the very next one in the sequence, $(2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13) + 1 = 30031$, which is 59 (prime) times 509 (also prime).

The number y — made by adding 1 to the product of all primes in a *supposedly complete* list of primes — always gives rise, one way or an-

¹ This proof is loosely adapted from an exposition in the book *Gödel's Proof*, by Ernest Nagel and James R. Newman, and expanded upon.

² How does supposing the existence of such a thing help? It helps because we can show that it is *logically impossible* for this x , this so-called largest prime to exist.

³ Do you see why?

other, to a prime not in the list.

It seems backwards to prove some negative (no largest prime) this way. And so it is. The technique is called *proof by contradiction* or *reducing to the absurd*, to translate the Latin version *reductio ad absurdum*, by which it has been known for centuries. A venerable proof technique well worth adding to your arsenal.

Theorem A.2 (Chapter 6) ($\text{GCD}(a,b) \cdot \text{LCM}(a,b) = ab$): *The equality $\text{gcd}(a,b) \cdot \text{lcm}(a,b) = ab$ holds for all pairs of positive integers:*

Proof:

Let $a = 2^{p_2} \cdot 3^{p_3} \cdot 5^{p_5} \cdot 7^{p_7} \dots$, and let $b = 2^{q_2} \cdot 3^{q_3} \cdot 5^{q_5} \cdot 7^{q_7} \dots$

Then $\text{gcd}(a,b) = 2^{\min(p_2, q_2)} \cdot 3^{\min(p_3, q_3)} \cdot 5^{\min(p_5, q_5)} \cdot 7^{\min(p_7, q_7)} \dots$, and $\text{lcm}(a,b) = 2^{\max(p_2, q_2)} \cdot 3^{\max(p_3, q_3)} \cdot 5^{\max(p_5, q_5)} \cdot 7^{\max(p_7, q_7)} \dots$, and by the laws of exponents, $ab = 2^{r_2} \cdot 3^{r_3} \cdot 5^{r_5} \cdot 7^{r_7} \dots$, where $r_i = p_i + q_i$.

This shows the product of a and b equals the product of their gcd and their lcm, because

$\min(p_i, q_i) + \max(p_i, q_i) = p_i + q_i$. Of two numbers, their min is one of them, and their max is the other one. Even if they are the same number.

Example:

The gcd of a and b is $2^2 \cdot 3^2 \cdot 5^0 \cdot 7^0 = 4 \cdot 9 = 36$.

The lcm of a and b is $2^3 \cdot 3^3 \cdot 5^1 \cdot 7^1 = 7560$.

The product of a and b is $2^{3+2} \cdot 3^{2+3} \cdot 5^{1+0} \cdot 7^{0+1} = 32 \cdot 243 \cdot 5 \cdot 7 = 272160$.

⁴ $ax + by$ is a *linear combination* because no higher power than 1 is used as an exponent of either x or y (or a or b for that matter).

Theorem A.3 (Chapter 6) (*Linear Combination*): If $d \mid a$ and $d \mid b$ then $d \mid ax + by$ for any integers x and y .⁴

Proof: Use the definition of ‘|’ to demonstrate: there is a j such that $dj = a$ (by definition of $d \mid a$), and there is a k such that $dk = b$ (by definition of $d \mid b$). These are given as premises of the theorem. Do these givens guarantee the existence of an m such that $dm = ax + by$ for any x and y ? Yes, because m is just $(ax + by)/d$, which (substituting equals for equals) is $(djx + dky)/d$, which is $jx + ky$.

For example, let $d = 3$, $a = 33$, and $b = 51$. Then $j = 11$, because $3 \cdot 11 = 33$, and $k = 17$, because $3 \cdot 17 = 51$. Then $m = 11x + 17y$ is such that $3m = 33x + 51y$ — in other words, $d \mid 33x + 51y$ — for any x and y .

Theorem A.4 (Chapter 6) (*Bézout*): If a and b are coprime, then there exist integers x and y such that $ax + by = 1$.

Proof Sketch: By the Linear Combination (LC) Theorem, if the LC $ax + by$ is greater than one, we can always find a new x and y to create a smaller positive integer. And a smaller one after that. And smaller still after that. “Rinse and repeat” the process and eventually you have to reach one.

For example, $33(-1) + 28(2) = 23$. Here we let $a = 33$ and $b = 28$, which are coprime because their greatest common divisor is 1. Note that 23 is certainly greater than 1, so the condition of the previous paragraph is

met. Also, 23 does not divide 33, so the Division Theorem tells us there will be a non-zero remainder. Thus by this is seen that $10 = 33(2) + 28(-2)$:

$$\begin{array}{rcl} n & = & d \quad (q) \quad + \quad r \\ 33 & = & 23 \quad (1) \quad + \quad 10 \end{array}$$

Figure A.1 and the following chain of equalities explain this result:

$$\begin{array}{lll} 10 & & 1 \\ = & & \\ 33 - 23(1) & & 2 \\ = & & \\ 33(1) + 23(-1) & & 3 \\ = & & \\ 33(1) + (33(-1) + 28(2))(-1) & & 4 \\ = & & \\ 33(1) + 33(-1)(-1) + 28(2)(-1) & & 5 \\ = & & \\ 33(1 + (-1)(-1)) + 28(2)(-1) & & 6 \\ = & & \\ 33(2) + 28(-2) & & . \end{array}$$

Hence 10 is also a linear combination of 33 and 28: $33(2) + 28(-2) = 10$, which is smaller than 23.

Rinse and repeat with 10.

10 does not divide 33, so use the Division Theorem again:

$$\begin{array}{rcl} n & = & d \quad (q) \quad + \quad r \\ 33 & = & 10 \quad (3) \quad + \quad 3 \end{array}$$

and by similar (albeit compressed) reasoning:

$$\begin{array}{lll} 3 & & = \\ = & & \\ 33 - 10(3) & & = \\ = & & \\ 33(1) + 10(-3) & & = \\ = & & \\ 33(1) + (33(2) + 28(-2))(-3) & & = \\ = & & \\ 33(1) + 33(2)(-3) + 28(-2)(-3) & & = \\ = & & \\ 33(1 + (2)(-3)) + 28(-2)(-3) & & = \\ = & & \\ 33(-5) + 28(6) & & = \\ = & & \\ 3 & & . \end{array}$$

Thus 3 is also a linear combination of 33 and 28: $33(-5) + 28(6) = 3$, which is smaller than 10.

Rinse and repeat with 3.

3 does divide 33, but it cannot also divide 28 because, remember, 33 and 28 are coprime. So use 28 for this last time:

$$\begin{array}{rcl} n & = & d \quad (q) \quad + \quad r \\ 28 & = & 3 \quad (9) \quad + \quad 1 \end{array}$$

and the coast is clear:

1. isolating 10 on the RHS and swapping LHS and RHS
2. to consistently use '+' and the $ax + by$ form
3. substituting for 23 the initial LC that equals it
4. distributing the rightmost -1
5. factoring out the common 33
6. simplifying

Figure A.1: Equal by virtue of the given reason. LHS is Left-Hand Side, and RHS is Right-Hand Side.

$$\begin{array}{rcl}
 1 & = & \\
 28 - 3(9) & = & \\
 28(1) + 3(-9) & = & \\
 28(1) + (33(-5) + 28(6))(-9) & = & \\
 28(1) + 33(-5)(-9) + 28(6)(-9) & = & \\
 28(1) + 28(6)(-9) + 33(-5)(-9) & = & \\
 28(1 + (6)(-9)) + 33(-5)(-9) & = & \\
 28(-53) + 33(45) & = & \\
 33(45) + 28(-53) & = & \\
 1 & . &
 \end{array}$$

So 45 and -53 are the x and y that make $33x + 28y = 1$.

You are encouraged to study this argument carefully, and convince yourself that it works for any coprime a and b .

Theorem A.5 (Chapter 6) (*Generalized Bézout*): *The existence claim of Bézout's Theorem can be generalized, thus: If $\gcd(a, b) = g$, then there exist x and y such that $ax + by = g$.*

Proof: We note that if g is a 's and b 's greatest common divisor, dividing both by g results in two coprime integers, call them $c (= a/g)$ and $d (= b/g)$. By Bézout's Theorem, $cx + dy = 1$, which after multiplying everything by g becomes $gcx + gdy = g$, which is $g(a/g)x + g(b/g)y = g$, which is $ax + by = g$, as desired.

Theorem A.6 (Chapter 6) (*Euclid's First*): *If p is prime and $p \mid ab$ then $p \mid a$ or $p \mid b$.*

This theorem is of prime importance, because the Fundamental Theorem of Arithmetic follows from (a generalization of) it (although we will not show that proof here). But it is good to walk through sketches of the steps of proving Euclid's First Theorem.⁵

Proof Sketch: Like we did above, we'll get our hands dirty and let $p = 3$, $a = 35$, and $b = 42$.

So $ab = 35 \cdot 42 = 1470$, and indeed $3 \mid 1470$ because $3 \cdot 490 = 1470$.

If $3 \mid 35$ were the case, we would be done; but it's not, so we must show the other alternative, $3 \mid 42$. (We only need to show one or the other.) But the fact that $3 \nmid 35$ means 35 and 3 are coprime, because a prime's only divisors are itself and 1. Hence by Bézout's Theorem we know that $35x + 3y = 1$ for some x and y . Thus, multiplying each side of that equation by $b = 42$ we get that $35(42)x + 3(42)y = 1470x + 3(42)y = 42$.

Hence by the following argument $3 \mid 42$ (so either way $3 \mid 35$ or $3 \mid 42$):

If $3 \mid 1470$ then $1470 = 3q$ for some integer q (which we know to be 490). Then $1470x + 3(42)y = 42$ becomes $3(490)x + 3(42)y = 42$ becomes $3(490x + 42y) = 42$, which means there exists an integer, $i = 490x + 42y$, such that $3i = 42$, which means $3 \mid 42$. Done!

⁵ It was his second theorem that asserted that there is an infinitude of primes.

Theorem A.7 (Chapter 6) (*Congruence Cancellation*): If $ax \equiv_m ay$ and if a and m are coprime, then $x \equiv_m y$.

Proof Sketch: Let 3 and 5 be representative of any coprime a and m to concretely walk through the proof. Note that $3x \equiv_5 3y$ implies $x \equiv_5 y$ because:

1. $3x \equiv_5 3y$ means $5 \mid 3x - 3y$, which means $5 \mid 3(x - y)$.
2. Since 3 and 5 are coprime, then by the same reasoning used in the proof of Euclid's First Theorem, $5 \mid x - y$.
3. Thus $x \equiv_5 y$.

Theorem A.8 (Chapter 6) (*Wilson*): n is prime if and only if $(n - 1)! \equiv_n (n - 1)$.

Proof Sketch: For example, 11 is prime because 11 divides 3628790, which is $10! - 10$.

The Congruence Cancellation Theorem (among others) can be used to prove this theorem. How exactly this can be done is a GPAO. (Hint: study Exercise 324.)

Theorem A.9 (Chapter 6) (*Modular Multiplicative Inverse Existence*): The modulo m multiplicative inverse of a , call it i , exists if and only if m and a are coprime.

Recall that the number a is said to have a modulo m multiplicative inverse (MMI) if there is some integer i such that $ai \equiv_m 1$.

For example, 9 is an MMI mod 17 of 2 because $2 \cdot 9 = 18$ and $18 \equiv_{17} 1$.

Proof: Recall that by Bézout's Theorem, a and m being coprime means that there are integers x and y for which $ax + my = 1$. But then x fits the definition of the mod m inverse i , because modulo m , any multiple of m (in particular, my) is equivalent to 0. Thus $ax + 0 = 1$ implies $ax \equiv_m 1$.

The inverse is in the following sense unique: if $ax \equiv_m 1$ and $ay \equiv_m 1$, then $ax \equiv_m ay$, and $x \equiv_m y$ follows from the Congruence Cancellation Theorem. In other words, if x and y are both MMIs of a , they are congruent to each other.

For example, $3 \cdot 4 = 12$, and $12 \equiv_{11} 1$. Also $3 \cdot 15 = 45$, and $45 \equiv_{11} 1$, so 4 and 15 are both mod 11 MMIs of 3. And indeed, $4 \equiv_{11} 15$. But likewise 26, 37, 48, ..., and as well, $-7, -18, -29, \dots$, are also mod 11 MMIs of 3. In fact, qualifying as a mod 11 MMI of 3 is every integer of the form $4 + 11 \cdot k$ where k is any integer.⁶

⁶ Remember, any multiple of a modulus m vanishes (looks like zero) when viewed through the lens of mod m arithmetic.

B

A Sieve to Remember

The **Sieve of Eratosthenes** starts with a list of numbers up to a certain limit (99 in this case):

2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	32	33	34	35	36
37	38	39	40	41	42	43
44	45	46	47	48	49	50
51	52	53	54	55	56	57
58	59	60	61	62	63	64
65	66	67	68	69	70	71
72	73	74	75	76	77	78
79	80	81	82	83	84	85
86	87	88	89	90	91	92
93	94	95	96	97	98	99

We take these integers and manually sieve them — filtering out all nonprimes — by crossing out every other number (after 2 — so 4, 6, 8, etc. are crossed out), which excludes the multiples of 2, every third number (after 3), which drops the multiples of 3, every fifth number (after 5) to filter out the multiples of 5, etc. Note that some numbers (e.g., the multiples of 6) get crossed out twice — once for the multiples-of-2 sieving, once for the multiples-of-3 sieving — and this is an acceptable redundancy, as it avoids the continual use of a conditional that says only cross a number out if it is not already crossed out.

2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	32	33	34	35	36
37	38	39	40	41	42	43
44	45	46	47	48	49	50
51	52	53	54	55	56	57
58	59	60	61	62	63	64
65	66	67	68	69	70	71
72	73	74	75	76	77	78
79	80	81	82	83	84	85
86	87	88	89	90	91	92
93	94	95	96	97	98	99

Now cross out the multiples of 3:

2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	32	33	34	35	36
37	38	39	40	41	42	43
44	45	46	47	48	49	50
51	52	53	54	55	56	57
58	59	60	61	62	63	64
65	66	67	68	69	70	71
72	73	74	75	76	77	78
79	80	81	82	83	84	85
86	87	88	89	90	91	92
93	94	95	96	97	98	99

Now cross out the multiples of 5:

2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	32	33	34	35	36
37	38	39	40	41	42	43
44	45	46	47	48	49	50
51	52	53	54	55	56	57
58	59	60	61	62	63	64
65	66	67	68	69	70	71
72	73	74	75	76	77	78
79	80	81	82	83	84	85
86	87	88	89	90	91	92
93	94	95	96	97	98	99

Now cross out the three remaining multiples of 7:

2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	32	33	34	35	36
37	38	39	40	41	42	43
44	45	46	47	48	49	50
51	52	53	54	55	56	57
58	59	60	61	62	63	64
65	66	67	68	69	70	71
72	73	74	75	76	77	78
79	80	81	82	83	84	85
86	87	88	89	90	91	92
93	94	95	96	97	98	99

Now to do with code what we just did manually (but only up to 25). We cross out a number by negating it (making it negative) and must use a conditional to avoid undoing that negation once done.

```

def negate_multiple(n, m):
    if n != m and n % m == 0:
        return n if n < 0 else -n
    else:
        return n

negate_multiples_of_2 = lambda n:negate_multiple(n, 2)
negate_multiples_of_3 = lambda n:negate_multiple(n, 3)
negate_multiples_of_5 = lambda n:negate_multiple(n, 5)

all_numbers = list(range(2, 26))
all_minus_multiples_of_2 = \
    list(map(negate_multiples_of_2, all_numbers))

all_minus_multiples_of_2_and_3 = \
    list(map(negate_multiples_of_3, all_minus_multiples_of_2))

all_minus_multiples_of_2_and_3_and_5 = \
    list(map(negate_multiples_of_5, all_minus_multiples_of_2_and_3))

positive = lambda n:n > 0

print(all_minus_multiples_of_2)
print(all_minus_multiples_of_2_and_3)
print(all_minus_multiples_of_2_and_3_and_5)
print(list(filter(positive, all_minus_multiples_of_2_and_3_and_5)))

```

Prints out:

```

[2, 3, -4, 5, -6, 7, -8, 9, -10, 11, -12, 13, -14, 15, \
-16, 17, -18, 19, -20, 21, -22, 23, -24, 25]
[2, 3, -4, 5, -6, 7, -8, -9, -10, 11, -12, 13, -14, -15, \
-16, 17, -18, 19, -20, -21, -22, 23, -24, 25]
[2, 3, -4, 5, -6, 7, -8, -9, -10, 11, -12, 13, -14, -15, \
-16, 17, -18, 19, -20, -21, -22, 23, -24, -25]
[2, 3, 5, 7, 11, 13, 17, 19, 23]

```

This clever sieve¹ exemplifies *not-literally-recursive* recursion:

```
def eratox(primes=[2]):
    """
    find more primes up to the limit of
    one plus the square of the largest known prime;

    Examples
    -----
    >>> eratox()
    [2, 3]
    >>> eratox(eratox())
    [2, 3, 5, 7]
    >>> eratox(eratox(eratox()))
    [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

    """
    px = primes[-1] # the largest known prime;
    n = px ** 2 + 1 # extend the limit of new primes search;
    multiples = [i for p in primes for i in range(p * 2, n, p) if i > px]
    newprimes = set(range(px + 1, n)) - set(multiples)
    return primes + sorted(newprimes)
```

¹A variant capable of recursion, aka incremental sieve.

C

An Astounding Fact

Theorem C.1 (chapter 6) (*The Consecutive Highly Composites Theorem*):

Let r and n be positive integers. There exist r consecutive numbers each divisible by at least n distinct primes.

Proof:

Let

$$\left\{ \begin{array}{ll} p_1, \dots, p_n & \text{be the first } n \text{ primes} \\ p_{n+1}, \dots, p_{2n} & \text{be the second } n \text{ primes} \\ \vdots & \\ p_{(r-1)n+1}, \dots, p_{rn} & \text{be the } r^{\text{th}} \text{ set of } n \text{ primes} \end{array} \right.$$
$$\left\{ \begin{array}{ll} m_1 & = p_1 \cdots p_n \\ m_2 & = p_{n+1} \cdots p_{2n} \\ \vdots & \\ m_r & = p_{(r-1)n+1} \cdots p_{rn} \end{array} \right.$$

and consider the simultaneous congruences

$$\left\{ \begin{array}{ll} x & \equiv -1 \pmod{m_1} \\ x & \equiv -2 \pmod{m_2} \\ \vdots & \\ x & \equiv -r \pmod{m_r} \end{array} \right.$$

The Chinese Remainder Theorem guarantees a solution

$$x = N \pmod{m_1 \cdots m_r}.$$

Hence $m_1 | (N+1)$, $m_2 | (N+2)$, ..., and $m_r | (N+r)$.

So the r numbers, $N+1, N+2, \dots, N+r$, are each divisible by at least n primes!

