

Theory of Computation

Making Connections



Jim Hefferon

<https://hefferon.net/computation>

Notation summary

Notation	Description
$\mathcal{P}(S)$	power set, collection of all subsets of S
S^c	complement of the set S
$\mathbb{1}_S$	characteristic function of the set S
$\langle a_0, a_1, \dots \rangle$	sequence
$\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$	natural numbers { 0, 1, ... }, integers, rationals, reals
a, b, ... 0, 1	character (note the typeface)
Σ	alphabet, set of characters
\mathbb{B}	alphabet of bits, $\mathbb{B} = \{ 0, 1 \}$
σ, τ	strings (any lower-case Greek letter except ϕ)
ε	empty string
Σ^*	set of all strings over the alphabet
\mathcal{L}	language, a subset of Σ^*
\mathcal{P}	Turing machine
ϕ	function computed by a Turing machine
$\phi(x)\downarrow, \phi(x)\uparrow$	function converges on that input, or diverges
\mathcal{G}	graph
\mathcal{M}	Finite State machine
$\mathcal{O}(f)$	order of growth of the function
C	complexity class
\mathbf{Prob}	problem
\mathcal{V}	verifier for an NP language

Greek letters with pronunciation

Character	Name	Character	Name
α	alpha AL-fuh	ν	nu NEW
β	beta BAY-tuh	ξ, Ξ	xi KSIGH
γ, Γ	gamma GAM-muh	\omicron	omicron OM-uh-CRON
δ, Δ	delta DEL-tuh	π, Π	pi PIE
ε	epsilon EP-suh-lon	ρ	rho ROW
ζ	zeta ZAY-tuh	σ, Σ	sigma SIG-muh
η	eta AY-tuh	τ	tau TOW as in cow
θ, Θ	theta THAY-tuh	υ, Υ	upsilon OOP-suh-LON
ι	iota eye-OH-tuh	ϕ, Φ	phi FEE or FI as in high
κ	kappa KAP-uh	χ	chi KI as in high
λ, Λ	lambda LAM-duh	ψ, Ψ	psi SIGH or PSIGH
μ	mu MEW	ω, Ω	omega oh-MAY-guh

COVER PHOTO: Bonus Bureau, Computing Division, 1924-Nov-24.
 Calculating the bonus owed to each US WW I veteran.
 (Auto-generated cropping decoration added.)

This PDF was compiled 2024-Jan-11.

Preface

The Theory of Computation is a wonderful thing. It is beautiful. It has deep connections with other areas in mathematics as well as with the wider intellectual world. It is full of ideas, exciting and arresting ideas, many of which apply directly to practical computing. And, looking forward into this century, clearly a theme will be the power and limits of computation. So it is timely, too.

It makes a delightful course. Its organizing question — what can be done? — is both natural and compelling. Students see the contrast between computation’s capabilities and limits. There are well understood principles and within reach are as-yet unknown areas.

This text aims to reflect all of that: to be precise, topical, insightful, stimulating, and perhaps sometimes even delightful.

For students Have you ever wondered, while you were learning to instruct computers to do your bidding, what cannot be done? And what can be done in principle but not in practice? In this course you will see the signpost results in the study of these questions and you will learn to use the tools to address these issues as they come up in your work.

We will consider the very nature of computation. This has been intensively studied for a century so you will not see all that is known, but you will see enough to end with key insights and with a better understanding of where the profession that you are entering stands.

We do not stint on precision — why would we want to? — but we approach the ideas liberally; in a way that, in addition to technical detail, also attends to a breadth of knowledge. We will be eager to make connections with other fields, with things that you have previously studied, and with a variety of modes of thinking, most importantly computational thinking. People learn best when the topic fits into a whole, as several of the quotes below express.

The presentation here encourages you to be an active learner: to explore and reflect on the motivation, development, and future of those ideas. It gives you the chance to follow things that intrigue you, including that in the back of the book are lots of notes to the main text, many of which contain links that will take you even deeper. There are also Extra sections at the end of each chapter to help you explore further. Whether or not your instructor covers them formally in class, they can further your understanding of the material and where it leads.

The subject is big and a challenge. It will change the way that you see the world. It is also a great deal of fun. Enjoy!

For instructors We cover the definition of computability, unsolvability, languages and grammars, automata, and complexity. The audience is undergraduate majors in Computer Science, Mathematics, and nearby areas.

The prerequisite, besides introductory programming, is Discrete Mathematics. We rely on propositional logic, proof methods including induction, graphs, basic

number theory, sets, functions, and relations. For non-Computer Science students without some of these topics, appendices establish notation and terminology for strings, functions, and propositional logic, and there are brief sections on graphs and Big- \mathcal{O} (this section requires derivatives).

A text does its readers a disservice if it is not precise. Details matter. But students can also fail to understand a subject because they have not had a chance to reflect on the underlying ideas. The presentation here stresses motivation and naturalness and, where practical, sets the results in a network of connections.

The first example comes right at the start, where we begin with Turing machines. The alternative of first covering Finite State machines may well be mathematically slicker but to a fresh learner it is more natural to instead start by asking what can be computed at all. We follow the definition of computable function with an extensive discussion of Church's Thesis, relying on the intuition that students have from their programming experience. This discussion also justifies giving algorithms throughout the book in outline rather than as programs for a computation model or as intricate recursions, which better communicates the ideas.

A second example of choosing naturalness and making connections is nondeterminism. We introduce it in the context of Finite State machines, along with a discussion promoting intuition so that when it appears again in Complexity, we can rely on this understanding to develop the standard definition that a language is in NP if it has a polytime verifier. A third example is the inclusion of a section introducing the kinds of problems that drive the work in Complexity today. Still another example is the discussion of the current state of P versus NP . These and many more, taken together, encourage students to develop the habit of inquiry. Stuff should make sense.

Exploration and Enrichment The Theory of Computation is fascinating. This book aims to showcase that, to draw readers in, to be absorbing. It uses lively language and many illustrations.

One way to stimulate readers is to make the material explorable. Where practical, references are clickable. For example, each picture of a founder of the subject is a link to their Wikipedia page. This makes them very much more likely to be the subject of further reading than is the same content in a physical library.

The presentation here also encourages engagement through the many notes in the back that fill out, and add a spark to, the core discussion.

Another example of enrichment in this text is a willingness to include informal discussions. Informality has the potential to be a problem, which is why it is carefully differentiated, but it can also be very valuable. Who among us has not had an Ah-ha! moment in a hand-wavy hallway conversation?

Finally, students can also explore the end of chapter topics. They cover a number of subjects related to the chapter and are suitable as one-day lectures, or for group work or extra credit, or for a student just to read for pleasure.

Schedule Chapter I defines models of computation, Chapter II covers unsolvability,

Chapter III does languages and graphs, Chapter IV is automata, and Chapter V is computational complexity. I assign the readings as homework and quiz on them. (For those working independently I have marked a selection of exercises with \checkmark .)

	Sections	Reading	Notes
Week 1	I.1, I.3	I.2	
2	I.4, II.1	II.A	
3	II.2, II.3		
4	II.4, II.5	II.B	
5	II.6, II.7	II.C	
6	II.9	III.A	EXAM
7	III.1–III.3		
8	IV.1, IV.2		
9	IV.3, IV.3	IV.A	
10	IV.4, IV.5		
11	IV.7	IV.1.4	EXAM
12	V.1, V.2	V.A	
13	V.3, V.4	V.3.2	
14	V.5, V.6	V.B	
15	V.7		

License This book is Free. You can download and use it without cost. If you are a teacher then you can post it on the Learning Management System for your course. (Or you can get paper copies if you like that better.) You can also modify the \LaTeX source. For the full details, see the home page <https://hefferon.net/computation>.

One reason that the book is Free is that it is written in \LaTeX , which is Free, as is Asymptote which drew the illustrations, along with Emacs and all of GNU software, and the entire Linux platform on which this book was developed. And anyway, the research that this text presents was all made freely available by scholars.

Beyond those reasons, there is a long tradition of making educational work open. I believe that the synthesis here adds value—I hope so, indeed—but the masters have left a well-marked trail and following it seems right.

Acknowledgments I owe a great debt to my wife, whose patience with this project has gone beyond all reasonable bounds. Thank you, Lynne.

My students have made the book better in so many ways. Thank you to you all for those contributions.

And, I must honor my teachers. First among them is M Lerman. Thank you, Manny.

They also include H Abelson, GJ Sussmann, and J Sussmann, whose *Structure and Interpretation of Computer Programs* dared to show students how mind-blowing it all is. When I see a computer text whose examples are about managing inventory in a used car dealership, I can only say: Thank you, for believing in me.

Memory works far better when you learn networks of facts rather than facts in isolation.

– T Gowers, WHAT MATHS A-LEVEL DOESN'T NECESSARILY GIVE YOU

Research into learning shows that content is best learned within context . . . , when the learner is active, and that above all, when the learner can actively construct knowledge by developing meaning and 'layered' understanding.

– A W (Tony) Bates, TEACHING IN A DIGITAL AGE

Teach concepts, not tricks.

– G Rota, TEN LESSONS I WISH I HAD LEARNED BEFORE I STARTED TEACHING DIFFERENTIAL EQUATIONS

[W]hile many distinguished scholars have embraced [the Jane Austen Society] and its delights since the founding meeting, ready to don period dress, eager to explore antiquarian minutiae, and happy to stand up at the Saturday-evening ball, others, in their studies of Jane Austen's works, . . . have described how, as professional scholars, they are rendered uneasy by this performance of pleasure at [the meetings]. . . . I am not going to be one of those scholars.

– E Bander, PERSUASIONS, 2017

The power of modern programming languages is that they are expressive, readable, concise, precise, and executable. That means we can eliminate middleman languages and use one language to explore, learn, teach, and think.

– A Downey, PROGRAMMING AS A WAY OF THINKING

Of what use are computers? They can only give answers.

– P Picasso, THE PARIS REVIEW, SUMMER-FALL 1964

Jim Hefferon
Jericho, VT USA
University of Vermont
hefferon.net
Version 1.10, 2023-Jun-23

Contents

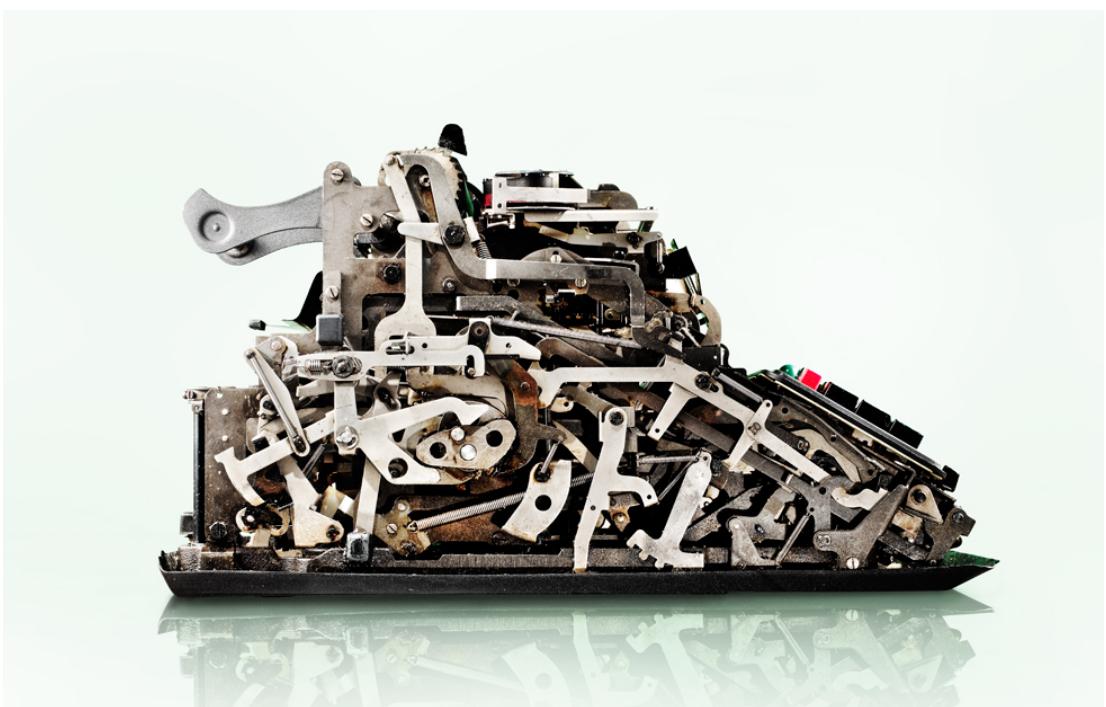
I Mechanical Computation	3
1 Turing machines	3
Definition	4
Computable functions	9
2 Church's Thesis	14
History	14
Evidence	15
What it does not say	17
An empirical question?	17
Using Church's Thesis	18
3 Recursion	21
Primitive recursion	21
4 General recursion	30
Ackermann functions	30
μ recursion	33
A Turing machine simulator	37
B Hardware	39
C Game of Life	43
D Ackermann's function is not primitive recursive	46
E LOOP programs	50
II Background	57
1 Infinity	57
Cardinality	57
2 Cantor's correspondence	64
3 Diagonalization	72
Diagonalization	72
4 Universality	78
Universal Turing machine	79
Uniformity	81
Parametrization	82
5 The Halting problem	87
Definition	87
General unsolvability	89
Discussion	92
6 Rice's Theorem	98
7 Computably enumerable sets	103
8 Oracles	108
9 Fixed point theorem	116
When diagonalization fails	117

Discussion	119																																																																
A Hilbert's Hotel	122																																																																
B The Halting problem in intellectual culture	123																																																																
C Self Reproduction	126																																																																
D Busy Beaver	128																																																																
E Cantor in Code	132																																																																
III Languages and graphs	139																																																																
1 Languages	139																																																																
2 Grammars	143	Definition	144	3 Graphs	155	Definition	155	Paths	156	Graph representation	157	Colors	158	Graph isomorphism	159	A BNF	165	B Tree traversal	169	IV Automata	175	1 Finite State machines	175	Definition	175	2 Nondeterminism	185	Motivation	186	Definition	188	ϵ transitions	190	Equivalence of the machine types	194	3 Regular expressions	200	Definition	200	Kleene's Theorem	203	4 Regular languages	210	Definition	210	Closure properties	211	5 Languages that are not regular	216	6 Minimization	222	7 Pushdown machines	233	Definition	234	A Regular expressions in the wild	241	B The Myhill-Nerode Theorem	249	V Computational Complexity	257	1 Big \mathcal{O}	257	Motivation	258
Definition	144																																																																
3 Graphs	155	Definition	155	Paths	156	Graph representation	157	Colors	158	Graph isomorphism	159	A BNF	165	B Tree traversal	169	IV Automata	175	1 Finite State machines	175	Definition	175	2 Nondeterminism	185	Motivation	186	Definition	188	ϵ transitions	190	Equivalence of the machine types	194	3 Regular expressions	200	Definition	200	Kleene's Theorem	203	4 Regular languages	210	Definition	210	Closure properties	211	5 Languages that are not regular	216	6 Minimization	222	7 Pushdown machines	233	Definition	234	A Regular expressions in the wild	241	B The Myhill-Nerode Theorem	249	V Computational Complexity	257	1 Big \mathcal{O}	257	Motivation	258				
Definition	155																																																																
Paths	156																																																																
Graph representation	157																																																																
Colors	158																																																																
Graph isomorphism	159																																																																
A BNF	165																																																																
B Tree traversal	169																																																																
IV Automata	175																																																																
1 Finite State machines	175	Definition	175	2 Nondeterminism	185	Motivation	186	Definition	188	ϵ transitions	190	Equivalence of the machine types	194	3 Regular expressions	200	Definition	200	Kleene's Theorem	203	4 Regular languages	210	Definition	210	Closure properties	211	5 Languages that are not regular	216	6 Minimization	222	7 Pushdown machines	233	Definition	234	A Regular expressions in the wild	241	B The Myhill-Nerode Theorem	249	V Computational Complexity	257	1 Big \mathcal{O}	257	Motivation	258																						
Definition	175																																																																
2 Nondeterminism	185	Motivation	186	Definition	188	ϵ transitions	190	Equivalence of the machine types	194	3 Regular expressions	200	Definition	200	Kleene's Theorem	203	4 Regular languages	210	Definition	210	Closure properties	211	5 Languages that are not regular	216	6 Minimization	222	7 Pushdown machines	233	Definition	234	A Regular expressions in the wild	241	B The Myhill-Nerode Theorem	249	V Computational Complexity	257	1 Big \mathcal{O}	257	Motivation	258																										
Motivation	186																																																																
Definition	188																																																																
ϵ transitions	190																																																																
Equivalence of the machine types	194																																																																
3 Regular expressions	200	Definition	200	Kleene's Theorem	203	4 Regular languages	210	Definition	210	Closure properties	211	5 Languages that are not regular	216	6 Minimization	222	7 Pushdown machines	233	Definition	234	A Regular expressions in the wild	241	B The Myhill-Nerode Theorem	249	V Computational Complexity	257	1 Big \mathcal{O}	257	Motivation	258																																				
Definition	200																																																																
Kleene's Theorem	203																																																																
4 Regular languages	210	Definition	210	Closure properties	211	5 Languages that are not regular	216	6 Minimization	222	7 Pushdown machines	233	Definition	234	A Regular expressions in the wild	241	B The Myhill-Nerode Theorem	249	V Computational Complexity	257	1 Big \mathcal{O}	257	Motivation	258																																										
Definition	210																																																																
Closure properties	211																																																																
5 Languages that are not regular	216																																																																
6 Minimization	222																																																																
7 Pushdown machines	233	Definition	234	A Regular expressions in the wild	241	B The Myhill-Nerode Theorem	249	V Computational Complexity	257	1 Big \mathcal{O}	257	Motivation	258																																																				
Definition	234																																																																
A Regular expressions in the wild	241																																																																
B The Myhill-Nerode Theorem	249																																																																
V Computational Complexity	257																																																																
1 Big \mathcal{O}	257	Motivation	258																																																														
Motivation	258																																																																

Definition	261
Tractable and intractable	265
Discussion	266
2 A problem miscellany	272
Problems, with stories	272
More problems, omitting the stories	276
3 Problems, algorithms, and programs	287
Problem types	288
Statements and representations	291
4 P	296
Definition	297
Effect of the model of computation	298
Naturalness	299
5 NP	303
Nondeterministic Turing machines	303
Speed	305
Definition	306
6 Reductions between problems	314
7 NP completeness	326
$P = NP?$	332
Discussion	335
8 Other classes	341
EXP	341
Time Complexity	342
Space Complexity	343
The Zoo	344
A RSA Encryption	345
B Good-enoughness	351
C SAT solvers	353
Appendix	361
A Strings	362
B Functions	364
C Propositional logic	368
Notes	374
Bibliography	409

Part One

Classical Computability



CHAPTER

I Mechanical Computation

What can be computed? For instance, the function that doubles its input, that takes in x and puts out $2x$, is intuitively mechanically computable. We shall call such functions **effective**.

The question asks for the things that can be computed more than it asks for how to compute them. In this Part we will be more interested in the function, in the input-output behavior, than in the details of implementing that behavior.

SECTION

I.1 Turing machines

Despite this desire to downplay implementation, we follow the approach of A Turing that the first step toward defining the set of computable functions is to reflect on the details of what mechanisms can do.

The context of Turing's thinking was the *Entscheidungsproblem*,[†] proposed in 1928 by D Hilbert and W Ackermann, which asks for an algorithm that decides, after taking as input a mathematical statement, whether that statement is true or false. So he considered the kind of symbol-manipulating computation familiar in mathematics, such as when we expand nested brackets or verify a step in a plane geometry proof.

After reflecting on it for a while, one day after a run[‡] Turing laid down in the grass and imagined a clerk doing by-hand multiplication with a sheet of paper that gradually becomes covered with columns of numbers. With this as a prototype, Turing posited conditions for the computing agent.

First, it (or he or she) has a memory facility, such as the clerk's paper, where it can put information for later retrieval.



Alan Turing 1912–
1954

Second, the computing agent must follow a definite procedure, a precise set of instructions with no room for creative leaps. Part of what makes the procedure definite is that the instructions don't involve random methods, such as counting clicks from radioactive decay to determine which of two possibilities to perform.

The other thing making the procedure definite is that the agent is discrete—it does not use continuous methods or analog devices. Thus there is no question about the precision of operations as there might be when reading results off of a

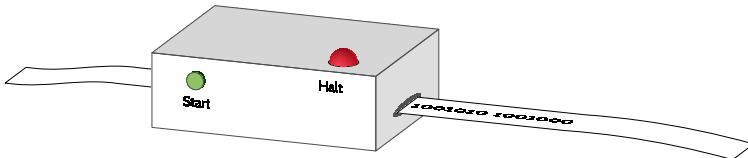
IMAGE: copyright Kevin Twomey, <http://kevintwomey.com/lowtech.html> [†]German for “decision problem.” Pronounced *en-SHY-duns-pob-lem*. [‡]He was a serious candidate for the 1948 British Olympic marathon team.

slide rule or an instrument dial. In line with this, the agent works in a step-by-step fashion. If needed they could pause between steps, note where they are (“about to carry a 1”), and pick up again later. We say that at each moment the clerk is in one of a finite set of possible **states**, which we denote q_0, q_1, \dots

Turing’s third condition arose because he wanted to investigate what is computable in principle. He therefore imposed no upper bound on the amount of available memory. More precisely, he imposed no finite upper bound—should a calculation threaten to run out of storage space then more is provided. This includes imposing no upper bound on the amount of memory available for inputs or for outputs and no bound on the amount of extra storage, scratch memory, needed in addition to that for inputs and outputs.[†] He similarly put no upper bound on the number of instructions. And, he left unbounded the number of steps that a computation performs before it finishes.[‡]

The final question Turing faced is: how smart is the computing agent? For instance, can it multiply? We don’t need to include a special facility for multiplication because we can in principle multiply via repeated addition. We don’t even need addition because we can repeat the successor operation, the add-one operation. In this way Turing pared the computing agent down until it is quite basic, quite easy to understand, until the operations are so elementary that we cannot easily imagine them further divided, while still keeping that agent powerful enough to do anything that can in principle be done.

Definition Based on these reflections, Turing pictured a box containing a mechanism and fitted with a tape.



The tape is the memory, sometimes called the ‘store’. The box can read from it and write to it, one character at a time, as well as move a read/write head relative to the tape in either direction. Thus, to multiply, the computing agent can start by reading the two input multiplicands from the tape (the drawing shows 74 and 72 in binary, separated by a blank), can use the tape for scratch work, and can halt with the output written on the tape.

The box is the computing agent, the CPU, sometimes called the ‘control’. The

[†]True, every existing physical computer has bounded memory, putting aside storing things in the Cloud. However, that space is extremely large. In this Part, when working with the model devices, imposing a bound on memory is a hindrance or at best irrelevant. [‡]Some authors describe the availability of resources such as the amount of memory as ‘infinite’. Turing himself does this. A reader may object that this violates the goal of the definition, to model in-principle-physically-realizable computations, and so the development here instead says that the resources have no finite upper bound. But really, it doesn’t matter. If we show that something cannot be computed when there are no bounds then we have shown that it cannot be computed on any real-world device.

Start button sets the computation going. When the computation is finished the Halt light comes on. The engineering inside the box is not important — perhaps it has integrated circuits like the machines that we are used to or perhaps it has gears and levers or perhaps LEGO's — what matters is that each of its finitely many parts can only be in finitely many states. If it has chips then each register has a finite number of possible values, while if it is made with gears or bricks then each settles in only a finite number of possible positions. Thus, however it is made, in total the box has only finitely many states.

While executing a calculation, the mechanism steps from state to state. For instance, an agent doing multiplication may determine, because of what state it is in now and because of what it is reading on the tape, that they next need to carry a 1. The agent transitions to a new state, one whose intuitive meaning is that it is where carries take place.

Consequently, machine steps involve four pieces of information. Call the present state q_p and the next state q_n . The symbol that the read/write head is presently pointing to is T_p . Finally, the next tape action is T_n . Possible actions are: moving the tape head left or right without writing, which we denote with $T_n = L$ or $T_n = R$,[†] or writing a symbol to the tape without moving the head, which we denote with that symbol, so that $T_n = 1$ means the machine will write a 1 to the tape. As to the set of characters that can go on the tape, we will choose whatever is convenient for the job we are doing. However every tape has blanks in all but finitely many places and so that must be one of the symbols. (We denote blank with B when an empty space could cause confusion.)

The four-tuple $q_p T_p T_n q_n$ is an **instruction**. For example, the instruction $q_3 1 B q_5$ is executed only if the machine is now in state q_3 and is reading a 1 on the tape. If so, the machine writes a blank to the tape, replacing the 1, and passes to state q_5 .

- 1.1 EXAMPLE This Turing machine with the tape symbol set $\Sigma = \{B, 1\}$ has six instructions.

$$\mathcal{P}_{\text{pred}} = \{q_0 B L q_1, q_0 1 R q_0, q_1 B L q_2, q_1 1 B q_1, q_2 B R q_3, q_2 1 L q_2\}$$

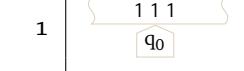
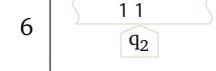
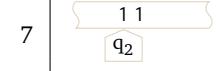
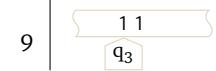
Below we've represented an initial configuration. It shows a stretch of tape along with the machine's state and the position of its read/write head.



We adopt the convention that when we press Start the machine is in state q_0 . The picture above shows the machine reading 1, so instruction $q_0 1 R q_0$ applies. Thus the first step is that the machine moves its tape head right and stays in state q_0 . The first line of the following table shows this and later lines show the configurations

[†]Whether we move the tape or the head doesn't matter, what matters is their relative motion. Thus $T_n = L$ means that either the tape or the head moves so that the head now points one place to the left. In drawings we hold the tape steady and move the head because it simplifies reading the graphics.

after later steps. Briefly, the head slides to the right, blanks out the final 1, and slides back to the start.

Step	Configuration	Step	Configuration
1		6	
2		7	
3		8	
4		9	
5			

Now because there is no $q_3 1$ instruction, the machine halts.

We can think of this as computing the predecessor function

$$\text{pred}(x) = \begin{cases} x - 1 & \text{if } x > 0 \\ 0 & \text{else} \end{cases}$$

because if the machine's initial tape is entirely blank except for n -many consecutive 1's and the head points to the first, then at the end the tape will have $n - 1$ -many 1's (except for $n = 0$, where the tape will end with no 1's).

- 1.2 EXAMPLE We can think of this machine with tape alphabet $\Sigma = \{B, 1\}$ as adding two natural numbers.

$$\begin{aligned} \mathcal{P}_{\text{add}} = \{ &q_0 B B q_1, q_0 1 R q_0, q_1 B 1 q_1, q_1 1 1 q_2, q_2 B B q_3, q_2 1 L q_2, \\ &q_3 B R q_3, q_3 1 B q_4, q_4 B R q_5, q_4 1 1 q_5 \} \end{aligned}$$

The input numbers are represented by two strings of 1's, separated with a blank. The read/write head starts under the first symbol in the first number. This shows the machine ready to compute $2 + 3$.



The machine scans right, looking for the blank separator. It changes that to a 1, then scans left until it finds the start. Finally, it trims off a 1 and halts with the read/write head pointing to the start of the string. Here are the steps.

Step	Configuration	Step	Configuration
1	1 1 1 1 1 q ₀	7	1 1 1 1 1 1 q ₂
2	1 1 1 1 1 q ₀	8	1 1 1 1 1 1 q ₂
3	1 1 1 1 1 q ₁	9	1 1 1 1 1 1 q ₃
4	1 1 1 1 1 1 q ₁	10	1 1 1 1 1 1 q ₃
5	1 1 1 1 1 1 q ₂	11	1 1 1 1 1 1 q ₄
6	1 1 1 1 1 1 q ₂	12	1 1 1 1 1 1 q ₅

Instead of giving a machine's instructions as a list, we can use a table or a diagram. Here is the **transition table** for $\mathcal{P}_{\text{pred}}$ and its **transition graph**.

Δ_{pred}	B	1
q_0	L q_1 R q_0	
q_1	L q_2 B q_1	
q_2	R q_3 L q_2	
q_3	-	-

```

graph LR
    q0((q0)) -- "B, L" --> q1((q1))
    q1 -- "1, B" --> q2((q2))
    q2 -- "1, L" --> q3((q3))
    q3 -- "B, R" --> q2
  
```

And here is the corresponding table and graph for \mathcal{P}_{add} .

Δ_{add}	B	1
q_0	B q_1 R q_0	
q_1	1 q_1 1 q_2	
q_2	B q_3 L q_2	
q_3	R q_3 B q_4	
q_4	R q_5 1 q_5	
q_5	-	-

```

graph LR
    q0((q0)) -- "B" --> q1((q1))
    q1 -- "1, 1" --> q2((q2))
    q2 -- "1, L" --> q3((q3))
    q3 -- "B, B" --> q4((q4))
    q4 -- "1, B" --> q5((q5))
    q5 -- "B, R" --> q4
    q4 -- "1, 1" --> q5
  
```

The graph is how we will most often present machines that are small but if there are lots of states then it can be visually confusing.

Next, a crucial observation. Some Turing machines, for at least some starting configurations, never halt.

- 1.3 EXAMPLE The machine $\mathcal{P}_{\text{inf loop}} = \{ q_0 BBq_0, q_0 11q_0 \}$ never halts, regardless of the input.



The exercises ask for examples of Turing machines that halt on some inputs and not on others.

High time for definitions. We take a **symbol** to be something that the device can write and read, for storage and retrieval.[†]

DEFINITION A **Turing machine** \mathcal{P}^{\ddagger} is a finite set of four-tuple **instructions** $q_p T_p T_n q_n$. In an instruction, the **present state** q_p and **next state** q_n are elements of a **set of states** Q . The **input symbol** or **current symbol** T_p is an element of the **tape alphabet** set Σ , which contains at least two members including one called **blank** (and does not contain L or R). The **action symbol** T_n is an element of the **action set** $\Sigma \cup \{L, R\}$.

The set \mathcal{P} must be **deterministic**: different four-tuples cannot begin with the same $q_p T_p$. Thus, over the set of instructions $q_p T_p T_n q_n \in \mathcal{P}$, the association of present pair $q_p T_p$ with next pair $T_n q_n$ defines a function, the **transition function** or **next-state function** $\Delta: Q \times \Sigma \rightarrow (\Sigma \cup \{L, R\}) \times Q$.

Of course, the point of these machines is what they do. To finish the formalization we now give a complete description of a machine's action.

In tracing through Example 1.1 and Example 1.2 we saw that a Turing machine acts by governing the transitions as that machine moves step by step. A **configuration** of a Turing machine is a four-tuple $\langle q, s, \tau_L, \tau_R \rangle$, where q is a state, a member of Q , s is a character from the tape alphabet Σ , and τ_L and τ_R are strings from Σ^* , including possibly the empty string ε . These signify the current state, the character under the read/write head, and the tape contents to the left and right of the head. For instance, in the trace table of Example 1.2, the 'Step 2' line shows that after two transitions the state is $q = q_0$, the character under the head is the blank $s = B$, to the left of the head is $\tau_L = 11$, and to the right is $\tau_R = 111$. Thus the graphic on that line pictures the configuration $\langle q_0, B, 11, 111 \rangle$. That is, a configuration is a snapshot, an instant in a computation.

We write $\mathcal{C}(t)$ for the machine's configuration after the t -th transition and say that this is the configuration at **step** t . We extend that to step 0 by saying that the **initial configuration** $\mathcal{C}(0)$ is the machine's configuration before we press Start.

Then to define the action: suppose that at step t the machine \mathcal{P} is in configuration $\mathcal{C}(t) = \langle q, s, \tau_L, \tau_R \rangle$. To make the next transition, look for an instruction $q_p T_p T_n q_n \in \mathcal{P}$ with $q_p = q$ and $T_p = s$. The condition of determinism ensures that the set \mathcal{P} has at most one such instruction. If there is no such instruction then at step $t + 1$ the machine \mathcal{P} **halts**.

Otherwise, there are three possibilities. (1) If T_n is a symbol in the tape alphabet set Σ then the machine writes that symbol to the tape, so that the next

[†]How the device does this depends on its construction details. It could read and write marks on a paper tape, align magnetic particles on a plastic tape, twiddle bits on a solid state drive, or it could push LEGO bricks to the left or right side of a slot. Discreteness ensures that the machine can cleanly distinguish between the symbols, in contrast with the trouble that can happen, for instance, in reading an instrument dial near a boundary. [‡]We denote a Turing machine with a \mathcal{P} because although these machines are hardware, the things from everyday experience that they are most like are programs.

configuration is $\mathcal{C}(t+1) = \langle q_n, T_n, \tau_L, \tau_R \rangle$. (2) If $T_n = L$ then the machine moves the tape head to the left. So the next configuration is $\mathcal{C}(t+1) = \langle q_n, \hat{s}, \hat{\tau}_L, \hat{\tau}_R \rangle$ where $\hat{\tau}_R$ is the concatenation of the one-character string $\langle s \rangle$ with τ_R , where if $\tau_L = \epsilon$ then \hat{s} is the blank and $\hat{\tau}_L = \epsilon$, and otherwise where $\hat{s} = \tau_L[-1]$ and $\hat{\tau}_L = \tau_L[: -1]$. (3) If $T_n = R$ then the machine moves the tape head to the right. This is like (2) so we omit the details.

If two configurations are related by being a step apart then we write $\mathcal{C}(i) \vdash \mathcal{C}(i+1)$.[†] A **computation** is a sequence $\mathcal{C}(0) \vdash \mathcal{C}(1) \vdash \mathcal{C}(2) \vdash \dots$. We abbreviate a sequence of \vdash 's with \vdash^* .[‡] If the computation halts then the sequence has a final configuration $\mathcal{C}(h)$ so we could write a halting computation as $\mathcal{C}(0) \vdash^* \mathcal{C}(h)$.

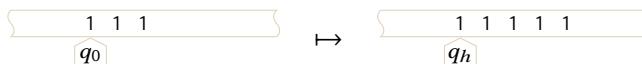
- 1.5 EXAMPLE In Example 1.1's table tracing the machine's steps, the graphics illustrate the successive configurations. Here is the same sequence as a computation.

$$\begin{aligned} & \langle q_0, 1, \epsilon, 11 \rangle \vdash \langle q_0, 1, 1, 1 \rangle \vdash \langle q_0, 1, 11, \epsilon \rangle \vdash \langle q_0, B, 111, \epsilon \rangle \vdash \langle q_1, 1, 11, \epsilon \rangle \\ & \vdash \langle q_1, B, 11, \epsilon \rangle \vdash \langle q_2, 1, 1, \epsilon \rangle \vdash \langle q_2, 1, \epsilon, 1 \rangle \vdash \langle q_2, B, \epsilon, 11 \rangle \vdash \langle q_3, 1, \epsilon, 1 \rangle \end{aligned}$$

Finally, as in that example, observe that our description of the action of a Turing machine emphasizes that it is a **state machine**—a computation is a sequence of discrete transitions.

Computable functions In this chapter's opening we expressed interest more in the things that the machines compute than in the machines themselves. We finish this section by defining the set of functions that are mechanically computable.

A function is an association of inputs with outputs. For Turing machines the natural association is to link the string on the tape when the machine starts with the one on the tape when it stops. The idea is to compute the value of a string to string function like $\phi(111) = 11111$ as here.



But there are a couple of things that the definition must take care with. First, a Turing machine may fail to halt on some input strings. Second, just specifying the input string is not enough since the initial position of the head can change the computation.

- 1.6 DEFINITION Let \mathcal{P} be a Turing machine with tape alphabet Σ . For input $\sigma \in \Sigma^*$, placing that on an otherwise blank tape and pointing \mathcal{P} 's read/write head to σ 's left-most symbol is **loading** that input. If we start \mathcal{P} with σ loaded and it eventually halts then we denote the associated output string as $\phi_{\mathcal{P}}(\sigma)$. If the machine never halts then σ has no associated output. The **function computed by the machine** \mathcal{P} is the set of associations $\sigma \mapsto \phi_{\mathcal{P}}(\sigma)$.

[†] Read ‘ \vdash ’ aloud as “yields.” [‡] Read this aloud as “yields eventually.”

- 1.7 **DEFINITION** For $\sigma \in \Sigma^*$, if the value of a Turing machine computation is not defined on σ then we say that the function computed by the machine **diverges** on that input, written $\phi_{\mathcal{P}}(\sigma)\uparrow$ (or $\phi_{\mathcal{P}}(\sigma) = \perp$). Otherwise we say that it **converges**, $\phi_{\mathcal{P}}(\sigma)\downarrow$.

Note the difference between the machine \mathcal{P} and the function computed by that machine, $\phi_{\mathcal{P}}$. For example, the machine $\mathcal{P}_{\text{pred}}$ is a set of four-tuples but the predecessor function is a set of input-output pairs, which we might denote $x \mapsto \text{pred}(x)$. Another example of the difference is that machines halt or fail to halt, while functions converge or diverge.

More points: (1) When there is only one machine under discussion then we write ϕ instead of $\phi_{\mathcal{P}}$. (2) In this book we like to write machines so that they also finish with the head under the first character of the output string, which isn't strictly necessary but is neater. (3) In other fields of mathematics a function comes with a domain, the set of inputs on which it is defined. In this field the convention is to write $\phi : \Sigma^* \rightarrow \Sigma^*$ and describe it as a **partial function**, where some $W \subseteq \Sigma^*$ is the set of input strings σ such that $\phi(\sigma)\downarrow$. If $W = \Sigma^*$ then ϕ is said to be a **total function**. (Every ϕ is partial but saying 'partial' usually connotes that the function is not total.)

There is one more point to raise about the definition. We will often consider a function that isn't an association of string input and output, and describe it as computed by a machine. For this we must impose an interpretation on the strings. For instance, with the predecessor machine in Example 1.1 we took the strings to represent natural numbers in unary. The same holds for computations with non-numbers, such as directed graphs, where we also just fix some encoding of the input and output strings. (We could worry that our interpretation might be so involved that, as with a horoscope, the work happens in the interpretation. But we will stick to cases such as the unary representation of numbers where this is not an issue.) Of course, the same thing happens on physical computers, where the machine twiddles bitstrings and then we interpret them as characters in a document or notes in a symphony, or however we like.

When we describe the function computed by a machine, we typically omit the part about interpreting the strings. We say, "this shows that $\phi(3) = 5$ " rather than, "this shows that ϕ takes a string representing 3 to a string representing 5." The details of the representation are usually not of interest in this chapter (in the fifth chapter we will count the time or space that they consume).

- 1.8 **REMARK** Early researchers, working before actual machines were widely available, needed airtight proofs that for instance there is a mechanical computation of the function that takes in a number and returns the exponent on 5 in its prime factorization. So they did the details, building up a large body of work which could be quite low level.

As an example of low-level detail, in the addition machine Example 1.2 we took the separator blank to be significant. Allowing significant blanks raises the issue of ambiguity: which of the blanks on the tape count as input and output

and which do not? We could handle this by adding a character to the alphabet to use exclusively as a begin/end marker. Or we could enforce that strings come in the form $\sigma = 1^n B \tau$ where the length of τ is n . Or we could code everything with integers, such as coding the triple $\langle 7, 8, 9 \rangle$ as $2^7 3^8 5^9$.

In this book we don't work through these details because that could hide the underlying ideas, and also simply to get to other material. Our everyday experience convinces us that machines can use their alphabet to reasonably represent anything computable. The next section says more.

- 1.9 **DEFINITION** A **computable function**, or **recursive function**,[†] is one computed by some Turing machine (it may be a total function or partial). A **computable set**, or **recursive set**, is one whose characteristic function is computable. A Turing machine **decides** a set if it computes the characteristic function of that set. A relation is computable if it is computable as a set.

We close with a summary. We have characterized mechanical computation. We view it as a process whereby a physical system evolves through a sequence of discrete steps that are local, meaning that all the action takes place within one cell of the head. This gives us a precise definition of which functions can be mechanically computed. The next subsection discusses why this characterization is widely accepted.

I.1 Exercises

Unless the exercise says otherwise, assume that $\Sigma = \{B, 1\}$. Also assume that any machine must start with its head under the leftmost input character and arrange for it to end with the head under the leftmost output character.

- 1.10 How is a Turing machine like a program? How is it unlike a program? How is it like the kind of computer we have on our desks? Unlike?
- 1.11 Why does the definition of a Turing machine, Definition 1.4, not include a definition of the tape?
- 1.12 Your study partner asks you, “The opening paragraphs talk about the *Entscheidungsproblem*, to mechanically determine whether a mathematical statement is true or false. I write programs with bits like `if (x>3)` all the time. What's the problem?” Help your friend out.
- ✓ 1.13 Trace each computation, as in Example 1.5. (A) The machine P_{pred} from Example 1.1 when starting on a tape with two 1's. (B) The machine P_{add} from Example 1.2 the addends are 2 and 2. (C) Give the two computations as configuration sequences, as on page 8.
- ✓ 1.14 For each of these false statements about Turing machines, briefly explain the fallacy. (A) Turing machines are not a complete model of computation because they can't do negative numbers. (B) The problem with Example 1.3 is

[†]The term ‘recursive’ used to be universal but is now old-fashioned.

that the instructions don't have any extra states where the machine goes to halt.
 (c) For a machine to reach state q_{50} it must run for at least fifty one steps.

1.15 We often have some states that are **halting states**, where we send the machine solely to make it halt. In this case the others are **working states**. For instance, Example 1.1 uses q_3 as a halting state and its working states are q_0 , q_1 , and q_2 . Name Example 1.2's halting and working states.

✓ 1.16 Trace the execution of Example 1.3's \mathcal{P}_{inf} loop for ten steps, from a blank tape.

1.17 Trace the execution on each input of this Turing machine with alphabet $\Sigma = \{\text{B}, \emptyset, 1\}$ for ten steps, or fewer if it halts.

$$\{ q_0 \text{B} \text{B} q_4, q_0 \emptyset \text{R} q_0, q_0 1 \text{R} q_1, q_1 \text{B} \text{B} q_4, q_1 \emptyset \text{R} q_2, q_1 1 \text{R} q_0, q_2 \text{B} \text{B} q_4, q_2 \emptyset \text{R} q_0, q_2 1 \text{R} q_3 \}$$

(A) 11 (B) 1011 (C) 110 (D) 1101 (E) ϵ

✓ 1.18 Give the transition table for the machine in the prior exercise.

✓ 1.19 Write a Turing machine that, if it is started with the tape blank except for a sequence of 1's, will replace those with a blank and then halt.

✓ 1.20 Produce Turing machines to perform these Boolean operations, using $\Sigma = \{\text{B}, \emptyset, 1\}$. (A) Take the 'not' of a bit $b \in \Sigma_0 = \Sigma - \{\text{B}\}$. That is, convert the input $b = \emptyset$ into the output 1, and convert 1 into \emptyset . (B) Take as input two characters drawn from Σ_0 and give as output the single character that is their logical 'and'. That is, if the input is 01 then the output should be \emptyset , while if the input is 11 then the output should be 1. (C) Do the same for 'or'.

1.21 Give a Turing machine that takes as input a bit string, using the alphabet $\{\text{B}, \emptyset, 1\}$, and adds 01 at the back.

1.22 Produce a Turing machine that computes the constant function $\phi(x) = 3$. It inputs a number written in unary, so that n is represented as n -many 1's, and outputs the number 3 in unary.

✓ 1.23 Produce a Turing machine that computes the successor function, that takes as input a number n and gives as output the number $n + 1$ (in unary).

✓ 1.24 Produce a doubler, a Turing machine that computes $f(x) = 2x$.

(A) Assume that the input and output is in unary. Hint: you can erase the first 1, move to the end of the 1's, past a blank, and put down two 1's. Then move left until you are at the start of the first sequence of 1's. Repeat.

(B) Instead assume that the alphabet is $\Sigma = \{\text{B}, \emptyset, 1\}$ and the input is represented in binary.

✓ 1.25 Produce a Turing machine that takes as input a number n written in unary, represented as n -many 1's, and if n is odd then it gives as output the number 1 in unary, with the head under that 1, while if n is even it gives the number 0 (which in a unary representation means the tape is blank).

1.26 Write a machine \mathcal{P} with tape alphabet Σ consisting of blank B, stroke 1, and the comma ',' character. Where $\Sigma_0 = \Sigma - \{\text{B}\}$, if we interpret the input $\sigma \in \Sigma_0$ as a

comma-separated list of natural numbers represented in unary, then this machine should return the sum, also in unary. Thus, $\phi_P(1111, 111, 1) = 11111111$.

1.27 Is there a Turing machine configuration without any predecessor? Restated, is there a configuration $C = \langle q, s, \tau_L, \tau_R \rangle$ for which there does not exist any configuration $\hat{C} = \langle \hat{q}, \hat{s}, \hat{\tau}_L, \hat{\tau}_R \rangle$ and instruction $I = \hat{q} \hat{s} T_n q_n$ such that if a machine is in configuration \hat{C} then instruction I applies and $\hat{C} \vdash C$?

1.28 One way to argue that Turing machines can do anything that a modern CPU can do involves showing how to do all of the CPU's operations on a Turing machine. For each, describe a Turing machine that will perform that operation. You need not produce the machine, just outline the steps. Use the alphabet $\Sigma = \{0, 1, B\}$. (A) Take as input a 4-bit string and do a bitwise NOT, so that each 0 becomes a 1 and each 1 becomes a 0. (B) Take as input a 4-bit string and do a bitwise circular left shift, so that from $b_3 b_2 b_1 b_0$ you end with $b_2 b_1 b_0 b_3$. (C) Take as input two 4-bit strings and perform a bitwise AND.

✓ 1.29 For each, produce a machine meeting the condition. (A) It halts on exactly one input. (B) It fails to halt on exactly one input. (C) It halts on infinitely many inputs and fails to halt on infinitely many.

1.30 A common alternative definition of Turing machine does not use what is on the tape when the machine halts. Rather, it designates one state as an **accepting state**. The language **decided** by the machine is the set of strings such that when the machine is started with such a string on the tape, and the machine halts, then when it halts it is in the accepting state. (There may also be a **rejecting state** and the machine must end in one or the other.) Write a Turing machine with alphabet $\{B, a, b\}$ that will halt in state q_3 if the input string contains two consecutive b's, and will halt in state q_4 otherwise.

Definition 1.9 says that a set is computable if there is a Turing machine that acts as its characteristic function. That is, the machine is started with the tape blank except for the input string σ , and with the head under the leftmost input character. This machine halts on all inputs, and when it halts, the tape is blank except for a single character, and the head points to that character. That character is either 1 (meaning that the string σ is in the set) or 0 (meaning it is not). For the next three exercises, produce a Turing machine that acts as the characteristic function of the set.

1.31 See the note above. Produce a Turing machine that acts as the characteristic function of the set, $\{\sigma \in \mathbb{B}^* \mid \sigma[0] = 0\}$, of bitstrings that start with 0.

1.32 Produce a Turing machine that acts as the characteristic function of the set $\{\sigma \in \mathbb{B}^* \mid \sigma[0:1] = 01\}$ of bitstrings that start with 01.

1.33 See the note before Exercise 1.31. Produce a Turing machine that acts as the characteristic function of the set of bitstrings that start with some number of 0's, including possibly zero-many of them, followed by a 1.

1.34 Definition 1.9 talks about computable relations. Consider the 'less than or equal' relation between two natural numbers. Produce a Turing machine with

$\Sigma = \{ 0, 1, B \}$ that takes in two numbers represented in unary and outputs $\tau = 1$ if the first number is less than or equal to the second, and $\tau = 0$ if not.

1.35 Write a Turing machine that decides if its input is a palindrome, a string that is the same backward as forward. Use $\Sigma = \{ B, 0, 1 \}$. Have the machine end with a single 1 on the tape if the input was a palindrome, and with a blank tape if not.

1.36 Turing machines tend to have many instructions and to be hard to understand. So rather than exhibit a machine, people often give an overview. Do that for a machine that replicates the input: if it is started with the tape blank except for a contiguous sequence of n -many 1's, then it will halt with the tape containing two sequences of n -many 1's separated by a single blank.

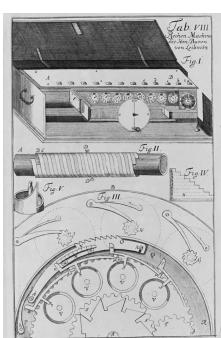
1.37 Show that if a Turing machine has the same configuration at two different steps then it will never halt. Is that sufficient condition also necessary?

1.38 Show that the steps in the execution of a Turing machine are not necessarily invertible. That is, produce a Turing machine and a configuration such that if you are told the machine was brought to that configuration after some number of steps, and you were asked what was the prior configuration, you couldn't tell.

SECTION

I.2 Church's Thesis

History Algorithms have always played a central role in mathematics. The simplest example is a formula such as the one giving the height of a ball dropped from the Leaning Tower of Pisa, $h(t) = -4.9t^2 + 56$. This is a kind of program: get the height output by squaring the time input, multiplying by -4.9 , and adding 56.



Leibniz's Stepped
Reckoner

In the 1670's the co-creator of Calculus, G Leibniz, constructed the first machine that could do addition, subtraction, multiplication, division, and square roots as well. This led him to speculate on the possibility of a machine that manipulates not just numbers but also symbols, and could thereby determine the truth of scientific statements. To settle any dispute, Leibniz wrote, scholars could say, "Let us calculate!" This is a version of the *Entscheidungsproblem*.

The real push to understand computation arose in 1927 from the Incompleteness Theorem of K Gödel. This says that for any (sufficiently powerful) axiom system there are statements that, while true in any model of the axioms, are not provable from those axioms. Gödel gave an algorithm that inputs the axioms and outputs the statement. This made evident the need to precisely define what is 'algorithmic' or 'effective'.

A number of mathematicians proposed formalizations. One was A Church,[†] who developed a system called the λ -calculus. Church and his students used it to

[†]After producing his machine model in 1935, Turing got a PhD in 1938 under Church at Princeton.

derive many intuitively computable functions such as number theoretic functions for divisibility and prime factorization. Church suggested to the most prominent expert in the area, Gödel, defining the set of effective functions as the set of functions that are λ -computable. But Gödel, who was notoriously careful, was unconvinced.

Everyone agreed that the doubler function $f(x) = 2x$ is effective; we can go from input to output in a way that is typographic, that pushes symbols. Church and his students had exhibited a wide class of functions that they argued are effective by proving that they are λ calculable. But the question is: where is the far end of this collection? Arguing that ‘derivable with the λ calculus’ implies effective, while persuasive, does not give the converse.

Everything changed when Gödel read Turing’s masterful analysis, outlined in the prior section. He wrote, “That this really is the correct definition of mechanical computability was established beyond any doubt by Turing.”

- 2.1 **CHURCH’S THESIS** The set of things that can be computed by a discrete and deterministic mechanism is the same as the set of things that can be computed by a Turing machine.[‡]

This is central to the Theory of Computation. It says that our technical results have a larger importance—they describe the devices that are on our desks and in our pockets.

Evidence We cannot give a mathematical proof of Church’s Thesis. The definition of a Turing machine, or of λ calculus or other equivalent schemes, formalizes ‘intuitively mechanically computable’. When a researcher consents to work within this formalization they are then free to reason about computation mathematically. So in a sense Church’s Thesis comes before the mathematics, or at any rate sits outside its usual derivation and verification work. Turing wrote, “All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically.”



Kurt Gödel
1906–1978



Alonzo
Church
1903–1995

Despite not being the conclusion of a deductive system, Church’s Thesis is generally accepted. We will give four points in its favor that persuaded Gödel, Church, and others at the time, and that still persuade researchers today: coverage, convergence, consistency, and clarity.

First, coverage. Everything that is intuitively computable has proven to be computable by a Turing machine. This includes not just the number theoretic functions investigated by researchers in the 1930’s but also everything ever computed by every program written for every existing computer, because all of them can be compiled to run on a Turing machine.

Despite this weight of evidence, the argument by coverage would collapse if someone exhibited even one counterexample, one operation that can be done in

[‡]This is often called the Church-Turing Thesis. Here we figure that because Turing has the machine, we can give Church nominal possession of the thesis.

finite time on an in-principle-physically-realizable discrete and deterministic device but that cannot be done on a Turing machine. So this argument is strong but at least conceivably not decisive.

The second argument is convergence: in addition to Turing and Church, many other researchers then and since have proposed models of computation. For instance, the next section defining general recursive functions will give us a taste of another influential model. However, despite a wide variation in approaches, all of the models yield the same set of computable functions. For instance, Turing showed that the set of functions computable with his machine model is equal to the set of functions computable with Church's λ -calculus.

Now, everyone could be wrong. There could be some systematic error in thinking around this point. For centuries, geometers seemed unable to imagine the possibility of Euclid's Parallel Postulate not holding and perhaps a similar cultural blindness is happening here. Nonetheless, if a number of very smart people go off and work independently on a question and when they come back you find that they have taken many different approaches but they all got the same answer, then you could think that it may be the right answer. At the least, convergence says that there is something natural and compelling about this set of functions.

The third argument was not completely available to researchers in the 1930's because it depends to some extent on work done since. It is consistency, that the details of the definition of a Turing machine are not essential to what it can compute. An example is that we can show that one-tape machines can compute all of the functions that can be computed by machines with two or more tapes. Thus the fact that Definition 1.4's machines have just one tape is not an essential point.

Similarly, machines whose tape is unbounded in only one direction can compute all the functions that are computable with a tape unbounded in both directions. And machines with more than one read/write head compute the same functions as those with only one. As to symbols, we need the blank for all but finitely-many cells on the starting tape and we need at least one more symbol to make marks distinguishable from the blank, but with only the symbols in $\Sigma = \{B, 1\}$ we can compute any Turing machine computable function. Likewise, write-once machines that cannot change any tape square after writing to it compute the same set of functions. And although restricting to machines having only one state is not enough, machines with two states are equipowerful with Definition 1.4's machines having arbitrarily-many states.

There is one more condition that does not change the set of computable functions, determinism. Recall that the definition of Turing machine given above does not allow, say, both of the instructions $q_5 1 R q_6$ and $q_5 1 L q_4$ in the same machine, because they both begin with $q_5 1$. If we drop this restriction then the machines that we get are called nondeterministic. We will have much more to say on this in the fifth chapter but the collection of nondeterministic Turing machines computes the same set of functions as does the collection of deterministic machines.

Thus, for any way in which the Turing machine definition seems to make an

arbitrary choice, making a different choice leads to the same set of computable functions. This is persuasive in that any proper definition of what is computable should possess this property. For instance, if two-tape machines computed more functions than one-tape machines and three-tape machines more than that, then identifying the set of computable functions with those computable by single-tape machines would be foolish. But as with the coverage and convergence arguments, while this means that the class of Turing machine-computable functions is natural and wide-ranging, it still leaves open a small crack of a possibility that the class does not exhaust the list of functions that are mechanically computable.

The most persuasive single argument for Church's Thesis — what caused Gödel to change his mind and what still convinces scholars today — is clarity: Turing's analysis is compelling. Gödel noted this in the quote given earlier and Church felt the same way, writing that Turing machines have, “the advantage of making the identification with effectiveness . . . evident immediately.”

What it does not say Church's Thesis does not say that in all circumstances the best way to understand a discrete and deterministic computation is via the Turing machine model. For example, a numerical analyst studying the performance of a floating point algorithm should use a computer model that has registers. Church's Thesis says that the calculation could in principle be done by a Turing machine but for this use registers are better because the researcher wants results that apply to in-practice machines.[†]

Church's Thesis also does not say that Turing machines are all there is to any computation in the sense that if, say, you are working on an automobile antilock braking system then while the Turing machine model can account for the logical and arithmetic computations, it cannot do the entire system including sensor inputs and actuator outputs. S Aaronson has made this point, “Suppose I . . . [argued] that . . . [Church's] Thesis fails to capture all of computation, because Turing machines can't toast bread. . . . No one ever claimed that a Turing machine could handle every possible interaction with the external world, without first hooking it up to suitable peripherals. If you want a Turing machine to toast bread, you need to connect it to a toaster; then the [Turing machine] can easily handle the toaster's internal logic.”

In the same vein, we can get physical devices that supply a stream of random bits. These are not pseudorandom bits that are computed by a method that is deterministic; instead, well-established physics says these are truly random. Turing machines are not lacking because they cannot produce the bits. Rather, Church's Thesis asserts that we can use Turing machines to model the discrete and deterministic computations that we can do after we get the bits.

An empirical question? This discussion raises a big question: even if we accept Church's Thesis, can we do more by going beyond discrete and deterministic?

[†] Scientists who study the brain also find Turing machines to be not the most suitable model. Note however that saying that another model is a better fit is different than saying that there are brain operations that could not in principle be done using a Turing machine as a substrate.

Would analog methods such as passing lasers through a gas or some kind of subatomic magic allow us to compute things that no Turing machine can compute? Or are Turing machines an ultimate in physically-possible machines? Did Turing, on that day, lying on that grassy river bank after his run, intuit everything that experiments with reality would ever find to be possible?

For a sense of the conversation, we know that the wave equation[†] can have computable initial conditions (for these real numbers x , there is a program that inputs $i \in \mathbb{N}$ and outputs x 's i -th decimal place) but the solution is not computable. So does the wave tank modeled by this equation compute something that Turing machines cannot? Stated for rhetorical effect, do the planets in their orbits compute an exact solution to the Three-Body Problem but our machines fail at it?

In this case we can object that an experimental apparatus can have noise and measurement problems, including a finite number of decimal places in the instruments, etc. But even if careful analysis of the physics of a wave tank leads us to discount it as a reliable computer of a function, we can still wonder whether there might be another apparatus that would work.

This big question remains open. No one has produced a generally accepted example of a non-discrete mechanism that computes a function that no Turing machine computes. However, there is also not yet an analysis of physically-possible mechanical computation in the non-discrete case which has the support enjoyed by Turing's analysis in its more narrow domain.

We will not pursue this further, instead only observing that the community of researchers has weighed in by taking Church's Thesis as the basis for its work. For us, 'computation' will refer to the kind of work that Turing analyzed. That's because we are interested in thinking about symbol-pushing, not toasting bread.

Using Church's Thesis Church's Thesis asserts that the three models of computation: Turing machines, λ calculus, the general recursive functions that we will see in the next section, and others that we won't describe, are maximally capable. By that we mean that these models all compute the same things — the set of functions that each model computes equals the set of functions that we have named earlier as the set of computable functions. So we can fix one of these models as our preferred formalization and get on with the analysis. Here we choose Turing machines.

One reason that we emphasize Church's Thesis is that it imbues our results with a larger importance. When for instance we will later describe a function that no Turing machine can compute then, with the thesis in mind, we will interpret the technical statement to mean that this function cannot be computed by *any* discrete and deterministic device.

But there is one more thing that we will do with Church's Thesis. We will leverage it to make life easier. As the exercises above illustrate, while writing a few Turing machines gives some insight, after a while you find that doing more machines does not give more illumination. Worse, focusing too much on machine details risks obscuring larger points. So if we can be clear and rigorous without

[†]A partial differential equation that describes the propagation of waves.

actually having to handle a mass of detail then we will be delighted.

Church's Thesis helps with this. Often when we want to show that something is computable, we will first argue that it is intuitively computable and then invoke Church's Thesis to assert that it is therefore Turing machine computable. With that we will proceed, “Let \mathcal{P} be that machine …” without ever having to exhibit a set of four-tuple instructions. The justification is that our intuition about what is computable—our sense of what can be done on a discrete and deterministic device—has developed through great experience using general purpose programming languages. Certainly there is a danger that we will get ‘intuitively computable’ wrong but we have so much practice that the danger is relatively minimal. The upside is that we can make much more rapid progress through the material.

To assert that something is intuitively computable we will sometimes produce programming language code. For this our language is a Scheme, specifically, Racket.

I.2 Exercises

2.2 Why is it Church's Thesis instead of Church's Theorem?

- ✓ 2.3 We've said that the thing from our everyday experience that Turing Machines are most like is programs. What is the difference between: (A) a Turing Machine and an algorithm? (B) a Turing Machine and a computer? (C) a program and a computer? (D) a Turing Machine and a program?
- 2.4 Your study partner is struggling with a point. “I don't get the excitement about computing with a mechanism. I mean, the Stepped Reckoner is like an old-timey calculator device: they can do some very limited computations, with numbers only. But I'm interested in a modern computer that is vastly more flexible in that it can also work with strings, for instance. I mean, a slide rule is not programmable, is it?” Help them understand.
- ✓ 2.5 Each of these is often given as a counterargument to Church's Thesis. Explain why each is mistaken. (A) Turing machines have an infinite tape so it is not a realistic model. (B) The universe is finite so there are only finitely many configurations possible for any computing device, whereas a Turing machine has infinitely many configurations, so it is not realistic.
- ✓ 2.6 One of these is a correct statement of Church's Thesis and the others are not. Which one is right? (A) Anything that can be computed by any mechanism can be computed by a Turing machine. (B) No human computer, or machine that mimics a human computer, can out-compute a Turing machine. (C) The set of things that are computable by a discrete and deterministic mechanism is the same as the set of things that are computable by a Turing machine. (D) Every product of a person's mind, or product of a mechanism that mimics the activity of a person's mind, can be produced by some Turing machine.
- 2.7 List two benefits from adopting Church's Thesis.

- ✓ 2.8 Refute this objection to Church's Thesis: "Some computations, such as an operating system, are designed to never halt. The Turing machine is an inadequate model for these."
- 2.9 The idea of 'intuitively computable' certainly has subtleties. Let $f, g: \mathbb{N} \rightarrow \mathbb{N}$.
 - (A) If both are intuitively computable then is $f \circ g$ also intuitively computable?
 - (B) What if g is computable but f is not?
- 2.10 The computers that we use are binary. Argue that if they were ternary, where instead of bits with two values they used trits with three, then they would compute exactly the same set of functions.
- 2.11 Use Church's thesis to argue that the indicated function exists and is computable.
 - (A) Suppose that $f_0, f_1: \mathbb{N} \rightarrow \mathbb{N}$ are computable partial functions. Show that $h: \mathbb{N} \rightarrow \mathbb{N}$ is a computable partial function where $h(x) = 1$ if x is in the intersection of the domain of f_0 and the domain of f_1 , and $h(x) \uparrow$ otherwise.
 - (B) Do the same as in the prior item, but take the union of the two domains.
 - (C) Suppose that $f: \mathbb{N} \rightarrow \mathbb{N}$ is a computable function that is total. Show that $h: \mathbb{N} \rightarrow \mathbb{N}$ is a computable partial function, where $h(x) = 1$ if x is in the range of f and $h(x) \uparrow$ otherwise.
 - (D) Suppose $f_0, f_1: \mathbb{N} \rightarrow \mathbb{N}$ are computable total functions. Show that their composition $h = f_1 \circ f_0$ is a computable function $h: \mathbb{N} \rightarrow \mathbb{N}$.
 - (E) Suppose $f_0, f_1: \mathbb{N} \rightarrow \mathbb{N}$ are computable partial functions. Show that their composition is a computable partial function $f_1 \circ f_0: \mathbb{N} \rightarrow \mathbb{N}$.
- ✓ 2.12 Suppose that $f: \mathbb{N} \rightarrow \mathbb{N}$ is a total computable function. Argue that this function is computable: $h(n) = 0$ if n is in the range of f , and $h(n) \uparrow$ otherwise.
- 2.13 Let $f, g: \mathbb{N} \rightarrow \mathbb{N}$ be computable functions that may be either total or partial functions. Use Church's Thesis to argue that this function is computable: $h(n) = 1$ if both $f(n) \downarrow$ and $g(n) \downarrow$, and $h(n) \uparrow$ otherwise.
- 2.14 Arguing by Church's Thesis relies on our having a solid intuition about what is implementable on a device. The following is not implementable; what goes wrong? "Given a polynomial $p(x_0, \dots, x_n)$, we can determine whether or not it has natural number roots by trying all possible settings of the input (x_0, \dots, x_n) to $n+1$ -tuples of integers."
- ✓ 2.15 If you allow processes to take infinitely many steps then you can have all kinds of fun. Suppose that you have infinitely many dollars. You run into the Devil. He proposes an infinite sequence of transactions, in each of which he will hand you two dollars and take from you one dollar. (The first will take $1/2$ hour, the second $1/4$ hour, etc.) You figure you can't lose. But he proves to be particular about the order in which you exchange bills. First he numbers your bills as 1, 3, 5, ... At each step he buys your lowest-numbered bill and pays you with two higher-numbered bills. Thus, he first accepts from you bill number 1 and pays you with his own bills, numbered 2 and 4. Next he buys from you bill number 2 and pays you with his bills numbered 6 and 8. How much do you end with?

The next two exercises involve multitape machines. Definition 1.4's transition function for a single tape is $\Delta: Q \times \Sigma \rightarrow (\Sigma \cup \{\text{L}, \text{R}\}) \times Q$. Define a machine with k -many

tapes by extending to Δ : $Q \times \Sigma^k \rightarrow (\Sigma \cup \{L, R\})^k \times Q$. Thus, a typical four-tuple for a $k = 2$ tape machine with alphabet $\Sigma = \{0, 1, B\}$ is $q_4 \langle 1, B \rangle \langle 0, L \rangle q_3$. It means that if the machine is in state q_4 and the head on tape 0 is reading 1 while that on tape 1 is reading a blank, then the machine writes 0 to tape 0, moves left on tape 1, and goes into state q_3 .

2.16 Write the transition table of a two-tape machine to complement a bitstring. The machine has alphabet $\{0, 1, B\}$. It starts with a string σ of 0's and 1's on tape 0 (the tape 0 head starts under the leftmost bit) and tape 1 is blank. When it finishes, on tape 1 is the complement of σ , with input 0's changed to 1's and input 1's changed to 0's, and with the tape 1 head under the leftmost bit.

2.17 Write a two-tape Turing machine to take the logical and of two bitstrings. The machine starts with two same-length strings of 0's and 1's on the two tapes. The tape 0 head starts under the leftmost bit, as does the tape 1 head. When the machine halts, the tape 1 head is under the leftmost bit of the result (we don't care about the tape 0 head).

SECTION

I.3 Recursion

We will outline an approach to defining computability that is different than Turing's, both to give a sense of another approach and because it is useful.[†] It lists some initial functions that are intuitively computable and then describes ways to combine existing functions to make new ones, where if the existing ones are intuitively computable then so is the new one. An example of an intuitively computable initial function is successor $S: \mathbb{N} \rightarrow \mathbb{N}$, described by $S(x) = x + 1$, and a combiner that preserves effectiveness is function composition. Thus the composition $S \circ S$, the plus-two operation, is also intuitively mechanically computable.

We now introduce another effectiveness-preserving combiner.

Primitive recursion Grade school students learn addition and multiplication as mildly complicated algorithms (multiplication, for example, involves arranging the digits into a table, doing partial products from right to left, and then adding). In 1861, H Grassmann produced a more elegant definition. Here is the formula for addition, plus: $\mathbb{N}^2 \rightarrow \mathbb{N}$, which takes as given the successor map, $S(n) = n + 1$.

$$\text{plus}(x, y) = \begin{cases} x & - \text{if } y = 0 \\ S(\text{plus}(x, z)) & - \text{if } y = S(z) \text{ for } z \in \mathbb{N} \end{cases}$$



Hermann
Grassmann
1809-1877

[†]One advantage of this approach is that it does not need the codings discussed for Turing machines as it works directly with the functions.

- 3.1 EXAMPLE This finds the sum of 3 and 2.

$$\text{plus}(3, 2) = \mathcal{S}(\text{plus}(3, 1)) = \mathcal{S}(\mathcal{S}(\text{plus}(3, 0))) = \mathcal{S}(\mathcal{S}(3)) = 5$$

Besides being compact, this has a very interesting feature: ‘plus’ recurs in its own definition.[†] This is definition by **recursion**. Whereas the grade school definition of addition is prescriptive in that it gives a procedure, this recursive definition has the advantage of being descriptive because it specifies the meaning, the semantics, of the operation.

A common reaction on first seeing recursion is to wonder whether it is logically problematic—isn’t it a fallacy to define something in terms of itself? The expansion above exposes that this reaction is confused, since $\text{plus}(3, 2)$ is not defined in terms of itself, it is defined in terms of $\text{plus}(3, 1)$ (and the successor function). Similarly, $\text{plus}(3, 1)$ is defined in terms of $\text{plus}(3, 0)$. And clearly the definition of $\text{plus}(3, 0)$ is not a problem. The key here is to define the function on higher-numbered inputs using only its values on lower-numbered ones.[‡]

A marvelous feature of Grassmann’s approach is that it extends naturally to other operations. Multiplication has the same form.

$$\text{product}(x, y) = \begin{cases} 0 & - \text{if } y = 0 \\ \text{plus}(\text{product}(x, z), x) & - \text{if } y = \mathcal{S}(z) \end{cases}$$

- 3.2 EXAMPLE The expansion of $\text{product}(2, 3)$ reduces to a sum of three 2’s.

$$\begin{aligned} \text{product}(2, 3) &= \text{plus}(\text{product}(2, 2), 2) \\ &= \text{plus}(\text{plus}(\text{product}(2, 1), 2), 2) \\ &= \text{plus}(\text{plus}(\text{plus}(\text{product}(2, 0), 2), 2), 2) \\ &= \text{plus}(\text{plus}(\text{plus}(0, 2), 2), 2) \end{aligned}$$

And exponentiation works the same way.

$$\text{power}(x, y) = \begin{cases} 1 & - \text{if } y = 0 \\ \text{product}(\text{power}(x, z), x) & - \text{if } y = \mathcal{S}(z) \end{cases}$$

Our interest in Grassmann’s definition is that it is effective—it translates immediately into a program. Starting with a successor operation,

```
(define (successor x)
  (+ x 1))
```

this code implements the definition given above of plus.[#]

[†]That is, this is a discrete form of feedback. [‡]So the idea behind this recursion is that addition of larger numbers reduces to addition of smaller ones. [#]Obviously Racket, like every general purpose programming language, comes with a built in addition operator, as in `(+ 3 2)`, along with a multiplication operator, as in `(* 3 2)`, and with other common arithmetic operators.

```
(define (plus x y)
  (let ((z (- y 1)))
    (if (= y 0)
        x
        (successor (plus x z)))))
```

(The `(let ...)` creates the local variable `z`.) The same is true for product and power.

```
(define (product x y)
  (let ((z (- y 1)))
    (if (= y 0)
        0
        (plus (product x z) x))))
```

```
(define (power x y)
  (let ((z (- y 1)))
    (if (= y 0)
        1
        (product (power x z) x))))
```

- 3.3 **DEFINITION** A function f is defined by the schema[‡] of **primitive recursion** from the functions g and h when this holds.

$$f(x_0, \dots, x_{k-1}, y) = \begin{cases} g(x_0, \dots, x_{k-1}) & - \text{if } y = 0 \\ h(f(x_0, \dots, x_{k-1}, z), x_0, \dots, x_{k-1}, z) & - \text{if } y = S(z) \end{cases}$$

The bookkeeping is that the arity of f , the number of inputs, is one more than the arity of g and one less than the arity of h .

- 3.4 **EXAMPLE** The function `plus` is defined by primitive recursion from $g(x_0) = x_0$ and $h(w, x_0, z) = S(w)$. The function `product` is defined by primitive recursion from $g(x_0) = 0$ and $h(w, x_0, z) = \text{plus}(w, x_0)$. The function `power` is defined by primitive recursion from $g(x_0) = 1$ and $h(w, x_0, z) = \text{product}(w, x_0)$.

The computer code above makes evident that primitive recursion fits into the plan of specifying combiners that preserve the property of effectiveness: if g and h are intuitively computable then so is f .

Primitive recursion, along with function composition, suffices to define many familiar functions.

- 3.5 **EXAMPLE** The predecessor function is like an inverse to successor. However, since we use the natural numbers we can't give a predecessor of zero, so instead we describe `pred`: $\mathbb{N} \rightarrow \mathbb{N}$ by: $\text{pred}(y)$ equals $y - 1$ if $y > 0$ and equals 0 if $y = 0$. This definition fits the primitive recursive schema.

$$\text{pred}(y) = \begin{cases} 0 & - \text{if } y = 0 \\ z & - \text{if } y = S(z) \end{cases}$$

The arity bookkeeping is that `pred` has no x_i 's so g is a function of zero-many inputs and is therefore constant, $g() = 0$, while h has two inputs, $h(a, b) = b$ (the

[‡]A ‘schema’ is an underlying organizational pattern or structure.

first input gets ignored, that is, a is just a dummy variable).

- 3.6 EXAMPLE Consider **proper subtraction**, denoted $x \dot{-} y$, described by: if $x \geq y$ then $x \dot{-} y$ equals $x - y$ and otherwise $x \dot{-} y$ equals 0. This definition of that function fits the primitive recursion schema.

$$\text{propersub}(x, y) = \begin{cases} x & \text{if } y = 0 \\ \text{pred}(\text{propersub}(x, z)) & \text{if } y = S(z) \end{cases}$$

In the terms of Definition 3.3, $g(x_0) = x_0$ and $h(w, x_0, z) = \text{pred}(w)$; the book-keeping works since the arity of g is one less than the arity of f , and, because h has dummy arguments, its arity is one more than the arity of f .

We sometimes abbreviate a sequence of inputs x_0, \dots, x_{k-1} as \vec{x} .

- 3.7 DEFINITION The set of **primitive recursive functions** consists of those that can be derived from the initial operations of the **zero** function $\mathcal{Z}(\vec{x}) = \mathcal{Z}(x_0, \dots, x_{k-1}) = 0$,[†] the **successor** function $S(\vec{x}) = x + 1$, and the **projection** functions $\mathcal{I}_i(\vec{x}) = x_i$ where $i \in \{0, \dots, k-1\}$, by a finite number of applications of the combining operations of function composition and primitive recursion.

Function composition covers not just the simple case of two functions f and g whose composition is defined by $f \circ g(\vec{x}) = f(g(\vec{x}))$. It also covers simultaneous substitution, where from $f(x_0, \dots, x_n)$ and $h_0(y_{0,0}, \dots, y_{0,m_0}), \dots$, and $h_n(y_{n,0}, \dots, y_{n,m_n})$, we get $f(h_0(y_{0,0}, \dots, y_{0,m_0}), \dots, h_n(y_{n,0}, \dots, y_{n,m_n}))$, which is a function with $(m_0 + 1) + \dots + (m_n + 1)$ -many inputs.

Besides the ones given above, many other everyday mathematical operations are primitive recursive. They include testing whether one number is less than another, finding remainders, and, given a number and a prime, finding the largest power of that prime that divides the number. See the exercises.

The list is so extensive that we may wonder whether every mechanically computed function is primitive recursive. The next section shows that the answer is no — although primitive recursion is very powerful, nonetheless there are intuitively mechanically computable functions that are not primitive recursive.

I.3 Exercises

- ✓ 3.8 What is the difference between primitive recursion and primitive recursive?
- 3.9 What is the difference between total recursive and primitive recursive?
- 3.10 In defining 0^0 there is a conflict between the desire to have that every power of 0 is 0 and the desire to have that every number to the 0 power is 1. What does the definition of power given above do?

[†]There are actually infinitely many zero functions, one for each natural number k , despite that we called it “the” zero function. The same holds for the other initial functions. Also, projection is a generalization of the identity function, which is why we use the letter \mathcal{I} .

- ✓ 3.11 As the section body describes, recursion doesn't have to be logically problematic. But some recursions are; consider this one.

$$f(n) = \begin{cases} 0 & - \text{if } n = 0 \\ f(2n - 2) & - \text{otherwise} \end{cases}$$

(A) Find $f(0)$ and $f(1)$. (B) Try to find $f(2)$.

- 3.12 Consider this function.

$$F(y) = \begin{cases} 42 & - \text{if } y = 0 \\ F(y - 1) & - \text{otherwise} \end{cases}$$

(A) Find $F(0), \dots, F(10)$.

(B) Show that F is primitive recursive by describing it in the form given in Definition 3.3, giving suitable functions g and h (*Hint*: g is a function of no arguments, a constant). You can use functions already defined in this section.

- 3.13 The function `plus_two`: $\mathbb{N} \rightarrow \mathbb{N}$ adds two to its input. Show that it is a primitive recursive function.

- 3.14 The Boolean function `is_zero` inputs natural numbers and return T if the input is zero, and F otherwise. Give a definition by primitive recursion, representing T with 1 and F with 0. *Hint*: you only need a zero function, successor, and the schema of primitive recursion.

- ✓ 3.15 These are the **triangular numbers** because if you make a square that has n dots on a side and divide it down the diagonal, including the diagonal, then the triangle that you get has $t(n)$ dots.

$$t(y) = \begin{cases} 0 & - \text{if } y = 0 \\ y + t(y - 1) & - \text{otherwise} \end{cases}$$

(A) Find $t(0), \dots, t(10)$.

(B) Show that t is primitive recursive by describing it in the form given in Definition 3.3, giving suitable functions g and h (*Hint*: g is a function of no arguments, a constant). You can use functions already defined in this section.

- ✓ 3.16 This is the first sequence of numbers ever computed on an electronic computer.

$$s(y) = \begin{cases} 0 & - \text{if } y = 0 \\ s(y - 1) + 2y - 1 & - \text{otherwise} \end{cases}$$

(A) Find $s(0), \dots, s(10)$.

(B) Verify that t is primitive recursive by putting it in the form given in Definition 3.3, giving suitable functions g and h (*Hint*: g is a function of no arguments, a constant). You can use functions already defined in this section.

3.17 Consider this recurrence.

$$d(y) = \begin{cases} 0 & - \text{if } y = 0 \\ s(y-1) + 3y^2 + 3y + 1 & - \text{otherwise} \end{cases}$$

- (A) Find $d(0), \dots, d(5)$.
- (B) Verify that d is primitive recursive by putting it in the form given in Definition 3.3, giving suitable functions g and h (*Hint*: g is a function of no arguments, a constant). You can use functions already defined in this section.
- ✓ 3.18 The Towers of Hanoi is a famous puzzle: *In the great temple at Benares ... beneath the dome which marks the center of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four discs of pure gold, the largest disc resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Brahma. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Brahma, which require that the priest on duty must not move more than one disc at a time and that he must place this disc on a needle so that there is no smaller disc below it. When the sixty-four discs shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple, and Brahmans alike will crumble into dust, and with a thunderclap the world will vanish.* It gives the recurrence below because to move a pile of discs you first move to one side all but the bottom, which takes $H(n-1)$ steps, then move that bottom one, which takes one step, then re-move the other disks into place on top of it, taking another $H(n-1)$ steps.

$$H(n) = \begin{cases} 1 & - \text{if } n = 1 \\ 2 \cdot H(n-1) + 1 & - \text{if } n > 0 \end{cases}$$

- (A) Compute the values for $n = 1, \dots, 10$.
- (B) Verify that H is primitive recursive by putting it in the form given in Definition 3.3, giving suitable functions g and h (*Hint*: g is a function of no arguments, a constant). You can use functions already defined in this section.
- 3.19 Define the factorial function $\text{fact}(y) = y \cdot (y-1) \cdots 1$ by primitive recursion, using product and a constant function.
- ✓ 3.20 Recall that the greatest common divisor of two positive integers is the largest integer that divides both. We can compute the greatest common divisor using Euclid's recursion

$$\gcd(n, m) = \begin{cases} n & - \text{if } m = 0 \\ \gcd(m, \text{rem}(n, m)) & - \text{if } m > 0 \end{cases}$$

where $\text{rem}(a, b)$ is the remainder when a is divided by b . Note that this fits the schema of primitive recursion. Use Euclid's method to compute these.
 (A) $\gcd(28, 12)$ (B) $\gcd(104, 20)$ (C) $\gcd(309, 25)$

3.21 As in the prior exercise, recall that the greatest common divisor of two positive integers is the largest integer that divides both. These properties are clear: $\gcd(a, 1) = 1$, $\gcd(a, a) = a$, $\gcd(a, b) = \gcd(b, a)$, and $\gcd(a + b, b) = \gcd(a, b)$. From them produce a recursion and use it to compute these. (A) $\gcd(28, 12)$ (B) $\gcd(104, 20)$ (C) $\gcd(309, 25)$

The following four exercises list functions and predicates. (A predicate is a truth-valued function; we take an output of 1 to mean ‘true’ while 0 is ‘false’.) Show that each is primitive recursive. For each, you may use functions already shown to be primitive recursive in this section body, or in a prior exercise item or subitem.

✓ 3.22 See the note above.

- (A) Constant function: for $k \in \mathbb{N}$, $C_k(\vec{x}) = C_k(x_0, \dots, x_{n-1}) = k$. Hint: instead of doing the general case with k , try $C_4(x_0, x_1)$, the function that returns 4 for all input pairs. Also, for this you need only use the zero function, successor, and function composition.
- (B) Maximum and minimum of two numbers: $\max(x, y)$ and $\min(x, y)$. Hint: use addition and proper subtraction.
- (C) Absolute difference function: $\text{absdiff}(x, y) = |x - y|$.

3.23 See the note before Exercise 3.22.

- (A) Sign predicate: $\text{sign}(y)$, which gives 1 if y is greater than zero and 0 otherwise.
- (B) Negation of the sign predicate: $\text{negsign}(y)$, which gives 0 if y is greater than zero and 1 otherwise.
- (C) Less-than predicate: $\text{lessthan}(x, y) = 1$ if x is less than y , and 0 otherwise. (The greater-than predicate is similar.)

✓ 3.24 See the note before Exercise 3.22.

- (A) Boolean functions: where x, y are inputs with values 0 or 1 there is the standard one-input function

$$\text{not}(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}$$

and two-input functions.

$$\text{and}(x, y) = \begin{cases} 1 & \text{if } x = y = 1 \\ 0 & \text{otherwise} \end{cases} \quad \text{or}(x, y) = \begin{cases} 0 & \text{if } x = y = 0 \\ 1 & \text{otherwise} \end{cases}$$

- (B) Equality predicate: $\text{equal}(x, y) = 1$ if $x = y$ and 0 otherwise.

✓ 3.25 See the note before Exercise 3.22.

- (A) Inequality predicate: $\text{notequal}(x, y) = 0$ if $x = y$ and 1 otherwise.
- (B) Functions defined by a finite and fixed number of cases, as with these.

$$m(x) = \begin{cases} 7 & \text{if } x = 1 \\ 9 & \text{if } x = 5 \\ 0 & \text{otherwise} \end{cases} \quad n(x, y) = \begin{cases} 7 & \text{if } x = 1 \text{ and } y = 2 \\ 9 & \text{if } x = 5 \text{ and } y = 5 \\ 0 & \text{otherwise} \end{cases}$$

3.26 Show that each of these is primitive recursive. You may use any function shown to be primitive recursive in the section body, in the prior exercise, or in a prior item.

- (A) Bounded sum function: the partial sums of a series whose terms $g(i)$ are given by a primitive recursive function, $S_g(y) = \sum_{0 \leq i < y} g(i) = g(0) + g(1) + \dots + g(y-1)$ (the sum of zero-many terms is $S_g(0) = 0$). Contrast this with the final item of the prior question; here the number of summands is finite but not fixed.
- (B) Bounded product function: the partial products of a series whose terms $g(i)$ are given by a primitive recursive function, $P_g(y) = \prod_{0 \leq i < y} g(i) = g(0) \cdot g(1) \cdots g(y-1)$ (the product of zero-many terms is $P_g(0) = 1$).
- (C) Bounded minimization: let $m \in \mathbb{N}$ and let $p(\vec{x}, i)$ be a predicate. Then the minimization operator $M(\vec{x}, i)$, typically written $\mu^m i[p(\vec{x}, i)]$, returns the smallest $i \leq m$ such that $p(\vec{x}, i) = 0$, or else returns m . Hint: Consider the bounded sum of the bounded products of the predicates.

3.27 Show that each is a primitive recursive function. You can use functions from this section or functions from the prior exercises.

- (A) Bounded universal quantification: suppose that $m \in \mathbb{N}$ and that $p(\vec{x}, i)$ is a predicate. Then $U(\vec{x}, m)$, typically written $\forall_{i \leq m} p(\vec{x}, i)$, has value 1 if $p(\vec{x}, 0) = 1, \dots, p(\vec{x}, m) = 1$ and value 0 otherwise. (The point of writing the functional expression $U(\vec{x}, m)$ is to emphasize the required uniformity. Stating one formula for the $m = 1$ case, $p(\vec{x}, 0) \cdot p(\vec{x}, 1)$, and another for the $m = 2$ case, $p(\vec{x}, 0) \cdot p(\vec{x}, 1) \cdot p(\vec{x}, 2)$, etc., is the best we can do. We can get a single derivation, that follows the rules in Definition 3.7, and that works for all m .)
 - (B) Bounded existential quantification: let $m \in \mathbb{N}$ and let $p(\vec{x}, i)$ be a predicate. Then $A(\vec{x}, m)$, typically written $\exists_{i \leq m} p(\vec{x}, i)$, has value 1 if $p(\vec{x}, 0) = 0, \dots, p(\vec{x}, m) = 0$ is not true, and has value 0 otherwise.
 - (C) Divides predicate: where $x, y \in \mathbb{N}$ we have $\text{divides}(x, y)$ if there is some $k \in \mathbb{N}$ with $y = x \cdot k$.
 - (D) Primality predicate: $\text{prime}(y)$ if y has no nontrivial divisor.
- ✓ 3.28 We will show that the function $\text{rem}(a, b)$ giving the remainder when a is divided by b is primitive recursive.

- (A) Fill in this table.

a	0	1	2	3	4	5	6	7
$\text{rem}(a, 3)$								

- (B) Observe that $\text{rem}(a+1, 3) = \text{rem}(a) + 1$ for many of the entries. When is this relationship not true?
- (C) Fill in the blanks.

$$\text{rem}(a, 3) = \begin{cases} (1) & - \text{ if } a = 0 \\ (2) & - \text{ if } a = S(z) \text{ and } \text{rem}(z, 3) + 1 = 3 \\ (3) & - \text{ if } a = S(z) \text{ and } \text{rem}(z, 3) + 1 \neq 3 \end{cases}$$

- (D) Show that $\text{rem}(a, 3)$ is primitive recursive. You can use the prior item, along with any functions shown to be primitive recursive in the section body, Exercise 3.22 and Exercise 3.24. (Compared with Definition 3.3, here the two arguments are switched, which is only a typographic difference.)
- (E) Extend the prior item to show that $\text{rem}(a, b)$ is primitive recursive.

3.29 The function $\text{div}: \mathbb{N}^2 \rightarrow \mathbb{N}$ gives the integer part of the division of the first argument by the second. Thus, $\text{div}(5, 3) = 1$ and $\text{div}(10, 3) = 3$.

- (A) Fill in this table.

a	0	1	2	3	4	5	6	7	8	9	10
$\text{div}(a, 3)$											

- (B) Much of the time $\text{div}(a + 1, 3) = \text{div}(a, 3)$. Under what circumstance does it not happen?
- (C) Show that $\text{div}(a, 3)$ is primitive recursive. You can use the prior exercise, along with any functions shown to be primitive recursive in the section body, Exercise 3.22 and Exercise 3.24. (Compared with Definition 3.3, here the two arguments are switched, which is only a difference of appearance.)
- (D) Show that $\text{div}(a, b)$ is primitive recursive.

3.30 The floor function $f(x/y) = \lfloor x/y \rfloor$ returns the largest natural number less than or equal to x/y . Show that it is primitive recursive. Hint: you may use any function defined in the section or stated in a prior exercise but bounded minimization is the place to start.

3.31 In 1202 Fibonacci asked: *A certain man put a pair of rabbits in a place surrounded on all sides by a wall. How many pairs of rabbits can be produced from that pair in a year if it is supposed that every month each pair begets a new pair which from the second month on becomes productive?* This leads to a recurrence.

$$F(n) = \begin{cases} 1 & - \text{if } n = 0 \text{ or } n = 1 \\ F(n - 1) + F(n - 2) & - \text{otherwise} \end{cases}$$

- (A) Compute $F(0)$ through $F(10)$. (Note: this is not now in a form that matches the primitive recursion schema, although we could rewrite it that way using Exercise 3.22 and Exercise 3.26.)
- (B) Show that F is primitive recursive. You may use the results from earlier, including Exercise 3.22, Exercise 3.24, Exercise 3.26, and Exercise 3.27.

3.32 Let $C(x, y) = 0 + 1 + 2 + \dots + (x + y) + y$.

- (A) Make a table of the values of $C(x, y)$ for $0 \leq x \leq 4$ and $0 \leq y \leq 4$.
- (B) Show that $C(x, y)$ is primitive recursive. You can use the functions shown to be primitive recursive in the section body, along with Exercise 3.22, Exercise 3.22, Exercise 3.27, and Exercise 3.27.

3.33 Pascal's Triangle gives the coefficients of the powers of x in the expansion of $(x + 1)^n$. For example, $(x + 1)^2 = x^2 + 2x + 1$ and row two of the triangle is

$\langle 1, 2, 1 \rangle$. This recurrence gives the value at row n , entry m , where $m, n \in \mathbb{N}$.

$$P(n, m) = \begin{cases} 0 & - \text{if } m > n \\ 1 & - \text{if } m = 0 \text{ or } m = n \\ P(n-1, m) + P(n-1, m-1) & - \text{otherwise} \end{cases}$$

(A) Compute $P(3, 2)$. (B) Compute the other entries from row three: $P(3, 0)$, $P(3, 1)$, and $P(3, 3)$. (C) Compute the entries in row four. (D) Show that this is primitive recursive. You may use the results from Exercise 3.22 and Exercise 3.26.

- ✓ 3.34 This is McCarthy's 91 function.

$$M(x) = \begin{cases} M(M(x+11)) & - \text{if } x \leq 100 \\ x-10 & - \text{if } x > 100 \end{cases}$$

(A) What is the output for inputs $x \in \{0, \dots, 101\}$? For larger inputs? (You may want to write a small script.) (B) Use the prior item to show that this function is primitive recursive. You may use the results from Exercise 3.22.

3.35 Show that every primitive recursive function is total.

3.36 Let g, h be natural number functions (that are total). Where f is defined by primitive recursion from g and h , show that f is well-defined. That is, show that if two functions both satisfy Definition 3.3 then they are equal, so the same inputs they will yield the same outputs.

SECTION

I.4 General recursion

Every primitive recursive function is intuitively mechanically computable. What about the converse: is every mechanically computable function primitive recursive? Here we will answer ‘no’.[†]

Ackermann functions One reason to think that there are functions that are mechanically computable but not primitive recursive is that some mechanically computable functions do not have an output for some inputs but all primitive recursive functions are total.

We could try to patch this, perhaps with: for any f that is mechanically computable, consider the function \hat{f} whose output is 0 if $f(x)$ is not defined, and whose output otherwise is $\hat{f}(x) = f(x) + 1$. Then \hat{f} is total and in a sense has the same computational content as f . Were we able to show that any such \hat{f} is primitive recursive then we would have simulated f with a primitive recursive function. However, we will show that no such patch is possible. We will give a function that is intuitively mechanically computable and total but that is not primitive recursive.

[†]That’s why the diminutive ‘primitive’ is in the name—while the class is interesting and important, it isn’t big enough to contain every effective function.

An important feature of this function is that it arises naturally so we will introduce it using familiar operations. Recall that the addition operation is repeated successor, that multiplication is repeated addition, and that exponentiation is repeated multiplication.

$$x + y = \underbrace{\mathcal{S}(\mathcal{S}(\dots \mathcal{S}(x)))}_{y \text{ many}} \quad x \cdot y = \underbrace{x + x + \dots + x}_{y \text{ many}} \quad x^y = \underbrace{x \cdot x \cdot \dots \cdot x}_{y \text{ many}}$$

This is a compelling pattern.

The pattern is especially striking when we express these functions using the schema of primitive recursion. For that, start by defining \mathcal{H}_0 to be the successor function, $\mathcal{H}_0 = \mathcal{S}$.

$$\begin{aligned} \text{plus}(x, y) &= \mathcal{H}_1(x, y) = \begin{cases} x & - \text{if } y = 0 \\ \mathcal{H}_0(x, \mathcal{H}_1(x, y - 1)) & - \text{otherwise} \end{cases} \\ \text{product}(x, y) &= \mathcal{H}_2(x, y) = \begin{cases} 0 & - \text{if } y = 0 \\ \mathcal{H}_1(x, \mathcal{H}_2(x, y - 1)) & - \text{otherwise} \end{cases} \\ \text{power}(x, y) &= \mathcal{H}_3(x, y) = \begin{cases} 1 & - \text{if } y = 0 \\ \mathcal{H}_2(x, \mathcal{H}_3(x, y - 1)) & - \text{otherwise} \end{cases} \end{aligned}$$

The pattern shows in the ‘otherwise’ lines. Each one satisfies that $\mathcal{H}_n(x, y) = \mathcal{H}_{n-1}(x, \mathcal{H}_n(x, y - 1))$.

Because of this pattern we call each \mathcal{H}_n the **level n** function, so that successor is level 0, addition is the level 1 operation, multiplication is the level 2 operation, and exponentiation is level 3. The definition below takes n as a parameter, writing $\mathcal{H}(n, x, y)$ in place of $\mathcal{H}_n(x, y)$, to get all of the levels into one formula.

4.1 **DEFINITION** This is the **hyperoperation** $\mathcal{H}: \mathbb{N}^3 \rightarrow \mathbb{N}$.

$$\mathcal{H}(n, x, y) = \begin{cases} y + 1 & - \text{if } n = 0 \\ x & - \text{if } n = 1 \text{ and } y = 0 \\ 0 & - \text{if } n = 2 \text{ and } y = 0 \\ 1 & - \text{if } n > 2 \text{ and } y = 0 \\ \mathcal{H}(n - 1, x, \mathcal{H}(n, x, y - 1)) & - \text{otherwise} \end{cases}$$

4.2 **LEMMA** $\mathcal{H}_0(x, y) = y + 1$, $\mathcal{H}_1(x, y) = x + y$, $\mathcal{H}_2(x, y) = x \cdot y$, $\mathcal{H}_3(x, y) = x^y$.

Proof The level 0 statement $\mathcal{H}_0(x, y) = y + 1$ is in the definition of \mathcal{H} .

We prove the level 1 statement $\mathcal{H}_1(x, y) = x + y$ by induction on y . For the $y = 0$ base step, the definition is that $\mathcal{H}(1, x, 0) = x$, which equals $x + 0 = x + y$. For the inductive step, assume that the statement holds for $y = 0, \dots, y = k$ and

consider the $y = k + 1$ case. The definition is $\mathcal{H}_1(x, k + 1) = \mathcal{H}_0(x, \mathcal{H}_1(x, k))$. Apply the inductive hypothesis to get $\mathcal{H}_0(x, x + k)$. By the prior paragraph this equals $x + k + 1 = x + y$.

The other two, \mathcal{H}_2 and \mathcal{H}_3 , are Exercise 4.15. \square

- 4.3 REMARK Level 4, the level above exponentiation, is **tetration**. The first few values are $\mathcal{H}_4(x, 0) = 1$, and $\mathcal{H}_4(x, 1) = \mathcal{H}_3(x, \mathcal{H}_4(x, 0)) = x^1 = x$, and $\mathcal{H}_4(x, 2) = \mathcal{H}_3(x, \mathcal{H}_4(x, 1)) = x^x$, as well as these two.

$$\mathcal{H}_4(x, 3) = \mathcal{H}_3(x, \mathcal{H}_4(x, 2)) = x^{x^x} \quad \text{and} \quad \mathcal{H}_4(x, 4) = x^{x^{x^x}}$$

This is a **power tower**. To evaluate these, recall that in exponentiation the parentheses are significant, so for instance these two are unequal: $(3^3)^3 = 27^3 = 3^9 = 19\,683$ and $3^{(3^3)} = 3^{27} = 7\,625\,597\,484\,987$. Tetration does it in the second, larger, way. The rapid growth of the output values is a striking aspect of tetration, and of the hyperoperation in general. For instance, $\mathcal{H}_3(4, 4)$ is much greater than the number of elementary particles in the universe.

Hyperoperation is mechanically computable. Its code is a transcription of the definition.

```
(define (H n x y)
  (cond
    [(= n 0) (+ y 1)]
    [(and (= n 1) (= y 0)) x]
    [(and (= n 2) (= y 0)) 0]
    [(and (> n 2) (= y 0)) 1]
    [else (H (- n 1) x (H n x (- y 1))))]))
```

However, hyperoperation's recursion line

$$\mathcal{H}(n, x, y) = \mathcal{H}(n - 1, x, \mathcal{H}(n, x, y - 1))$$

does not fit the form of primitive recursion.

$$f(x_0, \dots, x_{k-1}, y) = h(f(x_0, \dots, x_{k-1}, y - 1), x_0, \dots, x_{k-1}, y - 1)$$

The problem is not that the arguments are in a different order; that is cosmetic. The reason that \mathcal{H} does not work as h is that the definition of primitive recursive function, Definition 3.3, requires that h be a function for which we already have a primitive recursive derivation.

Of course, just because one definition has the wrong form doesn't mean that there is no definition with the right form. However, Ackermann[†] proved that there is no such definition, that \mathcal{H} is not primitive recursive. The proof is a detour so it is in Extra D but in summary, \mathcal{H} grows faster than any primitive recursive function. That is, for any primitive recursive function f of three inputs, there is a sufficiently large $N \in \mathbb{N}$ such that for all $n, x, y \in \mathbb{N}$, if $n, x, y > N$ then $\mathcal{H}(n, x, y) > f(n, x, y)$.

[†]We have seen Ackermann already as one of the people who stated the *Entscheidungsproblem*. Functions having the same recursion as \mathcal{H} are **Ackermann functions**.

This proof is about uniformity: at every level the function \mathcal{H}_n is primitive recursive, but no primitive recursive function encompasses all levels at once—there is no one primitive recursive way to compute them all.

4.4 **THEOREM** The hyperoperation \mathcal{H} is not primitive recursive.

This relates to a point from the discussion of Church's Thesis. We have observed that if a function is primitive recursive then it is mechanically computable. We have built a pile of natural and interesting functions that are mechanically computable, and demonstrated that they are primitive recursive. So 'primitive recursive' may seem to have many of the same characteristics as 'Turing machine computable'. The difference is that we now have an intuitively mechanically computable function that is not primitive recursive. That is, primitive recursive fails the 'coverage' test from the Church's Thesis discussion. So we need to expand from primitive recursive functions to a larger collection.



Wilhelm
Ackermann
1896–1962

μ recursion The right direction is hinted at in the prior section's Exercise 3.26 and Exercise 3.27. Primitive recursion does operations that are bounded, such as bounded sum $\sum_{0 \leq i < y} g(i) = g(0) + \dots + g(y-1)$. We can show that a programming language having only bounded loops computes all of the primitive recursive functions; see Extra E. To include all of the functions that are intuitively mechanically computable we must add an operation that is unbounded.

4.5 **DEFINITION** Suppose that $g: \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ is total, so that for every input tuple there is an output number. Then $f: \mathbb{N}^n \rightarrow \mathbb{N}$ is defined from g by **minimization** or **μ -recursion**, written $f(\vec{x}) = \mu y [g(\vec{x}, y) = 0]$,[‡] if $f(\vec{x})$ is the least number y such that $g(\vec{x}, y) = 0$.

This is unbounded search: we have in mind that g is mechanically computable and we calculate $g(\vec{x}, 0)$, then $g(\vec{x}, 1)$, etc., waiting until one of them gives the output 0. If that ever happens, so that $g(\vec{x}, n) = 0$ for some least n , then $f(\vec{x}) = n$. If it never happens that the output is 0 then $f(\vec{x})$ is undefined.

4.6 **EXAMPLE** Euler noticed in 1772 that the polynomial $p(y) = y^2 + y + 41$ initially seems to output only primes.

y	0	1	2	3	4	5	6	7	8	9
$p(y)$	41	43	47	53	61	71	83	97	113	131

We could test whether this ever fails by doing an unbounded search for non-primes. This function tests for the primality of a quadratic polynomial's output.[†]

$$g(x_0, x_1, x_2, y) = g(\vec{x}, y) = \begin{cases} 1 & \text{if } x_0y^2 + x_1y + x_2 \text{ is prime} \\ 0 & \text{otherwise} \end{cases}$$

Then the search is $f(\vec{x}) = \mu y [g(\vec{x}, y) = 0]$.

[‡]The vector \vec{x} abbreviates x_0, \dots, x_{n-1} .

Some code illustrates. Start with a test for primality provided with Racket,

```
(require math/number-theory) ;; provides the routine prime?
```

and use it for g .

```
(define (g x0 x1 x2 y)
  (if (prime? (+ (* x0 y y) (* x1 y) x2))
      1
      0))
```

With that, the search function

```
(define (f x0 x1 x2)
  (define (f-helper y)
    (if (= 0 (g x0 x1 x2 y))
        y
        (f-helper (add1 y)))))
  (f-helper 0))
```

finds that the polynomial first returns a non-prime for input 40.

```
> (f 1 1 41)
40
```

A by-hand check shows that $p(40) = 1681$, which is not prime because $1681 = 41^2$.

Using the minimization operator we can get functions whose output value is undefined for some inputs.

- 4.7 EXAMPLE If $g(x, y) = 1$ for all $x, y \in \mathbb{N}$ then $f(x) = \mu y [g(x, y) = 0]$ is undefined for all x .

In that example we imagine f searching and searching but it will never find such a y . In the next example no one currently knows whether the search will end.

- 4.8 EXAMPLE Goldbach's conjecture is that every even number can be written as the sum of at most two primes. Here are the first few instances: $2 = 2$, $4 = 2 + 2$, $6 = 3 + 3$, $8 = 5 + 3$, $10 = 7 + 3$. No one knows if it is true. A natural attack is to search for a counterexample. With this auxillary function

$$\text{gb-check}(n) = \begin{cases} i & - \text{the numbers } i \text{ and } n - i \text{ are both prime, and } i \leq n - i \\ 0 & - \text{otherwise} \end{cases}$$

we get a version of the definition's g (here \vec{x} is empty).

$$\text{gb-g}(y) = \begin{cases} 1 & - \text{if } \text{gb-check}(2y) \neq 0 \\ 0 & - \text{otherwise} \end{cases}$$

The unbounded search is $\text{gb-f}() = \mu y [\text{gb-g}(y) = 0]$.[†]

The code for the auxillary

```
;; Check if a number n is the sum of two primes
```

[†]We are using $x_0 y^2 + x_1 y + x_2$ rather than just $y^2 + y + 41$ so that the inputs to g include \vec{x} , to better illustrate the definition.

```
;; Returns minimal i < n such that i and n-i are prime; returns #f if no such i
(define (gb-check n)
  (for/first ([i (in-range 2 (add1 n))])
    #:when (and (prime? i)
                 (prime? (- n i))))
    i)))
```

immediately gives the code for gb-g

```
(define (gb-g y)
  (if (gb-check (* 2 y))
      1
      0))
```

and gb-f.

```
(define (gb-f)
  (define (gb-f-helper y)
    (if (= 0 (gb-g y))
        y
        (gb-f-helper (add1 y))))
  (gb-f-helper 2)) ; start with 4
```

The gb-f routine just runs and runs, without halting. This is not an infinite loop because gb-f is trying different numbers each time, but the experience of executing it is that it fails to halt until the user runs out of patience and pulls the plug. Researchers have gone past 4×10^{18} without finding a counterexample. While that is not a proof, it is nonetheless striking.

- 4.9 EXAMPLE Imagine that you have a mathematical theorem that you want to prove, perhaps Goldbach's Conjecture. Here is a sketch of a way to proceed. A proof is a sequence of statements in a suitable formal language whose final statement is the desired theorem. You could run a search as: for each n , interpret it as a string (perhaps convert it to binary and interpret the binary as a string) and check whether that string is a proof of the theorem. If this search halts then you are done. Obviously this relates to the *Entscheidungsproblem*.

These examples show that unbounded search is a natural computational construct. It is also a theme in this book. For instance, we will later consider the question of which programs halt and a natural way to think about a computation not halting is as a search for a step where it halts.

- 4.10 DEFINITION A function is **general recursive** or **partial recursive**, or μ -**recursive**, or just **recursive**, if it can be derived from the initial operations of the **zero** function $Z(\vec{x}) = 0$, the **successor** function $S(x) = x + 1$, and the **projection** functions $I_i(x_0, \dots, x_i \dots x_{k-1}) = x_i$ by a finite number of applications of function composition, the schema of primitive recursion, and minimization.

S Kleene showed that the set of functions satisfying this definition is the same as the Turing machine-based set of computable functions given in Definition 1.9.

I.4 Exercises

[†]Here, the vector of parameters $\vec{x} = x_0, \dots, x_{n-1}$ is not present, that is, $n = 0$.

Some of these have answers that are tedious to compute. It may help to use a computer, for instance by writing a Racket program or using Sage.

- ✓ 4.11 Find the value of $\mathcal{H}_4(2, 0)$, $\mathcal{H}_4(2, 1)$, $\mathcal{H}_4(2, 2)$, $\mathcal{H}_4(2, 3)$, and $\mathcal{H}_4(2, 4)$.
- 4.12 Graph $\mathcal{H}_1(2, y)$ up to $y = 9$. Also graph $\mathcal{H}_2(2, y)$ and $\mathcal{H}_3(2, y)$ over the same range. Put all three plots on the same axes.
- ✓ 4.13 How many years is $\mathcal{H}_4(3, 3)$ seconds?
- 4.14 What is the ratio $\mathcal{H}_3(3, 3)/\mathcal{H}_2(2, 2)$?
- ✓ 4.15 Finish the proof of Lemma 4.2 by verifying that $\mathcal{H}_2(x, y) = x \cdot y$ and $\mathcal{H}_3(x, y) = x^y$.
- 4.16 This variant of \mathcal{H} is often labeled “the” Ackermann function.

$$\mathcal{A}(k, y) = \begin{cases} y + 1 & - \text{if } k = 0 \\ \mathcal{A}(k - 1, 1) & - \text{if } y = 0 \text{ and } k > 0 \\ \mathcal{A}(k - 1, \mathcal{A}(k, y - 1)) & - \text{otherwise} \end{cases}$$

It has different boundary conditions but the same recursion, the same bottom line. (In general, any function with that recursion is an Ackermann function. Extra D has more about this variant.) Compute $\mathcal{A}(k, y)$ for $0 \leq k < 4$ and $0 \leq y < 6$.

- 4.17 Prove that the computation of $\mathcal{H}(n, x, y)$ always terminates.
- 4.18 In defining general recursive functions, Definition 4.10, we get all computable functions by starting with the primitive recursive functions and adding minimization. What if instead of minimization we had added Ackermann’s function; would we then have all computable functions?
- ✓ 4.19 Let $g(x, y) = 0$ if $x + y = 100$ and let $g(x, y) = 1$ otherwise. Now let $f(x) = \mu y [g(x, y) = 100]$. For each, find the value or say that it is not defined. (A) $f(0)$ (B) $f(1)$ (C) $f(50)$ (D) $f(100)$ (E) $f(101)$. Give an expression for f that does not include μ -recursion.
- 4.20 Let $g(x, y) = 0$ if $x \cdot y = 100$ and $g(x, y) = 1$ otherwise. Also let $f(x) = \mu y [g(x, y) = 100]$. For each, find the value or say that it is not defined. (A) $f(0)$ (B) $f(1)$ (C) $f(50)$ (D) $f(100)$ (E) $f(101)$
- ✓ 4.21 A **Fermat number** has the form $F_n = 2^{2^n} + 1$ for $n \in \mathbb{N}$. The first few, $F_0 = 3$, $F_1 = 5$, $F_2 = 17$, $F_3 = 257$, and $F_4 = 65537$, are prime. But F_5 is not prime, nor are F_6, \dots, F_{32} . (We don’t know of any primes for higher n .) Let $g(x, y) = 0$ if y is a Fermat prime and larger than F_x , and let $g(x, y) = 1$ otherwise. Also let $f(x) = \mu y [g(x, y) = 0]$. For each, what can you say? (A) $f(0)$ (B) $f(1)$ (C) $f(50)$ (D) $f(100)$ (E) $f(F_4)$
- 4.22 Let $g(x, y) = 0$ if y^2 is greater than or equal to x , and let $g(x, y) = 1$ otherwise. Also let $f(x) = \mu y [g(x, y) = 0]$. Find each, or state ‘undefined’. (A) $f(0)$ (B) $f(1)$ (C) $f(50)$ (D) $f(100)$ (E) $f(x)$
- 4.23 Let $\text{notrelprime}(x, y) = 0$ if $x > 1$ and $y > 1$ and the two are not relatively prime, and let $\text{notrelprime}(x, y) = 1$ otherwise. Find each $f(x) =$

$\mu y [\text{notrelprime}(x, y) = 0].$ (A) $f(0)$ (B) $f(1)$ (C) $f(2)$ (D) $f(3)$ (E) $f(4)$ (F) $f(42)$ (G) $f(x)$

4.24 Let $g(x, y) = \lceil (x+1)/(y+1) - 1 \rceil$ and let $f(x) = \mu y [g(x, y) = 0].$

(A) Find $f(x)$ for $0 \leq x < 6.$

(B) Give a description of f that does not use μ -recursion.

4.25 (A) Prove that the function remtwo: $\mathbb{N} \rightarrow \{0, 1\}$ giving the remainder on division by two is primitive recursive.

(B) Use that to prove that this function is μ -recursive: $f(n) = 1$ if n is even, and $f(n) \uparrow$ if n is odd.

✓ 4.26 Consider the Turing machine $\mathcal{P} = \{q_0B1q_1, q_01Rq_0, q_1BRq_2, q_11Lq_1\}.$ Define $g(x, y) = 0$ if the machine \mathcal{P} , when started on a tape that is blank except for x -many consecutive 1's and with the head under the leftmost 1, has halted after step $y.$ Otherwise, $g(x, y) = 1.$ Find $f(x) = \mu y [g(x, y) = 0]$ for $x < 6.$

✓ 4.27 Define $g(x, y)$ by: start $\mathcal{P} = \{q_0B1q_2, q_01Lq_1, q_1B1q_2, q_111q_2\}$ on a tape that is blank except for x -many consecutive 1's and with the head under the leftmost 1. If \mathcal{P} has halted after step y then $g(x, y) = 0$ and otherwise $g(x, y) = 1.$ Let $f(x) = \mu y [g(x, y) = 0].$ Find $f(x)$ for $x < 6.$ (This machine does the same task as the one in the prior exercise, but faster.)

4.28 Consider this Turing machine.

$$\{q_0BRq_1, q_01Rq_1, q_1BRq_2, q_11Rq_2, q_2BLq_3, q_21Lq_3, q_3BLq_4, q_31Lq_4\}$$

Let $g(x, y) = 0$ if this machine, when started on a tape that is all blank except for x -many consecutive 1's and with the head under the leftmost 1, has halted after y steps. Otherwise, $g(x, y) = 1.$ Let $f(x) = \mu y [g(x, y) = 0].$ Find: (A) $f(0)$ (B) $f(1)$ (C) $f(2)$ (D) $f(x).$

✓ 4.29 Define $h: \mathbb{N}^+ \rightarrow \mathbb{N}$ by: $h(n) = n/2$ if n is even, and otherwise $h(n) = 3n + 1.$ Let $H(n, k)$ be the k -fold composition of h with itself, so $H(n, 1) = h(n), H(n, 2) = h \circ h(n), H(n, 3) = h \circ h \circ h(n),$ etc. (We can take $H(n, 0) = 0,$ although its value isn't interesting.) Let $C(n) = \mu k [H(n, k) = 1].$

(A) Compute $H(4, 1), H(4, 2),$ and $H(4, 3).$

(B) Find $C(4),$ if it is defined.

(C) Find $C(5),$ it is defined.

(D) Find $C(11),$ it is defined.

(E) Find $C(n)$ for all $n \in [1..20],$ where defined.

The **Collatz conjecture** is that $C(n)$ is defined for all $n.$ No one knows if it is true.

EXTRA

I.A Turing machine simulator

The source repository for this book includes a program, written in Racket, to simulate a Turing machine. It is in the directory `src/scheme/prologue`. Here we will

show how to run this simulator. (The implementation tracks closely the description of the action of a Turing machine given on page 8.)

Example 1.1 gives a Turing machine that computes the predecessor function.

$$\mathcal{P}_{\text{pred}} = \{ q_0 \text{BL} q_1, q_0 \text{1R} q_0, q_1 \text{BL} q_2, q_1 \text{1B} q_1, q_2 \text{BR} q_3, q_2 \text{1L} q_2 \}$$

The simulation program will use this file, called `pred.tm`.

```
0 B L 1
0 1 R 0
1 B L 2
1 1 B 1
2 B R 3
2 1 L 2
```

Thus the simulator for any particular Turing machine is really the pair consisting of the Racket code along with the machine's description, as above.

Below is a run of the simulator, showing its command line invocation. It takes input from the file `machines/pred.tm`. The machine starts with a current symbol of 1 and with the tape to the right of the current symbol containing 11 (the tape to the left is empty). Thus, the entire tape input is $\tau = 111$. Since the predecessor of 3 is 2, we expect that when it finishes the tape will contain 11, with the rest blank.

```
$ ./turing-machine.rkt -f machines/pred.tm -c "1" -r "11"
step 0: q0: *1*11
step 1: q0: 1*1*1
step 2: q0: 11*1*
step 3: q0: 111*B*
step 4: q1: 11*1*B
step 5: q1: 11*B*B
step 6: q2: 1*1*BB
step 7: q2: *1*1BB
step 8: q2: *B*11BB
step 9: q3: B*1*1BB
step 10: HALT
```

The output is crude but good enough for small experiments. The command line `turing-machine.rkt --help` will tell you the simulator's options.

I.A Exercises

A.1 Run the simulator on $\mathcal{P}_{\text{pred}}$ starting with 11111. Also start with an empty tape.

A.2 Run the simulator on Example 1.2's \mathcal{P}_{add} to do 1 + 2. Also simulate 0 + 2 and 0 + 0.

A.3 Write a Turing machine to perform the operation of adding 3, so that given as input a tape containing only a string of n consecutive 1's, it returns a tape with a string of $n + 3$ consecutive 1's. Follow our convention that when the program starts and ends the head is under the first 1. Run it on the simulator, with an input of 4 consecutive 1's, and also with an empty tape.

A.4 Write a machine to decide if the input contains the substring 010. Fix $\Sigma = \{0, 1, B\}$. The machine starts with the tape blank except for a contiguous

string of 0's and 1's, and with the head under the first non-blank symbol. When it finishes, the tape will have either just a 1 if the input contained the desired substring, or otherwise just a 0. We will do this in stages, building a few of what amounts to subroutines.

- (A) Write instructions, starting in state q_{10} , so that if initially the machine's head is under the first of a sequence of non-blank entries then at the end the head will be to the right of the final such entry.
- (B) Write a sequence of instructions, starting in state q_{20} , so that if initially the head is just to the right of a sequence of non-blank entries, then at the end all entries are blank.
- (C) Write the full machine, including linking in the prior items.

EXTRA

I.B Hardware

Following Turing's approach, we've gone through a development of the definition of what is computable based on transitions. We produce a table of transition instructions and call that a mechanism. However, given a table can we be sure that there is an associated actual mechanism, a physical implementation with the behavior that we desire?

Put another way, in programming languages there are mathematical operators that are constructed from other, simpler, mathematical operators. For instance, $\sin(x)$ may be calculated via its Taylor polynomial from addition and multiplication. But the very simplest operations such as addition must happen on the hardware; how does that work?

We will show how to start with any desired behavior and from it produce a device with that behavior. For this, we will work with machines that take finite binary sequences, bitstrings, as inputs and outputs. The easiest approach is via propositional logic. (Appendix C has a review.)

Here are the three basic logic operators. These tables use 0 in place of F and 1 in place of T , as is the convention in electronics—in an electronic device these would typically stand for different voltage levels.

		<i>not P</i>		<i>P and Q</i> $P \wedge Q$		<i>P or Q</i> $P \vee Q$	
		<i>P</i>	$\neg P$	<i>P</i>	Q	<i>P and Q</i> $P \wedge Q$	<i>P or Q</i> $P \vee Q$
0	1			0	0	0	0
1	0			0	1	0	1
				1	0	0	1
				1	1	1	1

That is all we need. The three operators ' \neg ', ' \wedge ', and ' \vee ', are enough for us to go from a specified input-output behavior, a desired truth table, to a propositional logic expression having that table, having that behavior.

The two tables below show how. On the left, focus on the row with output 1. Obviously the expression $\neg P \wedge \neg Q$ makes this row take on value 1 and every other

row take on value 0.

P	Q		P	Q	R	
0	0	1	0	0	0	0
0	1	0	0	0	1	1
1	0	0	0	1	0	1
1	1	0	0	1	1	0
			1	0	0	1
			1	0	1	0
			1	1	0	0
			1	1	1	0

For the table on the right, again focus on the rows with 1's. Target the second row by considering the clause $\neg P \wedge \neg Q \wedge R$. The clause for the third row is $\neg P \wedge Q \wedge \neg R$ and the fifth row's is $P \wedge \neg Q \wedge \neg R$. Now put these clauses together with \vee 's to get the statement with the given table.

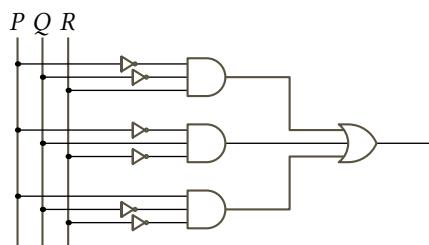
$$(\neg P \wedge \neg Q \wedge R) \vee (\neg P \wedge Q \wedge \neg R) \vee (P \wedge \neg Q \wedge \neg R) \quad (*)$$

Next we translate those expressions into physical devices. The observation that you can use this form of a propositional logic expression to systematically design logic circuits was made by C Shannon in his master's thesis. We can get electronic devices, called **gates**, that perform logical operations on signals (for this discussion we take a signal to be the presence of 5 volts). On the left below is the schematic symbol for an AND gate with two input wires and one output wire, whose behavior is that a signal only appears on the output if there is a signal on both inputs. Symbolized in the middle is an OR gate, where there is signal out if either input has a signal. On the right is a NOT gate.



Claude Shannon
1916–2001

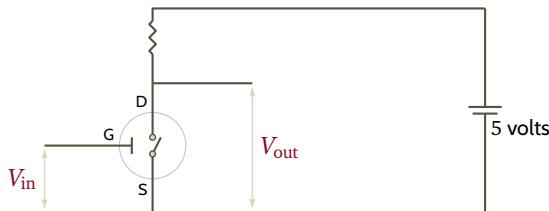
A schematic of a circuit that implements expression (*), given below, shows three input signals on the three wires at left. For instance, to implement the first clause, the top AND gate is fed the not P , the not Q , and the R signals. The second and third clauses are implemented in the other two AND gates. Then the output of the AND gates goes through the OR gate. (These AND and OR gates are engineered to take three inputs.)



Clearly by following this procedure we can in principle build a physical device with any desired input/output behavior. In particular, we can build a Turing machine in this way.

We will close with an aside. A person can wonder how these gates are constructed and in particular can wonder how a NOT gate is possible— isn't having voltage out when there is no voltage in creating something out of nothing?

The answer is that the descriptions above abstract out some things. Here is the internal construction of a type of NOT gate.



On the right is a battery, which as we shall see supplies the extra voltage. On the top left, shown as a wiggle, is a resistor. When current is flowing around the circuit, this resistor regulates the power output from the battery.

On the bottom left, shown with the circle, is a transistor. This is a semiconductor, with the property that if there is enough voltage between G and S then this component allows current from the battery to flow between D and S. (Because it is sometimes open and sometimes closed it is depicted as a switch, although it has no mechanical moving parts.) This transistor is manufactured such that an input voltage V_{in} of 5 volts will trigger this event.

To verify that this circuit inverts the signal, assume first that $V_{in} = 0$. Then there is a gap between D and S so no current flows. With no current the resistor provides no voltage drop. Consequently the output voltage V_{out} across the gap is all of the voltage supplied by the battery, 5 volts. So $V_{in} = 0$ results in $V_{out} = 5$.

Conversely, assume that $V_{in} = 5$. Then the gap disappears so that the current flows between D and S, the resistor drops the voltage, and the output is $V_{out} = 0$.

Thus, for this device the voltage out V_{out} is the opposite of the voltage in V_{in} . And, when $V_{in} = 0$ the output voltage of 5 doesn't come from nowhere; it is from the battery.

I.B Exercises

B.1 Make a table with inputs P , Q , and R for the behavior that the output is 1 if the input P equals the input R . Produce the associated disjunctive normal form expression. Draw the circuit.

B.2 Make a three-input table for the behavior: the output is 1 if a majority of the inputs are 1's. Produce the associated disjunctive normal form expression. Draw the circuit.

B.3 Make a four-input table for the behavior: the output is 1 if a majority of the inputs are 1's (this does not allow ties). Produce the associated disjunctive normal form expression. Draw the circuit.

B.4 Produce the disjunctive normal form expression with five inputs for this behavior: the output is 1 if a majority of the inputs are 1's. Draw the circuit.

B.5 For the table below, construct a disjunctive normal form propositional logic expression and use that to make a circuit.

P	Q	
0	0	0
0	1	1
1	0	1
1	1	0

B.6 One propositional logic operator that was not covered in the description is **Exclusive Or, XOR**. It is defined by: $P \text{ XOR } Q$ is 1 if $P \neq Q$, and is 0 otherwise. (A) Specify a truth table and from it construct a disjunctive normal form propositional logic expression. (B) Use that to make a circuit.

B.7 For the tables below, construct a disjunctive normal form propositional logic expression and use that to make a circuit (A) for the table on the left, (B) for the one on the right.

P	Q	R		P	Q	R	
0	0	0	0	0	0	0	1
0	0	1	1	0	0	1	0
0	1	0	1	0	1	0	1
0	1	1	0	0	1	1	0
1	0	0	0	1	0	0	0
1	0	1	1	1	0	1	0
1	1	0	0	1	1	0	1
1	1	1	0	1	1	1	1

B.8 A propositional logic operator that was not covered above is **implication, \rightarrow** . It is defined by: $P \rightarrow Q$ is 1 unless P is 1 and Q is 0.

- (A) Make a truth table and from it construct a disjunctive normal form propositional logic expression.
- (B) Use that to make a circuit.

B.9 The most natural way to add two binary numbers works like the grade school addition algorithm. Start at the right with the one's column. Add those two and possibly carry a 1 to the next column. Then add down the next column, including any carry. Repeat this from right to left.

- (A) Use this method to add the two binary numbers 1011 and 1101.
- (B) Make a truth table giving the desired behavior in adding the numbers in a column. It must have three inputs because of the possibility of a carry. It must also have two output columns, one for the total and one for the carry.
- (C) Draw the circuits.

B.10 Construct a circuit with the behavior specified in the tables below: (A) the table on the left, (B) the one on the right.

P	Q	R		P	Q	R	
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	1
0	1	1	0	0	1	1	0
1	0	0	0	1	0	0	0
1	0	1	1	1	0	1	1
1	1	0	0	1	1	0	1
1	1	1	0	1	1	1	0

EXTRA

I.C Game of Life

John von Neumann was one of the twentieth century's most prolific and influential mathematicians. One of the many things he studied was the problem of humans on Mars. He thought that to colonize Mars we should first send robots. Mars is red because it is full of rust, iron oxide. Robots could mine that rust, break it into iron and oxygen, and release the oxygen into the atmosphere. With all the iron, the robots could make more robots. So von Neumann was thinking about making machines that could self-reproduce. (We will study more about self-reproduction later.) A suggestion from S Ulam led him to use a cell-based approach. He placed computational devices in a grid, making a [cellular automaton](#).



John von Neumann 1903-1957

Interest in cellular automata greatly increased with the appearance of the Game of Life, by J Conway. It was featured in M Gardner's celebrated *Mathematical Games* column of *Scientific American* in October 1970. The rules are simple enough that a person could immediately start experimenting. Lots of people did. When personal computers appeared, Life became a computer craze since it is easy for a beginner to program.



John Conway 1937-2020

Start, by drawing a two-dimensional grid of square cells, like graph paper or a chess board. Each cell has eight neighbors, the ones that are horizontally, vertically, or diagonally adjacent. The game proceeds in stages, or [generations](#). At each generation each cell is in one of two states, alive or dead. For the next generation the next state is determined by: (1) a live cell with two or three live neighbors will again be live at the next generation but any other live cell dies, (2) a dead cell with exactly three live neighbors becomes alive at the next generation but other dead cells stay dead. (The backstory goes that live cells will die if they are either isolated or overcrowded, while if the environment is just right then life can spread.) To begin, we seed the board with some initial pattern.

backstory goes that live cells will die if they are either isolated or overcrowded, while if the environment is just right then life can spread.) To begin, we seed the board with some initial pattern.

As Gardner noted, the rules of the game balance tedious simplicity against impenetrable complexity.

The basic idea is to start with a simple configuration of counters (organisms), one to a cell, then observe how it changes as you apply Conway's "genetic laws" for births, deaths, and survivals. Conway chose his rules carefully, after a long period of experimentation, to meet three desiderata:

1. There should be no initial pattern for which there is a simple proof that the population can grow without limit.
2. There should be initial patterns that apparently do grow without limit.
3. There should be simple initial patterns that grow and change for a considerable period of time before coming to end in three possible ways: fading away completely (from overcrowding or becoming too sparse), settling into a stable configuration that remains unchanged thereafter, or entering an oscillating phase in which they repeat an endless cycle of two or more periods.

In brief, the rules should be such as to make the behavior of the population unpredictable.

The result is, as Conway says, a "zero-player game." It is a mathematical recreation in which patterns evolve in fascinating ways.

Many starting patterns do not result in any interesting behavior at all. The simplest nontrivial pattern, a single cell, immediately dies.



Generation 0



Generation 1

The pictures show the part of the game board containing the cells that are alive. Two generations suffice to show everything that happens, which isn't much.

Some other patterns don't die, but don't do much of anything, either. This is a **block**. It is stable from generation to generation.

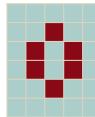


Generation 0

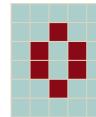


Generation 1

Because it doesn't change, Conway calls this a "still life." Another still life is the **beehive**.

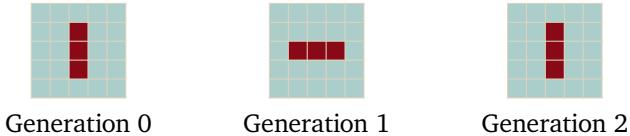


Generation 0



Generation 1

But many patterns are not still. This three-cell pattern, the **blinker**, does a simple oscillation.



Other patterns move. This is a **glider**, the most famous pattern in Life.

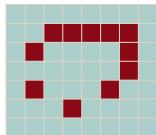


It moves one cell vertically and one horizontally every four generations, crawling across the screen.

C.1 ANIMATION: Gliding, left and right.

When Conway came up with the Life rules, he was not sure whether there is a pattern where the total number of live cells keeps on growing. Bill Gosper showed that there is, by building the **glider gun** which produces a new glider every thirty generations.

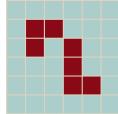
The glider pattern is an example of a **spaceship**, a pattern that reappears, displaced, after a number of generations. Here is another, the **medium weight spaceship**.



It also crawls across the screen.

C.2 ANIMATION: Moving across space.

Another important pattern is the **eater**, which eats gliders and other spaceships.



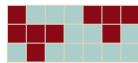
Here it eats a medium weight spaceship.

C.3 ANIMATION: Eating a spaceship.

How powerful is the game, as a computational system? Although it is beyond our scope, you can build Turing machines in the game and so it is able to compute anything that can be mechanically computed (Rendell 2011).

I.C Exercises

C.4 A **methuselah** is a small pattern that stabilizes only after a long time. This pattern is a **rabbit**. How long does it take to stabilize?



C.5 How many 3×3 blocks are there? 4×4 ? Write a program that inputs a dimension n and returns the number of $n \times n$ blocks.

C.6 How many of the 3×3 patterns will result in any cells on the board that survive into the next generation? That survive ten generations?

C.7 Write code that takes in a number of rows n , a number of columns m and a number of generations i , and returns how many of the $n \times m$ patterns will result in any surviving cells after i generations.

EXTRA

I.D Ackermann's function is not primitive recursive

The hyperoperation \mathcal{H} is intuitively mechanically computable.

$$\mathcal{H}(n, x, y) = \begin{cases} y + 1 & - \text{if } n = 0 \\ x & - \text{if } n = 1 \text{ and } y = 0 \\ 0 & - \text{if } n = 2 \text{ and } y = 0 \\ 1 & - \text{if } n > 2 \text{ and } y = 0 \\ \mathcal{H}(n - 1, x, \mathcal{H}(n, x, y - 1)) & - \text{otherwise} \end{cases}$$

We have cited that it is not primitive recursive. We will prove that here for a closely related variant.

In \mathcal{H} 's 'otherwise' line, while the level is n and the recursion is on y , the variable x does not play an active role. R Péter noted this and got a function with a simpler definition, lowering the number of variables by one, by considering $\mathcal{A}(n, y, y)$. That, and tweaking the initial value of each level, gives this.

$$\mathcal{A}(k, y) = \begin{cases} y + 1 & - \text{if } k = 0 \\ \mathcal{A}(k - 1, 1) & - \text{if } y = 0 \text{ and } k > 0 \\ \mathcal{A}(k - 1, \mathcal{A}(k, y - 1)) & - \text{otherwise} \end{cases}$$



Rózsa Péter
1905–1977

Any function with a recursion like the one in the bottom line is an **Ackermann function**. We will prove that \mathcal{A} is not primitive recursive.

Since the new function has only two variables, we can show a table.

	$y = 0$	$y = 1$	$y = 2$	$y = 3$	$y = 4$	$y = 5$	
$k = 0$	1	2	3	4	5	6	...
$k = 1$	2	3	4	5	6	7	...
$k = 2$	3	5	7	9	11	13	...
$k = 3$	5	13	29	61	125	253	...
$k = 4$	13	65 533	...				

The next two entries give a sense of the growth rate of this function.

$$\mathcal{A}(4, 2) = 2^{65536} - 3 \quad \mathcal{A}(4, 3) = 2^{(2^{65536})} - 3$$

Those are big numbers.

- D.1 LEMMA (A) $\mathcal{A}(k, y) > y$
 (B) $\mathcal{A}(k, y + 1) > \mathcal{A}(k, y)$, and in general if $\hat{y} > y$ then $\mathcal{A}(k, \hat{y}) > \mathcal{A}(k, y)$
 (C) $\mathcal{A}(k + 1, y) \geq \mathcal{A}(k, y + 1)$
 (D) $\mathcal{A}(k, y) > k$
 (E) $\mathcal{A}(k + 1, y) > \mathcal{A}(k, y)$ and in general if $\hat{k} > k$ then $\mathcal{A}(\hat{k}, y) > \mathcal{A}(k, y)$
 (F) $\mathcal{A}(k + 2, y) > \mathcal{A}(k, 2y)$

Proof We will verify the first item here and leave the others as exercises. They all proceed the same way, with an induction inside of an induction.

The first item is that for all k and for all y we have that $\mathcal{A}(k, y) > y$. We will prove it by induction on k .

The $k = 0$ base step holds because $\mathcal{A}(0, y) = y + 1$ and thus $\mathcal{A}(0, y) > y$. For the inductive step, assume that this holds for $k = 0, \dots, n$

$$\forall y [\mathcal{A}(k, y) > y] \tag{*}$$

and consider the $k = n + 1$ case. We must verify this.

$$\forall y [\mathcal{A}(n + 1, y) > y] \tag{**}$$

We will use induction on y . In the $y = 0$ base step of this inside induction, the definition gives $\mathcal{A}(n+1, 0) = \mathcal{A}(n, 1)$ and by the inductive hypothesis that statement $(*)$ is true when $k = n$, we have $\mathcal{A}(n, 1) > 1 > y = 0$.

For the inductive step, assume that statement $(**)$ holds for $y = 0, \dots, m$ and consider $y = m+1$. The definition gives $\mathcal{A}(n+1, m+1) = \mathcal{A}(n, \mathcal{A}(n+1, m))$. By the inductive hypothesis of this inside induction $(**)$, we have $\mathcal{A}(n+1, m) > m$. Then by the inductive hypothesis of the outer induction $(*)$, when $\mathcal{A}(n, \mathcal{A}(n+1, m))$ has a second argument greater than m then its result is greater than m , as required. \square

We will abbreviate the function input list x_0, \dots, x_{n-1} by the vector \vec{x} . And we will write the maximum of the vector, $\max(\vec{x})$, to mean the maximum of its components, $\max(\{x_0, \dots, x_{n-1}\})$.

- D.2 **DEFINITION** A function s is **level** k , where $k \in \mathbb{N}$, if $\mathcal{A}(k, \max(\vec{x})) > s(\vec{x})$ for all \vec{x} .

By Lemma D.1.E, if a function is level k then it is also level \hat{k} for any $\hat{k} > k$.

- D.3 **LEMMA** If for some $k \in \mathbb{N}$ each function g_0, \dots, g_{m-1}, h is level k , and if the function f is obtained by composition as $f(\vec{x}) = h(g_0(\vec{x}), \dots, g_{m-1}(\vec{x}))$, then f is level $k+2$.

Proof Apply Lemma D.1's item c, and then the definition of \mathcal{A} .

$$\mathcal{A}(k+2, \max(\vec{x})) \geq \mathcal{A}(k+1, \max(\vec{x}) + 1) = \mathcal{A}(k, \mathcal{A}(k+1, \max(\vec{x}))) \quad (*)$$

Focusing on the second argument of the right-hand expression, use Lemma D.1.E and the assumption that each function g_0, \dots, g_{m-1} is level k to get that for each index $i \in \{1, \dots, m-1\}$ we have $\mathcal{A}(k+1, \max(\vec{x})) > \mathcal{A}(k, \max(\vec{x})) > g_i(\vec{x})$. Thus $\mathcal{A}(k+1, \max(\vec{x})) > \max(\{g_1(\vec{x}), \dots, g_{m-1}(\vec{x})\})$.

Lemma D.1.B says that \mathcal{A} is monotone in the second argument, so returning to equation $(*)$ and swapping out $\mathcal{A}(k+1, \max(\vec{x}))$ gives the first inequality here

$$\begin{aligned} \mathcal{A}(k+2, \max(\vec{x})) &> \mathcal{A}(k, \max(\{g_1(\vec{x}), \dots, g_{m-1}(\vec{x})\})) \\ &> h(g_0(\vec{x}), \dots, g_{m-1}(\vec{x})) = f(\vec{x}) \end{aligned}$$

and the second holds because the function h is level k . \square

- D.4 **LEMMA** If for some $k \in \mathbb{N}$ the functions g and h are level k , and if the function f is obtained by the schema of primitive recursion as

$$f(\vec{x}, y) = \begin{cases} g(\vec{x}) & \text{if } y = 0 \\ h(f(\vec{x}, z), \vec{x}, z) & \text{if } y = S(z) \end{cases}$$

then f is level $k+3$.

Proof Let n be such that $f: \mathbb{N}^{n+1} \rightarrow \mathbb{N}$, so that $g: \mathbb{N}^n \rightarrow \mathbb{N}$ and $h: \mathbb{N}^{n+2} \rightarrow \mathbb{N}$. The core of the argument is to show that this statement holds.

$$\forall k \ [\mathcal{A}(k, \max(\vec{x}) + y) > f(\vec{x}, y)] \quad (*)$$

We show this by induction on y . The $y = 0$ base step is that $\mathcal{A}(k, \max(\vec{x}) + 0) = \mathcal{A}(k, \max(\vec{x}))$ is greater than $f(\vec{x}, 0) = g(\vec{x})$ because g is level k .

For the inductive step assume that $(*)$ holds for $y = 0, \dots, y = z$ and consider the $y = z + 1$ case. The definition is that $\mathcal{A}(k + 1, \max(\vec{x}) + z + 1) = \mathcal{A}(k, \mathcal{A}(k + 1, \max(\vec{x}) + z))$. The second argument $\mathcal{A}(k + 1, \max(\vec{x}) + z)$ is larger than $\max(\vec{x}) + z$ by Lemma D.1.A, and so is larger than any x_i and larger than z , and is larger than $f(\vec{x}, z)$ by the inductive hypothesis.

$$\mathcal{A}(k + 1, \max(\vec{x}) + z) > \max(\{f(\vec{x}, z), x_0, \dots, x_{n-1}, z\})$$

Use Lemma D.1.B, monotonicity of \mathcal{A} in the second argument, and the fact that h is a level k function.

$$\begin{aligned} \mathcal{A}(k + 1, \max(\vec{x}) + z + 1) &= \mathcal{A}(k, \mathcal{A}(k + 1, \max(\vec{x}) + z)) \\ &> \mathcal{A}(k, \max(\{f(\vec{x}, z), x_0, \dots, x_{n-1}, z\})) \\ &> h(f(\vec{x}, z), \vec{x}, z) = f(\vec{x}, z + 1) \end{aligned}$$

That finishes the inductive verification of statement $(*)$.

To finish the argument, Lemma D.1.F gives that for all x_0, \dots, x_{n-1}, y

$$\begin{aligned} \mathcal{A}(k + 3, \max(\{x_0, \dots, y\})) &> \mathcal{A}(k + 1, 2 \cdot \max(\{x_0, \dots, y\})) \\ &\geq \mathcal{A}(k + 1, \max(\vec{x}) + y) \end{aligned}$$

(the latter holds because $2 \cdot \max(\vec{x}, y) \geq \max(\vec{x}) + y$ and because of Lemma D.1.B). In turn, by the first part of this proof, that is greater than $f(\vec{x}, y)$. \square

D.5 THEOREM (ACKERMANN, 1925) For each primitive recursive function f there is a number $k \in \mathbb{N}$ such that f is level k .

Proof The definition of primitive recursive functions Definition 3.7 specifies that each f is built from a set of initial function by the operations of composition and primitive recursion. With Lemma D.3 and Lemma D.4 we need only show that each initial operation is of some level.

The zero function $Z(x) = 0$ is level 0 since $\mathcal{A}(0, x) = x + 1 > 0$. The successor function $S(x) = x + 1$ is level 1 since $\mathcal{A}(1, x) > \mathcal{A}(0, x) = x + 1$ by Lemma D.1.E. Each projection function $I_i(x_0, \dots, x_i, \dots, x_{n-1}) = x_i$ is level 0 since $\mathcal{A}(0, \max(\vec{x})) = \max(\vec{x}) + 1$ is greater than $\max(\vec{x})$, which is greater than or equal to x_i . \square

D.6 COROLLARY The function \mathcal{A} is not primitive recursive.

Proof If \mathcal{A} were primitive recursive then it would be of some level, k_0 , so $\mathcal{A}(k_0, \max(\{x, y\})) > \mathcal{A}(x, y)$ for all x, y . Taking x and y to be k_0 gives a contradiction. \square

I.D Exercises

D.7 If expressed in base 10, how many digits are in $\mathcal{A}(4, 2) = 2^{65536} - 3$?

D.8 Show that for any k, y the evaluation of $\mathcal{A}(k, y)$ terminates.

D.9 Prove these parts of Lemma D.1. (A) Item b (B) Item c (C) Item d (D) Item e (E) Item f

D.10 Verify each identity. (A) $\mathcal{A}(0, y) = y + 1$ (B) $\mathcal{A}(1, y) = 2 + (y + 3) - 3$ (C) $\mathcal{A}(2, y) = 2 \cdot (y + 3) - 3$ (D) $\mathcal{A}(3, y) = 2y + 3 - 3$ (E) $\mathcal{A}(4, y) = 2 \uparrow\uparrow (n+3) - 3$

In the last one, the up-arrow notation (due to D Knuth) means that there is a power tower containing $n + 3$ many 2's. Recall that powers do not associate, so $2^{(2^2)} \neq (2^2)^2$; the notation means the first type of association, from the top down.

D.11 The prior exercise shows that at least the initial levels of \mathcal{A} are primitive recursive. In fact, all levels are. But how does that work: all the parts of \mathcal{A} are primitive recursive but as a whole it is not?

D.12 $\mathcal{A}(k + 1, x) = \mathcal{A}(k, \mathcal{A}(k, \dots \mathcal{A}(k, 1) \dots))$ where there are $x + 1$ -many \mathcal{A} 's.

D.13 Prove that $\mathcal{A}(k, y) = \mathcal{H}(k, 2, n + 3) - 3$. Conclude that \mathcal{H} is not primitive recursive.

EXTRA

I.E LOOP programs

The primitive recursive functions form a proper subset of the general recursive functions. The set of general recursive functions consists of all functions that are mechanically computable (under Church's Thesis), so that collection is easy to understand. We will now give a concrete way to understand the partial recursive functions.

Here is a Racket for loop,

```
(define (show-numbers)
  (for ([i '(1 2 3)])
    (display i)))
```

and what happens is what you'd think would happen.

```
> (show-numbers)
123
```

This is a Racket do loop (which is like a while in some other languages).

```
(define (wait-until-yes)
  (printf "Please enter 'yes'\n")
  (do () ; initialization variables (here, none)
        ((string=? (read-line) "yes") (printf "Thanks\n")) ; stop condition
        (printf "Enter exactly the string 'yes'\n")))) ; body of do loop
```

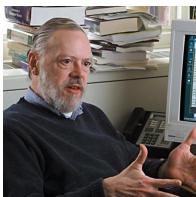
The difference between the two loop types is that in a for loop you know in advance the number of times that the machine will go through the code inside the loop (here, three times),[†] but a do allows the machine to go through its code an unbounded number of times.

[†]As long as you don't change the loop variable.

```
> (wait-until-yes)
Please enter 'yes'
yse
Enter exactly the string 'yes'
yes
Thanks
```

The next result says that a function is primitive recursive if and only if it can be computed using only for loops.

- E.1 THEOREM (MEYER AND RITCHIE, 1967) A function is primitive recursive if and only if it can be computed without using unbounded loops. More precisely, it is limited to loops where we can compute in advance, using only primitive recursive functions, how many iterations will occur.



Albert Meyer
b 1941 and Dennis Ritchie 1941-2011 (inventor of C)

We will show half of this, that if a function is primitive recursive then we can compute it using only bounded loops. We will do it by programming the primitive recursive functions in a language, called **LOOP**, that does not have unbounded loops. (Proof of the converse is outside our scope.)

Programs in LOOP execute on a machine model with registers r_0, r_1, \dots that hold natural numbers. There are four kinds of instructions, which we describe using r_0 and r_1 : (i) $r_0 = 0$ sets the contents of the register to zero, (ii) $r_0 = r_0 + 1$ increments the contents of the register, (iii) $r_1 = r_0$ copies the contents of r_0 into r_1 , leaving r_0 unchanged, and (iv) $\text{loop } r_0 \dots \text{end}$ executes a sequence of instructions repeatedly, with the number of repetitions given by the value of the register. For the last, the ‘ \dots ’ is a sequence that could contain any of the four kinds of statements, including that it could contain a nested loop (which might in turn contain its own nested loop, etc.).

An example is that the program below finishes with the register r_0 holding a value of 4, and with r_1 holding 2 (the indentation in the loop is just for visual clarity). Note that registers start with a value of zero, unless we preload the machine.

```
r1 = r1 + 1
r1 = r1 + 1
loop r1
    r0 = r0 + 1
    r0 = r0 + 1
end
```

Very important: changing the contents of the loop register inside of the loop does not change the number of times that the machine steps through that loop. Thus what's below is not an infinite loop. Instead, when the loop ends the value in r_0 will be twice what it was when the loop began.

```
loop r0
    r0 = r0 + 1
end
```

To interpret LOOP programs as computing functions we need a convention for input and output. Where the function takes n inputs, we will preload those inputs into the machine's first n registers. Similarly, where the function has m outputs we take those to be the final values of the first m registers.

With that convention, this LOOP program computes the two-input, one output addition function $\text{plus}(x, y) = x + y$.

```
# plus.loop  Return r0 + r1
loop r1
  r0 = r0 + 1
end
```

This book's source distribution comes with `loop.rkt`, a Racket program that interprets LOOP code. Here is an invocation running that code.[†]

```
jim@millstone:~/src/loop$ ./loop.rkt -f machines/plus.loop -p "3 2" -s
r0=3 r1=2
  --start loop of 2 repetitions--
r0=4 r1=2
r0=5 r1=2
  --end loop--
5
```

The program options are: `p` preloads the registers `r0` and `r1` with 3 and 2, while `s` shows the registers for each step of the computation. By default the simulator returns the value of the first register, here 5.

This LOOP program computes the multiplication function $\text{product}(x, y) = x \cdot y$.

```
# product.loop  Return r0 * r1
r2 = r1  # save the inputs in higher registers
r1 = r0  #
r0 = 0
loop r2
  loop r1
    r0 = r0 + 1
  end
end
```

It has a nested loop.

```
jim@millstone:~/src/loop$ ./loop.rkt -f machines/product.loop -p "3 2" -s
r0=3 r1=2
r0=3 r1=2 r2=2
r0=3 r1=3 r2=2
r0=0 r1=3 r2=2
  --start loop of 2 repetitions--
    --start loop of 3 repetitions--
      r0=1 r1=3 r2=2
      r0=2 r1=3 r2=2
      r0=3 r1=3 r2=2
    --end loop--
    --start loop of 3 repetitions--
      r0=4 r1=3 r2=2
      r0=5 r1=3 r2=2
      r0=6 r1=3 r2=2
    --end loop--
  --end loop--
```

6

Two more examples. This computes the predecessor function $\text{pred}(x)$,

[†]Running under Linux.

```
# pred.loop  Return r0 - 1 (or 0)
loop r0
  r2 = r1
  r1 = r1 + 1
  end
r0 = r2
```

which equals $x - 1$ unless x equals 0, when it equals 0.

```
jim@millstone:$ ./loop.rkt -f machines/pred.loop -p "3" -s
r0=3
  --start loop of 3 repetitions--
r0=3 r1=0 r2=0
r0=3 r1=1 r2=0
r0=3 r1=1 r2=1
r0=3 r1=2 r2=1
r0=3 r1=2 r2=2
r0=3 r1=3 r2=2
  --end loop--
r0=2 r1=3 r2=2
2
```

And this uses predecessor to compute proper subtraction $x - y$,

```
#proper-sub.loop  Return r0 - r1 (or 0)
loop r1
  r3 = 0
loop r0
  r2 = r3
  r3 = r3 + 1
  end
r0 = r2
end
```

which equals $x - y$ unless y is greater than x , when the outcome is 0.

```
jim@millstone:$ ./loop.rkt -f machines/proper-sub.loop -p "3 2" -s
r0=3 r1=2
  --start loop of 2 repetitions--
r0=3 r1=2 r3=0
  --start loop of 3 repetitions--
r0=3 r1=2 r2=0 r3=0
r0=3 r1=2 r2=0 r3=1
r0=3 r1=2 r2=1 r3=1
r0=3 r1=2 r2=1 r3=2
r0=3 r1=2 r2=2 r3=2
r0=3 r1=2 r2=2 r3=3
  --end loop--
r0=2 r1=2 r2=2 r3=3
r0=2 r1=2 r2=2 r3=0
  --start loop of 2 repetitions--
r0=2 r1=2 r2=0 r3=0
r0=2 r1=2 r2=0 r3=1
r0=2 r1=2 r2=1 r3=1
r0=2 r1=2 r2=1 r3=2
  --end loop--
r0=1 r1=2 r2=1 r3=2
  --end loop--
1
```

Finally, this illustrates a routine with more than one output.

```
# rotate-shift-right.loop  input x,y,z,  output z,x,y
r3 = r2
r2 = r1
```

```
r1 = r0
r0 = r3
```

The program's `o` option lets us show three registers instead of the default one

```
jim@millstone:$ ./loop.rkt -f machines/rotate-shift-right.loop -p "1 2 3" -o 3
3 1 2
```

(we've avoided showing the computation's steps by not using the `s` option).

We are now ready to prove that for each primitive recursive function there is a LOOP program that computes it. The strategy is to first show how to compute each initial function, and then show how to do the combining operations of function composition and primitive recursion.

The zero function $\mathcal{Z}(x) = 0$ is computed by the LOOP program whose single line is $r0 = 0$. The successor function $\mathcal{S}(x) = x + 1$ is computed by the one-line $r0 = r0 + 1$. Projection $\mathcal{I}_i(x_0, \dots x_i, \dots x_{n-1}) = x_i$ is computed by $r0 = ri$.

Composition of two functions is easy. Let $g(x_0, \dots x_n)$ and $f(y_0, \dots y_m)$ be computed by LOOP programs P_g and P_f . Suppose that the bookkeeping of the composition $f \circ g$ is right, that g is an m -output function to match the number of f 's inputs. Then concatenating the two programs, so that the instructions of P_g are just followed by the instructions of P_f , gives the desired LOOP program for composition, since it uses the output of g as input to compute the action of f .

General composition starts with

$$f(x_0, \dots x_n), \quad h_0(y_{0,0}, \dots y_{0,m_0}), \quad \dots \quad h_n(y_{n,0}, \dots y_{n,m_n})$$

and produces $f(h_0(y_{0,0}, \dots y_{0,m_0}), \dots h_n(y_{n,0}, \dots y_{n,m_n}))$. This needs a little more thought than the two-function case.

The issue is that were we to load the inputs $y_{0,0}, \dots y_{n,m_n}$ into the registers $r0, r1, \dots$ and then immediately begin computing h_0 , there would be a danger of overwriting the inputs for h_1 . (For instance, `rotate-shift-right.loop` above uses an extra register, $r3$, beyond those used to store inputs.) So we must move those inputs out of the way. Let $P_f, P_{h_0}, \dots P_{h_n}$ be LOOP programs to compute the functions. Each uses a limited number of registers and thus there is a number j so large that no program uses register j . By definition, the program P to compute the composition gets the sequence of inputs starting in the register numbered 0. The first step is to copy these inputs to start in the register j . Next, zero out the registers below register j , copy h_0 's arguments down to begin at $r0$, and run the program P_{h_0} . When it finishes, copy its output to the register numbered $j + m_0 + \dots + m_n + 1$. Do a similar thing for the other h_i 's. Finish by copying these outputs down to the initial registers, zeroing out the remaining registers, and running P_f .

The other combiner operation is primitive recursion.

$$f(x_0, \dots x_{k-1}, y) = \begin{cases} g(x_0, \dots x_{k-1}) & - \text{if } y = 0 \\ h(f(x_0, \dots x_{k-1}, z), x_0, \dots x_{k-1}, z) & - \text{if } y = S(z) \end{cases}$$

Suppose that we have LOOP programs P_g and P_h . The register swapping needed is similar to what happens for composition so we won't go through it. The

program P_f starts by running P_g . Then it sets a fresh register to 0; call that register t . Now it enters a loop based on the register y (that is, successive times through the loop count down as $y, y - 1$, etc.). The body of the loop computes $f(x_0, \dots x_{k-1}, t + 1) = h(f(x_0, \dots x_{k-1}, t), x_0, \dots x_{k-1}, t)$ by running P_h , and then incrementing t . That ends the argument.

We close with a remark on an interesting aspect of `loop . rkt`, the interpreter for LOOP. It works by replacing the C-like syntax used above with a LISP-ish one. For instance, the interpreter converts the string input on the left to the string on the right.

```
r1 = r1 + 1
loop r1
  r0 = r0 + 1
end
```

```
((incr r1) (loop r1 (incr r0)))
```

The advantage of this switch is that the parentheses automatically match the beginning of each `loop` with its `end` and thus we don't have to write into the interpreter some code including a stack to keep track of loop nesting. With the string on the right, `loop . rkt` computes the answer by running it through the `eval` command.

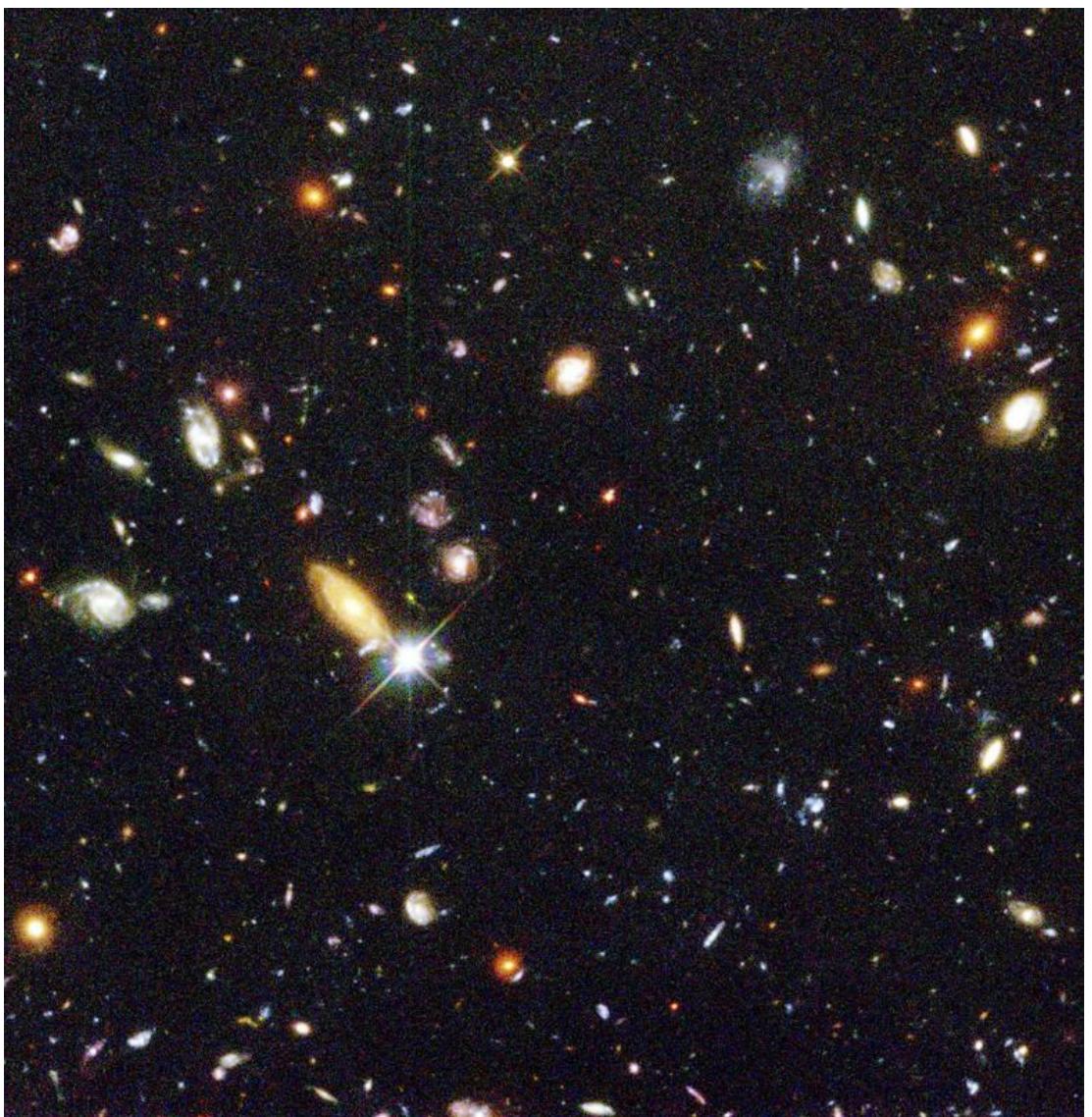
I.E Exercises

E.2 Write a LOOP program that inputs two numbers and swaps them, so that x, y becomes y, x .

E.3 Argue that the LOOP language would not gain strength if it were to allow statements like $r0 = r0 + 2$, or statements like $r0 = 1$.

E.4 The program `rotate-shift-right . loop` inputs three numbers, outputs three, and shifts the inputs right (with the third number ending in the first register). Write an three input/three output program that does a rotate shift left. Also write the program that composes the two. What does it compute?

E.5 In Ackermann's function, after the operations $\text{plus}(x, y)$ and $\text{product}(x, y)$ comes $\text{power}(x, y)$. Write a LOOP program for it.



CHAPTER

II Background

The first chapter began by saying that we are more interested in what things can be computed than in the details of how they are computed. We want to understand the set of functions that are effective, that are mechanically computable, which we formally defined as computable by a Turing machine. The major result of this chapter and the single most important result in the book is that there are functions that are uncomputable — there are functions for which there is no Turing machine that computes them. There are jobs that no machine can do.

SECTION

II.1 Infinity

We will show that there are more functions $f: \mathbb{N} \rightarrow \mathbb{N}$ than Turing machines and therefore there are functions with no associated machine.

Cardinality The set of functions and the set of Turing machines are both infinite. We will begin with two paradoxes that dramatize the challenge to our intuition posed by comparing the sizes of infinite sets. We will then produce the mathematics to resolve these puzzles, and apply it to the sets of functions and Turing machines.

The first puzzle is [Galileo's Paradox](#). It compares the size of the set of perfect squares with the size of the set of natural numbers. The first is a proper subset of the second. However, the figure below shows that the two sets can be made to correspond, to match element-to-element, so in this sense there are exactly as many squares as there are natural numbers.



Galileo Galilei
1564–1642

1.1 ANIMATION: Correspondence $n \leftrightarrow n^2$ between the natural numbers and the squares.

The second puzzle is [Aristotle's Paradox](#). On the left below are two circles. If we roll them through one revolution then the trail left by the smaller one is shorter. But if we put the smaller inside the larger and roll them, as in a train wheel, then they appear to leave equal-length trails.

IMAGE: This is the Hubble Deep Field image. It came from pointing the Hubble telescope to the darkest part of the sky, the very background, and soaking up light for eleven days. It covers an area of the sky about the same width as that of a dime viewed seventy five feet away. Every speck is a galaxy. There are thousand of them — there is a lot in the background. Robert Williams and the Hubble Deep Field Team (STScI) and NASA. (Also see the Deep Field movie.)

1.2 ANIMATION: Circles of different radiiuses have different circumferences.

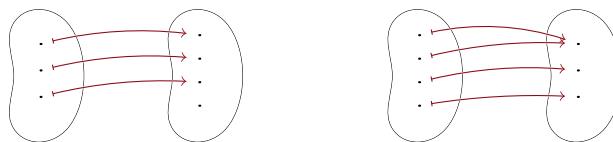
1.3 ANIMATION: Embedded circles rolling together.

As with Galileo's Paradox, we might think that the larger circle's set of points is in some way bigger. But the correct view is that the two sets have the same size, the same number of elements, in that they correspond. Point-for-point, the smaller circle matches the larger.

The animations below illustrate matching the points in two ways. The first shows them as nested circles, with points on the inside corresponding to points on the outside. The second straightens that out so that the circumferences make segments, and then for every point on the top there is a matching point on the bottom.

1.4 ANIMATION: Corresponding points on the circumferences $x \cdot (2\pi r_0) \leftrightarrow x \cdot (2\pi r_1)$.

Recall that mathematically a correspondence is a function that is both one-to-one and onto (Appendix B is a review). A function $f: D \rightarrow C$ is one-to-one if $f(x_0) = f(x_1)$ implies that $x_0 = x_1$ for $x_0, x_1 \in D$. It is onto if for any $y \in C$ there is an $x \in D$ such that $y = f(x)$. Below, the left map is one-to-one but not onto because there is a codomain element with no associated domain element. The right map is onto but not one-to-one because two inputs map to the same output.



1.5 LEMMA For a function with a finite domain, the number of elements in its domain is greater than or equal to the number of elements in its range. If the function is one-to-one then its domain has the same number of elements as its range, while if it is not one-to-one then its domain has more elements than its range. Consequently, two finite sets have the same number of elements if and only if they correspond, that is, if and only if there is a function from one to the other that is a correspondence.

Proof Exercise 1.49. □

- 1.6 LEMMA The relation between two sets of ‘there is a correspondence from one to the other’ is an equivalence relation.

Proof Fix sets S_0 and S_1 . Reflexivity is clear since a set corresponds to itself via the identity function. For symmetry, assume that there is a correspondence $f: S_0 \rightarrow S_1$ and recall that its inverse $f^{-1}: S_1 \rightarrow S_0$ exists and is a correspondence in the other direction. For transitivity, assume that there are correspondences $f: S_0 \rightarrow S_1$ and $g: S_1 \rightarrow S_2$ and recall also that the composition $g \circ f: S_0 \rightarrow S_2$ is a correspondence. \square

We now give that relation a name. This generalizes the observation of Lemma 1.5 about same-sized sets from the finite to the infinite.

- 1.7 DEFINITION Two sets have the **same cardinality** or are **equinumerous**, denoted $|S_0| = |S_1|$, if there is a correspondence between them.

- 1.8 EXAMPLE Galileo’s Paradox is that the set of squares $S = \{n^2 \mid n \in \mathbb{N}\}$ has the same cardinality as \mathbb{N} . The function $f: \mathbb{N} \rightarrow S$ given by $f(n) = n^2$ is one-to-one because if $f(x_0) = f(x_1)$ then $x_0^2 = x_1^2$ and thus, since these are natural numbers, $x_0 = x_1$. It is onto because any element of the codomain $y \in S$ is the square of some n from the domain \mathbb{N} , by the definition of S .

- 1.9 EXAMPLE Aristotle’s Paradox is that for $r_0, r_1 \in \mathbb{R}^+$, the interval $[0..2\pi r_0)$ has the same cardinality as the interval $[0..2\pi r_1)$. The map $g(x) = (2\pi r_1 / 2\pi r_0) \cdot x$ is a correspondence; verification is Exercise 1.43.

- 1.10 EXAMPLE The sets $S_0 = \{0, 1, 2, 3\}$ and $S_1 = \{10, 11, 12, 13\}$ have the same cardinality. One correspondence, from S_0 to S_1 , is $x \mapsto x + 10$.

- 1.11 EXAMPLE The set of natural numbers greater than zero, $\mathbb{N}^+ = \{1, 2, \dots\}$, has the same cardinality as \mathbb{N} . A correspondence is $f: \mathbb{N} \rightarrow \mathbb{N}^+$ given by $n \mapsto n + 1$.



Georg Cantor
1845–1918

Comparing the sizes of sets in this way was proposed by G Cantor in the 1870’s. As the paradoxes above dramatize, Definition 1.7 introduces a deep idea. We should convince ourselves that it captures what we mean by sets having the ‘same number’ of elements. One supporting argument is that it is the natural generalization of Lemma 1.5. A second is Lemma 1.6, that it partitions sets into classes so that inside of a class all of the sets have the same cardinality. That is, it gives the ‘equi’ in equinumerous. The most important supporting argument is that, as with Turing’s definition of his machine, Cantor’s definition is persuasive in itself. Gödel noted this, writing “Whatever ‘number’ as applied to infinite sets may mean, we certainly want it to have the property that the number of objects belonging to some class does not change if, leaving the objects the same, one changes in any way . . . e.g., their colors or their distribution in space . . . From this, however, it follows at once that two sets will have the same [cardinality] if their elements can be brought into one-to-one correspondence, which is Cantor’s definition.”

Comparing the sizes of sets in this way was proposed by G Cantor in the 1870’s. As the paradoxes above dramatize, Definition 1.7 introduces a deep idea. We should convince ourselves that it captures what we mean by sets having the ‘same number’ of elements. One supporting argument is that it is the natural generalization of Lemma 1.5. A second is Lemma 1.6, that it partitions sets into classes so that inside of a class all of the sets have the same cardinality. That is, it gives the ‘equi’ in equinumerous. The most important supporting argument is that, as with Turing’s definition of his machine, Cantor’s definition is persuasive in itself. Gödel noted this, writing “Whatever ‘number’ as applied to infinite sets may mean, we certainly want it to have the property that the number of objects belonging to some class does not change if, leaving the objects the same, one changes in any way . . . e.g., their colors or their distribution in space . . . From this, however, it follows at once that two sets will have the same [cardinality] if their elements can be brought into one-to-one correspondence, which is Cantor’s definition.”

- 1.12 **DEFINITION** A set is **finite** if it has the same cardinality as $\{0, 1, \dots, n\}$ for some $n \in \mathbb{N}$, or if it is empty. Otherwise it is **infinite**.

By far the most important infinite set is the natural numbers, $\mathbb{N} = \{0, 1, 2, \dots\}$.

- 1.13 **DEFINITION** A set with the same cardinality as the natural numbers is **countably infinite**. A set that is either finite or countably infinite is **countable**. If a set is the range of a function whose domain is the natural numbers then we say the function **enumerates**, or is an **enumeration of**, that set.[†]

The idea behind the term ‘enumeration’ is that $f: \mathbb{N} \rightarrow S$ lists its range: first $f(0)$, then $f(1)$, etc. (This listing might have repeats, where $f(n_0) = f(n_1)$ but $n_0 \neq n_1$.) We are often interested in enumerations that are **computable**.

- 1.14 **EXAMPLE** The set of multiples of three, $3\mathbb{N} = \{3k \mid k \in \mathbb{N}\}$, is countable. The natural map $g: \mathbb{N} \rightarrow 3\mathbb{N}$ is $g(n) = 3n$.

- 1.15 **EXAMPLE** The set $\mathbb{N} - \{2, 5\} = \{0, 1, 3, 4, 6, 7, \dots\}$ is countable. The function below, both formally defined and illustrated with a table, closes up the gaps.

$$f(n) = \begin{cases} n & - \text{if } n < 2 \\ n+1 & - \text{if } n \in \{2, 3\} \\ n+2 & - \text{if } n \geq 4 \end{cases}$$

n	$f(n)$	0	1	2	3	4	5	6	...
n	$f(n)$	0	1	3	4	6	7	8	...

This function is clearly both one-to-one and onto.

- 1.16 **EXAMPLE** The set of prime numbers P is countable. There is a function $p: \mathbb{N} \rightarrow P$ where $p(n)$ is the n -th prime, so that $p(0) = 2$, $p(1) = 3$, etc.

- 1.17 **EXAMPLE** Fix the set of symbols $\Sigma = \{a, \dots, z\}$. Consider the set of strings made of those symbols, such as az and $abba$. The set of all such strings, Σ^* , is countable. This table illustrates the correspondence that puts string in **lexicographic order**, where shorter strings come before longer ones and equal-length strings come in alphabetical order. (The first entry is the empty string, $\varepsilon = ''$.)

$n \in \mathbb{N}$	0	1	2	...	26	27	28	...
$f(n) \in \Sigma^*$	ε	a	b	...	z	aa	ab	...

- 1.18 **EXAMPLE** The set of integers $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ is countable. The natural correspondence alternates between positive and negative numbers.

$n \in \mathbb{N}$	0	1	2	3	4	5	6	...
$f(n) \in \mathbb{Z}$	0	+1	-1	+2	-2	+3	-3	...

We close this section by circling back to the paradoxes of infinity. In the prior example, a naive expectation might be that the positives and the negatives combine make \mathbb{Z} somehow twice as big as \mathbb{N} . But this is the point of Galileo’s Paradox: the

[†]The ‘a function whose domain is the natural numbers’ might seem to imply that the function is total but in section 7 we will apply this to some functions that are not total, that are not defined on some natural numbers.

correct way to measure how many elements a set has is not through superset and subset but through cardinality.

Finally, we will mention one more paradox, due to Zeno (circa 450 BC). He imagined a tortoise challenging swift Achilles to a race, asking only for a head start. Achilles laughs but the tortoise says that by the time Achilles reaches the spot x_0 of the head start, the tortoise will have moved on to some x_1 . On reaching x_1 , Achilles will find that the tortoise is ahead at x_2 . For any x_i , Achilles will always be behind and so, the tortoise reasons, Achilles can never get ahead. The heart of this argument is that while the distances $x_{i+1} - x_i$ shrink toward zero, there is always further to go because of the open-endedness at the left of the interval $(0.. \infty)$.



1.19 FIGURE: Zeno of Elea shows Youths the Doors to Truth and False, by covering half the distance to the door, and then half of that, etc. (By either B Carducci (1560–1608) or P Tibaldi (1527–1596).)

Zeno's Paradox is not directly connected to the material of this section. But it works by leveraging open-endedness and in this chapter we will often give arguments that use the unboundedness of the natural numbers, that is, that leverage the open-endedness of \mathbb{N} at infinity.

II.1 Exercises

- ✓ 1.20 Verify Example 1.14, that the function $g: \mathbb{N} \rightarrow \{3k \mid k \in \mathbb{N}\}$ given by $n \mapsto 3n$ is both one-to-one and onto.
- 1.21 A friend tells you, “The perfect squares and the perfect cubes have the same number of elements because these sets are both one-to-one and onto.” Straighten them out.
- 1.22 Let $f, g: \mathbb{Z} \rightarrow \mathbb{Z}$ be $f(x) = 2x$ and $g(x) = 2x - 1$. Give a proof or a counterexample for each. (A) If f one-to-one? Is it onto? (B) If g one-to-one? Onto? (C) Are f and g inverse to each other?
- ✓ 1.23 Decide if each function is one-to-one, onto, both, or neither. You cannot just answer ‘yes’ or ‘no’, you must justify the answer.
 - (A) $f: \mathbb{N} \rightarrow \mathbb{N}$ given by $f(n) = n + 1$
 - (B) $f: \mathbb{Z} \rightarrow \mathbb{Z}$ given by $f(n) = n + 1$
 - (C) $f: \mathbb{N} \rightarrow \mathbb{N}$ given by $f(n) = 2n$
 - (D) $f: \mathbb{Z} \rightarrow \mathbb{Z}$ given by $f(n) = 2n$
 - (E) $f: \mathbb{Z} \rightarrow \mathbb{N}$ given by $f(n) = |n|$.

1.24 Decide if each is a correspondence (you must also verify): (A) $f: \mathbb{Q} \rightarrow \mathbb{Q}$ given by $f(n) = n + 3$ (B) $f: \mathbb{Z} \rightarrow \mathbb{Q}$ given by $f(n) = n + 3$ (C) $f: \mathbb{Q} \rightarrow \mathbb{N}$ given by $f(a/b) = |a \cdot b|$.

1.25 Decide if each set is finite or infinite and justify your answer. (A) $\{1, 2, 3\}$ (B) $\{0, 1, 4, 9, 16, \dots\}$ (C) the set of prime numbers (D) the set of real roots of $x^5 - 5x^4 + 3x^2 + 7$

1.26 Show that each pair of sets has the same cardinality by producing a one-to-one and onto function from one to the other. You must verify that the function is a correspondence. (A) $\{0, 1, 2\}, \{3, 4, 5\}$ (B) $\mathbb{Z}, \{i^3 \mid i \in \mathbb{Z}\}$

✓ 1.27 Show that each pair of sets has the same cardinality by producing a correspondence (you must verify that the function is a correspondence): (A) $\{0, 1, 3, 7\}$ and $\{\pi, \pi + 1, \pi + 2, \pi + 3\}$ (B) the even natural numbers and the perfect squares (C) the real intervals $(1..4)$ and $(-1..1)$.

✓ 1.28 Verify that the function $f(x) = 1/x$ is a correspondence between the subsets $(0..1)$ and $(1..\infty)$ of \mathbb{R} .

1.29 Give a formula for a correspondence between the sets $\{1, 2, 3, 4, \dots\}$ and $\{7, 10, 13, 16, \dots\}$.

✓ 1.30 Consider the set of characters $C = \{0, 1, \dots, 9\}$ and the set of integers $A = \{48, 49, \dots, 57\}$.

(A) Produce a correspondence $f: C \rightarrow A$.

(B) Verify that the inverse $f^{-1}: A \rightarrow C$ is also a correspondence.

✓ 1.31 Show that each pair of sets have the same cardinality. You must give a suitable function and also verify that it is one-to-one and onto. (A) \mathbb{N} and the set of even numbers (B) \mathbb{N} and the odd numbers (C) the even numbers and the odd numbers

✓ 1.32 Although sometimes there is a correspondence that is natural, correspondences need not be unique. Produce the natural correspondence from $(0..1)$ to $(0..2)$, and then produce a different one, and then another different one.

1.33 Example 1.8 gives one correspondence between the natural numbers and the perfect squares. Give another.

1.34 Fix $c \in \mathbb{R}$ such that $c > 1$. Show that $f: \mathbb{R} \rightarrow (0..\infty)$ given by $x \mapsto c^x$ is a correspondence.

1.35 Show that the set of powers of two $\{2^k \mid k \in \mathbb{N}\}$ and the set of powers of three $\{3^k \mid k \in \mathbb{N}\}$ have the same cardinality. Generalize.

1.36 For each, give functions from \mathbb{N} to itself. You must justify your claims. (A) Give two examples of functions that are one-to-one but not onto. (B) Give two examples of functions that are onto but not one-to-one. (C) Give two that are neither. (D) Give two that are both.

1.37 Show that the intervals $(3..5)$ and $(-1..10)$ of real numbers have the same cardinality by producing a correspondence. Then produce a second one.

- 1.38 Show that the sets have the same cardinality. (A) $\{4k \mid k \in \mathbb{N}\}$, $\{5k \mid k \in \mathbb{N}\}$
 (B) $\{0, 1, \dots, 99\}$, $\{m \in \mathbb{N} \mid m^2 < 10000\}$ (C) $\{0, 1, 3, 6, 10, 15, \dots\}$, \mathbb{N}
- ✓ 1.39 Produce a correspondence between each pair of open intervals of reals.
 (A) $(0..1)$, $(0..2)$
 (B) $(0..1)$, $(a..b)$ for real numbers $a < b$
 (C) $(0..\infty)$, $(a..\infty)$ for the real number a
 (D) This shows a correspondence $x \mapsto f(x)$ between a finite interval of reals and an infinite one, $f: (0..1) \rightarrow (0..\infty)$.
- 
- The point P is at $(-1, 1)$. Give a formula for f .
- ✓ 1.40 Not every set containing irrational numbers is uncountable. Show that the set $S = \{\sqrt[n]{2} \mid n \in \mathbb{N} \text{ and } n \geq 2\}$ is countable.
- 1.41 Let \mathbb{B} be the set of characters from which bit strings are made, $\mathbb{B} = \{0, 1\}$.
 (A) Let B be the set of finite bit strings where the initial bit is 1. Show that B is countable. (B) Let \mathbb{B}^* be the set of finite bit strings, without the restriction on the initial bit. Show that it also is countable. Hint: use the prior item.
- 1.42 Use the arctangent function to prove that the sets $(0..1)$ and \mathbb{R} have the same cardinality.
- 1.43 Example 1.9 restates Aristotle's Paradox as: the intervals $I_0 = [0..2\pi r_0]$ and $I_1 = [0..2\pi r_1]$ have the same cardinality, for $r_0, r_1 \in \mathbb{R}^+$.
 (A) Verify it by checking that $g: I_0 \rightarrow I_1$ given by $g(x) = x \cdot (r_1/r_0)$ is a correspondence.
 (B) Show that where $a < b$, the cardinality of $[0..1]$ equals that of $[a..b]$.
 (C) Generalize by showing that where $a < b$ and $c < d$, the real intervals $[a..b]$ and $[c..d]$ have the same cardinality.
- 1.44 Suppose that $D \subseteq \mathbb{R}$. A function $f: D \rightarrow \mathbb{R}$ is **strictly increasing** if $x < \hat{x}$ implies that $f(x) < f(\hat{x})$ for all $x, \hat{x} \in D$. Prove that any strictly increasing function is one-to-one; it is therefore a correspondence between D and its range. (The same applies if the function is strictly decreasing.) Does this hold for $D \subseteq \mathbb{N}$?
- ✓ 1.45 A paradoxical aspect of both Aristotle's and Galileo's examples is that they gainsay Euclid's "the whole is greater than the part," because they name sets where that set is equinumerous with a proper subset. Here, show that each pair of a set and a proper subset has the same cardinality. (A) \mathbb{N} , $\{2n \mid n \in \mathbb{N}\}$
 (B) \mathbb{N} , $\{n \in \mathbb{N} \mid n > 4\}$
- 1.46 Example 1.15 illustrates that we can take away a finite number of elements from the set \mathbb{N} without changing the cardinality. Prove that if S is a finite subset of \mathbb{N} then $\mathbb{N} - S$ is countable.

- 1.47 (A) Let $D = \{0, 1, 2, 3\}$ and $C = \{\text{Spades, Hearts, Clubs, Diamonds}\}$, and let $f: D \rightarrow C$ be given by $f(0) = \text{Spades}$, $f(1) = \text{Hearts}$, $f(2) = \text{Clubs}$, $f(3) = \text{Diamonds}$. Find the inverse function $f^{-1}: C \rightarrow D$ and verify that it is a correspondence.
- (B) Let $f: D \rightarrow C$ be a correspondence. Show that the inverse function exists. That is, show that associating each $y \in C$ with the $x \in D$ such that $f(x) = y$ gives a well-defined function $f^{-1}: C \rightarrow D$.
- (C) Show that the inverse of a correspondence is also a correspondence, that the function defined in the prior item is a correspondence.
- 1.48 Prove that a set S is infinite if and only if it has the same cardinality as a proper subset of itself.
- 1.49 Prove Lemma 1.5 by proving each.
- (A) For any function with a finite domain, the number of elements in that domain is greater than or equal to the number of elements in the range. *Hint:* use induction on the number of elements in the domain.
- (B) If such a function is one-to-one then its domain has the same number of elements as its range. *Hint:* again use induction on the size of the domain.
- (C) If it is not one-to-one then its domain has more elements than its range.
- (D) Two finite sets have the same number of elements if and only if there is a correspondence from one to the other.

SECTION

II.2 Cantor's correspondence

Countability is a property of sets so we can ask how it interacts with set operations.

We start with the cross product operation, in part because we will want to count Turing machines, which are sets of four-tuples, $\mathcal{P} \in \mathcal{P}(\mathbb{N} \times \Sigma \times \Sigma \cup \{\text{L, R}\} \times \mathbb{N})$.

- 2.1 EXAMPLE The set $S = \{0, 1\} \times \mathbb{N}$ consists of ordered pairs $\langle i, j \rangle$ where $i \in \{0, 1\}$ and $j \in \mathbb{N}$. The diagram below shows two columns, each of which looks like the natural numbers in that it is discrete and unbounded in one direction. So informally, S is twice the natural numbers. As in Galelio's Paradox this might lead to a mistaken guess that it has more members than \mathbb{N} . But S is countable.

To count it, the key is to alternate between columns.

2.2 ANIMATION: Counting $S = \{0, 1\} \times \mathbb{N}$.

Here is that correspondence as a table.

$n \in \mathbb{N}$	0	1	2	3	4	5	...
$\langle i, j \rangle \in S$	$\langle 0, 0 \rangle$	$\langle 1, 0 \rangle$	$\langle 0, 1 \rangle$	$\langle 1, 1 \rangle$	$\langle 0, 2 \rangle$	$\langle 1, 2 \rangle$...

The map from the table's top row to the bottom is a pairing function. Its inverse, from bottom to top, is an unpairing function. This counting technique extends to three copies, $\{0, 1, 2\} \times \mathbb{N}$, to four copies, etc.

- 2.3 **LEMMA** The cross product of two finite sets is finite, and therefore countable. The cross product of a finite set and a countably infinite set, or of a countably infinite set and a finite set, is countably infinite.

Proof Exercise 2.41; use the above example as a model. \square

- 2.4 **EXAMPLE** The obvious next set to consider is the cross product of the two countably infinite sets $\mathbb{N} \times \mathbb{N}$. In the informal language of the prior example we can think of it as infinitely many column copies of the natural numbers.

⋮	⋮	⋮	⋮	⋮
$\langle 0, 3 \rangle$	$\langle 1, 3 \rangle$	$\langle 2, 3 \rangle$	$\langle 3, 3 \rangle$...
$\langle 0, 2 \rangle$	$\langle 1, 2 \rangle$	$\langle 2, 2 \rangle$	$\langle 3, 2 \rangle$...
$\langle 0, 1 \rangle$	$\langle 1, 1 \rangle$	$\langle 2, 1 \rangle$	$\langle 3, 1 \rangle$...
$\langle 0, 0 \rangle$	$\langle 1, 0 \rangle$	$\langle 2, 0 \rangle$	$\langle 3, 0 \rangle$...

Sticking to a single column or row won't work so here also we need to alternate. Starting from the lower left, do a breadth-first traversal: after $\langle 0, 0 \rangle$, next take pairs that are one away, $\langle 1, 0 \rangle$ and $\langle 0, 1 \rangle$, then those that are two away, $\langle 2, 0 \rangle$, $\langle 1, 1 \rangle$ and $\langle 0, 2 \rangle$, etc.

2.5 ANIMATION: Counting $\mathbb{N} \times \mathbb{N}$.

- 2.6 **DEFINITION** **Cantor's correspondence** $\text{cantor}: \mathbb{N}^2 \rightarrow \mathbb{N}$ or **unpairing function**, is the above correspondence. Its inverse $\text{cantor}^{-1}: \mathbb{N} \rightarrow \mathbb{N}^2$ is **Cantor's pairing function**. (A notation for $\text{cantor}(x, y)$ common elsewhere is $\langle x, y \rangle$.)

Here is the same correspondence as a table.

$n \in \mathbb{N}$	0	1	2	3	4	5	6	...
$\langle x, y \rangle \in \mathbb{N}^2$	$\langle 0, 0 \rangle$	$\langle 0, 1 \rangle$	$\langle 1, 0 \rangle$	$\langle 0, 2 \rangle$	$\langle 1, 1 \rangle$	$\langle 2, 0 \rangle$	$\langle 0, 3 \rangle$...

Clearly this function is a correspondence. It is clearly also effective, meaning that we can write a program to compute it.

- 2.7 REMARK There is also a simple formula for the unpairing function. It is amusing so we will produce it. Give the diagonals numbers.

⋮	⋮	⋮		
$\langle 0, 3 \rangle$	$\langle 1, 3 \rangle$	$\langle 2, 3 \rangle$	$\langle 3, 3 \rangle$	
$\langle 0, 2 \rangle$	$\langle 1, 2 \rangle$	$\langle 2, 2 \rangle$	$\langle 3, 2 \rangle$...
$\langle 0, 1 \rangle$	$\langle 1, 1 \rangle$	$\langle 2, 1 \rangle$	$\langle 3, 1 \rangle$...
$\langle 0, 0 \rangle$	$\langle 1, 0 \rangle$	$\langle 2, 0 \rangle$	$\langle 3, 0 \rangle$...
<i>Diagonal</i>	0	1	2	3

We first walk through finding the number of the pair $\langle 1, 2 \rangle$. It is on diagonal 3. Prior to that diagonal comes six pairs: diagonal 0 has a single entry, diagonal 1 has two entries, and diagonal 2 has three entries. Thus, because the counting starts at zero, diagonal 3's initial pair $\langle 0, 3 \rangle$ is number 6. With that, $\langle 1, 2 \rangle$ is number 7.

To find the number corresponding to $\langle x, y \rangle$, note first that it lies on diagonal $d = x + y$. Prior to diagonal d comes $1 + 2 + \dots + d$ pairs, which is an arithmetic series with total $d(d + 1)/2$. So on diagonal d the first pair, $\langle 0, x + y \rangle$, has number $(x + y)(x + y + 1)/2$. Next on that diagonal, $\langle 1, x + y - 1 \rangle$ gets the number $1 + [(x + y)(x + y + 1)/2]$, etc. In general, $\text{cantor}(x, y) = x + [(x + y)(x + y + 1)/2]$.

- 2.8 EXAMPLE Two early examples are $\text{cantor}(1, 2) = 7$ and $\text{cantor}(6, 2) = 42$. A later one is $\text{cantor}(0, 36) = 666$.

- 2.9 LEMMA The cross product $\mathbb{N} \times \mathbb{N}$ is countable, for instance under Cantor's correspondence $\text{cantor}: \mathbb{N}^2 \rightarrow \mathbb{N}$. Further, the sets $\mathbb{N}^3 = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, and \mathbb{N}^4, \dots are all countable.

Proof The function $\text{cantor}: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ is one-to-one and onto by construction. That is, the construction ensures that each output natural number is associated with one and only one input pair.

The prior paragraph with domain \mathbb{N}^2 forms the base step of an induction argument. To do \mathbb{N}^3 the idea is to take a triple $\langle x, y, z \rangle$ to be a pair whose first entry is a pair, $\langle \langle x, y \rangle, z \rangle$. More formally, define $\text{cantor}_3: \mathbb{N}^3 \rightarrow \mathbb{N}$ by $\text{cantor}_3(x, y, z) = \text{cantor}(\text{cantor}(x, y), z)$. Exercise 2.35 shows that this function is a correspondence. With that, the details of the full induction are routine. \square

- 2.10 COROLLARY The cross product of finitely many countable sets is countable.

Proof Suppose that S_0, \dots, S_{n-1} are countable and that each function $f_i: \mathbb{N} \rightarrow S_i$ is a correspondence. By the prior result, the function $\text{cantor}_n^{-1}: \mathbb{N} \rightarrow \mathbb{N}^n$ is a correspondence. Write $\text{cantor}_n^{-1}(k) = \langle k_0, k_1, \dots, k_{n-1} \rangle$. Then the composition $k \mapsto \langle f_0(k_0), f_1(k_1), \dots, f_{n-1}(k_{n-1}) \rangle$ from \mathbb{N} to $S_0 \times \dots \times S_{n-1}$ is a correspondence. Thus $S_0 \times S_1 \times \dots \times S_{n-1}$ is countable. \square

- 2.11 EXAMPLE Also countable is the set of rational numbers \mathbb{Q} . We have already used the technique of counting by alternating between positives and negatives. So it suffices to count the nonnegative rationals with some $f: \mathbb{N} \rightarrow \mathbb{Q}^+ \cup \{0\}$. A nonnegative rational number is a numerator-denominator pair $\langle n, d \rangle \in \mathbb{N} \times \mathbb{N}^+$, but the complication is that some pairs collapse, such as that $n = 10$ and $d = 5$ is the same rational as $n = 2$ and $d = 1$.

We will describe f with a program instead of a formula. Suppose that it is given an input i . It will use its prior values $f(0), f(1), \dots, f(i-1)$. It does that by looping, using cantor^{-1} to generate pairs: first $\text{cantor}^{-1}(0) = \langle 0, 0 \rangle$, then $\text{cantor}^{-1}(1) = \langle 0, 1 \rangle$, then $\text{cantor}^{-1}(2) = \langle 1, 0 \rangle, \dots$. For each such $\langle a, b \rangle$, if the second entry is 0 or if the rational number a/b is among the prior values then the program skips this pair and goes on to the next loop iteration, to generated a new candidate pair. Eventually it generates an acceptable $\langle \hat{a}, \hat{b} \rangle$ and that is $f(i)$.

Having a program save prior values to use later is **memoization** or **caching**. It is widely used. For example, when you visit a web site your browser saves any images to your disk so that the next time you visit the site, if the image has not changed then the browser reuses the prior copy, reducing download time. The next result also uses memoization.

- 2.12 LEMMA A set S is countable if and only if either S is empty or there is an onto map $f: \mathbb{N} \rightarrow S$.

Proof Assume first that S is countable. If it is empty then we are done. If it is finite but nonempty, $S = \{s_0, \dots, s_{n-1}\}$, then this map is onto.

$$f(i) = \begin{cases} s_i & - \text{if } i < n \\ s_0 & - \text{otherwise} \end{cases}$$

If S is infinite and countable then it has the same cardinality as \mathbb{N} so there is a correspondence $f: \mathbb{N} \rightarrow S$. Correspondences are onto.

For the converse assume that either S is empty or there is an onto map from \mathbb{N} to S . Definition 1.13 says that an empty set is countable so what's left is to consider an onto map $f: \mathbb{N} \rightarrow S$. If S is finite then it is countable we are down to the case where S is infinite. Define $\hat{f}: \mathbb{N} \rightarrow S$ by $\hat{f}(n) = f(k)$ where k is the least natural number such that $f(k) \notin \{\hat{f}(0), \dots, \hat{f}(n-1)\}$. Such a k exists because S is infinite and f is onto. Observe that \hat{f} is both one-to-one and onto, by construction. \square

- 2.13 COROLLARY (1) Any subset of a countable set is countable. (2) The intersection of two countable sets is countable. The intersection of any number of countable sets is countable. (3) The union of two countable sets is countable. The union of any finite number of countable sets is countable. The union of countably many countable sets is countable.

Proof Suppose that S is countable and that $\hat{S} \subseteq S$. If S is empty then so is \hat{S} , and thus it is countable. Otherwise by the prior lemma there is an onto $f: \mathbb{N} \rightarrow S$. If \hat{S} is empty then it is countable, and if not then fix some $\hat{s} \in \hat{S}$. Then this map

$\hat{f}: \mathbb{N} \rightarrow \hat{S}$ is onto.

$$\hat{f}(n) = \begin{cases} f(n) & - \text{if } f(n) \in \hat{S} \\ \hat{s} & - \text{otherwise} \end{cases}$$

Item (2) is immediate from (1) since the intersection is a subset of both sets.

Now item (3). In the two-set case suppose that S_0 and S_1 are countable. If either set is empty, or both, then the result is trivial because for instance $S_0 \cup \emptyset = S_0$. Otherwise, suppose that $f_0: \mathbb{N} \rightarrow S_0$ and $f_1: \mathbb{N} \rightarrow S_1$ are onto. Then count by alternating between the two sets. More precisely, Lemma 2.3 gives a correspondence $g: \mathbb{N} \rightarrow \{0, 1\} \times \mathbb{N}$ and this is a function that is onto the set $S_0 \cup S_1$.

$$f_2(n) = \begin{cases} f_0(j) & - \text{if } g(n) = \langle 0, j \rangle \\ f_1(j) & - \text{if } g(n) = \langle 1, j \rangle \end{cases}$$

This approach extends to any finite number of countable sets.

Finally, we start with countably many countable sets, S_i for $i \in \mathbb{N}$, and show that their union $S_0 \cup S_1 \cup \dots$ is countable. If all but finitely many are empty then we can fall back to the finite case so instead suppose that infinitely many of the sets are nonempty. Throw out the empty ones because they don't affect the union, write \hat{S}_j for the remaining sets, and assume that we have a family of correspondences $g_j: \mathbb{N} \rightarrow \hat{S}_j$. Then use Cantor's pairing function: the desired map from \mathbb{N} onto $S_0 \cup S_1 \cup \dots$ is $\hat{g}(n) = g_j(k)$ where $\text{cantor}^{-1}(n) = \langle j, k \rangle$. \square

Returning to Lemma 2.3 and Lemma 2.9 on the cross product of countable sets, notice that they are effectivizable. That is, if sets correspond to \mathbb{N} via some effective function then their cross product corresponds to \mathbb{N} via an effective function. We finish this section by using that to effectively number the Turing machines.

Each Turing machine instruction is a four-tuple, a member of $Q \times \Sigma \times (\Sigma \cup \{\text{L}, \text{R}\}) \times Q$, where Q is the set of states and Σ is the tape alphabet. The results of this section imply that we can enumerate the set of instructions, so that there is an instruction that corresponds to 0, one corresponding to 1, etc. This enumeration is effective—there is a program that takes in a natural number and outputs the corresponding instruction, as well as a program that takes in an instruction and outputs the corresponding number.

With that, we can effectively number the Turing machines. The exact numbering that we use doesn't matter much as long as it has the properties in the definition below.

But for illustration here is one way: starting with a Turing machine \mathcal{P} , use the prior paragraph to convert each of its instructions to a number, giving a set $\{i_0, i_1, \dots, i_n\}$. Then define the number e associated with \mathcal{P} to be the one that when written in binary has 1 in bits i_0, \dots, i_n , that is, $e = 2^{i_0} + 2^{i_1} + \dots + 2^{i_n}$. This association is effective. Its inverse is also effective: given $e \in \mathbb{N}$, represent it as $e = 2^{j_0} + \dots + 2^{j_k}$ and the set of instructions corresponding to the numbers j_0, \dots, j_k is the desired Turing machine. (Except that we must first check that

the instruction set is deterministic, that no two of the instructions begin with the same $q_p T_p$. We can check this effectively and if it is not true then let the machine associated with e be empty, $\mathcal{P} = \{ \}$.)

- 2.14 **DEFINITION** A **numbering** is a function that assigns to each Turing machine a natural number. A numbering is **acceptable** if it is effective: (1) there is a program that takes as input the set of instructions and gives as output the associated number, (2) the set of numbers for which there is an associated machine is computable, and (3) there is an effective inverse that takes as input a natural number and gives as output the associated machine.

For the rest of the book we will just fix a numbering and cite its properties rather than deal with its details. We call this the machine's **index number** or **Gödel number**. For the machine with index $e \in \mathbb{N}$ we write \mathcal{P}_e . For the function computed by \mathcal{P}_e we write ϕ_e .

Think of the machine's index as its name. We will refer to the index frequently, for instance by saying "the e -th Turing machine." The takeaway point is that because the numbering is acceptable we can go effectively from the machine's index to its source, the four-tuple instructions, or from the source to the index. The index is computationally equivalent to the source.

- 2.15 **REMARK** Here is an informal alternative index-source correspondence that can give some intuition about numbering. On a computer, a program's source code is saved as a bitstring. We can interpret this as a binary number. In the other direction, given a bitstring, we can disassemble it into an assembly code source. (This scheme is troublesome to make precise. One problem is what to do about issues such as zero extending to a machine's word length. Another question is what happens when we roundtrip from the source to the number and back; do we return to the original? This idea is helpful but best left informal.)

- 2.16 **LEMMA (PADDING LEMMA)** Every computable function has infinitely many indices: if f is computable then there are infinitely many distinct $e_i \in \mathbb{N}$ with $f = \phi_{e_0} = \phi_{e_1} = \dots$. We can effectively produce a list of such indices.

Proof Let $f = \phi_e$. Let q_j be the highest-numbered state in \mathcal{P}_e . For each $k \in \mathbb{N}^+$ consider the Turing machine obtained from \mathcal{P}_e by adding the instruction $q_{j+k} BB q_{j+k}$. This gives an effective sequence of Turing machines $\mathcal{P}_{e_1}, \mathcal{P}_{e_2}, \dots$ with distinct indices, all having the same behavior, $\phi_{e_k} = \phi_e = f$. \square

- 2.17 **REMARK** In programming terms, the lemma says that for any compiled behavior there are infinitely many different source codes. One way to get them is by starting with a single source code and padding it by adding to the bottom a comment line and varying the line's contents.

With the ability to number machines, we are set up for this book's most important result. The next section shows that while Turing machines can be counted, natural number functions $f : \mathbb{N} \rightarrow \mathbb{N}$ cannot. This will prove that there are functions that are not computable.

II.2 Exercises

- ✓ 2.18 Extend the table of Example 2.1 through $n = 12$. Where $f(n) = \langle x, y \rangle$, give formulas for x and y .
- ✓ 2.19 For each pair $\langle a, b \rangle$ find the pair before it and the pair after it in Cantor's correspondence. That is, where $\text{cantor}(a, b) = n$, find the pair associated with $n+1$ and the pair with $n-1$. (A) $\langle 50, 50 \rangle$ (B) $\langle 100, 4 \rangle$ (C) $\langle 4, 100 \rangle$ (D) $\langle 0, 200 \rangle$ (E) $\langle 200, 0 \rangle$
- ✓ 2.20 Corollary 2.13 says that the union of two countable sets is countable.
 - (A) For each of the two sets $T = \{2k \mid k \in \mathbb{N}\}$ and $F = \{5m \mid m \in \mathbb{N}\}$ produce a correspondence $f_T: \mathbb{N} \rightarrow T$ and $f_F: \mathbb{N} \rightarrow F$. Give a table listing the values of $f_T(0), \dots, f_T(9)$ and give another table listing $f_F(0), \dots, f_F(9)$.
 - (B) Give a table listing the first ten values for a correspondence $f: \mathbb{N} \rightarrow T \cup F$.
- 2.21 Give an enumeration of $\mathbb{N} \times \{0, 1\}$. Find the pair matching 0, 10, 100, and 101. Find the number corresponding to $\langle 2, 1 \rangle$, $\langle 20, 1 \rangle$, and $\langle 200, 1 \rangle$.
- ✓ 2.22 Example 2.1 says that the method for two columns extends to three. Give an enumeration of $\{0, 1, 2\} \times \mathbb{N}$. That is, where $g(n) = \langle x, y \rangle$ give a formula for x and y . Find the pair corresponding to 0, 10, 100, and 1000. Find the number corresponding to $\langle 1, 2 \rangle$, $\langle 1, 20 \rangle$, and $\langle 1, 200 \rangle$.
- 2.23 Give an enumeration f of $\{0, 1, 2, 3\} \times \mathbb{N}$. That is, where $f(n) = \langle x, y \rangle$, give a formula for x and y . Also give an enumeration f of $\{0, 1, 2, \dots, k\} \times \mathbb{N}$.
- ✓ 2.24 Extend the table of Example 2.4 to cover correspondences up to 16.
- 2.25 Definition 2.6's function $\text{cantor}(x, y) = x + [(x+y)(x+y+1)/2]$ is clearly effective since it is given as a formula. Show that its inverse, pair: $\mathbb{N} \rightarrow \mathbb{N}^2$, is also effective by sketching a way to compute it with a program.
- 2.26 Prove that if A and B are countable sets then their symmetric difference $A \Delta B = (A - B) \cup (B - A)$ is countable.
- 2.27 Show that the subset $S = \{a + bi \mid a, b \in \mathbb{Z}\}$ of the complex numbers is countable.
- 2.28 List the first dozen nonnegative rational numbers enumerated by the method described in Example 2.11.
- ✓ 2.29 Use Lemma 2.12 to give a much slicker, and shorter, proof that the rational numbers are countable than the one in Example 2.11.
- 2.30 Let S be countably infinite and let $T \subset S$ be finite.
 - (A) Show that $S - T$ is countable.
 - (B) Show that $S - T$ is countably infinite.
 - (C) Can there be an infinite subset T so that $S - T$ is infinite?
- 2.31 Show that every infinite set contains a countably infinite set.
- 2.32 A **binary sequence** is an infinite bitstring, so we can think of it as a list $b = \langle b_0, b_1, \dots \rangle$ or as a function $b: \mathbb{N} \rightarrow \mathbb{B}$. Suppose that we consider two binary sequences equivalent if they have the same tail, that is, $b \equiv \hat{b}$ if there is an N

so that $i > N$ implies $b(i) = \hat{b}$. Show that for any b , the number of equivalent binary sequences is countably infinite.

2.33 We will show that $\mathbb{Z}[x] = \{a_n x^n + \dots + a_1 x + a_0 \mid n \in \mathbb{N} \text{ and } a_n, \dots, a_0 \in \mathbb{Z}\}$, the set of polynomials in the variable x with integer coefficients, is countable.

(A) Fix a natural number n . Prove that the set of polynomials with $n+1$ -many terms $\mathbb{Z}_n[x] = \{a_n x^n + \dots + a_0 \mid a_n, \dots, a_0 \in \mathbb{Z}\}$ is countable.

(B) Finish the argument.

2.34 Show that if S is countably infinite then there is a $f: S \rightarrow S$ that is one-to-one but not onto.

✓ 2.35 The proof of Lemma 2.9 says that the function $\text{cantor}_3: \mathbb{N}^3 \rightarrow \mathbb{N}$ given by $\text{cantor}_3(a, b, c) = \text{cantor}(\text{cantor}(a, b), c)$ is a correspondence. Verify that.

2.36 Define $c_3: \mathbb{N}^3 \rightarrow \mathbb{N}$ by $\langle x, y, z \rangle \mapsto \text{cantor}(x, \text{cantor}(y, z))$. (A) Compute $c_3(0, 0, 0)$, $c_3(1, 2, 3)$, and $c_3(3, 3, 3)$. (B) Find the triples corresponding to 0, 1, 2, 3, and 4. (C) Give a formula.

2.37 Say that an entry in $\mathbb{N} \times \mathbb{N}$ is on the diagonal if it is $\langle i, i \rangle$ for some i . Show that an entry on the diagonal has a Cantor number that is a multiple of four.

2.38 Corollary 2.13 says that the union of any finite number of countable sets is countable. The base case is for two sets (and the inductive step covers larger numbers of sets). Give a proof specific to the three set case.

2.39 Show that the set of all functions from $\{0, 1\}$ to \mathbb{N} is countable.

2.40 Show that the image under any function of a countable set is countable. That is, show that if S is countable and there is a function $f: S \rightarrow T$ then the range set $f(S) = \text{ran}(f) = \{y \mid y = f(x) \text{ for some } x \in S\}$ is also countable.

2.41 Give a proof of Lemma 2.3.

✓ 2.42 Consider a programming language using the alphabet Σ consisting of the twenty six capital ASCII letters, the ten digits, the space character, open and closed parenthesis, and the semicolon. Show each.

(A) The set of length-5 strings Σ^5 is countable.

(B) The set of strings of length at most 5 over this alphabet is countable.

(C) The set of finite-length strings over this alphabet is countable.

(D) The set of programs in this language is countable.

2.43 There are other correspondences from \mathbb{N}^2 to \mathbb{N} besides Cantor's.

(A) Consider $g: \mathbb{N}^2 \rightarrow \mathbb{N}$ given by $\langle n, m \rangle \mapsto 2^n(2m+1)-1$. Find the number corresponding to the pairs in $\{\langle n, m \rangle \in \mathbb{N}^2 \mid 0 \leq n, m < 4\}$.

(B) Show that g is a correspondence.

(C) The box enumeration goes: $(0, 0)$, then $(0, 1)$, $(1, 1)$, $(1, 0)$, then $(0, 2)$, $(1, 2)$, $(2, 2)$, $(2, 1)$, $(2, 0)$, etc. To what value does $(3, 4)$ correspond?

2.44 The formula for Cantor's unpairing function $\text{cantor}(x, y) = x + [(x+y)(x+y+1)/2]$ give a correspondence for natural number input. What about for real number input? (A) Find $\text{cantor}(2, 1)$. (B) Fix $x = 1$ and find two different $y \in \mathbb{R}$ so that $\text{cantor}(1, y) = \text{cantor}(2, 1)$.

SECTION

II.3 Diagonalization

Following Cantor's definition of cardinality, we produced a number of correspondences between sets. After working through these example maps, a person could come to think that for any two infinite sets there is some sufficiently clever way to give a matching between them.

This impression is wrong. It can happen that there is no correspondence. We now introduce a very powerful technique to demonstrate this, which is central to the entire Theory of Computation.

Diagonalization There are sets so large that they are not countable. That is, there are infinite sets S for which no correspondence exists between S and \mathbb{N} . One such set is the set of reals, $S = \mathbb{R}$.

3.1 **THEOREM** There is no onto map $f: \mathbb{N} \rightarrow \mathbb{R}$. Hence, the set of reals is not countable.

This result is important but so is the proof technique. We start by developing the intuition behind it. The table below illustrates a function $f: \mathbb{N} \rightarrow \mathbb{R}$, listing some inputs and outputs, with the outputs aligned on the decimal point.

n	Decimal expansion of $f(n)$
0	4 2 . 3 1 2 7 7 0 4 ...
1	2 . 0 1 0 0 0 0 0 ...
2	1 . 4 1 4 1 5 9 2 ...
3	-20 . 9 1 9 5 9 1 9 ...
4	0 . 1 0 1 0 0 1 0 ...
5	-0 . 6 2 5 5 4 1 8 ...
:	:

We will show that this function is not onto by producing a number $z \in \mathbb{R}$ that does not equal any of the outputs, any of the $f(n)$'s.

Ignore what is to the left of the decimal point. To its right go down the diagonal, taking the digits 3, 1, 4, 5, 0, 1 ... Construct the desired z by making its first decimal place something other than 3, making its second decimal place something other than 1, etc. Specifically, if the diagonal digit is a 1 then z gets a 2 in that decimal place and otherwise z gets a 1 there. Thus, in this example $z = 0.121112\dots$

By construction, z differs from the number in the first row, $z \neq f(0)$, because they differ in the first decimal place. Similarly, $z \neq f(1)$ because they differ in the second place. In this way z does not equal any of the $f(n)$. Thus f is not onto. This technique is **diagonalization**.

(Here we have skirted a technicality, that some real numbers have two different decimal representations. For instance, $1.000\dots = 0.999\dots$ because the two differ by less than 0.1, less than 0.01, etc. This is a potential snag to the argument

because it means that even though we have constructed a representation that is different than all the representations on the list, it still might not be that the number is different than all the numbers on the list. However, dual representation only happens for decimals when one of the representations ends in 0's while the other ends in 9's. That's why we build z using 1's and 2's.)

Proof We will show that no map $f: \mathbb{N} \rightarrow \mathbb{R}$ is onto.

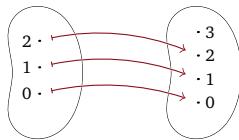
Denote the i -th decimal digit of $f(n)$ as $f(n)[i]$ (if $f(n)$ is a number with two decimal representations then use the one ending in 0's). Let g be the map on the decimal digits $\{0, \dots, 9\}$ given by: $g(j) = 2$ if j is 1 and $g(j) = 1$ otherwise.

Now let z be the real number that has 0 to the left of its decimal point, and whose i -th decimal digit is $g(f(i)[i])$. Then for all i , $z \neq f(i)$ because $z[i] \neq f(i)[i]$. So f is not onto. \square

3.2 **DEFINITION** A set that is infinite but not countable is **uncountable**.

3.3 **REMARK** Before going on, we pause to reflect that the work we have seen so far in this chapter, especially the prior theorem, is both startling and profound. Some infinite sets have more elements than others. In particular, the reals have more elements than the naturals. As dramatized by Galelio's Paradox, this is not just that the reals are a superset of the naturals. Instead, the set of naturals cannot be made to correspond with the set of reals. This is like the children's game of Musical Chairs. We have countably many chairs P_0, P_1, \dots but there are so many children—so many reals—that one of them is left without a chair.

We next define when one set has fewer, or more, elements than another. The intuition comes from the picture below, trying to make a correspondence between the two finite sets $\{0, 1, 2\}$ and $\{0, 1, 2, 3\}$. There are too many elements in the codomain for any map to cover them all. The best that we can do is to cover as many codomain elements as possible, with a function that is one-to-one but not onto.



3.4 **DEFINITION** The set S has **cardinality less than or equal to** that of the set T , denoted $|S| \leq |T|$, if there is a one-to-one function from S to T .

3.5 **EXAMPLE** The inclusion map that sends $n \in \mathbb{N}$ to itself, $n \in \mathbb{R}$, is one-to-one and so $|\mathbb{N}| \leq |\mathbb{R}|$. By Theorem 3.1 the cardinality is strictly less.

The wording of that definition suggests that if both $|S| \leq |T|$ and $|T| \leq |S|$ then $|S| = |T|$. That is true but the proof is beyond our scope; see Exercise 3.31. Also beyond our scope, and not necessary for the work here, is the argument that for any two sets, one of them has cardinality less than or equal to the other.

For the next result, recall that a set's **characteristic function** $\mathbb{1}_S$ is the Boolean function determining membership: $\mathbb{1}_S(s) = T$ if $s \in S$ and $\mathbb{1}_S(s) = F$ if $s \notin S$. (We sometimes instead use the bits 1 for T and 0 for F .) Thus for the set of two letters $S = \{a, c\}$, the characteristic function with domain $\Sigma = \{a, \dots, z\}$ is $\mathbb{1}_S(a) = T$, $\mathbb{1}_S(b) = F$, $\mathbb{1}_S(c) = T$, $\mathbb{1}_S(d) = F$, ... $\mathbb{1}_S(z) = F$.

Recall also that the **power set** $\mathcal{P}(S)$ is the collection of subsets of S . For instance, if $S = \{a, c\}$ then $\mathcal{P}(S) = \{\emptyset, \{a\}, \{c\}, \{a, c\}\}$.

- 3.6 **THEOREM (CANTOR'S THEOREM)** A set's cardinality is strictly less than that of its power set.

We first illustrate the proof. One half is easy: to start with a set S and produce a function to $\mathcal{P}(S)$ that is one-to-one, just map $s \in S$ to the set $\{s\}$.

The harder half is showing that no map from S to $\mathcal{P}(S)$ is onto. For example, consider the set $S = \{a, b, c\}$ and this function $f: S \rightarrow \mathcal{P}(S)$.

$$a \xrightarrow{f} \{b, c\} \quad b \xrightarrow{f} \{b\} \quad c \xrightarrow{f} \{a, b, c\} \quad (*)$$

Below, the first row lists the values of the characteristic function $\mathbb{1}_{f(a)}$ on the inputs a , b , and c . The second row lists the values for $\mathbb{1}_{f(b)}$. And, the third row lists $\mathbb{1}_{f(c)}$.

$s \in S$	$f(s)$	$\mathbb{1}_{f(s)}(a)$	$\mathbb{1}_{f(s)}(b)$	$\mathbb{1}_{f(s)}(c)$
a	{b, c}	F	T	T
b	{b}	F	T	F
c	{a, b, c}	T	T	T

We show that f is not onto by producing a member of $\mathcal{P}(S)$ that is not any of the three sets in (*). For that, take the table's diagonal *FTT* and flip the values to get *TFF*. That describes the characteristic function of the set $R = \{a\}$. This set is not equal to the set $f(a)$ because their characteristic functions differ on a , it is not the set $f(b)$ because the characteristic functions differ on b , and it is not $f(c)$ because they differ on c .

Proof First, $|S| \leq |\mathcal{P}(S)|$ because the inclusion map $\iota: S \rightarrow \mathcal{P}(S)$ given by $\iota(s) = \{s\}$ is one-to-one. For the second half we will show that there is no correspondence, that no map from a set to its power set is onto. Fix $f: S \rightarrow \mathcal{P}(S)$ and consider this element of $\mathcal{P}(S)$.

$$R = \{s \mid s \notin f(s)\}$$

We will demonstrate that no member of the domain maps to R and thus f is not onto. Suppose that there exists $\hat{s} \in S$ such that $f(\hat{s}) = R$. Consider whether \hat{s} is an element of R . We have that $\hat{s} \in R$ if and only if $\hat{s} \in \{s \mid s \notin f(s)\}$. By the definition of membership, that holds if and only if $\hat{s} \notin f(\hat{s})$, which holds if and only if $\hat{s} \notin R$. The contradiction means that no such \hat{s} exists. \square

- 3.7 COROLLARY The cardinality of the set \mathbb{N} is strictly less than the cardinality of the set of functions $f: \mathbb{N} \rightarrow \mathbb{N}$.

Proof Let the set of functions be F . There is a one-to-one map from $\mathcal{P}(\mathbb{N})$ to F , namely the one that associates each subset $S \subseteq \mathbb{N}$ with its characteristic function, $\mathbb{1}_S: \mathbb{N} \rightarrow \mathbb{N}$. Therefore $|\mathbb{N}| < |\mathcal{P}(\mathbb{N})| \leq |F|$. \square

- 3.8 COROLLARY (EXISTENCE OF UNCOMPUTABLE FUNCTIONS) There is a function $f: \mathbb{N} \rightarrow \mathbb{N}$ that is not computable: $f \neq \phi_e$ for all e .

Proof Lemma 2.9 shows that the cardinality of the set of Turing machines equals the cardinality of the set \mathbb{N} . The prior result shows that the cardinality of the set of functions from \mathbb{N} to itself is strictly greater than the cardinality of \mathbb{N} . So the cardinality of the set of functions from \mathbb{N} to itself is greater than the cardinality of the set of Turing machines. Consequently, some natural number function is without an associated Turing machine. \square

This is an epochal result. In the light of Church's Thesis, we take it to prove that there are jobs that no computer can do.

To a person trained in programming, where the focus is on getting a computer to do things, the existence of tasks that cannot be done can be a surprise, perhaps even a shock. One point that these results make is that the work here on sizes of infinities, which can at first seem uselessly abstract, leads to interesting and useful conclusions.

II.3 Exercises

- 3.9 Your study partner is confused about the diagonal argument. "If you had an infinite list of numbers, it would clearly contain every number, right? I mean, if you had a list that was truly INFINITE, then you simply couldn't find a number that is not on the list!" Help them out.
- 3.10 Your classmate says, "Professor, I'm confused. The set of numbers with one decimal place, such as 25.4 and 0.1, is clearly countable—just take the integers and shift all the decimal places by one. The set with two decimal places, such as 2.54 and 6.02 is likewise countable, etc. This is countably many sets, each of which is countable, and so the union is countable. The union is the whole reals, so I think that the reals are countable." Where is their mistake?
- 3.11 Verify Cantor's Theorem, Theorem 3.6, for these finite sets. (A) $\{0, 1, 2\}$
 (B) $\{0, 1\}$ (C) $\{0\}$ (D) $\{\}$
- ✓ 3.12 Use Definition 3.4 to prove that the first set has cardinality less than or equal to the second set.
 (A) $S = \{1, 2, 3\}$, $\hat{S} = \{11, 12, 13\}$
 (B) $T = \{0, 1, 2\}$, $\hat{T} = \{11, 12, 13, 14\}$
 (C) $U = \{0, 1, 2\}$, the set of odd numbers
 (D) the set of even numbers, the set of odds

- 3.13 One set is countable and the other is uncountable. Which is which?
 (A) $\{n \in \mathbb{N} \mid n + 3 < 5\}$ (B) $\{x \in \mathbb{R} \mid x + 3 < 5\}$
- ✓ 3.14 Characterize each set as countable or uncountable. You need only give a one-word answer. (A) $[1..4) \subset \mathbb{R}$ (B) $[1..4) \subset \mathbb{N}$ (C) $[5..\infty) \subset \mathbb{R}$
 (D) $[5..\infty) \subset \mathbb{N}$
- 3.15 List all of the functions with domain $A_2 = \{0, 1\}$ and codomain $\mathcal{P}(A_2)$. How many functions are there for a set A_3 with three elements? n elements?
- 3.16 List all of the functions from S to T . How many are one-to-one? (A) $S = \{0, 1\}$, $T = \{10, 11\}$ (B) $S = \{0, 1\}$, $T = \{10, 11, 12\}$
- ✓ 3.17 Short answer: fill each blank by choosing from (i) uncountable, (ii) countable or uncountable, (iii) finite, (iv) countable, (v) finite, countably infinite, or uncountable (you might use an answer more than once, or not at all). Give the sharpest conclusion possible. You needn't give a proof.
 (A) If A and B are finite then $A \cup B$ is _____.
 (B) If A is countable and B is finite then $A \cup B$ is _____.
 (C) If A is countable and B is uncountable then $A \cup B$ is _____.
 (D) if A is countable and B is uncountable then $A \cap B$ is _____.
- 3.18 Short answer: suppose that S is countable and consider $f: S \rightarrow T$. For both of the items below, list all of these that are possible: (i) S is finite, (ii) T is finite, (iii) S is countably infinite, (iv) T is countably infinite, (v) T is uncountable, where
 (A) the map is onto, (B) the map is one-to-one.
- ✓ 3.19 Give a set with a larger cardinality than \mathbb{R} .
- ✓ 3.20 Recall that $\mathbb{B} = \{0, 1\}$.
 (A) Show that the set of finite bit strings, $\langle b_0 b_1 \dots b_{k-1} \rangle$ where $b_i \in \mathbb{B}$ and $k \in \mathbb{N}$, is countable.
 (B) An infinite bit string $f = \langle b_0, b_1, \dots \rangle$ is a function $f: \mathbb{N} \rightarrow \mathbb{B}$. Show that the set of infinite bit strings is uncountable, using diagonalization.
- 3.21 Prove that for two sets, $S \subseteq T$ implies $|S| \leq |T|$.
- 3.22 Use diagonalization to show that this is false: all functions $f: \mathbb{N} \rightarrow \mathbb{N}$ with a finite range are computable.
- 3.23 You study with someone who says, “Yes, obviously there are different sizes of infinity. The plane \mathbb{R}^2 obviously has infinitely many more points than the line \mathbb{R} , so it is a larger infinity.” Convince them that their argument is wrong because the cardinality of the plane is the same as the cardinality of the line. Hint: consider the function $f: \mathbb{R}^2 \rightarrow \mathbb{R}$ such that $f(x, y)$ interleaves the digits of the two input numbers.
- 3.24 In mathematics classes we mostly work with rational numbers, perhaps leaving the impression that irrational numbers are rare. Actually, there are more irrational numbers than rationals. Prove that while the set of rational numbers is countable, the set of irrational numbers is uncountable.
- ✓ 3.25 Example 2.11 shows that the rational numbers are countable. What happens

when we apply diagonal argument given in Theorem 3.1 is to an enumeration of the rationals? Consider a sequence q_0, q_1, \dots that contains all of the rationals. For each of those numbers use a decimal expansion $q_i = d_i.d_{i,0}d_{i,1}d_{i,2}\dots$ (with $d_i \in \mathbb{Z}$ and $d_{i,j} \in \{0, \dots, 9\}$) that does not end in all 9's, so that the decimal expansion is unique.

- (A) Let g be the map on the decimal digits 0, 1, ... 9 given by $g(1) = 2$, and $g(0) = g(2) = g(3) = \dots = 1$. Define $z = \sum_{n \in \mathbb{N}} g(d_{n,n}) \cdot 10^{-(n+1)}$. Show that z is irrational.
- (B) Use the prior item to conclude that the diagonal number $d = \sum_{n \in \mathbb{N}} d_{n,n} \cdot 10^{-(n+1)}$ is irrational. *Hint:* show that, unlike a rational number, it has no repeating pattern in its decimal expansion.
- (C) Why is the fact that the diagonal is not rational not a contradiction to the fact that we can enumerate all of the rationals?

3.26 Verify Cantor's Theorem in the finite case by showing that if S is finite then the cardinality of its power set is $|\mathcal{P}(S)| = 2^{|S|}$.

3.27 The definition $R = \{s \mid s \notin f(s)\}$ is the key to the proof of Cantor's Theorem, Theorem 3.1. This story illustrates the idea: a high school yearbook asks each graduating student s_i make a list $f(s_i)$ of class members that they predict will someday be famous. Define the set of humble students H to be those who are not on their own list. Show that no student's list equals H .

3.28 The proof of Theorem 3.1 must work around the fact that some numbers have more than one base ten representation. Base two also has the property that some numbers have more than one representation; an example is 0.01000 ... and 0.00111 How could you make the argument work in base two?

3.29 The discussion after the statement of Theorem 3.1 includes that the real number 1 has two different decimal representations, $1.000\dots = 0.999\dots$

- (A) Verify this equality using the formula for an infinite geometric series, $a + ar + ar^2 + ar^3 + \dots = a \cdot 1/(1 - r)$.
- (B) Show that if a number has two different decimal representations then in the leftmost decimal place where they differ, they differ by 1. *Hint:* that is the biggest difference that the remaining decimal places can make up.
- (C) In addition show that, for the one with the larger digit in that first differing place, all of the digits to its right are 0, while the other representation has that all of the remaining digits are 9's. *Hint:* this is similar to the prior item.

3.30 Show that there is no set of all sets. *Hint:* use Theorem 3.6.

3.31 Definition 3.4 extends the definition of equal cardinality to say that $|A| \leq |B|$ if there is a one-to-one function from A to B . The **Schröder–Bernstein theorem** is that if both $|S| \leq |T|$ and $|T| \leq |S|$ then $|S| = |T|$. We will walk through the proof. It depends on finding chains of images: for any $s \in S$ we form the associated chain by iterating application of the two functions, both to the right and the left, as here.

$$\dots f^{-1}(g^{-1}(s)), g^{-1}(s), s, f(s), g(f(s)), f(g(f(s))) \dots$$

(Starting with s the chain to the right is $s, f(s), g(f(s)), f(g(f(s))), \dots$ while the chain to the left is $\dots, f^{-1}(g^{-1}(s)), g^{-1}(s), s.$) For any $t \in T$ define the associated chain similarly.

An example is to take a set of integers $S = \{0, 1, 2\}$ and a set of characters $T = \{a, b, c\}$, and consider the two one-to-one functions $f: S \rightarrow T$ and $g: T \rightarrow S$ shown here.

s	$f(s)$	t	$g(t)$
0	b	a	0
1	c	b	1
2	a	c	2

Starting at $0 \in S$ gives a single chain that is cyclic, $\dots, 0, b, 1, c, 2, a, 0, \dots$

- (A) Consider $S = \{0, 1, 2, 3\}$ and $T = \{a, b, c, d\}$. Let f associate $0 \mapsto a, 1 \mapsto b, 2 \mapsto d$ and $3 \mapsto c$. Let g associate $a \mapsto 0, b \mapsto 1, c \mapsto 2$ and $d \mapsto 3$. Check that these maps are one-to-one. List the chain associated with each element of S and the chain associated with each element of T .
- (B) For infinite sets a chain can have a first element, an element without any preimage. Let S be the even numbers and let T be the odds. Let $f: S \rightarrow T$ be $f(x) = x + 1$ and let $g: T \rightarrow S$ be $g(x) = x + 1$. Show each map is one-to-one. Show there is a single chain and that it has a first element.
- (C) Argue that we can assume without loss of generality that S and T are disjoint sets.
- (D) Assume that S and T are disjoint and that $f: S \rightarrow T$ and $g: T \rightarrow S$ are one-to-one. Show that every element of either set is in a unique chain, and that each chain is of one of four kinds: (i) those that repeat after some number of terms (ii) those that continue infinitely in both directions without repeating (iii) those that continue infinitely to the right but stop on the left at some element of S , and (iv) those that continue infinitely to the right but stop on the left at some element of T .
- (E) Show that for any chain the function below is a correspondence between the chain's elements from S and its elements from T .

$$h(s) = \begin{cases} f(s) & - \text{if } s \text{ is in a sequence of type (i), (ii), or (iii)} \\ g^{-1}(s) & - \text{if } s \text{ is in a sequence of type (iv)} \end{cases}$$

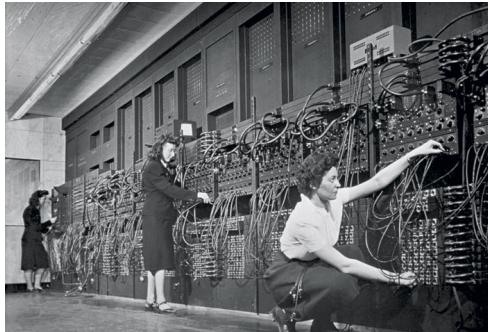
SECTION

II.4 Universality

We have seen a number of Turing machines, such as one whose output is the successor of its input, one that adds two input numbers, and others. These are single-purpose devices, where to get different input-output behavior we need a new machine, that is, new hardware. This was what we meant by saying that a good first take on Turing machines is that they are more like a modern computer

program than a modern computer—on a modern computer, to change behavior you don't change the hardware.

This picture shows programmers of an early computer. They are changing its behavior by changing its circuits, using the patch cords.



4.1 FIGURE: ENIAC, reconfigure by rewiring.

Imagine having a cellphone where to change from running a browser to taking a call you must pull one chip and replace it with another. The picture's patch cords are an improvement over a soldering iron, but are not a final answer.

Universal Turing machine A pattern in technology is for jobs done in hardware to migrate to software. The classic example is weaving.



Weaving by hand, as the loom operator on the left is doing, is intricate and slow. We can make a machine to reproduce her pattern. But what if we want a different pattern; do we need another machine? In 1801 J Jacquard built a loom like the one on the right, controlled by cards. Getting a different pattern does not require a new loom, it only requires swapping cards.

Turing introduced the analog to this for computers. He produced a Turing machine $\mathcal{U}\mathcal{P}$ that can be fed a tape containing a description of a Turing machine \mathcal{M} , along with input for that machine. Then $\mathcal{U}\mathcal{P}$ will have the same input-output behavior as would \mathcal{M} . If \mathcal{M} halts on the input then $\mathcal{U}\mathcal{P}$ will halt and give the same output, while if \mathcal{M} does not halt on that input then $\mathcal{U}\mathcal{P}$ also does not halt.

This single machine can be made to have any desired computable behavior. So we don't need infinitely many different machines, we can just use $\mathcal{U}\mathcal{P}$.



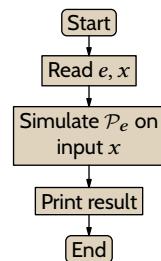
An ouroboros, a snake swallowing its own tail

Before stating Turing's theorem, we first address an often-asked question about whether it is even possible. The machine $\mathcal{U}\mathcal{P}$ may seem to present a chicken and egg problem: how can we give a Turing machine as input to a Turing machine? In particular, since $\mathcal{U}\mathcal{P}$ is itself a Turing machine, the theorem seems to allow the possibility of giving it to itself—won't feeding a machine to itself lead to infinite regress?

We run Turing machines by loading symbols on the tape and pressing Start. So we don't feed machines with machines; we feed them symbols, representations of things. True, we can feed $\mathcal{U}\mathcal{P}$ a specification of itself, such as a pair e, x where e is the index number of $\mathcal{U}\mathcal{P}$, and is thus computationally equivalent to that machine's source, and x is the input. Even so, the universe won't collapse—we can absolutely use a text editor to edit its own source, or ask a compiler to generate its own executable. Similarly, we can feed $\mathcal{U}\mathcal{P}$ its own index number. Lots of interesting things happen as a result, but there is no inherent impossibility.

- 4.2 THEOREM (TURING, 1936) There is a Turing machine that when given the inputs e and x will have the same output behavior as does \mathcal{P}_e on input x .

This is a **Universal Turing Machine**.[†] Universal machines are familiar from everyday computing. This figure[‡] outlines the action of a computer operating system that is given a program to run and some data for that program. Think of the program as like some Turing machine, \mathcal{P}_e , and think of the data as a string of 0's and 1's that we can interpret as a number, x . The operating system loads the program and then feeds it the input. That is, the operating system arranges that the hardware will behave like machine e with input x . As with an operating system, Universal Turing machines change their behavior in software. No patch chords.



Another computing experience that is like a universal machine is an interpreter. Below is an interaction with the Racket interpreter. At the first prompt we enter the source of a routine that inputs i and returns the sum of the first i numbers. At the second prompt the interpreter runs that source with $i = 4$.

```
$ racket
Welcome to Racket v8.3 [cs].
> (define (sum i tot)
  (if (= i 0)
      tot
      (sum (- i 1) (+ tot i))))
> (sum 4 0)
10
```

The most direct example of computing systems that act as universal machines is a language's eval statement. At the first prompt below we define a routine that

[†]We could also define a Universal Turing machine to take the single-number input $\text{cantor}(e, x)$. [‡]This is a flowchart, which gives a high level sketch of a routine. We use three types of boxes. Round corner boxes are for Start and End. Rectangles are for ordinary operations on data. Diamond boxes, which will appear in later charts, are for decisions, if statements.

has the interpreter evaluate the expression that is input. In the next prompt we define a list, quoted so that it is not interpreted. This list is the source of a function; `lambda (i) ...` defines a function of one input, i (in contrast to the definitions of the functions `sum` and `utm`, this function does not have a name and so the definition syntax is different). In the third and fourth prompts, the interpreter evaluates that list and applies it to the numbers 5 and 0. That is, like the loom's punched cards, different inputs cause `utm` to have different behaviors.

```
> (define (utm s)
  (eval s))
> (define test '(lambda (i) (if (= i 0) 1 0)))
> ((utm test) 5)
0
> ((utm test) 0)
1
```

Finally, as to the proof of the theorem, the simplest way to prove that something exists is to produce it. We have already exhibited what amounts to a Universal Turing machine. At the end of Chapter One, on page 37, we gave code for a Turing machine simulator, which reads a Turing machine from a file and then runs it. The code is in Racket but Church's Thesis asserts that we could write a Turing machine with the same behavior.

Uniformity Consider this job: given a real number $r \in \mathbb{R}$, write a program to produce its digits. More precisely, the job is to produce a \mathcal{P}_r such that when given $n \in \mathbb{N}$ as input, \mathcal{P}_r outputs the n -th decimal place of r (for $n = 0$, it outputs the integer to the left of the decimal point).

We know that this is not possible for all r because there are uncountably many real numbers but only countably many Turing machines. But what stops us? One of the enjoyable things about coding is the feeling of being able to get the machine to do anything we want—why can't we output whatever digits we like?

There certainly are real numbers for which there is such a program. One is 0.25.

```
(define (one-quarter-decimal-places n)
  (cond
    [(= n 0) 0]
    [(= n 1) 2]
    [(= n 2) 5]
    [else 0]))
```

For a more generic number, say, some $r = 0.703\dots$, we might momentarily imagine brute-forcing it.

```
(define (r-decimal-place n)
  (cond
    [(= n 0) 0]
    [(= n 1) 7]
    [(= n 2) 0]
    [(= n 3) 3]
    ...
  ))
```

But that's silly. Programs have finite length and so can't have infinitely many cases.

That is, because of the `if`, what the following program does on $n = 7$ is unconnected to what it does on other inputs.

```
(define (foo n)
  (if (= n 7)
    42
    (* 2 n)))
```

But a program can only have finitely many such differently-behaving branches. The fact that a Turing machine has only finitely many instructions imposes a condition of uniformity on its behavior.

- 4.3 EXAMPLE Connecting in this way the idea that ‘something is computable’ with ‘it is uniformly computable’ has some surprising consequences. Consider the problem of producing a program that inputs a number n and decides whether somewhere in the decimal expansion of $\pi = 3.14159\dots$ there are n consecutive nines.

There are two possibilities. Either for all n such a sequence exists, or else there is some n_0 where a sequence of nines exists for lengths less than n_0 and no sequence exists when $n \geq n_0$. Consequently the problem is solved: one of the two below is the right program (although at this moment we don’t know which).

<pre>(define (sequence-of-nines-0 n) 1)</pre>	<pre>(define (sequence-of-nines-1 n) (if (< n n0) 1 0))</pre>
---	--

One surprising aspect of this argument is that neither of the two routines appears to have much to do with π . Also surprising, and perhaps unsettling, is that we have shown that the problem is solvable without showing how to solve it. That is, there is a difference between showing that this function is computable

$$f(n) = \begin{cases} 1 & - \text{if } \pi \text{ has } n \text{ consecutive nines} \\ 0 & - \text{otherwise} \end{cases}$$

and possessing an algorithm to compute it. This observation shows that if the assertion “something is computable if you can write a program for it” is not actually false, at the very least it suppresses some important subtleties.

In contrast, consider a routine that inputs $i \in \mathbb{N}$ and outputs the i -th decimal place of π . Using it, we can write a program that takes in n and steps through the digits of π , looking for n consecutive nines. With this approach we are constructing the answer, not just saying that it exists. This approach is also uniform in the sense that we could modify it to use other routines and so look for strings of nines in other numbers. However this approach has the disadvantage that if there is an n_0 where π does not have n consecutive nines for $n \geq n_0$ then this program will search forever without discovering that.

Parametrization Universality says that there is a Turing machine that takes in inputs e and x and returns the same value as we would get by running \mathcal{P}_e on input x , including not halting if the machine does not halt on that input. That

is, there is a computable function $\phi: \mathbb{N}^2 \rightarrow \mathbb{N}$ such that $\phi(e, x) = \phi_e(x)$ if $\phi_e(x) \downarrow$ and $\phi(e, x) \uparrow$ if $\phi_e(x) \uparrow$.

There, the e travels from the function's argument to the index. We now generalize. Start with a program that takes two inputs such as this one.

```
(define (P x y)
  (+ x y))
```

Freeze the first argument. The result is a one-input program. Here we freeze x at 7 and at 8.

```
(define (P_7 y)
  (P 7 y))
```

```
(define (P_8 y)
  (P 8 y))
```

This is **partial application** because we are not freezing all of the input variables. Instead, we are **parametrizing** the variable x , resulting in a family of programs P_0, P_1 , etc.

The programs in the family are related to the starting one, obviously. Denoting the function computed by the above starting program P as $\psi(x, y) = x + y$, partial application gives a family of functions: $\psi_0(y) = y$, $\psi_1(y) = 1 + y$, $\psi_2(y) = 2 + y$, \dots . The next result says that in general, from the index of a starting Turing machine or computable function and from the values that are frozen, we can compute the family members.

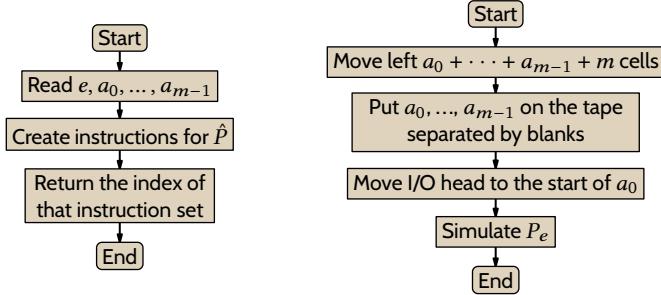
- 4.4 **THEOREM (S-M-N THEOREM, OR PARAMETER THEOREM)** For every $m, n \in \mathbb{N}$ there is a computable total function $s_{m,n}: \mathbb{N}^{1+m} \rightarrow \mathbb{N}$ such that for an $m+n$ -ary function $\phi_e(x_0, \dots x_{m-1}, x_m, \dots x_{m+n-1})$, freezing the initial m variables at $a_0, \dots a_{m-1} \in \mathbb{N}$ gives the n -ary computable function $\phi_{s(e, a_0, \dots a_{m-1})}(x_m, \dots x_{m+n-1})$.[†]

Proof We will produce the function s to satisfy three requirements: it must be effective, it must input an index e and an m -tuple $a_0, \dots a_{m-1}$, and it must output the index of a machine \hat{P} that, when given the input $x_m, \dots x_{m+n-1}$, will return the value $\phi_e(a_0, \dots a_{m-1}, x_m, \dots x_{m+n-1})$, or fail to halt if that function diverges.

The idea is that the machine that computes s will construct the instructions for \hat{P} . We can get from the instruction set to the index using Cantor's encoding, so with that we will be done.

Below on the left is the flowchart for the machine that computes the function s . In its third box it creates the set of four-tuple instructions shown on the right, which make the machine \hat{P} . The machine on the left needs $a_0, \dots a_{m-1}$ for the right side's second, third, and fourth boxes, and it needs e for the fifth box. (In this book we are flexible about the convention for input and output representations for Turing machines. However, in this proof, to be as clear as possible in the right side's flowchart, we assume that input is encoded in unary, that inputs are separated with a single blank, and that when the machine is started the head should be under the input's left-most 1.)

[†] This result is stated in terms of functions, not machines, because the machines need not be the same. That is, they are not equal sets of four-tuple instructions. But they will have the same input-output behavior, which means the same function.



The Turing machine \hat{P} does not first read its inputs x_m, \dots, x_{m+n-1} . Instead, it first moves left and writes a_0, \dots, a_{m-1} on the tape, in unary and separated by blanks, and with a blank between a_{m-1} and x_m . (Recall that the a_i are parameters, not variables. They are fixed. They are, so to speak, hard-coded into \hat{P} , which is how it knows what to write.) Then using universality, \hat{P} simulates Turing machine P_e and lets it run on the entire list of inputs now on the tape, $a_0, \dots, a_{m-1}, x_m, \dots, x_{m+n-1}$. \square

In the notation $s_{m,n}$, the subscript m is the number of inputs being frozen while n is the number of inputs left free. These subscripts can be a bother and we often omit them.

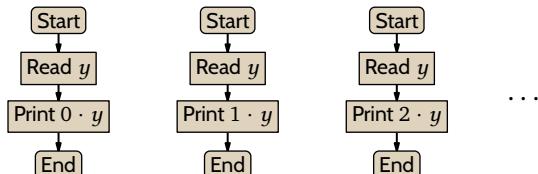
The key point about the $s\text{-}m\text{-}n$ theorem is that it gives not just one computable function but instead a family.

4.5 EXAMPLE Consider the two-input routine sketched by this flowchart.



By Church's Thesis there is a Turing machine following the sketch, computing the function $\psi(x, y) = x \cdot y$. Let that machine have index e_0 . (The subscript emphasizes that this index is a constant.)

On the left below is the flowchart sketching the machine $\mathcal{P}_{s(e_0,0)}$, which freezes the value of x to 0 and so computes the function $\phi_{s(e_0,0)}(y) = 0$. For example, $\phi_{s(e_0,0)}(5) = 0$. Similarly, the other two are flowcharts summarizing $\mathcal{P}_{s(e_0,1)}$ and $\mathcal{P}_{s(e_0,2)}$, freezing the value of x at 1 and 2 and therefore computing the functions $\phi_{s(e_0,1)}(y) = y$ and $\phi_{s(e_0,2)}(y) = 2y$.



Here is the generic member, $\mathcal{P}_{s(e_0, x)}$.



Compare (**) to (*). The difference is that this machine does not read x ; rather, thinking of these as programs instead of Turing machines, x is hard-coded into the source body.

In summary, the $s\text{-}m\text{-}n$ Theorem gives a sequence of computable functions such as $\phi_{s(e_0, x)}$ that is a family in that the indices are given by a computable function. The family is parametrized by x , since e_0 is fixed.

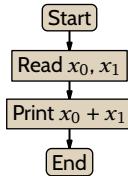
Restated, this family is uniformly computable — there is a computable function s (more precisely, $s_{1,1}$) going from the index e and the parameter value x to the index of the result in (**). So the $s\text{-}m\text{-}n$ Theorem is about uniformity.

II.4 Exercises

4.6 Someone in your study group asks, “What can a Universal Turing machine do that a regular Turing machine cannot?” Help them out.

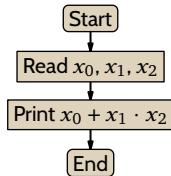
- ✓ 4.7 Has anyone ever built a Universal Turing machine or a device equivalent to one, or is it a theory-only thing?
- 4.8 Can a Universal Turing machine simulate another Universal Turing machine, or for that matter can it simulate itself?
- ✓ 4.9 Your class has someone who says, “Universal Turing machines make no sense to me. How could a machine simulate another machine that has more states?” Correct their misimpression.
- 4.10 Is there more than one Universal Turing machine?
- 4.11 What happens if we feed a Universal Turing machine to itself? For instance, where the index e_0 is such that $\phi_{e_0}(e, x) = \phi_e(x)$ for all x , what is the value of $\phi_{e_0}(e_0, 5)$?
- 4.12 Consider the function $f(x_0, x_1) = 3x_0 + x_0 \cdot x_1$.
 - (A) Freeze x_0 to have the value 4. What is the resulting one-variable function?
 - (B) Freeze x_0 at 5. What is the resulting one-variable function?
 - (C) Freeze x_1 to be 0. What is the resulting function?
- 4.13 Consider $f(x_0, x_1, x_2) = x_0 + 2x_1 + 3x_2$.
 - (A) Freeze x_0 to have the value 1. What is the resulting two-variable function?
 - (B) What two-variable function results from fixing x_0 to be 2?
 - (C) Let a be a natural number. What two-variable function results from fixing x_0 to be a ?
 - (D) Freeze x_0 at 5 and x_1 at 3. What is the resulting one-variable function?

- (E) What one-variable function results from fixing x_0 to be a and x_1 to be b , for $a, b \in \mathbb{N}^*$?
✓ 4.14 Suppose that the Turing machine sketched by this flowchart has index e_0 .



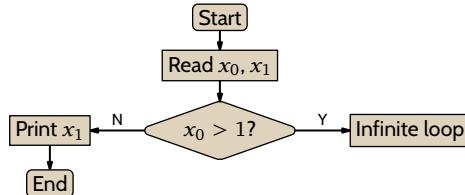
- (A) Describe the function $\phi_{s_{1,1}(e_0,1)}$.
(B) What are the values of $\phi_{s_{1,1}(e_0,1)}(0)$, $\phi_{s_{1,1}(e_0,1)}(1)$, and $\phi_{s_{1,1}(e_0,1)}(2)$?
(C) Describe the function $\phi_{s_{1,1}(e_0,0)}$.
(D) What are the values of $\phi_{s_{1,1}(e_0,0)}(0)$, $\phi_{s_{1,1}(e_0,0)}(1)$, and $\phi_{s_{1,0}(e_0,0)}(2)$?

4.15 Let the Turing machine sketched by this flowchart have index e_0 .

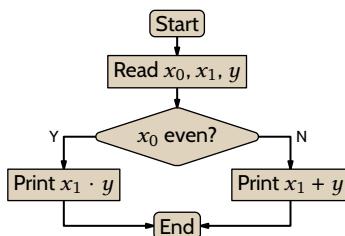


- (A) Describe the function $\phi_{s_{1,2}(e_0,1)}$.
(B) Find $\phi_{s_{1,2}(e_0,1)}(0, 1)$, $\phi_{s_{1,2}(e_0,1)}(1, 0)$, and $\phi_{s_{1,2}(e_0,1)}(2, 3)$
(C) Describe the function $\phi_{s_{2,1}(e_0,1,2)}$.
(D) Find $\phi_{s_{2,1}(e_0,1,2)}(0)$, $\phi_{s_{2,1}(e_0,1,2)}(1)$, and $\phi_{s_{2,1}(e_0,1,2)}(2)$.

- ✓ 4.16 Suppose that the Turing machine sketched by this flowchart has index e_0 .



- (A) Describe $\phi_{s_{1,1}(e_0,0)}$. (B) Find $\phi_{s_{1,1}(e_0,0)}(5)$. (C) Describe $\phi_{s_{1,1}(e_0,1)}$. (D) Find $\phi_{s_{1,1}(e_0,1)}(5)$. (E) Describe $\phi_{s_{1,1}(e_0,2)}$. (F) Find $\phi_{s_{1,1}(e_0,2)}(5)$.
✓ 4.17 Let the Turing machine sketched by this flowchart have index e_0 .



We will describe the family of functions parametrized by the arguments x_0 and x_1 .

- (A) Use Theorem 4.4, the $s\text{-}m\text{-}n$ theorem, to fix $x_0 = 0$ and $x_1 = 3$. Describe $\phi_{s(e_0,0,3)}$. What is $\phi_{s(e_0,0,3)}(5)$?
 - (B) Use the $s\text{-}m\text{-}n$ theorem to fix $x_0 = 1$. Describe $\phi_{s(e_0,1,3)}$. What is $\phi_{s(e_0,1,3)}(5)$?
 - (C) Describe $\phi_{s(e_0,a,b)}$.
- ✓ 4.18 Show that there is a total computable function $g: \mathbb{N} \rightarrow \mathbb{N}$ such that Turing machine $\mathcal{P}_{g(n)}$ computes the function $y \mapsto y + n^2$.
- ✓ 4.19 Show that there is a total computable function $g: \mathbb{N}^2 \rightarrow \mathbb{N}$ such that Turing machine $\mathcal{P}_{g(m,b)}$ computes $x \mapsto mx + b$.
- ✓ 4.20 Suppose that e_0 is such that ϕ_{e_0} is the function computed by a Universal Turing machine, meaning that if given the input $\text{cantor}(e, x)$ then it returns the same value as $\phi_e(x)$. Suppose also that e_1 is such that $\phi_{e_1}(x) = 4x$ for all $x \in \mathbb{N}$. Determine, if possible, the value of these (if it is not possible then briefly describe why not).
 - (A) $\phi_{e_0}(\text{cantor}(e_1, 5))$
 - (B) $\phi_{e_1}(\text{cantor}(e_0, 5))$
 - (C) $\phi_{e_0}(\text{cantor}(e_0, \text{cantor}(e_1, 5)))$
 4.21 Suppose that e_0 is such that $\phi_{e_0}(\text{cantor}(e, x))$ returns the same value as $\phi_e(x)$ (or does not converge if that function does not converge). Suppose also that $\phi_{e_1}(x) = x + 2$ and that $\phi_{e_2}(x) = x^2$, for all $x \in \mathbb{N}$. If possible determine the value of these (if it is not possible, say why not).
 - (A) $\phi_{e_0}(\text{cantor}(e_1, 4))$
 - (B) $\phi_{e_0}(\text{cantor}(4, e_1))$
 - (C) $\phi_{e_1}(\text{cantor}(e_0, \text{cantor}(e_2, 3)))$
 - (D) $\phi_{e_0}(\text{cantor}(e_0, \text{cantor}(e_0, 4)))$

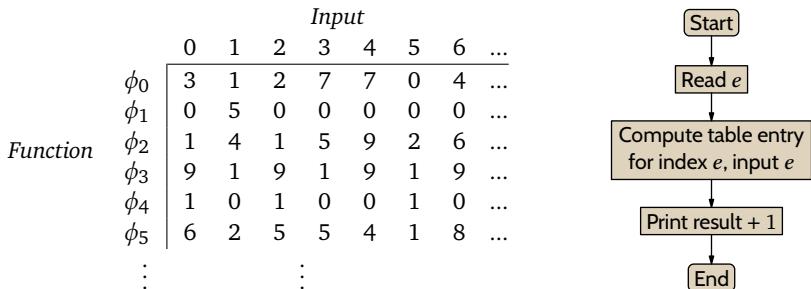
SECTION

II.5 The Halting problem

We've showed that there are functions that are not mechanically computable. We gave a counting argument, that there are countably many Turing machines but uncountably many functions and so there are functions with no associated machine. While knowing what's true is great, even better is to exhibit a specific function that is unsolvable. We will now do that.

Definition The natural approach to producing such a function is to go through Cantor's Theorem and effectivize it, to turn the proof into a construction.

Here is an illustrative table adapted from the discussion of Cantor's Theorem on page 72. Imagine that this table's rows are the computable functions and its columns are the inputs. For instance, this table lists $\phi_2(3) = 5$.



Diagonalizing means considering the machine on the right. It moves down the array's diagonal, changing the 3, changing the 5, etc. Thus, when $e = 0$ then the output is 4, when $e = 1$ then the output is 6, etc. Our goal with this machine is to ensure that no computable function, none of the table's rows, has the same input-output relationship as the machine.

But that's a puzzle, an apparent contradiction. The flowchart outlines an effective procedure—we can implement this using a Universal Turing machine—and thus its output should be one of the rows.

What's the puzzle's resolution? The program's first, second, fourth, and fifth boxes are trivial, so the issue must involve the box in the middle. The answer is that there must be an $e \in \mathbb{N}$ so that $\phi_e(e) \uparrow$, and for that number the machine in the flowchart never gets through the middle box and consequently never prints the output. That is, to avoid a contradiction the above table must contain \uparrow 's.

So this puzzle has led to an important insight: the fact that some computations fail to halt on some inputs is very important.

5.1 **PROBLEM (Halting PROBLEM)**[†] Given $e \in \mathbb{N}$, determine whether $\phi_e(e) \downarrow$, that is, whether Turing machine \mathcal{P}_e halts on input e .

5.2 **DEFINITION** The **Halting problem set** is $K = \{e \in \mathbb{N} \mid \phi_e(e) \downarrow\}$.

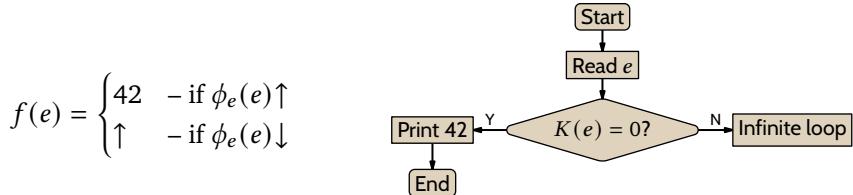
5.3 **THEOREM** The Halting problem is unsolvable by any Turing machine.

Proof Assume otherwise, that there exists a Turing machine with this behavior.

$$\mathbb{1}_K(e) = K(e) = \text{halt_decider}(e) = \begin{cases} 1 & \text{if } \phi_e(e) \downarrow \\ 0 & \text{if } \phi_e(e) \uparrow \end{cases}$$

Then the function below is also mechanically computable. The flowchart illustrates how f is constructed; it uses the above function in its decision box. (In the top case the particular output value 42 doesn't matter, all that matters is that f converges.)

[†]We use a distinct typeface for problem names, as in 'Halting'.



Since this is mechanically computable, it has a Turing machine index. Let that index be e_0 , so that $f(x) = \phi_{e_0}(x)$ for all inputs x .

Now consider $f(e_0) = \phi_{e_0}(e_0)$ (that is, feed the machine \mathcal{P}_{e_0} its own index). If it diverges then the first clause in the definition of f means that $f(e_0) \downarrow$, which contradicts the assumption of divergence. If it converges then f 's second clause means that $f(e_0) \uparrow$, also a contradiction. Since assuming that `halt_decider` is mechanically computable leads to a contradiction, that function is not mechanically computable. \square

We say that a problem is **unsolvable** if no Turing machine has the desired input-output behavior. If the problem is to compute the answers to ‘yes’ or ‘no’ questions, that is, to decide membership in a set, then we say that the set is **undecidable**. With Church’s Thesis in mind, we interpret these to mean that the problem or set is unsolvable by any discrete mechanism.

General unsolvability We have named one task, the **Halting problem**, that no mechanical device can solve. With that one in hand we are able to show that a wide class of jobs cannot be done. That is, the **Halting problem** is part of a larger unsolvability phenomenon.

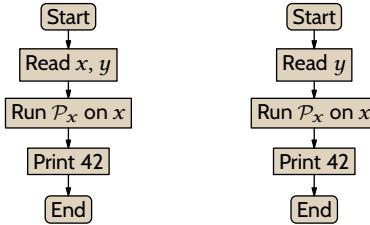
- 5.4 EXAMPLE Consider this problem: we want to know if a given Turing machine halts on the input 3. That is: given e , does $\phi_e(3) \downarrow$?

We will show that if this

$$\text{halts_on_three_decider}(e) = \begin{cases} 1 & - \text{if } \phi_e(3) \downarrow \\ 0 & - \text{otherwise} \end{cases}$$

were a computable function then we could compute the solution of the **Halting problem**. That’s impossible, so we will then know that `halts_on_three_decider` is also not computable.

Our strategy is to create a scheme where being able to determine whether an arbitrary machine halts on 3 allows us to settle questions about the **Halting problem**. Imagine that we have a particular e and want to know whether $\phi_e(x) \downarrow$. Consider the machine outlined on the right below. It reads the input y and ignores it, and also gives a nominal output. Its action is in the middle box, where the code uses a universal Turing machine to simulate running \mathcal{P}_x on input x . If that halts then the machine on the right as a whole halts, for any input. If not then this machine as a whole does not halt.



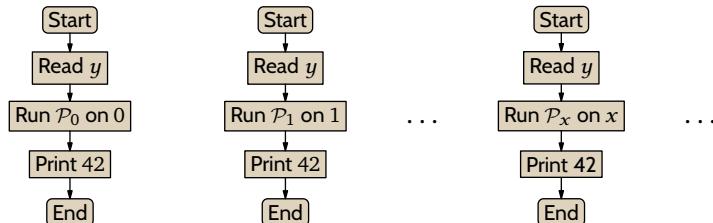
In particular, the machine on the right halts on input $y = 3$ if and only if P_x halts on x (this is also true for all other y 's but we don't care). So with this we can leverage being able to answer questions about halting on 3 to answer questions about whether P_x halts on x .

We are ready for the argument. Consider this function.

$$\psi(x, y) = \begin{cases} 42 & - \text{if } \phi_x(x) \downarrow \\ \uparrow & - \text{otherwise} \end{cases}$$

Observe that ψ is computed by the flowchart above on the left. So it is intuitively computable. By Church's Thesis there is a Turing machine whose input-output behavior is ψ . That machine has some index, e_0 , meaning that $\psi = \phi_{e_0}$.

Use the $s\text{-}m\text{-}n$ theorem to parametrize x , giving $\phi_{s(e_0, x)}$. This is a family of functions, one for $x = 0$, one for $x = 1$, etc. Below are the associated machines. Note that each has a 'Read y ' but no 'Read x '; for these machines the value used in the middle box is hard-coded into the source. Note also that the flowchart on the right is the same as the one on the right in the strategy discussion above. That is, machine $P_{s(e_0, x)}$ halts on input $y = 3$ if and only if P_x halts on input x .



Therefore, for all $x \in \mathbb{N}$ we have this.

$$\phi_x(x) \downarrow \quad \text{if and only if} \quad \text{halts_on_three_decider}(s(e_0, x)) = 1 \quad (*)$$

The function s is computable so if `halts_on_three_decider` were computable then the entire right side of $(*)$ would be computable. That would in turn imply that the Halting problem on the left side is computably solvable, which it isn't. Therefore `halts_on_three_decider` is not computable.

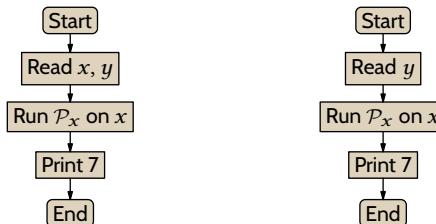
- 5.5 REMARK The subscript 0 is there as signal that e_0 is constant, since it is the index of the machine that computes ψ . In the above family of infinitely many computable functions, the parameter x is what varies. We use that in (*).
- 5.6 EXAMPLE We will show that this problem is not mechanically solvable: given e , determine whether \mathcal{P}_e outputs 7 for some input.

$$\text{outputs_seven_decider}(e) = \begin{cases} 1 & - \text{if } \phi_e(y) = 7 \text{ for some } y \\ 0 & - \text{otherwise} \end{cases}$$

We will follow the prior example as a model. Consider this.

$$\psi(x, y) = \begin{cases} 7 & - \text{if } \phi_x(x) \downarrow \\ \uparrow & - \text{otherwise} \end{cases}$$

The flowchart on the left below sketches how to compute ψ . Thus it is intuitively mechanically computable and Church's Thesis says that there is a Turing machine whose input-output behavior is ψ . That Turing machine has an index, e_0 , so that $\psi = \phi_{e_0}$.



The *s-m-n* theorem gives a family of functions $\phi_{s(e_0, x)}$ parametrized by x . On the right is the flowchart for the x -th machine in the family. As in the prior example note that x is hard-coded into its source, so it is a single-input machine. This $\mathcal{P}_{s(e_0, x)}$ has the property that there exists an input y such that the machine outputs a 7 if and only if \mathcal{P}_x halts on x .

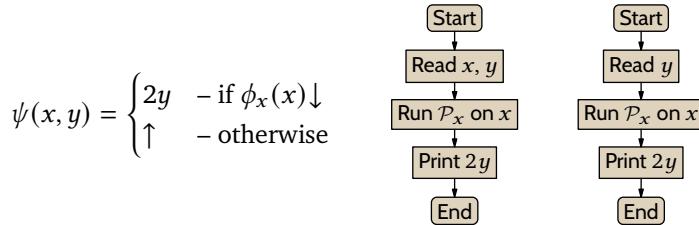
Then $\phi_x(x) \downarrow$ if and only if $\text{outputs_seven_decider}(s(e_0, x)) = 1$. If the function $\text{outputs_seven_decider}$ were computable then because the composition of two computable functions $\text{outputs_seven_decider} \circ s$ is computable we would have that the Halting problem is computably solvable, which is not right. Therefore $\text{outputs_seven_decider}$ is not computable.

- 5.7 EXAMPLE We next show that this problem is unsolvable: given e , decide whether ϕ_e is the doubler function, that is, whether $\phi_e(y) = 2y$ for all y .

We will show that this function is not computable.

$$\text{doubler_decider}(e) = \begin{cases} 1 & - \text{if } \phi_e(y) = 2y \text{ for all } y \\ 0 & - \text{otherwise} \end{cases}$$

The function on the left below is intuitively computable by the flowchart in the middle.



So Church's Thesis says that there is a Turing machine that computes it. Let that machine's index be e_0 . Apply the $s\text{-}m\text{-}n$ theorem to get a family of functions $\phi_{s(e_0, x)}$ parametrized by x . The machine $P_{s(e_0, x)}$ is sketched by the flowchart on the right. Then $\phi_x(x) \downarrow$ if and only if $\text{doubler_decider}(s(e_0, x)) = 1$. So the supposition that `doubler_decider` is computable implies that the Halting problem is computably solvable, which is wrong.

These examples show that the Halting problem serves as a touchstone for unsolvability: often we prove that something is unsolvable by demonstrating that if we could solve it then we could solve the Halting problem. We say that the Halting problem **reduces to** the given problem.[†] Thus for instance the Halting problem reduces to the problem of determining whether a given Turing machine halts on input 3.

Discussion The unsolvability of the Halting problem is one of the most important results in the Theory of Computation. Before closing this section we will make a few points.

First, to reiterate, when we say that a problem is unsolvable we mean that it is unsolvable by a mechanism, that no Turing machine computes the solution to the problem. There may be functions that solve it, but they are not effectively computable.

Second, the fact that the Halting problem is unsolvable does not mean that, given any program, we cannot tell if it halts. Obviously this program halts for every input.

```
> (define (successor i)
  (+ 1 i))
```

Nor does the unsolvability of the Halting problem mean that we cannot tell if a program does not halt. This one,

```
> (define (f x)
  (displayln x)
  (f (+ 1 x)))
```

once started, just keeps going (below, control-C interrupted the run).

[†]Often newcomers get this terminology backwards. We are using 'reduces to' in the same sense that we would in saying in Calculus, "finding the area under the graph of a polynomial reduces to antidifferentiating that polynomial," meaning that if we can do the latter then we can do the former.

```
> (f 0)
0
1
...
97806
97807
; user break [,bt for context]
```

Instead, the unsolvability of the Halting problem says that there is no single program that for all e correctly decides in a finite time whether \mathcal{P}_e halts on input e .

That sentence contains the qualifier ‘single program’ because for any index e , either \mathcal{P}_e halts on e or else it does not and thus for any e one of these two programs produces the right answer.

```
(define (yes e)
  (display 1))
```

```
(define (no e)
  (display 0))
```

Of course, guessing which one applies is not what we have in mind for solving the Halting problem. We want uniformity. We want a single effective procedure, one program, that inputs e and that outputs the right answer.

The sentence above also includes the qualifier ‘finite time’. We could write code that reads an input e and simulates \mathcal{P}_e on input e . This is a uniform approach because it is a single program. If \mathcal{P}_e on input e halts then our code would discover that. But if it does not then our code would not get that result in a finite time.

In short, the second point is that the unsolvability of the Halting Problem is about the non-existence of a single program that works across all indices. The result speaks to uniformity, or rather, it says that uniformity is impossible.

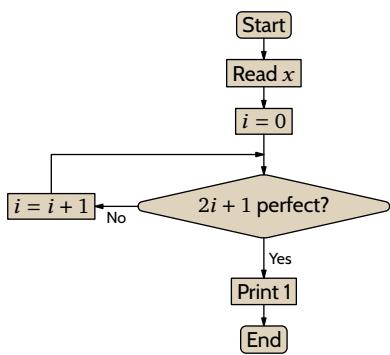
Our third point is about why unsolvability of the Halting problem is so important in the subject. A beginning programming class could leave the impression that if a program doesn’t halt then it just has a bug, something fixable. So it could seem to a student in that course that the Halting problem is not interesting.

That impression is wrong. Imagine that we could somehow write a utility `always_halt` that inputs any source \mathcal{P} and adjusts it so that for any input where \mathcal{P} does not halt, the modified program will halt (with some nominal output) but the utility does not change any outputs where \mathcal{P} does halt. That would give a list of total functions like the one on page 87, and diagonalization would give a contradiction. Thus, in any general computational scheme there must be some computations that halt on all inputs, some that halt on no inputs, and some that halt on a proper subset of inputs but not on the rest. Unsolvability of the Halting problem is inherent in the nature of computation.

This alone is enough to justify study of the problem but we will give another reason. If we had a computable `halt_decider` then we could solve many other problems. Some we saw above in this section but there are others that we currently don’t know how to solve that involve unbounded search.

For instance, a **perfect number** is a natural number that is the sum of its proper positive divisors. An example is that 6 is perfect because $6 = 1 + 2 + 3$. Another is

$28 = 1 + 2 + 4 + 7 + 14$. The next two perfect numbers are 496 and 8128. These numbers have been studied since Euclid and today we understand the form of all even perfect numbers. But no one knows if there are any odd perfect numbers.[†]



With a solution to the Halting Problem we could settle the question. This program searches for an odd perfect number.[‡] If it finds one then it halts. If not then it does not halt. So if we had a `halt_decider` and we gave it the index of this program, then that would settle whether there exists any odd perfect numbers.

There are many open questions that would fall to this approach. Just to name one more: no one knows if there is any $n > 4$ such that $2^{(2^n)} + 1$ is prime. We could answer by writing a \mathcal{P} to search for one and give its index to `halt_decider`.

Before moving to the last point, note that unbounded search is a theme in this subject. We saw it in defining general recursion using μ recursion. And, the obvious way to test whether $\phi_e(e) \downarrow$ is to search for a stage at which the computation halts. It even lurks on this book's first page, since the natural algorithm for the *Entscheidungsproblem* is to, given a statement, start with the axioms and do a breadth-first enumeration of the theorems, looking for the given one. Turing and Church independently used this section's approach to show that the *Entscheidungsproblem* is unsolvable because if there were a computable way to answer all mathematical questions then we could mathematically express questions about Turing machines halting and thereby solve the Halting problem.

Our final point is to make a contrast, that not every problem involving Turing machines is unsolvable. There are problems beginning with “Given e ” that we can do. One is: given e , decide whether q_0BLq_1 is an instruction in \mathcal{P}_e . Note the difference between this and the ones earlier in this section. The unsolvable ones are about the behavior of the machine but the solvable one is about the machine’s source — about \mathcal{P}_e rather than ϕ_e . This contrast is like the difference in languages between syntax and semantics.

This point brings us back to the opening of the first chapter where we said that we are most interested in the input-output behavior of the machines, in what they do, and less interested in things such as internal program structure.

II.5 Exercises

5.8 Someone in your class asks the professor, “I don’t get the point of the Halting problem. If you want programs to halt then just watch them and when they exceed a set number of cycles, send a kill signal.” How to respond?

[†]People have done computer checks up to 10^{1500} and not found any. [‡]This program takes an input x but ignores it; in this book we find it convenient to have the machines that we use take an input and give an output.

- 5.9 True or false: there is no function that solves the Halting Problem, that is, there is no f such that $f(e) = 1$ if $\phi_e(e) \downarrow$ and $f(e) = 0$ if $\phi_e(e) \uparrow$.
- ✓ 5.10 Your study partner asks you, “The Turing machine $\mathcal{P} = \{q_0\text{BB}q_0, q_011q_0\}$ fails to halt for all inputs, that’s obvious. But these unsolvability results say that I cannot know that. Why not?” Explain what they missed.
- 5.11 A person in your class asks, “What is wrong with this approach to solving the Halting problem? For any given Turing machine there are a finite number of states, isn’t that right? And the tape alphabet is finite, right? So there are only finitely many state and character pairs that can happen. As the machine runs, just monitor it for a repeat of some pair. A repeat means that the machine is looping, and so it won’t halt. No repeat, no loop.” What are they missing?
- 5.12 (This is related to the prior exercise.) Would it be possible for a computer to detect infinite loops and subsequently stop the associated process, or would implementing such logic be solving the Halting problem? Specifically, could the runtime environment do this: after each instruction is executed, it makes a snapshot of all of the relevant memory, the stack and heap data, the registers, the instruction pointer, etc., and before executing a instruction it checks its snapshot against all prior ones, and if there is a repeat then it declares that the program is in an infinite loop?

5.13 This is the **hailstone function**, which inputs natural numbers.

$$h(n) = \begin{cases} 1 & - \text{if } n = 0 \text{ or } n = 1 \\ h(n/2) & - \text{if } n \text{ is even} \\ h(3n + 1) & - \text{else} \end{cases}$$

The **Collatz conjecture** is that $h(n) = 1$ for all $n \in \mathbb{N}$, that is, $h(n)$ halts in that it does not keep expanding forever. No one knows whether the Collatz conjecture is true. Is it an unsolvable problem to determine whether h halts on all input?

- ✓ 5.14 True or false?
- (A) The problem of determining, given e , whether $\phi_e(3) \downarrow$ is unsolvable because no function `halts_on_three_decider` exists.
 - (B) The existence of unsolvable problems indicates weaknesses in the models of computation, and we need stronger models.

5.15 A set is computable if its characteristic function is a computable function. Consider the set consisting of 1 if Mallory reached the summit of Everest in 1924, and otherwise consisting of 0. Is that set computable?

- 5.16 Describe the family of computable functions that you get by using the *s-m-n* Theorem to parametrize x in each function. Also give flowcharts sketching the associated machines for $x = 0$, $x = 1$, and $x = 2$. (A) $f(x, y) = 3x + y$
- (B) $f(x, y) = xy^2$ (C) $f(x, y) = \begin{cases} x & - \text{if } x \text{ is odd} \\ 0 & - \text{otherwise} \end{cases}$

5.17 Show that each of these is a solvable problem.

- (A) Given an index e , determine whether Turing machine \mathcal{P}_e runs for at least 42 steps on input 3.
- (B) Given an index e , determine whether \mathcal{P}_e runs for at least 42 steps on input e .
- (C) Given n , decide whether \mathcal{P}_e runs for at least e steps on input e .

Each exercise from 5.18 through 5.24 states a problem. Show that the problem is unsolvable by reducing the Halting problem to it.

- ✓ 5.18 *See the instructions above.* Given an index e , determine if ϕ_e is total, that is, if it converges on every input.
- ✓ 5.19 *See the instructions above.* Given an index e , decide if the Turing machine \mathcal{P}_e squares its input. That is, decide if ϕ_e maps $y \mapsto y^2$.
- 5.20 *See the instructions above.* Given e , determine if the function ϕ_e returns the same value on two consecutive inputs, so that $\phi_e(y) = \phi_e(y+1)$ for some $y \in \mathbb{N}$.
- ✓ 5.21 *See the instructions above.* Given e , decide whether ϕ_e fails to converge on input 5.
- 5.22 *See the instructions above.* Given an index, determine if the computable function with that index fails to converge on all odd numbers.
- 5.23 *See the instructions above.* Given e , decide if the function ϕ_e computed by machine \mathcal{P}_e has the action $x \mapsto x + 1$.
- 5.24 *See the instructions above.* Given e , decide if the function ϕ_e fails to converge on both inputs x and $2x$, for some x .
- ✓ 5.25 For each problem, fill in the blanks to show that it is unsolvable.

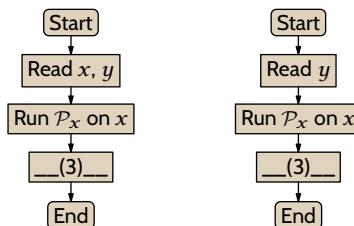
We will show that this is not mechanically computable.

$$\underline{(1)}\text{-decider}(e) = \begin{cases} 1 & - \text{if } \underline{(2)} \\ 0 & - \text{otherwise} \end{cases}$$

For that, consider this function.

$$\psi(x, y) = \begin{cases} \underline{(3)} & - \text{if } \phi_x(x) \downarrow \\ 0 & - \text{otherwise} \end{cases}$$

The flowchart on the left shows that ψ is intuitively mechanically computable.



By Church's Thesis there is a Turing machine with that behavior. Let that machine have index e_0 , so that $\psi(x, y) = \phi_{e_0}(x, y)$. Apply the s-m-n Theorem to parametrize x . A member of the resulting family of Turing machines is sketched above on the right.

Observe that $\phi_x(x) \downarrow \iff \text{decider}(s(e_0, x)) = 1$. Because the function s is mechanically computable, if decider were to be mechanically computable then the right side would be mechanically computable. But the left side is not mechanically computable, by the unsolvability of the Halting problem. Therefore decider is not mechanically computable.

- (A) Given machine index e , decide if there is an input y so that \mathcal{P}_e outputs y .
- (B) Given e , decide if there is a y so that $\phi_e(y) = 42$.
- (C) Given e , decide if there is a y so that $\phi_e(y) = y + 2$.

5.26 Fix integers $a, b, c \in \mathbb{Z}$. Consider the problem of determining, given $\text{cantor}(x, y)$, whether $ax + by = c$. Is that problem solvable or unsolvable?

5.27 For each problem, state whether it is solvable, unsolvable, or you cannot tell. You needn't give a proof, just decide. (A) Given e , decide if \mathcal{P}_e halts on all even numbers y . (B) Given e , decide if \mathcal{P}_e halts on three or fewer inputs y . (C) Given e , decide if \mathcal{P}_e halts on input e . (D) Given e , decide if \mathcal{P}_e contains an instruction with state q_e .

5.28 In some ways a more natural set than $K = \{x \in \mathbb{N} \mid \phi_x(x) \downarrow\}$ is $K_0 = \{\langle e, x \rangle \in \mathbb{N}^2 \mid \phi_e(x) \downarrow\}$. Use the fact that K is not computable to prove that K_0 is not computable.

5.29 The Halting problem of determining membership in the set $K = \{x \mid \phi_x(x) \downarrow\}$ cuts across all Turing machines.

- (A) Produce a single Turing machine, \mathcal{P}_e , such that the question of determining membership in $\{y \mid \phi_e(y) \downarrow\}$ is undecidable.
- (B) Fix a number y . Show that the question of whether \mathcal{P} halts on y is decidable.

✓ 5.30 For each, if it is mechanically solvable then sketch a algorithm to solve it. If it is unsolvable then show that.

- (A) Given e , determine the number of states in \mathcal{P}_e .
- (B) Given e , determine whether \mathcal{P}_e halts when the input is the empty string.
- (C) Given e , determine if \mathcal{P}_e halts on input n within one hundred steps.

5.31 Is K infinite?

5.32 True or false: the number of unsolvable problems is countably infinite.

5.33 Show that for any Turing machine, the problem of determining whether it halts on all inputs is solvable.

5.34 **Goldbach's conjecture**, is that every even natural number greater than two is the sum of two primes. It is one of the oldest and best-known unsolved problems in mathematics. Show that if we could solve the Halting problem then we could in principle settle Goldbach's conjecture.

5.35 **Brocard's problem** asks whether there are any numbers for which $n! + 1$ is a perfect square besides 4, 5, and 7 (no other solutions exist up to a quadrillion). Show that if we could solve the Halting problem then we could in principle settle this problem.

5.36 If we could solve the Halting problem then could we solve all problems?

5.37 Show that most problems are unsolvable by showing that there are uncountably many functions $f: \mathbb{N} \rightarrow \mathbb{N}$ that are not computed by any Turing machine, while the number of function that are computable is countable.

5.38 Give an example of a computable function that is total, meaning that it converges on all inputs, but whose range is not computable.

5.39 A set of bit strings is a **decidable language** if its characteristic function is computable. Prove each. (A) The union of two decidable languages is a decidable language. (B) The intersection of two decidable languages is a decidable language (C) The complement of a decidable language is a decidable language.

SECTION

II.6 Rice's Theorem

Our finishing point in the prior section was that the results and examples there give the intuition that we cannot mechanically analyze the behavior of Turing machines. In this section we will make this intuition precise.

Mechanical analysis does apply to some properties of Turing machines. We can write a routine that, given e , determines whether or not \mathcal{P}_e has a four-tuple instruction whose first entry is the state q_5 . The analogue in ordinary programming is that we can write a program to parse source code for a variable named `x1`. But these are not what we mean by “behavior.” Instead, they are properties of the implementation.

A property is **semantic** if it has to do with the meaning of a thing; in our case, with what the machine does.[†] Briefly, a property is semantic if it is about ϕ more than \mathcal{P} . For example, consider a Turing machine \mathcal{P} that inputs a natural number and outputs 1 if the number is prime and 0 otherwise. This machine’s behavior is that it acts as the characteristic function of the set of primes. In contrast, suppose that it has a state q_5 but no state q_6 . If we change all of the q_5 ’s in its transition table to q_6 ’s then we get a machine that is different but with the same behavior.

6.1 **DEFINITION** Two computable functions have the same behavior, $\phi_e \simeq \phi_{\hat{e}}$, if they converge on the same inputs $x \in \mathbb{N}$ and when they do converge, they have the same outputs.[†]

6.2 **DEFINITION** A set \mathcal{I} of natural numbers is an **index set**[‡] when for all $e, \hat{e} \in \mathbb{N}$, if $e \in \mathcal{I}$ and $\phi_e \simeq \phi_{\hat{e}}$ then also $\hat{e} \in \mathcal{I}$.

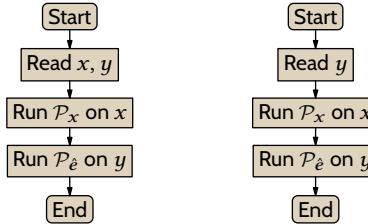
[†]Often writers contrast these with properties that are **syntactic**, which have to do with how a formal object is constructed. We covered the syntax of Turing machines in the book’s first section. [†]Strictly speaking, we don’t need the symbol \simeq . By definition, a function is a set of ordered pairs. If $\phi_e(0) \downarrow$ while $\phi_e(1) \uparrow$ then the set ϕ_e contains a pair with first entry 0 but no pair starting with 1. Thus for partial functions, if they converge on the same inputs and when they do converge they have the same outputs, then we can simply say that the two are equal, $\phi = \hat{\phi}$. But we use \simeq as a reminder that the functions may be partial. [‡]It is called an index set because it is a set of indices.

- 6.3 EXAMPLE If we fix a behavior and consider the indices of all of the Turing machines with that behavior then we get an index set. Thus the set $\mathcal{I} = \{e \in \mathbb{N} \mid \phi_e(x) = 2x \text{ for all } x\}$ is an index set. To verify, suppose that $e \in \mathcal{I}$ and that $\hat{e} \in \mathbb{N}$ is such that $\phi_e \simeq \phi_{\hat{e}}$. Then $\phi_{\hat{e}}$ also doubles its input: $\phi_{\hat{e}}(x) = 2x$ for all x . Thus $\hat{e} \in \mathcal{I}$ also.
- 6.4 EXAMPLE We can also get an index set by collecting multiple behaviors together. The set $\mathcal{J} = \{e \in \mathbb{N} \mid \phi_e(x) = 3x \text{ for all } x, \text{ or } \phi_e(x) = x^3 \text{ for all } x\}$ is an index set. For, suppose that $e \in \mathcal{J}$ and that $\phi_e \simeq \phi_{\hat{e}}$ where $\hat{e} \in \mathbb{N}$. Because $e \in \mathcal{J}$, either $\phi_e(x) = 3x$ for all x or $\phi_e(x) = x^3$ for all x . From $\phi_e \simeq \phi_{\hat{e}}$ we know that either $\phi_{\hat{e}}(x) = 3x$ for all x or $\phi_{\hat{e}}(x) = x^3$ for all x , and consequently $\hat{e} \in \mathcal{J}$.
- 6.5 EXAMPLE The set $\{e \in \mathbb{N} \mid \mathcal{P}_e \text{ contains an instruction starting with } q_{10}\}$ is not an index set. We can easily produce two Turing machines having the same behavior where one machine contains such an instruction while the other does not.
- 6.6 THEOREM (RICE'S THEOREM) Every index set that is not trivial, that is not empty and not all of \mathbb{N} , is not computable.

Proof Let \mathcal{I} be an index set. Choose an $e \in \mathbb{N}$ so that $\phi_e(y) \uparrow$ for all y . Then either $e \in \mathcal{I}$ or $e \notin \mathcal{I}$. We shall show that in the second case \mathcal{I} is not computable. The first case is similar and is Exercise 6.34.

So assume $e \notin \mathcal{I}$. Since \mathcal{I} is not empty there is an index $\hat{e} \in \mathcal{I}$. Because \mathcal{I} is an index set, $\phi_e \neq \phi_{\hat{e}}$. Thus there is an input y such that $\phi_{\hat{e}}(y) \downarrow$.

Consider the flowchart on the left below. Note that \hat{e} is not an input, it is the number from the prior paragraph. By Church's Thesis there is a Turing machine with that behavior, let it be \mathcal{P}_{e_0} . Apply the *s-m-n* theorem to parametrize x , resulting in the uniformly computable family of functions $\phi_{s(e_0, x)}$ whose computation is outlined on the right.

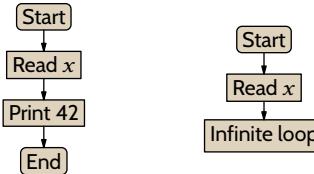


We've constructed the machine on the right so that if $\phi_x(x) \uparrow$ then $\phi_{s(e_0, x)} \simeq \phi_e$ and thus $s(e_0, x) \notin \mathcal{I}$. Further, if $\phi_x(x) \downarrow$ then $\phi_{s(e_0, x)} \simeq \phi_{\hat{e}}$ and thus $s(e_0, x) \in \mathcal{I}$. It follows that if \mathcal{I} were mechanically computable, so that we could effectively check whether $s(e_0, x) \in \mathcal{I}$, then we could solve the Halting problem. \square

- 6.7 EXAMPLE We will use Rice's Theorem to show that this problem is unsolvable: given e , decide if $\phi_e(3) \downarrow$. We must define an appropriate set \mathcal{I} and then verify that it is not empty, that it is not all of \mathbb{N} , and that it is an index set.

Let $\mathcal{I} = \{e \in \mathbb{N} \mid \phi_e(3) \downarrow\}$. The simplest way to verify that this set is not empty is to exhibit a member. The routine sketched on the left below is intuitively

computable and so Church's Thesis says there is a Turing machine with that behavior. That machine's index is a member of \mathcal{I} and thus $\mathcal{I} \neq \emptyset$.



Likewise, to verify that \mathcal{I} does not contain every number, consider the routine on the right. Church's Thesis gives that there is a Turing machine with that behavior. That machine's index is not a member of \mathcal{I} and so $\mathcal{I} \neq \mathbb{N}$.

We finish by verifying that \mathcal{I} is an index set. Assume that $e \in \mathcal{I}$ and let $\hat{e} \in \mathbb{N}$ be such that $\phi_e \simeq \phi_{\hat{e}}$. Because $e \in \mathcal{I}$, we have that $\phi_e(3) \downarrow$. Because $\phi_e \simeq \phi_{\hat{e}}$, we have that $\phi_{\hat{e}}(3) \downarrow$ also, and thus $\hat{e} \in \mathcal{I}$. Hence, \mathcal{I} is an index set.

The above example is the same problem as in the first example of the prior subsection. Note that Rice's Theorem makes the answer considerably simpler. (Of course, our development of the theorem requires the prior section's work.)

- 6.8 EXAMPLE We can use Rice's Theorem to show that the prior section's second problem is unsolvable: given e , decide if $\phi_e(x) = 7$ for some x . Rice's Theorem asks us to produce an appropriate \mathcal{I} and verify that it is nontrivial and that it is an index set.

Let $\mathcal{I} = \{ e \in \mathbb{N} \mid \phi_e(x) = 7 \text{ for some } x \}$. This set is not empty because there is a Turing machine that acts as the identity function, so that $\phi(x) = x$, and the index of that machine is a member of \mathcal{I} . This set is not all of \mathbb{N} because there is a Turing Machine that never halts, $\phi(x) \uparrow$ for all x , and that machine's index is not a member of \mathcal{I} . Hence \mathcal{I} is nontrivial.

To show that \mathcal{I} is an index set, assume that $e \in \mathcal{I}$ and let $\hat{e} \in \mathbb{N}$ be such that $\phi_e \simeq \phi_{\hat{e}}$. By the assumption, $\phi_e(x_0) = 7$ for some input x_0 . Since the two have the same behavior, the same input gives $\phi_{\hat{e}}(x_0) = 7$. Consequently, $\hat{e} \in \mathcal{I}$.

- 6.9 EXAMPLE This problem is also unsolvable: given e , decide whether ϕ_e equals this.

$$f(x) = \begin{cases} 4 & - \text{if } x \text{ is prime} \\ x + 1 & - \text{otherwise} \end{cases}$$

Let $\mathcal{I} = \{ j \in \mathbb{N} \mid \phi_j = f \}$. The set \mathcal{I} is not empty because we can write a program with this behavior and so by Church's Thesis there is a Turing machine with this behavior, and its index is a member of \mathcal{I} . Also, $\mathcal{I} \neq \mathbb{N}$ because there is a Turing machine that fails to halt on any input and its index is not a member of \mathcal{I} .

To finish we argue that \mathcal{I} is an index set. So suppose that $e \in \mathcal{I}$ and that $\phi_e \simeq \phi_{\hat{e}}$. Because $e \in \mathcal{I}$ we have that $\phi_e(x) = f(x)$ for all inputs x . Because $\phi_e \simeq \phi_{\hat{e}}$ we have that $\phi_e(x) = \phi_{\hat{e}}(x)$ for all x , and so \hat{e} is also a member of \mathcal{I} . Hence, \mathcal{I} is an index set.

We close by reflecting on the significance of Rice's Theorem.

At this chapter's start we saw that unsolvable problems exist. However, the proof had the disadvantage of being a counting argument that left us without natural examples. In the prior section, starting with the Halting problem we saw unsolvable problems that are interesting in practice.

In this section the definition of index set gave us a way to formalize a ‘behavior’. It brings us back to the declaration that opened the first chapter that we wish to concentrate more what the machines do than on the implementation details. This definition led to Rice's Theorem, which says that *every* nontrivial index set is unsolvable. So we've gone from taking unsolvable problems as exotic, to taking them as things that genuinely do come up, to a point where it may seem that there are too many of them.

Of course, that characterization of Rice's Theorem is an overstatement, an overly narrow interpretation of the word interesting; we've all seen and written real-world programs with behaviors that we would at least informally characterize as interesting. Nonetheless, Rice's Theorem is especially significant for understanding what can be done with a computer.

II.6 Exercises

6.10 Your friend is confused, “According to Rice's Theorem, everything is impossible. Every property of a computer program is non-computable. But I do this supposedly impossible stuff all the time!” Help them out.

6.11 Is $\mathcal{I} = \{ e \mid \mathcal{P}_e \text{ runs for at least 100 steps on input 5} \}$ an index set?

6.12 Why does Rice's theorem not show that this problem is unsolvable: given e , decide whether $\emptyset \subseteq \{ x \mid \phi_e(x) \downarrow \}$?

6.13 True or false: the given property of machines is semantic. (A) The machine halts on input 5. (B) It has exactly four instructions. (C) It computes twice its input. (D) The number of steps that a machine takes is five times its input.

6.14 Give a trivial index set: fill in the blanks $\mathcal{I} = \{ e \mid \underline{\quad} \mathcal{P}_e \underline{\quad} \}$ so that the set \mathcal{I} is empty.

6.15 Give a trivial index set: fill in the blanks $\mathcal{I} = \{ e \mid \underline{\quad} \mathcal{P}_e \underline{\quad} \}$ so that the set \mathcal{I} is all of \mathbb{N} .

6.16 For each problem, produce the index file needed to apply Rice's Theorem. (You needn't give the entire argument, just produce the file.)

(A) Given e , determine if $\phi_e(7) = 7$ (and, of course, it converges).

(B) Given e , determine if $\phi_e(e) = e$.

(C) Given e , determine if $\phi_{2e}(y) = 7$ for any $y \in \mathbb{N}$.

(D) Given e , determine if $\phi_e(7)$ converges and is a prime number.

For each of the problems from Exercise 6.17 to Exercise 6.23, show that it is unsolvable by applying Rice's theorem. (These repeat the problems from Exercise 5.18 to Exercise 5.24.)

- ✓ 6.17 Given an index e , determine if ϕ_e is total, that is, if it converges on every input.
- ✓ 6.18 Given an index e , decide if the Turing machine \mathcal{P}_e squares its input. That is, decide if ϕ_e maps $y \mapsto y^2$.
- 6.19 Given e , determine if the function ϕ_e returns the same value on two consecutive inputs, so that $\phi_e(y) = \phi_e(y + 1)$ for some $y \in \mathbb{N}$.
- 6.20 Given an index e , determine whether ϕ_e fails to converge on input 5.
- 6.21 Given an index, determine if the computable function with that index fails to converge on all odd numbers.
- 6.22 Given an index e , decide if the function ϕ_e computed by machine \mathcal{P}_e is $x \mapsto x + 1$.
- 6.23 Given an index e , decide if the function ϕ_e fails to converge on both inputs x and $2x$, for some x .
- ✓ 6.24 Show that each of these is an unsolvable problem by applying Rice's Theorem.
 - (A) The problem of determining if a function is total, that is, if it converges on every input.
 - (B) The problem of determining if a function is partial, that is, if it fails to converge on some input.
- ✓ 6.25 For each problem, fill in the blanks to prove that it is unsolvable.

We will show that $\mathcal{I} = \{e \in \mathbb{N} \mid \underline{(1)}\}$ is a nontrivial index set. Then Rice's theorem will give that the problem of determining membership in \mathcal{I} is algorithmically unsolvable.

First we argue that $\mathcal{I} \neq \emptyset$. The routine sketched here: $\underline{(2)}$ is intuitively computable so by Church's Thesis there is such a Turing machine. That machine's index is an element of \mathcal{I} .

Next we argue that $\mathcal{I} \neq \mathbb{N}$. The sketch: $\underline{(3)}$ is intuitively computable so by Church's Thesis there is such a Turing machine. Its index is not an element of \mathcal{I} .

To finish, we show that \mathcal{I} is an index set. Suppose that $e \in \mathcal{I}$ and that \hat{e} is such that $\phi_e \simeq \phi_{\hat{e}}$. Because $e \in \mathcal{I}$, $\underline{(4)}$. Because $\phi_e \simeq \phi_{\hat{e}}$, $\underline{(5)}$. Thus, $\hat{e} \in \mathcal{I}$. Consequently \mathcal{I} is an index set.

- (A) Given e , determine if Turing machine e halts on all inputs x that are multiples of five.
- (B) Given e , decide if Turing machine e ever outputs a seven.

6.26 Define that a Turing machine **accepts** a set of bit strings $\mathcal{L} \subseteq \mathbb{B}^*$ if that machine inputs bit strings, and it halts on all inputs, and it outputs 1 if and only if the input is a member of \mathcal{L} . Show that each problem is unsolvable, using Rice's Theorem. (A) The problem of deciding, given $e \in \mathbb{N}$, whether \mathcal{P}_e accepts an infinite language. (B) The problem of deciding, given $e \in \mathbb{N}$, whether \mathcal{P}_e accepts the string 101.

6.27 Show that this problem is mechanically unsolvable: give e , determine whether there is an input x so that $\phi_e(x) \downarrow$.

6.28 We say that a Turing machine has an **unreachable state** if for all inputs,

during the course of the computation the machine never enters that state. Show that $\mathcal{J} = \{ e \mid \mathcal{P}_e \text{ has an unreachable state} \}$ is not an index set.

6.29 Your classmate says, “Here is a problem that is about the behavior of machines but is also solvable: given e , determine whether \mathcal{P}_e only halts on an empty input tape. To solve this problem, give machine \mathcal{P}_e an empty input and see whether it goes on.” Where are they mistaken?

6.30 Show that no set that is nonempty and finite is an index set.

6.31 Show that each of these is an index set.

- (A) $\{ e \in \mathbb{N} \mid \text{machine } \mathcal{P}_e \text{ halts on at least five inputs} \}$
- (B) $\{ e \in \mathbb{N} \mid \text{the function } \phi_e \text{ is one-to-one} \}$
- (C) $\{ e \in \mathbb{N} \mid \text{the function } \phi_e \text{ is either total or else } \phi_e(3) \uparrow \}$

6.32 Index sets can seem abstract. Here is an alternate characterization. The Padding Lemma on page 69 says that every computable function has infinitely many indices. Thus, there are infinitely many indices for the doubling function $f(x) = 2x$, infinitely many for the function that diverges on all inputs, etc. In the rectangle below imagine the set of all integers and group them together when they are indices of equal computable functions. The picture below shows such a partition. Select a few parts, such as the ones shown shaded. Take their union. That’s an index set.



More formally stated, consider the relation \simeq between natural numbers given by $e \simeq \hat{e}$ if $\phi_e \simeq \phi_{\hat{e}}$. (A) Show that this is an equivalence relation. (B) Describe the parts, the equivalence classes. (C) Show that each index set is the union of some of the equivalence classes. *Hint:* show that if an index set contains one element of a class then it contains them all.

6.33 Because being an index set is a property of a set, we naturally consider how it interacts with set operations. (A) Show that the complement of an index set is also an index set. (B) Show that the collection of index sets is closed under union. (C) Is it closed under intersection? If so prove that and if not then give a counterexample.

6.34 Do the $e_0 \in \mathcal{I}$ case in the proof of Rice’s Theorem, Theorem 6.6.

SECTION

II.7 Computably enumerable sets

To attack the Halting problem the natural thing is to start by simulating \mathcal{P}_0 on input 0 for a single step. Then simulate \mathcal{P}_0 on input 0 for a second step and

also simulate \mathcal{P}_1 on input 1 for one step. After that, run \mathcal{P}_0 on 0 for a third step, followed by \mathcal{P}_1 on 1 for a second step, and then \mathcal{P}_2 on 2 for one step. We cycle among the \mathcal{P}_e on e simulations, running each for a step.[†] Eventually some of these halt and the elements of K start to fill in. On computer systems this interleaving is called time-slicing but in theory discussions it is called **dovetailing**.

We are imagining a computable f such that $f(0) = e$, where it happens that \mathcal{P}_e on input e is the first of these to halt in the dovetailing, etc. The stream of numbers $f(0), f(1), \dots$ gives the elements of K .

Why won't this process solve the Halting problem? If $e \in K$ then this we will eventually find that out. But if $e \notin K$ then we will not.

Recall that a set of natural numbers is computable if its characteristic function is computable. We are contrasting that with another way to describe a set, listing its members. Definition 1.13 gives the terminology that a function f with domain \mathbb{N} 'enumerates' its range.

7.1 DEFINITION A set of natural numbers is **computably enumerable** if it is empty or if it is effectively listable, that is, if it is the range of a total computable function. Alternate terms are **recursively enumerable** (or **c.e.**, or **r.e.**), or **semicomputable** or **semidecidable**.

Picture the stream $\phi(0), \phi(1), \phi(2), \dots$ gradually filling out the set. (It may contain repeats and the numbers may appear not in ascending order.)

7.2 REMARK Some streams are particularly interesting. For example, fix a mathematical topic such as elementary number theory. Statements in that topic are strings of symbols and we can give each a number (perhaps by writing that statement in Unicode and the number is its binary encoding, prefixed with a 1 to mitigate against the ambiguity of leading 0's). Set up a process that starts with the axioms for this topic and does a breadth-first traversal of all logical derivations from those axioms. Perhaps first it combines axiom 1 with axiom 2, then next it combines axiom 1 with axiom 3, etc. In this way it generates a list of all of this topic's possible proofs. Whenever it finishes a proof, the process outputs the number of the final statement in the derivation, the proved statement.

Suppose that we have a statement from this topic that we want to prove, such as **Goldbach's conjecture** that every even number is the sum of at most two primes. We could watch the process enumerate this theory's theorems. If our statement is provable then its number will eventually appear.

7.3 LEMMA The following are equivalent for a set of natural numbers.

- It is computably enumerable, that is, either it is empty or it is the range of a total computable function.
- It is the domain of a partial computable function.
- It is the range of a partial computable function.

Proof We will show that the first two are equivalent. That the second and third are

[†]That is, we run a loop that at iteration i will simulate \mathcal{P}_e on input e for s steps, where $i = \text{cantor}(e, s)$.

equivalent is Exercise 7.34. Assume first that a set S is computably enumerable. If it is empty then it is the domain of the partial computable function that diverges on all inputs. The other possibility is that S is the range of a total computable f , and we will describe a partial computable g with domain S . To compute g : given the input $x \in \mathbb{N}$, enumerate $f(0), f(1), \dots$ and wait for x to appear as one of these values. If x does appear then halt the computation of g , with some nominal output. If x never appears then the computation of g never halts. The domain of g is S .

For the other direction, assume that S is the domain of a partial computable function \hat{g} , to show that S is computably enumerable. If it is empty then it is computably enumerable by definition.

Otherwise we must produce a total computable \hat{f} whose range is S . Fix some $s_0 \in S$. Given $n \in \mathbb{N}$, run the computations of $\hat{g}(0), \hat{g}(1), \dots, \hat{g}(n)$, each for n -many steps. Possibly some of these halt. Define $\hat{f}(n)$ to be the least k where $\hat{g}(k)$ halts within n steps and so that $k \notin \{\hat{f}(0), \hat{f}(1), \dots, \hat{f}(n-1)\}$. If no such k exists then define $\hat{f}(n) = s_0$ (this makes \hat{f} a total function).

We finish by verifying that \hat{f} is the desired enumeration. If $t \notin S$ then $\hat{g}(t)$ never converges and so t is never enumerated by \hat{f} . If $t \in S$ then eventually $\hat{g}(t)$ must converge, in some number of steps, n_t . The number t is then queued for output by \hat{f} in the sense that it will be enumerated as, at most, $\hat{f}(n_t + t)$. \square

Many authors define computably enumerable sets using the second or third items. Definition 7.1 is more natural but is also more technically awkward in that it has the clause about the empty set.

7.4 DEFINITION $W_e = \{x \mid \phi_e(x) \downarrow\}$

The contrast between computable and computably enumerable is that a set S is computable if there is a Turing machine that decides membership, that inputs a number x and decides either ‘yes’ or ‘no’ whether $x \in S$. But with computably enumerable, given some x we can set up a machine to monitor the number stream and if x appear then this machine decides ‘yes’. However this machine need not ever discover ‘no’. (That there may well be a way to get no’s, as the first item in the next result notes.) Restated, a set is computable if there is a Turing machine that recognizes both members and nonmembers while a set is computably enumerable if there is a Turing machine that recognizes members.

7.5 LEMMA (A) If a set is computable then it is computably enumerable.

(B) A set is computable if and only if both it and its complement are computably enumerable.

Proof Let $S \subseteq \mathbb{N}$ be computable so that its characteristic function is the computable function ϕ . We will produce a partial computable f whose domain is S . Begin by finding the smallest element of S by testing whether $0 \in S$, that is, whether $\phi(0) = 1$. Then test whether $1 \in S, \dots$. If this testing ever halts with a k_0 so that $\phi(k_0) = 1$, then set $f(0) = k_0$. Next, iterate: find the second smallest element of S by testing whether $k_0 + 1 \in S, k_0 + 2 \in S, \dots$ and if this testing ever halts with a

k_1 then set $f(1) = k_1$. Clearly the domain of f consists of the elements of S , even when $S = \emptyset$.

As to the second item, suppose first that S is computable. The complement S^c is also computable because its characteristic function is $\mathbb{1}_{S^c} = 1 - \mathbb{1}_S$. Then this result's first item gives that both S and S^c are computably enumerable.

Finally, suppose that both S and S^c are computably enumerable. Let S be enumerated by the function g and let S^c be enumerated by \hat{g} . Given $x \in \mathbb{N}$, we will run an effective procedure to determine whether $x \in S$. We dovetail the two enumerations: first run the computation of $g(0)$ for a step and the computation of $\hat{g}(0)$ for a step. Next run the computations of $g(0)$ and $\hat{g}(0)$ for a second step, along with the computations of $g(1)$ and $\hat{g}(1)$ for a step each. Under this procedure, eventually x will be enumerated into either S or S^c . \square

- 7.6 COROLLARY The Halting problem set K is computably enumerable. Its complement K^c is not.

Proof The set K is the domain of $f(x) = \phi_x(x)$, which is a partial computable function by Church's Thesis. If the complement K^c were computably enumerable then Lemma 7.5 would imply that K is computable, but it isn't. \square

That result gives one reason to be interested in computably enumerable sets, namely that the Halting problem set K falls into the class of computably enumerable sets, as do such sets as $\{e \mid \phi_e(3) \downarrow\}$ and $\{e \mid \text{there is an } x \text{ so that } \phi_e(x) = 7\}$. So the collection of computably enumerable sets contains lots of interesting members.

Another reason that these sets are interesting is philosophical: with Church's Thesis we can think that in a sense computable sets are the only sets that we will ever know, and computably enumerable sets are those that we at least half know.

II.7 Exercises

- ✓ 7.7 A question on the quiz asked you to define computably enumerable. A friend says that they answered, "A set that can be enumerated by a Turing machine but that is not computable." Is that right?
- ✓ 7.8 For each set, produce a function that enumerates it (A) \mathbb{N} (B) the even numbers (C) the perfect squares (D) the set $\{5, 7, 11\}$.
- 7.9 For each, produce a function that enumerates it (A) the prime numbers (B) the natural numbers whose digits are in non-increasing order (e.g., 531 or 5331 but not 513).
- 7.10 Are there computably enumerable sets that are infinite? Finite? Empty? All of the natural numbers?
- 7.11 One of these two is computable and the other is computably enumerable but not computable. Which is which?
 - (A) $\{e \mid \mathcal{P}_e \text{ halts on input 4 in less than twenty steps}\}$
 - (B) $\{e \mid \mathcal{P}_e \text{ halts on input 4 in more than twenty steps}\}$

- 7.12 Which of these sets are decidable, which are semidecidable but not decidable, and which are neither? Justify in one sentence. (A) The set of indices e such that \mathcal{P}_e takes more than 100 steps on input 7. (B) The set of indices e such that \mathcal{P}_e takes less than 100 steps on input 7.
- 7.13 Short answer: for each set state whether it is computable, computably enumerable but not computable, or neither. (A) The set of indices e of Turing machines that contain an instruction using state q_4 . (B) The set of indices of Turing machines that halt on input 3. (C) The set of indices of Turing machines that halt on input 3 in fewer than 100 steps.
- ✓ 7.14 Someone online says, “every countable set S is computably enumerable because if $f: \mathbb{N} \rightarrow \mathbb{N}$ has range S then you have the enumeration S as $f(0), f(1), \dots$ ” Explain why this is wrong.
- ✓ 7.15 The set $A_5 = \{e \mid \phi_e(5)\downarrow\}$ is not computable. Show that it is computably enumerable.
- 7.16 Show that the set $\{e \mid \phi_e(2) = 4\}$ is computably enumerable.
- 7.17 Name a set that has an enumeration but not a computable enumeration.
- 7.18 Name three sets that are computably enumerable but not computable.
- ✓ 7.19 Let $K_0 = \{\langle e, x \rangle \mid \mathcal{P}_e \text{ halts on input } x\}$. (A) Show that it is computably enumerable. (B) Show that its columns, the sets $C_e = \{\langle e, x \rangle \mid \mathcal{P}_e \text{ halts on input } x\}$, make up all of the computable enumerable sets.
- 7.20 We know that there are subsets of \mathbb{N} that are not computable. Are the computably enumerable sets the rest of the subsets?
- ✓ 7.21 Show that the set $\text{Tot} = \{e \mid \phi_e(x)\downarrow \text{ for all } x\}$ is not computable and not computably enumerable. Hint: if this collection is computably enumerable then we can get a table like the one that starts Section II.5 on the Halting problem.
- 7.22 Prove that the set $\{e \mid \phi_e(3)\uparrow\}$ is not computably enumerable.
- 7.23 Can there be a set such that the problem of determining membership in that set is unsolvable, and also the set is computably enumerable?
- 7.24 Show that the collection of computably enumerable sets is countable.
- 7.25 (A) Prove that every finite set is computably enumerable. (B) Sketch a program that inputs a finite set and returns a function enumerating the set.
- 7.26 Prove that every infinite computably enumerable set has an infinite computable subset.
- ✓ 7.27 Consider the function $\text{steps}(e)$ defined by: $\text{steps}(e)$ is the minimal number of steps so that Turing machine \mathcal{P}_e halts if started with e on its input tape, or is undefined if the machine never halts. (A) Argue that this function is partial computable. (B) Argue that it is not total. (C) Prove that it has no total extension, no total computable $f: \mathbb{N} \rightarrow \mathbb{N}$ so that if $\text{steps}(e)\downarrow$ then $\text{steps}(e) = f(e)$
- 7.28 Let f be a partial computable function that enumerates an infinite set $R \subseteq \mathbb{N}$. Produce a total computable function that enumerates R .

7.29 A set is **computable enumerable in increasing order** if there is a computable function f that is increasing: $n < m$ implies $f(n) < f(m)$, and whose range is the set. Prove that an infinite set S is computable if and only if it is computably enumerable in increasing order.

7.30 A set is **computably enumerable without repetition** if it is the range of a computable function that is one-to-one. Prove that a set is computably enumerable and infinite if and only if it is computably enumerable without repetition.

7.31 A set is **co-computably enumerable** if its complement is computably enumerable. Produce a set that is neither computably enumerable nor co-computably enumerable.

7.32 (*See also the next exercise.*) Computability is a property of sets so we can consider its interaction with set operations. (A) Must a subset of a computable set be computable? (B) Must the union of two computable sets be computable? (C) Intersection? (D) Complement?

7.33 (*See also the prior exercise.*) We can consider the interaction of computable enumerability with set operations. (A) Must a subset of a computably enumerable set be computably enumerable? (B) Must the union of two computably enumerable sets be computably enumerable? (C) Intersection? (D) Complement?

7.34 Finish the proof of Lemma 7.3 by showing that the second and third items are equivalent.

SECTION

II.8 Oracles

The problem of deciding whether a machine halts is so hard that it is unsolvable. Is this the absolutely hardest problem or are there ones that are even harder?

What does it mean to say that one problem is harder than another? We have compared problem hardness already, for instance when we considered the problem of whether a Turing machine halts on input 3. There we proved that if we could solve the halts-on-3 problem then we could solve the Halting problem. That is, we proved that halts-on-3 is at least as hard as the Halting problem. So, the idea is that one problem is at least as hard as a second one if solving the first would also give us a solution to the second.[†]

Under Church's Thesis we interpret the unsolvability of the Halting problem to say that no mechanism can answer all questions about membership in K . So if we want to answer questions about problems

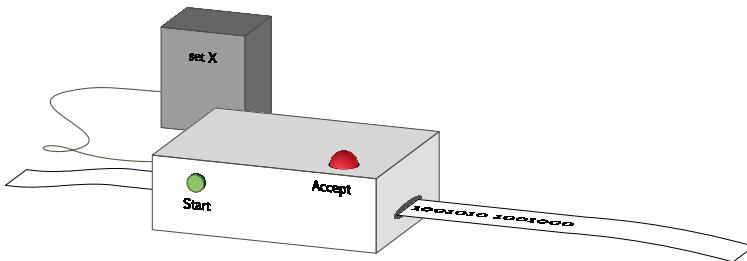


Priestess of Delphi (Collier 1891)

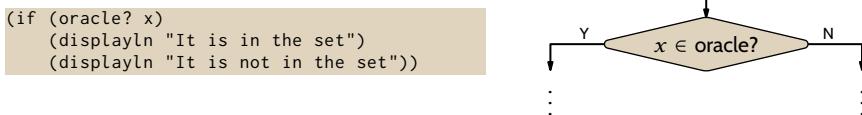
[†]We can instead conceptualize that the first problem is at least as general as the second. For instance, the problem of inputting a natural number and outputting its prime factors is at least as general as the problem of inputting a natural and determining whether it is divisible by seven.

that are related to K then we need the answers to be supplied in some way that isn't an in-principle physically realizable discrete machine[†]

Consequently, we posit an **oracle** that we attach to the Turing machine and that acts as the characteristic function of a set. Thus to see what could be computed if we could solve the Halting problem, we can attach a K -oracle that answers, “Is $x \in K?$ ” This oracle is a black box, meaning that we can't open it to see how it works.[‡]



We can formally define computation with an oracle $X \subseteq \mathbb{N}$ by extending the definition of Turing machines. But we will instead describe it conceptually. Imagine adding to a programming language a Boolean function `oracle ?`, or allowing queries to an oracle in a flowchart.[#]



We can change the oracle without changing the program code — in the picture if we swap out the black box X oracle for a Y oracle then the white box is unchanged. (Of course, the values returned by the oracle may change, which may change the tape output when we run the two-box system.)

The rest of what we have previously developed about Turing machines carries over. In particular, each such machine has an index. That index is source-equivalent, meaning that from an index we can compute the machine source and from the source we can find the index.

Therefore to fully specify a relative computation, we must specify which machine we are using and which inputs, along with specifying the oracle set. That explains the notations for the white box, the Turing machine that can make oracle calls, \mathcal{P}_e^X , and for the function computed relative to an oracle, $\phi_e^X(x)$.

[†]Turing introduced oracles in his PhD thesis. He said, “We shall not go any further into the nature of this oracle apart from saying that it cannot be a machine.” [‡]Opening it would let out the magic smoke, the stuff inside an integrated circuit that makes it work. After all, once the smoke get out, the circuit no longer works. [#]We allow a program to use one such query, or more than one, or none at all.

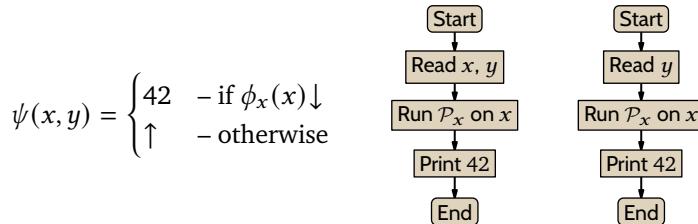
- 8.1 **DEFINITION** Let X be a set. We say that the set S is **computable from the oracle X** or **X -computable** or **computable relative to X** , or **reduces to X** or is **Turing reducible to X** , denoted $S \leq_T X$, if the characteristic function of S is one of the functions computed relative to X , that is, if $\mathbb{1}_S = \phi_e^X$ for some e .

Observe that the reduction function ϕ_e^X is total since the characteristic function is total.

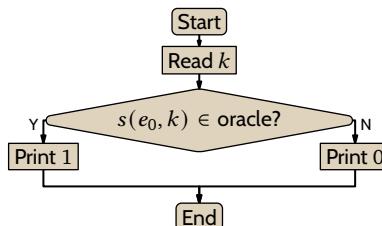
The terminology ‘ S reduces to X ’ can cause confusion.[†] The notation $S \leq_T X$ is more suggestive: X is more general than S , or “knows more” than S , in that we can get answers for questions about S by using answers to questions about X .

- 8.2 **EXAMPLE** Recall the problem of determining, given e , whether \mathcal{P}_e halts on input 3. It asks for a machine that acts as the characteristic function $\mathbb{1}_A$ of the set $A = \{e \mid \mathcal{P}_e \text{ halts on 3}\}$. We will show that $K \leq_T A$.

To satisfy the above definition we produce an oracle machine \mathcal{P}_e^K . We first reprise Example 5.4. We started the function $\psi: \mathbb{N}^2 \rightarrow \mathbb{N}$ below. The flowchart in the middle, by Church’s Thesis, shows that there is a Turing machine whose input-output behavior is ψ ; let it have index e_0 . The $s\text{-}m\text{-}n$ theorem gives a family of machines $\mathcal{P}_{s(e_0, x)}$ parametrized by x , as outlined on the right. Then $k \in K$ if and only if $s(e_0, k) \in A$, for any $k \in \mathbb{N}$.



With that we can build the oracle machine. The machine below is \mathcal{P}_e^K . It uses the constant e_0 from the prior paragraph. If it is connected to an A oracle then it computes the characteristic function of K , by the prior paragraph’s final sentence.



- 8.3 **THEOREM** (a) (REFLEXIVITY) Every set is computable from itself, $A \leq_T A$.
 (b) (TRANSITIVITY) If $A \leq_T B$ and $B \leq_T C$ then $A \leq_T C$.

Proof For Reflexivity we must show how to compute $\mathbb{1}_A$ using an A oracle. So, given a number x , we want to decide whether x is an element of A by using the

[†]We have discussed this earlier, in the Halting problem section on page 92.

A oracle. That's trivial because those are exactly the questions that the oracle answers.

For Transitivity suppose that \mathcal{P}_e^B computes the characteristic function of A and that \mathcal{P}_e^C computes the characteristic function of B . Then we can compute the characteristic function of A directly from C , by going through in the computation of A from B and replacing the B -oracle calls with calls to \mathcal{P}_e^C . \square

So the sets of natural numbers are arranged in a partial order, with some sets preceding others in the sense that S is before X if $S \leq_T X$. We next show that there are sets that come at the very beginning, not preceded by any others.

8.4 LEMMA For any $X \subseteq \mathbb{N}$, all computable sets are Turing reducible to X . In particular, $\emptyset \leq_T X$ and $\mathbb{N} \leq_T X$. Further, a set is computable if and only if it is reducible to the empty set, or to any computable set.

Proof Assume that a set is computable, so that its characteristic function is computable. That characteristic function can be computed from X using an oracle Turing machine simply by ignoring the oracle.

For the second statement suppose that the characteristic function of S can be oracle computed by reference to a computable set, so that $\mathbb{1}_S = \phi_e^X$ where $\mathbb{1}_X$ is computable. Then replacing oracle calls in the machine \mathcal{P}_e^X with direct computations of $\mathbb{1}_X$ will compute $\mathbb{1}_S$ without using an oracle. \square

8.5 DEFINITION Two sets A, B are **Turing equivalent** or **T -equivalent**, denoted $A \equiv_T B$, if both $A \leq_T B$ and $B \leq_T A$.

Showing that two sets are T -equivalent, that they are inter-computable, shows that the two seemingly-different problems are actually versions of the same problem.

We next give another example of equivalent problems, beyond pairs of computable sets. Of course, the Halting problem is to decide whether \mathcal{P}_e halts on input e . A person may perceive that there is a more natural problem, that of deciding whether \mathcal{P}_e halts on input x .

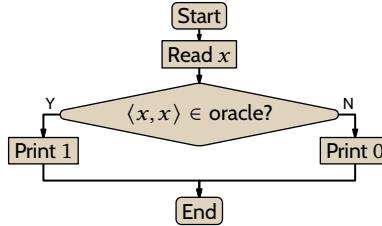
8.6 DEFINITION $K_0 = \{\langle e, x \rangle \mid \mathcal{P}_e \text{ halts on input } x\}$

We will show that the two are equivalent, that if you can solve the one problem then you can solve the other. Thus our choice of K for a touchstone problem is a matter of convenience and convention.[†]

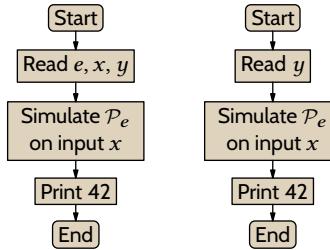
8.7 THEOREM $K \equiv_T K_0$.

Proof For $K \leq_T K_0$, suppose that we have access to a K_0 -oracle. This will determine K from that oracle.

[†]We use K because it is the standard in the literature and because it has some technical advantages, including that it falls out of the diagonalization development that we did at the start of this subsection.

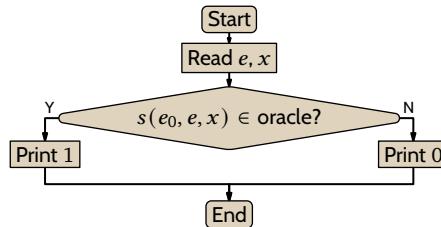


For the $K_0 \leq_T K$ half, consider the flowchart on the left below. This machine halts for all input triples $\langle e, x, y \rangle$ if and only if $\langle e, x \rangle \in K_0$. By Church's Thesis there is a Turing machine implementing it; let that machine be \mathcal{P}_{e_0} .



Get the flowchart on the right by applying the *s-m-n* theorem to parametrize e and x . That is, on the right is a sketch of $\mathcal{P}_{s(e_0, e, x)}$.

Now to make the oracle Turing machine. Given a pair $\langle e, x \rangle$, the right-side machine above $\mathcal{P}_{s(e_0, e, x)}$ either halts on all inputs y or fails to halt on all inputs, depending on whether $\phi_e(x) \downarrow$. In particular, $\mathcal{P}_{s(e_0, e, x)}$ halts on input $s(e_0, e, x)$ — that is, $s(e_0, e, x) \in K$ — if and only if $\phi_e(x) \downarrow$.



Given oracle K , this machine acts as the characteristic function of K_0 . □

8.8 COROLLARY The Halting problem is at least as hard as any computably enumerable problem: $W_e \leq_T K$ for all $e \in \mathbb{N}$.

Proof By Lemma 7.3 the computably enumerable sets are the columns of K_0 , as $W_e = \{y \mid \phi_e(y) \downarrow\} = \{y \mid \langle e, y \rangle \in K_0\}$. So $W_e \leq_T K_0 \equiv_T K$. □

Because the Halting problem is in this sense the hardest of the computably enumerable problems, we say that it is **complete** among the c.e. sets.

We have started to picture the partial order of computability. All computable sets sit together at the beginning. The Halting problem set K is as far from the

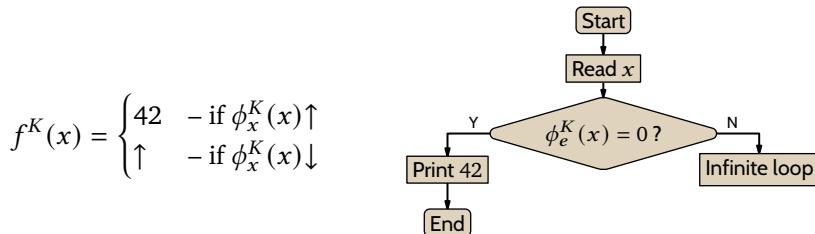
beginning as any computably enumerable set can be. We finish this section by describing a way, given a set, to jump further up the partial order than that set.

- 8.9 **THEOREM** There is no index $e \in \mathbb{N}$ such that ϕ_e^K is the characteristic function of $K^K = \{x \mid \phi_x^K(x) \downarrow\}$. That is, where the **relativized Halting problem** is the problem of determining membership in K^K , its solution is not computable from a K oracle.

Proof Assume otherwise, that there is a computation relative to a K oracle, \mathcal{P}_e^K , that acts as the characteristic function of K^K .

$$\mathbb{1}_{K^K}(x) = \begin{cases} 1 & \text{if } \phi_x^K(x) \downarrow \\ 0 & \text{otherwise} \end{cases} \quad (*)$$

We will adapt the proof that the Halting problem is unsolvable. Following that argument, consider the function below and note that it is computable relative to a K oracle. To make that clear, the flowchart illustrates the function's construction; for the branch it uses the assumption that $\mathbb{1}_{K^K}$ is computable as ϕ_e^K .

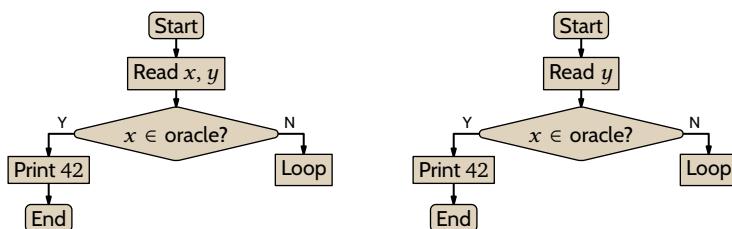


Since f is computable, it has an index. Let that index be \hat{e} , so that $f^K = \phi_{\hat{e}}^K$.

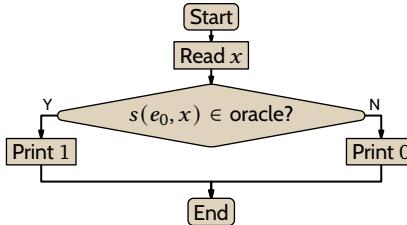
Now feed f its own index—that is, consider $f^K(\hat{e}) = \phi_{\hat{e}}^K(\hat{e})$. If that diverges then the first clause in the definition of f gives that $f^K(\hat{e}) \downarrow$, which is a contradiction. If instead $f^K(\hat{e})$ converges then the second clause in the definition of f gives $f^K(\hat{e}) \uparrow$, also a contradiction. Either way, assuming that $(*)$ can be computed relative to a K oracle gives an impossibility. \square

- 8.10 **THEOREM** Any set S is reducible to its relativized Halting problem, $S \leq_T K^S$.

Proof On the left below is an oracle machine that is intuitively mechanically computable. So Church's Thesis gives that it has an index: it is $\mathcal{P}_{e_0}^X$ for some e_0 . Use the $s\text{-}m\text{-}n$ theorem to parametrize x , giving the uniformly computable family of machines $\mathcal{P}_{s(e_0, x)}^X$ charted on the right.



The machine $\mathcal{P}_{s(e_0,x)}^X$ halts for any input y if and only if x is a member of the oracle. Taking the oracle to be S and y to be $s(e_0, x)$ gives: $x \in S$ if and only if $\phi_{s(e_0,x)}^S(s(e_0, x)) \downarrow$, which in turn holds if and only if $s(e_0, x) \in K^S$. So if K^S is the oracle for the following machine



then it acts as the characteristic function of S . Thus $S \leq_T K^S$. \square

8.11 COROLLARY $K \leq_T K^K$, but $K^K \not\leq_T K$

Proof This follows from the prior result, and the one before that. \square

The start of this section asks whether there are any problems harder than the Halting problem. We now have the answer: one problem strictly harder than computing the characteristic function of K is to compute the characteristic function of K^K .

II.8 Exercises

Recall from page 11 that a Turing machine is a decider for a set if it computes the characteristic function of that set.

- ✓ 8.12 Suppose that the set A is Turing-reducible to the set B . Which of these are true?
 - (A) A decider for A can be used to decide B .
 - (B) If A is computable then B is computable also.
 - (C) If A is uncomputable then B is uncomputable too.
- ✓ 8.13 Both oracles and deciders take in a number and return 0 or 1, giving whether that number is in the set. What's the difference?
- ✓ 8.14 Your friend says, “Oracle machines are not real, so why talk about them?” What do you say?
- 8.15 Your classmate says they answered a quiz question to define an oracle with, “A set to solve unsolvable problems.” Give them a gentle critique.
- 8.16 Is there an oracle for every problem? For every problem is there an oracle?
- 8.17 A person in your class asks, “Oracles can solve unsolvable problems, right? And K^K is unsolvable. So an oracle like the K oracle should solve it.” Help your prof out here; suggest a response.
- 8.18 Your study partner confesses, “I don’t understand relative computation. Any halting computation must take only finitely many steps. So if there are oracle

calls there are only finitely many of them. But a finite oracle is computable, and so by Lemma 8.4 it is reducible to any set.” Give them a prompt.

8.19 Where $B \subseteq \mathbb{N}$ is a set, let $2B = \{2b \mid b \in B\}$. We will show that $B \equiv_T 2B$.

(A) Give a flowchart sketching a machine that, given access to oracle $2B$, will act as the characteristic function of B . That is, this machine witnesses that $B \leq_T 2B$.

(B) Sketch a machine that, given access to oracle B , will act as the characteristic function of $2B$. This machine witnesses that $2B \leq_T B$.

8.20 We can use oracles for things other than determining the characteristic functions of sets. Sketch a machine that, with access to the oracle $P = \{p \in \mathbb{N} \mid p \text{ is prime}\}$, will input a number and print out a list of the primes dividing that number.

✓ 8.21 The set $S = \{x \mid \phi_e(3) \downarrow \text{ and } \phi_e(4) \downarrow\}$ is not computable. Sketch how to compute it using a K oracle. That is, sketch an oracle machine that shows $S \leq_T K$. *Hint:* as in Example 8.2, you can use the $s\text{-}m\text{-}n$ theorem to produce a family of machines where the x -th member halts on all inputs if and only if $x \in S$.

✓ 8.22 For the set $S = \{e \mid \phi_e(3) \downarrow\}$, show that $S \leq_T K_0$.

✓ 8.23 Show that $K \leq_T \{x \mid \phi_x(y) = 2y \text{ for all input } y\}$. *Hint:* one way is to use the $s\text{-}m\text{-}n$ theorem to produce a family of machines where the x -th member halts on all inputs if and only if it is a doubler.

8.24 Consider the set $\{x \mid \phi_x(j) = 7 \text{ for some } j\}$.

(A) Show that it is not computable using Rice’s theorem.

(B) Sketch how to compute it using a K oracle. *Hint:* one way is to use the $s\text{-}m\text{-}n$ theorem to produce a family of machines where the x -th member halts on all inputs if and only if machine \mathcal{P}_x outputs 7 on some input.

8.25 Let $S = \{x \in \mathbb{N} \mid \phi_x(3) \downarrow \text{ and } \phi_{2x}(3) \downarrow \text{ and } \phi_x(3) = \phi_{2x}(3)\}$. Show $S \leq_T K$ by producing a way to answer questions about membership in S from a K oracle. *Hint:* one way is to apply the $s\text{-}m\text{-}n$ theorem to produce a family of machines whose x -th member halts on all inputs if and only if $x \in S$.

8.26 Recall that a computable function ϕ is total if $\phi(y) \downarrow$ for all $y \in \mathbb{N}$. The set of total functions is Tot . Show that $K \leq_T \text{Tot}$.

8.27 A computable partial function ϕ_x is **extensible** if there is a computable total function ϕ where whenever $\phi_x(y) \downarrow$ then the two agree, $\phi_x(y) = \phi(y)$. The set of extensible functions is Ext .

(A) Show that this function is not a member of Ext : if $x \in K$ then $\text{steps}(x)$ is the smallest step number s where \mathcal{P}_x halts on input x by step s , and $\text{steps}(x) \uparrow$ otherwise.

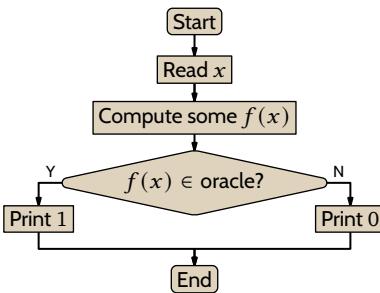
(B) Prove that $K \leq_T \text{Ext}$.

8.28 Let A and B be sets. Show that if $A(q) = B(q)$ for all $q \in \mathbb{N}$ used in the oracle computation $\phi^A(x)$ then $\phi^A(x) = \phi^B(x)$.

✓ 8.29 Prove that $A \leq_T A^c$ for all $A \subseteq \mathbb{N}$.

✓ 8.30 Show that $K \not\leq_T \emptyset$.

8.31 Assume $A \leq_T B$ and suppose that an outline of the oracle computation looks like this. (This is not the general case. The actual machine might move more than one test. Or, it might write a number of 1's and then conditionally erase all of them, or all but one of them.)



Decide whether each is True or False, and briefly explain. (A) $A^c \leq_T B$
 (B) $A \leq_T B^c$ (C) $A^c \leq_T B^c$

8.32 Is the number of oracles countable or uncountable?

8.33 Let A and B be sets. Produce a set C so that $A \leq_T C$ and $B \leq_T C$.

8.34 Fix an oracle. Prove that the collection of sets computable from that oracle is countable.

8.35 The relation \leq_T involves sets, so we naturally ask how it interacts with set operations. (A) Does $A \subseteq B$ imply $A \leq_T B$? (B) Is $A \leq_T A \cup B$? (C) Is $A \leq_T A \cap B$? (D) Is $A \leq_T A^c$?

8.36 Let $A \subseteq \mathbb{N}$. (A) Define when a set is **computably enumerable in an oracle**.
 (B) Show that \mathbb{N} is computably enumerable in A for all sets A . (C) Show that K^A is computably enumerable in A .

SECTION

II.9 Fixed point theorem

Recall our first example of diagonalization, the proof that the set of real numbers is not countable, on page 72. We assumed that there is a $f: \mathbb{N} \rightarrow \mathbb{R}$ and considered its inputs and outputs, as illustrated in this table.

n	$f(n)$'s decimal expansion
0	4 2 . 3 1 2 7 7 0 4 ...
1	2 . 0 1 0 0 0 0 0 ...
2	1 . 4 1 4 1 5 9 2 ...
3	-20 . 9 1 9 5 9 1 9 ...
:	:

Let row n 's decimal representation be $d_n = \hat{d}_n.d_{n,0}d_{n,1}d_{n,2}\dots$. Go down the diagonal to the right of the decimal point to get the sequence of digits $d_{0,0}, d_{1,1}, d_{2,2}, \dots$, which above is 3, 1, 4, 5, Using that, construct a number $z = 0.z_0z_1z_2\dots$ by making its n -th decimal place be something other than $d_{n,n}$. In our earlier example we took a digit transformation t given by $t(d_{n,n}) = 2$ if $d_{n,n} = 1$, and $t(d_{n,n}) = 1$ otherwise, so that the table above yields $z = 0.1211\dots$. Then the diagonalization argument culminates in verifying that z is not any of the rows.

When diagonalization fails What if the transformation is such that the diagonal is a row, that $z = f(n_0)$? Then the array member where the diagonal crosses that row is unchanged by the transformation, $d_{n_0,n_0} = t(d_{n_0,n_0})$. Conclusion: if diagonalization fails then the transformation has a fixed point.

We will apply this to sequences of computable functions, $\phi_{i_0}, \phi_{i_1}, \phi_{i_2} \dots$. We are interested in effectiveness so we don't consider arbitrary sequences but instead take the indices $i_0, i_1, i_2 \dots$ to be computable, meaning that for some e we have $i_0 = \phi_e(0), i_1 = \phi_e(1), i_2 = \phi_e(2)$, etc. In short, a sequence of computable functions has this form.

$$\phi_{\phi_e(0)}, \phi_{\phi_e(1)}, \phi_{\phi_e(2)} \dots$$

Below is a table with all such sequences, all effective sequences of effective functions.

		Sequence term				
		$n = 0$	$n = 1$	$n = 2$	$n = 3$	\dots
Sequence	$e = 0$	$\phi_{\phi_0(0)}$	$\phi_{\phi_0(1)}$	$\phi_{\phi_0(2)}$	$\phi_{\phi_0(3)}$	\dots
	$e = 1$	$\phi_{\phi_1(0)}$	$\phi_{\phi_1(1)}$	$\phi_{\phi_1(2)}$	$\phi_{\phi_1(3)}$	\dots
	$e = 2$	$\phi_{\phi_2(0)}$	$\phi_{\phi_2(1)}$	$\phi_{\phi_2(2)}$	$\phi_{\phi_2(3)}$	\dots
	$e = 3$	$\phi_{\phi_3(0)}$	$\phi_{\phi_3(1)}$	$\phi_{\phi_3(2)}$	$\phi_{\phi_3(3)}$	
	\vdots	\vdots	\vdots	\vdots		

Each entry $\phi_{\phi_e(n)}$ is a computable function. If the index $\phi_e(n)$ diverges then the function as whole diverges.

The natural transformation is this, where f is a computable function.

$$\phi_x \xrightarrow{t_f} \phi_{f(x)}$$

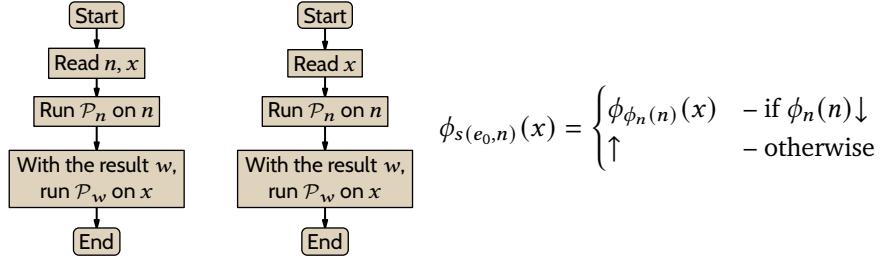
We next argue that under this transformation, diagonalization fails. Thus, the transformation t_f has a fixed point.

9.1 **THEOREM (FIXED POINT THEOREM, KLEENE 1938)[†]** For any total computable function f there is a number k such that $\phi_k = \phi_{f(k)}$.

Proof The flowchart on the left below sketches a function $f(n, x) = \phi_{\phi_n(n)}(x)$. Church's Thesis says that some Turing machine computes this function; let that

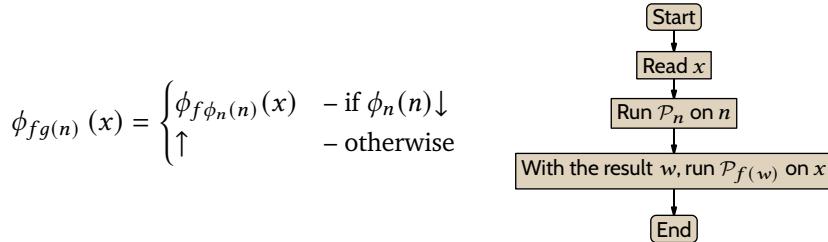
[†]This is also known as the Recursion Theorem but there is another widely used result of that name. This name is more descriptive for this result.

machine have index e_0 . Apply the $s\text{-}m\text{-}n$ theorem to parametrize n , giving the right chart, which describes a family of machines. The n -th member of that family, $\phi_{s(e_0,n)}$, computes the n -th function on the diagonal of the above array, $\phi_{\phi_0(0)}, \phi_{\phi_1(1)}, \phi_{\phi_2(2)} \dots$



The index e_0 is fixed, so $s(e_0, n)$ is a function of one variable. Let $g(n) = s(e_0, n)$, so that the diagonal functions are $\phi_{\phi_0(0)} = \phi_{g(0)}, \phi_{\phi_1(1)} = \phi_{g(1)}, \dots$. The function g is computable and total, because s is computable and total.

Under t_f those functions are transformed to $\phi_{fg(0)}, \phi_{fg(1)}, \phi_{fg(2)}, \dots$. The composition $f \circ g$ is computable and total since f is specified as total.



As the flowchart underlines, $\phi_{fg(0)}, \phi_{fg(1)}, \phi_{fg(2)}, \dots$ is a computable sequence of computable functions. Hence it is one of the table's rows. Let it be row v , making $\phi_{fg(m)} = \phi_{\phi_v(m)}$ for all m . Consider where the diagonal sequence $\phi_{g(n)}$ intersects that row: $\phi_{g(v)} = \phi_{\phi_v(v)} = \phi_{fg(v)}$. The fixed point for f is $k = g(v)$. \square

The Fixed Point Theorem says that when we try to diagonalize out of the partial computable functions, diagonalization fails. That is, the notion of partial computable function seems to have an in-built defense against diagonalization.

The Fixed Point Theorem applies to any total computable function. Consequently, it leads to many surprising results.

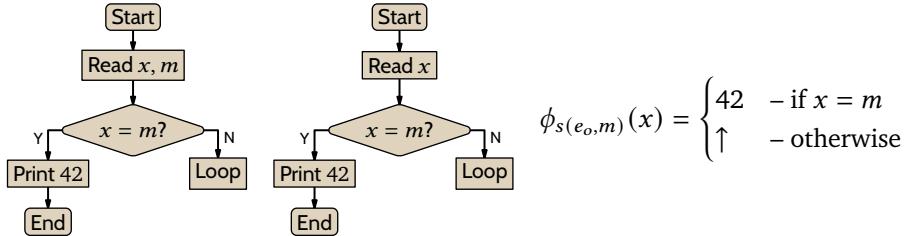
9.2 COROLLARY There is an index e so that $\phi_e = \phi_{e+1}$.

Proof The function $f(x) = x + 1$ is computable and total. So there is an $e \in \mathbb{N}$ such that $\phi_e = \phi_{f(e)}$. \square

9.3 COROLLARY There is an index e such that \mathcal{P}_e halts only on e .

Proof Consider the program described by the flowchart on the left below. By Church's Thesis it can be done with a Turing machine, \mathcal{P}_{e_0} . Parametrize to get the

program on the right, $\mathcal{P}_{s(e_0, m)}$.

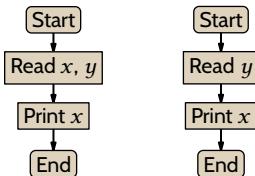


Since e_0 is fixed (it is the index of the machine on the left), $s(e_0, x)$ is a total computable function of one variable, $f(m) = s(e_0, m)$, where the associated Turing machine halts only on input m . That function has a fixed point, $\phi_{f(e)} = \phi_e$, and the associated Turing machine \mathcal{P}_e halts only on e . \square

This says that there is a Turing machine that halts only on one input, its index. Rephrased for rhetorical effect, this machine's name is its behavior.[†]

9.4 COROLLARY There is an $m \in \mathbb{N}$ such that $\phi_m(x) = m$ for all inputs x .

Proof Consider the function $\psi(x, y) = x$. As the flowchart on the left below shows, it is computable.[‡] By Church's Thesis there is a Turing machine that computes it. Let that machine have index e_0 , so that $\psi(x, y) = \phi_{e_0}(x, y) = x$.



Apply the $s\text{-}m\text{-}n$ theorem to get a uniformly computable family of functions parametrized by x , given by $\phi_{s(e_0, x)}(y) = x$. Define $f: \mathbb{N} \rightarrow \mathbb{N}$ by $f(x) = s(e_0, x)$. Because e_0 is fixed as it is the number of the Turing machine that computes ψ , this is a total function of one variable. The Fixed Point Theorem says that there is an $m \in \mathbb{N}$ with $\phi_m(y) = \phi_{f(m)}(y) = \phi_{s(e_0, m)}(y) = m$ for all y . \square

This machine's behavior is the same as its index. Imagine finding that the machine \mathcal{P}_7 outputs 7 on all inputs. This may seem to be an accident of our choice of scheme for numbering machines. But the corollary says that such a thing must happen for any acceptable machine numbering. Further, since a Turing machine's index is source-equivalent, this result suggests that there is a program that prints its own source, that self-replicates. See Extra C.

Discussion The Fixed Point Theorem and its proof are often considered mysterious, or at any rate obscure. Here we will develop a point about names.

[†]The use of 'name' for 'index' is meant to be evocative. [‡]The flowchart is overkill here since the function is obviously computable. But when it is not obvious, as in some exercises, we need an outline of how to compute the function.

Compare the sentence *Atlantis is a mythical city* with *There are two t's in 'Atlantis'*. In the first we say that 'Atlantis' is **used** because it has a value, it names something. In the second 'Atlantis' is not referring to something—its value is itself—so we say that it is **mentioned**.[†] This is the **use-mention distinction**; we are using 'Atlantis' on two different levels.

A version of this happens in computer programming. In the C language program below the second line's asterisk means that *x* and *y* are pointers. The four vertical arrays show a machine's memory cells over time. They imagine the compiler associates *x* with register 123 and *y* with 124. The first has that cell 123 holds the number 901 and cell 124 is 902.

Variables such as *x* and *y* are names for their cells. If there were an ordinary variable *a* (not a pointer) that the compiler associated with cell 122 then the statement *a = 5* would result in memory location 122 receiving a value of 5. But because these variables are pointers, we have declared interest in the contents of the memory cells that they point to: cell 123 is itself a name for location 901, and 124 names 902. The second vertical array illustrates, showing the effect of running the **x = 42* statement. The system does not put 42 into 123, rather it puts 42 into 901. Next, with *y = x* the system sets the cell named by *y* to point to the same address as *x*'s cell, address 901. Finally, the last line puts 13 where *y* points, which is at this moment the same cell to which *x* points.



Courtesy xkcd.com

9.5 ANIMATION: Pointers in a C program.

The *x* and *y* variables are being used at two different levels of meaning. On one level, *x* refers to the contents of register 123, so it names 123. But the other level is that the system is set up to refer to the contents of the contents, that is, to what's in address 901. On this level, *x* and *y* are names for names.

As to the role played by names in the Fixed Point Theorem, the proof takes $g(n)$ to be this.

$$\phi_{g(n)}(x) = \phi_{s(e_0, n)}(x) = \begin{cases} \phi_{\phi_n(n)}(x) & - \text{if } \phi_n(n) \downarrow \\ \uparrow & - \text{otherwise} \end{cases} \quad (*)$$

[†]We see this distinction in programming books. In the sentence, "The number of players is *players*" the first 'players' refers to people while the second is a program variable. The typewriter font helps with the distinction. Similarly in this book we use *a* for variables, which have a value, and *a* for characters, which are a value.

It is not the case that $g(n)$ must equal $\phi_n(n)$. Instead, $g(n)$ is an index of what is sketched in the right-hand of the pair of flowcharts in the proof. It is a name for a procedure that computes the function above.

Informally, what $g(n)$ names is, “Given input x , run \mathcal{P}_n on input n and if it halts with output w then run \mathcal{P}_w on input x .” Shorter: “Produce $\phi_n(n)$ and then do $\phi_n(n)$.”

Next, from f the proof considers the composition and gives it a name $f \circ g = \phi_v$. Putting v into the prior paragraph has $g(v)$ naming, “Compute $\phi_v(v)$ and then do $\phi_v(v)$.” That’s the same as “Compute $f \circ g(v)$ and then do $f \circ g(v)$.”

Note the self-reference: it may at first glance appear that to compute $g(v)$ we need $f \circ g(v)$, which involves computing $g(v)$, and so the instructions for $g(v)$ paradoxically contains itself as a subpart. However, that impression is wrong. Instead, $g(v)$ first computes the name, the index $s(e_0, v)$ of $f \circ g(v)$, and after that runs the machine numbered $f \circ g(v)$. So $g(v)$ and $f \circ g(v)$ are names for machines that compute the same function. Thus $g(v)$ does not contain itself—more precisely, the set of instructions for computing $g(v)$ does not contain itself. Instead, it contains a name for the instructions for computing itself.

In particular, in (*) above, regardless of whether $\phi_n(n)$ converges we can nonetheless compute the index $g(n)$ and from it the instructions for the function. There is an analogy here with ‘Atlantis’—despite that the referred-to city doesn’t exist we can still sensibly assert things about its name.

In summary, the Fixed Point Theorem is deep, showing that surprising and interesting behaviors occur in any sufficiently powerful computation system.

II.9 Exercises

- ✓ 9.6 Show each. (A) There is an index e such that $\phi_e = \phi_{e+7}$. (B) There is an e such that $\phi_e = \phi_{2e}$.
- 9.7 Show that there must be a Turing machine \mathcal{P}_e whose input-output behavior is the same as $\mathcal{P}_{\hat{e}}$, but where all of the digits in the index \hat{e} are one larger than those in e , except that a 9 in e changes to a 0 in \hat{e} . For instance we might have $\phi_{35} = \phi_{46}$.
- 9.8 What conclusion can you draw about acceptable enumerations of Turing machines by applying the Fixed Point Theorem to each of these? (A) the tripling function $x \mapsto 3x$ (B) the squaring function $x \mapsto x^2$ (C) the function that gives 0 except for $x = 5$, when it gives 1 (D) the constant function $x \mapsto 42$
- 9.9 We will prove that there is an m such that $W_m = \{x \mid \phi_m(x) \downarrow\} = \{m^2\}$.
 - (A) You want to show that there is a uniformly computable family of functions like this.

$$\phi_{s(e_0, x)}(y) = \begin{cases} 42 & - \text{if } y = x^2 \\ \uparrow & - \text{otherwise} \end{cases}$$

- Define a suitable $\psi: \mathbb{N}^2 \rightarrow \mathbb{N}$, argue that it is intuitively mechanically computable, and apply the *s-m-n* Theorem to get the family of $\phi_{s(e_0,x)}$.
- (B) Observe that e_0 is fixed so that $s(e_0,x)$ is a function of one variable only, and name that function $g: \mathbb{N} \rightarrow \mathbb{N}$.
- (C) Apply the Fixed Point Theorem to get the desired m .
- ✓ 9.10 We will show there is an index m so that $W_m = \{y \mid \phi_m(y)\downarrow\}$ is the set consisting of one element, the m -th prime number. (A) Argue that $p: \mathbb{N} \rightarrow \mathbb{N}$ such that $p(x)$ is the x -th prime is computable. (B) Use it and the *s-m-n* Theorem to get that this family of functions is uniformly computable: $\phi_{s(e_0,x)}(y) = 42$ if $y = p(x)$ and diverges otherwise. (C) Draw the desired conclusion.
- ✓ 9.11 Prove that there exists $m \in \mathbb{N}$ such that $W_m = \{y \mid \phi_m(y)\downarrow\} = 10^m$.
- 9.12 Show there is an index n so that $W_n = \{\phi_n(x)\downarrow\} = \{0, 1, \dots, n\}$.
- 9.13 The Fixed Point Theorem says that for all f (which are computable and total) there is an n so that $\phi_n = \phi_{f(n)}$. What about the statement in which we flip the quantifiers: for all $n \in \mathbb{N}$, does there exist a total and computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ so that $\phi_n = \phi_{f(n)}$?
- 9.14 Prove or disprove the existence of the set. (A) $W_m = \{\phi_m(y)\downarrow\} = \mathbb{N} - \{m\}$
 (B) $W_m = \{x \mid \phi_m(x) \text{ diverges}\}$
- 9.15 Corollary 9.3 shows that there is a computable function ϕ_n with domain $\{n\}$.
 (A) Show that there is a computable function ϕ_m with range $\{m\}$.
 (B) Is there a computable function ϕ_m with range $\{2m\}$?
- 9.16 Prove that K is not an index set. *Hint:* use Corollary 9.3 and the Padding Lemma, Lemma 2.16.

EXTRA

II.A Hilbert's Hotel

A famous mathematical fable dramatizes the question of countable and uncountable sets.

Once upon a time there was an infinite hotel. The rooms were numbered $0, 1, \dots$, naturally. One day, when every room was occupied, someone new came to the front desk; could the hotel accommodate? The clerk hit on the right idea. They moved each guest up a room, that is, the guest in room n moved to room $n + 1$, leaving room 0 empty. So this hotel always has space for a new guest, or a finite number of new guests.

Next a bus rolls in with infinitely many people p_0, p_1, \dots The clerk has the idea to move each guest to a room with twice the number, putting the guest from room n into room $2n$. Now the odd-numbered rooms are empty, so p_i can go in room $2i + 1$, and everyone has a room.

Then in rolls a convoy of buses, infinitely many of them, each with infinitely many people: $B_0 = \{p_{0,0}, p_{0,1}, \dots\}$, and $B_1 = \{p_{1,0}, p_{1,1}, \dots\}$,



Plenty of empty rooms.

etc. By now the spirit is clear: move each current guest to a new room with twice the number and the new people go into the odd-numbered rooms, in the breadth-first order that we use to count $\mathbb{N} \times \mathbb{N}$.

After this experience the clerk may well suppose that there is always room in the infinite hotel, that with a sufficiently clever method it can fit any set of guests at all. Restated, this story makes natural the guess that all infinite sets have the same cardinality. That guess is wrong. There are sets so large that their members could not all fit. One such set is \mathbb{R} .[†]

II.A Exercises

- A.1 Imagine that the hotel is empty. A hundred buses arrive, where bus B_i contains passengers $b_{i,0}, b_{i,1}$, etc. Give a scheme for putting them in rooms.
- A.2 Give a formula assigning a room to each person from the infinite bus convoy.
- A.3 The hotel builds a parking lot. Each floor F_i has infinitely many spaces $f_{i,0}, f_{i,1}, \dots$. And, no surprise, there are infinitely many floors F_0, F_1, \dots . One day the hotel is empty and buses arrive, one per parking space, each with infinitely many people. Give a way to accommodate all these people.
- A.4 The management is irked that this hotel cannot fit all of the real numbers. So they announce plans for a new hotel, with a room for each $r \in \mathbb{R}$. Can they now cover every possible set of guests?

EXTRA

II.B The Halting problem in intellectual culture

The Halting problems and the related results are about limits. Interpreted in the light of Church's Thesis, they say that there are things that we cannot do. These results had an impact on the culture of mathematics but they also had a significant impact on the wider intellectual world. We will briefly outline that impact.

Note that the statements here are in the context of the history of European intellectual culture, the context in which early Theory of Computation results appeared. A broader view is beyond our scope.

With Napoleon's downfall in the early 1800's, many people in Europe felt a swing back to a sense of order and optimism, and progress. For example, in the history of Turing's native England, Queen Victoria's reign from 1837 to 1901 seemed to many English commentators to be an extended period of prosperity and peace. Across wider Europe, people



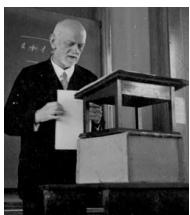
Queen Victoria opens the Great Exhibition, 1851

[†]Alas, the infinite hotel does not now exist. The guest in room 0 said that the guest from room 1 would cover both of their bills. The guest from room 1 said yes, but in addition the guest from room 2 had agreed to pay for all three rooms. Room 2 said that room 3 would pay, etc. So Hilbert's Hotel made no money despite having infinitely many rooms, or perhaps because of it.

perceived that the natural world was being tamed with science and engineering — witness the introduction of steam railways in 1825, the opening of the Suez Canal in 1869, and the invention of the electric light in 1879.[†]

In science this optimism was expressed by A A Michelson, who wrote in 1899, “The more important fundamental laws and facts of physical science have all been discovered, and these are now so firmly established that the possibility of their ever being supplanted in consequence of new discoveries is exceedingly remote.”

The twentieth century physicist R Feynman likened science to working out a game by watching it being played, “to try to understand nature is to imagine that the gods are playing some great game like chess. . . . And you don’t know the rules of the game, but you’re allowed to look at the board from time to time, in a little corner, perhaps. And from these observations, you try to figure out what the rules are of the game.” Around the year 1900 many observers thought that we basically had got the rules and that although there might remain a couple of obscure things like castling, those would be worked out soon enough.



David Hilbert
1862–1943

In Mathematics, this view was most famously voiced in an address given by Hilbert in 1930, “We must not believe those, who today, with philosophical bearing and deliberative tone, prophesy the fall of culture and accept the *ignorabimus*. For us there is no *ignorabimus*, and in my opinion none whatever in natural science. In opposition to the foolish *ignorabimus* our slogan shall be: We must know — we will know.” (*Ignorabimus*’ means ‘that which we must be forever ignorant of’ or ‘that thing we will never fully penetrate.’) There was of course a range of opinion but the zeitgeist was that we could expect that any question would be settled, and perhaps soon.[‡]

But starting in the early 1900’s, that changed. Exhibit A is the picture to the right. That the modern mastery of mechanisms can have terrible effect became apparent to everyone during World War I, 1914–1918. Ten million military men died. Overall, seventeen million people died. With universal conscription, probably the men in this picture did not want to be here. They were killed by someone who probably also did not want to be there, who never knew that he killed them, and who simply entered coordinates into a firing mechanism. If you were at those coordinates then it didn’t matter how brave you were, or how strong, or how right was your cause — you died. The zeitgeist shifted: Pandora’s box was



World War I
trench dead

[†]This is not to say the perception is justified. Disease and poverty were rampant, imperialism ruined millions of lives around the world, for much of the time the horrors of industrial slavery in the US south went unchecked, and Europe was hardly an oasis of calm, with for instance the revolutions of 1848. Nonetheless the general feeling included a sense of progress, of winning. [‡]Below we will cite some things as turning points that occur before 1930; how can that be? For one thing, cultural shifts always involve muddled timelines. And, this is Hilbert’s retirement address so we can reasonably take his as a lagging view. Finally, in Mathematics the shift occurred later than in the general culture. We mark that shift with the announcement of Gödel’s Incompleteness Theorem, discussed below. That announcement came at the same meeting as Hilbert’s speech, on the day before. Gödel was in the audience for Hilbert’s address and during it whispered to O Taussky-Todd, “He doesn’t get it.”

opened and the world was now not at all ordered, reasoned, or sensible.

At something like the same time in science, Michelson's assertion that physics was a solved problem was destroyed by the discovery of radiation. This brought in quantum theory, that has at its heart randomness, that included the uncertainty principle, and that led to the atom bomb.

With Einstein we see directly the cultural shift. After experiments during a solar eclipse in 1919 provided strong support for his theories, Einstein became an overnight celebrity. He was seen as having changed our view of the universe from Newtonian clockwork to one where "everything is relative." His work showed that the universe has limits and that old certainties break down: nothing can travel faster than light and even the commonsense idea of two things happening at the same instant falls apart.

There were many reflections of this loss of certainty. For example, the generation of writers and artists who came of age in World War I—including Fitzgerald, Hemingway, and Stein—became known as the Lost Generation. They expressed their experience through themes of alienation, isolation, and dismay. In music, composers such as Debussy and Mahler broke with the traditional forms in ways that were often hard for listeners—Stravinsky's *Rite of Spring* caused a near riot at its premiere in 1913. As for visual arts, the painting here shows the same themes.



S. Dalí's 1931 *Persistence of Memory*.



Gödel and friend, 1947

In mathematics, much the same inversion of the standing order happened in 1930 with K. Gödel's announcement of the Incompleteness Theorem. This says that if we fix a sufficiently strong formal system such as the elementary theory of \mathbb{N} with addition and multiplication then there are statements that, while true in the system, cannot be proved in that system.[†] The theory of \mathbb{N} and proving are both at the heart of mathematics, and the theorem says that they are somehow wanting.

This statement of hard limits seemed to many to be especially striking in mathematics, which held a traditional place as the most solid of knowledge. For example, I Kant said, "I assert that in any particular natural science, one encounters genuine scientific substance only to the extent that mathematics is present."

Gödel's Theorem is closely related to the Halting problem. In a mathematical proof, each step must be verifiable as either an axiom or as a deduction that is valid from the prior steps. So proving a mathematical theorem is a kind of computation.[‡] Thus, Gödel's Theorem and other uncomputability results are in the same vein. In fact, from a proof of the Halting problem, we can get to a proof of Gödel's

[†] Gödel produces a statement that asserts, in a coded way, "This statement cannot be proved." If false then it could be proved but false statements cannot be proved in the natural numbers. So it must be true. But then it, indeed, is true but cannot be proved to be so. [‡] This implies that you could start with all of the axioms and apply all of the logic rules to get a set of theorems. Then application of all of the logic rules to those will give all the second-rank theorems, etc. In this way, by dovetailing from the axioms you can in principle computably enumerate the theorems.

Theorem in a way that is reasonably straightforward. (Of course, while part of the battle is the technical steps, a larger part is the genius of envisioning the statement at all.)

To people at the time these results were deeply shocking, revolutionary. And while we work in an intellectual culture that has absorbed this shock, we must still recognize them as a bedrock.

EXTRA

II.C Self Reproduction

Where do babies come from?

Some early investigators, working without a microscope, thought that the development of a fetus is that it basically just expands, while retaining its essential features of one head, two arms, etc. Projecting backwards, they posited a *homunculus*, a small human-like figure that when given life swells to become a person.

One issue is that the person may become a parent. So each homunculus contains its children? And grandchildren? That is, one problem is infinite regress. Of course today we know that sperm and egg don't contain bodies, they contain DNA, code to create bodies.



Sperm drawing, 1695

Paley's watch In 1802, W Paley argued for the existence of a god from a perception of unexplained order in the natural world.

In crossing a heath, . . . suppose I had found a watch upon the ground . . . [W]hen we come to inspect the watch we perceive . . . that its several parts are framed and put together for a purpose, e.g., that they are so formed and adjusted as to produce motion, and that motion so regulated as to point out the hour of the day . . . the inference we think is inevitable, that the watch must have a maker—that there must have existed, at some time and at some place or other, an artificer or artificers who formed it for the purpose which we find it actually to answer, who comprehended its construction and designed its use.

The marks of design are too strong to be got over. Design must have had a designer. That designer must have been a person. That person is GOD.



William Paley

1743–1805

of all the rest.

Paley then gives his strongest argument, that the most incredible thing in the natural world, that which distinguishes living things from stones or machines, is that they can, if given a chance, self-reproduce.

Suppose, in the next place, that the person, who found the watch, would, after some time, discover, that, in addition to all the properties which he had hitherto observed in it, it possessed the unexpected property of producing, in the course of its movement, another watch like itself . . . If that construction without this property, or which is the same thing, before this property had been noticed, proved intention and art to have been employed about it; still more strong would the proof appear, when he came to the knowledge of this further property, the crown and perfection

This argument was very influential before the discovery by Darwin and Wallace of descent with modification through natural selection. It shows that from among all the things in the natural world to marvel at—the graceful shell of a nautilus, the precision of an eagle's eye, or consciousness—for many observers the greatest wonder was self-reproduction.

Many thinkers contended that self-reproduction had a special position, that mechanisms cannot self-reproduce. Picture a robot that assembles cars; it seemed plausible that this is possible only because the car is in some way less complex than the robot. In this line of reasoning, machines are only able to produce things that are less complex than themselves. But that contention is wrong. The Fixed Point Theorem gives self-reproducing mechanisms.

Quines The Fixed Point Theorem shows that there is a number m such that $\phi_m(x) = m$ for all inputs. Think of m as the function's name, so that this machine names itself; this is self-reference. Said another way, P_m 's name is its behavior.

Since we can go effectively from the index m to the machine source, in a sense this machine knows its source. A **quine** is a program that outputs its own source code. We will next step through the nitty-gritty of making a quine.[†] We will use the C language since it is low-level and so the details are not hidden.

The first thing a person might think is to include the source as a string within the source code itself. Below is a start at that, which we can call `try0.c`.[‡] But this is obviously naive. The string would have to contain another string, etc. Like the homunculus theory, this leads to an infinite regress. Instead, we need a program that somehow contains instructions for computing a part of itself.

```
main() {
    printf("main(){\n ... }");
}
```

A more sophisticated approach leverages our discussion of the Fixed Point Theorem in that it mentions the code before using it. This is `try1.c`.[#]

```
char *e="main(){printf(e);}";
main(){printf(e);};
```

Here is the printout.

```
main(){printf(e);};
```

Ratcheting up this approach gives `try2.c`.

```
char *e="main(){printf("char*e=\"");printf(e); printf("");\n");printf(e);"
main(){printf("char *e=\"");printf(e); printf("");\n");printf(e);}
```



Courtesy xkcd.com

[†]The easiest such program finds its source file on the disk and prints it. That is cheating. [‡]The backslash-n gives a newline character. [#]The `char *e="..."` construct gives a string. In the C language `printf(...)` command the first argument is a string. In that string double quotes expand to single quotes, %c takes a character substitution from any following arguments, and %s takes a string substitution.

This is close. Escaping some newlines and quotation marks[†] leads to this program, `try3.c`, which works.

```
char *e="char*e=\"%c %s %c; %c main() {printf(e,34,e,34,10,10);}%c";
main(){printf(e,34,e,34,10,10);}
```

Quines are possible in any complete model of computation; the exercises ask for them in a few languages.

Quining The verb ‘to quine’ means “to write a sentence fragment a first time, and then to write it a second time, but with quotation marks around it” For example, from ‘say’ we get “say ‘say’”. And, quining ‘quine’ gives “quine ‘quine’.”

This is a linguistic analogy of the self-reproducing programs where the word plays the role of the data, the part played by the machine \mathcal{A} or the part played by `try3.c`’s string `char *e`. In the slogan “Produce the machine, and then do the machine,” they are the ‘produce’ part. The machine \mathcal{B} plays the role of the verb ‘quine’, and is the ‘do’ part.

Reflections on Trusting Trust K Thompson is one of the two main creators of the UNIX operating system. For this and other accomplishments he won the Turing Award, the highest honor in computer science. He began his acceptance address with this.

In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. . . .

More precisely stated, the problem is to write a source program that, when compiled and executed, will produce as output an exact copy of its source. If you have never done this, I urge you to try it on your own. The discovery of how to do it is a revelation that far surpasses any benefit obtained by being told how to do it. The part about “shortest” was just an incentive to demonstrate skill and determine a winner.



Ken Thompson
b 1943

This celebrated essay develops a quine and goes on to show how the existence of such code poses a security threat that is very subtle and just about undetectable. The entire address (Thompson 1984) is widely available; everyone should read it.

II.C Exercises

- C.1 Produce a Racket quine.
- C.2 Produce a Python quine.

EXTRA

II.D Busy Beaver

For any $n \in \mathbb{N}$, the set of Turing machines having n or fewer instructions (and using states no larger than q_n) is finite. For some members of this set \mathcal{P}_e halts on

[†]The 10 is the ASCII encoding for newline and 34 is ASCII for a double quotation mark.

input e and for some members it does not but because the set is finite, the list of which machines halt is also finite. Finite sets are computable. So we are thinking about starting all of the machines having n or fewer instructions and dovetailing their computations until no more of them will ever converge. That is, consider $D: \mathbb{N} \rightarrow \mathbb{N}$, where $D(n)$ is the minimal number of steps after which all of the n -instruction machines that will ever converge have done so.

We can prove that D is not computable. For, assume otherwise. Then to compute whether \mathcal{P}_e halts on input e , find how many instructions n are in the machine \mathcal{P}_e , compute $D(n)$, and run $\mathcal{P}_e(e)$ for $D(n)$ -many steps. If $\mathcal{P}_e(e)$ has not halted by then, it never will. Of course, this contradicts the unsolvability of the Halting problem.

Now, D may seem to be just another uncomputable function; why is it especially enlightening? It has the property that any function \hat{D} that grows faster than D , where if $\hat{D}(n) \geq D(n)$ for all sufficiently large n , is also not computable, for the same reason that D is not computable. This gives us an insight into one way that functions can fail to be computable: they can grow faster than any computable function.[†]



Rare moment of rest

So, which n -line program takes the longest? Which does the computational work? The **Busy Beaver problem** is: which n -state Turing Machine leaves the most 1's after halting, when started on an empty tape?

Think of this as a competition for who can write the busiest machine. To have a computation we need rules and here it is traditional to fix a definition of Turing Machines where there is a single tape that is unbounded at one end, there are two tape symbols 1 and B, and where transitions are of the form $\Delta(\text{state}, \text{tape symbol}) = (\text{state}, \text{tape symbol}, \text{head shift})$.

Busy Beaver is unsolvable Write $\Sigma(n)$ for the largest number of 1's that any n state machine, when started on a blank tape, leaves on the tape after halting. Write $S(n)$ for the most moves, that is, transitions.

Why isn't Σ computable? The obvious thing is to do a breadth-first search: there are finitely many n -state machines, start them all on a blank tape, and await developments.

That won't work because some of the machines won't halt. At any moment you have some machines that have halted and you can see how many 1's are on each such tape, so you know the longest so far. But as to the not-yet-halted ones, who knows? You can by-hand see that this one or that one will never halt and so you can figure out the answer for $n = 1$ or $n = 2$. But there is no algorithm to decide the question for an arbitrary number of states.



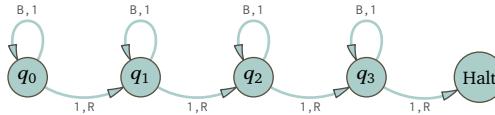
Tibor Radó
1895–1965

[†]Note the connection with the Ackermann function: we showed that it is not primitive recursive because it grows faster than any primitive recursive function.

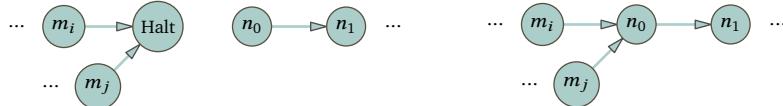
D.1 THEOREM (RADÓ, 1962) The function Σ is not computable.

Proof Let $f: \mathbb{N} \rightarrow \mathbb{N}$ be computable. We will show that $\Sigma \neq f$ by showing that $\Sigma(n) > f(n)$ for infinitely many n .

First note that there is a Turing Machine \mathcal{M}_j having j many states that writes j -many 1's to a blank tape. For instance, here is \mathcal{M}_4 .



Also note that we can compose two Turing machines. The illustration below shows two machines on the left. On the right, we have combined the final states of the first machine with the start state of the second.



Let $F: \mathbb{N} \rightarrow \mathbb{N}$ be this function.

$$F(m) = (f(0) + 0^2) + (f(1) + 1^2) + (f(2) + 2^2) + \cdots + (f(m) + m^2)$$

It has the properties: if $0 < m$ then $f(m) < F(m)$, and $m^2 \leq F(m)$, and $F(m) < F(m+1)$. It is intuitively computable so Church's Thesis says there is a Turing machine \mathcal{M}_F that computes it. Let that machine have n_F many states.

Now consider the Turing machine \mathcal{P} that performs \mathcal{M}_j and follows that with the machine \mathcal{M}_F , and then follows that with another copy of the machine \mathcal{M}_F . If started on a blank tape this machine will first produce j -many 1's, then produce $F(j)$ -many 1's, and finally will leave the tape with $F(F(j))$ -many 1's. Thus its productivity is $F(F(j))$. It has $j + 2n_F$ many states.

Compare that with the $j + 2n_F$ -state Busy Beaver machine. By definition $F(F(j)) \leq \Sigma(j + 2n_F)$. Because n_F is constant (it is the number of states in the machine \mathcal{M}_F), the relation $j + 2n_F \leq j^2 < F(j)$ holds for sufficiently large j . Since F is strictly increasing, $F(j + 2n_F) < F(F(j))$. Combining gives $f(j + 2n_F) < F(j + 2n_F) < F(F(j)) \leq \Sigma(j + 2n_F)$, as required. \square

What is known That $\Sigma(0) = 0$ and $\Sigma(1) = 1$ follow straight from the definition. (The convention is to not count the halt state, so $\Sigma(0)$ refers to a machine consisting only of a halting state.) Radó noted in his 1962 paper that $\Sigma(2) = 4$. In 1964 Radó and Lin showed that $\Sigma(3) = 6$.

D.2 EXAMPLE This is the three state Busy Beaver machine.

Δ	B	1
q_0	$q_1, 1, R$	$q_4, 1, R$
q_1	$q_2, 0, R$	$q_1, 1, R$
q_2	$q_3, 1, L$	$q_0, 1, L$

In 1983 A Brady showed that $\Sigma(4) = 107$. As to $\Sigma(5)$, even today no one knows.

Here are the current world records.

n	1	2	3	4	5	6
$\Sigma(n)$	1	4	6	13	≥ 4098	$\geq 10 \uparrow\uparrow 15$
$S(n)$	1	6	21	107	$\geq 47\,176\,870$	$\geq 10 \uparrow\uparrow 15$

The notation $10 \uparrow\uparrow 15$ means 10 raised to the 10 raised to the $10 \dots$, a total of 15 times, that is, $10^{(10^{(10^{\dots})})}$ with fifteen 10's. This result is the most recent, from 2022. For more, there are a number of websites that cover the topic, including the latest results. These may involve variations on machine standards such as restricting to two-symbol machines but even without getting into details certainly the size of the numbers is compelling.

Not only are Busy Beaver numbers very hard to compute, at some point they become impossible. In 2016, A Yedida and S Aaronson obtained an n for which $\Sigma(n)$ is unknowable. To do that, they created a programming language where programs compile down to Turing machines. With this, they constructed a 7918-state Turing machine that halts if there is a contradiction within the standard axioms for Mathematics, and never halts if those axioms are consistent. We believe that these axioms are consistent, so we believe that this machine doesn't halt. However, Gödel's Second Incompleteness Theorems shows that there is no way to prove the axioms are consistent using the axioms themselves, so $\Sigma(n)$ is unknowable in that even if we were given the number n we could not use our axioms to prove that it is right, to prove that this machine halts.

So one way for a function to fail to be computable is if it grows faster than any computable function. Note, however, that this is not the only way. There are functions that grow slower than some computable function but are nonetheless not computable.

II.D Exercises

- ✓ D.3 Give the computation history, the sequence of configurations, that come from running the three-state Busy Beaver machine. *Hint:* you can run it on the Turing machine simulator.
- ✓ D.4 (A) How many Turing machines with tape alphabet {B, 1} are there having one state? (B) Two? (C) How many with n states?
- D.5 How many Turing machines are there, with a tape alphabet Σ of n characters and having n states?
- D.6 Show that there are uncomputable functions that grow slower than some computable function. *Hint:* There are uncountably many functions with output in the set \mathbb{B} .

D.7 Give a diagonal construction of a function that is greater than any computable function.

EXTRA

II.E Cantor in Code

In this section we show that Cantor's correspondence between \mathbb{N} and $\mathbb{N} \times \mathbb{N}$ is effective. The most straightforward way to show that these functions can be computed is to exhibit code.

Recall that in this table

$n \in \mathbb{N}$	0	1	2	3	4	5	...
$\langle i, j \rangle \in \mathbb{N} \times \mathbb{N}$	$\langle 0, 0 \rangle$	$\langle 0, 1 \rangle$	$\langle 1, 0 \rangle$	$\langle 0, 2 \rangle$	$\langle 1, 1 \rangle$	$\langle 2, 0 \rangle$...

the map from top row to bottom is Cantor's pairing function because it outputs pairs, while its inverse, from bottom to top, is the unpairing function.

First, unpairing. Given $\langle x, y \rangle$, we determine the diagonal that it lies on,

```
;; triangle-num  return 1+2+3+..+n
;;  natural number -> natural number
(define (triangle-num n)
  (/ (* (+ n 1)
        n)
    2))
```

and then use that to find the value.

```
;; cantor-unpairing  Cantor number of the pair (x,y)
;;  natural number, natural number -> natural number
(define (cantor-unpairing x y)
  (let ([d (+ x y)])
    (+ (triangle-num d)
       x)))
```

Using the function is easy.

```
$ racket
Welcome to Racket v8.3 [cs].
> (require "counting.rkt")
> (cantor-unpairing 1 1)
4
> (cantor-unpairing 34 10)
1024
>
```

A person may wonder about the choice of the elements of the pair as the arguments to `cantor-unpairing`. Perhaps it should instead input the pair itself? But the `apply` operator makes the switch easy.

```
> (cantor-unpairing 10 12)
263
> (apply cantor-unpairing '(10 12))
263
```

Next, pairing. Given a natural number c , to find the associated $\langle x, y \rangle$, we first find the diagonal on which it will fall. Where the diagonal is $d(x, y) = x + y$,

let the associated triangle number be $t(x, y) = d(d + 1)/2 = (d^2 + d)/2$. Then $0 = d^2 + d - 2t$. Applying the familiar formula $(-b \pm \sqrt{b^2 - 4ac})/(2a)$ gives this.

$$d = \frac{-1 + \sqrt{1 - 4 \cdot 1 \cdot (-2t)}}{2 \cdot 1} = \frac{-1 + \sqrt{1 + 8t}}{2}$$

(Of the ‘ \pm ’, we kept only the ‘ $+$ ’ because the other root is negative.) To find the number of the diagonal containing the pair $\langle x, y \rangle$ with $\text{pair}(x, y) = c$, take the floor, $d = \lfloor(-1 + \sqrt{1 + 8c})/2\rfloor$.

```
;; diag-num Give number of diagonal containing Cantor pair numbered c
;; natural number -> natural number
(define (diag-num c)
  (let ([s (integer-sqrt (+ 1 (* 8 c)))]
    (floor (quotient (- s 1)
      2))))
```

and then

```
;; cantor-pairing Given the cantor number, return the pair with that number
;; natural number -> (natural number natural number)
(define (cantor-pairing c)
  (let* ([d (diag-num c)]
    [t (triangle-num d)])
  (list (- c t)
    (- d (- c t))))))
```

Use this in the natural way.

```
> (cantor-pairing 15)
'(0 5)
> (cantor-pairing (cantor-unpairing 10 12))
'(10 12)
```

We can reproduce the table from the section’s start.

```
> (for ([i '(0 1 2 3 4 5)])
  (display (cantor-pairing i))(newline))
(0 0)
(0 1)
(1 0)
(0 2)
(1 1)
(2 0)
```

Extending to triples is straightforward. These routines are perhaps misnamed—they might be better named `cantor-tupling-3` and `cantor-untupling-3`—but we will stick with what we have.

```
;; cantor-unpairing-3 Cantor number of a triple
;; natural number, natural number, natural number -> natural number
(define (cantor-unpairing-3 x0 x1 x2)
  (cantor-unpairing x0 (cantor-unpairing x1 x2)))

;; cantor-pairing-3 Return the triple for (cantor-unpairing-3 x0 x1 x2) => c
;; natural number -> (natural natural natural)
(define (cantor-pairing-3 c)
  (cons (car (cantor-pairing c))
    (cantor-pairing (cadr (cantor-pairing c))))))
```

Using these routines is also straightforward.

```
> (cantor-pairing-3 172)
'(1 2 3)
> (for ([i '(0 1 2 3 4 5 6 7 8 9)])
  (display (cantor-pairing-3 i))(newline))
(0 0 0)
(0 0 1)
(1 0 0)
(0 1 0)
(1 0 1)
(2 0 0)
(0 0 2)
(1 1 0)
(2 0 1)
(3 0 0)
```

Similar routines do four-tuples.

```
;; cantor-unpairing-4 Number quads
;; natural natural natural natural -> natural
(define (cantor-unpairing-4 x0 x1 x2 x3)
  (cantor-unpairing x0 (cantor-unpairing-3 x1 x2 x3)))
```

```
;; cantor-pairing-4 Find the quad that corresponds to the given natural
;; natural -> natural natural natural natural
(define (cantor-pairing-4 c)
  (let ((pr (cantor-pairing c)))
    (cons (car pr)
          (cantor-pairing-3 (cadr pr)))))
```

Here are the first few four-tuples.

```
> (for ([i '(0 1 2 3 4 5 6)])
  (display (cantor-pairing-4 i))(newline))
(0 0 0 0)
(0 0 0 1)
(1 0 0 0)
(0 1 0 0)
(1 0 0 1)
(2 0 0 0)
(0 0 1 0)
```

The routines for triples and four-tuples show that there is a general pattern so what the heck, let's extend to tuples of any size. We don't need these but they are fun.

For the function unpair: $\mathbb{N}^k \rightarrow \mathbb{N}$, which we also call cantor, we can determine k by peeking at the number of inputs. Thus `cantor-unpairing-n` generalizes `cantor-unpairing`, `cantor-unpairing-3`, etc., by taking a tuple of any length.

```
;; cantor-unpairing-n any-sized tuple Be cantor-unpairing-n where n is the tuple length
;; (natural ..) of n elets -> natural
(define (cantor-unpairing-n . args)
  (cond
    [(null? args) 0]
    [(= 1 (length args)) (car args)]
    [(= 2 (length args)) (cantor-unpairing (car args) (cadr args))]
    [else
      (cantor-unpairing (car args) (apply cantor-unpairing-n (cdr args))))]))
```

```
> (cantor-unpairing-n 0 0 1 0)
6
> (cantor-unpairing-n 1 2 3 4)
159331
```

To generalize to the function pair: $\mathbb{N} \rightarrow \mathbb{N}^k$, the difficulty is that we don't know the arity k and we must specify it separately.

```
; ; cantor-pairing-arity  return the list of the given arity making the cantor number c
; ; If arity=0 then only c=0 is valid (others return #f)
; ; natural natural -> (natural .. natural) with arity-many elements
(define (cantor-pairing-arity arity c)
  (cond
    [(= 0 arity)
     (if (= 0 c)
         '()
         (begin
           (display "ERROR: cantor-pairing-arity with arity=0 requires c=0") (newline)
           #f))]
    [(= 1 arity) (list c)]
    [else (cons (car (cantor-pairing c))
                 (cantor-pairing-arity (- arity 1) (cadr (cantor-pairing c))))]))
```

This shows the routine acting like `cantor-pairing -4`.

```
> (for ([i '(0 1 2 3 4 5 6)])
   (display (cantor-pairing-arity 4 i))(newline))
(0 0 0 0)
(0 0 0 1)
(1 0 0 0)
(0 1 0 0)
(1 0 0 1)
(2 0 0 0)
(0 0 1 0)
```

The `cantor-pairing-arity` routine is not uniform in that it covers only one arity at a time. Said another way, `cantor-unpairing-arity` is not the inverse of `cantor-pairing-n` in that we have to tell it the tuple's arity.

```
> (cantor-unpairing-n 3 4 5)
1381
> (cantor-pairing-arity 3 1381)
'(3 4 5)
```

To cover tuples of all lengths, to give a correspondence between the natural numbers and the set of sequences of natural numbers, we define two matched routines, `cantor-pairing-omega` and `cantor-unpairing-omega`.

```
> (for ([i '(0 1 2 3 4 5 6 7 8)])
   (display (cantor-pairing-omega i))(newline))
()
(0)
(0 0)
(1)
(0 1)
(0 0 0)
(2)
(1 0)
(0 0 1)
```

The idea of `cantor-pairing-omega` is to interpret its input c as a pair $\langle x, y \rangle$, that is, $c = \text{pair}(x, y)$. It returns a tuple of length $x + 1$, where y is the tuple's cantor number. (The reason for the $+1$ in $x + 1$ is that the empty tuple is associated with $c = 0$. Then rather than have all later pairs $\langle 0, y \rangle$ not be associated with any number, we next use the one-tuple $\langle 0 \rangle$, and after that we use $\langle 1 \rangle$, etc.)

```

;; cantor-pairing-omega  Inverse of cantor-unpairing-omega (but with arguments inserted)
;;   natural  -> (natural .. )
(define (cantor-pairing-omega c)
  (let* ([pr (cantor-pairing c)]
         [a (car pr)]
         [cantor-number (cadr pr)])
    (cond
      [(and (= a 0)
            (= cantor-number 0)) '()]
      [= (a 0) (list (- cantor-number 1))]
      [else (cantor-pairing-arity (+ 1 a) cantor-number)])))

```



```

;; cantor-unpairing-omega encode the arity in the first component
;;   natural natural ..  -> natural
(define (cantor-unpairing-omega . tuple)
  (let ([arity (length tuple)])
    (cond
      [= (arity 0) (cantor-unpairing 0 0)]
      [= (arity 1) (cantor-unpairing 0 (+ 1 (car tuple)))]
      [else
        (let ([newtuple (list (- arity 1)
                             (apply cantor-unpairing-n tuple))])
          (apply cantor-unpairing newtuple)))))

```

This shows their use.

```

> (cantor-unpairing-omega 1 2 3 4)
12693741448
> (cantor-pairing-omega 12693741448)
'(1 2 3 4)

```

II.E Exercises

- E.1 What is the pair with Cantor number 42?
- E.2 What is the pair with the number 666?
- E.3 What is the formula for the gaps between one-tuples?
- E.4 What is the first number matched by cantor - pairing - omega with a four-tuple?

Part Two
Automata



CHAPTER

III Languages and graphs

Turing machines input strings and output strings, sequences of tape symbols. So a natural way to work is to represent a problem as a string, put it on the tape, run a computation, and end with the solution as a string. The same happens in everyday programming. A program to finds the shortest driving distance between cities probably inputs the map distances as a bitstrings, does the same for the desired two cities, and then gives the directions as a string.

As to graphs, they are the vocabulary in which we will state many of the problems that we look to solve.

SECTION

III.1 Languages

Our machines input and output strings of symbols. We take a **symbol** (sometimes called a **token**) to be an atomic unit that a machine can read and write.[†] On everyday binary computers the symbols are the bits, 0 and 1. An **alphabet** is a nonempty and finite set of symbols. We usually denote an alphabet with the upper case Greek letter Σ , although an exception is the alphabet of bits, $\mathbb{B} = \{ 0, 1 \}$. A **string** over an alphabet is a sequence of symbols from that alphabet. We use lower case Greek letters such as σ and τ to denote strings. We use ε to denote the empty string, the length zero sequence of symbols. The set of all strings over Σ is Σ^* .[‡]

- 1.1 DEFINITION A **language** \mathcal{L} over an alphabet Σ is a set of strings drawn from that alphabet. That is, $\mathcal{L} \subseteq \Sigma^*$.
- 1.2 EXAMPLE The set of bitstrings that begin with 1 is $\mathcal{L} = \{ 1, 10, 11, 100, \dots \}$.
- 1.3 EXAMPLE Another language over \mathbb{B} is the finite set $\{ 1000001, 1100001 \}$.
- 1.4 EXAMPLE Let $\Sigma = \{ a, b \}$. The language consisting of strings where the number of a's is twice the number of b's is $\mathcal{L} = \{ \varepsilon, aab, aba, baa, aaaabb, \dots \}$.
- 1.5 EXAMPLE Let $\Sigma = \{ a, b, c \}$. The language of length-two strings over that alphabet is $\mathcal{L} = \Sigma^2 = \{ aa, ab, ba, \dots, cc \}$. Over the same alphabet, this language consists of string of length three that are in ascending order.

$$\{ aaa, bbb, ccc, aab, aac, abb, abc, acc, bbc, bcc \}$$

IMAGE: *The Tower of Babel*, by Pieter Bruegel the Elder (1563) [†]We can imagine Turing's clerk calculating without reading and writing symbols, for instance by keeping track of information by having elephants move to the left side of a road or to the right. But we could translate any such procedure into one using marks that our mechanism's read/write head can handle. So readability and writeability are not essential but we require them in the definition of symbols as a convenience; after all, elephants are inconvenient. [‡]For more on strings see the Appendix on page 362.

1.6 **DEFINITION** A **palindrome** is a string that reads the same forwards as backwards.

Some English palindromes are kayak, noon, and racecar.

1.7 **EXAMPLE** The language of palindromes over $\Sigma = \{a, b\}$ is $\mathcal{L} = \{\sigma \in \Sigma^* \mid \sigma = \sigma^R\}$. A few members are abba, aaabaaa, a, and ϵ .

1.8 **EXAMPLE** Let $\Sigma = \{a, b, c\}$. Recall that a Pythagorean triple of integers has the sum of the squares of the first two equal to the square of the third, as with 3, 4, and 5, or 5, 12, and 13. One way to describe Pythagorean triples is with the language $\mathcal{L} = \{a^i b^j c^k \in \Sigma^* \mid i, j, k \in \mathbb{N} \text{ and } i^2 + j^2 = k^2\}$. Some members are aaabbbbcccccc = $a^3 b^4 c^5$, $a^5 b^{12} c^{13}$, and $a^8 b^{15} c^{17}$.

1.9 **EXAMPLE** The empty set is a language $\mathcal{L} = \{\}$ over any alphabet. So is the set whose single element is the empty string $\hat{\mathcal{L}} = \{\epsilon\}$. These two languages are different, because the first has no members while the second has one.

We can think that the English language consists of sentences. Then Σ is the set of words from the dictionary and a sentence is a string of words, $\sigma \in \Sigma^*$. This explains our use of ‘language’ for a set of strings. On the other hand, our definition allows a language to be any set of strings at all but in English a sentence must be constructed according to rules. The next section studies rules for languages, grammars.

1.10 **DEFINITION** A collection of languages is a **class**.

1.11 **EXAMPLE** For any alphabet, the collection of all finite languages over that alphabet is a class.

1.12 **EXAMPLE** Let \mathcal{P}_e be a Turing machine using the input alphabet $\Sigma = \{B, 1\}$. The set of strings $W_e = \{\sigma \in \Sigma^* \mid \mathcal{P}_e \text{ halts on input } \sigma\}$ is a language. The collection of all such languages, of the W_e for all $e \in \mathbb{N}$, is the class of computably enumerable languages over Σ .

These are the natural operations on languages.

1.13 **DEFINITION (OPERATIONS ON LANGUAGES)** The **concatenation of languages**, $\mathcal{L}_0 \mathcal{L}_1$ or $\mathcal{L}_0 \mathcal{L}_1$, is the language of concatenations, $\{\sigma_0 \mathcal{L}_1 \mid \sigma_0 \in \mathcal{L}_0 \text{ and } \sigma_1 \in \mathcal{L}_1\}$.

For any language \mathcal{L} , the **power \mathcal{L}^k** is the language consisting of the concatenation of k -many members, $\mathcal{L}^k = \{\sigma_0 \mathcal{L} \cdots \mathcal{L} \sigma_{k-1} \mid \sigma_i \in \mathcal{L}\}$ when $k > 0$. We take $\mathcal{L}^0 = \{\epsilon\}$.[†] The **Kleene star of a language \mathcal{L}^*** is the language consisting of the concatenation of any number of strings.

$$\mathcal{L}^* = \{\sigma_0 \mathcal{L} \cdots \mathcal{L} \sigma_{k-1} \mid k \in \mathbb{N} \text{ and } \sigma_0, \dots, \sigma_{k-1} \in \mathcal{L}\} = \mathcal{L}^0 \cup \mathcal{L}^1 \cup \mathcal{L}^2 \cup \dots$$

This includes the concatenation of 0-many strings, so $\epsilon \in \mathcal{L}^*$ even if $\mathcal{L} = \emptyset$.

The **reversal** of a language is the language of reversals, $\mathcal{L}^R = \{\sigma^R \mid \sigma \in \mathcal{L}\}$.

[†]We take $\mathcal{L}^0 = \{\epsilon\}$ even when $\mathcal{L} = \emptyset$. One reason is that for all strings, $\sigma^0 = \epsilon$, since ϵ is the identity element for string concatenation. We saw the same reasoning on page 21, which defines the sum of zero-many numbers to be 0 and the product of zero-many numbers to be 1,

- 1.14 **REMARK** Languages are sets so the operations of union, intersection, etc., apply. However, the union of a language over $\Sigma_0 = \{a\}$ with a language over $\Sigma_1 = \{b\}$ is an awkward mixing, being a combination of strings of a's with strings of b's. That is, the union of a language over Σ_0 with a language over Σ_1 is a language over $\Sigma_0 \cup \Sigma_1$ and the two parts don't naturally mix. Intersection is similar.
- 1.15 **EXAMPLE** Where the language is the set of bitstrings $\mathcal{L} = \{1000001, 1100001\}$ then the reversal is $\mathcal{L}^R = \{1000001, 10000011\}$.
- 1.16 **EXAMPLE** If the language \mathcal{L} consists of two strings $\{a, bc\}$ then the second power of that language is $\mathcal{L}^2 = \{aa, abc, bca, bcba\}$. Its Kleene star is this.

$$\mathcal{L}^* = \{\varepsilon, a, bc, aa, abc, bca, bcba, \dots\}$$

Earlier, for an alphabet Σ we defined Σ^* to be the set of strings over that alphabet. That definition and the one above agree if we don't distinguish between a character and a length-one string.

Also, we have two choices for the above definition of the operation of repeatedly choosing strings. We could choose a string σ once and then repeat it, getting the set of all σ^k . Or we could repeat choosing strings, getting the set of all $\sigma_0^\wedge \sigma_1^\wedge \dots \wedge \sigma_k$. The second is more useful and that's the definition of \mathcal{L}^k and \mathcal{L}^* .

We finish by describing two ways that a machine can relate to a language. We previously discussed these in terms of sets but languages are sets so the ideas apply here also. We have already defined that a machine decides a language if it computes whether or not a given input is a member of that language.

For the other, suppose that the language is computably enumerable but not computable. Then there is a machine that determines whether a given input is a member of the language but it is not able to determine whether the input is not in the language. For instance, there is a Turing machine that, given input e , can determine whether $e \in K$ but no machine can determine whether $e \notin K$.

We will say that a machine **recognizes** (or **accepts**, or **semidecides**) a language when, given an input, the machine computes in a finite time whether the input is in the language, and if it is not in the language then the machine will never incorrectly report that it is. (The machine may determine that it is not, or it may simply fail to report a conclusion such as by failing to halt.) In short, deciding means that on any input the machine correctly computes both 'yes' and 'no' answers, while recognizing requires only that it correctly computes 'yes' answers.

III.1 Exercises

- 1.17 List five of the shortest strings in each language, if there are five.
- (A) $\{\sigma \in \mathbb{B}^* \mid \text{the number of 0's plus the number of 1's equals 3}\}$
 - (B) $\{\sigma \in \mathbb{B}^* \mid \sigma\text{'s first and last characters are equal}\}$
- ✓ 1.18 Is the set of decimal representations of real numbers a language?
- 1.19 Which of these is a palindrome: ()() or)()()? (A) Only the first (B) Only the second (C) Both (D) Neither

- ✓ 1.20 Show that if β is a string then $\beta^\sim \beta^R$ is a palindrome. Do all palindromes have that form?
- ✓ 1.21 Let $\mathcal{L}_0 = \{\varepsilon, a, aa, aaa\}$ and $\mathcal{L}_1 = \{\varepsilon, b, bb, bbb\}$. (A) List all the members of $\mathcal{L}_0^\sim \mathcal{L}_1$. (B) List all the members of $\mathcal{L}_1^\sim \mathcal{L}_0$. (C) List all the members of \mathcal{L}_0^2 . (D) List ten members, if there are ten, of \mathcal{L}_0^* .
- ✓ 1.22 List five members of each language, if there are five, and if not list them all. (A) $\{\sigma \in \{a, b\}^* \mid \sigma = a^n b \text{ for } n \in \mathbb{N}\}$ (B) $\{\sigma \in \{a, b\}^* \mid \sigma = a^n b^n \text{ for } n \in \mathbb{N}\}$ (C) $\{1^n 0^{n+1} \in \mathbb{B}^* \mid n \in \mathbb{N}\}$ (D) $\{1^n 0^{2n} 1 \in \mathbb{B}^* \mid n \in \mathbb{N}\}$
- ✓ 1.23 Where $\mathcal{L} = \{a, ab\}$, list each. (A) \mathcal{L}^2 (B) \mathcal{L}^3 (C) \mathcal{L}^1 (D) \mathcal{L}^0
- 1.24 Where $\mathcal{L}_0 = \{a, ab\}$ and $\mathcal{L}_1 = \{b, bb\}$ find each. (A) $\mathcal{L}_0^\sim \mathcal{L}_1$ (B) $\mathcal{L}_1^\sim \mathcal{L}_0$ (C) \mathcal{L}_0^2 (D) \mathcal{L}_1^2 (E) $\mathcal{L}_0^2 \cap \mathcal{L}_1^2$
- 1.25 Suppose that the language \mathcal{L}_0 has three elements and \mathcal{L}_1 has two. Knowing only that information, for each of these, what is the least number of elements possible and what is the greatest number possible? (A) $\mathcal{L}_0 \cup \mathcal{L}_1$ (B) $\mathcal{L}_0 \cap \mathcal{L}_1$ (C) $\mathcal{L}_0^\sim \mathcal{L}_1$ (D) \mathcal{L}_1^2 (E) \mathcal{L}_1^R (F) $\mathcal{L}_0^* \cap \mathcal{L}_1^*$
- 1.26 What is the language that is the Kleene star of the empty set, \emptyset^* ?
- ✓ 1.27 Is the k -th power of a language the same as the language of k -th powers?
- 1.28 Does \mathcal{L}^* differ from $(\mathcal{L} \cup \{\varepsilon\})^*$?
- 1.29 We can ask how many elements are in the set \mathcal{L}^2 .
 (A) Prove that if two strings are unequal then their squares are also unequal.
 Conclude that if \mathcal{L} has k -many elements then \mathcal{L}^2 has at least k -many elements.
 (B) Provide an example of a nonempty language that achieves this lower bound.
 (C) Prove that where \mathcal{L} has k -many elements, \mathcal{L}^2 has at most k^2 -many.
 (D) Provide an example, for each $k \in \mathbb{N}$, of a language that achieves this upper bound.
- ✓ 1.30 Prove that $\mathcal{L}^* = \mathcal{L}^0 \cup \mathcal{L}^1 \cup \mathcal{L}^2 \cup \dots$
- 1.31 Consider the empty language $\mathcal{L}_0 = \emptyset$. For any language \mathcal{L}_1 , describe $\mathcal{L}_1^\sim \mathcal{L}_0$.
- 1.32 We've stated that the union of a language over Σ_0 with a language over Σ_1 is a language over $\Sigma_0 \cup \Sigma_1$. Formulate and justify the similar statement for intersection.
- 1.33 Let the language \mathcal{L} over some Σ be finite, that is, suppose that $|\mathcal{L}| < \infty$.
 (A) If the language is finite must the alphabet be finite?
 (B) Show that there is some bound $B \in \mathbb{N}$ where $|\sigma| \leq B$ for all $\sigma \in \mathcal{L}$.
 (C) Show that the class of finite languages is closed under finite union. That is, show that if $\mathcal{L}_0, \dots, \mathcal{L}_k$ are finite languages over a shared alphabet for some $k \in \mathbb{N}$ then their union is also finite.
 (D) Show also that the class of finite languages is closed under finite intersection and finite concatenation.
 (E) Show that the class of finite languages is not closed under complementation or Kleene star.

1.34 What is the difference between the languages $\mathcal{L} = \{\sigma \in \Sigma^* \mid \sigma = \sigma^R\}$ and $\hat{\mathcal{L}} = \{\sigma \cap \sigma^R \mid \sigma \in \Sigma^*\}$?

1.35 For any language $\mathcal{L} \subseteq \Sigma^*$ we can form the set of prefixes.

$$\text{Pref}(\mathcal{L}) = \{\tau \in \Sigma^* \mid \sigma \in \mathcal{L} \text{ and } \tau \text{ is a prefix of } \sigma\}$$

Where $\Sigma = \{a, b\}$ and $\mathcal{L} = \{abaaba, bba\}$, find $\text{Pref}(\mathcal{L})$.

1.36 This explains why we define $\mathcal{L}^0 = \{\epsilon\}$ even when $\mathcal{L} = \emptyset$.

(A) Show that $\mathcal{L}^m \cap \mathcal{L}^n = \mathcal{L}^{m+n}$ for any $m, n \in \mathbb{N}^+$.

(B) Show that if $\mathcal{L}_0 = \emptyset$ then $\mathcal{L}_0 \cap \mathcal{L}_1 = \mathcal{L}_1 \cap \mathcal{L}_0 = \emptyset$.

(C) Argue that if $\mathcal{L} \neq \emptyset$ then the only sensible definition for \mathcal{L}^0 is $\{\epsilon\}$.

(D) Why would $\mathcal{L} = \emptyset$ throw a monkey wrench if the works unless we define $\mathcal{L}^0 = \{\epsilon\}$?

1.37 Prove these for any alphabet Σ .

(A) For any natural number n the language Σ^n is countable.

(B) The language Σ^* is countable.

1.38 Another way of defining the powers of a language is: $\mathcal{L}^0 = \{\epsilon\}$, and $\mathcal{L}^{k+1} = \mathcal{L}^k \cap \mathcal{L}$. Show this is equivalent to the one given in Definition 1.13.

1.39 True or false: if $\mathcal{L} \cap \mathcal{L} = \mathcal{L}$ then either $\mathcal{L} = \emptyset$ or $\epsilon \in \mathcal{L}$? If it is true then prove it and if it is false give a counterexample.

1.40 Prove that no language contains a representation for each real number.

1.41 The operations of languages form an algebraic system. Assume these languages are over the same alphabet. Show each.

(A) Language union and intersection are commutative, $\mathcal{L}_0 \cup \mathcal{L}_1 = \mathcal{L}_1 \cup \mathcal{L}_0$ and $\mathcal{L}_0 \cap \mathcal{L}_1 = \mathcal{L}_1 \cap \mathcal{L}_0$.

(B) The language consisting of the empty string is the identity element with respect to language concatenation, so $\mathcal{L} \cap \{\epsilon\} = \mathcal{L}$ and $\{\epsilon\} \cap \mathcal{L} = \mathcal{L}$.

(C) Language concatenation need not be commutative; there are languages such that $\mathcal{L}_0 \cap \mathcal{L}_1 \neq \mathcal{L}_1 \cap \mathcal{L}_0$.

(D) Language concatenation is associative, $(\mathcal{L}_0 \cap \mathcal{L}_1) \cap \mathcal{L}_2 = \mathcal{L}_0 \cap (\mathcal{L}_1 \cap \mathcal{L}_2)$.

(E) $(\mathcal{L}_0 \cap \mathcal{L}_1)^R = \mathcal{L}_1^R \cap \mathcal{L}_0^R$.

(F) Concatenation is left distributive over union, $(\mathcal{L}_0 \cup \mathcal{L}_1) \cap \mathcal{L}_2 = (\mathcal{L}_0 \cap \mathcal{L}_2) \cup (\mathcal{L}_1 \cap \mathcal{L}_2)$, and also right distributive.

(G) The empty language is an annihilator for concatenation, $\emptyset \cap \mathcal{L} = \mathcal{L} \cap \emptyset = \emptyset$.

(H) The Kleene star operation is idempotent, $(\mathcal{L}^*)^* = \mathcal{L}^*$.

SECTION

III.2 Grammars

We have defined that a language is a set of strings. But this allows for any willy-nilly set. In practice a language is usually given by rules.

Here is an example. Native English speakers will say that the noun phrase “the big red barn” sounds fine but that “the red big barn” sounds wrong. That is, sentences in natural languages are constructed in patterns and the second of those does not follow the English pattern. Artificial languages such as programming languages also have syntax rules, usually very strict rules.

A grammar a set of rules for the formation of strings in a language, that is, it is an analysis of the structure of its language. In an aphorism, grammars are the language of languages.

Definition Before the formal definition we'll first see an example.

- 2.1 EXAMPLE This is a subset of the rules for English: (1) a sentence can be made from a noun phrase followed by a verb phrase, (2) a noun phrase can be made from an article followed by a noun, (3) a noun phrase can also be made from an article then an adjective then a noun, (4) a verb phrase can be made with a verb followed by a noun phrase, (5) one article is ‘the’, (6) one adjective is ‘young’, (7) one verb is ‘caught’, (8) two nouns are ‘man’ and ‘ball’.

This is a convenient notation for the rules just listed.

```

⟨sentence⟩ → ⟨noun phrase⟩ ⟨verb phrase⟩
⟨noun phrase⟩ → ⟨article⟩ ⟨noun⟩
⟨noun phrase⟩ → ⟨article⟩ ⟨adjective⟩ ⟨noun⟩
⟨verb phrase⟩ → ⟨verb⟩ ⟨noun phrase⟩
⟨article⟩ → the
⟨adjective⟩ → young
⟨verb⟩ → caught
⟨noun⟩ → man | ball

```

Each line is a **production** or **rewrite rule**. Each has one arrow, \rightarrow .[†] To the left of each arrow is a **head** and to the right is a **body** or **expansion**. Sometimes two rules have the same head, as with $\langle \text{noun phrase} \rangle$. There are also two rules for $\langle \text{noun} \rangle$ but we have abbreviated by combining the bodies using the ‘|’ pipe symbol.[‡]

The rules use two different components. The ones written in typewriter type, such as *young*, are from the alphabet Σ of the language. These are **terminals**.[#] The ones with angle brackets and italics, such as $\langle \text{article} \rangle$, are **nonterminals**. These are like variables and are used for intermediate steps.

The two symbols ‘ \rightarrow ’ and ‘|’ are neither terminals nor nonterminals. They are **metacharacters**, part of the syntax of the rules themselves.

The rewrite rules govern the **derivation** of strings in the language. Under the grammar for English every derivation starts with $\langle \text{sentence} \rangle$. During a derivation, intermediate strings contain a mix of nonterminals and terminals. In our grammars

[†] Read the arrow aloud as “may produce,” or “may expand to,” or “may be constructed as.” [‡] Read the vertical bar aloud as “or.” [#] So the alphabet Σ is not the set of twenty six letters, it is the dictionary of allowed English words.

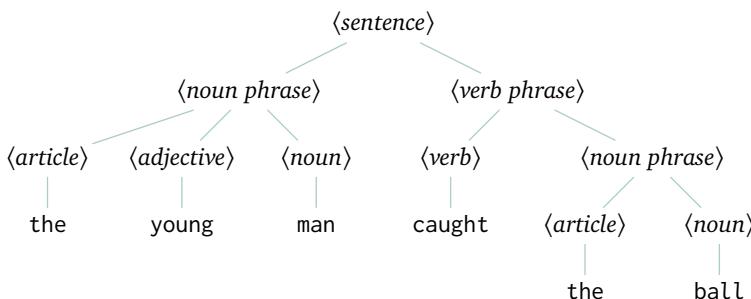
every rule has a head with a single nonterminal. So to get the next string pick a nonterminal in the present string and substitute a matching body.

$$\begin{aligned}
 \langle \text{sentence} \rangle &\Rightarrow \langle \text{noun phrase} \rangle \langle \text{verb phrase} \rangle \\
 &\Rightarrow \langle \text{article} \rangle \langle \text{adjective} \rangle \langle \text{noun} \rangle \langle \text{verb phrase} \rangle \\
 &\Rightarrow \text{the } \langle \text{adjective} \rangle \langle \text{noun} \rangle \langle \text{verb phrase} \rangle \\
 &\Rightarrow \text{the young } \langle \text{noun} \rangle \langle \text{verb phrase} \rangle \\
 &\Rightarrow \text{the young man } \langle \text{verb phrase} \rangle \\
 &\Rightarrow \text{the young man } \langle \text{verb} \rangle \langle \text{noun phrase} \rangle \\
 &\Rightarrow \text{the young man caught } \langle \text{noun phrase} \rangle \\
 &\Rightarrow \text{the young man caught } \langle \text{article} \rangle \langle \text{noun} \rangle \\
 &\Rightarrow \text{the young man caught the } \langle \text{noun} \rangle \\
 &\Rightarrow \text{the young man caught the ball}
 \end{aligned}$$

Note that the single line arrow \rightarrow is for rules while the double line arrow \Rightarrow is for derivations.[†]

The derivation above always substitutes for the leftmost nonterminal, so it is a **leftmost derivation**. However, in general we could substitute for any nonterminal.

The **derivation tree** or **parse tree** is an alternative representation.[‡]



- 2.2 **DEFINITION** A **context-free grammar** is a four-tuple $\mathcal{G} = \langle \Sigma, N, S, P \rangle$. The set Σ is an alphabet, whose elements are the **terminal symbols**, and the elements of the set N are the **nonterminals** or **syntactic categories**. (We assume that Σ and N are disjoint and that neither contains metacharacters.) The symbol $S \in N$ is the **start symbol**. Finally, P is a set of **productions** or **rewrite rules**.

Our convention is to indicate the start symbol by making it the head of the first rule.

- 2.3 **EXAMPLE** This context free grammar describes algebraic expressions that involve only addition, multiplication, and parentheses.

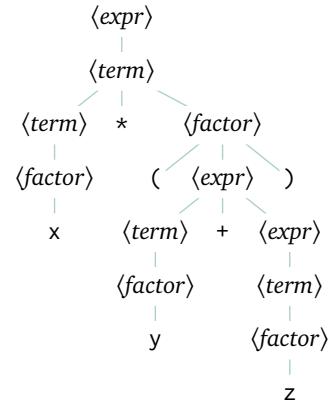
$$\begin{aligned}
 \langle \text{expr} \rangle &\rightarrow \langle \text{term} \rangle + \langle \text{expr} \rangle \mid \langle \text{term} \rangle \\
 \langle \text{term} \rangle &\rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{factor} \rangle
 \end{aligned}$$

[†]Read ' \Rightarrow ' aloud as "derives" or "expands to." [‡]The words 'terminal' and 'nonterminal' come from where the components lie in this tree.

$$\langle \text{factor} \rangle \rightarrow (\langle \text{expr} \rangle) \mid a \mid b \mid \dots \mid z$$

Here is a derivation of the string $x*(y+z)$.

$$\begin{aligned}\langle \text{expr} \rangle &\Rightarrow \langle \text{term} \rangle \\&\Rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \\&\Rightarrow \langle \text{factor} \rangle * \langle \text{factor} \rangle \\&\Rightarrow x * \langle \text{factor} \rangle \\&\Rightarrow x * (\langle \text{expr} \rangle) \\&\Rightarrow x * (\langle \text{term} \rangle + \langle \text{expr} \rangle) \\&\Rightarrow x * (\langle \text{term} \rangle + \langle \text{term} \rangle) \\&\Rightarrow x * (\langle \text{factor} \rangle + \langle \text{term} \rangle) \\&\Rightarrow x * (\langle \text{factor} \rangle + \langle \text{factor} \rangle) \\&\Rightarrow x * (y + \langle \text{factor} \rangle) \\&\Rightarrow x * (y + z)\end{aligned}$$



In that example the rules for $\langle \text{expr} \rangle$ and $\langle \text{term} \rangle$ are recursive. But we don't get stuck in an infinite regress because the question is not whether you could perversely keep expanding $\langle \text{expr} \rangle$ forever; the question is whether, given a string such as $x*(y+z)$, you can find a terminating derivation.

In the prior example the nonterminals such as $\langle \text{expr} \rangle$ or $\langle \text{term} \rangle$ describe the role of those components in the language, as did the English grammar fragment's $\langle \text{noun phrase} \rangle$ and $\langle \text{article} \rangle$. That is why nonterminals are sometimes called 'syntactic categories'. But our examples often use small grammars whose terminals and nonterminals do not have any particular meaning. For these we often instead write productions using single letters, with nonterminals in upper case and terminals in lower case.

2.4 EXAMPLE This two-rule grammar has one nonterminal, S.

$$S \rightarrow aSb \mid \epsilon$$

Here is a derivation of the string a^2b^2 .

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\epsilon bb \Rightarrow aabb$$

Similarly, $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaa\epsilon bbb = aaabbb$ is a derivation of a^3b^3 . For this grammar, derivable strings have the form $a^n b^n$ for $n \in \mathbb{N}$.

We next give a complete description of how the production rules govern the derivations. Each rule in a context free grammar has the form 'head \rightarrow body' where the head consists of a single nonterminal. The body is a sequence of terminals and nonterminals. Each step of a derivation has the form below, where τ_0 and τ_1 are sequences of terminals and non-terminals.

$$\tau_0 \stackrel{\text{head}}{\sim} \tau_1 \Rightarrow \tau_0 \stackrel{\text{body}}{\sim} \tau_1$$

That is, if there is a match for the rule's head then we can replace it with the body.

Where σ_0, σ_1 are sequences of terminals and nonterminals, if they are related by a sequence of derivation steps then we may write $\sigma_0 \Rightarrow^* \sigma_1$. Where $\sigma_0 = S$ is the start symbol, if there is a sequence $\sigma_0 \Rightarrow^* \sigma_1$ that finishes with a string of terminals $\sigma_1 \in \Sigma^*$ then we say that σ_1 has a **derivation** from the grammar.[†]

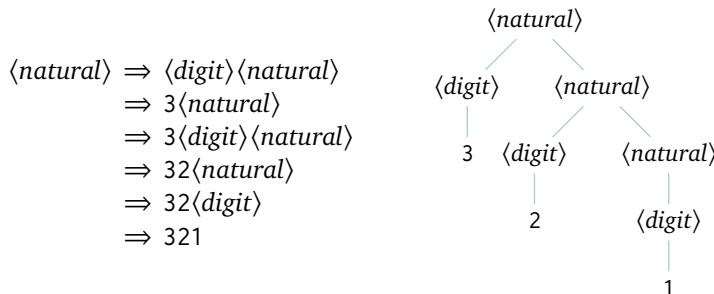
This description is like the one on page 8 detailing how a Turing machine's instructions determine the evolution of the sequence of configurations that is a computation. That is, production rules are like a program, directing a derivation. However, one difference from that page's description is that there Turing machines are deterministic, so that from a given input string there is a determined sequence of configurations. Here, from a given start symbol a derivation can branch out to go to many different ending strings.

- 2.5 **DEFINITION** The **language derived from a grammar** is the set of strings of terminals having derivations that begin with the start symbol.

- 2.6 **EXAMPLE** This grammar's language is the set of representations of natural numbers.

$$\begin{aligned}\langle \text{natural} \rangle &\rightarrow \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{natural} \rangle \\ \langle \text{digit} \rangle &\rightarrow 0 \mid \dots \mid 9\end{aligned}$$

This is a derivation for the string 321, along with its parse tree.



- 2.7 **EXAMPLE** The language of this grammar

$$\langle \text{natural} \rangle \rightarrow \epsilon \mid 1 \langle \text{natural} \rangle$$

is the set of strings representing natural numbers in unary.

- 2.8 **EXAMPLE** Any finite language is derived from a grammar. This one gives the language of all length 2 bitstrings, using the brute force approach of just listing all the member strings.

[†]This definition of rules, grammars, and derivations suffices for us but it is not the most general one. One more general definition allows heads of the form $\sigma_0 X \sigma_1$, where σ_0 and σ_1 are strings of terminals. (The σ_i 's can be empty.) For example, consider this grammar: (i) $S \rightarrow aBSc \mid abc$, (ii) $Ba \rightarrow aB$, (iii) $Bb \rightarrow bb$. Rule (ii) says that if you see a string with something followed by a then you can replace that string with a followed by that thing. Grammars with heads of the form $\sigma_0 X \sigma_1$ are **context sensitive** because we can only substitute for X in the context of σ_0 and σ_1 . Context sensitive grammars describe more languages than the context free ones that we are using, and there are grammar classes even more general. But our definition satisfies our needs and is the class of grammars that you most often see in practice.

$$S \rightarrow 00 \mid 01 \mid 10 \mid 11$$

This gives the length 3 bitstrings by using the nonterminals to keep count.

$$A \rightarrow 0B \mid 1B$$

$$B \rightarrow 0C \mid 1C$$

$$C \rightarrow 0 \mid 1$$

A derivation of 101 is $A \Rightarrow 1B \Rightarrow 10C \Rightarrow 101$.

2.9 EXAMPLE For this grammar

$$S \rightarrow aSb \mid T \mid U$$

$$T \rightarrow aS \mid a$$

$$U \rightarrow Sb \mid b$$

an alternative is to replace T and U by their expansions to get this.

$$S \rightarrow aSb \mid aS \mid a \mid Sb \mid b$$

It generates the language $\mathcal{L} = \{a^i b^j \in \{a, b\}^* \mid i \neq 0 \text{ or } j \neq 0\}$.

This grammar is the first that we have seen where the generated language is not clear, so we will do a formal verification. We will show mutual containment, first that the generated language is a subset of \mathcal{L} and then that it is also a superset.

The rules in the second grammar show that any derivation step $\tau_0 \cap \text{head} \cap \tau_1 \Rightarrow \tau_0 \cap \text{body} \cap \tau_1$ only adds a's on the left and b's on the right, so every string in the language has the form $a^i b^j$. That same rules show that in any terminating derivation S must eventually be replaced by either a or b. Together these two give that the generated language is a subset of \mathcal{L} .

For containment the other way, we will prove that every $\sigma \in \mathcal{L}$ has a derivation. We will use induction on the length $|\sigma|$. By the definition of \mathcal{L} the base case is $|\sigma| = 1$. In this case either $\sigma = a$ or $\sigma = b$, each of which obviously has a derivation.

For the inductive step, fix $n \geq 1$ where every string from \mathcal{L} of length $k = 1, \dots, k = n$ has a derivation, and let σ have length $n + 1$. By the definition of \mathcal{L} it has the form $\sigma = a^i b^j$. There are three cases: either $i = j = 1$, or $i > 1$, or $j > 1$. The $\sigma = a^1 b^1$ case is easy. For the $i > 1$ case, $\hat{\sigma} = a^{i-1} b^j$ is a string of length n , so by the inductive hypothesis it has a derivation $S \Rightarrow \dots \Rightarrow \hat{\sigma}$. Prefixing that derivation with a $S \Rightarrow aS$ step will put an additional a on the left. The $j > 1$ case works the same way.

2.10 EXAMPLE The fact that derivations can go more than one way leads to an important issue with grammars, that they can be ambiguous. Consider this fragment of a grammar for if statements in a C-like language

$$\langle \text{stmt} \rangle \rightarrow \text{if } \langle \text{bool} \rangle \langle \text{stmt} \rangle$$

$$\langle \text{stmt} \rangle \rightarrow \text{if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$$

and this code string.

```
if enrolled(s) if studied(s) grade='P' else grade='F'
```

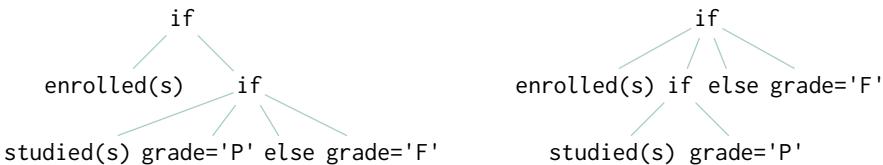
Here are the first two lines of one derivation

$$\begin{aligned}\langle \text{stmt} \rangle &\Rightarrow \text{if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \\ &\Rightarrow \text{if } \langle \text{bool} \rangle \text{ if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle\end{aligned}$$

and here are the first two of another.

$$\begin{aligned}\langle \text{stmt} \rangle &\Rightarrow \text{if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle \\ &\Rightarrow \text{if } \langle \text{bool} \rangle \text{ if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle\end{aligned}$$

That is, we cannot tell whether the `else` in the code line is associated with the first `if` or the second. The resulting parse trees for the full code line dramatize the difference



as do these copies of the code string indented to dramatize the association.

```
if enrolled(s)
  if studied(s)
    grade='P'
  else
    grade='F'
```

```
if enrolled(s)
  if studied(s)
    grade='P'
else
  grade='F'
```

Obviously, those programs behave differently. This is known as a **dangling else**. (In a language such as C a programmer uses curly braces to make clear which of the two possibilities is the intended one.)

2.11 EXAMPLE This grammar for elementary algebra expressions

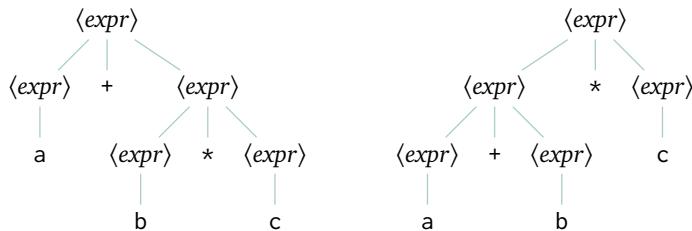
$$\begin{aligned}\langle \text{expr} \rangle &\rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle \\ &\quad | \langle \text{expr} \rangle * \langle \text{expr} \rangle \\ &\quad | (\langle \text{expr} \rangle) \quad | \text{a} \mid \text{b} \mid \dots \text{z}\end{aligned}$$

is ambiguous because `a+b*c` has two leftmost derivations.

$$\begin{aligned}\langle \text{expr} \rangle &\Rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle \Rightarrow \text{a} + \langle \text{expr} \rangle \\ &\Rightarrow \text{a} + \langle \text{expr} \rangle * \langle \text{expr} \rangle \Rightarrow \text{a} + \text{b} * \langle \text{expr} \rangle \Rightarrow \text{a} + \text{b} * \text{c} \\ \langle \text{expr} \rangle &\Rightarrow \langle \text{expr} \rangle * \langle \text{expr} \rangle \Rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle * \langle \text{expr} \rangle \\ &\Rightarrow \text{a} + \langle \text{expr} \rangle * \langle \text{expr} \rangle \Rightarrow \text{a} + \text{b} * \langle \text{expr} \rangle \Rightarrow \text{a} + \text{b} * \text{c}\end{aligned}$$

Again, the issue is that we get two different behaviors. For instance, take 1 for `a`, and 2 for `b`, and 3 for `c`. The first derivation gives $1 + (2 \cdot 3) = 7$ while the second one gives $(1 + 2) \cdot 3 = 9$.

That difference is reflected in different parse trees.



In contrast, this grammar for elementary algebra expressions is unambiguous.

```

 $\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle$ 
|  $\langle \text{term} \rangle$ 
 $\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle$ 
|  $\langle \text{factor} \rangle$ 
 $\langle \text{factor} \rangle \rightarrow (\langle \text{expr} \rangle)$ 
| a | b | ... | z
  
```

Choosing grammars that are unambiguous is important in practice.

III.2 Exercises

- ✓ 2.12 Use the grammar of Example 2.3. (A) What is the start symbol? (B) What are the terminals? (C) What are the nonterminals? (D) How many rewrite rules does it have? (E) Give three strings derived from the grammar, besides the string in the example. (F) Give three strings in the language $\{+, *\}, (\, , a \dots, z\}^*$ that cannot be derived.

- 2.13 Use the grammar of Exercise 2.15. (A) What is the start symbol? (B) What are the terminals? (C) What are the nonterminals? (D) How many rewrite rules does it have? (E) Give three strings derived from the grammar besides the ones in the exercise, or show that there are not three such strings. (F) Give three strings in the language $\mathcal{L} = \{ \sigma \in \Sigma \cup \{\text{space}\}^* \mid \Sigma \text{ is the set of terminals} \}$ that cannot be derived from this grammar, or show there are not three such strings.

- 2.14 Use this grammar.

```

 $\langle \text{natural} \rangle \rightarrow \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{natural} \rangle$ 
 $\langle \text{digit} \rangle \rightarrow 0 \mid 1 \mid \dots \mid 9$ 
  
```

- (A) What is the alphabet? What are the terminals? The nonterminals? What is the start symbol? (B) For each production, name the head and the body. (C) Which are the metacharacters that are used? (D) Derive 42. Also give the associated parse tree. (E) Derive 993 and give its parse tree. (F) How can $\langle \text{natural} \rangle$ be defined in terms of $\langle \text{natural} \rangle$? Doesn't that lead to infinite regress? (G) Extend this grammar to cover the integers. (H) With your grammar, can you derive $+0$? -0 ?

- ✓ 2.15 From this grammar

$\langle \text{sentence} \rangle \rightarrow \langle \text{subject} \rangle \langle \text{predicate} \rangle$
 $\langle \text{subject} \rangle \rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle$
 $\langle \text{predicate} \rangle \rightarrow \langle \text{verb} \rangle \langle \text{direct object} \rangle$
 $\langle \text{direct object} \rangle \rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle$
 $\langle \text{article} \rangle \rightarrow \text{the} \mid \text{a}$
 $\langle \text{noun} \rangle \rightarrow \text{car} \mid \text{wall}$
 $\langle \text{verb} \rangle \rightarrow \text{hit}$

derive each of these: (A) the car hit a wall (B) the car hit the wall
 (C) the wall hit a car.

2.16 Consider this grammar.

$\langle \text{sentence} \rangle \rightarrow \langle \text{subject} \rangle \langle \text{predicate} \rangle$
 $\langle \text{subject} \rangle \rightarrow \langle \text{article} \rangle \langle \text{noun}_1 \rangle$
 $\langle \text{predicate} \rangle \rightarrow \langle \text{verb} \rangle \langle \text{direct-object} \rangle$
 $\langle \text{direct-object} \rangle \rightarrow \langle \text{article} \rangle \langle \text{noun}_2 \rangle$
 $\langle \text{article} \rangle \rightarrow \text{the} \mid \text{a} \mid \epsilon$
 $\langle \text{noun}_1 \rangle \rightarrow \text{dog} \mid \text{flea}$
 $\langle \text{noun}_2 \rangle \rightarrow \text{man} \mid \text{dog}$
 $\langle \text{verb} \rangle \rightarrow \text{bites} \mid \text{licks}$

- (A) Give a derivation for dog bites man.
 (B) Show that there is no derivation for man bites dog.

✓ 2.17 Your friend tries the prior exercise and you see their work so far.

$\langle \text{sentence} \rangle \Rightarrow \langle \text{subject} \rangle \langle \text{predicate} \rangle$
 $\Rightarrow \langle \text{article} \rangle \langle \text{noun}_1 \rangle \langle \text{predicate} \rangle$
 $\Rightarrow \langle \text{article} \rangle \langle \text{noun}_1 \rangle \langle \text{verb} \rangle \langle \text{direct object} \rangle$
 $\Rightarrow \langle \text{article} \rangle \langle \text{dog} \mid \text{flea} \rangle \langle \text{verb} \rangle \langle \text{article} \rangle \langle \text{noun}_2 \rangle$
 $\Rightarrow \langle \text{article} \rangle \langle \text{dog} \mid \text{flea} \rangle \langle \text{verb} \rangle \langle \text{article} \rangle \langle \text{man} \mid \text{dog} \rangle$

Stop them and explain what they are doing wrong.

2.18 With the grammar of Example 2.3, derive $(a+b)^*c$.

✓ 2.19 Use this grammar

$S \rightarrow TbU$
 $T \rightarrow aT \mid \epsilon$
 $U \rightarrow aU \mid bU \mid \epsilon$

for each part. (A) Give both a leftmost derivation and rightmost derivation of aabab. (B) Do the same for baab. (C) Show that there is no derivation of aa.

2.20 Use this grammar.

$S \rightarrow aABb$

$$A \rightarrow aA \mid a$$

$$B \rightarrow Bb \mid b$$

(A) Derive three strings.

(B) Name three strings over $\Sigma = \{a, b\}$ that are not derivable.

(C) Describe the language generated by this grammar.

2.21 Give a grammar for the language $\{a^n b^{n+m} a^m \mid n, m \in \mathbb{N}\}$.

✓ 2.22 Give the parse tree for the derivation of aabb in Example 2.4.

2.23 Verify that the language derived from the grammar in Example 2.4 is $\mathcal{L} = \{a^n b^n \mid n \in \mathbb{N}\}$.

2.24 What is the language generated by this grammar?

$$A \rightarrow aA \mid B$$

$$B \rightarrow bB \mid cA$$

✓ 2.25 In many programming languages identifier names consist of a string of letters or digits, with the restriction that the first character must be a letter. Create a grammar for this, using ASCII letters.

2.26 Early programming languages had strong restrictions on what could be a variable name. Create a grammar for a language that consists of strings of at most four characters, upper case ASCII letters or digits, where the first character must be a letter.

2.27 What is the language generated by a grammar with a set of production rules that is empty?

2.28 Here is a grammar for propositional logic expressions in Conjunctive Normal form.

$$\begin{aligned} \langle \text{CNF} \rangle &\rightarrow (\langle \text{Disjunction} \rangle) \wedge \langle \text{DNF} \rangle \mid (\langle \text{Disjunction} \rangle) \\ \langle \text{Disjunction} \rangle &\rightarrow \langle \text{Literal} \rangle \vee \langle \text{Disjunction} \rangle \mid \langle \text{Literal} \rangle \\ \langle \text{Literal} \rangle &\rightarrow \neg \langle \text{Variable} \rangle \mid \langle \text{Variable} \rangle \\ \langle \text{Variable} \rangle &\rightarrow x_0 \mid x_1 \mid \dots \end{aligned}$$

For more, see Appendix C.

(A) Derive $(x_0 \vee \neg x_1) \wedge (x_1 \vee x_2)$.

(B) Show that you cannot derive $(\neg x_0 \wedge \neg x_1) \vee (\neg x_0 \wedge \neg x_2)$.

2.29 Create a grammar for each of these languages.

(A) the language of all character strings $\mathcal{L} = \{a, \dots, z\}^*$

(B) the language of strings of at least one digit $\{\sigma \in \{0, \dots, 9\}^* \mid |\sigma| \geq 1\}$

✓ 2.30 This is a grammar for postal addresses. Note the use of the empty string ϵ to make some components optional, such as $\langle \text{opt suffix} \rangle$ and $\langle \text{apt num} \rangle$.

$$\langle \text{postal address} \rangle \rightarrow \langle \text{name} \rangle \langle \text{EOL} \rangle \langle \text{street address} \rangle \langle \text{EOL} \rangle \langle \text{town} \rangle$$

$$\langle \text{name} \rangle \rightarrow \langle \text{personal part} \rangle \langle \text{last name} \rangle \langle \text{opt suffix} \rangle$$

$$\langle \text{street address} \rangle \rightarrow \langle \text{house num} \rangle \langle \text{street name} \rangle \langle \text{apt num} \rangle$$

$\langle \text{town} \rangle \rightarrow \langle \text{town name} \rangle , \langle \text{state or region} \rangle$
 $\langle \text{personal part} \rangle \rightarrow \langle \text{initial} \rangle . \mid \langle \text{first name} \rangle$
 $\langle \text{last name} \rangle \rightarrow \langle \text{char string} \rangle$
 $\langle \text{opt suffix} \rangle \rightarrow \text{Sr.} \mid \text{Jr.} \mid \epsilon$
 $\langle \text{house num} \rangle \rightarrow \langle \text{digit string} \rangle$
 $\langle \text{street name} \rangle \rightarrow \langle \text{char string} \rangle$
 $\langle \text{apt num} \rangle \rightarrow \langle \text{char string} \rangle \mid \epsilon$
 $\langle \text{town name} \rangle \rightarrow \langle \text{char string} \rangle$
 $\langle \text{state or region} \rangle \rightarrow \langle \text{char string} \rangle$
 $\langle \text{initial} \rangle \rightarrow \langle \text{char} \rangle$
 $\langle \text{first name} \rangle \rightarrow \langle \text{char string} \rangle$
 $\langle \text{char string} \rangle \rightarrow \langle \text{char} \rangle \mid \langle \text{char} \rangle \langle \text{char string} \rangle \mid \epsilon$
 $\langle \text{char} \rangle \rightarrow \text{A} \mid \text{B} \mid \dots \text{z} \mid \text{0} \mid \dots \text{9} \mid (\text{space})$
 $\langle \text{digit string} \rangle \rightarrow \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{digit string} \rangle \mid \epsilon$
 $\langle \text{digit} \rangle \rightarrow \text{0} \mid \dots \text{9}$

The nonterminal $\langle \text{EOL} \rangle$ expands to an end of line such as ASCII 10, while (space) signifies a whitespace character such as ASCII 0 or ASCII 32, or even more exotic characters such as en-space or em-space.

(A) Give a derivation for this address.

President
1600 Pennsylvania Avenue
Washington, DC

(B) Why is there no derivation for this address?

Sherlock Holmes
221B Baker Street
London, UK

Suggest a modification of the grammar so that this address is in the language.

(c) Give three reasons why this grammar is inadequate. (Perhaps no grammar would suffice other than just listing every address in the world.)

2.31 Recall Turing's prototype computer, a clerk doing the symbolic manipulations to multiply two large numbers. Deriving a string from a grammar has a similar feel and we can write grammars to do computations. Fix the alphabet $\Sigma = \{1\}$, so that we can interpret derived strings as numbers represented in unary.

(A) Produce a grammar whose language is the even numbers, $\{1^{2n} \mid n \in \mathbb{N}\}$.

(B) Do the same for the multiples of three, $\{1^{3n} \mid n \in \mathbb{N}\}$.

✓ 2.32 Here is a grammar notable for being small.

$\langle \text{sentence} \rangle \rightarrow \text{buffalo } \langle \text{sentence} \rangle \mid \epsilon$

- (A) Derive a sentence of length one, one of length two, and one of length three.
 (B) Give those sentences semantics, that is, make sense of them as English sentences.

2.33 Here is a grammar for LISP.

$$\begin{aligned}
 \langle s\ expression \rangle &\rightarrow \langle atomic\ symbol \rangle \\
 &| (\langle s\ expression \rangle . \langle s\ expression \rangle) \\
 &| \langle list \rangle \\
 \langle list \rangle &\rightarrow (\langle list-entries \rangle) \\
 \langle list-entries \rangle &\rightarrow \langle s\ expression \rangle \\
 &| \langle s\ expression \rangle \langle list-entries \rangle \\
 \langle atomic\ symbol \rangle &\rightarrow \langle letter \rangle \langle atom\ part \rangle \\
 \langle atom\ part \rangle &\rightarrow \epsilon \\
 &| \langle letter \rangle \langle atom\ part \rangle \\
 &| \langle number \rangle \langle atom\ part \rangle \\
 \langle letter \rangle &\rightarrow a | \dots z \\
 \langle number \rangle &\rightarrow 0 | \dots 9
 \end{aligned}$$

Give a derivation for each string. (A) (a . b) (B) (a . (b . c))
 (C) ((a . b) . c)

2.34 Using the Example 2.11's unambiguous grammar, produce a derivation for $a^+(b*c)$.

2.35 The simplest example of an ambiguous grammar is

$$S \rightarrow S \mid \epsilon$$

- (A) What is the language generated by this grammar?
 (B) Produce two different derivations of the empty string.

2.36 This is a grammar for the language of bitstrings $\mathcal{L} = \mathbb{B}^*$.

$$\langle bit-string \rangle \rightarrow 0 \mid 1 \mid \langle bit-string \rangle \langle bit-string \rangle$$

Show that it is ambiguous.

2.37 (A) Show that this grammar is ambiguous by producing two different leftmost derivations for a-b-a.

$$E \rightarrow E - E \mid a \mid b$$

- (B) Derive a-b-a from this grammar, which is unambiguous.

$$E \rightarrow E - T \mid T$$

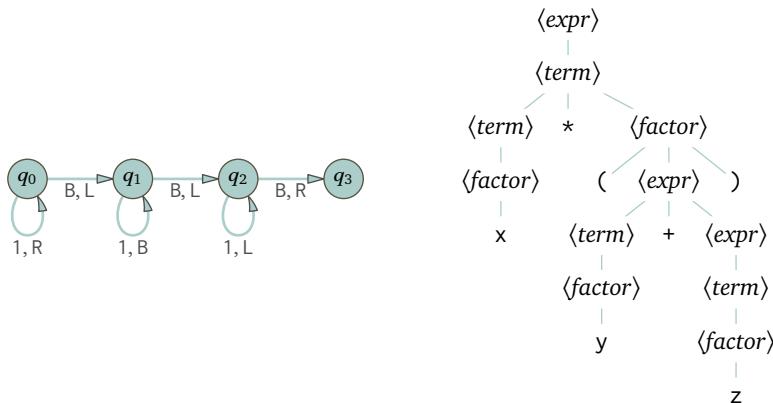
$$T \rightarrow a \mid b$$

2.38 Use the grammar from the footnote on 147 to derive aaabbcc.

SECTION

III.3 Graphs

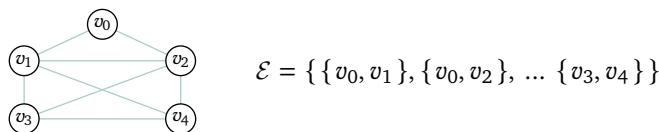
In the Theory of Computation we often state problems, using the language of Graph Theory. Here are two examples we have already seen. Both have vertices, and those vertices are connected by edges that represent a relationship between the vertices.



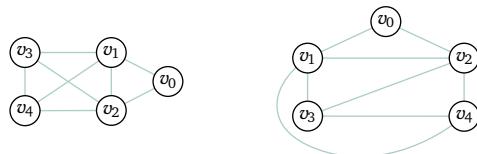
Definition We start with the basics.

- 3.1 **DEFINITION** A **simple graph** is an ordered pair $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ where \mathcal{N} is a finite set of **vertices**[†] or **nodes** and \mathcal{E} is a set of **edges**. Each edge is a set of two distinct vertices; these vertices are **adjacent** or **neighbors**.

- 3.2 **EXAMPLE** This simple graph \mathcal{G} has five vertices $\mathcal{N} = \{v_0, \dots, v_4\}$ and eight edges.



Important: a graph is not its picture. Both of these pictures show the same graph as above because they show the same vertices and the same connections.



Instead of writing $e = \{v, \hat{v}\}$ we often write $e = v\hat{v}$. Since edges are sets and sets are unordered we could write the same edge as $e = \hat{v}v$.

[†]Graphs can have infinitely many vertices but we won't ever need that. For convenience we will stick to finite ones.

There are many variations of that definition, for modeling different circumstances. We could allow some vertices to connect to themselves, forming a **loop**. Another variant is a **multigraph**, which allows two vertices to share more than one edge.

Still another is a **weighted graph**, which gives each edge a real number **weight**, perhaps signifying the distance or the cost in money or time to traverse that edge.

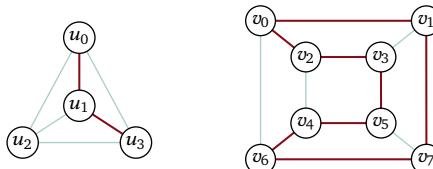
A very common variation is a **directed graph** or **digraph**, where edges have a direction as in a road map that includes one-way streets. If an edge is directed from v to \hat{v} then we can write it as $v\hat{v}$ but not in the other order. The Turing machine graph above is a digraph and also has loops.

Some important variations involve whether the graph has cycles. A cycle is a closed path around the graph; see the complete definition just below. A **tree** is an undirected connected graph with no cycles. A **directed acyclic graph** or **DAG** is a directed graph with no directed cycles.

Paths Many problems involve moving through a graph.

3.3 **DEFINITION** Two graph edges are **adjacent** if they share a vertex, so that they have the form $e_0 = uv$ and $e_1 = vw$. A **walk** is a sequence of adjacent edges $\langle v_0v_1, v_1v_2, \dots, v_{n-1}v_n \rangle$. Its **length** is the number of edges, n . If the initial vertex v_0 equals the final vertex v_n then the walk is **closed**, otherwise it is **open**. A **trail** is a walk where no edge occurs twice. A **circuit** is a closed trail. A **path** is a walk with no repeated edges or vertices, except that it may be closed and so the first and last vertices are equal. A closed path with at least one edge is a **cycle**.[†]

3.4 **EXAMPLE** On the left is a path from u_0 to u_3 ; it is also a trail and a walk. On the right is a cycle.



3.5 **DEFINITION** A vertex v_1 is **reachable** from the vertex v_0 if there is a path from v_0 to v_1 . A graph is **connected** if between any two vertices there is a path.

3.6 **DEFINITION** If a circuit contains all of a graph's edges then it is an **Euler circuit**. If it contains all of the vertices then it is a **Hamiltonian circuit**.

3.7 **EXAMPLE** In Example 3.4 the path in the graph on the left is not a circuit because it is not closed. The path in the graph on the right is a Hamiltonian circuit but it is not an Euler circuit.

[†]These terms are not completely standardized so you may see them used in other ways, especially in older work.

3.8 **DEFINITION** Where $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ is a graph, a **subgraph** $\hat{\mathcal{G}} = \langle \hat{\mathcal{N}}, \hat{\mathcal{E}} \rangle$ satisfies $\hat{\mathcal{N}} \subseteq \mathcal{N}$ and $\hat{\mathcal{E}} \subseteq \mathcal{E}$. A subgraph with every possible edge, that is such that $v_i, v_j \in \hat{\mathcal{N}}$ and $e = v_i v_j \in \mathcal{E}$ implies that $e \in \hat{\mathcal{E}}$ also, is an **induced subgraph**.

3.9 **EXAMPLE** In the graph \mathcal{G} on the left of Example 3.4, consider the highlighted path $\mathcal{E} = \{u_0 u_1, u_1 u_3\}$. Taking those edges along with the vertices that they contain, $\hat{\mathcal{N}} = \{u_0, u_1, u_3\}$, gives a subgraph $\hat{\mathcal{G}}$.

With the same set of vertices, $\{u_0, u_1, u_3\}$, the induced subgraph is the triangle that adds the outer edge, $\mathcal{E} \cup \{u_0 u_3\}$.

We will need the next result in Chapter Five.

3.10 **LEMMA (KÖNIG'S LEMMA)** Suppose that in a connected graph each vertex is adjacent to only finitely many other vertices. If the graph has infinitely many vertices then it has an infinite path.

Proof Fix a vertex v_0 . The graph is connected, so starting at v_0 we can reach every other vertex via some path. For each of v_0 's neighbors v , there is a set of vertices that can be reached from v_0 via a path through v . There are infinitely many vertices so there is a neighbor where this set is infinite. Pick such a neighbor and call it v_1 . Now iterate: by choice of v_1 there are infinitely many vertices reachable by a path starting with v_0 and v_1 , and v_1 has finitely many neighbors, so there is a v_2 adjacent to v_1 through which there are paths to infinitely many vertices, etc. \square

Graph representation We can represent graphs in a computer with reasonable efficiency. A simple and common way is with a matrix. This example represents Example 3.2's graph: it has a 1 in the i, j entry if the graph has an edge from v_i to v_j and a 0 if there is no such edge.

$$\mathcal{M}(\mathcal{G}) = \begin{pmatrix} & v_0 & v_1 & v_2 & v_3 & v_4 \\ v_0 & 0 & 1 & 1 & 0 & 0 \\ v_1 & 1 & 0 & 1 & 1 & 1 \\ v_2 & 1 & 1 & 0 & 1 & 1 \\ v_3 & 0 & 1 & 1 & 0 & 1 \\ v_4 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \quad (*)$$

We can extend this to cover other graph variants that were listed earlier. For instance, the graph represented in $(*)$ is a simple graph because the matrix has only 0 and 1 entries, because all the diagonal entries are 0, and because the matrix is symmetric, meaning that the i, j entry has a 1 if and only if the j, i entry is also 1. If the graph is directed and has a one-way edge from v_i to v_j then the i, j entry records that edge but the j, i entry does not. If the graph is a multigraph, where there can be multiple edges from one vertex to another, the associated entry can be larger than 1. And, if the graph has a loop then the matrix has a diagonal entry that is an integer larger than zero.

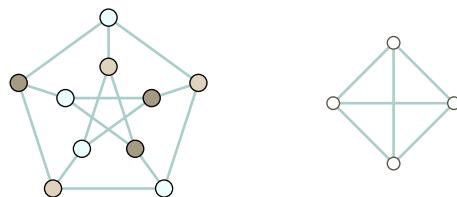
- 3.11 **DEFINITION** For a graph \mathcal{G} , the **adjacency matrix** $M(\mathcal{G})$ representing the graph has i, j entries equal to the number of edges from v_i to v_j .
- 3.12 **LEMMA** Let the matrix $M(\mathcal{G})$ represent the graph \mathcal{G} . Then in its matrix multiplicative n -th power the i, j entry is the number of paths of length n from vertex v_i to vertex v_j .

Proof Exercise 3.39. □

Colors We sometimes partition a graph's vertices.

- 3.13 **DEFINITION** A **k -coloring** of a graph, for $k \in \mathbb{N}$, is a partition of vertices into k -many classes such that no two adjacent vertices are in the same class.

On the left is a graph that is 3-colored.

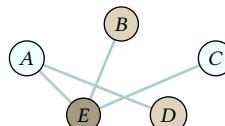


On the right the graph has no 3-coloring. The argument goes: the four vertices are completely connected to each other. If two get the same color then they will be adjacent same-colored vertices. So a coloring requires four colors.

- 3.14 **EXAMPLE** This shows five committees, where some committees share some members. How many time slots do we need in order to schedule all committees so that no member has to be in two meetings at once?

A	B	C	D	E
Armis	Crump	Burke	India	Burke
Jones	Edwards	Frank	Harris	Jones
Smith	Robinson	Ke	Smith	Robinson

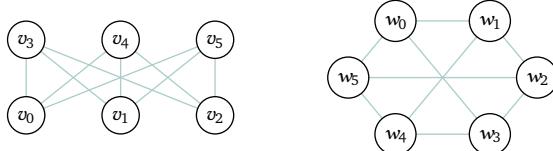
Model this with a graph by taking each vertex to be a committee and if committees are related by sharing a member then put an edge between them.



The picture shows that three colors is enough, that is, three time slots suffice. But there is also a two-coloring, $\mathcal{C}_0 = \{A, B, C\}$ and $\mathcal{C}_1 = \{D, E\}$.

A graph's **chromatic number** is the minimum number k where the graph has a k -coloring.

Graph isomorphism We sometimes want to know when two graphs are essentially identical. Consider these two.



They have the same number of vertices and the same number of edges. Further, on the right as well as on the left there are two classes of vertices where all the vertices in the first class connect to all the vertices in the second class (on the left the two classes are the top and bottom rows while on the right they are $\{w_0, w_2, w_4\}$ and $\{w_1, w_3, w_5\}$). A person may suspect that as in Example 3.2 these are two ways to draw the same graph, with the vertex names changed for further obfuscation.

That's true; if we make a correspondence between the vertices in this way

Vertex on left	v_0	v_1	v_2	v_3	v_4	v_5
Vertex on right	w_0	w_2	w_4	w_1	w_3	w_5

then as a consequence the edges also correspond.

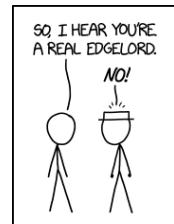
Edge on left	$\{v_0, v_3\}$	$\{v_0, v_4\}$	$\{v_0, v_5\}$	$\{v_1, v_3\}$	$\{v_1, v_4\}$	$\{v_1, v_5\}$
Edge on right	$\{w_0, w_1\}$	$\{w_0, w_3\}$	$\{w_0, w_5\}$	$\{w_2, w_1\}$	$\{w_2, w_3\}$	$\{w_2, w_5\}$
(Cont.) Edge on left	$\{v_2, v_3\}$	$\{v_2, v_4\}$	$\{v_2, v_5\}$			
Edge on right	$\{w_2, w_1\}$	$\{w_2, w_3\}$	$\{w_2, w_5\}$			

3.15 DEFINITION Two graphs \mathcal{G} and $\hat{\mathcal{G}}$ are **isomorphic** if there is a one-to-one and onto map $f: \mathcal{N} \rightarrow \hat{\mathcal{N}}$ such that \mathcal{G} has an edge $\{v_i, v_j\} \in \mathcal{E}$ if and only if $\hat{\mathcal{G}}$ has the associated edge $\{f(v_i), f(v_j)\} \in \hat{\mathcal{E}}$.

To verify that two graphs are isomorphic the most natural thing is to produce the map f and then verify that in consequence the edges also correspond. The exercises have examples.

Showing that graphs are not isomorphic usually entails finding some graph-theoretic way in which they differ. A common and useful such property is to consider the **degree of a vertex**, the total number of edges touching that vertex with the proviso that a loop from the vertex to itself counts as two. The **degree sequence** of a graph is the non-increasing sequence of its vertex degrees. Thus, the graph in Example 3.14 has degree sequence $\langle 3, 2, 1, 1, 1 \rangle$.

Exercise 3.38 shows that if graphs are isomorphic then associated vertices have the same degree and thus graphs with different degree sequences are not isomorphic. Also, if the degree sequences are equal then they help us construct an isomorphism, if there is one; examples of this are in the exercises. (Note, though, that there are graphs with the same degree sequence that are not isomorphic.)



HOW TO ANNOY A
GRAPH THEORY PH.D.

Courtesy xkcd
.com

We do not know if we can quickly determine whether two graphs are isomorphic. There are several algorithms that in practice do well for most graphs but in the worst case take exponential time. More on algorithm speed, including the speed of a number of graph algorithms, is in the final chapter.

III.3 Exercises

- ✓ 3.16 Draw a picture of a graph illustrating each relationship. Some graphs will be digraphs, or may have loops or multiple edges between some pairs of vertices.
 - (A) Maine is adjacent Massachusetts and New Hampshire. Massachusetts is adjacent to every other state. New Hampshire is adjacent to Maine, Massachusetts, and Vermont. Rhode Island is adjacent to Connecticut and Massachusetts. Vermont is adjacent to Massachusetts and New Hampshire. Give the graph describing the adjacency relation.
 - (B) In the game of Rock-Paper-Scissors, Rock beats Scissors, Paper beats Rock, and Scissors beats Paper. Give the graph of the ‘beats’ relation; note that this is a directed relation.
 - (C) The number $m \in \mathbb{N}$ is related to the number $n \in \mathbb{N}$ by being its divisor if there is a $k \in \mathbb{N}$ with $m \cdot k = n$. Give the divisor relation graph among positive natural numbers less than or equal to 12.
 - (D) The river Pregel cut the town of Königsberg into four land masses. There were two bridges from mass 0 to mass 1 and one bridge from mass 0 to mass 2. There was one bridge from mass 1 to mass 2, and two bridges from mass 1 to mass 3. Finally, there was one bridge from mass 2 to 3. Consider masses related by bridges. Give the graph (it is a multigraph).
 - (E) In our Mathematics program you must take Calculus II before you take Calculus III, and you must take Calculus I before II. You must take Calculus II before Linear Algebra, and to take Real Analysis you must have both Linear Algebra and Calculus III.

- 3.17 Put ‘Y’ or ‘N’ in the array cells for these kinds of walks.

	Vertices can repeat?	Edges can repeat?	Can be closed?	Can be open?
Walk	—	—	—	—
Trail	—	—	—	—
Circuit	—	—	—	—
Path	—	—	—	—
Cycle	—	—	—	—

- 3.18 Let a graph \mathcal{G} have vertices $\{v_0, \dots, v_5\}$ and the edges $v_0v_1, v_0v_3, v_0v_5, v_1v_4, v_3v_4$, and v_4v_5 . (A) Draw \mathcal{G} . (B) Give its adjacency matrix. (C) Find all subgraphs with four nodes and four edges. (D) Find all induced subgraphs with four nodes and four edges.

- 3.19 The **complete graph** on n vertices, K_n is the simple graph with all possible

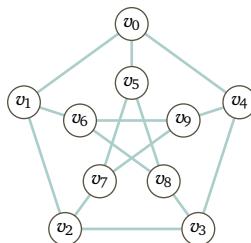
edges. (A) Draw K_4 , K_3 , K_2 , and K_1 . (B) Draw K_5 . (C) How many edges does K_n have?

- ✓ 3.20 Morse code represents text with a combination of a short sound, written ‘·’ and pronounced “dit,” and a long sound, written ‘—’ and pronounced “dah.” Here are the representations of the twenty six English letters.

A	··	F	····	K	···	O	---	S	··	W	···
B	····	G	---	L	····	P	····	T	-	X	····
C	····	H	····	M	--	Q	····	U	··	Y	····
D	··	I	..	N	--	R	···	V	····	Z	····
E	·	J	····								

Some representations are prefixes of others. Give the graph for the prefix relation.

- 3.21 This is the **Petersen graph**, often used for examples in Graph Theory.

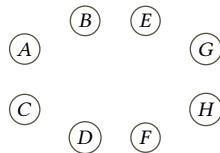


- (A) List the vertices and edges. (B) Give two walks from v_0 to v_7 . What is the length of each? (C) List both a closed walk and an open walk of length five, starting at v_4 . (D) Give a cycle starting at v_5 . (E) Is this graph connected?

- 3.22 A graph is not a drawing, it is a set of vertices and edges. So a single graph may have quite different pictures. Consider a graph \mathcal{G} with the vertices $\mathcal{N} = \{A, \dots, H\}$ and these edges.

$$\mathcal{E} = \{AB, AC, AG, AH, BC, BD, BF, CD, CE, DE, DF, EF, EG, FH, GH\}$$

- (A) Connect the dots below to get one drawing.



- (B) A **planar graph** is one that can be drawn in the plane so that its edges do not cross. Show that \mathcal{G} is planar.

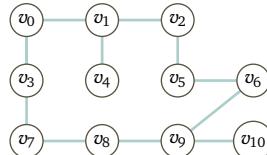
- 3.23 Show that every tree has a 2-coloring.

- 3.24 A person keeps six species of fish as pets. Species A cannot be in a tank with species B or C. Species B cannot be with A, C, or E. Species C cannot be with A, B, D, or E. Species D cannot be with C or F. Species E cannot be together with B, C, or F. Finally, species F cannot be in with D or E. (A) Draw the graph where

the nodes are species and the edges represent the relation ‘cannot be together’.

(B) Find the chromatic number. (C) Interpret it.

- ✓ 3.25 If two cell towers are within line of sight of each other then they must get different frequencies. Below each tower is a vertex and an edge between towers denotes that they can see each other. What is the minimal number of frequencies? Give an assignment of frequencies to towers.



3.26 For the graph in the prior exercise, give the degree sequence.

- ✓ 3.27 For a blood transfusion, unless the recipient is compatible with the donor’s blood type they can have a severe reaction. Compatibility depends on the presence or absence of two antigens, called A and B, on the red blood cells. This creates four major groups: A, B, O (the cells have neither antigen), and AB (the cells have both). There is also a protein called the Rh factor that can be either present (+) or absent (-). Thus there are eight common blood types, A+, A-, B+, B-, O+, O-, AB+, and AB-. If the donor has the A antigen then the recipient must also have it, and the B antigen and Rh factor work the same way. Draw a directed graph where the nodes are blood types and there is an edge from the donor to the recipient if transfusion is safe. Produce the adjacency matrix.

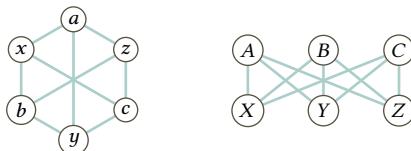
3.28 Find the degree sequence of the graph in Example 3.2 and of the two graphs of Example 3.4.

3.29 Give the array representation, like that in equation (*), for the graphs of Example 3.4.

3.30 Draw a graph for this adjacency matrix.

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

- ✓ 3.31 These two graphs are isomorphic.

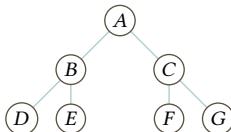


(A) Define the function giving the correspondence.

(B) Verify that under that function the edges then also correspond.

3.32 For the two graphs in the prior exercise, give the degree sequences. Are they the same?

- ✓ 3.33 Consider this tree.



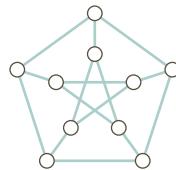
(A) Verify that $\langle BA, AC \rangle$ is a path from B to C .

(B) Why is $\langle BD, DB, BA, BC \rangle$ not also a path from B to C ?

(C) Show that in any tree, for any two vertices there is a unique path from one to the other.

3.34 For the tree in the prior exercise, give the degree sequence.

3.35 This is the Petersen graph.



(A) Show that it has no 2-coloring.

(B) Give a 3-coloring.

- ✓ 3.36 A **graph traversal** is a sequence listing each vertex in a graph. The sequence need not follow the edges and some vertices may repeat. (See Extra B.)

(A) In a connected tree, the **rank** of a vertex is the number of edges in the shortest path between that vertex and the root. Thus, vertex E has rank 2. A breadth first traversal lists vertices in rank order, so all vertices of rank k are listed before any of rank $k + 1$. That is, a breadth first traversal visits sibling vertices before visiting any child vertices. Give a breadth first traversal of Exercise 3.33's tree.

(B) A depth first traversal visits child vertices before sibling vertices. Give a depth first traversal of the same tree.

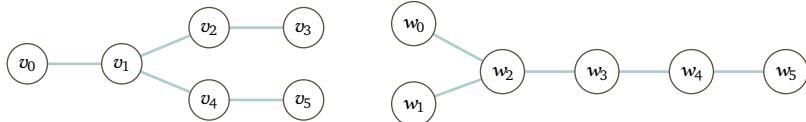
3.37 Consider building a simple graph by starting with n vertices. (A) How many potential edges are there? (B) How many such graphs are there? (C) List the number of such graphs for $n = 0$ through $n = 6$.

3.38 We can use degrees and degree sequences to help find isomorphisms, or to show that graphs are not isomorphic. (Here we allow graphs to have loops and to have multiple edges between vertices, but we do not make the extension to directed edges or edges with weights.)

(A) Show that if two graphs are isomorphic then they have the same number of vertices.

(B) Show that if two graphs are isomorphic then they have the same number of edges.

- (c) Show that if two graphs are isomorphic and one has a vertex of degree k then so does the other.
- (d) Show that if two graphs are isomorphic then for each degree k , the number of vertices of the first graph having that degree equals the number of vertices of the second graph having that degree. Thus isomorphic graphs have degree sequences that are equal.
- (e) Verify that while these two graphs have the same degree sequence, they are not isomorphic. *Hint:* consider the paths starting at the degree 3 vertex.



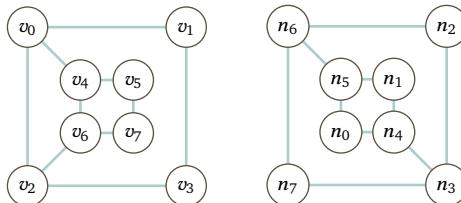
- (f) Use the prior result to show that the two graphs of Example 3.4 are not isomorphic.

As in the final item, in arguments we often use the contrapositive of these statements. For instance, the first item implies that if they do not have the same number of vertices then they are not isomorphic.

3.39 Prove Lemma 3.12.

- (a) An edge is a length-one walk. Show that in the product of the matrix with itself $(\mathcal{M}(\mathcal{G}))^2$ the entry i, j is the number of length-two walks.
- (b) Show that for $n > 2$, the i, j entry of the power $(\mathcal{M}(\mathcal{G}))^n$ equals the number of length n walks from v_i to v_j .

3.40 Consider these two graphs, \mathcal{G}_0 and \mathcal{G}_1 .



- (a) List the vertices and edges of \mathcal{G}_0 . Do the same for \mathcal{G}_1 .
- (b) Give the degree sequences of \mathcal{G}_0 and \mathcal{G}_1 .
- (c) Consider this correspondence between the vertices.

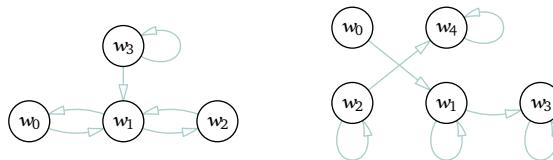
vertex of \mathcal{G}_0	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
vertex of \mathcal{G}_1	n_6	n_2	n_7	n_3	n_5	n_1	n_0	n_4

- Find the image, under the correspondence, of the edges of \mathcal{G}_0 . Do they match the edges of \mathcal{G}_1 ?
- (d) Of course, failure of any one proposed map does not imply that the two cannot be isomorphic. Nonetheless, argue that they are not isomorphic.

3.41 In a graph, for a node q_0 there may be some nodes q_i that are unreachable, so there is no path from q_0 to q_i .

- (a) Devise an algorithm that inputs a directed graph and a start node q_0 , and finds the set of nodes that are unreachable from q_0 .

(b) Apply your algorithm to these two starting with w_0 .



EXTRA

III.A BNF



John Backus
1924–2007

We shall introduce some grammar notation conveniences that are widely used. Together they are called **Backus-Naur form, BNF**.

The study of grammar, the rules for phrase structure and forming sentences, has a long history, dating back as early as the fifth century BC. Mathematicians, including A Thue and E Post, began systematizing it as rewriting rules by the early 1900's. The variant we see here was produced by J Backus in the late 1950's as part of the design of the early computer language ALGOL60. Since then these rules have become the most common way to express grammars.

One difference from the prior subsection is a minor typographical change. Originally the metacharacters were not typeable with a standard keyboard. The advantage of having metacharacters not on a keyboard is that most likely all of the language characters are typeable so there is no need to distinguish, say, between the pipe character | when used as a part of a language and when used as a metacharacter. But the disadvantage lies in having to type the untypeable. In the end the convenience of having typeable characters won over the technical gain of having typographically distinguish metacharacters. For instance, for a long time standard keyboards had no arrows so in place of the arrow symbol ' \rightarrow ', BNF uses ' $::=$ '. (These adjustments were made by P Naur as editor of the ALGOL60 report.)[†]



Peter Naur
1928–2016

BNF is both clear and concise, it can express the range of languages that we ordinarily want to express, and it smoothly translates to a parser. That is, BNF is an impedance match—it fits with what we want to do. Here we will incorporate some extensions for grouping and replication that are like what you typically see in the wild.[‡]

- A.1 EXAMPLE This is a BNF grammar for real numbers with a finite decimal part. To the rules for $\langle \text{natural} \rangle$ from Example 2.6, add these.

```
 $\langle \text{start} \rangle ::= -\langle \text{fraction} \rangle \mid +\langle \text{fraction} \rangle \mid \langle \text{fraction} \rangle$ 
 $\langle \text{fraction} \rangle ::= \langle \text{natural} \rangle \mid \langle \text{natural} \rangle . \langle \text{natural} \rangle$ 
```

[†]There are other typographical issues that arise with grammars. While many authors write nonterminals with diamond brackets, as we do, others use other conventions such as a separate type style or color.

[‡]BNF is only loosely defined. While there are standards, often what you see does not conform exactly to any standard.

This derivation for 2.718 is rightmost.

```

⟨start⟩ ⇒ ⟨fraction⟩ ⇒ ⟨natural⟩ . ⟨natural⟩
⇒ ⟨natural⟩ . ⟨digit⟩⟨natural⟩ ⇒ ⟨natural⟩ . ⟨digit⟩⟨digit⟩⟨natural⟩
⇒ ⟨natural⟩ . ⟨digit⟩⟨digit⟩⟨digit⟩ ⇒ ⟨natural⟩ . ⟨digit⟩⟨digit⟩8
⇒ ⟨natural⟩ . ⟨digit⟩18 ⇒ ⟨natural⟩ . 718 ⇒ 2.718

```

Here is a derivation for 0.577 that is neither leftmost nor rightmost.

```

⟨start⟩ ⇒ ⟨fraction⟩ ⇒ ⟨natural⟩ . ⟨natural⟩
⇒ ⟨natural⟩ . ⟨digit⟩⟨natural⟩ ⇒ ⟨natural⟩ . 5⟨natural⟩
⇒ ⟨natural⟩ . 5⟨digit⟩⟨natural⟩ ⇒ ⟨digit⟩ . 5⟨digit⟩⟨natural⟩
⇒ ⟨digit⟩ . 5⟨digit⟩⟨digit⟩ ⇒ ⟨digit⟩ . 5⟨digit⟩7 ⇒ 0.5⟨digit⟩7
⇒ 0.577

```

- A.2 EXAMPLE Time is a difficult engineering problem. One issue is representing times and one standard in that area is RFC 3339, *Date and Time on the Internet: Timestamps*. It uses strings such as 1958-10-12T23:20:50.52Z. Here is part of a BNF grammar. This grammar includes some metacharacter extensions discussed below.

```

⟨date-fullyear⟩ ::= <4-digits>
⟨date-month⟩ ::= <2-digits>
⟨date-mday⟩ ::= <2-digits>
⟨time-hour⟩ ::= <2-digits>
⟨time-minute⟩ ::= <2-digits>
⟨time-second⟩ ::= <2-digits>
⟨time-secfrac⟩ ::= . <1-or-more-digits>
⟨time-numoffset⟩ ::= (+ | -) <time-hour⟩ : <time-minute⟩
⟨time-offset⟩ ::= Z | <time-numoffset⟩
⟨partial-time⟩ ::= <time-hour⟩ : <time-minute⟩ : <time-second⟩
[⟨time-secfrac⟩]
⟨full-date⟩ ::= <date-fullyear⟩ - <date-month⟩ - <date-mday⟩
⟨full-time⟩ ::= <partial-time⟩ <time-offset⟩
⟨date-time⟩ ::= <full-date⟩ T <full-time⟩

```

There are a number of BNF notations that are more than simple substitutions, instead extending our grammar rules. One is shown above in the $\langle\text{partial-time}\rangle$ rule, which includes square brackets as metacharacters to denote that the $\langle\text{time-secfrac}\rangle$ is optional. This is a very common construct: another example of it is this syntax description for if ... then ... with an optional else ...

$\langle \text{if-stmt} \rangle ::= \text{if } \langle \text{boolean-expr} \rangle \text{ then } \langle \text{stmt-sequence} \rangle$
 $\quad [\text{else } \langle \text{stmt-sequence} \rangle] \langle \text{end if} \rangle ;$

To show repetition, BNF may use a Kleene star * for ‘zero or more’, as here.

$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle (\langle \text{letter} \rangle \mid \langle \text{digit} \rangle)^*$

To express ‘one or more’ in BNF you often see a +

A.3 EXAMPLE This grammar for Python floating point numbers shows both square brackets and the plus sign.

```

 $\langle \text{floatnumber} \rangle ::= \langle \text{pointfloat} \rangle \mid \langle \text{exponentfloat} \rangle$ 
 $\langle \text{pointfloat} \rangle ::= [ \langle \text{intpart} \rangle ] \langle \text{fraction} \rangle \mid \langle \text{intpart} \rangle .$ 
 $\langle \text{exponentfloat} \rangle ::= ( \langle \text{intpart} \rangle \mid \langle \text{pointfloat} \rangle ) \langle \text{exponent} \rangle$ 
 $\langle \text{intpart} \rangle ::= \langle \text{digit} \rangle ^+$ 
 $\langle \text{fraction} \rangle ::= . \langle \text{digit} \rangle ^+$ 
 $\langle \text{exponent} \rangle ::= (\text{e } \mid \text{E}) [ + \mid - ] \langle \text{digit} \rangle ^+$ 

```

In the $\langle \text{pointfloat} \rangle$ rule the first $\langle \text{intpart} \rangle$ is optional. And, an $\langle \text{intpart} \rangle$ consists of one or more digits.

Each of these extension constructs is not necessary in that we can express the grammars without the extensions. For instance, we could replace the this use of Kleene star

$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle (\langle \text{letter} \rangle \mid \langle \text{digit} \rangle)^*$

with this.

```

 $\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{letter} \rangle \langle \text{atoms} \rangle$ 
 $\langle \text{atoms} \rangle ::= \langle \text{letter} \rangle \langle \text{atoms} \rangle \mid \langle \text{digit} \rangle \langle \text{atoms} \rangle \mid \epsilon$ 

```

But these constructs come up often enough that adopting an abbreviation is a significant convenience.

Passing from the grammar to a parser for that grammar is mechanical. There are programs that take as input a grammar, often one in BNF, and give as output source code that will parse files following that grammar’s format. Such a program is a **parser-generator** (sometimes instead called a **compiler-compiler**, which is a fun term but is misleading because a parser is only part of a compiler).

III.A Exercises

- ✓ A.4 US ZIP codes have five digits, and may have a dash and four more digits at the end. Give a BNF grammar.
- A.5 Write a grammar in BNF for the language of palindromes.
- ✓ A.6 At a college, course designations have a form like ‘MA 208’ or ‘PSY 101’, where the department is two or three capital letters and the course is three digits. Give a BNF grammar.

- ✓ A.7 Example A.3 uses some BNF convenience abbreviations.
 - (A) Give a grammar equivalent to $\langle pointfloat \rangle$ that doesn't use square brackets.
 - (B) Do the same for the repetition operator in $\langle intpart \rangle$'s rule, and for the grouping in $\langle exponent \rangle$'s rule (you can use $\langle intpart \rangle$ here).
- ✓ A.8 In Roman numerals the letters I, V, X, L, C, D, and M stand for the values 1, 5, 10, 50, 100, 500, and 1 000. We write the letters from left to right in descending order of value, so that XVI represents the number that we would ordinarily write as 16, and MDCCCCCLVIII represents 1958. We always write the shortest possible string, so we do not write IIIII because we can instead write V. However, as we don't have a symbol whose value is larger than 1 000 we must represent large numbers with lots of M's.
 - (A) Give a grammar for the strings that make sense as Roman numerals.
 - (B) Often Roman numerals are written in subtractive notation: for instance, 4 is represented as IV, because four I's are hard to distinguish from three of them in a setting such as a watch face. In this notation 9 is IX, 40 is XL, 90 is XC, 400 is CD, and 900 is CM. Give a grammar for the strings that can appear in this notation.

A.9 This grammar is for a small C-like programming language.

```

⟨program⟩ ::= { ⟨statement-list⟩ }

⟨statement-list⟩ ::= [ ⟨statement⟩ ; ]*

⟨statement⟩ ::= ⟨data-type⟩ ⟨identifier⟩
| ⟨identifier⟩ = ⟨expression⟩
| print ⟨identifier⟩
| while ⟨expression⟩ { ⟨statement-list⟩ }

⟨data-type⟩ ::= int | boolean

⟨expression⟩ ::= ⟨identifier⟩ | ⟨number⟩ | ( ⟨expression⟩ ⟨operator⟩
⟨expression⟩ )

⟨identifier⟩ ::= ⟨letter⟩ [ ⟨letter⟩ ]*

⟨number⟩ ::= ⟨digit⟩ [ ⟨digit⟩ ]*

⟨operator⟩ ::= + | ==

⟨letter⟩ ::= A | B | ... | Z

⟨digit⟩ ::= 0 | 1 | ... | 9

```

(A) Give a derivation and parse tree for this program.

```

{ int A ;
  A = 1 ;
  print A ;
}

```

(B) Must all programs be surrounded by curly braces?

A.10 Here is a grammar for LISP.

```

⟨s-expression⟩ ::= ⟨atomic-symbol⟩
| ( ⟨s-expression⟩ . ⟨s-expression⟩ )
| ⟨list⟩
⟨list⟩ ::= ( ⟨s-expression⟩* )
⟨atomic-symbol⟩ ::= ⟨letter⟩ ⟨atom-part⟩
⟨atom-part⟩ ::= ⟨empty⟩
| ⟨letter⟩ ⟨atom-part⟩
| ⟨number⟩ ⟨atom-part⟩
⟨letter⟩ ::= a | b | ... z
⟨number⟩ ::= 1 | 2 | ... 9

```

Derive the s-expression (cons (car x) y).

A.11 Python 3's Format Specification Mini-Language is used to describe string substitution.

```

⟨format-spec⟩ ::= [[⟨fill⟩]⟨align⟩][⟨sign⟩][#][0][⟨width⟩][⟨gr⟩][.⟨precision⟩][⟨type⟩]
⟨fill⟩ ::= ⟨any character⟩
⟨align⟩ ::= < | > | = | ^
⟨sign⟩ ::= + | - |
⟨width⟩ ::= ⟨integer⟩
⟨gr⟩ ::= - | ,
⟨precision⟩ ::= ⟨integer⟩
⟨type⟩ ::= b | c | d | e | E | f | F | g | G | n | o | s | x | X | %

```

Take ⟨integer⟩ to produce ⟨digit⟩ ⟨integer⟩ or ⟨digit⟩. Give a derivation of these strings: (A) 03f (B) +#02X.

EXTRA

III.B Tree traversal

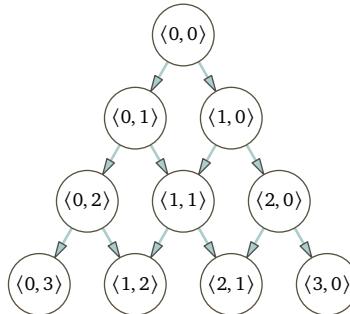
We sometimes need to visit each vertex in a graph. An example is that when we described Cantor's correspondence enumerating the set $\mathbb{N} \times \mathbb{N}$, we drew this array.

⋮	⋮	⋮	⋮	⋮
⟨0, 3⟩	⟨1, 3⟩	⟨2, 3⟩	⟨3, 3⟩	...
⟨0, 2⟩	⟨1, 2⟩	⟨2, 2⟩	⟨3, 2⟩	...
⟨0, 1⟩	⟨1, 1⟩	⟨2, 1⟩	⟨3, 1⟩	...
⟨0, 0⟩	⟨1, 0⟩	⟨2, 0⟩	⟨3, 0⟩	...

The enumeration starts like this.

Number	0	1	2	3	4	5	6	...
Pair	⟨0, 0⟩	⟨0, 1⟩	⟨1, 0⟩	⟨0, 2⟩	⟨1, 1⟩	⟨2, 0⟩	⟨0, 3⟩	...

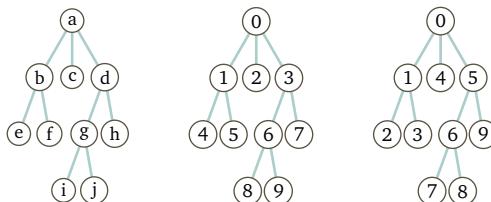
To view this as a graph traversal, below we make a graph by connecting each pair in the array to two of its neighbors, above and to the right.[†] This redraws the graph with the array's lower-left rotated to the top.



With this presentation, Cantor's enumeration lists the nodes in order of how far they are from the top: first the top node, then the nodes that are one away, then those that are two away, etc.[‡]

We will present Racket code to traverse trees. The above graph isn't a tree, instead it is a DAG, a directed acyclic graph. But the two share the key feature that they have no cycles and for our purposes that is enough.

On the left below is a tree with ten nodes. In the middle we visit the nodes in **breadth-first** order, where we cover all nodes that are at rank k before visiting any nodes of rank $k + 1$ (a node is **rank** k when a minimal path to the tree's root has k edges). On the right we visit the nodes in **depth-first** order, where we visit child nodes before going on to visit sibling nodes.



A tree is a graph, and thus is collection of connected nodes. We first define a node. Essentially it is an ordered pair.

```
(struct node (name children))
```

The **struct** construct has the system automatically define a number of useful functions. One is a creator function (`node n s`). We use that to make the next function, which knows that `children` is a set.

```
(define (node-create name)
  (node name (mutable-set)))
```

It is a mutable set because as we create the tree we will add children to that set, so we will want to change it. In turn, we use that routine to create new trees.

[†]If we only did above or only to the right then the graph would not be connected. [‡]A traversal does not have to follow the edges, it just has to reach all the nodes.

```
(define (tree-create root-node-name)
  (node-create root-node-name))
```

This adds a child to the set of children of the given node.

```
(define (node-add-child! parent child-name)
  (let ([n (node-create child-name)])
    (set-add! (node-children parent) n)
    n))
```

Note the `node-children` routine. Another thing that `struct` does is to create accessor functions—we didn’t separately write `node-children`, the system creates it from the two names when we declared the `struct`. (LISP-derived languages have a convention of using an exclamation mark for the names of procedures whose main role is to cause side effects, not to return something.)[†]

This makes the ten-node tree drawn above.

```
(define (sample-tree-make)
  (let* ([t (tree-create "a")]
        [nb (node-add-child! t "b")]
        [nc (node-add-child! t "c")]
        [nd (node-add-child! t "d")]
        [ne (node-add-child! nb "e")]
        [nf (node-add-child! nb "f")]
        [ng (node-add-child! nd "g")]
        [nh (node-add-child! nd "h")]
        [ni (node-add-child! ng "i")]
        [nj (node-add-child! ng "j")]
        )
    t))
```

We are ready for the traversal code. At each node we will just print out the node name,

```
(define (show-node-name n r)
  (printf "~a~a\n" (string-pad r) (node-name n)))
```

indented by some spaces to indicate the rank.

```
(define (string-pad n)
  (apply string-append (build-list n (lambda (x) " "))))
```

The first traversal is breadth-first.

```
(define (tree-bfs node fcn #:maxrank [maxrank MAXIMUM-RANK])
  (tree-bfs-helper (list node) 0 fcn #:maxrank maxrank))

(define (tree-bfs-helper level rank fcn #:maxrank [maxrank MAXIMUM-RANK])
  (when (< rank maxrank)
    (let ([next-level '()])
      (for ([node level])
        (fcn node rank)
        (for ([child-node (node-children node)])
          (set! next-level (cons child-node next-level))
          )))
      (when (not (null? next-level))
        (tree-bfs-helper next-level (+ 1 rank) fcn)))))
```

[†]The code here has a way to tie from parent to child but no direct way to tie back. So we could consider that this isn’t a tree because trees aren’t directed. But below this won’t matter so we will ignore it.

It comes in two functions, `tree-bfs` and `tree-bfs-helper`. In Scheme-derived languages such as Racket, routines are often organized with a caller and a helper. This is because the helper function is **tail-recursive**. Its very last thing is a recursion, and in a Scheme language the compiler knows that it can translate such a routine into executable code that is iterative. This combines the expressiveness of recursion with the memory conservation of iteration.

There is a parameter `MAXIMUM-RANK` that is defined elsewhere in the file.

```
(define MAXIMUM-RANK 100)
```

This is handy for testing, to keep the routine from running away. In the declaration of `tree-bfs`, the syntax `#: maxrank [maxrank MAXIMUM-RANK]` makes it an optional keyword argument.

The routine `tree-bfs` inputs the node that is the top of the tree, naturally, but also has a `fcn` input. Racket and other LISP-derived languages allow you to pass functions. We will pass `show-node-name` and it gets called on the `(fcn node rank)` line.

The strategy of `tree-bfs-helper` is that when `rank` is k then the routine traverses all nodes at that rank, which it gets from the list `level`. As it traverses that list, it stores all of the children of those nodes in the list `next-level`.

Here is the result of running the routine.

```
> (define t (sample-tree-make))
> (tree-bfs t show-node-name)
a
  c
  d
  b
    e
    f
    g
    h
      i
      j
```

The children are stored in a set, not a list. So they can come out in an order that is different than how they went in, which is why it says `c`, then `d`, and then `b`. Nonetheless, this is breadth first because it traverses one level at a time.

The depth first routine is in some ways more natural in Racket.

```
(define (tree-dfs node rank fcn #:maxrank [maximumrank MAXIMUM-RANK])
  (fcn node rank)
  (when (< rank maximumrank)
    (let ([children (node-children node)])
      (for ([child children])
        (tree-dfs child (+ rank 1) fcn #:maxrank maximumrank)))
    )))
```

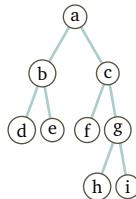
For example, there is no need for a helper function. Here it is in action.

```
> (tree-dfs t 0 show-node-name)
a
  b
  f
    e
  d
```



III.B Exercises

B.1 This is a **binary tree** because each node has either two children or none (in the definition some authors also allow one child).



- (A) Give the Racket code to define this tree.
- (B) Perform a breadth first traversal.
- (C) Do depth first.

B.2 We've noted that the graph for the Cantor enumeration is not a tree, it is a directed acyclic graph.

- (A) Give the Racket code to define this graph.
- (B) What happens if you run the `tree-bfs` routine on it?
- (C) Adjust the `tree-bfs` code to make it work.

COMPUT ECCLESIASTIQUE

POUR LA GRACE

2009

This close-up view of the clock's dial focuses on the "CYCLE SOLAIRE" section. The dial features a dark background with gold-colored numbers from 1 to 25 around the perimeter. In the center, there are two sets of hands: one for the sun and one for the moon. The sun's path is indicated by a series of gold gears, while the moon's path is shown as a smaller circle with a crescent phase indicator. The text "CYCLE SOLAIRE" is printed at the top of the dial.

18 9 1
17 NOMBRE D'OR 2
6

14
INDICTION

B LETTRE DOMINICALE G

EPACTE

CHAPTER

IV Automata

Our touchstone model of computation is the Turing machine. It has two components, the CPU and the tape. In this chapter we will focus on the CPU—we will consider what can be done with states alone, what can be done by a machine having a number of possible configurations that is bounded.[†]

SECTION

IV.1 Finite State machines

We produce a new model of computation, the Finite State machine, by modifying the Turing machine definition. We will strip out the capability to write, changing from read/write to read-only. It will turn out that these machines can do many things, but not as many as Turing machines.

Definition We begin with some examples.

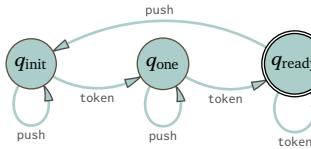
- 1.1 EXAMPLE This power switch has two states, q_{off} and q_{on} , and its input alphabet has one token, `toggle`.



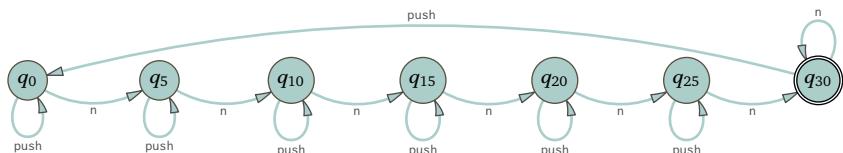
The state q_{on} is drawn with a double circle, denoting that it is a different kind of state than q_{off} . Finite State machines can't write to the tape so they need some other way to declare the computation's outcome. We say that q_{on} is an **accepting state** or **final state**. A computation accepts its input string if it ends with the machine in an accepting state.

- 1.2 EXAMPLE Operate the turnstile below by putting in two tokens and then pushing through. It has three states and its input alphabet is $\Sigma = \{\text{token}, \text{push}\}$. As with Turing machines, the states here serve as a form of memory, although a limited one. For instance, q_{one} is how the turnstile “remembers” that it has so far received one token.

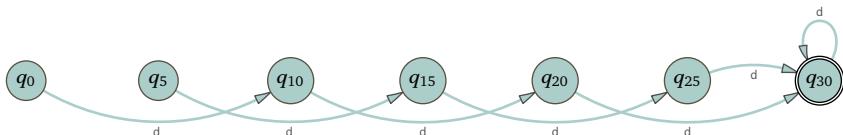
IMAGE: The astronomical clock in Notre-Dame-de-Strasbourg Cathedral, for computing the date of Easter. Easter falls on the first Sunday after the first full moon on or after March 21, the nominal spring equinox. Calculation of this date was a great challenge for mechanisms of that time, 1843. [†] Studying the parts of the machine is natural but there is another motivation. A person could object to Turing's model that there is a machine that iterates writing a character and then moving right, and thereby occupies unboundedly many configurations, but no physical device can do that. A rejoinder is that we for instance define a ‘book’ to be pages with writing and don't worry whether physics limits the number of pages. Happily, we don't need to go into this to justify our interest. For one thing, Finite State machines are quite practical and appear often in everyday computing.



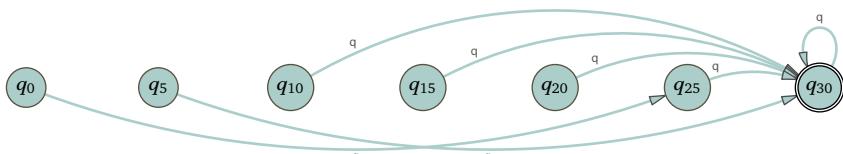
- 1.3 EXAMPLE This vending machine dispenses items that cost 30 cents.[†] The picture is complex so we will show it in three layers. First are the arrows for nickels.



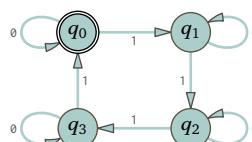
After receiving 30 cents and getting another nickel, this machine does something not very sensible: it stays in q_{30} . In practice a machine would have further states to keep track of overages so that it could give change but here we ignore that. Next comes the arrows for dimes



and for quarters.



- 1.4 EXAMPLE This machine's alphabet is the set of bits, $\mathbb{B} = \{0, 1\}$. As an example, if the input string is $\sigma = 101101$ then the machine reads those bits, first passing from q_0 into q_1 and q_1 again, then through q_2 and q_3 and q_3 again, before finally returning to q_0 . As q_0 has a double circle, the machine accepts σ .



This machine accepts a bitstring if the number of 1's in its input is a multiple of four.

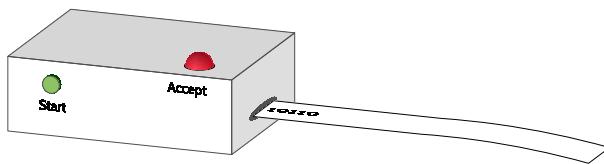
We defined Turing machines as sets of instructions. Instructions have the

[†]US coins are: 1 cent coins not used here, nickels are 5 cents, dimes are 10 cents, and quarters are 25.

advantage of being intuitive, and of matching how we think about CPU's on everyday computers. While we have later occasionally referred to instructions, what has mattered most is that they describe the machine's next-state function, Δ . For the definition of Finite State machines we will cut right to giving it in terms of the next-state function.

- 1.5 **DEFINITION** A **Finite State machine** or **Finite State automata** $\langle Q, q_0, F, \Sigma, \Delta \rangle$ has a finite set of states Q , one of which is the **start state** q_0 , a subset $F \subseteq Q$ of **final states** or **accepting states**, a finite **input alphabet** set Σ , and a **next-state function** or **transition function** $\Delta: Q \times \Sigma \rightarrow Q$.

A full description of the action of these machines comes after a few more examples. But basically, to work a machine just load a string input on the tape and press Start. At each step the machine consumes one tape token, that is, it reads, acts on, and deletes that token, and then the head moves to the next token.



The picture shows a light labeled 'Accept'. When the machine stops, when the input string is fully consumed, if the current state is an accepting state then the light comes on. In this case we say that the machine **accepts** the input string, otherwise it **rejects** that string.

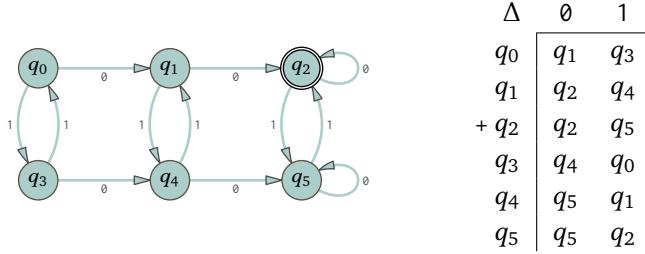
Here is a trace of the steps when Example 1.4's modulo 4 machine is started with the input string $\tau = 10110$ on the tape.

<i>Step</i>	<i>Configuration</i>	<i>Step</i>	<i>Configuration</i>
0	1 0 1 1 0 q0	3	1 0 q2
1	0 1 1 0 q1	4	0 q3
2	1 1 0 q1	5	blank q3

It rejects τ since q_3 is not an accepting state.

Because these machines consume one character per step and stop once all the characters are gone, they are sure to halt—there is no **Halting problem** for Finite State machines.

- 1.6 **EXAMPLE** The machine below accepts a string if and only if it contains at least two 0's as well as an even number of 1's. (In tables we denote accepting states with '+', as here next to q_2).

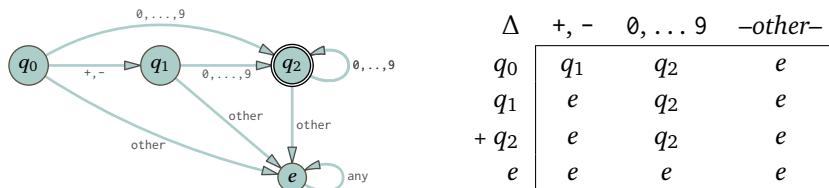


- 1.7 REMARK This is a chance to articulate the key to designing Finite State machines, that each state should have an intuitive purpose that is directed at the machine's purpose, and that is unique. The drawing from that example brings this out. Its top row has states where the machine has so far seen an even number of 1's (possibly zero many), while its states in the bottom row have seen an odd number. Its first column holds states where the machine has seen no 0's, the second column holds states where there has been one, and the third has states where there have been two. Thus q_4 means "so far the machine has seen one 0 and an odd number of 1's," and q_5 means "so far the machine has seen two 0's but an odd number of 1's."

Our Finite State machine descriptions often take the alphabet to be clear from the context. Thus, Example 1.6's alphabet is $\mathbb{B} = \{ 0, 1 \}$. For in-practice machines, the alphabet is the set of characters that the machine could conceivably receive, so that a text-handling routine built to modern standards might well accept all of Unicode. But for the examples and exercises in this book we will use small alphabets.[†]

- 1.8 EXAMPLE This machine accepts strings that are valid decimal representations of integers. So it accepts the strings 21 and -707 but does not accept 501-.

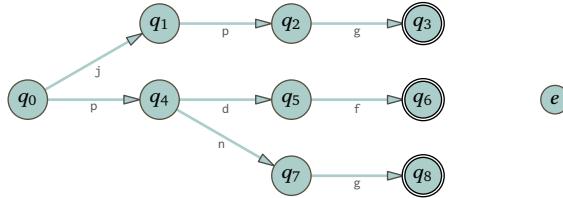
The transition graph and the table both group some inputs together when they result in the same action. For instance, when in state q_0 this machine does the same thing whether the input is + or -, namely it passes into q_1 .



Any wrong input character sends the machine to the state e . Finite State machines often have an error state, which is a sink in that once the machine enters that state then it never leaves.

- 1.9 EXAMPLE This machine accepts strings that are members of the set { jpg, pdf, png }. It is our first example with more than one final state.

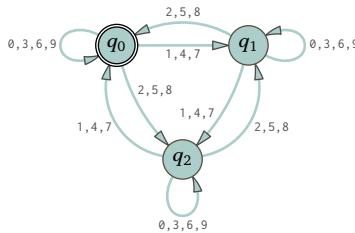
[†]We often use a, b, c, etc., in the alphabet because something like 'c²' is clearer than '0²' or 'two 0's'.



That drawing omits many edges, the ones involving the error state e . For instance, from state q_0 any input character other than j or p is an error. We omit all of these edges because they would make the drawing hard to read. This illustrates that for complex machines a transition table presentation is better than a picture.

That example points out that if a language is finite then there is a Finite State machine that accepts a string if and only if it is a member of that language.

- 1.10 EXAMPLE Finite State machines can accomplish reasonably hard tasks. This one accepts strings representing natural numbers that are multiples of three such as 15 and 8013, and does not accept non-multiples such as 14 and 8012.



This machine accepts the empty string. Exercise 1.26 asks for a modification to accept only non-empty strings.

- 1.11 EXAMPLE Finite State machines translate easily to code. Here is the Racket code for the delta function of the prior example's multiple of three machine.

```
(define (delta state ch)
  (cond
    [(= state 0)
     (cond
       ((memv ch '(#\0 #\3 #\6 #\9)) 0)
       ((memv ch '(#\1 #\4 #\7)) 1)
       (else 2))]
    [(= state 1)
     (cond
       ((memv ch '(#\0 #\3 #\6 #\9)) 1)
       ((memv ch '(#\1 #\4 #\7)) 2)
       (else 0))]
    [else
     (cond
       ((memv ch '(#\0 #\3 #\6 #\9)) 2)
       ((memv ch '(#\1 #\4 #\7)) 0)
       (else 1))]))
```

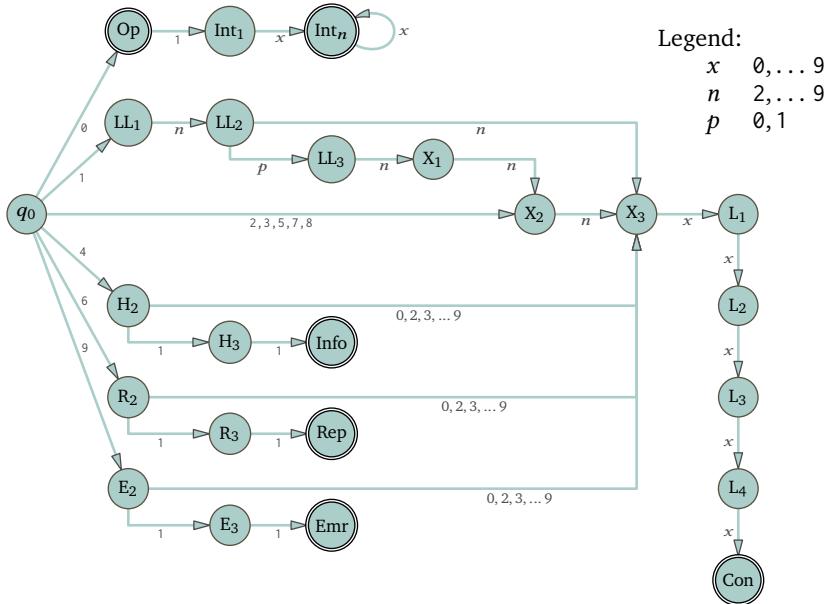
(In Racket, a character such as ‘0’ is denoted `#\0`. The routine `memv` decides if the character `ch` is in the list, that is, it is a boolean function.) All that's left is to supply a calling function

```
(define (multiple-of-three-fsm input-string)
  (let ((state 0))
    (if (= 0 (multiple-of-three-fsm-helper state (string->list input-string)))
        "accept"
        "reject")))
```

and a helper.

```
(define (multiple-of-three-fsm-helper state tau-list)
  (if (null? tau-list)
      state
      (multiple-of-three-fsm-helper (delta state (car tau-list))
                                    (cdr tau-list))))
```

- 1.12 EXAMPLE For many years this was how phone dialing was handled in North America. In the 1940's call connections were handled by simple physical devices for local calls or by operators for long distance calls. This plan worked within that hardware limitation, to change to allowing customers to directly dial long distance. Consider 1-802-555-0101. The initial 1 means that the call leaves the local office. The 802 is an area code; the system can tell that this is not a same-area local exchange because its second digit is 0 or 1. Next, the 555 routes the call to a local office. Then that office makes the connection to line 0101.



Today, no longer are area codes required to have a middle digit of 0 or 1. This additional flexibility is possible because switching now happens entirely in software.

After the definition of Turing machine we gave a formal description of the action of those machines. We will do the same here for Finite State machines. A **configuration** of a Finite State machine is a pair $C = \langle q, \tau \rangle$ where q is a state, $q \in Q$, and τ is a (possibly empty) string, $\tau \in \Sigma^*$. We start a machine in some

initial configuration $\mathcal{C}_0 = \langle q_0, \tau_0 \rangle$, so q_0 is the **initial state** and τ_0 is the **input string**.

Machines act by making a sequence of **transitions** from one configuration to another. For $s \in \mathbb{N}^+$ the machine's configuration after the s -th transition is its configuration at **step s** , denoted \mathcal{C}_s .

Here is the rule for making one transition. Begin with $\mathcal{C}_s = \langle q, \tau_s \rangle$. Either τ_s is empty or it is not. If τ_s is not empty then pop its leading character c . That is, because τ_s is not empty it decomposes into a first character and a remaining string, $\tau_s = c \tau_{s+1}$. With that character, the machine's next state is $\hat{q} = \Delta(q, c)$ and its next configuration is $\mathcal{C}_{s+1} = \langle \hat{q}, \tau_{s+1} \rangle$. We denote this before-after relationship between configurations with $\mathcal{C}_s \vdash \mathcal{C}_{s+1}$.[†]

The other possibility is that τ_s is the empty string. Then \mathcal{C}_s is a **halting configuration**, denoted \mathcal{C}_h . No transitions follow. Every Finite State machine eventually reaches a halting configuration because at each transition the tape string loses a character. A **Finite State machine computation** is a sequence $\mathcal{C}_0 \vdash \mathcal{C}_1 \vdash \dots \vdash \mathcal{C}_h$, which we abbreviate $\mathcal{C}_0 \vdash^* \mathcal{C}_h$.[‡] If \mathcal{C}_h 's state is a final state, so that $q \in F$ where $\mathcal{C}_h = \langle q, \varepsilon \rangle$, then the machine **accepts** the initial string τ_0 , otherwise it **rejects** τ_0 .

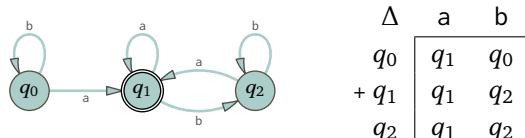
- 1.13 **EXAMPLE** The multiple of three machine of the prior example gives the computation $\langle q_0, 5013 \rangle \vdash \langle q_2, 013 \rangle \vdash \langle q_2, 13 \rangle \vdash \langle q_0, 3 \rangle \vdash \langle q_0, \varepsilon \rangle$. Since q_0 is accepting, the machine accepts 5013.

- 1.14 **DEFINITION** The set of strings accepted by a Finite State machine \mathcal{M} is the **language of that machine**, $\mathcal{L}(\mathcal{M})$, or the language **recognized** or **decided**, or **accepted**, by the machine.[#]

- 1.15 **EXAMPLE** The language of the power switch machine from Example 1.1 is the set of strings consisting of the token toggle given an even number of times, $\mathcal{L} = \{ \text{toggle}^{2k} \mid k \in \mathbb{N} \}$. The language of the turnstile machine, Example 1.2 is the set of strings from the two-token alphabet $\Sigma = \{ \text{token}, \text{push} \}$ that match the diagram (a non-visual description of the set of accepted strings is awkward; later in this chapter we will see regular expressions, which describe such strings clearly and concisely).

- 1.16 **DEFINITION** For any Finite State machine, the **extended transition function** $\hat{\Delta}: \Sigma^* \rightarrow Q$ gives the state in which the machine ends after starting in the start state and consuming the given string.

- 1.17 **EXAMPLE** Consider this machine and its transition function.



[†]As earlier, read \vdash aloud as “yields.” [‡]Read \vdash^* as “yields eventually.” [#]For Finite State machines, deciding a language is equivalent to recognizing it, because the machine must halt. ‘Recognized’ is more the common term.

Its extended transition function $\hat{\Delta}$ extends Δ in that it repeats the first row of Δ 's table.

$$\hat{\Delta}(a) = q_1 \quad \hat{\Delta}(b) = q_0$$

(We disregard the difference between Δ 's input of characters and $\hat{\Delta}$'s input of length one strings.) This is $\hat{\Delta}$ on the length two strings.

$$\hat{\Delta}(aa) = q_1 \quad \hat{\Delta}(ab) = q_2 \quad \hat{\Delta}(ba) = q_1 \quad \hat{\Delta}(bb) = q_0$$

Observe that a string $\sigma \in \{a, b\}^*$ is accepted by the machine if and only if $\hat{\Delta}(\sigma)$ is a final state. For instance, the string aa is in the language of this machine as $\hat{\Delta}(aa) = q_1$, which is accepting.

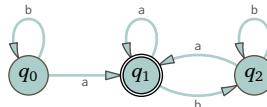
Here is a constructive definition of $\hat{\Delta}$. Fix a Finite State machine \mathcal{M} with transition function $\Delta: Q \times \Sigma \rightarrow Q$. Begin with $\hat{\Delta}(\epsilon) = \{q_0\}$. Then for $\tau \in \Sigma^*$ and $t \in \Sigma$, define $\hat{\Delta}(\tau^+ t) = \Delta(q, t)$ where $\hat{\Delta}(\tau) = q$.

This brings us back to determinism because $\hat{\Delta}$ would not be well-defined without it; by determinism Δ has one next state for all input configurations and so, by induction, for all input strings $\hat{\Delta}$ has one and only one output state.

IV.1 Exercises

A useful practice, for the exercises that describe a language, is to think through that description by naming five strings that are in the language and five that are not.

- ✓ 1.18 Using this machine, trace through the computation when the input is
(A) abba (B) bab (C) bbaabbaaa.



1.19 True or false: because a Finite State machine is finite, its language must be finite.

1.20 Rebut “no Finite State machine can recognize the language $\{a^n b \mid n \in \mathbb{N}\}$ because n is infinite.”

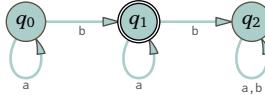
1.21 Your classmate says, “I have a language \mathcal{L} that recognizes the empty string ϵ .” Explain to them the mistake.

- ✓ 1.22 How many transitions does an input string of length n cause a Finite State machine to undergo? n ? $n + 1$? $n - 1$? How many (not necessarily distinct) states will the machine have visited after consuming the string?
- ✓ 1.23 For each of these descriptions of a language, give a one or two sentence informal English-language description. Also list five strings that are elements as well as five that are not, if there are that many.
 - (A) $\mathcal{L} = \{\alpha \in \{a, b\}^* \mid \alpha = a^n b a^n \text{ for } n \in \mathbb{N}\}$
 - (B) $\{\beta \in \{a, b\}^* \mid \beta = a^n b a^m \text{ for } n, m \in \mathbb{N}\}$

- (c) $\{ba^n \in \{a,b\}^* \mid n \in \mathbb{N}\}$
 (d) $\{a^nba^{n+2} \in \{a,b\}^* \mid n \in \mathbb{N}\}$
 (e) $\{\gamma \in \{a,b\}^* \mid \gamma \text{ has the form } \gamma = \alpha^\wedge \alpha \text{ for } \alpha \in \{a,b\}^*\}$
- ✓ 1.24 For the machines of Example 1.6, Example 1.8, Example 1.9, and Example 1.10, answer these. (A) What are the accepting states? (B) Does it recognize the empty string ϵ ? (C) What is the shortest string that each accepts? (D) Is the language of accepted strings infinite?
- 1.25 As in Example 1.13, give the computation for the multiple of three machine with the initial string 2332.
- 1.26 Modify the machine of Example 1.10 so that it accepts only non-empty strings.
- 1.27 Produce the transition graph picturing this transition function. What is the machine's language?

Δ	a	b
q_0	q_2	q_1
$+ q_1$	q_0	q_2
q_2	q_2	q_2

- ✓ 1.28 What language is recognized by this machine?



- ✓ 1.29 For each language produce a Finite State machine that recognizes that language. Give both a circle diagram and a transition function table. The alphabet is $\Sigma = \{a, b\}$.
- (A) $\mathcal{L}_1 = \{\sigma \in \Sigma^* \mid \sigma \text{ has at least one } a \text{ and at least one } b\}$
 (B) $\mathcal{L}_2 = \{\sigma \in \Sigma^* \mid \sigma \text{ has fewer than three } a's\}$
 (C) $\mathcal{L}_3 = \{\sigma \in \Sigma^* \mid \sigma \text{ ends in } ab\}$
 (D) $\mathcal{L}_4 = \{a^n b^m \in \Sigma^* \mid n, m \geq 2\}$
 (E) $\mathcal{L}_5 = \{a^n b^m a^p \in \Sigma^* \mid m = 2 \text{ and } a, p \in \mathbb{N}\}$
- 1.30 Consider the language of strings over $\Sigma = \{a, b\}$ containing at least two a's and at least two b's. Name five elements of the language and five non-elements, if there are that many. Then produce a Finite State machine recognizing this language. As in Example 1.6, briefly describe the intuitive meaning of each state.
- ✓ 1.31 For each language, name five strings in the language and five that are not (if there are not five, name as many as there are). Then give a transition graph and table for a Finite State machine recognizing the language. Use $\Sigma = \{a, b\}$.
- (A) $\{\sigma \in \Sigma^* \mid \sigma \text{ has at least two } a's\}$
 (B) $\{\sigma \in \Sigma^* \mid \sigma \text{ has exactly two } a's\}$
 (C) $\{\sigma \in \Sigma^* \mid \sigma \text{ has less than three } a's\}$

- (D) $\{\sigma \in \Sigma^* \mid \sigma \text{ has at least one } a \text{ followed by at least one } b\}$
- ✓ 1.32 Give a Finite State machine over $\Sigma = \{a, b, c\}$ that accepts any string containing the substring abc. Give a brief explication of each state's role in the machine, as in Example 1.6.
- 1.33 For each language, give five strings from that language and five that are not (if there are not that many then list all of the strings that are possible). Also give a Finite State machine that recognizes the language. Use $\Sigma = \{a, b\}$.
- (A) $\mathcal{L} = \{\sigma \in \{a, b\}^* \mid \sigma \text{ ends in } aa\}$
 - (B) $\{\sigma \in \{a, b\}^* \mid \sigma = \epsilon\}$
 - (C) $\{\sigma \in \{a, b\}^* \mid \sigma = a^3b \text{ or } \sigma = ba^3\}$
 - (D) $\{\sigma \in \{a, b\}^* \mid \sigma = a^n \text{ or } \sigma = b^n \text{ for } n \in \mathbb{N}\}$
- 1.34 Produce a Finite State machine over the alphabet $\Sigma = \{A, \dots, Z, 0, \dots, 9\}$ that accepts only the string 911, and a machine that accepts any string but that one.
- 1.35 Using Example 1.17, apply the extended transition function to all of the length three and length four string inputs.
- 1.36 What happens when the input to an extended transition function is the empty string?
- ✓ 1.37 Consider a language of comments, which begin with the two-character string /#, end with #/, and have no #/ substrings in the middle. Give a Finite State machine to recognize that language. (Just producing the transition graph is enough.)
- ✓ 1.38 Produce a Finite State machine that recognizes each.
- (A) $\{\sigma \in \{0, \dots, 9\}^* \mid \sigma \text{ has either no } 0\text{'s or no } 2\text{'s}\}$
 - (B) $\{\sigma \in \{0, \dots, 9\}^* \mid \sigma \text{ is the decimal representation of a multiple of } 5\}$
- ✓ 1.39 Give a Finite State machine over the alphabet $\Sigma = \{A, \dots, Z\}$ that accepts only strings in which the vowels occur in ascending order. (The traditional vowels, in ascending order, are A, E, I, O, and U.)
- ✓ 1.40 Consider this grammar.
- $$\begin{aligned} \langle \text{real} \rangle &\rightarrow \langle \text{posreal} \rangle \mid + \langle \text{posreal} \rangle \mid - \langle \text{posreal} \rangle \\ \langle \text{posreal} \rangle &\rightarrow \langle \text{natural} \rangle \mid \langle \text{natural} \rangle . \mid \langle \text{natural} \rangle . \langle \text{natural} \rangle \\ \langle \text{natural} \rangle &\rightarrow \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{natural} \rangle \\ \langle \text{digit} \rangle &\rightarrow 0 \mid \dots 9 \end{aligned}$$
- (A) Give five strings that are in its language and five that are not. (B) Is the string .12 in the language? (C) Briefly describe the language. (D) Give a Finite State machine that recognizes the language.
- 1.41 Produce a Finite State machine for each.
- (A) $\{\sigma \in \mathbb{B}^* \mid \text{every } 1 \text{ in } \sigma \text{ has a } 0 \text{ just before it and just after}\}$
 - (B) $\{\sigma \in \mathbb{B}^* \mid \sigma \text{ represents in binary a number divisible by } 4\}$
 - (C) $\{\sigma \in \{0, \dots, 9\}^* \mid \sigma \text{ represents in decimal an even number}\}$

- (D) $\{\sigma \in \{0, \dots 9\}^* \mid \sigma \text{ represents in decimal a multiple of } 100\}$
- 1.42 Consider $\{\sigma \in \{0, \dots 9\}^* \mid \sigma \text{ represents in decimal a multiple of } 4\}$. Briefly describe a Finite State machine. You need not give the full graph or table.
- 1.43 We will apply the constructive definition of the extended transition function that follows Definition 1.16 to the machine in Example 1.6. (A) Use the definition to find $\hat{\Delta}(0)$ and $\hat{\Delta}(1)$. (B) Use the definition to find $\hat{\Delta}$'s output on inputs 00, 01, 10, and 11. (C) Find its action on all length three strings.
- ✓ 1.44 Produce a Finite State machine that recognizes the language over $\Sigma = \{a, b\}$ containing no more than one occurrence of the substring aa. That is, it may contain zero-many such substrings or one, but not two. Note that the string aaa contains two occurrences of that substring.
- 1.45 Let $\Sigma = \mathbb{B}$. (A) List all of the different Finite State machines over Σ with a single state, $Q = \{q_0\}$. (Ignore whether a state is final or not; we will do that below.) (B) List all the ones with two states, $Q = \{q_0, q_1\}$. (C) How many machines are there with n states? (D) What if we distinguish between machines with different sets of final states?
- ✓ 1.46 *Propositiones ad acuendos iuvenes* (problems to sharpen the young) is the oldest collection of mathematical problems in Latin. It is by Alcuin of York (735–804), royal advisor to Charlemagne and head of the Frankish court school. One problem, *Propositio de lupo et capra et fasciculo cauli*, is particularly famous: *A man had to transport to the far side of a river a wolf, a goat, and a bundle of cabbages. The only boat he could find was one that could carry only two of them. For that reason, he sought a plan which would enable them all to get to the far side unhurt. Let him, who is able, say how it could be possible to transport them safely.* A wolf cannot be left alone with a goat nor can a goat be left alone with cabbages. Construct the relevant Finite State machine and use it to solve the problem.
- 1.47 Show that for any finite language there is a Finite State machine recognizing that language.
- 1.48 There are languages not recognized by any Finite State machine. Fix an alphabet Σ with at least two members. (A) Show that the number of Finite State machines with that alphabet is infinite. (B) Show that it is countable. (C) Show that the number of languages over that alphabet is uncountable.

SECTION

IV.2 Nondeterminism

Turing machines and Finite State machines both have the property that, given the current state and current character, the next state is completely determined. Once you lay out an initial tape and push Start then the machine just walks through the steps. We now consider machines that are nondeterministic: for which there may be configurations where it could move to more than one next state, or

configurations where there is just one, or even configurations without a next state at all.

Motivation Imagine a grammar with some rules and a start symbol. You are given a string and asked to find a derivation. The challenge is that you sometimes don't know which rules the derivation should follow. For instance, if you have $S \rightarrow \text{BaS} \mid \text{AbA}$ then from S you can do two different things; which will work?

In that section's exercises we expected that an intelligent person would have the insight to guess the right way. If instead you were writing a program then you might have it try every case—you might do a breadth-first traversal of the tree of all derivations—until you find a success.



Yogi Berra
1925–2015

The American philosopher and Hall of Fame baseball catcher Y Berra said, "When you come to a fork in the road, take it." That's a natural way to attack this problem: when you come up against multiple possibilities, fork a child for each. Thus, the routine might begin with the start state S and for each rule that could apply, it spawns a child process, deriving a string one removed from the start. After that, each child finds each rule that could apply to its string and spawns its own children, each of which now has a string that is two removed from the start. Continue until the desired string appears, if it ever does.

The prototypical example is the celebrated **Traveling Salesman** problem, that of finding the shortest circuit visiting every city in a list. For instance, suppose that we want to know if there is a trip that visits each state capital in the US lower forty eight states and returns back to where it began in less than, say, 16 000 kilometers. We start at Montpelier, the capital of Vermont. From there we could fork a process for each potential next capital, making forty seven new processes. Thus the process that is assigned Concord, New Hampshire would know that the trip so far is 188 kilometers. In the next round, each child would fork its own child processes, forty six of them. At the end, many processes will have failed to find a short-enough trip but if even one succeeds then we consider the overall search a success.

That computation description is nondeterministic in that while it is happening the machine is simultaneously in many different states. It happens on an unboundedly-parallel machine, where whenever we need an additional computing agent, another CPU, one is available.[†] Such a machine is angelic in that if we just ask for more resources, then we shall receive.

We will have two ways to think about nondeterminism, two mental models.[‡] The first is the one introduced above: when a machine is presented with multiple possible next states then it forks, so that it is in all of them simultaneously. The next example illustrates.

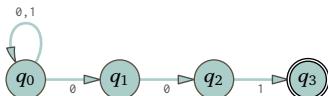


Persian angel,
1555

[†]This is like our experience with everyday computers, where we may be writing an email in one window and watching a video in another. The machine appears to be in multiple states simultaneously.

[‡]While these models are helpful in learning and thinking about nondeterminism, they are not part of the formal definitions and proofs.

- 2.1 EXAMPLE The Finite State machine below is nondeterministic because leaving q_0 are two arrows labeled 0. It also has states with a deficit of edges such as that no arrow for 1 leaves q_1 so if it is in that state and reads that input then it passes to no state at all.



The graphic below shows what happens with input 00001. It pictures the computation history as a tree. For instance, on the first 0 the computation splits in two so the machine is now in two states at once.

2.2 ANIMATION: Steps in the nondeterministic computation.

When we considered the forking approach to string derivations or to the Traveling Salesman problem, we observed that if a solution exists then there exists a child process that would find it. The same happens here; there is a branch of the computation tree that accepts the input string. There are also branches that are not successful. The one at the bottom dies after step 2 because when the present state q_2 and the input is 0 this machine passes to no-state.[†] The branch at the top does not die early but it also does not accept the input. However, we don't care about unsuccessful branches, we only care that there is a successful one. Consequently, we will define that a nondeterministic machine accepts an input if the computation tree has at least one branch that accepts the input.

The machine in the above example accepts any string that ends in two 0's and a 1. Having been fed the input $\sigma = 00001$, the problem that the machine faces is: when it should stop going around q_0 's loop and start to the right? This machine accepts this input, so it has solved this problem—viewed from the outside at least, we could say that the machine has correctly guessed.

This is our second model for nondeterminism. We can imagine programming by calling a function, some `amb(...)`, that guesses a successful sequence if there is

[†]No-state cannot be an accepting state, since it isn't a state at all.

one to guess.

Saying that a mechanism guesses is jarring. Based on programming classes, a person's intuition may well be that "guessing" is not mechanically accomplishable. As an alternative, we can imagine that the machine is furnished with the answer ("go around twice, then off to the right") and only has to check it. This way of expressing the second mental model is demonic because the furnisher seems to be a supernatural being who somehow knows answers that cannot otherwise be found, but we must be suspicious and check that the answer is not a trick. Under this model, a nondeterministic computation accepts the input if there exists a branch of the computation tree that a deterministic machine, if told what branch to take, could verify.



Flauros, Duke of Hell

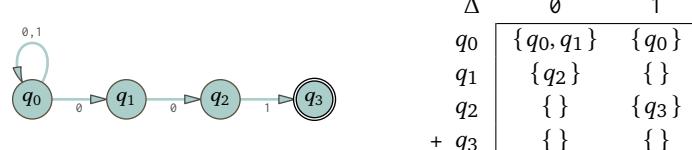
Below we shall describe nondeterminism using both paradigms: as a machine being in multiple states at once, and also as a machine guessing. In this chapter we will do that for Finite State machines and in the fifth chapter we will return to it for Turing machines.

Definition A nondeterministic Finite State machine's next-state function does not output single states, it outputs sets of states.

2.3 **DEFINITION** A **nondeterministic Finite State machine** $\langle Q, q_{\text{start}}, F, \Sigma, \Delta \rangle$ consists of a finite **set of states** Q , one of which is the **start state** q_{start} , a subset $F \subseteq Q$ of **accepting states** or **final states**, a finite **input alphabet** set Σ , and a **next-state function** $\Delta: Q \times \Sigma \rightarrow \mathcal{P}(Q)$.

We will use these machines in three ways. First, they are useful in practice; both below and in the exercises are examples of jobs that are more easily solved with nondeterministic machines than with deterministic ones. Second, we will use them to prove Kleene's Theorem, Theorem 3.11. Finally, they give us an initial encounter with nondeterminism, which is a critical concept for this book's fifth chapter.

2.4 **EXAMPLE** This is Example 2.1's nondeterministic Finite State machine, along with its transition function. The outputs of Δ are not states, they are sets of states.



The above introduction uses informal terms such as "guess" and "demon." The imagery helps us to understand the ideas but may also give a sense that those ideas are fuzzy. Consequently, going through the formalities is a help; in particular, we can absolutely give a precise mathematical definition.

A **configuration** is a pair $C = \langle q, \tau \rangle$ where $q \in Q$ and $\tau \in \Sigma^*$. A machine starts in an **initial configuration** $C_0 = \langle q_0, \tau_0 \rangle$, so that τ_0 is the **input**. Following C_0 , there

may be sequences of \vdash transitions. This machine is nondeterministic so there may be many such sequences, or one, or none. Each such sequence is a branch of the computation tree.

Consider a branch that includes $C_s = \langle q, \tau_s \rangle$. If τ_s is the empty string then C_s is a halting configuration and no transitions follow.

Otherwise, τ_s is not empty. We will describe when some $\hat{C} = \langle \hat{q}, \hat{\tau} \rangle$ is an allowed subsequent configuration, that is, under what circumstances $C_s \vdash \hat{C}$. If there is such a transition then the relationship between the strings in the two configurations is that $\hat{\tau}$ comes from removing τ_s 's leading character $c \in \Sigma$ (τ_s must have one because it is not the empty string), so that $\tau_s = c \cap \hat{\tau}$. As for the next state \hat{q} , it can be any member of the set $\Delta(q, c)$. Because this machine is nondeterministic, it could be that the set $\Delta(q, c)$ is empty. In this case there is no transition to a \hat{C} , and the branch containing C_s dead-ends without yielding any halting configuration.

With that, a nondeterministic Finite State machine computation is a set containing all of the branches, the sequences of transitions, that are described in the prior paragraphs. If any branch in that set ends in a halting configuration $C_0 = \langle q_0, \tau_0 \rangle \vdash C_1 \vdash C_2 \vdash \dots \vdash C_h = \langle q, \varepsilon \rangle$ such that $q \in F$ then the machine accepts τ_0 . Otherwise, it rejects τ_0 .

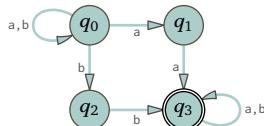
- 2.5 EXAMPLE Example 2.1 shows a number of sequences of allowed transitions. Since at least one of them contains a configuration where the string is empty and the state is accepting, the machine accepts the intial string 00001.

$$\langle q_0, 00001 \rangle \vdash \langle q_0, 0001 \rangle \vdash \langle q_0, 001 \rangle \vdash \langle q_1, 01 \rangle \vdash \langle q_2, 1 \rangle \vdash \langle q_3, \varepsilon \rangle$$

- 2.6 DEFINITION For a nondeterministic Finite State machine \mathcal{M} , the set of accepted strings is the language of the machine $\mathcal{L}(\mathcal{M})$, or the language recognized or accepted, by that machine.[†]

We will also adapt the definition of the extended transition function $\hat{\Delta}: \Sigma^* \rightarrow Q$. Fix a nondeterministic \mathcal{M} with transition function $\Delta: Q \times \Sigma \rightarrow Q$. Start with $\hat{\Delta}(\varepsilon) = \{q_0\}$. Where $\hat{\Delta}(\tau) = \{q_{i_0}, q_{i_1}, \dots, q_{i_k}\}$, for $\tau \in \Sigma^*$ define $\hat{\Delta}(\tau \cap t)$ to be $\Delta(q_{i_0}, t) \cup \Delta(q_{i_1}, t) \cup \dots \cup \Delta(q_{i_k}, t)$ for any $t \in \Sigma$. Then the machine accepts $\sigma \in \Sigma^*$ if and only if any element of $\hat{\Delta}(\sigma)$ is a final state.

- 2.7 EXAMPLE The language recognized by this nondeterministic machine



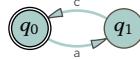
is the set of strings containing the substring aa or bb. For instance, the machine accepts abaaba because there is a sequence of allowed transitions ending in an

[†] Below we will define something called ε transitions that make ‘recognized’ the right idea here, rather than ‘decided’.

accepting state, namely this one.

$$\langle q_0, abaaba \rangle \vdash \langle q_0, baaba \rangle \vdash \langle q_0, aaba \rangle \vdash \langle q_1, aba \rangle \vdash \langle q_2, ba \rangle \vdash \langle q_2, a \rangle \vdash \langle q_2, \varepsilon \rangle$$

- 2.8 EXAMPLE With $\Sigma = \{ a, b, c \}$, this nondeterministic machine



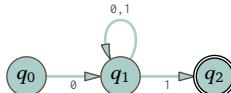
recognizes the language $\{ (ac)^n \mid n \in \mathbb{N} \} = \{ \varepsilon, ac, acac, \dots \}$. The symbol b isn't attached to any arrow so it won't play a part in any accepting string.

Often a nondeterministic Finite State machines is easier to write than a deterministic machine that does the same job.

- 2.9 EXAMPLE These machines accept any string whose next to last character is a. The nondeterministic one on the left is simpler than the deterministic one.



- 2.10 EXAMPLE This machine accepts $\{ \sigma \in \mathbb{B}^* \mid \sigma = 0^\wedge \tau^\wedge 1 \text{ where } \tau \in \mathbb{B}^* \}$.



- 2.11 EXAMPLE This is a remote control listener that waits to hear the signal 0101110. That is, it recognizes the language $\{ \sigma^\wedge 0101110 \mid \sigma \in \mathbb{B}^* \}$.



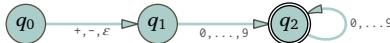
- 2.12 REMARK Having seen a couple of examples we pause to again acknowledge that something about nondeterminism is perplexing. If we feed $\tau = 010101110$ to the prior example's machine then it accepts. But it is set up for a string that begins with two sets of 01's, while τ begins with three. How can a mechanism guess that it should ignore the first 01 but act on the second? In mathematics we can consider whatever we can make precise, but we have so far stuck to devices that are in principle physically realizable. So this may seem to be a shift.

It is not. After doing one more machine variation, this section closes by showing how to convert any nondeterministic Finite State machine into a deterministic one that does the same job. So we can take a nondeterministic Finite State machine to be an abbreviation, a shorthand, a convenience, a way of alternatively specifying a deterministic machine. This obviates at least some of the paradox of guessing.

ε transitions Another extension, beyond nondeterminism, is to allow **ε transitions** or **ε moves**. We alter the definition of a nondeterministic Finite State machine so that instead of $\Delta: Q \times \Sigma \rightarrow \mathcal{P}(Q)$ the transition function's signature

is $\Delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$.[†] The associated behavior is that the machine can transition spontaneously, without consuming any input.

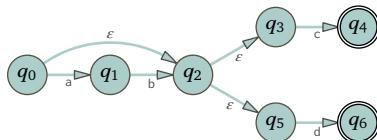
- 2.13 EXAMPLE This machine recognizes valid integer representations. The ε between q_0 and q_1 means that the machine can jump from the state q_0 to q_1 without consuming any input.



The practical effect of the ε is that this machine can accept strings that do not start with a + or - sign. For instance, with input 123 the machine can begin as below by following the ε transition to state q_1 , then read the 1 and transition to q_2 , and stay there while processing the 2 and 3. This is a branch of the computation tree accepting the input and so the string 123 is in the machine's language.

$$\langle q_0, 123 \rangle \vdash \langle q_1, 123 \rangle \vdash \langle q_2, 23 \rangle \vdash \langle q_2, 3 \rangle \vdash \langle q_2, \varepsilon \rangle$$

- 2.14 EXAMPLE A machine may follow two or more ε transitions. From q_0 this machine may stay in that state, or transition to q_2 , or q_3 , or q_5 , all without consuming any input.

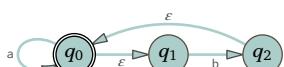


These two sequences show that the machine accepts abc, and also d.

$$\begin{aligned} \langle q_0, abc \rangle &\vdash \langle q_1, bc \rangle \vdash \langle q_2, c \rangle \vdash \langle q_3, c \rangle \vdash \langle q_4, \varepsilon \rangle \\ \langle q_0, d \rangle &\vdash \langle q_5, d \rangle \vdash \langle q_6, \varepsilon \rangle \end{aligned}$$

This machine's language is $\mathcal{L} = \{abc, abd, c, d\}$.

- 2.15 EXAMPLE For this machine this is the computation tree on input aab.



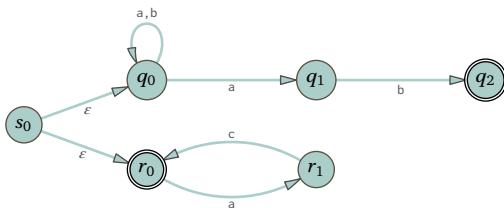
- 2.16 ANIMATION: Computation tree for a nondeterministic machine with ε moves.

[†]The 'ε' is a character, not a representation of the empty string. Assume that it is not an element of Σ .

The ε moves are in the white stripes. At each step, think of the machine as being in all of the states inside of the white stripe. So at step 0 the machine is in both q_0 and q_1 . At step 3 it is in both q_0 and q_2 . That exhausts the tape and q_0 is accepting so the machine accepts its input.

One reason to consider ε transitions is that they can simplify building Finite State machines.

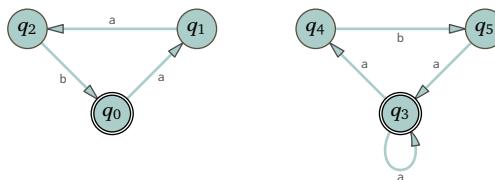
- 2.17 EXAMPLE An ε transition can put two machines together with a parallel connection. This shows a machine whose states are named with q 's combined with one whose states are named with r 's.



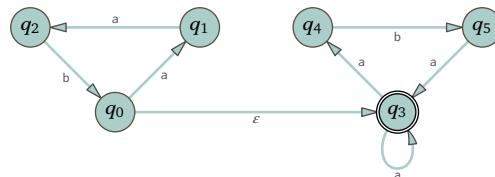
The top nondeterministic machine's language is $\{ \sigma \in \Sigma^* \mid \sigma \text{ ends in } ab \}$ and the bottom machine's language is $\{ \sigma \in \Sigma^* \mid \sigma = (ac)^n \text{ for some } n \in \mathbb{N} \}$, where $\Sigma = \{a, b, c\}$. The language for the entire machine is the union.

$$\mathcal{L} = \{ \sigma \in \Sigma^* \mid \text{either } \sigma \text{ ends in } ab \text{ or } \sigma = (ac)^n \text{ for } n \in \mathbb{N} \}$$

- 2.18 EXAMPLE An ε transition can also make a serial connection between machines. The machine on the left below recognizes the language of repetitions of the string aab, $\mathcal{L}_0 = \{ (aab)^i \mid i \in \mathbb{N} \}$. The machine on the right recognizes repetitions of either a or aba, $\mathcal{L}_1 = \{ \sigma_0 \cdots \sigma_{j-1} \mid j \in \mathbb{N} \text{ and } \sigma_k = a \text{ or } \sigma_k = aba \text{ for } 0 \leq k \leq j-1 \}$.

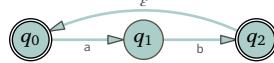


If we insert an ε bridge to the right side's initial state from each of the left side's final states (here there happens to be only one such final state), and de-finalize those states on the left,



then the combined machine accepts strings in the concatenation of those languages, $\mathcal{L}(\mathcal{M}) = \mathcal{L}_0 \cap \mathcal{L}_1$. For example, it accepts aabaababa, and aabaa, as well as abaa.

- 2.19 EXAMPLE Without the ϵ edge this machine's language is $\mathcal{L} = \{\epsilon, ab\}$, while with it the language is $\mathcal{L}^* = \{(ab)^n \mid n \in \mathbb{N}\}$.



To describe the action of these machines we first need a definition. The ϵ closure function $\hat{E}: Q \rightarrow \mathcal{P}(Q)$ inputs a state q and returns the set of states that are reachable from q without consuming any input, that is, via ϵ moves alone.

- 2.20 EXAMPLE We can compute a state's ϵ closure in steps. This uses Example 2.14's machine.

$m = 0$	1	2	3	$\hat{E}(q)$
$\{q_0\}$	$\{q_0, q_2\}$	$\{q_0, q_2, q_3, q_5\}$	$\{q_0, q_2, q_3, q_5\}$	$\{q_0, q_2, q_3, q_5\}$
$\{q_1\}$	$\{q_1\}$	$\{q_1\}$	$\{q_1\}$	$\{q_1\}$
$\{q_2\}$	$\{q_2, q_3, q_5\}$	$\{q_2, q_3, q_5\}$	$\{q_2, q_3, q_5\}$	$\{q_2, q_3, q_5\}$
$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$
$\{q_4\}$	$\{q_4\}$	$\{q_4\}$	$\{q_4\}$	$\{q_4\}$
$\{q_5\}$	$\{q_5\}$	$\{q_5\}$	$\{q_5\}$	$\{q_5\}$

The initial column's set contains only the row's state. To move to column 1, add the states reachable with one ϵ move from the initial state. Thus for the top row, that machine has an ϵ move from q_0 to q_2 so the $m = 1$ set adds q_2 to the $m = 0$ set. To move to column 2, add states reachable with an ϵ move from any element of column 1's set (which are thus at most two ϵ moves away from the initial state). Doing the same for the third column does not change the sets in any row, so we are done.

In words, construct \hat{E} by first defining $E: Q \times \mathbb{N} \rightarrow \mathcal{P}(Q)$, as in the above table. Start with $E(q, 0) = \{q\}$. For $m > 0$, where $E(q, m) = \{q_{m_0}, \dots, q_{m_k}\}$, let $E(q, m+1) = E(q, m) \cup \Delta(q_{m_0}, \epsilon) \cup \dots \cup \Delta(q_{m_k}, \epsilon)$. The resulting sets are nested, $E(q, 0) = \{q\} \subseteq E(q, 1) \subseteq \dots$. There are only finitely many states in Q so for any initial q there must be an $m_q \in \mathbb{N}$ where the sequence of sets stops growing, $E(q, m_q) = E(q, m_q + 1) = \dots$ (in the example above, for $q = q_0$ this happens at $m = 2$). Then the ϵ closure is the limit, $\hat{E}(q) = E(q, m_q)$.

With that, we can describe the action. A **configuration** is a pair $C = \langle q, \tau \rangle \in Q \times \Sigma^*$. These machines start in an **initial configuration** $C_0 = \langle q_0, \tau_0 \rangle$, with **initial state** q_0 and **input** τ_0 . As earlier, we will specify when two configurations are related by ' \vdash ' and then we will say that a machine accepts its input if there is at least one sequence of related configurations that ends in a halting configuration whose state is accepting.

Consider a configuration $C_s = \langle q, \tau_s \rangle$, in order to describe under what circumstances $C_s \vdash \hat{C}$ for some next $\hat{C} = \langle \hat{q}, \hat{\tau} \rangle$. There are two possibilities. The first is the same as for any nondeterministic machine: τ is not empty and $\hat{\tau}$ comes

from popping τ 's leading character, $\tau = c \hat{\wedge} \hat{\tau}$ where $c \in \Sigma$, and \hat{q} is a member of $\Delta(q, c)$. The second possibility differs from the description for machines without ϵ moves: $\hat{\tau} = \tau$ and \hat{q} is a member of the ϵ closure $\hat{E}(q)$ with $\hat{q} \neq q$ (the significance of $\hat{\tau} = \tau$ is that the machine doesn't consume any input characters).

With that, a **computation of a nondeterministic Finite State machine with ϵ transitions** is the set containing all of the sequences of transitions that are as described. A sequence ends in a **halting configuration** if its final pair has an empty tape string, so that the sequence has the form $C_0 = \langle q_0, \tau_0 \rangle \vdash C_1 \vdash \dots \vdash C_h = \langle q_h, \epsilon \rangle$. The machine **accepts** τ_0 if there is at least one such sequence where q_h is an element of F . If there is no such sequence then the machine **rejects** τ_0 .

In addition, we can define the **extended transition function** $\hat{\Delta}: Q \times \Sigma^* \rightarrow Q$ to suit this new machine type. Begin by defining $\hat{\Delta}(q, \epsilon) = \hat{E}(q)$ for all $q \in Q$. The inductive step is that for $\tau \in \Sigma^*$ and $c \in \Sigma$, where $\hat{\Delta}(q, \tau) = \{q_{i_0}, q_{i_1}, \dots, q_{i_k}\}$ we have $\hat{\Delta}(q, \tau \hat{\wedge} c) = \hat{E}(\hat{\Delta}(q, \tau), c) \cup \dots \cup \hat{E}(\hat{\Delta}(q, \tau), c)$.

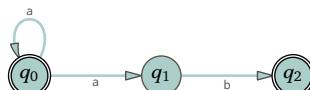
Equivalence of the machine types We next prove that nondeterminism does not change what Finite State machines can do.

- 2.21 **THEOREM** The class of languages recognized by nondeterministic Finite State machines equals the class of languages recognized by deterministic Finite State machines. This remains true if we allow the nondeterministic machines to have ϵ transitions.

Inclusion in one direction is easy because any deterministic machine is, essentially, a nondeterministic machine. In a deterministic machine the next-state function outputs single states and to make it a nondeterministic machine, just convert those states into singleton sets. Thus the set of languages recognized by deterministic machines is a subset of the set recognized by nondeterministic machines.

We will demonstrate inclusion in the other direction constructively, starting with nondeterministic machines and building deterministic machines that recognize the same language. The two examples below show the **powerset construction**. Our complete description of the algorithm comes after the first example. We won't give a proof that this construction works simply because the examples are entirely convincing.

- 2.22 **EXAMPLE** This nondeterministic machine \mathcal{M}_N has no ϵ transitions.



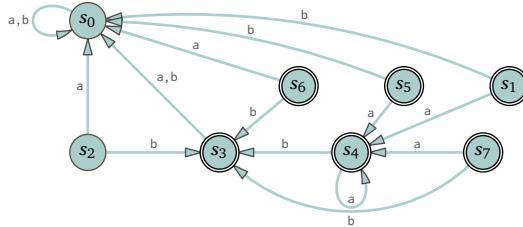
The associated deterministic machine \mathcal{M}_D is shown below. Each member of \mathcal{M}_D is a set of \mathcal{M}_N 's states, $s_i = \{q_{i_1}, \dots, q_{i_k}\}$. See below. The start state of \mathcal{M}_D is $s_1 = \{q_0\}$, and a state of \mathcal{M}_D is accepting if any of its elements are accepting states in \mathcal{M}_N .

As an illustration of constructing \mathcal{M}_D 's transition table, suppose that \mathcal{M}_N is in

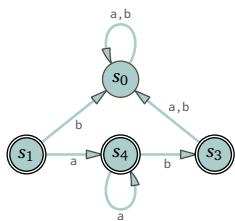
$s_5 = \{q_0, q_2\}$ and its read head is pointing to a. The next state combines the next states due to q_0 with those due to q_2 . Thus in total, $\Delta_D(s_5, a) = \{q_0, q_1\}$, which in the table is s_4 .

Δ_D	a	b
$s_0 = \{ \}$	s_0	s_0
$+ s_1 = \{q_0\}$	s_4	s_0
$s_2 = \{q_1\}$	s_0	s_3
$+ s_3 = \{q_2\}$	s_0	s_0
$+ s_4 = \{q_0, q_1\}$	s_4	s_3
$+ s_5 = \{q_0, q_2\}$	s_4	s_0
$+ s_6 = \{q_1, q_2\}$	s_0	s_3
$+ s_7 = \{q_0, q_1, q_2\}$	s_4	s_3

Besides the notational convenience, naming the sets of states as s_i 's makes clear that \mathcal{M}_D is deterministic. So does its transition graph. Again, the start state is s_1 .



Many of those states are unreachable; here is the pared down picture.

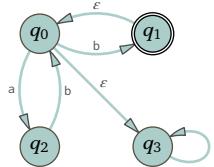


Now we give the algorithm for starting with a nondeterministic machine \mathcal{M}_N and constructing a deterministic machine \mathcal{M}_D with the same behavior, whether or not \mathcal{M}_N has ϵ transitions. Elements of \mathcal{M}_D are sets of states from \mathcal{M}_N . Define the ϵ closure of a set of states to be the union of the ϵ closures of its members.

The transition function Δ_D inputs a state $s_i \in \mathcal{M}_D$, that is, $s_i = \{q_{k_0}, \dots, q_{k_i}\}$, along with a tape character $c \in \Sigma$. First apply \mathcal{M}_N 's next state function to s_i 's elements to get $\hat{s}_{i,c} = \Delta_N(q_{k_0}, c) \cup \dots \cup \Delta_N(q_{k_i}, c)$. Second, where $\hat{s}_{i,c} = \{q_{j_0}, \dots, q_{j_i}\}$, finish by taking the ϵ closure of all of \hat{s} 's elements, $\Delta_D(s_i, c) = \hat{E}(q_{j_0}) \cup \dots \cup \hat{E}(q_{j_i})$. (For machines without ϵ transitions this second part has no effect.)

The start state of \mathcal{M}_D is the ϵ closure of q_0 (for machines without ϵ moves this is $\{q_0\}$). A state of \mathcal{M}_D is accepting if it contains any of \mathcal{M}_N 's accepting states.

2.23 EXAMPLE This nondeterministic machine has ε transitions.



The table below goes through the computation of the associated deterministic machine. The start state is $\hat{E}(q_0) = \{q_0, q_3\} = s_7$. A state is accepting if it contains q_1 .

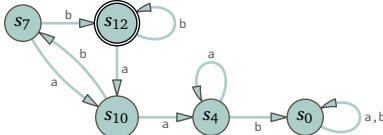
For example, when the machine is in state $s_1 = \{q_0\}$ and the head reads b then the above diagram shows that the machine goes to $\{q_1\}$. In the terms used in the powerset algorithm description, $\hat{s}_{0,b} = \{q_1\}$. The ε closure is $\hat{E}(\{q_1\}) = \{q_0, q_1, q_3\}$, because from q_1 the machine can reach those three states without consuming any input. Thus $\Delta_D(s_1, b) = \{q_0, q_1, q_3\} = s_{12}$.

Another example is the computation of $\Delta_D(s_8, a)$. Since $s_8 = \{q_1, q_2\}$, and in the machine above neither member has a leaving arrow labeled 'a', we get that $\hat{s}_{s_8,a}$ is the empty set. The ε closure of the empty set is the empty set, $\{\} = s_0$. Thus $\Delta_D(s_8, a) = s_0$.

	$\hat{s}_{i,a}$	$\Delta_D(s_i, a)$	$\hat{s}_{i,b}$	$\Delta_D(s_i, b)$
$s_0 = \{\}$	$\{\}$	$\{\} = s_0$	$\{\}$	$\{\} = s_0$
$s_1 = \{q_0\}$	$\{q_2\}$	$\{q_2\} = s_3$	$\{q_1\}$	$\{q_0, q_1, q_3\} = s_{12}$
$+ s_2 = \{q_1\}$	$\{\}$	$\{\} = s_0$	$\{\}$	$\{\} = s_0$
$s_3 = \{q_2\}$	$\{\}$	$\{\} = s_0$	$\{q_0\}$	$\{q_0, q_3\} = s_7$
$s_4 = \{q_3\}$	$\{q_3\}$	$\{q_3\} = s_4$	$\{\}$	$\{\} = s_0$
$+ s_5 = \{q_0, q_1\}$	$\{q_2\}$	$\{q_2\} = s_3$	$\{q_1\}$	$\{q_0, q_1, q_3\} = s_{12}$
$s_6 = \{q_0, q_2\}$	$\{q_2\}$	$\{q_2\} = s_3$	$\{q_0, q_1\}$	$\{q_0, q_1, q_3\} = s_{12}$
$s_7 = \{q_0, q_3\}$	$\{q_2, q_3\}$	$\{q_2, q_3\} = s_{10}$	$\{q_1\}$	$\{q_0, q_1, q_3\} = s_{12}$
$+ s_8 = \{q_1, q_2\}$	$\{\}$	$\{\} = s_0$	$\{q_0\}$	$\{q_0, q_3\} = s_7$
$+ s_9 = \{q_1, q_3\}$	$\{q_3\}$	$\{q_3\} = s_4$	$\{\}$	$\{\} = s_0$
$s_{10} = \{q_2, q_3\}$	$\{q_3\}$	$\{q_3\} = s_4$	$\{q_0\}$	$\{q_0, q_3\} = s_7$
$+ s_{11} = \{q_0, q_1, q_2\}$	$\{q_2\}$	$\{q_2\} = s_3$	$\{q_0, q_1\}$	$\{q_0, q_1, q_3\} = s_{12}$
$+ s_{12} = \{q_0, q_1, q_3\}$	$\{q_2, q_3\}$	$\{q_2, q_3\} = s_{10}$	$\{q_1\}$	$\{q_0, q_1, q_3\} = s_{12}$
$s_{13} = \{q_0, q_2, q_3\}$	$\{q_2, q_3\}$	$\{q_2, q_3\} = s_{10}$	$\{q_0, q_1\}$	$\{q_0, q_1, q_3\} = s_{12}$
$+ s_{14} = \{q_1, q_2, q_3\}$	$\{q_3\}$	$\{q_3\} = s_4$	$\{q_0\}$	$\{q_0, q_3\} = s_7$
$+ s_{15} = \{q_0, q_1, q_2, q_3\}$	$\{q_2, q_3\}$	$\{q_2, q_3\} = s_{10}$	$\{q_0, q_1\}$	$\{q_0, q_1, q_3\} = s_{12}$

The machine diagram for this deterministic machine is below. Although this machine has sixteen states, many of those are unreachable from the starting state. One example is that looking through the table shows that never is s_2 the output of Δ_D . The diagram omits these nodes.

Again, the start state is s_7 .

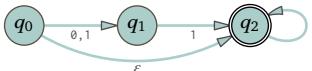


The powerset construction shows that if the nondeterministic machine has n states then there is a deterministic machine with at most 2^n states. In fact, 2^n is the best that we can do in that for any n there is an n state nondeterministic machine requiring that the deterministic machine has 2^n states. However, in practice it does often happen that the deterministic machine is not too big, once we minimize the number of states (Section 6 below shows how to minimize).

IV.2 Exercises

2.24 Give the transition function for the machine of Example 2.7, and of Example 2.8.

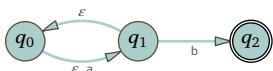
- ✓ 2.25 Consider this machine.



- (A) Does it accept the empty string? (B) The string 0? (C) 011? (D) 010? (E) List all length five accepted strings.

2.26 Your class has someone who asks, “I get that it is interesting, but isn’t all this machine-guessing stuff just mathematical abstractions that are not real?” How might the prof respond?

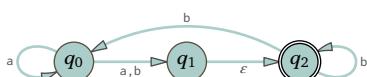
2.27 Your friend objects, “Epsilon transitions don’t make any sense because the machine below will never get its first step done; it just endlessly follows the epsilon.” Correct their misimpression.



- ✓ 2.28 With Example 2.23’s nondeterministic and deterministic machines, verify that they agree on whether to accept or reject these input strings (A) the empty string (B) a (C) b (D) aa (E) ab (F) ba (G) bb.

2.29 Give a sequence of ‘ \vdash ’ relations showing that Example 2.11’s machine accepts $\tau = 01010110$.

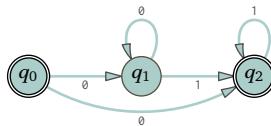
2.30 This machine has $\Sigma = \{a, b\}$.



- (A) What is the ϵ closure of q_0 ? Of q_1 ? q_2 ? (B) Does it accept the empty string? (C) a? b? (D) Show that it accepts aab by producing a suitable sequence of \vdash relations. (E) List five strings of minimal length that it accepts. (F) List five of minimal length that it does not accept.

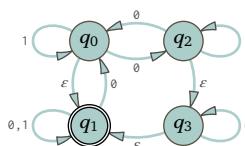
2.31 Produce the table description of the next-state function Δ for the machine in the prior exercise. It should have three columns, for a , b , and ϵ .

2.32 Consider this machine.



- (A) Show that it accepts 011 by producing a suitable sequence of \vdash relations.
- (B) Show that the machine accepts 00011 by producing a suitable sequence of \vdash relations.
- (C) Does it accept the empty string?
- (D) 0^* ? 1^* ?
- (E) List five strings of minimal length that it accepts.
- (F) List five of minimal length that it does not accept.
- (G) What is the language of this machine?

- ✓ 2.33 Find the ϵ closures of the states of this nondeterministic machine using a table like the Example 2.20's.



2.34 Draw the transition graph of a nondeterministic machine that recognizes the language $\{\sigma = \tau_0\tau_1\tau_2 \in \mathbb{B}^* \mid \tau_0 = 1, \tau_2 = 1, \text{ and } \tau_1 = (00)^k \text{ for some } k \in \mathbb{N}\}$.

- ✓ 2.35 Give diagrams for nondeterministic Finite State machines that recognize the given language and that have the given number of states. Use $\Sigma = \mathbb{B}$.

- (A) $\mathcal{L}_0 = \{\sigma \mid \sigma \text{ ends in } 00\}$, having three states
- (B) $\mathcal{L}_1 = \{\sigma \mid \sigma \text{ has the substring } 0110\}$, with five states
- (C) $\mathcal{L}_2 = \{\sigma \mid \sigma \text{ contains an even number of } 0\text{'s or exactly two } 1\text{'s}\}$, with six states
- (D) $\mathcal{L}_3 = \{0\}^*$, with one state

- ✓ 2.36 Draw the graph of a nondeterministic Finite State machine over \mathbb{B} that accepts strings with the suffix 111000111 .

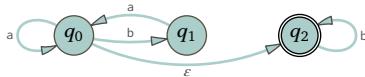
2.37 Find a nondeterministic Finite State machine that recognizes this language of three words: $\mathcal{L} = \{\text{cat}, \text{cap}, \text{carumba}\}$.

2.38 Give a nondeterministic Finite State machine over $\Sigma = \{a, b, c\}$ recognizing the language of strings that omit at least one of the characters in the alphabet.

- ✓ 2.39 For each, draw the transition graph for a Finite State machine, which may be nondeterministic, that accepts the given strings from $\{a, b\}^*$.

- (A) Accepted strings have a second character of a and next to last character of b .
- (B) Accepted strings have second character a and the next to last character is also a .

- ✓ 2.40 What is the language of this nondeterministic machine with ϵ transitions?



- 2.41 Find a deterministic machine and a nondeterministic machine that recognizes the set of bitstrings containing the substring 11. You need not derive the deterministic machine with the powerset construction.

- 2.42 This table

Δ	a	b
q_0	$\{q_0\}$	$\{q_1, q_2\}$
q_1	$\{q_3\}$	$\{q_3\}$
q_2	$\{q_1\}$	$\{q_3\}$
$+ q_3$	$\{q_3\}$	$\{q_3\}$

gives the next-state function for a nondeterministic Finite State machine. (A) Draw the transition graph. (B) What is the recognized language? (C) Give the next-state table for a deterministic machine that recognizes the same language.

- ✓ 2.43 Find the nondeterministic Finite State machine that accepts all bitstrings that begin with 10. Use the powerset construction to produce the transition function table of a deterministic machine that does the same.
- ✓ 2.44 For each, follow the construction above to make a deterministic machine with the same language.



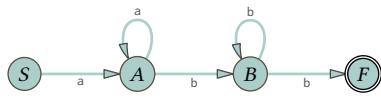
- ✓ 2.45 For each give a nondeterministic Finite State machine over $\Sigma = \{0, 1, 2\}$.
 - (A) The machine recognizes the language of strings whose final character appears exactly twice in the string.
 - (B) The machine recognizes the language of strings whose final character appears exactly twice in the string, but in between those two occurrences is no higher digit.
- ✓ 2.46 For each give a nondeterministic Finite State machine with ϵ transitions over that recognizes each language over \mathbb{B} . (Some of these illustrate that sometimes deterministic machines are harder.)
 - (A) In each string, every 0 is followed immediately by a 1.
 - (B) Each string contains 000 followed, possibly with some intermediate characters, by 001.
 - (C) In each string the first two characters equals the final two characters, in order. (Hint: what about 000?)
 - (D) There is either an even number of 0's or an odd number of 1's.

- 2.47 Give a minimal-sized nondeterministic Finite State machine over $\Sigma = \{a, b, c\}$ that accepts only the empty string. Also give one that accepts any string except the empty string. For both, produce the transition graph and table.

2.48 Use the extended transition function to give a definition of when a nondeterministic machine accepts its input that is an alternate to the definition given in the section.

2.49 A grammar is **right linear** if every production rule has the form $\langle N \rangle \rightarrow x\langle M \rangle$, where the right side has a single terminal followed by a single nonterminal. With this right linear grammar we can associate this nondeterministic Finite State machine.

$$\begin{aligned} S &\rightarrow aA \\ A &\rightarrow aA \mid bB \\ B &\rightarrow bB \mid b \end{aligned}$$



- (A) Give three strings from the language of the grammar and show that they are accepted by the machine. (B) Describe the language of the grammar and the machine.

2.50 Decide whether each problem is solvable or unsolvable by a Turing machine.

- (A) $\mathcal{L}_{DFA} = \{\langle \mathcal{M}, \sigma \rangle \mid \text{the deterministic Finite State machine } \mathcal{M} \text{ accepts } \sigma\}$
(B) $\mathcal{L}_{NFA} = \{\langle \mathcal{M}, \sigma \rangle \mid \text{the nondeterministic machine } \mathcal{M} \text{ accepts } \sigma\}$

2.51 (A) For the machine of Example 2.23, for each $q \in Q$ produce $E(q, 0)$, $E(q, 1)$, $E(q, 2)$, and $E(q, 3)$. List $\hat{E}(q)$ for each $q \in Q$.

- (B) Do the same for Exercise 2.30's machine.

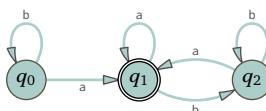
SECTION

IV.3 Regular expressions

In 1951, S Kleene[†] was studying a mathematical model of neurons. These are a kind of Finite State machine in that they do not have scratch memory. He noted patterns to the languages that they recognize. For instance, the Finite State machine below accepts strings that have some number of b's (perhaps zero many) followed by at least one a, possibly then followed by some number of repetitions of a pattern of at least one b and then at least one a. He introduced a convenient way, called regular expressions, to denote constructs such as ‘any number of’ and ‘followed by’. He gave the definition in the first subsection below and supported it with the theorem in the second subsection.



Stephen
Kleene
1909–1994



Definition A regular expression is a string that describes a language. We will first introduce these with a few examples (they use the alphabet $\Sigma = \{a, \dots, z\}$).

[†]Pronounced KLAY-nee. He was a PhD student of Church's.

- 3.1 EXAMPLE The regular expression $p(a|e|i|o|u)t$ describes the language of strings that start with p, have a vowel in the middle, and end with t. That is, this regular expression describes the language consisting of five words, $\mathcal{L} = \{pat, pet, pit, pot, put\}$.

The pipe operator, ‘|’, which is a kind of ‘or’, and the parentheses, which provide grouping, are not part of the strings being described; they are **metacharacters**.

Besides the pipe and parentheses, the regular expression in that example also describes concatenation since the language consists of strings where the initial p is concatenated with a vowel, which in turn is concatenated with t.

- 3.2 EXAMPLE The regular expression ab^*c describes the language whose words begin with an a, followed by any number of b's (including possibly zero-many b's), and ending with a c. Thus, ‘*’ means ‘repeat the prior thing any number of times, including possibly zero-many times’. This regular expression describes the language $\mathcal{L} = \{ac, abc, abbc, \dots\}$.
- 3.3 EXAMPLE There is an interaction between pipe and star. Consider the regular expression $(b|c)^*$. It could mean either ‘any number of repetitions of picking a b or c’ or ‘pick a b or c and repeat that character any number of times’.

The definition and semantics below has it mean the first. Thus the language described by $a(b|c)^*$ consists of words starting with a and ending with any mixture of b's and c's, so that it describes the language $\mathcal{L} = \{a, ab, ac, abb, abc, acb, acc, \dots\}$.

In contrast, to describe the language whose members begin with an a and end with any number of b's or any number of c's, $\hat{\mathcal{L}} = \{a, ab, abb, \dots, ac, acc, \dots\}$, use the regular expression $a(b^*|c^*)$.

The rules for operator precedence are: star binds most tightly, then concatenation, and then the pipe alternation operator, |. To get another order, use parentheses.

- 3.4 DEFINITION Let Σ be an alphabet not containing the metacharacters ‘)’, ‘(’, ‘|’, or ‘*’. A **regular expression** over Σ is a string that can be derived from the grammar

```

⟨regex⟩ → ⟨concat⟩
| ( ⟨regex⟩ ‘|’ ⟨concat⟩ )
⟨concat⟩ → ⟨simple⟩
| ( ⟨concat⟩ ⟨simple⟩ )
⟨simple⟩ → ⟨char⟩
| ( ⟨simple⟩ * )
⟨char⟩ → ∅ | ε |  $x_0$  |  $x_1$  | ...

```

where the x_i characters are members of Σ .[†]

[†]As we have done with other grammars, here we use the pipe symbol | as a metacharacter, to collapse rules with the same left side. But pipe also appears in regular expressions. For that usage the grammar wraps it in single quotes, as ‘|’.

As to their semantics, what regular expressions mean, we will define that recursively. The base is the single-character regular expressions. The language described by the regular expression \emptyset is the empty language, $\mathcal{L}(\emptyset) = \emptyset$. The language described by the regular expression consisting of only the character ε is the one-element language containing only the empty string, $\mathcal{L}(\varepsilon) = \{\varepsilon\} = \{''\}$. If the regular expression consists of a single character from the alphabet Σ then the language that it describes contains only one string and that string has only that single character, as in $\mathcal{L}(a) = \{a\}$.

We finish by doing the operations. Start with regular expressions R_0 and R_1 describing languages $\mathcal{L}(R_0)$ and $\mathcal{L}(R_1)$. Then the pipe symbol describes the union of the languages, so that $\mathcal{L}(R_0|R_1) = \mathcal{L}(R_0) \cup \mathcal{L}(R_1)$. Concatenation of the regular expressions describes concatenation of the languages, $\mathcal{L}(R_0R_1) = \mathcal{L}(R_0) \cap \mathcal{L}(R_1)$. And $\mathcal{L}(R_0^*) = \mathcal{L}(R_0)^*$, so the Kleene star of the regular expression describes the star of the language.[†]

- 3.5 EXAMPLE Consider the regular expression ab over $\Sigma = \{a, b\}$. It is the concatenation of a and b . The first describes a single-element language $\mathcal{L}(a) = \{a\}$ and likewise the second describes $\mathcal{L}(b) = \{b\}$. Thus, the string ab describes the concatenation of the two, another language $\mathcal{L}(ab) = \mathcal{L}(a) \cap \mathcal{L}(b) = \{ab\}$ with only one element.
- 3.6 EXAMPLE Now consider aba^* . The regular expression a^* describes the star of the language $\mathcal{L}(a)$, namely $\mathcal{L}(a^*) = \{a^n \mid n \in \mathbb{N}\} = \{\varepsilon, a, aa, \dots\}$. Concatenating $\mathcal{L}(ab)$ with $\mathcal{L}(a^*)$ gives this.

$$\begin{aligned}\mathcal{L}(aba^*) &= \{\sigma \in \Sigma^* \mid \sigma = \sigma_0 \cap \sigma_1 \text{ where } \sigma_0 \in \mathcal{L}(ab) \text{ and } \sigma_1 \in \mathcal{L}(a^*)\} \\ &= \{ab, aba, abaa, aba^3, \dots\} = \{aba^n \mid n \in \mathbb{N}\}\end{aligned}$$

We finish this subsection with some constructs that appear often. These examples use $\Sigma = \{a, b, c\}$. Also see Extra A for extensions of these constructs that are widely used in practice.

- 3.7 EXAMPLE The language consisting of strings of a 's whose length is a multiple of three, $\mathcal{L} = \{a^{3k} \mid k \in \mathbb{N}\} = \{\varepsilon, aaa, aaaaa, \dots\}$, is described by $(aaa)^*$.

Note that the empty string is a member of that language. A common mistake is to forget that star includes zero-many repetitions.

- 3.8 EXAMPLE To match any character we can list them all. The language over $\Sigma = \{a, b, c\}$ of three-letter words ending in bc is $\{abc, bbc, cbc\}$. The regular expression $(a|b|c)bc$ describes it. Another regular expression that describes this language is $(abc) | (bbc) | (cbc)$.

[†]We usually omit parentheses unless they are required. Thus here we wrote $\mathcal{L}(R_0|R_1)$ without parentheses, that is, not as $\mathcal{L}((R_0|R_1))$. We did the same with $\mathcal{L}(R_0R_1)$ and $\mathcal{L}(R_0^*)$. But there are expressions where the parentheses matter: in general, $(R_0|R_1)^*$ represents a different language than $R_0|(R_1^*)$.

- 3.9 EXAMPLE The empty string character ϵ is handy to mark things as optional. Thus $a^*(\epsilon|b)$ describes the language of strings that have any number of a's and optionally end in one b, $\mathcal{L} = \{\epsilon, b, a, ab, aa, aab, \dots\}$. Similarly, to describe the language consisting of words with between three and five a's, $\mathcal{L} = \{aaa, aaaa, aaaaa\}$, we can use $aaa(\epsilon|a|aa)$.
- 3.10 EXAMPLE The language $\{b, bc, bcc, ab, abc, abcc, aab, \dots\}$ has words starting with any number of a's (including zero-many a's), followed by a single b, and then ending in fewer than three c's. To describe it we can use $a^*b(\epsilon|c|cc)$.

Kleene's Theorem The next result justifies our study of regular expressions because it shows that they describe the languages of interest.

- 3.11 **THEOREM (KLEENE'S THEOREM)** A language is recognized by a Finite State machine if and only if that language is described by a regular expression.

We will prove this in separate halves. The proofs use nondeterministic machines but since we can convert those to deterministic machines, the result holds for them also.

- 3.12 **LEMMA** If a language is described by a regular expression then there is a Finite State machine recognizing that language.

Proof We will show that for any regular expression R there is a machine accepting exactly the strings matching that expression. We use induction on the structure of regular expressions.

Start with regular expressions consisting of a single character. If $R = \emptyset$ then $\mathcal{L}(R) = \{\}$ and the machine on the left below recognizes this language. If $R = \epsilon$ then $\mathcal{L}(R) = \{\epsilon\}$ and the machine in the middle recognizes it. If the regular expression is a character from the alphabet such as $R = a$ then the machine on the right works.



We finish by handling the three operations. Let R_0 and R_1 be regular expressions. The inductive hypothesis gives a machine \mathcal{M}_0 whose language is described by R_0 and a machine \mathcal{M}_1 whose language is described by R_1 .

First consider alternation, $R = R_0 | R_1$. Create the machine recognizing the language described by R by joining those two machines in parallel: introduce a new state s and use ϵ transitions to connect s to the start states of \mathcal{M}_0 and \mathcal{M}_1 . See Example 2.17 in the prior section.

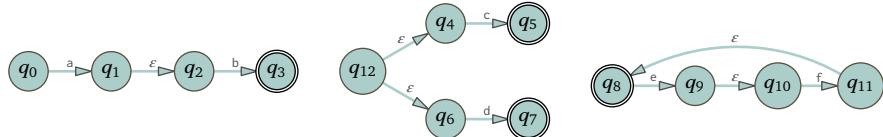
Next consider concatenation, $R = R_0 \cdot R_1$. Join the two machines serially: for each accepting state in \mathcal{M}_0 , make an ϵ transition to the start state of \mathcal{M}_1 and then convert all of the accepting states of \mathcal{M}_0 to be non-accepting states. See Example 2.18.

Finally consider Kleene star, $R = (R_0)^*$. For each accepting state in the machine \mathcal{M}_0 that is not the start state, make an ϵ transition to the start state and then make the start state an accepting state. See Example 2.19. \square

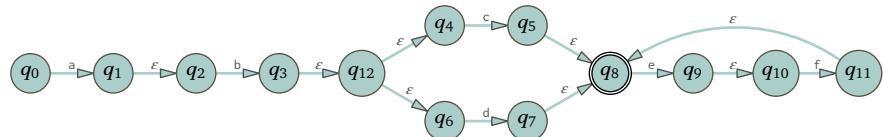
- 3.13 EXAMPLE Building a machine for the regular expression $ab(c|d)(ef)^*$ starts with machines for the single characters.



Put these atomic components together



to get the complete machine.



This is a nondeterministic machine with ϵ transitions. If you want a deterministic machine then use the powerset construction.

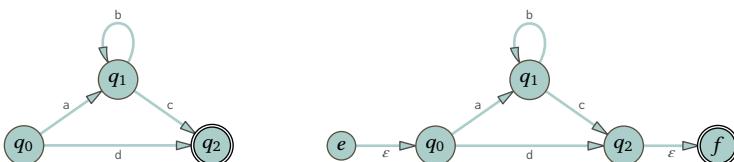
- 3.14 LEMMA Any language recognized by a Finite State machine is described by a regular expression.

Our strategy starts with a Finite State machine and eliminates its states one at a time. Below is an illustration, before and after pictures of part of a larger machine, where we eliminate the state q .



In the after picture the edge is labeled ab , with more than just one character. For this proof we will generalize transition graphs to allow edge labels that are regular expressions. As we eliminate states, we keep the recognized language of the machines the same. We will be done when what remains is two states, with one edge between them. The desired regular expression will be that edge's label.

Before the proof, one more illustration. Start with the machine on the left.



The proof goes as on the right, by introducing a new start state, e , and a new final state, f . Then the proof eliminates q_1 as below.



Clearly this machine recognizes the same language as the starting one.

Proof Call the machine \mathcal{M} . If it has no accepting states then the regular expression is \emptyset and we are done. Otherwise, we start by transforming \mathcal{M} to a new machine, $\hat{\mathcal{M}}$, that has the same language and that is ready for the state-elimination strategy.

First we arrange that $\hat{\mathcal{M}}$ has a single accepting state. Create a new state f and for each of \mathcal{M} 's accepting states make an ϵ transition to f (by the prior paragraph there is at least one such accepting state so f is connected to the rest of $\hat{\mathcal{M}}$). Change all the accepting states to non-accepting ones and then make f accepting. Clearly this does not change the language of accepted strings.

Next introduce a new start state, e . Make a ϵ transition from it to q_0 , again leaving the language of the machine unchanged. (Putting e in $\hat{\mathcal{M}}$ allows us to uniformly eliminate each state in \mathcal{M} when we say below, “Pick any q not equal to e or f .”)

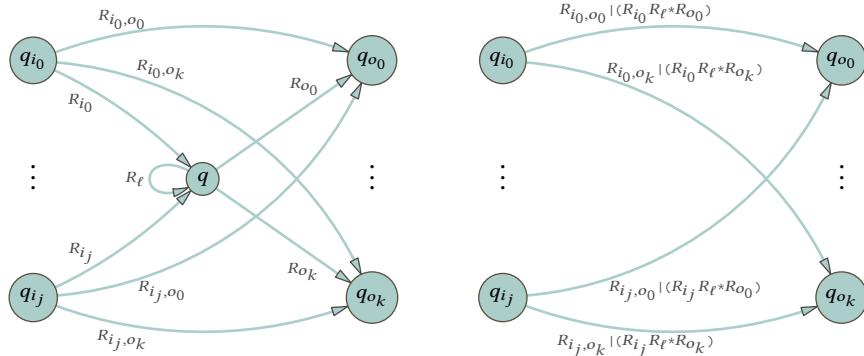
Because the edge labels are regular expressions, we can arrange that from any q_i to any q_j there is at most one edge, since if \mathcal{M} has more than one edge then in $\hat{\mathcal{M}}$ we can use the pipe, $|$, to combine the labels.



Do the same with loops, that is, cases where $i = j$. These adjustments do not change the language of accepted strings.

The last part of transforming to $\hat{\mathcal{M}}$ is to drop useless states. If a state node other than f has no outgoing edges then omit it, along with the edges into it. The language of the machine will not change because this state is not itself accepting as only f is accepting, and cannot lead to an accepting state since it doesn't lead anywhere. Along the same lines, if a state node is not reachable from the start e then drop that node along with its incoming and outgoing edges. (The idea of useless states is clear but it has some technical aspects. Omitting a no-outgoing-edges node, along with its incoming edges, can result in another node now having no outgoing edges, which in turn needs the same treatment. But these machines have only finitely many nodes and so this omitting process must run out eventually. For a full definition of unreachability see Exercise 3.35.)

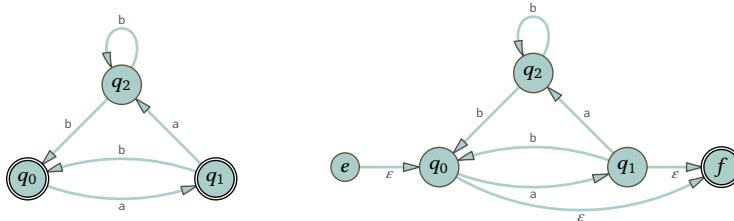
With that, $\hat{\mathcal{M}}$ is ready for state elimination. Pick any q not equal to e or f . Below are before and after pictures. There are states q_{i_0}, \dots, q_{i_j} with an edge leading into q , and states q_{o_0}, \dots, q_{o_k} that receive an edge leading out of q . (By the setup work above, q has at least one incoming and at least one outgoing edge.) In addition, q may have a loop. (Here is a fine point of the diagrams: in each, possibly some of the states shown on the left equal some shown on the right. For example, possibly q_{i_0} equals q_{o_0} , in which case the edge shown as R_{i_0, o_0} is a loop.)



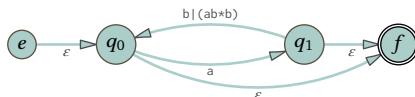
Eliminate q and the associated edges by making the replacements shown in the after picture. Observe that the set of strings taking the machine from any incoming state q_i to any outgoing state q_o is unchanged and consequently the language recognized by the machine is unchanged.

Each iteration of this elimination procedure reduces the number of states by one. Keep going until the only states remaining are e and f . Since the language of the machines never changes, the desired regular expression is the one labeling the edge between them. \square

3.15 EXAMPLE Consider \mathcal{M} on the left below. Introduce e and f to get $\hat{\mathcal{M}}$ on the right.



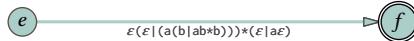
We start by eliminating q_2 . (We could have instead done q_0 or q_1 first, but just picked this one to start somewhere.) In the notation used in the proof's key step, $q_1 = q_{i_0}$ and $q_0 = q_{o_0}$. The regular expressions are $R_{i_0} = a$, $R_{o_0} = b$, $R_{i_0,o_0} = b$, and $R_\ell = b$. That gives this machine.



Next eliminate q_1 . (Again, we could have instead done q_0 next.) There is one incoming node $q_0 = q_{i_0}$ and there are two outgoing nodes $q_0 = q_{o_0}$ and $f = q_{o_1}$. Note that q_0 is both an incoming and outgoing node; this is the “fine point” mentioned in the proof. The regular expressions are $R_{i_0} = a$, $R_{o_0} = b | (ab^*b)$, and $R_{o_1} = \epsilon$.



Final step. The sole incoming node is $e = q_{i_0}$ and the sole outgoing node is $f = q_{o_0}$, and so $R_{i_0} = \epsilon$, $R_{o_0} = \epsilon | a\epsilon$, and $R_\ell = \epsilon | a(b|ab^*b)$.



This regular expression describes the language of the starting machine (it simplifies; for instance, $a\epsilon=a$ so the final parenthesis becomes $(\epsilon | a)$).

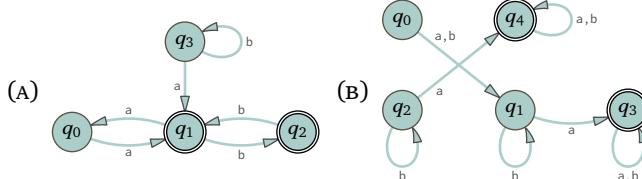
IV.3 Exercises

- 3.16 Decide if the string σ matches the regular expression R . (A) $\sigma = 0010$, $R = 0*10$ (B) $\sigma = 101$, $R = 1*01$ (C) $\sigma = 101$, $R = 1*(0|1)$ (D) $\sigma = 101$, $R = 1*(0|1)*$ (E) $\sigma = 01$, $R = 1*01*$
- ✓ 3.17 For each regular expression produce five bitstrings that match and five that do not, or as many as there are if there are not five. (A) $01*$ (B) $(01)^*$ (C) $1(0|1)1$ (D) $(0|1)(\epsilon|1)0*$ (E) \emptyset
- 3.18 Give a brief plain English description of the language for each regular expression. (A) $a\epsilon cb^*$ (B) aa^* (C) $a(a|b)*bb$
- ✓ 3.19 For these regular expressions and for each element of $\{a,b\}^*$ that is of length less than or equal to 3, decide if it is a match. (A) a^*b (B) a^* (C) \emptyset (D) ϵ (E) $b(a|b)a$ (F) $(a|b)(\epsilon|a)a$
- 3.20 For these regular expressions, decide if each element of \mathbb{B}^* of length at most 3 is a match. (A) $0*1$ (B) $1*0$ (C) \emptyset (D) ϵ (E) $0(0|1)^*$ (F) $(100)(\epsilon|1)0^*$
- ✓ 3.21 A friend says to you, “The point of parentheses is that you first do inside the parentheses and then do what’s outside. So Kleene star means ‘match the inside and repeat’, and the regular expression $(0*1)^*$ matches the strings 001001 and 010101 but not 01001 and 00000101 , where the substrings are unequal.” Straighten them out.
- 3.22 Produce a regular expression for the language of bitstrings with a substring consisting of at least three consecutive 1's.
- 3.23 Someone who sits behind you in class says, “I don’t get it. I got a regular expression that I am sure is right. But the book got a different one.” Explain what is up.
- 3.24 For each language, give five strings that are in the language and five that are not. Then give a regular expression describing the language. Finally, give a Finite State machine that accepts the language. (A) $\mathcal{L}_0 = \{a^n b^{2m} \mid m, n \geq 1\}$ (B) $\mathcal{L}_0 = \{a^n b^{3m} \mid m, n \geq 1\}$

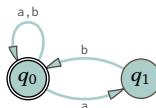
- 3.25 Give a regular expression for the language over $\Sigma = \{ a, b, c \}$ whose strings are missing at least one letter, that is, the strings that are either without any a's, or without any b's, or without any c's.
- 3.26 Give a regular expression for each language. Use $\Sigma = \{ a, b \}^*$. (A) The set of strings starting with b. (B) The set of strings whose second-to-last character is a. (C) The set of strings containing at least one of each character. (D) The strings where the number of a's is divisible by three.
- 3.27 Give a regular expression to describe each language over the alphabet $\Sigma = \{ a, b, c \}$. (A) The set of strings starting with aba. (B) The set of strings ending with aba. (C) The set of strings containing the substring aba.
- ✓ 3.28 Give a regular expression to describe each language over \mathbb{B} . (A) The set of strings of odd parity, where the number of 1's is odd. (B) The set of strings where no two adjacent characters are equal. (C) The set of strings representing in binary multiples of eight.
- ✓ 3.29 Give a regular expression to describe each language over the alphabet $\Sigma = \{ a, b \}$. (A) Every a is both immediately preceded and immediately followed by a b character. (B) Each string has at least two b's that are not followed by an a.
- 3.30 Give a regular expression for each language of bitstrings. (A) The number of 0's is even. (B) There are more than two 1's. (C) The number of 0's is even and there are more than two 1's.
- 3.31 Give a regular expression to describe each language.
- (A) $\{ \sigma \in \{ a, b \}^* \mid \sigma \text{ ends with the same symbol it began with, and } \sigma \neq \epsilon \}$
 - (B) $\{ a^i b a^j \mid i \text{ and } j \text{ leave the same remainder on division by three} \}$
- ✓ 3.32 Give a regular expression for each language over \mathbb{B}^* .
- (A) The strings representing a binary number that is a multiple of eight.
 - (B) The bitstrings where the first character differs from the final one.
 - (C) The bitstrings where no two adjacent characters are equal.
- ✓ 3.33 Produce a Finite State machine whose language equals the language described by each regular expression. (A) a^*ba (B) $ab^*(a|b)^*$
- 3.34 Fix a Finite State machine \mathcal{M} . Kleene's Theorem shows that the set of strings taking \mathcal{M} from the start state to the set of final states is regular.
- (A) Show that for any set of states $S \subseteq Q_{\mathcal{M}}$ the set of strings taking \mathcal{M} from the start state to one of the states in S is regular.
 - (B) Show that the set of strings taking \mathcal{M} from any single state to any other single state is regular.
- 3.35 Part of the proof of Lemma 3.14 involves unreachable states. Here is a definition. Given a state q , construct the set of states reachable from it by first setting $S_0 = \{ q \} \cup \hat{E}(q)$, where $\hat{E}(q)$ is the ϵ closure. Then iterate: starting with the set S_i of states that are reachable in i -many steps, for each $\tilde{q} \in S_i$ follow each outbound edge for a single step and also include the elements of the ϵ closure.

The union of S_i with the collection of the states reached in this way is the set S_{i+1} . Stop when $S_i = S_{i+1}$, at which point it is the set of ever-reachable states. The unreachable states are the others.

For each machine use that definition to find the set of unreachable states.

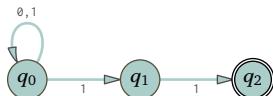


- ✓ 3.36 Show that the set of languages over Σ that are described by a regular expression is countable. Conclude that there are languages not recognized by any Finite State machine.
- 3.37 For the grammar of Definition 3.4, construct the parse tree for these regular expressions over $\Sigma = \{a, b\}$. (A) $a(b|c)$ (B) $ab^*(a|c)$
- 3.38 Using Definition 3.4, parse Example 3.3's $a(b|c)^*$ and $a(b^*|c^*)$.
- ✓ 3.39 Get a regular expression by applying the method of Lemma 3.14's proof to this machine.



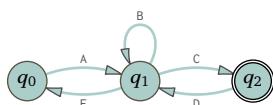
- (A) Get $\hat{\mathcal{M}}$ by introducing e and f . (B) Where $q = q_0$, describe which state from the machine is playing the diagram's before picture role of q_{i_0} , which edge is R_{i_0} , etc. (c) Eliminate q_0 .

- 3.40 Apply method of Lemma 3.14's proof to this machine. At each step describe which state from the machine is playing the role of q_{i_0} , which edge is R_{i_0} , etc.



- (A) Eliminate q_0 . (B) Eliminate q_1 . (C) q_2 (D) Give the regular expression.

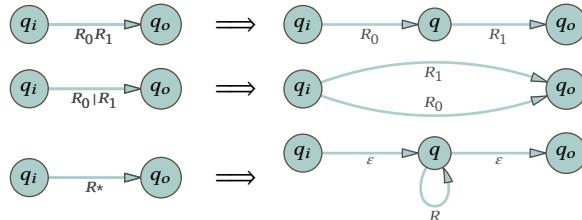
- 3.41 Apply the state elimination method of Lemma 3.14's proof to eliminate q_1 . Note that each of the states q_0 and q_2 are of the kind described in the proof's comment on the subtle point.



- 3.42 An alternative proof of Lemma 3.12 reverses the steps of Lemma 3.14. This is the **subset method**. Start by labeling the single edge on a two-state machine with the given regular expression.



Then instead of eliminating nodes, introduce them.



Use this approach to get a machine that recognizes the language described by these regular expressions. (A) $a|b$ (B) ca^* (C) $(a|b)c^*$ (D) $(a|b)(b^*|a^*)$

SECTION

IV.4 Regular languages

We have seen that deterministic Finite State machines, nondeterministic Finite State machines, and regular expressions all describe the same set of languages. The fact that we can describe these languages in so many different ways says that there is something natural and important about them.[†]

Definition We will isolate and study the languages in this collection.

- 4.1 **DEFINITION** A **regular language** is one that is recognized by some Finite State machine or equivalently, described by a regular expression.
- 4.2 **LEMMA** Fix an alphabet. The set of regular languages over it is countably infinite. The collection of all languages over that alphabet is uncountable, and consequently there are languages that are not regular.

Proof Let the alphabet be Σ . Appendix A specifies that any alphabet is nonempty and finite. This implies that there are infinitely many regular languages over that alphabet because where x is a character from Σ , each of these languages is finite and therefore regular: $\mathcal{L}_0 = \{\}$, $\mathcal{L}_1 = \{x\}$, $\mathcal{L}_2 = \{xx\}$, ...

Next we argue that the number of regular languages is countable. This holds because there is a language for each regular expression over Σ and the number of regular expressions is countable: there are finitely many regular expressions of length 1, of length 2, etc. The union of those is a countable union of countable sets, and so is countable.

We finish by showing that the set of all languages over Σ , the set of all $\mathcal{L} \subseteq \Sigma^*$, is uncountable. By the same argument as in the prior two paragraphs, Σ^* is countably

[†]This is just like how the fact that Turing machines, general recursive functions, and many other models all compute the same sets says that these computable sets are a natural and important collection. This collection is not just a historical artifact of what happened to be first investigated.

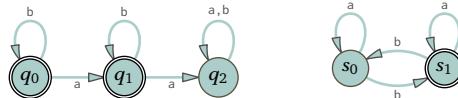
infinite (there are countably many $\sigma \in \Sigma^*$ of length zero, of length one, etc.). In contrast, the set of all $\mathcal{L} \subseteq \Sigma^*$ is the power set of Σ^* and so has cardinality greater than the cardinality of Σ^* , which makes it uncountable. \square

Closure properties In proving Lemma 3.12, the first half of Kleene's Theorem, we showed that if \mathcal{L}_0 and \mathcal{L}_1 are regular then their union $\mathcal{L}_0 \cup \mathcal{L}_1$ is regular, as is their concatenation $\mathcal{L}_0 \cdot \mathcal{L}_1$ and the Kleene star \mathcal{L}_0^* .[†] A set is said to be **closed** under an operation if performing that operation on its members always yields another member. This restates Lemma 3.12 using that term.

- 4.3 **LEMMA** The collection of regular languages is closed under the union of two languages, the concatenation of two languages, and the Kleene star of a language.

We can ask about the closure of regular languages under other operations. To answer we will use the **product construction**.

- 4.4 **EXAMPLE** The machine on the left, \mathcal{M}_0 , accepts strings with fewer than two a's. The one on the right, \mathcal{M}_1 , accepts strings with an odd number of b's.



The transition tables contain the same information as the pictures.

Δ_0	a	b	Δ_1	a	b
+ q_0	q_1	q_0	s_0	s_0	s_1
+ q_1	q_2	q_1	s_1	s_1	s_0
q_2	q_2	q_2			

This machine \mathcal{M} has states that are the members of the cross product $Q_0 \times Q_1$ and transitions that are given by $\Delta((q_i, r_j)) = (\Delta_0(q_i), \Delta_1(r_j))$. It starts in (q_0, r_0) .

Δ	a	b
(q_0, s_0)	(q_1, s_0)	(q_0, s_1)
(q_0, s_1)	(q_1, s_1)	(q_0, s_0)
(q_1, s_0)	(q_2, s_0)	(q_1, s_1)
(q_1, s_1)	(q_2, s_1)	(q_1, s_0)
(q_2, s_0)	(q_2, s_0)	(q_2, s_1)
(q_2, s_1)	(q_2, s_1)	(q_2, s_0)

If we feed the string aba to \mathcal{M} then the machine's states go from (q_0, s_0) to (q_1, s_0) , to (q_1, s_1) , and then to (q_2, s_1) . This is simply because \mathcal{M}_0 passes from q_0 to q_1 , to q_1 again, and then to q_2 , while \mathcal{M}_1 does s_0 to s_0 , to s_1 , and to s_1 . That is, we can view that \mathcal{M} runs \mathcal{M}_0 and \mathcal{M}_1 in parallel.

[†]In that proof the languages have the same alphabet, the alphabet of the regular expressions. But we have already noted that, for instance, even if two languages have different alphabets Σ_0 and Σ_1 then nonetheless the two languages and their union are all over the alphabet $\Sigma_0 \cup \Sigma_1$.

We have not yet specified which states are accepting. We can vary the language accepted by a product machine by varying the set of accepting states. One possibility is to stipulate, as on the left below, that the accepting states (q_i, s_j) are the ones where both q_i and s_j are accepting. That means \mathcal{M} accepts a string σ if and only if both \mathcal{M}_0 and \mathcal{M}_1 accept it.

	a	b		a	b
(q_0, s_0)	(q_1, s_0)	(q_0, s_1)	$+ (q_0, s_0)$	(q_1, s_0)	(q_0, s_1)
$+ (q_0, s_1)$	(q_1, s_1)	(q_0, s_0)	(q_0, s_1)	(q_1, s_1)	(q_0, s_0)
(q_1, s_0)	(q_2, s_0)	(q_1, s_1)	$+ (q_1, s_0)$	(q_2, s_0)	(q_1, s_1)
$+ (q_1, s_1)$	(q_2, s_1)	(q_1, s_0)	(q_1, s_1)	(q_2, s_1)	(q_1, s_0)
(q_2, s_0)	(q_2, s_0)	(q_2, s_1)	(q_2, s_0)	(q_2, s_0)	(q_2, s_1)
(q_2, s_1)	(q_2, s_1)	(q_2, s_0)	(q_2, s_1)	(q_2, s_1)	(q_2, s_0)

Another possibility is to stipulate, as on the right, that the accepting states (q_i, s_j) are the ones where q_i is accepting but s_j is not. That means the machine accepts strings that are in the language of \mathcal{M}_0 but not that of \mathcal{M}_1 .

4.5 **THEOREM** The collection of regular languages is closed under the intersection of two languages, the set difference of two languages, and the set complement of a language.

Proof Start with two Finite State machines, \mathcal{M}_0 and \mathcal{M}_1 , which accept languages \mathcal{L}_0 and \mathcal{L}_1 over some alphabet Σ . Perform the product construction to get \mathcal{M} . If the accepting states of \mathcal{M} are those pairs where both the first and second component states are accepting, in \mathcal{M}_0 and \mathcal{M}_1 , then \mathcal{M} accepts the intersection of the languages, $\mathcal{L}_0 \cap \mathcal{L}_1$. If the accepting states of \mathcal{M} are those pairs where the first component state is accepting but the second is not, then \mathcal{M} accepts the set difference of the languages, $\mathcal{L}_0 - \mathcal{L}_1$. A special case of this second one is when \mathcal{L}_0 is the set of all strings, Σ^* , and in this case \mathcal{M} accepts the complement, \mathcal{L}_1^c . \square

These closure properties often make it easier to show that a language is regular.

4.6 **EXAMPLE** To show that the language

$$\mathcal{L} = \{ \sigma \in \mathbb{B}^* \mid \sigma \text{ has an even number of } 0\text{'s and more than two } 1\text{'s} \}$$

is regular, we could produce a machine that recognizes it or exhibit a regular expression. Or, we could instead note that \mathcal{L} is the intersection of the two languages $\{ \sigma \in \mathbb{B}^* \mid \sigma \text{ has an even number of } 0\text{'s} \}$ and $\{ \sigma \in \mathbb{B}^* \mid \sigma \text{ has more than two } 1\text{'s} \}$, and producing machines or regular expressions for those two is easy.

IV.4 Exercises

- ✓ 4.7 True or false? You must justify each answer.
 - (A) Every regular language is finite.
 - (B) Over \mathbb{B} , the empty language is not regular.
 - (C) The intersection of two regular languages is regular.

- (D) Over \mathbb{B} , the language of all strings, \mathbb{B}^* , is not regular.
- (E) Every Finite State machine accepts at least one string.
- (F) For every Finite State machine there is one that has fewer states but that recognizes the same language.

4.8 Is $\{\sigma \in \mathbb{B}^* \mid \sigma \text{ represents in binary a power of 2}\}$ a regular language?

4.9 Is English a regular language?

4.10 One of these is true and one is false. Which is which? (A) Any finite language is regular. (B) Any regular language is finite.

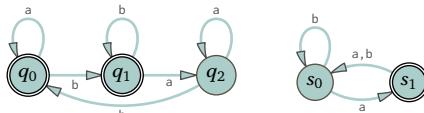
✓ 4.11 Show that each language over $\Sigma = \{a, b\}$ is regular.

- (A) $\{\sigma \in \Sigma^* \mid \sigma \text{ starts and ends with } a\}$
- (B) $\{\sigma \in \Sigma^* \mid \text{the number of } a\text{'s is even}\}$

✓ 4.12 Suppose that the language \mathcal{L} over \mathbb{B} is regular. Show that the language $\hat{\mathcal{L}} = \{1^\sigma \mid \sigma \in \mathcal{L}\}$, also over \mathbb{B} , is also regular.

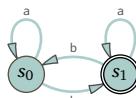
4.13 If two machines have n_0 states and n_1 states, then how many states does their product have?

✓ 4.14 For these two,



give the transition table for the cross product. Specify the accepting states so that the result will accept (A) the intersection of the languages of the two machines, and (B) the union of the languages.

4.15 Find the cross product of this machine, \mathcal{M}_1 from Example 4.4, with itself.



Set the accepting states so that it accepts the same language, \mathcal{L}_1 .

4.16 One of our first examples of Finite State machines, Example 1.6, accepts a string when it contains at least two 0's as well as an even number of 1's. Make such a machine as a product of two simpler machines.

4.17 For each, decide whether it is true or false, with justification.

- (A) Every language is the subset of a regular language.
- (B) The union of a regular language and a language that is not regular must be not regular.
- (C) Every language has a subset that is not regular.
- (D) The union of two regular languages is regular, without exception.

4.18 Fill in the blank (with justification): The concatenation of a regular language with a not-regular language _____ regular. (A) must be (B) might be, or might be not (C) cannot be

4.19 True or false? Justify your answer.

- (A) If \mathcal{L}_0 is a regular language and $\mathcal{L}_1 \subseteq \mathcal{L}_0$ then \mathcal{L}_1 is also a regular language.
- (B) If \mathcal{L}_0 is not regular and $\mathcal{L}_0 \subseteq \mathcal{L}_1$ then \mathcal{L}_1 is also not regular.
- (C) If $\mathcal{L}_0 \cap \mathcal{L}_1$ is regular then each of the two is regular.

4.20 Where \mathcal{L} is a language, define \mathcal{L}^+ as the language $\mathcal{L} \cap \mathcal{L}^*$. Show that if \mathcal{L} is regular then so is \mathcal{L}^+ .

4.21 Use closure properties to show that if \mathcal{L} is regular then the set of even-length strings in \mathcal{L} is also regular.

4.22 Example 4.6 shows that closure properties can make easier some arguments that a language is regular. It can do the same for arguments that a language is not regular. The next section shows that $\{a^n b^n \in \{a, b\}^* \mid n \in \mathbb{N}\}$ is not regular (this is a restatement of Example 5.2). Use that and closure properties to show that $\{\sigma \in \{a, b\}^* \mid \sigma \text{ contains the same number of } a\text{'s as } b\text{'s}\}$ is not regular.
Hint: one way is to use closure under intersection.

4.23 Prove that the collection of regular languages over Σ is closed under each of the operations.

- (A) $\text{pref}(\mathcal{L})$ contains those strings that are a prefix of some string in the language, that is, $\text{pref}(\mathcal{L}) = \{\sigma \in \Sigma^* \mid \text{there is a } \tau \in \Sigma^* \text{ such that } \sigma \hat{\cdot} \tau \in \mathcal{L}\}$
- (B) $\text{suff}(\mathcal{L})$ contains the strings that are a suffix of some string in the language, that is, $\text{suff}(\mathcal{L}) = \{\sigma \in \Sigma^* \mid \text{there is a } \tau \in \Sigma^* \text{ such that } \tau \sigma \in \mathcal{L}\}$
- (C) $\text{allprefs}(\mathcal{L})$ contains the strings such that all of the prefixes are in the language, that is, $\text{allprefs}(\mathcal{L}) = \{\sigma \in \mathcal{L} \mid \text{for all } \tau \in \Sigma^* \text{ that is a prefix of } \sigma, \tau \in \mathcal{L}\}$

4.24 Lemma 4.2 gives a counting argument, a pure existence proof, that there are languages that are not regular. But we can also exhibit one. Prove that $\mathcal{L} = \{1^k \mid k \in K\}$ is not regular, where K is the Halting problem set, $K = \{e \in \mathbb{N} \mid \phi_e(e) \downarrow\}$.

4.25 Show each.

- (A) The collection of regular languages is not closed under the union of infinitely many sets.
- (B) Nor is it closed under the intersection of infinitely many sets.

4.26 Lemma 4.2 shows that the collection of regular languages over \mathbb{B} is countable. Show that not every individual language in it is countable.

4.27 An alternative definition of a regular language is one generated by a **regular grammar**, where rewrite rules have three forms: $X \rightarrow tY$, or $X \rightarrow t$, or $X \rightarrow \epsilon$. That is, the rule head has one nonterminal and rule body has a terminal followed by a nonterminal, or possibly a single nonterminal, or the empty string. This is an example, with the language that it generates.

$$S \rightarrow aS \mid bS$$

$$S \rightarrow aA$$

$$A \rightarrow aB$$

$$B \rightarrow \epsilon \mid b$$

$$\mathcal{L} = \{\sigma \in \{a, b\}^* \mid \sigma = \tau \hat{\cdot} aa \text{ or } \sigma = \tau \hat{\cdot} aab\}$$

Here we outline an algorithm that inputs a regular grammar and produces a Finite State machine that recognizes the same language. Apply these steps to the above grammar.

- For each nonterminal X make a machine state q_X , where the start state is the one for the start symbol.
- For each $X \rightarrow \epsilon$ rule make state q_X accepting.
- For each $X \rightarrow tY$ rule put a transition from q_X to q_Y labeled t .
- If there are any $X \rightarrow t$ rules then make an accepting state \bar{q} , and for each such rule put a transition from q_X to \bar{q} labeled t .

- ✓ 4.28 We can give an alternative proof of Theorem 4.5, that the collection of regular languages is closed under set intersection, set difference, and set complement, that does not rely on a vague “by construction.”

- Observe that the identity $S \cap T = (S^c \cup T^c)^c$ gives intersection in terms of union and complement. Use Lemma 4.3 to argue that if regular languages are closed under complement then they are also closed under intersection.
- Use the identity $S - T = S \cap T^c$ to make a similar observation about set difference.
- Show that the complement of a regular language is also a regular language.

- 4.29 Prove that the language recognized by a Finite State machine with n states is infinite if and only if the machine accepts at least one string of length k , where $n \leq k < 2n$.

- 4.30 Fix two alphabets Σ_0, Σ_1 . A function $h: \Sigma_0 \rightarrow \Sigma_1^*$ induces a **homomorphism** on Σ_0^* via the operation $h(\sigma^\frown \tau) = h(\sigma)^\frown h(\tau)$ and $h(\epsilon) = \epsilon$.

- Take $\Sigma_0 = \mathbb{B}$ and $\Sigma_1 = \{a, b\}$. Fix a homomorphism $\hat{h}(0) = a$ and $\hat{h}(1) = ba$. Find $\hat{h}(01)$, $\hat{h}(10)$, and $\hat{h}(101)$.
- Define $h(\mathcal{L}) = \{h(\sigma) \mid \sigma \in \Sigma_0^*\}$. Let $\hat{\mathcal{L}} = \{\sigma^\frown 1 \mid \sigma \in \mathbb{B}^*\}$; describe it with a regular expression. Using the homomorphism \hat{h} from the prior item, describe $\hat{h}(\hat{\mathcal{L}})$ with a regular expression.
- Prove that the collection of regular languages is closed under homomorphism, that if \mathcal{L} is regular then so is $h(\mathcal{L})$.

- 4.31 The proofs here work with deterministic Finite State machines. Find a nondeterministic Finite State machine \mathcal{M} so that producing another machine $\hat{\mathcal{M}}$ by taking the complement of the accepting states, $F_{\hat{\mathcal{M}}} = (F_{\mathcal{M}})^c$, will not result in the language of the second machine being the complement of the language of the first.

- 4.32 We will show that the class of regular languages is closed under reversal. Recall that the reversal of the language is defined to be the set of reversals of the strings in the language $\mathcal{L}^R = \{\sigma^R \mid \sigma \in \mathcal{L}\}$.

- Show that for any two strings the reversal of the concatenation is the concatenation, in the opposite order, of the reversals $(\sigma_0^\frown \sigma_1)^R = \sigma_1^R \frown \sigma_0^R$.
Hint: do induction on the length of σ_1 .
- We will prove the result by showing that for any regular expression R , the reversal $\mathcal{L}(R)^R$ is described by a regular expression. We will construct this expression by defining a reversal operation on regular expressions. Fix an

alphabet Σ and let (i) $\emptyset^R = \emptyset$, (ii) $\varepsilon^R = \varepsilon$, (iii) $x^R = x$ for any $x \in \Sigma$, (iv) $(R_0 \cap R_1)^R = R_1^R \cap R_0^R$, (v) $(R_0 \cup R_1)^R = R_0^R \cup R_1^R$, and (vi) $(R^*)^R = (R^*)^*$. (Note the relationship between (iv) and the prior exercise item.) Now show that R^R describes $\mathcal{L}(R)^R$. Hint: use induction on the length of the regular expression R .

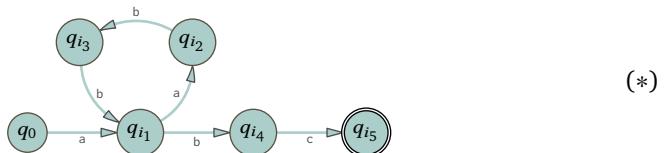
SECTION

IV.5 Languages that are not regular

The prior section gave a counting argument to show that there are languages that are not regular. We now cover a technique to show that specific languages are not regular.[†]

The idea is that although Finite State machines are finite, they can handle arbitrarily long inputs. For instance, the power switch from Example 1.1 has only two states but even if we toggle it hundreds of times, it still keeps track of whether the switch is on or off. To handle long inputs with only a small number of states a machine must revisit states, that is, it must cycle.

Cycles inside a machine causes a pattern in what that machine accepts. The diagram below shows a machine that accepts aabbabc (it only shows some of the states, those that the machine traverses in processing this input).



Because of the cycle, in addition to aabbabc this machine must also accept $a(ab)^2bc$ since that string takes the machine through the cycle twice. Likewise, this machine accepts $a(ab)^3bc$, and cycling more times pumps out more accepted strings.

- 5.1 **THEOREM (PUMPING LEMMA)** Let \mathcal{L} be a regular language. Then there is a constant $p \in \mathbb{N}$, the **pumping length** for the language, such that every string $\sigma \in \mathcal{L}$ with $|\sigma| \geq p$ decomposes into three substrings $\sigma = \alpha\beta\gamma$ satisfying: (1) the first two components are short, $|\alpha\beta| \leq p$, (2) β is not empty, and (3) all of the strings $\alpha\gamma, \alpha\beta^2\gamma, \alpha\beta^3\gamma, \dots$ are also members of the language \mathcal{L} .

Proof Suppose that \mathcal{L} is recognized by the Finite State machine \mathcal{M} . Take p to be the number of states in \mathcal{M} .

Consider a string $\sigma \in \mathcal{L}$ with $|\sigma| \geq p$. Finite State machines perform one transition per character so the number of characters in an input string equals the number of transitions. Thus the number of states, not necessarily distinct ones, that the machine visits is one more than the number of transitions. (For instance,

[†]This is like in the second chapter, where we first used a counting argument to prove that there are unsolvable problems and later showed that the Halting problem and related problems are unsolvable.

with a one-character input a machine visits two states.) So in processing σ , the machine revisits at least one state. It cycles.

Of the states that are repeated as the machine processes σ , fix the one q that it revisits first. Also fix σ 's shortest two prefixes $\langle s_0, \dots, s_i \rangle$ and $\langle s_0, \dots, s_i, \dots, s_j \rangle$ that take the machine to q . That is, i and j are minimal such that $i \neq j$ and the extended transition function gives $\hat{\Delta}(\langle s_0, \dots, s_i \rangle) = \hat{\Delta}(\langle s_0, \dots, s_j \rangle) = q$. Let $\alpha = \langle s_0, \dots, s_i \rangle$, let $\beta = \langle s_{i+1}, \dots, s_j \rangle$, and let $\gamma = \langle s_{j+1}, \dots, s_k \rangle$. (Think of α as the initial substring of σ that transitions the machine up to the loop, think of β as the substring that takes the machine around the loop once, and γ is the rest of σ . Possibly α is empty. Possibly also γ is empty, or perhaps it includes a substring that transitions the machine around the loop a number of times after β has taken it around the first time. For the machine in (*) above, with $\sigma = a(ab)^2bc$ we have $\alpha = a$, $\beta = ab$, and $\gamma = bbb$.)

These strings satisfy conditions (1) and (2). In particular, choosing q , i , and j to be minimal guarantees that that $|\alpha \cap \beta| \leq p$ because the machine has p -many states and so a state must repeat by at most the p -th input character. For condition (3), this string

$$\alpha \cap \gamma = \langle s_0, \dots, s_i, s_{j+1}, \dots, s_k \rangle$$

brings the machine from the start state q_0 to q , and then to the same ending state as did σ . That is, $\hat{\Delta}(\alpha\gamma) = \hat{\Delta}(\alpha\beta\gamma)$ and so the machine accepts $\alpha\gamma$. The other strings in (3) work the same way. For instance, for

$$\alpha \cap \beta^2 \cap \gamma = \langle s_0, \dots, s_i, s_{i+1}, \dots, s_j, s_{i+1}, \dots, s_j, s_{j+1}, \dots, s_k \rangle$$

the substring α brings the machine from q_0 to q , the first β brings it from q around to q again, the second β makes the machine cycle to q yet again, and finally γ brings it to the same ending state as did σ . \square

Typically we use the Pumping Lemma to show that a language is not regular, via an argument by contradiction.

- 5.2 EXAMPLE The canonical example is to show that this language of matched parentheses is not regular.

$$\mathcal{L} = \{ (n)^n \in \Sigma^* \mid n \in \mathbb{N} \} = \{ \epsilon, (), ((()), ((((), ((^4)^4, \dots \}) \}$$

The alphabet is the set of parentheses characters, $\Sigma = \{(), ()\}$.

For contradiction, assume that \mathcal{L} is regular. Then the Pumping Lemma says that \mathcal{L} has a pumping length. Call it p . Consider the string $\sigma = (p)^p$.

It is an element of \mathcal{L} and its length is greater than or equal to p and therefore it decomposes into three substrings $\sigma = \alpha \cap \beta \cap \gamma$ that satisfy the conditions. Condition (1) is that the length of the prefix $\alpha \cap \beta$ is less than or equal to p . Because of this condition and because the first p -many characters of σ are all open parentheses, we know that both α and β are composed entirely of open parentheses. Condition (2) is that β is not empty so it consists of at least one $($.

Condition (3) is that all of the strings $\alpha\gamma$, $\alpha\beta^2\gamma$, $\alpha\beta^3\gamma$, \dots are members of \mathcal{L} . To

get the contradiction, consider $\alpha\beta^2\gamma$ (other choices are also possible). Compared with $\sigma = \alpha\beta\gamma$, this string has an extra β , which adds at least one open parenthesis without also adding any closed parentheses. That is, $\alpha\beta^2\gamma$ has more '('s than ')'s. It is therefore not a member of \mathcal{L} . But the Pumping Lemma says that it must be a member of \mathcal{L} , and consequently the assumption that \mathcal{L} is regular is impossible.

- 5.3 EXAMPLE Recall that a palindrome is a string that reads the same backwards as forwards, such as bab, abbaabba, or a^5ba^5 . We will prove that the language $\mathcal{L} = \{\sigma \in \Sigma^* \mid \sigma^R = \sigma\}$ of all palindromes over $\Sigma = \{a, b\}$ is not regular.

For contradiction assume that this language is regular. The Pumping Lemma says that \mathcal{L} has a pumping length. Let it be p . Consider $\sigma = a^pba^p$.

The string σ is an element of \mathcal{L} and has more than p characters. Thus it decomposes as $\sigma = \alpha\beta\gamma$, subject to the three conditions. Condition (1) is that $|\alpha\beta| \leq p$ and so both substrings α and β are composed entirely of a's. Condition (2) is that β is not the empty string and so β consists of at least one a. Condition (3) states that all of the strings $\alpha\gamma, \alpha\beta^2\gamma, \alpha\beta^3\gamma, \dots$ are also members of \mathcal{L} . Consider the first, $\alpha\gamma$ (other list members also give contradictions but we just need one).

Compared to $\sigma = \alpha\beta\gamma$, in $\alpha\gamma$ the substring β is gone. Because α and β consist entirely of a's, the substring γ has the b character from σ , and hence also has the a^p that follows this b. So compared to $\sigma = \alpha\beta\gamma$, the string $\alpha\gamma$ omits at least one a before the b but none of the a's after it. Therefore $\alpha\gamma$ is not a palindrome, which contradicts the Pumping Lemma's third condition.

- 5.4 REMARK In that example the string σ has three parts, $\sigma = a^p \cap b \cap a^p$, and it decomposes into three parts, $\sigma = \alpha \cap \beta \cap \gamma$. Don't make the mistake of thinking that the two decompositions match. The Pumping Lemma does not say that $\alpha = a^p$, $\beta = b$, and $\gamma = a^p$ — indeed, we've shown that β does not contain the b. Instead, the Pumping Lemma only says that the first two substrings together, $\alpha \cap \beta$, consists exclusively of a's. So it could be that $\alpha\beta = a^p$, or it could be that the γ starts with some a's that are then followed by ba^p .

- 5.5 EXAMPLE Consider $\mathcal{L} = \{0^m1^n \in \mathbb{B}^* \mid m = n + 1\} = \{0, 001, 00011, \dots\}$. Its members have a number of 0's that is one more than the number of 1's. We will prove that it is not regular.

For contradiction assume otherwise, that \mathcal{L} is regular, and denote its pumping length by p . Consider $\sigma = 0^{p+1}1^p \in \mathcal{L}$. Because $|\sigma| \geq p$, the Pumping Lemma gives a decomposition $\sigma = \alpha\beta\gamma$ satisfying the three conditions. Condition (1) says that $|\alpha\beta| \leq p$, so that the substrings α and β have only 0's. Condition (2) says that β has at least one character, necessarily 0. Consider Condition (3)'s list: $\alpha\gamma, \alpha\beta^2\gamma, \alpha\beta^3\gamma, \dots$. Compare its first entry, $\alpha\gamma$, to σ . The string $\alpha\gamma$ has fewer 0's than does σ but the same number of 1's. So the number of 0's in $\alpha\gamma$ is not one more than its number of 1's. Thus $\alpha\gamma \notin \mathcal{L}$, which contradicts the Pumping Lemma.

We can interpret that example to say that Finite State machines cannot correctly recognize a predecessor-successor relationship. We can similarly use the Pumping Lemma to show Finite State machines cannot recognize other arithmetic relations.

- 5.6 EXAMPLE The language $\mathcal{L} = \{a^n \mid n \text{ is a perfect square}\} = \{\epsilon, a, a^4, a^9, a^{16}, \dots\}$ is not regular. For, suppose otherwise. Denote the pumping length by p and consider $\sigma = a^{(p^2)}$, so that $\sigma \in \mathcal{L}$ and $|\sigma| \geq p$.

By the Pumping Lemma, σ decomposes as $\alpha\beta\gamma$, subject to the three conditions. Condition (1) is that $|\alpha\beta| \leq p$, which implies that $|\beta| \leq p$. Condition (2) is that $0 < |\beta|$. Now consider the strings $\alpha\gamma, \alpha\beta^2\gamma, \dots$

We will get a contradiction from $\alpha\beta^2\gamma$. The definition of \mathcal{L} is that after σ the next longest string has length $(p+1)^2 = p^2 + 2p + 1$. The difference between p^2 and $p^2 + 2p + 1$ is strictly greater than p . However the gap between the length $|\sigma| = |\alpha\beta\gamma|$ and the length $|\alpha\beta^2\gamma|$ is at most p because $0 < |\beta| \leq p$. Hence the length of $\alpha\beta^2\gamma$ is not a perfect square, which contradicts the Pumping Lemma.

Sometimes we use the Pumping Lemma in conjunction with the closure properties of regular languages.

- 5.7 EXAMPLE The language $\mathcal{L} = \{\sigma \in \{a, b\}^* \mid \sigma \text{ has as many } a\text{'s as } b\text{'s}\}$ is not regular. To prove that, observe that the language $\hat{\mathcal{L}} = \{a^m b^n \in \{a, b\}^* \mid m, n \in \mathbb{N}\}$ is regular, described by the regular expression $a^* b^*$. Recall that the intersection of two regular languages is regular. But $\mathcal{L} \cap \hat{\mathcal{L}}$ is the set $\{a^n b^n \mid n \in \mathbb{N}\}$ and Example 5.2 shows that this language isn't regular, where we substitute a and b for the parentheses.

We have seen many examples of things that Finite State machines can do and here we have seen things that they cannot. This is a pleasing balance. But our interest is motivated by more than symmetry.

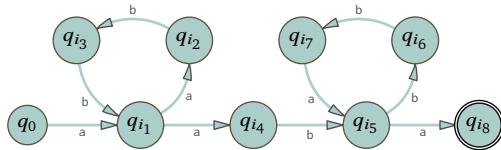
For instance, recognizing the language of balanced parentheses as in Example 5.2 is something that we often want to do in a compiler. A Turing machine can solve this problem but we now know that a Finite State machine cannot. We therefore now know that to solve this problem we must have some kind of scratch memory. So the results in this section speak to the resources needed to solve problems.

IV.5 Exercises

- ✓ 5.8 Example 5.5 shows that $\{0^m 1^n \in \mathbb{B}^* \mid m = n + 1\}$ is not regular but your friend doesn't get it and asks you, "What's wrong with the regular expression $0^{n+1} 1^n$?" Explain it to them.
- 5.9 Example 5.2 uses $\alpha\beta^2\gamma$ to show that the language of balanced parentheses is not regular. Instead get the contradiction by showing that $\alpha\gamma$ is not a member of the language.
- 5.10 Your friend has been thinking. They say, "Hey, the diagram (*) before Theorem 5.1 doesn't apply unless the language is infinite. Sometimes languages are regular because they only have like three or four strings. So the Pumping Lemma is wrong." In what way do they need to further refine their thinking?

- 5.11 Someone in the class emails you, “If a language has a string with length greater than the number of states, which is the pumping length, then it cannot be a regular language.” Correct?
- ✓ 5.12 Your study partner has read Remark 5.4 but it is still sinking in. About the matched parentheses example, Example 5.2, they say, “So $\sigma = ({}^p)^p$, and $\sigma = \alpha\beta\gamma$. We know that $\alpha\beta$ consists only of (’s, so it must be that γ consists of)’s.” Give them a prompt.
- 5.13 In class someone asks, “Isn’t it true that languages don’t have a unique pumping length? That if a length of $p = 5$ will do then $p = 6$ will also do?” Before the prof answers, what do you think?
- ✓ 5.14 For each, give five strings that are elements of the language and five that are not, and then show that the language is not regular.
- $\mathcal{L}_0 = \{ a^n b^m \mid n + 2 = m \}$
 - $\mathcal{L}_1 = \{ a^n b^m c^n \mid n, m \in \mathbb{N} \}$
 - $\mathcal{L}_2 = \{ a^n b^m \mid n < m \}$
- ✓ 5.15 For each language over $\Sigma = \{ a, b \}$ produce five strings that are members. Then decide if that language is regular. You must prove each assertion by either producing a regular expression or using the Pumping Lemma.
- $\{ a^n b^m \in \Sigma^* \mid n = 3 \}$
 - $\{ a^n b^m \in \Sigma^* \mid n + 3 = m \}$
 - $\{ \alpha^\sim \alpha \mid \alpha \in \Sigma^* \}$
 - $\{ a^n b^m \in \Sigma^* \mid n, m \in \mathbb{N} \}$
 - $\{ a^n b^m \in \Sigma^* \mid m - n > 12 \}$
- ✓ 5.16 Use the Pumping Lemma to prove that $\mathcal{L} = \{ a^{m-1} cb^m \mid m \in \mathbb{N}^+ \}$ is not regular. It may help to first produce five strings from the language.
- 5.17 Show that the language over $\{ a, b \}$ consisting of strings having more a’s than b’s is not regular.
- 5.18 One of these is regular and one is not. Which is which? You must prove your assertions.
- $\{ a^n b^m \in \{ a, b \}^* \mid n = m^2 \}$
 - $\{ a^n b^m \in \{ a, b \}^* \mid 3 < m, n \}$
- 5.19 Is $\{ \sigma \in \mathbb{B}^* \mid \sigma = \alpha\beta\alpha^R \text{ for } \alpha, \beta \in \mathbb{B}^* \}$ regular? Either way, prove it.
- 5.20 Prove that the language $\mathcal{L} = \{ \sigma \in \{ 1 \}^* \mid |\sigma| = n! \text{ for some } n \in \mathbb{N} \}$ is not regular. Hint: the differences $(n+1)! - n!$ grow without bound.
- 5.21 One of these is regular, one is not: $\{ 0^m 1 0^n \mid m, n \in \mathbb{N} \}$ and $\{ 0^n 1 0^n \mid n \in \mathbb{N} \}$. Which is which? Of course, you must prove your assertions.
- ✓ 5.22 Show that there is a Finite State machine that recognizes this language of all sums totaling less than four, $\mathcal{L}_4 = \{ a^i b^j c^k \mid i, j, k \in \mathbb{N} \text{ and } i + j = k \text{ and } k < 4 \}$. Use the Pumping Lemma to show that no Finite State machine recognizes the language of all sums, $\mathcal{L} = \{ a^i b^j c^k \mid i, j, k \in \mathbb{N} \text{ and } i + j = k \}$.

- 5.23 Decide if each is a regular language of bitstrings:
- the number of 0's plus the number of 1's equals five,
 - the number of 0's minus the number of 1's equals five.
- ✓ 5.24 Show that $\{0^m 1^n \in \mathbb{B}^* \mid m \neq n\}$ is not regular. *Hint:* use the closure properties of regular languages.
- 5.25 Example 5.7 shows that $\{\sigma \in \{a, b\}^* \mid \sigma \text{ has as many } a\text{'s as } b\text{'s}\}$ is not regular. In contrast, show that $\mathcal{L} = \{\sigma \in \{a, b\}^* \mid \sigma \text{ has as many } ab\text{'s as } ba\text{'s}\}$ is regular. *Hint:* think of ab and ba as marking a transition from a block of one character to a block of another.
- ✓ 5.26 Rebut someone who says to you, “Sure, for the machine before Theorem 5.1 on page 216, a single loop will cause $\sigma = \alpha^\frown \beta^\frown \gamma$. But if the machine had a double loop like below then you’d need a longer decomposition.”



5.27 Show that $\{\sigma \in \mathbb{B}^* \mid \sigma = 1^n \text{ where } n \text{ is prime}\}$ is not a regular language. *Hint:* the third condition’s sequence has a constant positive length difference.

- 5.28 Consider $\{a^i b^j c^{i+j} \mid i, j \in \mathbb{N}\}$.
- Give five strings from this language.
 - Show that it is not regular.

5.29 The language \mathcal{L} described by the regular expression $a^* bbbb^*$ is a regular language. We can apply the Pumping Lemma to it. The proof of the Pumping Lemma says that for the pumping length we can use the number of states in a machine that recognizes the language. Here that gives $p = 4$.

- Consider $\sigma = abbb$. Give a decomposition $\sigma = \alpha\beta\gamma$ that satisfies the three conditions.
- Do the same for $\sigma = b^{15}$.

5.30 For a regular language, a pumping length p is a number with the property that every word of length p or more can be pumped, that is, can be decomposed so that it satisfies the three properties of Theorem 5.1. The proof of that theorem shows that where a Finite State machine recognizes the language, the number of states in the machine suffices as a pumping length. But p can be smaller.

- Consider the language \mathcal{L} described by $(01)^*$. Construct a deterministic Finite State machine with three states that recognizes this language.
- Show that the minimal pumping length for \mathcal{L} is 1.

5.31 Nondeterministic Finite State machines can always be made to have a single accepting state. For deterministic machines that is not so.

- Show that any deterministic Finite State machine that recognizes the finite language $\mathcal{L}_1 = \{\epsilon, a\}$ must have at least two accepting states.

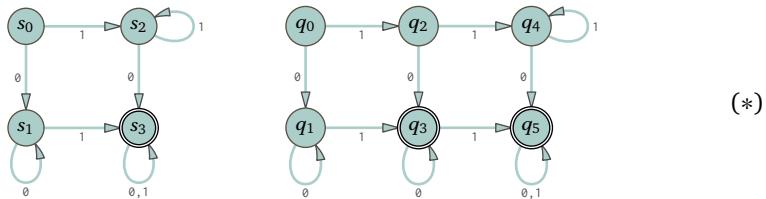
- (b) Show that any deterministic Finite State machine that recognizes $\mathcal{L}_2 = \{\varepsilon, a, aa\}$ must have at least three accepting states.
(c) Show that for any $n \in \mathbb{N}$ there is a regular language that is not recognized by any deterministic Finite State machine with at most n accepting states.

SECTION

IV.6 Minimization

We will give an algorithm that inputs a Finite State machine and the machine with the fewest states that also recognizes the same language.

The algorithm collapses together redundant states so we begin with an example of that. Compare these two machines. Each has the language $\mathcal{L} = \{\sigma \in \mathbb{B}^* \mid \sigma \text{ has at least one } 0 \text{ and at least one } 1\}$.



Our experience in designing machines is that each state needs a meaning that is well-defined in terms of the machine's language, and that is unique. For instance, on the left s_2 means "have seen the required 1 but still waiting for a 0." The machine on the right doesn't satisfy this design principle because the meaning of q_4 , "have seen the required 1, and another 1, but still waiting for a 0," is subsumed by the meaning of q_2 , with respect to \mathcal{L} . Similarly, q_5 's meaning is subsumed by q_3 's. This machine has redundant states.

This table lists what happens if the machine is now in state q_2 or q_4 and the given string is now on the tape. An entry says A if running the machine will end in an accepting state and R otherwise. The two states behave the same; if σ contains a 0 then both states give A, otherwise both give R.

σ	ε	0	1	00	01	10	11	000	001	010	...
q_2	R	A	R	A	A	A	R	A	A	A	...
q_4	R	A	R	A	A	A	R	A	A	A	...

As a contrast, if we give the string $\sigma = 0$ to the state q_2 then the machine ends in an accepting state, while if we give σ to q_0 then the machine rejects. So q_0 and q_2 are not redundant.

6.1 **DEFINITION** In a Finite State machine over Σ , two states q, \hat{q} are **indistinguishable**, $q \sim \hat{q}$, if for every string $\sigma \in \Sigma^*$, the two cases of starting the machine in q while giving it σ and starting the machine in \hat{q} while giving it σ always result in the same

outcome: either the machine ends in an accepting state in both cases or it ends in a rejecting state in both. States that are not indistinguishable are **distinguishable**.

We will compute when states are indistinguishable by checking whether they can be distinguished by strings of length 0, strings of length 1, etc. Fix a Finite State machine over an alphabet Σ and a length $n \in \mathbb{N}$. We say that q and \hat{q} are **n -indistinguishable**, $q \sim_n \hat{q}$, if for every string $\sigma \in \Sigma^*$ of length at most n , starting the machine in q while giving it σ and starting the machine in \hat{q} while giving it σ results in the same outcome, either that for both the machine ends in an accepting state or that for both it ends in a rejecting state. States are **n -distinguishable** if they are not n -indistinguishable, that is, if there exists a string $\sigma \in \Sigma^*$ with $|\sigma| \leq n$ such that starting the machine in one of the states and giving it input σ ends in an accepting state while starting it in the other and giving it σ does not.

Observe that states are 0-indistinguishable if either both are accepting states or both are not accepting states.

- 6.2 EXAMPLE Consider the machine on the left in (*) above and its s_0 and s_2 . Take the length one string $\sigma = 0$. Starting in state s_0 and processing σ ends in the rejecting state s_1 , but starting in s_2 and processing the same input ends in the accepting state s_3 . So s_0 and s_2 are 1-distinguishable and therefore are distinguishable.
- 6.3 EXAMPLE For the machine on the right above this table gives the result of starting in each state and processing each string of length 0, length 1, and length 2.

	ϵ	0	1	00	01	10	11
q_0	R	R	R	R	A	A	R
q_1	R	R	A	R	A	A	A
q_2	R	A	R	A	A	A	R
q_3	A	A	A	A	A	A	A
q_4	R	A	R	A	A	A	R
q_5	A	A	A	A	A	A	A

The initial column just records that members of $\{q_0, q_1, q_2, q_4\}$ are rejecting states while members of $\{q_3, q_5\}$ are accepting. We write $\mathcal{E}_{0,0} = \{q_0, q_1, q_2, q_4\}$ and $\mathcal{E}_{0,1} = \{q_3, q_5\}$ to denote that states are in the same class $\mathcal{E}_{0,i}$ if they are 0-indistinguishable.

The two columns for the length 1 strings show all four possible behaviors. The state q_0 goes to a rejecting result for both inputs, that is, the columns say R and R. For q_1 we have R,A. Both q_2 and q_4 show A,R. And q_3 and q_5 go to A,A. So there are four classes for 1-indistinguishability: $\mathcal{E}_{1,0} = \{q_0\}$, $\mathcal{E}_{1,1} = \{q_1\}$, $\mathcal{E}_{1,2} = \{q_2, q_4\}$, and $\mathcal{E}_{1,3} = \{q_3, q_5\}$. Note that the prior paragraph's class $\mathcal{E}_{0,1}$ has split into three, $\mathcal{E}_{1,0}$ and $\mathcal{E}_{1,1}$ and $\mathcal{E}_{1,2}$. The other class did not split, $\mathcal{E}_{0,1} = \mathcal{E}_{1,3}$.

Next, the length 2 strings. The only classes that could split are $\mathcal{E}_{1,2} = \{q_2, q_4\}$, and $\mathcal{E}_{1,3} = \{q_3, q_5\}$. Both q_2 and q_4 have A,A,A,R, while both q_3 and q_5 have A,A,A,A. So going to these longer strings does not further divide the classes.

- 6.4 LEMMA The relations \sim_0, \sim_1, \dots and \sim are equivalences. As such, they partition a machine's states into the **0-distinguishable classes**, the **1-distinguishable classes**, \dots along with the **distinguishable classes**.

Proof Exercise 6.24. □

This summarizes the prior example by listing the equivalence classes.

n	\sim_n classes			
0	$\mathcal{E}_{0,0} = \{q_0, q_1, q_2, q_4\}$	$\mathcal{E}_{0,1} = \{q_3, q_5\}$		
1	$\mathcal{E}_{1,0} = \{q_0\}$	$\mathcal{E}_{1,1} = \{q_1\}$	$\mathcal{E}_{1,2} = \{q_2, q_4\}$	$\mathcal{E}_{1,3} = \{q_3, q_5\}$
2	$\mathcal{E}_{2,0} = \{q_0\}$	$\mathcal{E}_{2,1} = \{q_1\}$	$\mathcal{E}_{2,2} = \{q_2, q_4\}$	$\mathcal{E}_{2,3} = \{q_3, q_5\}$

When we first saw this machine we spotted that q_2 and q_4 are redundant. Notice that all three lines in this table have them together in their class. The same holds for q_3 and q_5 .

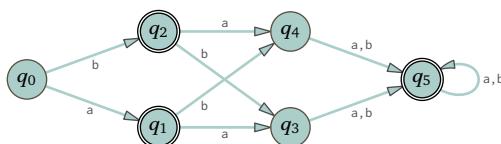
We now develop an algorithm, Moore's algorithm, based on this pattern. Given a machine, it finds the \sim_0 classes, then the \sim_1 classes, \dots until the classes stop splitting. At that point the classes are the \sim classes, and they serve as the states of the minimal machine.

There is just one difficulty to overcome. As presented in Example 6.3, at each stage we considered all length n string inputs. Of course there are 2^n -many such inputs to check, which would make an algorithm intolerably slow. Moore's algorithm refines that example, reducing the number of things to check.

For that, consider what we need for a splitting, that is, consider two states q and \hat{q} that are not n -distinguishable but that are $n + 1$ -distinguishable. Let them be distinguished by the length $n + 1$ string $\sigma = \langle s_0, s_1, \dots, s_{n-1}, s_n \rangle$, which we can write as $\sigma = \tau \hat{s}_n$. Because they are not n -distinguishable, where τ brings the machine from q to some state r and also brings the machine from \hat{q} to some \hat{r} , then r and \hat{r} are in the same n -equivalence class, $\mathcal{E}_{n,i}$. Therefore, the distinguishing between these two states must happen with σ 's final character, s_n . It must take the machine from state r to a state in one class, $\mathcal{E}_{n+1,j}$, and also take the machine from \hat{r} to a state in a different class, $\mathcal{E}_{n+1,\hat{j}}$. In short, if there is going to be a split in passing from the n -distinguishability classes to the $n + 1$ -distinguishability classes then a single character suffices to produce such a split.

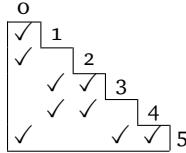
The examples of Moore's algorithm that follow leverage this reasoning. They also show how to use the \sim classes to make a minimal machine.

- 6.5 EXAMPLE We will find a machine that recognizes the same language as this one but that has a minimum number of states.



For bookkeeping we will use triangular tables that have an entry for every pair of different states q_i and q_j . Checkmarks show state pairs that are distinguishable.

Start by checkmarking the i, j entries that are 0-distinguishable, that is, where one of q_i and q_j is accepting while the other is not. For example, the table below has a check denoting that state q_0 is 0-distinguishable from state q_1 .



Blank entries in this initial table mark state pairs that are 0-indistinguishable. For instance, q_1 and q_2 are 0-indistinguishable. By eye we can read off the \sim_0 -equivalence classes.

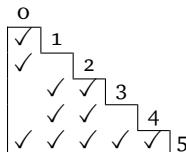
$$\mathcal{E}_{0,0} = \{q_0, q_3, q_4\} \quad \mathcal{E}_{0,1} = \{q_1, q_2, q_5\}$$

Next the algorithm determines whether these classes split. We check each pair q_i, q_j that are together in a \sim_0 class. We take the characters from the alphabet and see where that character sends q_i and q_j . We see splitting when they are sent to members of unequal \sim_0 classes. For instance, on the q_0, q_3 line, the character a takes q_0 to q_1 and takes q_3 to q_5 . The two outcomes, q_1 and q_5 , are both members of $\mathcal{E}_{0,1}$, so we have not yet seen splitting.

	a	b	a	b
q_0, q_3	q_1, q_5	q_2, q_5	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$
q_0, q_4	q_1, q_5	q_2, q_5	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$
q_3, q_4	q_5, q_5	q_5, q_5	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$
q_1, q_2	q_3, q_4	q_4, q_3	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$
q_1, q_5	q_3, q_5	q_4, q_5	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$
q_2, q_5	q_4, q_5	q_3, q_5	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$

But the table does show class splitting. One example is in the q_1, q_5 line. Those two states are together in $\mathcal{E}_{0,1}$ but on the right the entry under a has two different classes, $\mathcal{E}_{0,0}$ and $\mathcal{E}_{0,1}$. Therefore at least one of the \sim_0 classes splits and the algorithm goes on to find \sim_1 classes.

We refine that triangular table to make the next one. Because of the q_1, q_5 splitting in the prior paragraph we add a checkmark in location 1, 5. Similarly, the prior table shows a q_2, q_5 splitting so we add a checkmark at 2, 5.



We can read off the \sim_1 classes.

$$\mathcal{E}_{1,0} = \{q_0, q_3, q_4\} \quad \mathcal{E}_{1,1} = \{q_1, q_2\} \quad \mathcal{E}_{1,2} = \{q_5\}$$

The \sim_0 class $\mathcal{E}_{0,1} = \{q_1, q_2, q_5\}$ has split into two \sim_1 classes.

Now iterate. Obviously $\mathcal{E}_{1,2}$ can't split so we only consider state pairs from the other two classes. The right side shows that q_0 is 2-distinguishable from q_3 and q_4 . The next triangular table adds checkmarks in the 0, 3 and 0, 4 entries.

	a	b	a	b	o
$\{q_0, q_3\}$	$\{q_1, q_5\}$	$\{q_2, q_5\}$	$\mathcal{E}_{1,1}, \mathcal{E}_{1,2}$	$\mathcal{E}_{1,1}, \mathcal{E}_{1,2}$	✓ 1
$\{q_0, q_4\}$	$\{q_1, q_5\}$	$\{q_2, q_5\}$	$\mathcal{E}_{1,1}, \mathcal{E}_{1,2}$	$\mathcal{E}_{1,1}, \mathcal{E}_{1,2}$	✓ 2
$\{q_3, q_4\}$	$\{q_5, q_5\}$	$\{q_5, q_5\}$	$\mathcal{E}_{1,2}, \mathcal{E}_{1,2}$	$\mathcal{E}_{1,2}, \mathcal{E}_{1,2}$	✓ 3
$\{q_1, q_2\}$	$\{q_3, q_4\}$	$\{q_4, q_3\}$	$\mathcal{E}_{1,0}, \mathcal{E}_{1,0}$	$\mathcal{E}_{1,0}, \mathcal{E}_{1,0}$	✓ 4
					✓ 5

That is, the \sim_1 class $\mathcal{E}_{1,0}$ splits into two \sim_2 classes.

$$\mathcal{E}_{2,0} = \{q_0\} \quad \mathcal{E}_{2,1} = \{q_1, q_2\} \quad \mathcal{E}_{2,2} = \{q_3, q_4\} \quad \mathcal{E}_{2,3} = \{q_5\}$$

One more iteration gives this.

	a	b	a	b	o
$\{q_1, q_2\}$	$\{q_3, q_4\}$	$\{q_4, q_3\}$	$\mathcal{E}_{2,2}, \mathcal{E}_{2,2}$	$\mathcal{E}_{2,2}, \mathcal{E}_{2,2}$	✓ 1
$\{q_3, q_4\}$	$\{q_5, q_5\}$	$\{q_5, q_5\}$	$\mathcal{E}_{2,3}, \mathcal{E}_{2,3}$	$\mathcal{E}_{2,3}, \mathcal{E}_{2,3}$	✓ 2
					✓ 3
					✓ 4
					✓ 5

That calculation shows no more splitting. The algorithm terminates.

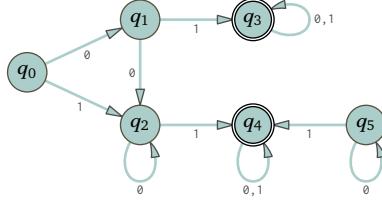
$$\mathcal{E}_0 = \{q_0\} \quad \mathcal{E}_1 = \{q_1, q_2\} \quad \mathcal{E}_2 = \{q_3, q_4\} \quad \mathcal{E}_3 = \{q_5\}$$

To finish we will produce the minimal machine. Take r_0 as a name for \mathcal{E}_0 , take r_1 for \mathcal{E}_1 , r_2 for \mathcal{E}_2 , and r_3 for \mathcal{E}_3 . The start state is the one containing q_0 , namely r_0 . The final states are the ones containing final states of the original machine, r_1 and r_3 .



As to the transitions between states, consider what happens when we feed the character a to $r_0 = \mathcal{E}_0 = \{q_0\}$. In the original machine, q_0 under input a goes to q_1 . Since q_1 is an element of $\mathcal{E}_1 = r_1$, in the minimal machine the a arrow out of r_0 goes to r_1 . Other connections work the same way.

- 6.6 EXAMPLE We will minimize the machine below. The algorithm has an additional step, which did not come up in the prior example but does come up here. This machine has an unreachable state, q_5 . So we start by omitting that state.



That leaves these \sim_0 classes and this initial triangular table.

o	1	2	3	4
✓ ✓ ✓ ✓	1	2	3	4
✓ ✓ ✓ ✓				
✓ ✓ ✓ ✓				
✓ ✓ ✓ ✓				

$$\mathcal{E}_{0,0} = \{q_0, q_1, q_2\} \quad \mathcal{E}_{0,1} = \{q_3, q_4\}$$

The algorithm calls for us to check whether the \sim_0 classes split. Below, the calculation's q_0, q_1 row shows that on being fed a 1 the outcomes, q_2 and q_3 , are in different \sim_0 classes, and its second row shows a similar thing. We've included the updated triangular table.

0	1	0	1	0	1
q_0, q_1	q_1, q_2	q_2, q_3	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$	
q_0, q_2	q_1, q_2	q_2, q_4	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$	
q_1, q_2	q_2, q_2	q_3, q_4	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$	
q_3, q_4	q_3, q_4	q_3, q_4	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$	

✓ ✓ ✓ ✓	1	2	3	4
✓ ✓ ✓ ✓				
✓ ✓ ✓ ✓				
✓ ✓ ✓ ✓				

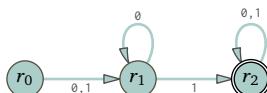
That is, $\mathcal{E}_{0,0} = \{q_0, q_1, q_2\}$ splits, and these are the \sim_1 classes.

$$\mathcal{E}_{1,0} = \{q_0\} \quad \mathcal{E}_{1,1} = \{q_1, q_2\} \quad \mathcal{E}_{1,2} = \{q_3, q_4\}$$

On the next iteration

0	1	0	1	0	1
q_1, q_2	q_2, q_2	q_3, q_4	$\mathcal{E}_{1,1}, \mathcal{E}_{1,1}$	$\mathcal{E}_{1,2}, \mathcal{E}_{1,2}$	
q_3, q_4	q_3, q_4	q_3, q_4	$\mathcal{E}_{1,2}, \mathcal{E}_{1,2}$	$\mathcal{E}_{1,1}, \mathcal{E}_{1,1}$	

no more splitting happens. The minimized machine has three states, one for each \sim equivalence class, $\mathcal{E}_0 = \{q_0\}$, $\mathcal{E}_1 = \{q_1, q_2\}$, and $\mathcal{E}_2 = \{q_3, q_4\}$.

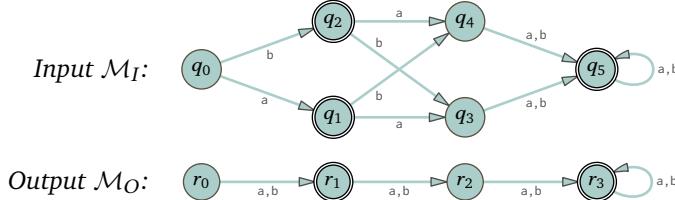


We end with a lemma verifying that Moore's algorithm works. We will cover that the algorithm halts, as well as that the machine it produces recognizes the same language and is minimal.

But there is another thing to verify that may not be obvious, namely that the algorithm produces a machine whose Δ function is well-defined. Consider the minimal machine in Example 6.5 and the arrows leaving its state $r_1 = \mathcal{E}_1 = \{q_1, q_2\}$.

What if feeding an a to q_1 and q_2 sends them to states in different \sim classes? That would be trouble—which state r_j should get the a arrow leaving r_1 ? Now, Example 6.5 doesn't have this trouble. Its states q_1 and q_s are both sent by a to states in \mathcal{E}_2 (namely q_3 and q_4). Nonetheless, we need to argue that for the machines produced by the algorithm, the transition function is well-defined.

- 6.7 REMARK Before that, we pause to suggest where this well-definedness comes from with two illustrations. Consider again the machines from Example 6.5. Below we picture the connection between their states as a projection, $p(q_0) = r_0$, $p(q_1) = p(q_2) = r_1$, $p(q_3) = p(q_4) = r_2$, and $p(q_5) = r_3$. When the algorithm collapses q_2 and q_1 into the single r_1 , it ensures that the two arrows from q_0 to those redundant states become the single a, b arrow from r_0 to r_1 . We say that the transformation from \mathcal{M}_I to \mathcal{M}_O respects the starting machine's operation.



The other illustration uses tables. Moore's algorithm combines the states such that the transitions in \mathcal{M}_O fit with—that is, respect—the transitions in \mathcal{M}_I . By this we mean that, for instance, $q_1 \mapsto r_1$ and when presented with input a those transition to q_3 and r_2 , and the map respects the transition because $q_3 \mapsto r_2$.

$\Delta_{\mathcal{M}_I}$	a	b	$\Delta_{\mathcal{M}_O}$	a	b
q_0	q_1	q_2	r_0	r_1	r_1
q_1	q_3	q_4	r_1	r_2	r_2
q_2	q_4	q_3			
q_3	q_5	q_5	r_2	r_3	r_3
q_4	q_5	q_5			
q_5	q_5	q_5	r_3	r_3	r_3

In a formula, the map $p: \mathcal{M}_I \rightarrow \mathcal{M}_O$ satisfies that $\Delta_{\mathcal{M}_O}(p(q), x) = p(\Delta_{\mathcal{M}_I}(q, x))$ for all $q \in \mathcal{M}_I$ and $x \in \Sigma$.

- 6.8 LEMMA Moore's algorithm returns a machine that recognizes the same language as the input machine, $\mathcal{L}(\mathcal{M}_O) = \mathcal{L}(\mathcal{M}_I)$, and that from among all of the machine recognizing the same language has the minimal number of states.

Proof We first argue that the algorithm halts for all input machines. This is easy: the algorithm halts at a step when no class splits and since these machines have only finitely many states, that step must eventually appear.

We next argue that \mathcal{M}_O 's transition function is well defined. Simply, if there is a \sim_n class where some character sends two states in that class to different \sim_n classes then the algorithm does not terminate at that step. Instead it splits that class and

goes to the next step. So the algorithm cannot end with an ill-defined Δ .

Next we verify that \mathcal{M}_I and \mathcal{M}_O recognize the same language. The input machine \mathcal{M}_I starts in q_0 while the output machine \mathcal{M}_O starts in a state $r_0 = \mathcal{E}_0$ that contains q_0 . Feed both machines the same string, $\sigma \in \Sigma^*$. The first character of σ moves \mathcal{M}_I from q_0 to some state q_{i_1} and moves \mathcal{M}_O from a state that contains q_0 to a state that contains q_{i_1} . The second character of σ moves \mathcal{M}_I from q_{i_1} to some q_{i_2} and moves \mathcal{M}_O from a state that contains q_{i_1} to a state that contains q_{i_2} . The machines proceed in this way until the string runs out. At the end \mathcal{M}_I is in a final state if and only if \mathcal{M}_O is in a state that contains that final state, which is itself a final state of \mathcal{M}_O . Thus the two machines accept the same set of strings.

The last verification is that \mathcal{M}_O has a minimal number of states. Let $\hat{\mathcal{M}}$ recognize the same language as \mathcal{M}_O ; we will show that it has at least as many states. We will do this by associating each state in \mathcal{M}_O with at least one state in $\hat{\mathcal{M}}$ in such a way that never are different states in \mathcal{M}_O associated with the same state in $\hat{\mathcal{M}}$.

Consider the union of the sets of states of the two machines (assume that they have no states in common, or else rename). As we did with Moore's algorithm, begin by saying that two states in the union are 0-indistinguishable if both are final in their own machine or if neither is final. Step $n + 1$ of this process starts with the \sim_n classes $\mathcal{E}_{n,0}, \dots, \mathcal{E}_{n,k}$ of states from the union. For each such class $\mathcal{E}_{n,i}$, see if it splits. That is, see if there are two states in $\mathcal{E}_{n,i}$ that are sent by a character $x \in \Sigma$ to different \sim_n classes. If so then this gives the \sim_{n+1} classes. When we reach a step with no splitting then we know which states are indistinguishable and these form the \sim classes.

Because $\mathcal{L}(\mathcal{M}_O) = \mathcal{L}(\hat{\mathcal{M}})$, the start states in the two machines are indistinguishable and are in the same \sim class. Also, if two states are indistinguishable then their successor states on any one input symbol $x \in \Sigma$ are also indistinguishable from each other, since if a string σ distinguishes between the successors then $\sigma^\frown x$ distinguishes between the original two states. In turn, the successors of these successors are indistinguishable, etc.

We are ready to define the association: states in \mathcal{M}_O and $\hat{\mathcal{M}}$ are associated if they are in the same \sim class. We first show that every state q of \mathcal{M}_O is associated with at least one state of $\hat{\mathcal{M}}$. Because \mathcal{M}_O is the output of the minimization process, it has no inaccessible state. So there is a string that takes the start state of \mathcal{M}_O to q . This string takes the start state of $\hat{\mathcal{M}}$ to some \hat{q} , and the prior paragraph applies to give that $q \sim \hat{q}$.

We finish by showing that there cannot be two different states of \mathcal{M}_O that are both associated with the same state of $\hat{\mathcal{M}}$. If there were two such states then by Lemma 6.4 they would be indistinguishable from each other. But that's impossible because \mathcal{M}_O is the output of the minimization process, which ensures that all of its states are distinguishable from each other. \square

In this chapter we have seen that when we have a problem to solve with a Finite State machine, often a nondeterministic machine is easier and more natural. An

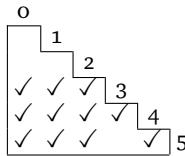
example is the algorithm that inputs a regular expression and outputs a machine recognizing that expression.

But our algorithm for converting a nondeterministic machine to a deterministic machine has a problem. Where the nondeterministic machine has n states, the deterministic machine has 2^n .

This section's result alleviates that. We now have a three-step process: from a problem we start with a nondeterministic answer, convert that to an equivalent deterministic machine, and then minimize to get a reasonably-sized final answer. In practice, this gives good results.

IV.6 Exercises

6.9 From the triangular table find the \sim_i classes.



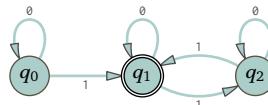
6.10 From the \sim_i classes find the associated triangular table. (A) $\mathcal{E}_{i,0} = \{q_0, q_1\}$, $\mathcal{E}_{i,1} = \{q_2\}$, and $\mathcal{E}_{i,2} = \{q_3, q_4\}$, (B) $\mathcal{E}_{i,0} = \{q_0\}$, $\mathcal{E}_{i,1} = \{q_1, q_2, q_4\}$, and $\mathcal{E}_{i,2} = \{q_3\}$, (C) $\mathcal{E}_{i,0} = \{q_0, q_1, q_5\}$, $\mathcal{E}_{i,1} = \{q_2, q_3\}$, and $\mathcal{E}_{i,2} = \{q_4\}$,

✓ 6.11 Suppose that $\mathcal{E}_{0,0} = \{q_0, q_1, q_2, q_5\}$ and $\mathcal{E}_{0,1} = \{q_3, q_4\}$, and from the machine you compute this table.

	a	b	a	b
q_0, q_1	q_1, q_1	q_2, q_3	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$
q_0, q_2	q_1, q_2	q_2, q_4	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$
q_0, q_5	q_1, q_5	q_2, q_5	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$
q_1, q_2	q_1, q_2	q_3, q_4	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$
q_1, q_5	q_1, q_5	q_3, q_5	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,0}$
q_2, q_5	q_2, q_5	q_4, q_5	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,0}$
q_3, q_4	q_3, q_4	q_5, q_5	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$

(A) Which lines of the table do you checkmark? (B) Give the resulting \sim_1 classes.

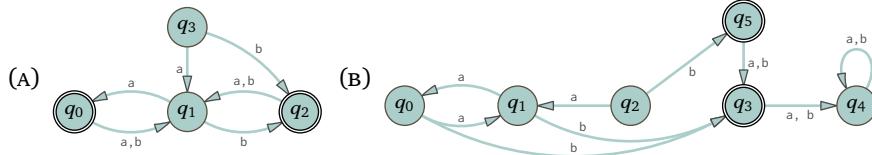
✓ 6.12 This machine accepts strings with an odd parity, with an odd number of 1's. Minimize it, using the algorithm described in this section. Show your work.



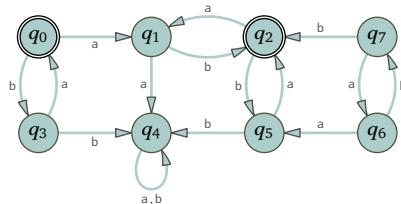
✓ 6.13 For many machines we can find the unreachable states by eye, but there is an algorithm. It inputs a machine \mathcal{M} and initializes the set of reachable states to $R_0 = \{q_0\}$. For $n > 0$, step n of the algorithm is: for each $q \in R_n$ find all states \hat{q} reachable from q in one transition and add those to make R_{n+1} . That

is, $R_{n+1} = R_n \cup \{ \hat{q} = \Delta_M(q, x) \mid q \in R_n \text{ and } x \in \Sigma \}$. The algorithm stops when $R_k = R_{k+1}$ and the set of reachable states is $R = R_k$. The unreachable states are the others, $Q - R$.

For each machine, perform this algorithm. Show the steps.

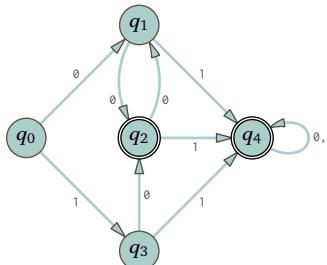


- ✓ 6.14 Perform the minimization algorithm on the machine with redundant states at the start of this section, the one on the right in (*) on page 222.
- 6.15 What happens when you minimize a machine that is already minimal?
- ✓ 6.16 This machine accepts strings described by $(ab|ba)^*$. Minimize it, using the algorithm of this section and showing the work.

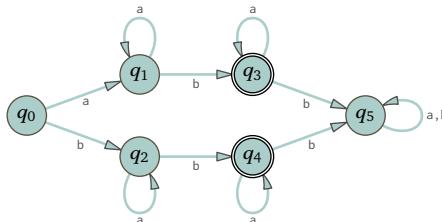


6.17 If a machine's start state is accepting, must the minimized machine's start state be accepting? If you think “yes” then prove it, and if you think “no” then give an example machine where it is false.

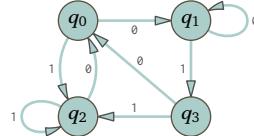
6.18 Minimize this machine.



6.19 Minimize this. Show the work, including producing the diagram of the minimized machine.

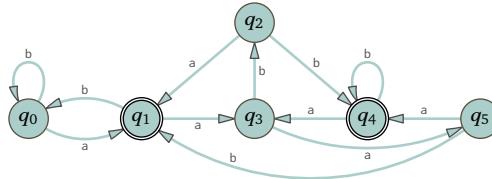


6.20 This machine has no accepting states. Minimize it.



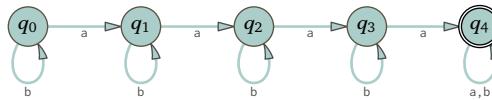
What happens to a machine where all states are accepting?

6.21 Minimize this machine.



6.22 What happens if you perform the minimization procedure in Example 6.6 without first omitting the unreachable state?

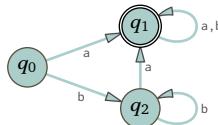
✓ 6.23 Minimize.



Note that the algorithm takes, roughly, a number of steps that are equal to the number of states in the machine.

6.24 Verify Lemma 6.4. (A) Verify that each \sim_n is an equivalence relation between states. (B) Verify that \sim is an equivalence.

6.25 There are ways to minimize Finite State machines other than the one given in this section. One is Brzozowski's algorithm, which has the advantage of being surprising and fun in that you perform some steps that seem a bit wacky and unrelated to elimination of states and then at the end it has worked. (However, it has the disadvantage of taking worst-case exponential time.) We will walk through the algorithm using this Finite State machine, \mathcal{M} .



(A) Use the algorithm in this section to minimize it.

(B) Instead, get a new machine by taking \mathcal{M} , changing the state names to be t_i instead of q_i , and reversing all the arrows. This gives a nondeterministic machine. Mark what used to be the initial state as an accepting state and mark what used to be the accepting state as an initial state. (In general, this may result in a machine with more than one initial state.)

- (c) Use the method described in an earlier section to convert this into a deterministic machine, whose states are named u_i . Omit unreachable states.
- (d) Repeat the second item by changing the state names to v_i instead of u_i , and reversing all the arrows. Mark what used to be the initial state as an accepting state and mark what used to be the accepting state as an initial state.
- (e) Convert to a deterministic machine and compare with the one in the first item.

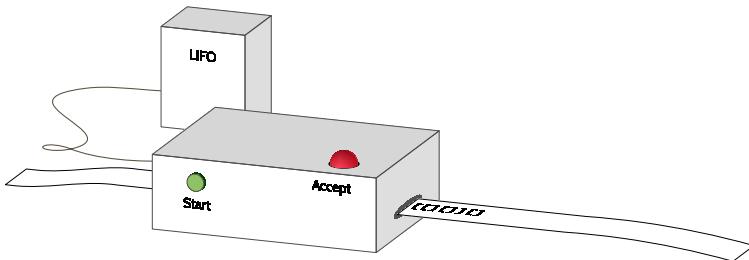
6.26 For each language \mathcal{L} recognized by some Finite State machine, let $\text{rank}(\mathcal{L})$ be the smallest number $n \in \mathbb{N}$ such that \mathcal{L} is accepted by a Finite State machine with n states. Prove that for every n there is a language with that rank.

SECTION

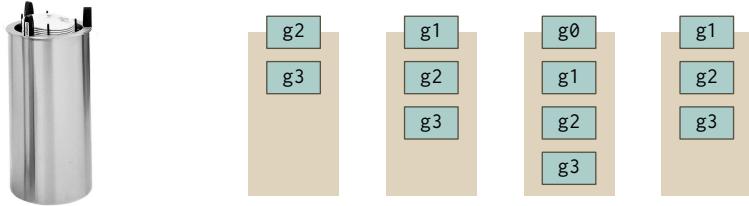
IV.7 Pushdown machines

No Finite State machine can recognize the language of balanced parentheses. So this machine model is not powerful enough to, for instance, decide whether input strings are valid programs in most programming languages. To handle nested parentheses, the natural data structure is a pushdown stack. We now supplement a Finite State machine by giving it access to a stack.

A stack is like the restaurant dish dispenser below: when you push a new dish on, its weight compresses a spring underneath, so that the old ones move down and the most recent dish is the only one that you can reach. When you pop that top dish off, the spring pushes the remaining dishes up and now you can reach the next one. We say that this stack is **LIFO**: Last-In, First-Out.



Below on the right is a sequence of views of a stack. Initially the stack has two characters, g_3 and g_2 . We push g_1 on the stack, and then g_0 . Now, although g_1 is on the stack, we don't have immediate access to it. To get at g_1 we must first pop off g_0 , as in the last stack shown.



Like a Turing machine tape, a stack provides storage that is unbounded. But it has restrictions that the tape does not. Once something is popped, it is gone. We could include in the machine a state whose intuitive meaning is that we have just popped g_0 but as there are finitely many states and unboundedly many stack arrangements, that strategy has limits.

- 7.1 REMARK Stack machines models are often used in practice for running hardware. Here is a ‘Hello World’ program in the PostScript printer language.

```
/Courier           % name the font
20 selectfont     % font size in points, 1/72 of an inch
72 500 moveto     % position the cursor
(Hello world!) show % stroke the text
showpage          % print the page
```

The interpreter pushes `Courier` on the stack, and then on the second line pushes `20` on the stack. It then executes `selectfont`, which pops two things off the stack to set the font name and size. After that it moves the current point and places the text on the page. Finally, it draws that page to paper.

Definition We will extend the definition of Finite State machines by adding a stack. This stack holds tokens from an alphabet, $\Gamma = \{g_0, g_1, \dots\}$. We reserved a character, \perp ,[†] to mark the stack bottom and so we stipulate that it is not a member of Γ . Similarly, we will use a single B to mark the end of the tape input, and so assume it is not a member of the tape alphabet Σ .[‡] And, because the variety of Finite State machine that we will extend is the nondeterministic machine with ε moves, we also assume that the tape alphabet does not contain the character ε .

We start with an example.

- 7.2 EXAMPLE We will give a Pushdown machine that recognizes the language of balanced parentheses, \mathcal{L}_{BAL} , where each `[` has a matching `]` and these matched pairs are nested. Some members of this language are of the form `[n]n`, including `[]` and `[[]]`. But there are other members not of this form, such as `[[][]]` and `[][]`. Precisely stated, $\sigma \in \mathcal{L}_{BAL}$ if it contains the same number of `[`’s as `]`’s and further, no prefix of σ contains more `]`’s than `[`’s.

The Pumping Lemma shows that no Finite State machine recognizes \mathcal{L}_{BAL} . But it is recognized by a Pushdown machine. This machine has two states $Q = \{q_0, q_1\}$, including one accepting state, $F = \{q_1\}$. Its tape alphabet has two characters, $\Sigma = \{[,]\}$. The stack alphabet has only one member, $\Gamma = \{g_0\}$.

[†]Read aloud as “bottom.” [‡]These machines sometimes need to do final work triggered by the end of the input. This doesn’t happen for Finite State machines and so for them we don’t mark the input end in the same way.

The table below gives Δ . Instruction numbers are for ease of reference.

Instruction number	Input	Output
0	$q_0, [, \perp$	$q_0, 'g_0 \perp'$
1	$q_0, [, g_0$	$q_0, 'g_0 g_0'$
2	$q_0,], g_0$	$q_0, ''$
3	$q_0,], \perp$	$q_0, ''$
4	q_0, B, \perp	$q_1, '\perp'$
5	q_0, B, g_0	$q_0, 'g_0'$

Consider line 0. Every computation step begins with the machine popping the top character off the stack. After that, if the machine is in state q_0 , is reading $[$ on the tape, and the stack character is \perp , then this instruction applies. The result is that machine goes into state q_0 (which is not a change) and pushes the two-token string $g_0 \perp$ onto the stack. Because the machine had just popped the stack, this \perp only replaces what was there already, but the g_0 makes the new stack top.

Here is an example computation accepting the string $[[[]][]$.

Step	Configuration
0	$[[[]][]B$ \perp q_0
1	$[[[]][B$ $g_0 \perp$ q_0
2	$[[[]]B$ $g_0 g_0 \perp$ q_0
3	$[[[]B$ $g_0 \perp$ q_0
4	$[[]B$ \perp q_0
5	$[[B$ $g_0 \perp$ q_0
6	$[B$ \perp q_0
7	\perp \perp q_1

After step 1 there are two g_0 's on the stack, which is how the machine remembers that the number of $[$'s it has consumed is two more than the number of $]$'s. At the end it has an empty tape and is in an accepting state, so it accepts the input.

Here is a rejection example, whose initial string does not have balanced parentheses.

Step	Configuration	
0		
1		
2		
3		

At the end, although the tape still has content, the stack is empty. The machine cannot start the next step by popping the top stack character because there is no such character. The computation dies, without accepting the input.

We are ready for the definition.

- 7.3 **DEFINITION** A nondeterministic **Pushdown machine** $\langle Q, q_0, F, \Sigma, \Gamma, \Delta \rangle$ consists of a finite set of **states** $Q = \{q_0, \dots, q_{n-1}\}$, including a **start state** q_0 , a subset $F \subseteq Q$ of **accepting states**, a nonempty **input alphabet** Σ , a nonempty **stack alphabet** Γ , and a **transition function** $\Delta: Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\perp\}) \rightarrow \mathcal{P}(Q \times (\Gamma \cup \{\perp\})^*)$.

A **configuration** C of a machine is a triple listing its present state, present sequence of characters remaining on the tape, and present sequence of characters on the stack. The transition function specifies how the machine moves from configuration to configuration. We can represent this specification either as $\Delta(q_i, t, g) = S$ where S is a set of pairs $\langle q_k, \gamma \rangle$ or as a set of 5-tuple $\langle q_i, t, g, q_k, \gamma \rangle$ **instructions**, where $t \in \Sigma \cup \{\epsilon\}$, $g \in \Gamma \cup \{\perp\}$, and $\gamma \in \Gamma^*$.

The starting configuration C_0 has the machine in state q_0 , with a stack containing only the \perp character, and with the read head at the first character of $\tau_0 \cap B$ where $\tau_0 \in \Sigma^*$ is the **input string**.

As for actions, suppose that the machine's configuration C_s has it in state q_i with the tape head reading t . If there is nothing on the stack, including not even \perp , then the computation dead-ends—there is no configuration C such that $C_s \vdash C$ and the machine does not accept the input string τ_0 .

Otherwise let g be the token at the top of the stack. Suppose that one of the outputs that Δ associates with $\langle q_i, t, g \rangle$ is $\langle q_k, \gamma \rangle$. Then a next configuration, a C_{s+1} so that $C_s \vdash C_{s+1}$, has these properties. (1) If $t \in \Sigma$ then the machine consumes one tape character, necessarily t , goes into state q_k , pops g off the stack, and then pushes the characters of the sequence $\gamma = \langle g_0, \dots, g_m \rangle$ onto the stack in the order that leaves g_0 now at the stack top. (2) If $t = \epsilon$ (that is, t is the single character ' ϵ '), then everything is the same except that the read head does not consume its input character.

A computation ends when the tape is exhausted, including the end-marking B . The machine accepts its initial string τ_0 if at that point it is in a final state.

7.4 EXAMPLE Palindromes that are odd length have a character in the middle that acts as its own reverse. That is, they have the form $\sigma^{\wedge} s \cap \sigma^R$ for $s \in \Sigma$. The language \mathcal{L}_{MM} uses $\sigma \in \{a, b\}^*$ along with the character $s = c$ as a middle marker so that $\mathcal{L}_{MM} = \{\sigma \in \{a, b, c\}^* \mid \sigma = \tau^{\wedge} c \cap \tau^R \text{ for some } \tau \in \{a, b\}^*\}$.

The machine below accepts this language. It has $Q = \{q_0, q_1, q_2\}$, $F = \{q_2\}$, $\Sigma = \{a, b, c\}$, and $\Gamma = \{g0, g1\}$.

<i>Inst</i>	<i>Input</i>	<i>Output</i>	<i>Inst</i>	<i>Input</i>	<i>Output</i>
0	q_0, a, \perp	$q_0, 'g0 \perp'$	8	$q_0, c, g0$	$q_1, 'g0'$
1	q_0, b, \perp	$q_0, 'g1 \perp'$	9	$q_0, c, g1$	$q_1, 'g1'$
2	q_0, c, \perp	$q_1, '\perp'$	10	$q_1, a, g0$	$q_1, ''$
3	q_0, B, \perp	$q_3, '\perp'$	11	$q_1, b, g1$	$q_1, ''$
4	$q_0, a, g0$	$q_0, 'g0 g0'$	12	q_1, B, \perp	$q_2, '\perp'$
5	$q_0, a, g1$	$q_0, 'g0 g1'$			
6	$q_0, b, g0$	$q_0, 'g1 g0'$			
7	$q_0, b, g1$	$q_0, 'g1 g1'$			

In state q_0 , when this machine sees a on the tape then it pushes $g0$ onto the stack, and for b it pushes $g1$. Reading the c switches the machine to state q_2 . In this phase, if it is reading a and the character on top of the stack is $g0$ then the machine consumes the a, pops the $g0$, and goes on. The same happens with b and $g1$. Otherwise, the computation dead-ends. Finally, if the machine reaches the end of the input string at the same moment that it reaches the bottom of the stack then it goes to the accepting state q_2 .

Here is an example computation accepting the input bacab.

<i>Step</i>	<i>Configuration</i>	
0	b a c a b B q_0	\perp
1	a c a b B q_0	$g1 \perp$
2	c a b B q_0	$g0 g1 \perp$
3	a b B q_1	$g0 g1 \perp$
4	b B q_1	$g1 \perp$
5	B q_1	\perp
6		\perp

Our final example makes essential use of guessing, by relying on ε transitions.

- 7.5 EXAMPLE Consider the language of all even-length palindromes over \mathbb{B}^* , $\mathcal{L}_{\text{ELP}} = \{\sigma\sigma^R \mid \sigma \in \mathbb{B}^*\} = \{\varepsilon, 00, 11, 0000, 0110, 1001, 1111, \dots\}$. The Pumping Lemma shows that no Finite State machine recognizes this language. But this nondeterministic Pushdown machine does.

This machine has three states, $Q = \{q_0, q_1, q_2\}$ with $F = \{q_2\}$, as well as $\Sigma = \mathbb{B}$ and $\Gamma = \{g0, g1\}$.

<i>Inst</i>	<i>Input</i>	<i>Output</i>	<i>Inst</i>	<i>Input</i>	<i>Output</i>
0	q_0, θ, \perp	$q_0, 'g0\perp'$	7	$q_0, \varepsilon, g0$	$q_1, 'g0'$
1	$q_0, 1, \perp$	$q_0, 'g1\perp'$	8	$q_0, \varepsilon, g1$	$q_1, 'g1'$
2	q_0, ε, \perp	q_1, \perp	9	$q_1, \theta, g0$	$q_1, ''$
3	$q_0, \theta, g0$	$q_0, 'g0g0'$	10	$q_1, 1, g1$	$q_1, ''$
4	$q_0, \theta, g1$	$q_0, 'g0g1'$	11	q_1, ε, \perp	q_2, \perp
5	$q_0, 1, g0$	$q_0, 'g1g0'$			
6	$q_0, 1, g1$	$q_0, 'g1g1'$			

The machine runs in two phases. Where the input is $\sigma\sigma^R$, the first phase works with σ . If the tape character is θ then the machine pushes the token $g0$ onto the stack, and if it is 1 then the machine pushes $g1$. This is done while in state q_0 .

The second phase works with σ^R . If θ is on the tape and $g0$ tops the stack, or 1 and $g1$, then the machine proceeds. Otherwise there is no matching instruction and the computation branch dies. This is done while in state q_1 .

Without a middle marker how does the machine know when to change from phase one to two, from pushing to popping? It is nondeterministic—it guesses. That happens in lines 7 and 8. The ε character in the input means that the machine can spontaneously transition from q_0 to q_1 .

We will show three example computations. For the first, we exhibit a successful branch of the computation tree.

<i>Step</i>	<i>Configuration</i>	
0	$0\ 1\ 1\ 0$	\perp
1	$1\ 1\ 0$	$g0\ \perp$
2	$1\ 0$	$g1\ g0\ \perp$
3	0	$g0\ \perp$
4		\perp

First note a point about the input. Because this machine can guess, it can guess

whether the input is finished. (Instruction 11 says that if the machine is in state q_1 and the stack has only \perp then the machine can spontaneously transition to q_2 , which is the only accepting state. If this happens after the input string has run out then the computation branch succeeds.) So we can omit the terminating B that we used earlier.

Next is the computation for input 00. The picture below shows the entire computation tree. The ε transitions are drawn vertically (note the difference between the vertical ‘↑’ and the bottom symbol). The machine accepts the input because the highlighted branch ends with an empty tape and in the accepting state q_2 .

7.6 ANIMATION: Computation tree for 00. Next to the ↑'s are instruction numbers.

The third example computation is a rejection. The input is 100, which isn't an even-length palindrome, and none of the branches end both with an empty string and in an accepting state.

7.7 ANIMATION: Computation tree rejecting the input 100.

Our intuition is that Pushdown machines have more power than Finite State machines, in that they have a kind of unbounded read/write memory. The prior examples support that, by showing Pushdown machines that recognize languages that cannot be recognized by any Finite State machine.

We close this section with a number of related results that together make a bigger picture, that the machine models form a linear hierarchy. Full coverage is outside our scope so we will only discuss some of these results without proof.

The first result we have already seen, that deterministic Finite State machines do the same jobs as nondeterministic ones. That also holds for Turing machines,

although we will not consider nondeterministic Turing machines in depth until the final chapter.

Another relevant result, which we won't prove, is that there are things that Turing machines can do but that no Pushdown machine can do. One is to decide membership in the language $\{\sigma^\frown \sigma \mid \sigma \in \mathbb{B}^*\}$, which contains strings such as 1010 and 011011. A Pushdown machine can remember the characters by pushing them onto the stack, and if that machine is nondeterministic then it can guess when the first half of the input ends. But then to check that the second half of the string matches the first it would need to pop the characters off to reverse them, and reversing an arbitrary length string requires being able to write to the tape.

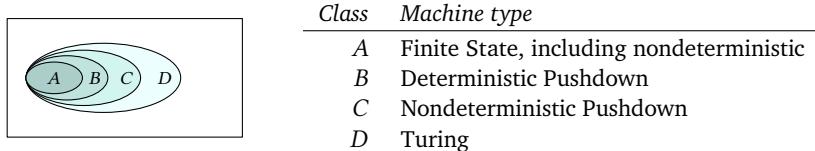
We know that Finite State machines accept Regular languages, and Turing machines accept computable languages. As to nondeterministic Pushdown machines, recall that in the section on Grammars we restricted our attention to production rules where the head consists of a single nonterminal, such as $S \rightarrow aSb$.[†] If a language has a grammar in which all the rules are of this type then it is a **context free language**. Most familiar programming languages are context free, including C, Java, Python, and Racket. We will state but not prove that a language is accepted by some nondeterministic Pushdown machine if and only if it has a context free grammar.

The last result needs deterministic Pushdown machines so we first outline how to define them. In contradistinction to a nondeterministic machine, in a deterministic machine at any step there is exactly one legal move. So to adjust the definition we have for nondeterministic Pushdown machines to one for deterministic ones we eliminate situations where the machine has choices. There are two situations. One is that $\Delta(q_i, t, g)$ is a set and so we will require that in a deterministic machine that set must have exactly one element. The other is evident in the tree diagrams above: nondeterministic machines can have that $\Delta(q_i, \epsilon, g)$ is a nonempty set and also that $\Delta(q_i, t, g)$ is nonempty for $t \neq \epsilon$ (see for instance the prior example's machine in lines 0–2). So we outlaw the possibility that both are nonempty. Example 7.2 and Example 7.4 are both deterministic.[‡]

With that, the last relevant result is that the collection of languages accepted by deterministic Pushdown machines is a proper subset of the collection accepted by nondeterministic Pushdown machines. While we won't prove that, we can give a good idea of why it is true. We have shown that there is a nondeterministic Pushdown machine that accepts the language of even-length palindromes. It uses ϵ moves to guess when to change from pushing to popping. But a deterministic Pushdown machine has recourse to no such tactic. Nor is there a middle marker to rely on. In short, no deterministic Pushdown machine accepts \mathcal{L}_{ELP} . So Pushdown machines are different than Finite State machines and Turing machines—for Pushdown machines, nondeterminism changes what can be done.

[†]An example of a rule where the head is not of that form is $cSb \rightarrow aS$. With this rule we can substitute for S only if it is preceded by c and followed by b . A grammar with rules of this type is called **context sensitive** because substitutions can only be done in a context. [‡]Deterministic Pushdown machines need the end-marker B , which is why we used it for those examples.

The diagram below summarizes our bigger picture. The universal box encloses all languages of bitstrings, all subsets of \mathbb{B}^* . The nested sets enclose those languages recognized by some Finite State machine, etc.



IV.7 Exercises

- ✓ 7.8 Produce a Pushdown machine that does not halt.
- 7.9 Consider the Pushdown machine in Example 7.2.
 - (A) With the input $[][]$, step through the computation as a sequence of \vdash relations.
 - (B) Do the same but with the input $][][]$.
- ✓ 7.10 Produce a Pushdown machine to accept each language over $\Sigma = \{a, b, c\}$.
 - (A) $\{a^n cb^{2n} \mid n \in \mathbb{N}\}$
 - (B) $\{a^n cb^{n-1} \mid n > 0\}$
- ✓ 7.11 Give a Pushdown machine that accepts $\{\theta^\frown \tau^\frown 1 \mid \tau \in \mathbb{B}^*\}$.
- ✓ 7.12 Write a Pushdown machine that accepts $\{a^{2n} \mid n \in \mathbb{N}\}$.
- 7.13 Give a Pushdown machine that accepts $\{a^{2n}b^n \mid n \in \mathbb{N}\}$.
- ✓ 7.14 Example 7.5 discusses the view of a nondeterministic computation as a tree. Draw the tree for that machine these inputs. (A) 0110 (B) 010
- ✓ 7.15 Give a grammar for the language in Example 7.5, the even-length palindromes over \mathbb{B} .
- 7.16 Use the Pumping Lemma to show that the language of even-length palindromes from Example 7.5 is not recognized by any Finite State machine.
- 7.17 Fix an alphabet Σ .
 - (A) Show that the set of Pushdown machines over Σ is countable.
 - (B) Show that the collection of languages accepted by Pushdown machines is countable.
 - (C) Conclude that there are languages that no Pushdown machine accepts.
- 7.18 Use Church's Thesis to argue that any language recognized by a Pushdown machine is recognized by some Turing machine.

EXTRA

IV.A Regular expressions in the wild

Regular expressions are an important tool in practice. Modern programming languages such as Racket and Python include capabilities for extensions to regular

expressions, which we will call **regexes**. These go beyond the small-scale theory examples that we saw earlier.

As an example, imagine a system administrator searching a web server log for the PDF's downloaded from a directory. They might give this command.

```
grep "/linearalgebra/.*\pdf" /var/log/apache2/access.log
```

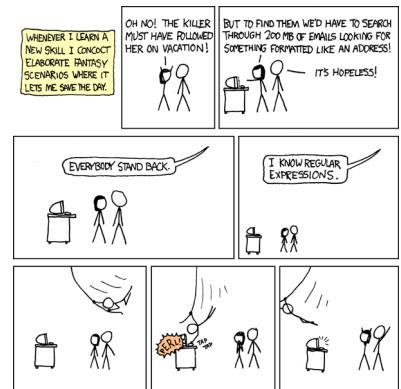
The grep utility program looks through the log file line by line. If a line has a substring matching the regex pattern then grep prints that line.

Different languages have small variations on the syntax of their regexes. Below, we will use Racket regexes. As a prototype to experiment with those regexes, (`(regexp-match? #px "^[A-Z][A-Z][0-9][A-Z][A-Z]\$" "KE1AZ")`) returns `#t`. Note the caret `^` at the start of the string and the dollar sign at the end. They are anchors, making Racket match the entire string from start to finish. They are needed because the most common use case is for programmers to want to find the expression anywhere in the string. Thus for instance, (`(regexp-match? #px "[0-9]" "KE1AZ")`) also returns `#t`, although its expression doesn't account for the letters, because it asks for at least one digit somewhere in KE1AZ. However, here we will use the caret and dollar sign because for the purpose of this explication, they better describe the matching.

The extensions that languages make in going from the theoretical regular expressions that we have seen earlier to in-practice regexes fall into two categories. First come convenience constructs that ease doing something that otherwise would be possible but awkward. Second comes extensions that give capabilities that are just not possible with regular expressions.

Many of the convenience extensions are about the problem of sheer scale: in the theory discussion our alphabets had two or three characters but in practice an alphabet must include at least ASCII's printable characters: `a–z`, `A–Z`, `0–9`, space, tab, period, dash, exclamation point, percent sign, dollar sign, open and closed parenthesis, open and closed curly braces, etc. These days it may even contain all of Unicode's more than one hundred thousand characters. We need manageable ways to describe such large sets.

Consider matching a digit. The regular expression `(0|1|2|3|4|5|6|7|8|9)` works, but is too verbose for an often-needed list. One abbreviation that modern languages allow is `[0123456789]`, omitting the pipe characters and using square brackets, which in regexes are metacharacters. Or, because the digit characters are contiguous in the character set,[†] we can shorten it further to `[0-9]`. Along the same lines, `[A-Za-z]` matches a singleton English letter.



Courtesy xkcd.com

[†]The digits 0 through 9 are contiguous in both ASCII and Unicode.

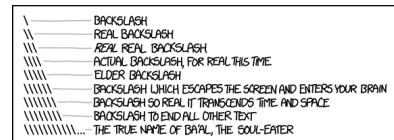
To invert the set of matched characters, put a caret ‘^’ as the first thing inside the bracket (and note that it is a metacharacter). Thus, `[^0-9]` matches a non-digit and `[^A-Za-z]` matches a character that is not an ASCII letter.

The most common lists have short abbreviations. Another abbreviation for the digits is `\d`. Use `\D` for the ASCII non-digits, `\s` for the whitespace characters (space, tab, newline, formfeed, and line return) and `\S` for ASCII characters that are non-whitespace. Cover the alphanumeric characters (upper and lower case ASCII letters, digits, and underscore) with `\w` and cover the ASCII non-alphanumeric characters with `\W`. And—the big kahuna—the dot ‘.’ is a metacharacter that matches any member of the alphabet at all.[†]

- A.1 EXAMPLE Canadian postal codes have seven characters: the fourth is a space, the first, third, and sixth are letters, and the others are digits. The regular expression `[a-zA-Z]\d[a-zA-Z] \d[a-zA-Z]\d` describes them.
- A.2 EXAMPLE Dates are often given in the ‘dd/mm/yy’ format. This matches: `\d\d/\d\d/\d\d/\d\d`.
- A.3 EXAMPLE In the twelve hour time format some typical times strings are ‘8:05 am’ or ‘10:15 pm’. You could use this (note the empty string at the start).

`(|0|1)\d:\d\d\s(am|pm)`

To match a metacharacter, prefix it with a backslash, ‘\’. Thus, to look for the string ‘(Note’ put a backslash before the open parentheses, `\(Note`. Similarly, `\|` matches a pipe and `\[` matches an open square bracket. Match backslash itself with `\\\`. This is called **escaping** the metacharacter. The scheme described above for representing lists with `\d`, `\D`, etc. is an extension of escaping.



Courtesy xkcd.com

Operator precedence is: repetition binds most strongly, then concatenation, and then alternation (force different meanings with parentheses). Thus, `ab*` is equivalent to `a(b*)`, and `ab|cd` is equivalent to `(ab)|(cd)`.

Quantifiers In the theoretical cases we saw earlier, to match ‘at most one a’ we used `\epsilon|a`. In practice we can write something like `(|a)`, as we did above for the twelve hour times. But depicting the empty string by just putting nothing there can be confusing. Modern languages make question mark a metacharacter and allow you to write `a?` for ‘at most one a’.

For ‘at least one a’ modern languages use `a+`, so the plus sign is another metacharacter. More generally, we often want to specify quantities. For instance, to match five a’s regexes use the curly braces as metacharacters, with `a{5}`. Match between two and five of them with `a{2,5}` and match at least two with `a{2,}`. Thus, `a+` is shorthand for `a{1,}`.

[†]Programming languages in practice by default have the dot match any character except newline. In addition, these languages have a way to make it also match newline.

As earlier, to match any of these metacharacters you must escape them. For instance, To be or not to be\? matches the famous question.

Cookbook All of the extensions to regular expressions that we are seeing are driven by the desires of working programmers. Here is a pile of examples showing them accomplishing practical work, matching things you'd want to match.

- A.4 EXAMPLE US postal codes, called ZIP codes, are five digits. Match them with \d{5}.
- A.5 EXAMPLE North American phone numbers match \d{3} \d{3}-\d{4}.
- A.6 EXAMPLE The regex (-|\+)?\d+ matches an integer, positive or negative. The question mark makes the sign optional. The plus sign makes sure there is at least one digit.
- A.7 EXAMPLE A natural number represented in hexadecimal can contain the usual digits, along with the additional characters ‘a’ through ‘f’ (sometimes capitalized). Programmers often prefix such a representation with 0x, so the regex is (0x)?[a-fA-F0-9]+.
- A.8 EXAMPLE A C language identifier begins with an ASCII letter or underscore and then can have arbitrarily many more letters, digits, or underscores: [a-zA-Z_]\w*.
- A.9 EXAMPLE Match a user name of between three and twelve letters, digits, underscores, or periods with [\w\.]{3,12}. Match a password that is at least eight characters long with .{8,}.
- A.10 EXAMPLE The International Standards Organization date format calls for dates like ‘yyyy-mm-dd HH:MM:SS’ (along with many other variants). The regex \d{4}-\d{2}-\d{2} (\d{2}:\d{2}(:\d{2})?)? will match them.
- A.11 EXAMPLE Match the text inside a single set of parentheses with \([^\(\)]*\)\).
- A.12 EXAMPLE We next match a URL, a web address such as https://hefferon.net/computation. This regex is more intricate than prior ones. It is based on breaking URL's into three parts: a scheme such as ‘http’ along with a colon and two forward slashes, a host such as hefferon.net and a slash, and then a path such as computing (the standard also allows a trailing query string but this regex does not handle that).

```
(https?|ftp)://([^\s/?\.\#]+\.\?){0,3}[^\s/?\.\#]+(/[^\\s]*/?)?
```

Notice the question mark in https?, so that the scheme can be http or https. Notice also that the host part, consists of between one and four fields separated by periods. We allow almost any character in those fields, except for a space, a question mark, a period or a hash. At the end comes the path.

But wait! there’s more! We have already noted that you can match the start of a line and end of line with the metacharacters caret ‘^’ and dollar sign ‘\$’.

- A.13 EXAMPLE Match lines starting with ‘Theorem’ using `^Theorem`. Match lines ending with `end{equation*}` using `end{equation*}$`.

Regex engines in modern languages let you specify that the match is case insensitive, although they differ in the syntax that you use to achieve this.

- A.14 EXAMPLE The web document language HTML document tag for an image, such as ``, uses either of the keys `src` or `img` to give the name of the file containing the image. Those strings can be in upper case or lower case, or any mix. Racket uses a ‘`?i:`’ syntax to mark part of the regex as insensitive: `\s+(?i:(img|src))=`. (Note also the double backslash, which is how Racket escapes the backslash.)

Beyond convenience The regular expression engines that come with recent programming languages have capabilities beyond matching only those languages that are recognized by Finite State machines.

- A.15 EXAMPLE The language HTML uses tags such as `boldface text` and `<i>italicized text</i>`. Matching any one tag is straightforward, for instance `[^<]*`. But for a single expression that matches them all, you would seem to have to do each as a separate case and then combine cases with a pipe. However, instead we can have the system remember what it finds at the start and look for that again at the end. Thus, the regex `<([^\>]+)>.*</\1>` matches HTML tags like the ones given. Its second character is an open parenthesis, and the `\1` refers to everything between that open parenthesis and the matching close parenthesis (and, that is not a typo; Racket’s syntax calls for double backslashes). As is hinted by the `1`, you can also have a second match with `\2`, etc.

That is a **back reference**. It is very convenient. However, it gives regexes more power than the theoretical regular expressions that we studied earlier.

- A.16 EXAMPLE This is the language of **square strings** over $\Sigma = \{a, b\}$.

$$\mathcal{L} = \{\sigma \in \Sigma^* \mid \sigma = \tau^\wedge \tau \text{ for some } \tau \in \Sigma^*\}$$

Some members are `aabaab`, `baaabaaa`, and `aa`. The Pumping Lemma shows that the language of squares is not regular; see Exercise A.36. Describe this language with the regex `(.+)\1`; note the back-reference.

- A.17 EXAMPLE Another language that the Pumping Lemma shows cannot be represented using regular expressions, but that can be described with regexes is the language of numbers that are nonprime, represented in unary, $\mathcal{L} = \{1^n \mid n \text{ is not prime}\}$. It is described by the regex `^1?$_|^^(11+?)\\1+$`. A brief explanation: the `^1?$` matches a string that is either zero-many or one-many 1’s. The `^(11+?)\\1+$` matches a group of 1’s repeated one or more times. Being able to divide the number of 1’s into some number of subgroups is what characterizes a unary number as composite, that is, not prime.

Tradeoffs Regexes are powerful tools. But they come with downsides.

For instance, the regular expression for twelve hour time from Example A.3 ($\varepsilon|0|1)\backslash d:\backslash d\backslash d\s(am|pm)$) does indeed match ‘8:05 am’ and ‘10:15 pm’ but it falls short in some respects. One is that it requires *am* or *pm* at the end, but times are often given without them. We could change the ending to ($\varepsilon|\s am|\s pm$).

Another issue is that it also matches some strings that you don't want, such as 13:00 am or 9:61 pm. We can solve this as with the prior paragraph, by listing the cases.[†] `(01|02|...|11|12):(01|02|...|59|60)(\s am|\s pm)`. This is like the prior patch in that it fixes the issue but at a cost of complexity, since it amounts to a list of allowed substrings. Regexes have a tendency to grow, to accrete subcases like this.

Another example is the Canadian postal expression in Example A.1. Not every matching string has a corresponding physical post office—for one thing, no valid codes begin with Z. And US ZIP codes work the same way; there are fewer than 50 000 assigned ZIP codes, so many five digits strings are not in use. Changing the regexes to cover only those codes actually in use would make them just lists of strings, which would change frequently.

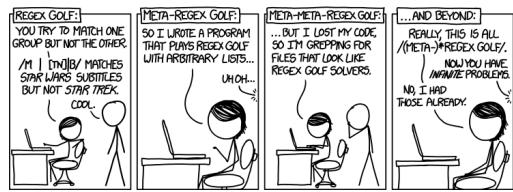
REGEX GOLF:
YOU TRY TO MATCH ONE GROUP BUT NOT THE OTHER.
M | FINALLY MATCHES STAR WARS SURNAMES BUT NOT STAR TREK.
COOL.

META-REGEX GOLF:
SO I WROTE A PROGRAM THAT PLAYS REGEX GOLF WITH ARBITRARY LISTS...

META-META-REGEX GOLF:
...BUT I LOST MY CODE SO I'M GREPPING FOR FILES THAT LOOK LIKE REGEX GOLF SOLVERS.
UH OH...

REALITY:
INSTEAD
NO I HAD THOSE ALL

Courtesy xkcd.com



Courtesy xkcd.com

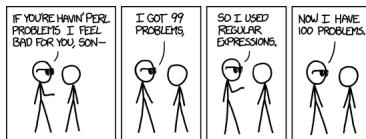
The canonical example of this is the regex describing the official standard for valid email addresses. We show here just five lines out of its 81 but that's enough to make the point about its complexity.

And, even if you do have an address that fits the standard, you don't know if there is an email server listening at that address. In practice, people often use the regex `\S+@\S+` as a sanity check, for instance on a web form that expects users to input an email address.

At this point regexes may be starting to seem a less like a fast and neat problem-solver and a little more like a potential development and maintenance problem. The full story is that sometimes a regex is just what you need for a quick job, and sometimes they are good for more complex tasks also. But some of the time the cost of complexity outweighs the gain in expressiveness. This power/complexity tradeoff is often referred to online by citing this quote from J Zawinski.

The notion that [regexes] are the solution to all problems is . . . braindead. . . . Some people, when confronted with a problem, think "I know, I'll use regular expressions."

Now they have two problems.



Courtesy xkcd.com

[†] Some substrings are elided so it fits in the margins.

IV.A Exercises

- ✓ A.18 Which of the strings matches the regex $ab+c$? (A) abc (B) ac (C) abbb (D) bbc
- A.19 Which of the strings matches the regex $[a-z]^+[\.\.\?!\?]$? (A) battle! (B) Hot (C) green (D) swamping. (E) jump up. (F) undulate? (G) is..?
- ✓ A.20 Give a regex for each. (A) Match a string that has ab followed by zero or more c's, (B) ab followed by one or more c's, (C) ab followed by zero or one c, (D) ab followed by two c's, (E) ab followed by between two and five c's, (F) ab followed by two or more c's, (G) a followed by either b or c.
- ✓ A.21 Give a regex to accept a string for each description.
 - (A) Containing the substring abe.
 - (B) Containing only upper and lower case ASCII letters and digits.
 - (C) Containing a string of between one and three digits.
- A.22 Give a regex to accept a string for each description. Take the English vowels to be a, e, i, o, and u.
 - (A) Starting with a vowel and containing the substring bc.
 - (B) Starting with a vowel and containing the substring abc.
 - (C) Containing the five vowels in ascending order.
 - (D) Containing the five vowels.
- A.23 Give a regex matching strings that contain an open square bracket and an open curly brace.
- ✓ A.24 Every lot of land in New York City is denoted by a string of digits called BBL, for Borough (one digit), Block (five digits), and Lot (four digits). Give a regex.
- ✓ A.25 Example A.5 gives a regex for North American phone numbers.
 - (A) They are sometimes written with parentheses around the area code. Extend the regex to cover this case.
 - (B) Sometimes phone numbers do not include the area code. Extend to cover this also.
- A.26 Most operating systems come with file that has a list of words, for spell-checking, etc. For instance, on Linux it may be at `/usr/share/dict/words`. Use that file to find how many words fit the criteria.
 - (A) contains the letter a
 - (B) starts with A
 - (C) contains a or A
 - (D) contains X
 - (E) contains x or X
 - (F) contains the string st
 - (G) contains the string ing
 - (H) contains an a, and later a b
 - (I) contains none of the usual vowels a, e, i, o or u
 - (J) contains all the usual vowels
 - (K) contains all the usual vowels, in ascending order

- ✓ A.27 Give a regex to accept time in a 24 hour format. It should match times of the form ‘hh:mm:ss.sss’ or ‘hh:mm:ss’ or ‘hh:mm’ or ‘hh’.
- A.28 Give a regex describing a floating point number.
- ✓ A.29 Give a suitable regex.
 - (A) All Visa card numbers start with a 4. New cards have 16 digits. Old cards have 13.
 - (B) MasterCard numbers either start with 51 through 55, or with the numbers 2221 through 2720. All have 16 digits.
 - (C) American Express card numbers start with 34 or 37 and have 15 digits.
- ✓ A.30 Postal codes in the United Kingdom have six possible formats. They are:
 - (i) A11 1AA, (ii) A1 1AA, (iii) A1A 1AA, (iv) AA11 1AA, (v) AA1 1AA, and (vi) AA1A 1AA, where A stands for a capital ASCII letter and 1 stands for a digit.
 - (A) Give a regex.
 - (B) Shorten it.
- ✓ A.31 You are stuck on a crossword puzzle. You know that the first letter (of eight) is an g, the third is an n and the seventh is an i. You have access to a file that contains all English words, each on its own line. Give a suitable regex.
- A.32 In the Tradeoffs discussion, we change the ending to $(\epsilon \mid \text{ }\backslash s \text{ am} \mid \text{ }\backslash s \text{ pm})$. Why not $\backslash s(\epsilon \mid \text{am} \mid \text{pm})$, which factors out the whitespace?
- A.33 Imagine that you decide to avoid regexes but still want to do the sanity check for email addresses discussed above, of accepting the string if and only if it consists of a nonempty string of characters, followed by @, followed by a nonempty string of characters. Implement that as a routine in your favorite language.
- A.34 Give a regex that matches no string.

- ✓ A.35 The Roman numerals from grade school use the letters I, V, X, L, C, D, and M to represent 1, 5, 10, 50, 100, 500, and 1000. They are written in descending order of magnitude, from M to I, and are written greedily so that we don’t write six I’s but rather VI. Thus, the date written on the book held by the Statue of Liberty is MDCCCLXXVI, for 1776. Further, we replace IIII with IV, and replace VIII with IX. Give a regular expression for valid Roman numerals less than 5000.

A.36 Example A.16 says that the language of square strings over $\Sigma = \{a, b\}$

$$\mathcal{L} = \{\sigma \in \Sigma^* \mid \sigma = \tau^\wedge \tau \text{ for some } \tau \in \Sigma^*\}$$

is not regular. Verify that.

A.37 Consider $\mathcal{L} = \{0^n 10^n \mid n > 0\}$. (A) Show that it is not regular. (B) Find a regex.

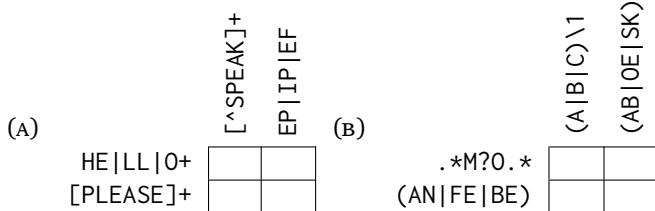
A.38 In **regex golf** you are given two lists and must produce a regex that matches all the words in the first list but none of the words in the second. The ‘golf’ aspect is that the person who finds the shortest regex, the one with the fewest characters, wins. Try these: accept the words in the first list and not the words in the second.

(A) Accept: Arthur, Ester, le Seur, Silverter

Do not accept: Bruble, Jones, Pappas, Trent, Zikle

- (b) Accept: alight, bright, kite, mite, tickle
 Do not accept: buffing, curt, penny, tart
- (c) Accept: afoot, catfoot, dogfoot, fanfoot, foody, foolery, foolish, fooster, footage, foothot, footle, footpad, footway, hotfoot, jawfoot, mafoo, nonfood, padfoot, prefool, sfoot, unfool
 Do not accept: Atlas, Aymoro, Iberic, Mahran, Ormazd, Silipan, altared, chandoo, crenel, crooked, fardo, folksy, forest, hebamic, idgah, manlike, marly, palazzi, sixfold, tarrock, unfold

A.39 In a **regex crossword** each row and column has a regex. You have to find strings for those rows and columns that meet the constraints.

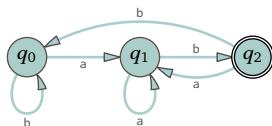


EXTRA

IV.B The Myhill-Nerode Theorem

We defined regular languages in terms of Finite State machines. Here we will give a characterization that does not depend on that.

This deterministic Finite State machine accepts strings that end in ab.

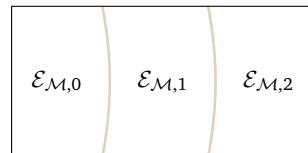


Consider other strings over $\Sigma = \{a, b\}$, not just the accepted ones, and see where they bring the machine.

Input string σ	ϵ	a	b	aa	ab	ba	bb	aaa	aab	aba	abb
Ending state $\hat{\Delta}(\sigma)$	q_0	q_1	q_0	q_1	q_2	q_1	q_0	q_1	q_2	q_1	q_0

The collection of all strings Σ^* , pictured below, breaks into three sets, those that bring the machine to q_0 , those that bring the machine to q_1 , and those that bring the machine to q_2 .

$$\begin{aligned}\mathcal{E}_{M,0} &= \{ \epsilon, b, bb, abb, \dots \} \\ \mathcal{E}_{M,1} &= \{ a, aa, ba, aba, \dots \} \\ \mathcal{E}_{M,2} &= \{ ab, aab, abb, \dots \}\end{aligned}$$



B.1 **DEFINITION** Let \mathcal{M} be a Finite State machine with alphabet Σ . Two strings $\sigma_0, \sigma_1 \in \Sigma^*$ are **\mathcal{M} -related** if starting the machine with input σ_0 ends with it in the same state as does starting the machine with input σ_1 .

B.2 **LEMMA** The binary relation of \mathcal{M} -related is an equivalence and so partitions the collection of all strings Σ^* into equivalence classes.

Proof We must show that the relation is reflexive, symmetric, and transitive. Reflexivity, that any input string σ brings the machine to the same state as itself, is obvious. So is symmetry, that if σ_0 brings the machine to the same state as σ_1 then σ_1 brings it to the same state as σ_0 . Transitivity is straightforward: if σ_0 brings \mathcal{M} to the same state as σ_1 , and σ_1 brings it to the same state as σ_2 , then σ_0 brings it to the same state as σ_2 . \square

So a machine gives rise to a partition. Does it go the other way—if we have a partition of a language, is there an associated machine?

This converse is ruled out by a counting argument. Consider the alphabet $\mathbb{B} = \{0, 1\}$. There are uncountably many partitions of the language \mathbb{B}^* but there are only countably many Finite State machines with that alphabet. Thus it is not the case that for any partition there is an associated machine.

But some further reflection on the above example gives a limit on which partitions arise due to the action of Finite State machines. The \mathcal{M} relation gives rise to a partition where there is one class for each state, and thus the partition has only finitely many classes. We will show that if a partition of a language has finitely many classes then there is an associated machine. Further, the argument below is constructive, meaning that it shows us how to make the machine from the partition.

B.3 **DEFINITION** Let \mathcal{L} be a language over Σ . Two strings $\sigma, \hat{\sigma} \in \Sigma^*$ are **\mathcal{L} -related** or **\mathcal{L} -indistinguishable**, denoted $\sigma \sim_{\mathcal{L}} \hat{\sigma}$, when for every suffix $\tau \in \Sigma^*$ we have $\sigma^\wedge \tau \in \mathcal{L}$ if and only if $\hat{\sigma}^\wedge \tau \in \mathcal{L}$. Otherwise, the two strings are **\mathcal{L} -distinguishable**.

Said another way, the two strings σ and $\hat{\sigma}$ can be \mathcal{L} -distinguished when there is a suffix τ that separates them: of the two $\sigma^\wedge \tau$ and $\hat{\sigma}^\wedge \tau$, one is an element of \mathcal{L} while the other is not.

B.4 **LEMMA** For any language \mathcal{L} , the binary relation $\sim_{\mathcal{L}}$ is an equivalence, and thus gives rise to a partition of all strings.

Proof Reflexivity, that $\sigma \sim_{\mathcal{L}} \sigma$, is trivial. So is symmetry, that $\sigma_0 \sim_{\mathcal{L}} \sigma_1$ implies $\sigma_1 \sim_{\mathcal{L}} \sigma_0$. For transitivity suppose $\sigma_0 \sim_{\mathcal{L}} \sigma_1$ and $\sigma_1 \sim_{\mathcal{L}} \sigma_2$. If $\sigma_0^\wedge \tau \in \mathcal{L}$ then by the first supposition $\sigma_1^\wedge \tau \in \mathcal{L}$, and the second supposition in turn gives $\sigma_2^\wedge \tau \in \mathcal{L}$. Similarly $\sigma_0^\wedge \tau \notin \mathcal{L}$ implies that $\sigma_2^\wedge \tau \notin \mathcal{L}$. Thus $\sigma_0 \sim_{\mathcal{L}} \sigma_2$. \square

B.5 **EXAMPLE** Let \mathcal{L} be the set $\{\sigma \in \mathbb{B}^* \mid \sigma \text{ has an even number of } 1's\}$. We can find the parts of the partition. If two strings σ_0, σ_1 both have an even number of 1's then they are \mathcal{L} -related. That's because for any $\tau \in \mathbb{B}^*$, if τ has an even number of 1's then $\sigma_0^\wedge \tau \in \mathcal{L}$ and $\sigma_1^\wedge \tau \in \mathcal{L}$, while if τ has an odd number of 1's then the

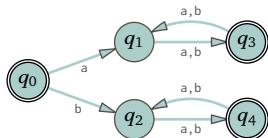
concatenations will not be members of \mathcal{L} . Similarly, two strings both have an odd number of 1's then they are \mathcal{L} -related. So the relationship $\sim_{\mathcal{L}}$ gives rise to this partition of \mathbb{B}^* .

$$\mathcal{E}_{\mathcal{L},0} = \{\varepsilon, 0, 00, 11, 000, 011, 101, 110, \dots\} \quad \mathcal{E}_{\mathcal{L},1} = \{1, 01, 10, 001, 010, \dots\}$$

- B.6 EXAMPLE Let \mathcal{L} be $\{\sigma \in \{a, b\}^* \mid \sigma \text{ has the same number of } a\text{'s as } b\text{'s}\}$. Then two members of \mathcal{L} , two strings $\sigma_0, \sigma_1 \in \Sigma^*$ with the same number of a 's as b 's, are \mathcal{L} -related. This is because for any suffix τ , the string $\sigma_0 \cap \tau$ is an element of \mathcal{L} if and only if $\sigma_1 \cap \tau$ is an element of \mathcal{L} , which happens if and only if τ has the same number of a 's as b 's.

Similarly, two strings σ_0, σ_1 such that the number of a 's is one more than the number of b 's are \mathcal{L} -related because for any suffix τ , the string $\sigma_0 \cap \tau$ is an element of \mathcal{L} if and only if $\sigma_1 \cap \tau$ is an element of \mathcal{L} , namely if and only if τ has one fewer a than b . Following this reasoning, $\sim_{\mathcal{L}}$ partitions $\{a, b\}^*$ into the infinitely many parts $\mathcal{E}_{\mathcal{L},i} = \{\sigma \in \{a, b\}^* \mid \text{the number of } a\text{'s minus the number of } b\text{'s equals } i\}$, where $i \in \mathbb{Z}$.

- B.7 EXAMPLE This machine \mathcal{M} accepts $\mathcal{L} = \{\sigma \in \{a, b\}^* \mid \sigma \text{ has even length}\}$.



We will compare the partitions induced by the two relations above. The \mathcal{M} -related relation breaks $\{a, b\}^*$ into five parts, one for each state (since each state in \mathcal{M} is reachable).

$$\mathcal{E}_{\mathcal{M},0} = \{\varepsilon\}$$

$$\mathcal{E}_{\mathcal{M},1} = \{a, aaa, aab, aba, abb, aaaaa, aaaab, \dots\}$$

$$\mathcal{E}_{\mathcal{M},2} = \{b, baa, bab, bba, bbb, baaaa, baaab, \dots\}$$

$$\mathcal{E}_{\mathcal{M},3} = \{aa, ab, aaaa, aaab, aaba, aabb, abaa, abab, abba, abbb, aaaaaa, \dots\}$$

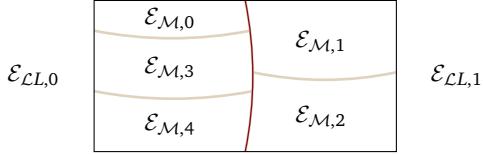
$$\mathcal{E}_{\mathcal{M},4} = \{ba, bb, baaa, baab, baba, babb, bbaa, bbab, bbba, bbbb, baaaaa, \dots\}$$

The \mathcal{L} -related relation breaks $\{a, b\}^*$ into two parts.

$$\mathcal{E}_{\mathcal{L},0} = \{\sigma \mid \sigma \text{ has even length}\} \quad \mathcal{E}_{\mathcal{L},1} = \{\sigma \mid \sigma \text{ has odd length}\}$$

Verify this by noting that if two strings are in $\mathcal{E}_{\mathcal{L},0}$ then adding a suffix τ will result in a string that is a member of \mathcal{L} if and only if the length of τ is even, and the same reasoning holds for $\mathcal{E}_{\mathcal{L},1}$ and odd-length τ 's.

The sketch below shows the universe of strings $\{a, b\}^*$, partitioned in two ways. There are two \mathcal{L} -related parts, the left and right halves. The five \mathcal{M} -related parts are subsets of the \mathcal{L} -related parts.



That is, the \mathcal{M} -related partition is finer than the \mathcal{L} -related partition ('fine' in the sense that sand is finer than gravel).

- B.8 **LEMMA** Let \mathcal{M} be a Finite State machine that recognizes \mathcal{L} . If two strings are \mathcal{M} -related then they are \mathcal{L} -related.

Proof Assume that σ_0 and σ_1 are \mathcal{M} -related, so that starting \mathcal{M} with input σ_0 causes it to end in the same state as starting it with input σ_1 . Thus for any suffix τ , giving \mathcal{M} the input $\sigma_0 \hat{\cdot} \tau$ causes it to end in the same state as does the input $\sigma_1 \hat{\cdot} \tau$. In particular, $\sigma_0 \hat{\cdot} \tau$ takes \mathcal{M} to a final state if and only if $\sigma_1 \hat{\cdot} \tau$ does. So the two strings are \mathcal{L} -related. \square

- B.9 **LEMMA** Let \mathcal{L} be a language. (1) If two strings σ_0, σ_1 are \mathcal{L} -related, $\sigma_0 \sim_{\mathcal{L}} \sigma_1$, then adjoining a common extension β gives strings that are also \mathcal{L} -related, $\sigma_0 \hat{\cdot} \beta \sim_{\mathcal{L}} \sigma_1 \hat{\cdot} \beta$. (2) If one member of a part $\sigma_0 \in \mathcal{E}_{\mathcal{L},i}$ is an element of \mathcal{L} then every member of that part $\sigma_1 \in \mathcal{E}_{\mathcal{L},i}$ is also an element of \mathcal{L} .

Proof For the first item, start with two strings σ_0, σ_1 that are \mathcal{L} -related. By definition, no extension τ will \mathcal{L} -distinguish the two—it is not the case that one of $\sigma_0 \hat{\cdot} \tau, \sigma_1 \hat{\cdot} \tau$ is in \mathcal{L} while the other is not. Taking $\beta \hat{\cdot} \hat{\tau}$ for τ gives that for the two strings $\sigma_0 \hat{\cdot} \beta$ and $\sigma_1 \hat{\cdot} \beta$, no extension $\hat{\tau}$ will \mathcal{L} -distinguish the two. So they are \mathcal{L} -related.

The second item is: if $\sigma_0 \sim_{\mathcal{L}} \sigma_1$ and $\sigma_0 \in \mathcal{L}$ but $\sigma_1 \notin \mathcal{L}$ then they are distinguished by the empty string, which contradicts that they are \mathcal{L} -related. \square

- B.10 **EXAMPLE** We will milk Example B.7 for another observation. Take a string σ from $\mathcal{E}_{\mathcal{M},1}$ and append an a. The result $\sigma \hat{\cdot} a$ is a member of $\mathcal{E}_{\mathcal{M},3}$, simply because if the machine is in state q_1 and it receives an a then it moves to state q_3 . Likewise, if $\sigma \in \mathcal{E}_{\mathcal{M},4}$, then $\sigma \hat{\cdot} b$ is a member of $\mathcal{E}_{\mathcal{M},2}$. If adding the alphabet character $x \in \Sigma$ to one string σ from $\mathcal{E}_{\mathcal{L},i}$ results in a string $\sigma \hat{\cdot} x$ from $\mathcal{E}_{\mathcal{L},j}$ then the same will happen for any string from $\mathcal{E}_{\mathcal{L},i}$.

In this example we see that's true because the $\mathcal{E}_{\mathcal{M}}$'s are contained in the $\mathcal{E}_{\mathcal{L}}$'s. The key step of the next result is to find it even in a context where there is no machine.

- B.11 **THEOREM (MYHILL-NERODE)** A language \mathcal{L} is regular if and only if the relation $\sim_{\mathcal{L}}$ has only finitely many equivalence classes.

Proof One direction is easy. Suppose that \mathcal{L} is a regular language. Then it is recognized by a Finite State machine \mathcal{M} . By Lemma B.8 the number of elements in the partition induced by $\sim_{\mathcal{L}}$ is finite because the number of elements in the partition associated with being \mathcal{M} -related is finite, as there is one part for each of

\mathcal{M} 's reachable states.

For the other direction suppose that the number of elements in the partition associated with being \mathcal{L} -related is finite. We will show that \mathcal{L} is regular by producing a Finite State machine that recognizes \mathcal{L} .

The machine's states are the partition's elements, the $\mathcal{E}_{\mathcal{L},i}$'s. That is, s_i is $\mathcal{E}_{\mathcal{L},i}$. The start state is the part containing the empty string ε . A state is final if that part contains strings from the language \mathcal{L} (Lemma B.9 (2) says that each part contains either no strings from \mathcal{L} or consists entirely of strings from \mathcal{L}).

The transition function is: for any state $s_i = \mathcal{E}_{\mathcal{L},i}$ and alphabet element x , compute the next state $\Delta(s_i, x)$ by starting with any string in that part $\sigma \in \mathcal{E}_{\mathcal{L},i}$, appending the character to get a new string $\hat{\sigma} = \sigma \hat{x}$, and then finding the part containing that string, the $\mathcal{E}_{\mathcal{L},j}$ such that $\hat{\sigma} \in \mathcal{E}_{\mathcal{L},j}$. Then $\Delta(s_i, x) = s_j$. (Here is an equivalent way to describe Δ . For the part containing σ we can use the notation $[\![\sigma]\!]$. Then our definition of the transition function becomes $\Delta([\![\sigma]\!], x) = [\![\sigma \hat{x}]\!]$.)

We must verify that this transition function is well-defined. That is, the definition of $\Delta(s_i, x)$ as given potentially depends on which string σ you choose from $s_i = \mathcal{E}_{\mathcal{L},i}$. We must check that choosing a different string cannot lead to a different resulting part. This follows from (1) in Lemma B.9: take two starting strings from the same part $\sigma_0, \sigma_1 \in \mathcal{E}_{\mathcal{L},i}$ and make a common extension by the one-element string $\beta = \langle x \rangle$ so the results are in the same part $\sigma_0 \sim_{\mathcal{L}} \sigma_1$.

Finally, we must verify that the language recognized by this machine is \mathcal{L} . For any string $\sigma \in \Sigma^*$, starting this machine with σ as input will cause the machine to end in the partition containing σ ; this is what the prior paragraph says. This string will be accepted by this machine if and only if $\sigma \in \mathcal{L}$. \square

The Myhill–Nerode theorem gives a necessary and sufficient condition for a language to be regular. So it is in some sense the “right” way to think about what makes a language regular.

For instance, to show that some languages are not regular we have used the Pumping Lemma, but we have no reason to suppose that it works in all cases (in fact, it does not). However, the Myhill–Nerode theorem says that we can always depend on the number of equivalence classes to distinguish regular languages from non-regular ones.

IV.B Exercises

- ✓ B.12 Find the \mathcal{L} equivalence classes for each regular set. The alphabet is $\Sigma = \{a, b\}$.
 - (A) $\mathcal{L}_0 = \{a^n b \mid n \in \mathbb{N}\}$
 - (B) $\mathcal{L}_1 = \{a^2 b^n \mid n \in \mathbb{N}\}$
- ✓ B.13 For each language describe the \mathcal{L} equivalence classes. The alphabet is \mathbb{B} .
 - (A) The set of strings ending in 01
 - (B) The set of strings where every 0 is immediately followed by two 1's
 - (C) The set of strings with the substring 0110

- (D) The set of strings without the substring 0110
- ✓ B.14 The language of palindromes $\mathcal{L} = \{\sigma \in a, b^* \mid \sigma^R = \sigma\}$ is not regular. Find infinitely many \mathcal{L} equivalence classes.
- ✓ B.15 Show that the language $\mathcal{L} = \{a^n b^n \mid n \in \mathbb{N}\}$ is not regular by using the Myhill-Nerode Theorem.

Part Three

Computational Complexity



CHAPTER

V Computational Complexity

Earlier, we asked what can be done with a mechanism at all. This mirrors the subject's history: when the Theory of Computing began there were no physical computers. Researchers were driven by considerations such as the *Entscheidungsproblem*. The subject was interesting, the questions compelling, and there were plenty of problems, but the initial phase had a theory-driven feel.

A natural next step is to look to do jobs efficiently. When physical computers became widely available, that's exactly what happened. Today, the Theory of Computing has incorporated many questions that at least originate in applied fields, and that need answers that are feasible.

We will review how we determine the practicality of algorithms, the order of growth of functions. Then we will see a collection of the kinds of problems that drive the field today. By the end of this chapter we will be at the research frontier and we will state some things without proof, as well as discuss some things about which we are not sure. In particular, we will consider the celebrated question of P versus NP .

SECTION

V.1 Big \mathcal{O}

We begin by reviewing the definition of the order of growth of functions. We will study this because it measures how algorithms consume computational resources.

First, an anecdote. Here is a grade school multiplication.

$$\begin{array}{r} 678 \\ \times 42 \\ \hline 1356 \\ 2712 \\ \hline 28476 \end{array}$$

The algorithm combines each digit of the multiplier 42 with each digit of the multiplicand 678, in a nested loop. A person could sensibly feel that this is the right way to compute multiplication—indeed, the only reasonable way—and that in general, to multiply two n digit numbers requires about n^2 -many operations.

IMAGE: Striders can walk on water because they are five orders of magnitude smaller than us. This change of scale changes the world—bugs see surface tension as more important than gravity. Similarly, finding an algorithm that changes the time that it takes to solve a problem from n^2 to $n \cdot \lg n$ can make something easy that was previously not practical.

In 1960, A Kolmogorov organized a seminar at Moscow State University aimed at proving this. But before the seminar's second meeting one of the students, A Karatsuba, discovered that it is false. He produced a clever algorithm that used only $n^{\lg(3)} \approx n^{1.585}$ operations. At the next meeting Kolmogorov explained the result and closed the seminar.

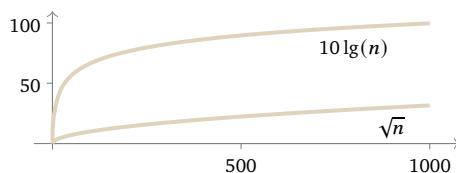
And this continues. Every day researchers produce results saying, “for this problem, here is a way to solve it in less time, or less space, etc.”[†] People are good at finding algorithms that solve a problem using less of some computational resource. But we are not as good at finding lower bounds, at proving “no algorithm, no matter how clever, can solve the problem faster than such and such.” This is one reason that we will compare the growth rates of resources consumed by algorithms using a tool, Big \mathcal{O} , that is like ‘less than or equal to’.



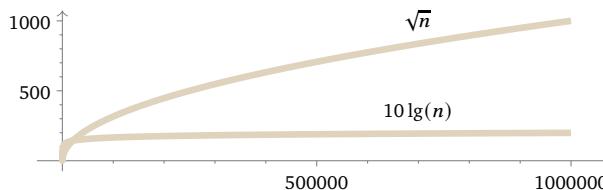
Andrey Kolmogorov 1903–1987

Motivation To compare algorithms we need a way to measure how they perform. Typically, an algorithm takes longer on longer input. So we describe performance by using a function whose argument is the input size and whose value is the maximum time that the algorithm takes on all inputs of at most that size.

Next we develop the criteria for the definition of Big \mathcal{O} , the tool that we use to compare performance functions. Suppose that we have two algorithms. When the input is size $n \in \mathbb{N}$, one takes \sqrt{n} many ticks while the other takes $10 \cdot \lg(n)$.[‡] Initially, \sqrt{n} looks better. For instance, $\sqrt{1\,000} \approx 31.62$ and $10 \lg(1\,000) \approx 99.66$.[#]



However, for large n the value \sqrt{n} is much bigger than $10 \lg(n)$. For instance, $\sqrt{1\,000\,000} = 1\,000$ while $10 \lg(1\,000\,000) \approx 199.32$.

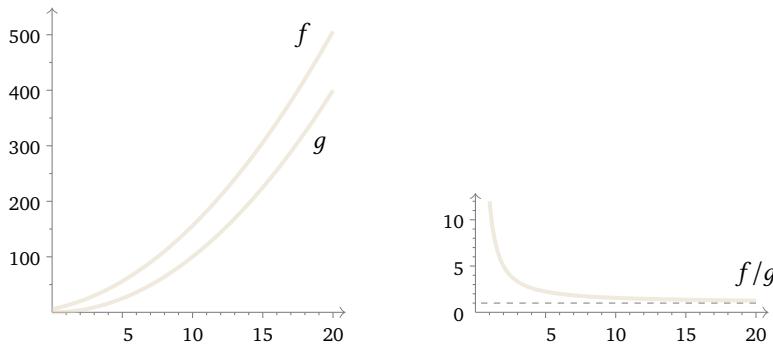


Thus the first criteria is that big \mathcal{O} must focus on what happens in the long run.

[†]See the Theory of Computing blog feed at <https://theory.report> (Various authors 2017). [‡]We write $\lg(n)$ for $\log_2(n)$. That is, compute $\lg(n)$ by finding the power of 2 that produces n , so if $n = 8$ then $\lg(n) = 3$, while if $n = 10$ then $\lg(n) \approx 3.32$. [#]These graphs show functions where the domain is the real numbers. Turing machines are discrete devices so it may seem more natural to have the domain be the natural numbers. But we will see that real functions are much more convenient for complexity measures.

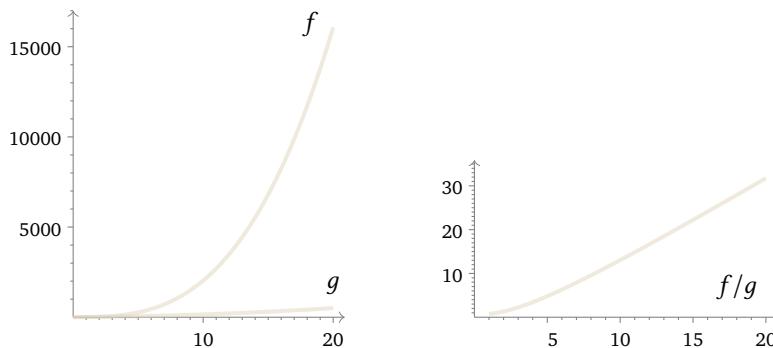
The second criteria is more subtle. The next four examples illustrate.

- 1.1 EXAMPLE These graphs compare $f(n) = n^2 + 5n + 6$ with $g(n) = n^2$. The graph on the right compares them in ratio, f/g .



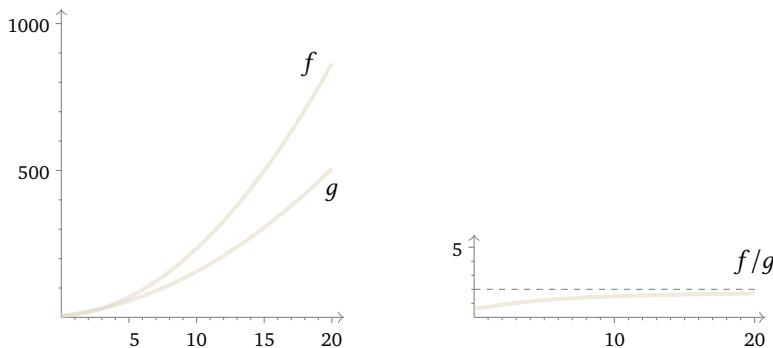
On the left we are struck that $n^2 + 5n + 6$ is ahead of n^2 . But on the right the ratios show that this is misleading. For large n 's, f 's 5n and 6 are swamped by the n^2 . Consequently in the long run these two functions track together—by far the biggest ingredient in their behavior is that they are both quadratic.

- 1.2 EXAMPLE Next compare the quadratic $g(n) = n^2 + 5n + 6$ with the cubic $f(x) = n^3 + 2n + 3$. In contrast to the prior example, these two don't track together. Initially g is larger, with $g(0) = 6 > f(0) = 3$ and $g(1) = 12 > f(1) = 6$. But then the cubic accelerates ahead of the quadratic, so much that at the scale of the image, the graph of g doesn't rise much above the axis.



On the right, the ratio grows without bound. So f is a faster-growing function than g . They both go to infinity but f goes there faster.

- 1.3 EXAMPLE Now compare the quadratics $f(x) = 2n^2 + 3n + 4$ and $g(n) = n^2 + 5n + 6$. We've already seen, as described above, that the function comparison definition needs to discount the initial behavior that $f(0) = 4 < g(0) = 6$ and $f(1) = 9 < g(1) = 12$, and instead focus on the long run.

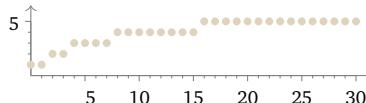


This example differs from Example 1.1 in that in the long run, f stays ahead of g and also gains in an absolute sense, because of f 's dominant term $2n^2$ is twice as large as g 's n^2 . So it may appear that we should view g 's rate as less than f 's. However unlike in Example 1.2, f does not accelerate away. Instead, the ratio between the two is bounded. We will take g to be equivalent to f .

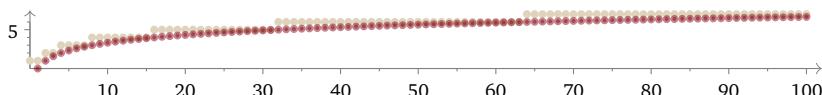
- 1.4 EXAMPLE We close the motivation with a very important example. Let the function $\text{bits} : \mathbb{N} \rightarrow \mathbb{N}$ give the number of bits needed to represent its input in binary. The bottom line of this table shows $\lg(n)$, the power of 2 that equals n .

<i>Input n</i>	0	1	2	3	4	5	6	7	8	9
<i>Binary</i>	0	1	10	11	100	101	110	111	1000	1001
$\text{bits}(n)$	1	1	2	2	3	3	3	3	4	4
$\lg(n)$	—	0	1	1.58	2	2.32	2.58	2.81	3	3.17

Here is a graph of $\text{bits}(n)$, the table's third line, for $n \in \{1, \dots, 30\}$.



The relationship between the third and fourth lines is that $\text{bits}(n) = 1 + \lfloor \lg(n) \rfloor$, except for the boundary value that $\text{bits}(0) = 1$ ($\lg(0)$ is undefined). The graph below compares $\text{bits}(n)$ with $\lg(n)$. Note the change in the horizontal and vertical scales.



This illustrates that in the formula $\text{bits}(n) = 1 + \lfloor \lg(n) \rfloor$, over the long run the '1+' and the floor don't matter much. A reasonable summary is that the base 2 logarithm, $\lg n$, describes the number of bits required to represent the number n .

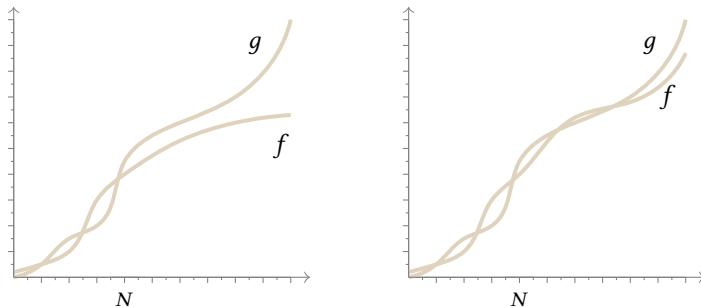
Further, the formula for converting among logarithmic functions with other bases, $\log_c(x) = \log_b(x)/\log_b(c)$, shows that they differ only by the constant factor

$1/\log_b(c)$. As Example 1.3 notes, with the function comparison definition given below we will disregard constant factors. So even the base does not matter—another reasonable summary is that the number of bits is “a” logarithmic function.

Definition Machine resource sizes, such as the number of bits of the input and of memory, are natural numbers. So to describe the performance of algorithms we may think to focus on functions that input and output natural numbers. However, above we have already found useful a function, \lg , that inputs and outputs reals. So instead we will consider a subset of the functions from \mathbb{R} to \mathbb{R} .[†]

- 1.5 **DEFINITION** A **complexity function** f is one that inputs real number arguments and outputs real number values, and (1) has an **unbounded domain**, so that there is a number $N \in \mathbb{R}^+$ such that $x \geq N$ implies that $f(x)$ is defined, and (2) is **eventually nonnegative**, so that there is a number $M \in \mathbb{R}^+$ so that $x \geq M$ implies that $f(x) \geq 0$.
- 1.6 **DEFINITION** Let g be a complexity function. Then **Big \mathcal{O}** of g , $\mathcal{O}(g)$, is the set of complexity functions f satisfying that there are constants $N, C \in \mathbb{R}^+$ so that if $x \geq N$ then both $g(x)$ and $f(x)$ are defined and $C \cdot g(x) \geq f(x)$. We say that f is $\mathcal{O}(g)$, or that $f \in \mathcal{O}(g)$, or that f is of order at most g , or that $f = \mathcal{O}(g)$.
- 1.7 **REMARKS** (1) We use the letter ‘ \mathcal{O} ’ because this is about the order of growth. (2) The term ‘complexity function’ is not standard but we find it convenient. (3) The ‘ $f = \mathcal{O}(g)$ ’ notation is very common, but awkward. It does not follow the usual rules of equality, such as that $f = \mathcal{O}(g)$ does not allow us to write ‘ $\mathcal{O}(g) = f$ ’. Another is that $x = \mathcal{O}(x^2)$ and $x^2 = \mathcal{O}(x^2)$ together do not imply that $x = x^2$. (4) Some authors do something a little different, they allow negative real outputs and write the inequality with absolute values, $f(x) \leq C \cdot |g(x)|$. (5) Sometimes you see ‘ f is $\mathcal{O}(g)$ ’ stated as ‘ $f(x)$ is $\mathcal{O}(g(x))$ ’. Speaking strictly, this is wrong because $f(x)$ and $g(x)$ are numbers, not functions.

Think of ‘ f is $\mathcal{O}(g)$ ’ as meaning that f ’s growth rate is less than or equal to g ’s rate. The sketches below illustrate the two possibilities.



[†]Using real functions has the disadvantage that it can seem to leave out natural number functions such as $n!$. One way to deal with this is to extend these natural number functions to take real number arguments, for instance, extending the factorial to $|x|!$, whose domain is the set of nonnegative reals (or to the more advanced Γ function).

On the left, g appears to accelerate away, suggesting that f 's rate of growth is strictly less than g 's. On the right the two seem to track together so that they have the same rate of growth, that is, f is $\mathcal{O}(g)$ and also g is $\mathcal{O}(f)$.

The definition requires that to show f is $\mathcal{O}(g)$, we must produce suitable N and C and verify that they work.

- 1.8 EXAMPLE Let $f(x) = x^2$ and $g(x) = x^3$. Then f is $\mathcal{O}(g)$, as witnessed by $N = 2$ and $C = 1$. The verification is: $x > N = 2$ implies that $g(x) = x^3 = x \cdot x^2$ is greater than $2 \cdot x^2$, which in turn is greater than $x^2 = C \cdot f(x) = 1 \cdot f(x)$.

- 1.9 EXAMPLE If $f(x) = 5x^2$ and $g(x) = x^4$ then to show f is $\mathcal{O}(g)$ take $N = 2$ and $C = 2$. The verification is that $x > N = 2$ implies that $C \cdot x^4 = 2 \cdot x^2 \cdot x^2 \geq 8x^2 > 5x^2$.

Don't confuse a function having smaller values with it having a smaller growth rate. Take $g(x) = x^2$ and $f(x) = x^2 + 1$, so that $g(x) < f(x)$ for all x . But g 's growth rate is not smaller; rather, f is $\mathcal{O}(g)$. To verify, take $N = 2$ and $C = 2$. Then $x \geq N = 2$ gives $C \cdot g(x) = 2x^2 = x^2 + x^2 > x^2 + 1 = f(x)$.

- 1.10 EXAMPLE Let $Z: \mathbb{R} \rightarrow \mathbb{R}$ be the zero function, $Z(n) = 0$. Then Z is $\mathcal{O}(g)$ for every complexity function g . Verify that with $N = 1$ and $C = 1$.

- 1.11 EXAMPLE Some pairs of functions aren't comparable, so that neither $f \in \mathcal{O}(g)$ nor $g \in \mathcal{O}(f)$. For an instance, let $g(x) = x^3$ and consider this function.

$$f(x) = \begin{cases} x^2 & \text{if } \lfloor x \rfloor \text{ is even} \\ x^4 & \text{if } \lfloor x \rfloor \text{ is odd} \end{cases}$$

Observe that f is not $\mathcal{O}(g)$, because for inputs where $\lfloor x \rfloor$ is odd there is no constant C that gives $C \cdot x^3 \geq x^4$ for all x . Likewise, g is not $\mathcal{O}(f)$ because of f 's behavior when $\lfloor x \rfloor$ is even.

- 1.12 LEMMA (ALGEBRAIC PROPERTIES) Let these be complexity functions.

- (A) If f is $\mathcal{O}(g)$ then for any constant $a \in \mathbb{R}^+$, the function $a \cdot f$ is $\mathcal{O}(g)$.
- (B) If f_0 is $\mathcal{O}(g_0)$ and f_1 is $\mathcal{O}(g_1)$ then the sum $f_0 + f_1$ is $\mathcal{O}(g)$ where $g(x) = \max(g_0(x), g_1(x))$. So if both f_0 and f_1 are $\mathcal{O}(g)$ then $f_0 + f_1$ is also $\mathcal{O}(g)$.
- (C) If f_0 is $\mathcal{O}(g_0)$ and f_1 is $\mathcal{O}(g_1)$ then the product $f_0 f_1$ is $\mathcal{O}(g_0 g_1)$.

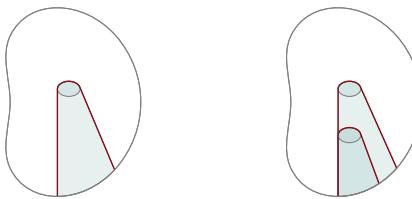
(As this section is a review, for a number of results the proofs are exercises. This one's proof is Exercise 1.58.)

That result gives us two principles for simplifying Big \mathcal{O} expressions. First, if an expression is a sum of finitely many terms of which one has the largest growth rate then we can drop the other terms. Second, if an expression is a product of factors then we can drop constants, factors that do not depend on the input.

- 1.13 EXAMPLE Consider $f(n) = 5x^3 + 3x^2 + 12x$. Looking to the first principle, the term with the largest growth rate is $5x^3$ (this is intuitively clear and it will follow from Theorem 1.17 below) and applying the lemma's second item with $g(x) = 5x^3$ gives that f is $\mathcal{O}(5x^3)$. Then the second simplification principle, applying the

lemma's first item with $a = 1/5$, gives that f is $\mathcal{O}(x^3)$.

- 1.14 **DEFINITION** Complexity functions f and g have **equivalent growth rates** or the **same order of growth** if f is $\mathcal{O}(g)$ and also g is $\mathcal{O}(f)$. We say that f is $\Theta(g)$ (read ‘ f is Big-Theta of g ’), or, what is the same thing, that g is $\Theta(f)$.
- 1.15 **LEMMA** The Big- \mathcal{O} relation is reflexive, so f is $\mathcal{O}(f)$. It is also transitive, so that if f is $\mathcal{O}(g)$ and g is $\mathcal{O}(h)$ then f is $\mathcal{O}(h)$. Thus having equivalent growth rates, which forces symmetry, is an equivalence relation between functions.



1.16 **FIGURE:** Each bean holds the complexity functions. Faster growing functions are higher, so that if they were shown then x^5 would be above x^4 . On the left is the cone $\mathcal{O}(g)$ for some g . The ellipse at the top is $\Theta(g)$, holding functions with growth rate equivalent to g 's. The sketch on the right adds the cone $\mathcal{O}(f)$ for some f in $\mathcal{O}(g)$.

The next result eases Big \mathcal{O} calculations for most of the functions that we encounter, such as polynomial, exponential, and logarithmic functions.

- 1.17 **THEOREM** Let f, g be complexity functions. Suppose that $\lim_{x \rightarrow \infty} f(x)/g(x)$ exists and equals L , which is a member of $\mathbb{R} \cup \{\infty\}$.
- (A) If $L = 0$ then g grows faster than f , that is, f is $\mathcal{O}(g)$ but g is not $\mathcal{O}(f)$.[†]
 - (B) If $L = \infty$ then f grows faster than g , so that g is $\mathcal{O}(f)$ but f is not $\mathcal{O}(g)$.[‡]
 - (C) If L is between 0 and ∞ then the two functions have something like the same growth rates, so that f is $\Theta(g)$ and g is $\Theta(f)$.[#]

It pairs well with the following result familiar from Calculus I.

- 1.18 **THEOREM (L'HÔPITAL'S RULE)** Let f and g be complexity functions such that both $f(x) \rightarrow \infty$ and $g(x) \rightarrow \infty$ as $x \rightarrow \infty$, and such that both are differentiable for large enough inputs. If $\lim_{x \rightarrow \infty} f'(x)/g'(x)$ exists and equals $L \in \mathbb{R} \cup \{\infty\}$ then $\lim_{x \rightarrow \infty} f(x)/g(x)$ also exists and also equals L .

- 1.19 **EXAMPLE** Let $f(x) = x^2 + 5x + 6$ and $g(x) = x^3 + 2x + 3$. Here we apply L'Hôpital's Rule multiple times.

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{x \rightarrow \infty} \frac{x^2 + 5x + 6}{x^3 + 2x + 3} = \lim_{x \rightarrow \infty} \frac{2x + 5}{3x^2 + 2} = \lim_{x \rightarrow \infty} \frac{2}{6x} = 0$$

[†]This case is denoted f is $o(g)$, read aloud as “Little Oh of g .” [‡]We also denote ‘ g is $\mathcal{O}(f)$ ’ by f is $\Omega(g)$, read aloud as “Big Omega of g .” [#]If $L = 1$ then f and g are **asymptotically equivalent**.

So f is $\mathcal{O}(g)$ but g is not $\mathcal{O}(f)$. That is, f 's growth rate is strictly less than g 's.

- 1.20 EXAMPLE Next consider $f(x) = 3x^2 + 4x + 5$ and $g(x) = x^2$.

$$\lim_{x \rightarrow \infty} \frac{3x^2 + 4x + 5}{x^2} = \lim_{x \rightarrow \infty} \frac{6x + 4}{2x} = \lim_{x \rightarrow \infty} \frac{6}{2} = 3$$

So their growth rates are roughly the same. That is, f is $\Theta(g)$.

- 1.21 EXAMPLE For $f(x) = 5x^4 + 15$ and $g(x) = x^2 - 3x$, this

$$\lim_{x \rightarrow \infty} \frac{5x^4 + 15}{x^2 - 3x} = \lim_{x \rightarrow \infty} \frac{20x^3}{2x - 3} = \lim_{x \rightarrow \infty} \frac{60x^2}{2} = \infty$$

shows that f 's growth rate is strictly greater than g 's rate— g is $\mathcal{O}(f)$ but f is not $\mathcal{O}(g)$.

- 1.22 EXAMPLE The logarithmic function $f(x) = \log_b(x)$ grows very slowly: $\log_b(x)$ is $\mathcal{O}(x)$, and $\log_b(x)$ is $\mathcal{O}(x^{0.1})$, and is $\mathcal{O}(x^{0.01})$. In fact by this equation, for any $d > 0$ no matter how small, $\log_b(x)$ is $\mathcal{O}(x^d)$ and x^d is not $\mathcal{O}(\log_b(x))$.

$$\lim_{x \rightarrow \infty} \frac{\log_b(x)}{x^d} = \lim_{x \rightarrow \infty} \frac{(1/x \ln(b))}{dx^{d-1}} = \frac{1}{d \ln(b)} \cdot \lim_{x \rightarrow \infty} \frac{1}{x^d} = 0$$

The difference in growth rates is even more marked than that. L'Hôpital's Rule, along with the Chain Rule, gives that $(\log_b(x))^2$ is $\mathcal{O}(x)$.

$$\begin{aligned} \lim_{x \rightarrow \infty} \frac{(\log_b(x))^2}{x} &= \lim_{x \rightarrow \infty} \frac{2 \ln(x)/(x \ln(b))}{1} \\ &= \frac{2}{\ln(b)} \cdot \lim_{x \rightarrow \infty} \frac{\ln(x)}{x} = \frac{2}{\ln(b)} \cdot \lim_{x \rightarrow \infty} \frac{1/x}{1} = 0 \end{aligned}$$

Further, Exercise 1.49 shows that for every power k the function $(\log_b(x))^k$ is $\mathcal{O}(x^d)$ for any $d > 0$.

The log-linear function $x \cdot \lg(x)$ has a similar relationship to the polynomials x^d , where $d > 1$.

$$\begin{aligned} \lim_{x \rightarrow \infty} \frac{x \lg(x)}{x^d} &= \lim_{x \rightarrow \infty} \frac{x \cdot (1/x \ln(2)) + 1 \cdot \ln(x)}{dx^{d-1}} = \frac{1}{d} \cdot \lim_{x \rightarrow \infty} \frac{(1/\ln(2)) + \ln(x)}{x^{d-1}} \\ &= \frac{1}{d(d-1)} \cdot \lim_{x \rightarrow \infty} \frac{(1/x)}{x^{d-2}} = \frac{1}{d(d-1)} \cdot \lim_{x \rightarrow \infty} \frac{1}{x^{d-1}} = 0 \end{aligned}$$

- 1.23 EXAMPLE We can compare the polynomial function $f(x) = x^2$ with the exponential function $g(x) = 2^x$.

$$\lim_{x \rightarrow \infty} \frac{2^x}{x^2} = \lim_{x \rightarrow \infty} \frac{2^x \cdot \ln(2)}{2x} = \lim_{x \rightarrow \infty} \frac{2^x \cdot (\ln(2))^2}{2} = \infty$$

Thus $f \in \mathcal{O}(g)$ but $g \notin \mathcal{O}(f)$. Induction gives that $\lim_{x \rightarrow \infty} 2^x/x^k = \infty$ for any k .

- 1.24 LEMMA Logarithmic functions grow more slowly than polynomial functions: if $f(x) = \log_b(x)$ for some base b and $g(x) = a_m x^m + \dots + a_0$ then f is $\mathcal{O}(g)$ but g is not $\mathcal{O}(f)$. Polynomial functions grow more slowly than exponential functions: where $h(x) = b^x$ for some base $b > 1$ then h is $\mathcal{O}(g)$ but g is not $\mathcal{O}(h)$.

We've defined complexity functions as mapping \mathbb{R} to \mathbb{R} , rather than the more natural \mathbb{N} to \mathbb{N} . One motivation is that some functions that we want to work with are real functions, such as logarithms. Another is that L'Hôpital's Rule, which uses the derivative and so needs reals, is a big convenience. The next result ensures that our conclusions in the continuous context carry over to the discrete.

- 1.25 LEMMA Let $f_0, f_1: \mathbb{R} \rightarrow \mathbb{R}$, and consider the restrictions to a discrete domain $g_0 = f_0|_{\mathbb{N}}$ and $g_1 = f_1|_{\mathbb{N}}$. Where $L \in \mathbb{R} \cup \{\infty\}$,
- (A) for $a \in \mathbb{R}$, if $L = \lim_{x \rightarrow \infty}(af_0)(x)$ then $L = \lim_{n \rightarrow \infty}(ag_0)(n)$
 - (B) if $L = \lim_{x \rightarrow \infty}(f_0 + f_1)(x)$ then $L = \lim_{n \rightarrow \infty}(g_0 + g_1)(n)$,
 - (C) if $L = \lim_{x \rightarrow \infty}(f_0 \cdot f_1)(x)$ then $L = \lim_{n \rightarrow \infty}(g_0 \cdot g_1)(n)$, and
 - (D) when the expressions are defined, if $L = \lim_{x \rightarrow \infty}(f_0/f_1)(x)$ then $L = \lim_{n \rightarrow \infty}(g_0/g_1)(n)$.

Tractable and intractable The table below lists orders of growth that are most common in practice.

Order	Name	Examples
$\mathcal{O}(1)$	Bounded	$f(n) = 15$
$\mathcal{O}(\lg(\lg(n)))$	Double logarithmic	$f(n) = \ln(\ln(n))$
$\mathcal{O}(\lg(n))$	Logarithmic	$f_0(n) = \ln(n), f_1(n) = \lg(n^3)$
$\mathcal{O}((\lg(n))^c)$	Polylogarithmic	$f(n) = (\lg(n))^3$
$\mathcal{O}(n)$	Linear	$f(n) = 3n + 4$
$\mathcal{O}(n \lg(n))$	Log-linear	$f_0(n) = 5n \lg(n) + n, f_1(n) = \lg(n!)$
$\mathcal{O}(n^2)$	Polynomial (quadratic)	$f(n) = 5n^2 + 2n + 12$
$\mathcal{O}(n^3)$	Polynomial (cubic)	$f(n) = 2n^3 + 12n^2 + 5$
\vdots		
$\mathcal{O}(2^{\text{poly}(\lg(n))})$	Quasipolynomial	$f_0(n) = 2^{(\lg(n))^2+3\lg(n)}, f_1(n) = n^{\lg(n)}$
\vdots		
$\mathcal{O}(2^n)$	Exponential	$f(n) = 10 \cdot 2^n$
$\mathcal{O}(3^n)$	Exponential	$f(n) = 6 \cdot 3^n + n^2$
\vdots		
$\mathcal{O}(n!)$	Factorial	$f(n) = 5 \cdot n! + n^{15} - 7$
$\mathcal{O}(n^n)$	-No standard name-	$f(n) = 2 \cdot n^n + 3 \cdot 2^n$

1.26 TABLE: The order of growth hierarchy

We often draw a line in this hierarchy after the polynomial functions; the next

table shows why. It lists how long a job would take if we used an algorithm that runs in time $\lg n$, time n , etc. (A modern computer runs at 10 GHz, 10 000 million ticks per second, and there are 3.16×10^7 seconds in a year.)

	$n = 1$	$n = 10$	$n = 50$	$n = 100$
$\lg n$	—	1.05×10^{-17}	1.79×10^{-17}	2.11×10^{-17}
n	3.17×10^{-18}	3.17×10^{-17}	1.58×10^{-16}	3.17×10^{-16}
$n \lg n$	—	1.05×10^{-16}	8.94×10^{-16}	2.11×10^{-15}
n^2	3.17×10^{-18}	3.17×10^{-16}	7.92×10^{-15}	3.17×10^{-14}
n^3	3.17×10^{-18}	3.17×10^{-15}	3.96×10^{-13}	3.17×10^{-12}
2^n	6.34×10^{-18}	3.24×10^{-15}	3.57×10^{-3}	4.02×10^{12}

1.27 TABLE: Time taken in years by algorithms whose behavior is given by a few functions, on a few size n 's.

In the $n = 100$ column, between the first few rows the relative change is an order of magnitude but the absolute times are small. Then we get to the final row. That's not a typo—the last entry really is on order of 10^{12} years. It is huge—the universe is 14×10^9 years old so this computation, even with input size of only 100, would take longer than the age of the universe. Exponential growth is very, very much larger than polynomial growth.

Another way to understand this is to think about, say, a string algorithm. Consider adding one more character to the input, for example by passing from the length ten string $\sigma_{10} = 11\ 0100\ 1010$ to the length eleven $\sigma_{11} = 110\ 1001\ 0101$. An algorithm that loops through the characters and so runs in $|\sigma|$ time must do one more loop, so that in this example it takes ten percent more time. But an algorithm that takes $2^{|\sigma|}$ time will take double the time.

Cobham's thesis is that the **tractable** problems—the ones that are at least conceivably solvable in practice—are those for which there is an algorithm whose resource consumption is at most polynomial.[†] For instance, if a problem's best available algorithm runs in exponential time then we may say that as we understand it today, the problem appears **intractable**.

Discussion Big \mathcal{O} is about relative scalability: an algorithm whose runtime behavior is $\mathcal{O}(n^2)$ scales worse than one whose behavior is $\mathcal{O}(n \lg n)$ but better than one whose behavior is $\mathcal{O}(n^3)$. Is there more to say?

Certainly that is the essence. Nonetheless, experience shows that there are points about Big \mathcal{O} that can puzzle learners and we will pause to elaborate on those.

[†]Cobham's Thesis is not universally accepted. Some researchers object that if an algorithm runs in time Cn^k but with an enormous k or an enormous C , or both, then the algorithm is not practical. A rejoinder to that objection notes a pattern that when someone announces an algorithm with a large exponent or large constant then typically the approach gets refined over time, shrinking the number. In any event, polynomial time is significantly better than exponential time. Here we accept Cobham's thesis because it gives technical meaning to the informal 'tractable'.

The first point is that Big \mathcal{O} is not the right tool for characterizing fine coding details. Contrast these.

```
(define (g0 n)
  (for ([i '(0 1 2 3 4)])
    (let ([x (* n n)])
      (printf "~a " (+ i x)))))
```

```
(define (g1 n)
  (let ([x (* n n)])
    (for ([i '(0 1 2 3 4)])
      (printf "~a " (+ i x))))))
```

They do the same thing but their run times are different. On the left $g0$ sets the local variable x inside the loop. That makes the code on the left slower than the right by four calculations. Big \mathcal{O} disregards this constant time difference. Big \mathcal{O} is good for comparing running times among algorithms but not as good for comparing running times among programs.

That fits with our second point about Big \mathcal{O} . We use it to help pick the best algorithm, to rank them according to how much they use of some computing resources. But algorithms are tied to an underlying computing model.[†]

Besides the Turing machine, another model that is widely used in this context is the **Random Access machine (RAM)**. Whereas a Turing machine cell stores only a single symbol, so that big numbers need multiple cells, on a RAM model machine each register holds an entire integer. And whereas to get to a cell a Turing machine may spend lots of steps traversing the tape, the RAM model gets each register's contents in a single step.

Close analysis shows that if we start with an algorithm intended for a RAM model machine and execute it on a Turing machine then this may add as much as n^3 extra ticks to the runtime, so that if the algorithm is $\mathcal{O}(n^2)$ on the RAM then on the Turing machine it can be $\mathcal{O}(n^5)$. Thus, to understand the cost of an algorithm, we must first fix a model and only then discuss the Big \mathcal{O} .

- 1.28 **DEFINITION** A machine \mathcal{M} with input alphabet Σ takes time $t_{\mathcal{M}}: \Sigma^* \rightarrow \mathbb{N} \cup \{\infty\}$ if that function gives the number of steps that the machine takes to halt on input $\sigma \in \Sigma^*$. If \mathcal{M} does not halt then $t_{\mathcal{M}}(\sigma) = \infty$. The machine runs in input length time $\hat{t}_{\mathcal{M}}: \mathbb{N} \rightarrow \mathbb{N} \cup \{\infty\}$ if $\hat{t}_{\mathcal{M}}(n)$ is the maximum of the $t(\sigma)$ over all inputs $\sigma \in \Sigma^*$ of length n . The machine runs in time $\mathcal{O}(f)$ if $\hat{t}_{\mathcal{M}}$ is $\mathcal{O}(f)$.

We have already, in relation to Cobham's Thesis, brought up our third point about Big \mathcal{O} . Its definition ignores constant factors; does that greatly reduce its value for comparing algorithms? If for instance an algorithm takes time given by Cn^2 for inputs $n > N$ then don't we need to know C and N ? After all, if one algorithm has runtime C_0n with an enormous C_0 while another is C_1n^2 for tiny C_1 , could that not make the second algorithm more useful in practice? Similarly, could a huge N mean that we need to describe what happens to inputs less than that number?

Part of the answer is that finding these constants is hard.[‡] Machines vary widely in their low-level details such as the memory addressing and paging, cache

[†]More discussion of the relationship between algorithms and machine models is in Section 3. [‡]Authors do sometimes state the order of magnitude of these constants.

hits, and whether the CPU can do some operations in parallel, and these details can make a tremendous difference in constants such as C and N . Imagine doing the analysis on a commercially available machine and then the vendor releases a new model, so it is all to do again. That would be discouraging. And what's more, experience shows that doing the work to find the exact numbers usually does not change the algorithm that gets picked. As Table 1.27 illustrates, knowing at a Big \mathcal{O} level how the algorithm grows is, at least past some point, much more important than knowing the values of the constants.



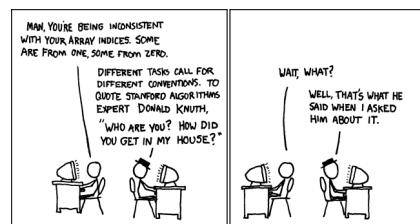
Donald Knuth
b 1938

Instead of analyzing commercial machines, we could agree on specifications for a reference architecture — this is the approach taken by D Knuth in the monumental *Art of Computer Programming* series — but again there is the risk that we might have to update that standard. Then published results from some time ago might no longer apply because they refer to an old standard. Again, discouraging. So the analysis that we do to find the Big \mathcal{O} behavior of an algorithm usually refers to an abstract computer model, such as a Turing machine or a RAM model, and it usually does not go to the extent of finding the constants. That is, being reference independent is an advantage, particularly in a quickly changing field.

This echos the paragraph at the start of this discussion. Not taking into account the precise difference between, say, the cost of a division and the cost of a memory write (as long as those costs lie between reasonable limits) discounts the meaningfulness of the constant factors and instead focuses on relative comparisons. Here is an analogy: absolute measurement of distance involves units such as miles or kilometers, but being able to make statements irrespective of the unit constants requires making statements that are relative, such as “from here, New York City is twice as far as Boston.”

That is, if we have an algorithm that on input of size n will take $3n$ ticks then we say it is $\mathcal{O}(n)$ in order to express, roughly, that doubling the input size will no more than double the number of steps taken. Similarly, if an algorithm is $\mathcal{O}(n^2)$ then doubling the input size will roughly at most quadruple the number of steps. The Big \mathcal{O} notation ignores constants because that is inherent in being a unit free, relative measurement.

However, if a person has the sense that leaving out the constants makes this measure approximate, then certainly, Big \mathcal{O} is only a rough comparison. It may suggest that a $\mathcal{O}(n^2)$ algorithm is better than a $\mathcal{O}(n^5)$ one. But it cannot say which of two $\mathcal{O}(n^2)$ algorithms will be absolutely better when they are coded and run on a particular platform, for input sizes in a given range.[†]



Courtesy xkcd.com

[†]For that, use benchmarks.

This leads to our fourth point about Big \mathcal{O} . Understanding how an algorithm performs as the input size grows requires that we define the input size.

Consider an algorithm for testing primality that inputs a natural number n and tries each $k \in \{2, \dots, n-1\}$ to see if it divides n . The worst case n is that it tests all of those k 's, roughly n of them. Take the size of n to be the number of bits needed to represent n , approximately $\lg n$. So for this algorithm the input is of size $\lg n$ and the number of operations is about n . That's exponential growth—passing from $\lg n$ to n requires exponentiating.

However, in a programming class this algorithm would likely be described as linear because for the input n there are about n -many divisions. How to explain the difference?

This is about the relationship between the algorithm and the underlying computing model. We may make an engineering judgment that for every use of our program the input will fit into a 64 bit word. We are choosing a computation model that is like the RAM model, where large numbers take the same time to read as small ones. With this model the relationship between size of the input and the runtime is linear.

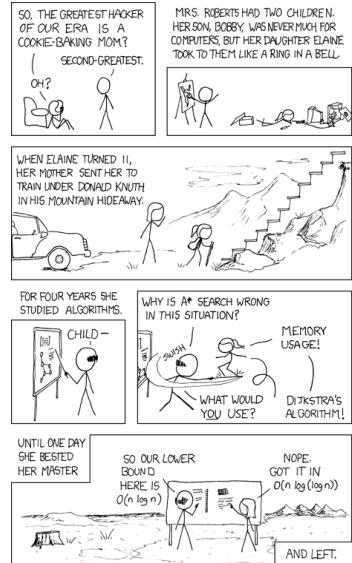
This difference is in part a theory-versus-application thing. In a common programming setting where the input is bounded, the behavior is $\mathcal{O}(n)$. In a theoretical setting the algorithm accepts arbitrarily large input and so the runtime is a function of the bit size of the inputs, the algorithm is $\mathcal{O}(2^b)$. An algorithm whose behavior as a function of the input is polynomial but whose behavior as a function of the bit size of the input is exponential is **pseudopolynomial**.

A final point about Big \mathcal{O} . When analyzing an algorithm we can consider the behavior that is the worst case for any input of that size as in Definition 1.28, or the behavior that is the average over all inputs of that size (worst-case analysis is much more common). For instance, the quicksort algorithm takes quadratic time $\mathcal{O}(n^2)$ at worst but on average is $\mathcal{O}(n \lg n)$.

Related to that, imagine a machine with this runtime behavior.

$$f(n) = \begin{cases} n! & - n \text{ is a power of ten} \\ n & - \text{otherwise} \end{cases}$$

This machine runs in superexponential time for rare inputs (called “black holes”). The definition gives that overall this machine runs in time $\mathcal{O}(n!)$, while for most inputs it would be quite fast.[†]



Courtesy xkcd.com

[†]A real life example of such a thing is that the simplex algorithm, which is very widely used for linear optimization, runs in exponential time in the worst case but typically seems to run in polynomial time.

V.1 Exercises

- 1.29 True or false: if a function is $\mathcal{O}(n^2)$ then it is $\mathcal{O}(n^3)$.
- ✓ 1.30 Your classmate emails you a draft of an assignment answer that says, “I have an algorithm with running time that is $\mathcal{O}(n^2)$. So with input $n = 5$ it will take 25 ticks.” Make two corrections.
- 1.31 Suppose that someone posts to a group that you are in, “I’m working on a problem that is $\mathcal{O}(n^3)$.” Explain to them, gently, how their sentence is mistaken.
- ✓ 1.32 How many bits does it take to express each number in binary? (A) 5 (B) 50 (C) 500 (D) 5 000
- ✓ 1.33 One is true, the other one is not. Which is which?
 (A) If f is $\mathcal{O}(g)$ then f is $\Theta(g)$.
 (B) If f is $\Theta(g)$ then f is $\mathcal{O}(g)$.
- ✓ 1.34 For each find the function on the order of growth hierarchy, Table 1.26, that has the same rate of growth. (A) $n^2 + 5n - 2$ (B) $2^n + n^3$ (C) $3n^4 - \lg \lg n$ (D) $\lg n + 5$
- 1.35 For each give the function on the order of growth hierarchy, Table 1.26, that has the same rate of growth. That is, find g in that table where f is $\Theta(g)$.
- (A) $f(n) = \begin{cases} n & \text{if } n < 100 \\ 0 & \text{else} \end{cases}$
- (B) $f(n) = \begin{cases} 1\,000\,000 \cdot n & \text{if } n < 10\,000 \\ n^2 & \text{else} \end{cases}$
- (C) $f(n) = \begin{cases} 1\,000\,000 \cdot n^2 & \text{if } n < 100\,000 \\ \lg n & \text{else} \end{cases}$
- ✓ 1.36 For each pair, find the limit of the ratio f/g to decide of f is $\mathcal{O}(g)$, or g is $\mathcal{O}(f)$, or both, or neither.
 (A) $f(n) = 3n^3 + 2n + 4$, $g(n) = \ln(n) + 6$
 (B) $f(n) = 3n^3 + 2n + 4$, $g(n) = n + 5n^3$ (C) $f(n) = (1/2)n^3 + 12n^2$, $g(n) = n^2 \ln(n)$
 (D) $f(n) = \lg(n) = \log_2(n)$, $g(n) = \ln(n)$ (E) $f(n) = n^2 + \lg(n)$, $g(n) = n^4 - n^3$
 (F) $f(n) = 55$, $g(n) = n^2 + n$
- ✓ 1.37 For each pair of functions simplify using Lemma 1.12 to decide if f is $\mathcal{O}(g)$, or g is $\mathcal{O}(f)$, or both, or neither.
 (A) $f(n) = 4n^2 + 3$, $g(n) = (1/2)n^2 - n$
 (B) $f(n) = 53n^3$, $g(n) = \ln n$ (C) $f(n) = 2n^2$, $g(n) = \sqrt{n}$ (D) $f(n) = n^{1.2} + \lg n$, $g(n) = n^{\sqrt{2}} + 2n$ (E) $f(n) = n^6$, $g(n) = 2^{n/6}$ (F) $f(n) = 3^n$, $g(n) = 3 \cdot 2^n$
 (G) $f(n) = \lg(3n)$, $g(n) = \lg(n)$
- 1.38 Which of these are $\mathcal{O}(n^2)$? (A) $\lg n$ (B) $3 + 2n + n^2$ (C) $3 + 2n + n^3$
 (D) $10 + 4n^2 + \lfloor \cos(n^3) \rfloor$ (E) $\lg(5^n)$
- ✓ 1.39 For each, state true or false. (A) $5n^2 + 2n$ is $\mathcal{O}(n^3)$ (B) $2 + 4n^3$ is $\mathcal{O}(\lg n)$
 (C) $\ln n$ is $\mathcal{O}(\lg n)$ (D) $n^3 + n^2 + n$ is $\mathcal{O}(n^3)$ (E) $n^3 + n^2 + n$ is $\mathcal{O}(2^n)$
- 1.40 For each find the smallest $k \in \mathbb{N}$ so that the given function is $\mathcal{O}(n^k)$.

- (A) $n^3 + (n^4/10\,000\,000)$ (B) $(n+2)(n+3)(n^2 - \lg n)$ (C) $5n^3 + 25 + \lceil \cos(n) \rceil$
 (D) $9 \cdot (n^2 + n^3)^4$ (E) $\lfloor \sqrt{5n^7 + 2n^2} \rfloor$

1.41 Consider Table 1.27. (A) Add a column for $n = 200$. (B) Add a row for 3^n .

- ✓ 1.42 On a computer that performs at 10 GHz, at 10 000 million instructions per second, what is the longest input that can be done in a year under an algorithm with each time performance function? (A) $\lg n$ (B) \sqrt{n} (C) n (D) n^2 (E) n^3 (F) 2^n

1.43 What is the least input number such that $f(n) = 100\,000 \cdot n^2$ is less than $g(n) = n^3$?

1.44 What is the order of growth of the run time of a deterministic Finite State machine?

- ✓ 1.45 (A) Verify that $f(x) = 7$ is $\mathcal{O}(1)$. (B) Verify that $f(x) = 7 + \sin(x)$ is $\mathcal{O}(1)$. So if a function is in $\mathcal{O}(1)$, that does not mean that it is a constant function. (C) Verify that $f(x) = 7 + (1/x)$ is also $\mathcal{O}(1)$. (D) Show that a complexity function f is $\mathcal{O}(1)$ if and only if it is bounded above by a constant, that is, if and only if there exists $L \in \mathbb{R}$ so that $f(x) \leq L$ for all inputs $x \in \mathbb{R}$.

1.46 Where does $g(x) \leq x^{\mathcal{O}(1)}$ place the function g in the order of growth hierarchy? Hint: see the prior question.

1.47 Let $f(x) = 2x$ and $g(x) = x^2$. Prove directly from Definition 1.6 that f is $\mathcal{O}(g)$, but that g is not $\mathcal{O}(f)$.

1.48 Prove that 2^n is $\mathcal{O}(n!)$. Hint: because of the factorial, consider these natural number functions and find suitable $N, C \in \mathbb{N}$.

1.49 Use L'Hôpital's Rule as in Example 1.22 to verify these for any $d \in \mathbb{R}^+$:
 (A) $(\log_b(x))^3$ is $\mathcal{O}(x^d)$ (B) for any $k \in \mathbb{N}^+$, $(\log_b(x))^k$ is $\mathcal{O}(x^d)$.

1.50 Assume that $g: \mathbb{R} \rightarrow \mathbb{R}$ is increasing, so that $x_1 \geq x_0$ implies that $g(x_1) \geq g(x_0)$. Let $f: \mathbb{R} \rightarrow \mathbb{R}$ be a constant function. Show that f is $\mathcal{O}(g)$.

1.51 (A) Show that there is a computable function whose output values grow at a rate that is $\mathcal{O}(1)$, one whose values grow at a rate that is $\mathcal{O}(n)$, one for $\mathcal{O}(n^2)$, etc.
 (B) The Halting problem function K is uncomputable. Place its rate of growth in the order of growth hierarchy, Table 1.26. (C) Produce a function that is not computable because its output values are larger than those of any computable function. (You need not show that the rate of growth is greater, only that the outputs are larger.)

1.52 Show that $x^{\lg x}$ is quasipolynomial.

1.53 Show that the quasipolynomial function $f(x) = x^{\lg x}$ grows faster than any polynomial but slower than any exponential function.

- ✓ 1.54 Show that $\mathcal{O}(2^x) \in \mathcal{O}(3^x)$ but $\mathcal{O}(2^x) \neq \mathcal{O}(3^x)$.

1.55 Table 1.26 states that $n!$ grows slower than n^n .

(A) Verify this. Hint: although $n!$ is a natural number function, Theorem 1.17 still applies.

- (B) Stirling's formula is that $n! \approx \sqrt{2\pi n} \cdot (n^n/e^n)$. Doesn't this imply that $n!$ is $\Theta(n^n)$?
- ✓ 1.56 Two complexity functions f, g are **asymptotically equivalent**, $f \sim g$, if $\lim_{x \rightarrow \infty} (f(x)/g(x)) = 1$. Show that each pair is asymptotically equivalent:
 (A) $f(x) = x^2 + 5x + 1$ and $g(x) = x^2$, (B) $\lg(x+1)$ and $\lg(x)$.
- 1.57 Is there an f so that $\mathcal{O}(f)$ is the set of all polynomials?
- 1.58 Verify the clauses of Lemma 1.12. (A) If $a \in \mathbb{R}^+$ then af is also $\mathcal{O}(g)$.
 (B) The function $f_0 + f_1$ is $\mathcal{O}(g)$, where g is defined by $g(n) = \max(g_0(n), g_1(n))$.
 (C) The product $f_0 f_1$ is $\mathcal{O}(g_0 g_1)$.
- 1.59 Verify these clauses of Lemma 1.15. (A) The Big- \mathcal{O} relation is reflexive.
 (B) It is also transitive.
- 1.60 Assume that f and g are complexity functions. (A) Suppose that $\lim_{x \rightarrow \infty} f(x)/g(x)$ exists and equals 0. Show that f is $\mathcal{O}(g)$. (*Hint:* this requires a rigorous definition of the limit.) (B) We can give an example where f is $\mathcal{O}(g)$ even though $\lim_{x \rightarrow \infty} f(x)/g(x)$ does not exist. Verify that, where $g(x) = x$ and where $f(x) = x$ when $\lfloor x \rfloor$ is odd and $f(x) = 2x$ when $\lfloor x \rfloor$ is even.
- 1.61 Prove Lemma 1.24.

SECTION

V.2 A problem miscellany

Much of today's work in the Theory of Computation is driven by problems that originate outside of the subject. We will describe some of these problems to get a sense of the ones that people work on and also to use for examples and exercises. All of these problems are well-known to anyone in the field.

Problems, with stories We start with a few that come with stories. These stories are fun and an important part of the culture, and they also give a sense of where in general problems come from.

WR Hamilton was a polymath whose genius was recognized early and he was given a sinecure as Astronomer Royal of Ireland. He made important contributions to classical mechanics, where his reformulation of Newtonian mechanics is now called Hamiltonian mechanics. Other work of his in physics helped develop classical field theories such as electromagnetism and laid the groundwork for the development of quantum mechanics. In mathematics, he is best known as the inventor of the quaternion number system.



William Rowan
Hamilton
1805–1865

One of his ventures was a game, *Around the World*. The vertices in the graph below were holes in a wooden board, labeled with the names of world cities. Players put pegs in the holes, looking for a circuit that visits each city once and only once.

2.1 ANIMATION: Hamilton's *Around the World* game

It did not make Hamilton rich. But it did get him associated with a great problem.

- 2.2 **PROBLEM (Hamiltonian Circuit)** Given a graph, decide if it contains a cyclic path that includes each vertex once and only once.

A special case is the **Knight's Tour** problem, to use a chess knight to make a circuit of the squares on the board. (Recall that a knight moves three squares at a time, with the first two squares in one direction and then the third one perpendicular to that direction.)

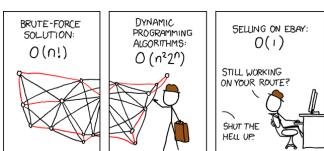


42	59	44	9	40	21	46	7
61	10	41	58	45	8	39	20
12	45	60	55	22	57	6	47
53	62	11	30	25	28	19	38
32	13	54	27	56	23	48	5
63	52	31	24	29	26	37	18
14	33	2	51	16	35	4	49
1	64	15	34	3	50	17	36

This is the solution given by L Euler. In graph terms, there are sixty four vertices, representing the board squares. An edge goes between two vertices if they are connected by a single knight move. **Knight's Tour** asks for a Hamiltonian circuit of that graph.

Hamiltonian Circuit has another famous variant.

- 2.3 **PROBLEM (Traveling Salesman)** Given a weighted undirected graph, where we call the vertices $S = \{c_0, \dots c_{k-1}\}$ ‘cities’ and we call the edge weight $d(c_i, c_j) \in \mathbb{N}^+$ for $i \neq j$ the ‘distance’ between the cities, find the shortest-distance circuit that visits every city and returns back to the start.



Courtesy xkcd.com

We can start with a map of the state capitals of the forty eight contiguous US states and the distances between them: Montpelier VT to Albany NY is 254 kilometers, etc. From among all trips that visit each city and return back to the start, such as Montpelier → Albany → Harrisburg → ... → Montpelier, we want the shortest one.

As stated, this is an optimization problem. However we can recast it as a decision problem. Introduce a parameter bound $B \in \mathbb{N}$ and change the problem statement to ‘decide if there is a circuit of total distance less

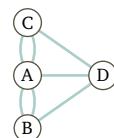
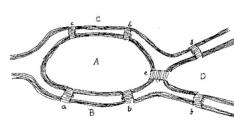
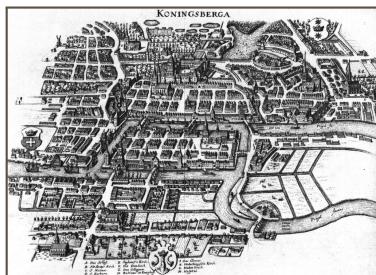
than B' . If we had an algorithm to quickly solve this decision problem then we could also solve the optimization problem: ask whether there is a trip bounded by length $B = 1$, then ask if there is a trip of length $B = 2$, etc. When we eventually get a ‘yes’, we know the length of the shortest trip.

The next problem sounds much like Hamiltonian Circuit, in that it involves exhaustively traversing a graph. But it proves to act very differently.

Today the city of Kaliningrad is a Russian enclave between Poland and Lithuania. But in 1727 it was in Prussia and was called Königsberg. The Pregel river divides the city into four areas, connected by seven bridges. The citizens used to promenade, to take leisurely walks or drives where they could see and be seen. The question arose: can a person cross each bridge once and only once, and arrive back at the start? No one could think of a way but no one could think of a reason that there was no way. A local mayor wrote to Euler, who proved that no circuit is possible. This paper founded Graph Theory.



Leonhard Euler
1707–1783



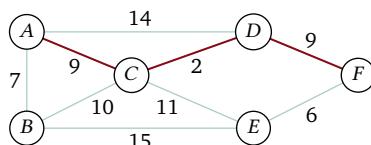
Euler’s summary sketch is in the middle and the graph is on the right.

- 2.4 **PROBLEM (Euler Circuit)** Given a graph, find a circuit that traverses each edge once and only once, or find that no such circuit exists.

Next is a problem that sounds hard. But all of us see it solved every day, for instance when we ask our smartphone for the shortest route to some place.

- 2.5 **PROBLEM (Shortest Path)** Given a weighted graph and two vertices, find the least-weight path between them, or find that no path exists.

There is an algorithm that solves this problem quickly.[†] For instance, with the graph below we could look for the path from A to F of least cost.



[†]Dijkstra’s algorithm is at worst quadratic in the number of vertices.

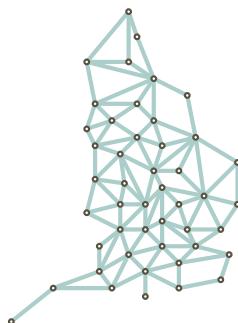
The next problem was discovered in 1852 by a young mathematician, F Guthrie, who was drawing a map of England's counties. He wanted to color them, with different colors for counties that share a border. His map required only four colors and he conjectured that for any map, four colors were enough.



Augustus De
Morgan
1806–1871

Guthrie imposed the condition that the countries must be contiguous and he defined ‘sharing a border’ to mean sharing an interval, not just a point (see Exercise 2.48). Below is a map and a graph version of the same problem. In the graph, counties are vertices and edges connect ones that are adjacent. A crucial point is that the graph is **planar**— we can draw it in the plane so that its edges do not cross.

The **Four Color** problem is to start with a planar graph and end with the vertices partitioned into no more than four sets, called **colors**, such that adjacent vertices are in different colors. Guthrie consulted his former professor, A De Morgan, who was also unable to either prove or disprove the conjecture. But he did make the problem famous by promoting it among his friends.



2.6 ANIMATION: Counties of England and the derived planar graph



Appel and Haken's post office celebrating

It remained unsolved until 1976, when K Appel and W Haken reduced the proof to 1 936 cases and got a computer to check those cases. This was the first major proof that was done on a computer and it was controversial. Many mathematicians felt that the purpose of the subject was to understand things and not just be satisfied when a computer program (that conceivably had bugs) assures us that theorems are verified.[†] However, today's generation of mathematicians is more comfortable with this and now computer proofs are routine.

- 2.7 **PROBLEM (Graph Colorability)** Given a graph and a number $k \in \mathbb{N}$, decide whether the graph is **k -colorable**, whether we can partition its vertices into k -many sets, $\mathcal{N} = \mathcal{C}_0 \cup \dots \cup \mathcal{C}_{k-1}$, such that no two same-set vertices are connected.

[†]This is in contrast to the *Entscheidungsproblem*.

- 2.8 **PROBLEM (Chromatic Number)** Given a graph, find the smallest number $k \in \mathbb{N}$ such that the graph is k -colorable.

Our final story introduces a problem that will be a benchmark to which we compare other problems. In 1847, G Boole outlined what we today call Boolean algebra. A variable is Boolean if it takes only the values T or F . We focus on Boolean expressions that connect variables using the **and operator \wedge** , the **or operator \vee** , and the **not operator \neg** . (For more, see Appendix C.) This Boolean function is given by an expression with three variables.

$$f(P, Q, R) = (P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (\neg P \vee \neg Q \vee \neg R)$$



George
Boole
1815–
1864

An expression is **satisfiable** if some combination of input T 's and F 's makes it evaluate to T . It is in **Conjunctive Normal form** if it consists of clauses of variables or negations connected by \vee 's, where the clauses are connected with \wedge 's. This **truth table** shows the input-output behavior of the function defined by the expression.

P	Q	R	$P \vee Q$	$P \vee \neg Q$	$\neg P \vee Q$	$\neg P \vee \neg Q \vee \neg R$	$f(P, Q, R)$
F	F	F	F	T	T	T	F
F	F	T	F	T	T	T	F
F	T	F	T	F	T	T	F
F	T	T	T	F	T	T	F
T	F	F	T	T	F	T	F
T	F	T	T	T	F	T	F
T	T	F	T	T	T	T	T
T	T	T	T	T	T	F	F

That T in the final column witnesses that this formula is satisfiable.

- 2.9 **PROBLEM (Satisfiability, SAT)** Decide if a given Boolean expression is satisfiable.

- 2.10 **PROBLEM (3-Satisfiability, 3-SAT)** Given a propositional logic formula in Conjunctive Normal form in which each clause has at most three variables, decide if it is satisfiable.

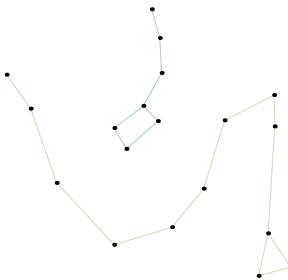
Observe that if the number of input variables is v then the number of rows in the truth table is 2^v . So solving **SAT** appears to require exponential time. Whether that is right is a very important question, as we will see in later sections.

More problems, omitting the stories We will list more example problems but leaving out the background (although for some of them the motivation is clear even without a story). All of these problems are also widely known in the field.

- 2.11 **PROBLEM (Vertex-to-Vertex Path)** Given a graph and two vertices, find if the second is reachable from the first.[†]

[†]The name Vertex-to-Vertex Path is nonstandard. It is usually known as *st*-Path, *st*-Connectivity, or STCON (*s* and *t* are generic names for vertices).

- 2.12 EXAMPLE These are two Western-tradition constellations, Ursa Minor and Draco.

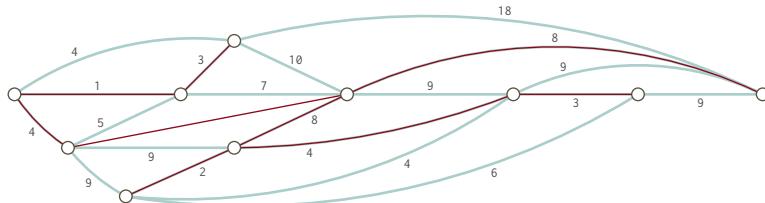


Here we can solve the Vertex-to-Vertex Path problem by eye. For any two vertices in Ursa Minor there is a path and for any two vertices in Draco there is a path. But if the two are in different constellations then there is no path.

For a graph with many thousands of nodes, such as a computer network, the problem is harder than in the prior example. A close variant problem is to decide, given a graph, whether all vertex pairs are connected.

- 2.13 PROBLEM (Minimum Spanning Tree) Given a weighted undirected graph, find a subgraph containing all the vertices of the original graph such that its edges have a minimum total.

This is an undirected graph with weights on the edges.

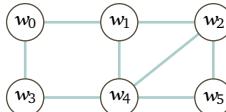


The highlighted subgraph includes all of the vertices, that is, it spans the graph. In addition, its weights total to a minimum from among all of the spanning subgraphs. From that it follows that this subgraph is a tree, meaning that it has no cycles, or else we could eliminate an edge from the cycle and thereby lower the edge weight total without dropping any vertices.

This looks somewhat like the Hamiltonian Circuit problem in that the sought-for subgraph contains all of the vertices. However, for the Minimum Spanning Tree problem we know algorithms that are quick, $\mathcal{O}(n \lg n)$.

- 2.14 PROBLEM (Vertex Cover) Given a graph and a bound $B \in \mathbb{N}$, decide if the graph has a size B set of vertices, C , such that for any edge, at least one of its ends is a member of C .

- 2.15 EXAMPLE A museum posts guards to watch their exhibits. There are eight halls, laid out as below. They will put the guards at some of the corners w_0, \dots, w_5 . What is the smallest number of guards that will suffice to watch all of the hallways?

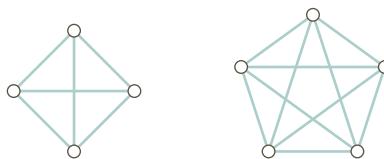


Checking each corner shows that one guard will not suffice. The two-element set $C = \{w_0, w_4\}$ is a vertex cover: every hallway has at least one end in C .

- 2.16 **PROBLEM (Clique)** Given a graph and a bound $B \in \mathbb{N}$, decide if the graph has a size B set vertices such that any two are connected.

The term ‘clique’ comes from social networks; if the nodes represent people and the edges connect friends then a clique is a set of people who are all friends.

A graph with a 4-clique has the subgraph like the one below on the left and any graph with a 5 clique has the subgraph like the one the right.



- 2.17 **EXAMPLE** This graph has a 4-clique.

2.18 ANIMATION: Instance of the Clique problem

- 2.19 **PROBLEM (Max Cut)** A **graph cut** partitions the vertices into two disjoint subsets. The **cut set** contains the edges with a vertex in each subset. The **Max Cut** problem is to find the partition with the largest cut set.

- 2.20 **EXAMPLE** For this graph the largest cut set contains six edges, the ones connecting differently colored vertices here.[†]

2.21 ANIMATION: A partition for the graph with a maximum cut set.

[†]One way to verify this is with a script that checks all two-set partitions of the vertices.

2.22 **PROBLEM (Three Dimensional Matching)** Let the sets X, Y, Z all have the same number of elements, n . Given as input a set $M \subseteq X \times Y \times Z$, decide if there is a **matching**, a set $\hat{M} \subseteq M$ containing n elements such that no two of the triples in \hat{M} agree on their first coordinates, or their second or third coordinates either.

2.23 **EXAMPLE** Let $X = \{a, b\}$, $Y = \{b, c\}$, and $Z = \{a, d\}$, so that $n = 2$. Below is a subset of $X \times Y \times Z$ (it actually equals $X \times Y \times Z$).

$$M = \{\langle a, b, a \rangle, \langle a, c, a \rangle, \langle b, b, a \rangle, \langle b, c, a \rangle, \langle a, b, d \rangle, \langle a, c, d \rangle, \langle b, b, d \rangle, \langle b, c, d \rangle\}$$

The set $\hat{M} = \{\langle a, b, a \rangle, \langle b, c, d \rangle\}$ has 2 elements. They disagree in their first coordinates, and their second, and their third.

2.24 **EXAMPLE** Fix $n = 4$ and consider $X = \{1, 2, 3, 4\}$, $Y = \{10, 20, 30, 40\}$, and $Z = \{100, 200, 300, 400\}$, all four-element sets. Also fix this subset of $X \times Y \times Z$.

$$M = \{\langle 1, 10, 200 \rangle, \langle 1, 20, 300 \rangle, \langle 2, 30, 400 \rangle, \langle 3, 10, 400 \rangle, \\ \langle 3, 40, 100 \rangle, \langle 3, 40, 200 \rangle, \langle 4, 10, 200 \rangle, \langle 4, 20, 300 \rangle\}$$

A matching is $\hat{M} = \{\langle 1, 20, 300 \rangle, \langle 2, 30, 400 \rangle, \langle 3, 40, 100 \rangle, \langle 4, 10, 200 \rangle\}$.

2.25 **PROBLEM (Subset Sum)** Given a multiset of natural numbers $S = \{n_0, \dots, n_{k-1}\}$ and a target $T \in \mathbb{N}$, decide if a subset of S sums to the target.[†]

2.26 **EXAMPLE** Do some of the numbers $\{911, 22, 821, 563, 405, 986, 165, 732\}$ add to $T = 1173$? One such collection is $\{165, 986, 22\}$.

In contrast, no subset of $\{831, 357, 63, 987, 117, 81, 6785, 606\}$ adds to $T = 2105$. All of the numbers are multiples of three, while the target T is not.

2.27 **PROBLEM (Knapsack)** Given a finite multiset S whose elements s have a natural number weight $w(s)$ and value $v(s)$, and also given a weight bound B and a value target T , find a subset $\hat{S} \subseteq S$ whose elements have a total weight less than or equal to the bound and total value greater than or equal to the target.

2.28 **EXAMPLE** Our knapsack can carry at most $B = 10$ pounds. Can we pack items with total worth at least $T = 100$?

Item	a	b	c	d
Weight	3	4	5	6
Value	50	40	10	30

The best that we can do is take items a and b. We cannot meet the value target.

[†]Recall that in a multiset repeats do not collapse, so the multiset $\{1, 2, 2, 3\}$ is different than the multiset $\{1, 2, 3\}$. But a multiset is like a set in that the order of the elements is not significant, so the multiset $\{1, 2, 2, 3\}$ is the same as the multiset $\{1, 2, 3, 2\}$. In short, a multiset is an unordered list.

2.29 **PROBLEM (Partition)** Given a finite multiset A such that each element has an associated natural number size $s(a)$, decide if the set splits into two, \hat{A} and $A - \hat{A}$, so that the total of the sizes is the same, $\sum_{a \in \hat{A}} s(a) = \sum_{a \notin \hat{A}} s(a)$.

2.30 **EXAMPLE** The set $A = \{ \text{I}, \text{a}, \text{my}, \text{go}, \text{rivers}, \text{cat}, \text{hotel}, \text{comb} \}$ has eight words. The size of a word, $s(\sigma)$, is the number of letters. Then $\hat{A} = \{ \text{cat}, \text{rivers}, \text{I}, \text{a}, \text{go} \}$ gives $\sum_{a \in \hat{A}} s(a) = \sum_{a \notin \hat{A}} s(a) = 12$.

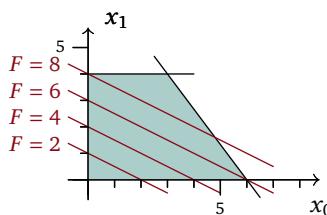
2.31 **EXAMPLE** The US President is elected by having states send representatives to the Electoral College. The number depends in part on the state's population.

<i>Reps</i>	No. states	States	<i>Reps</i>	No. states	States
55	1	CA	11	4	AZ, IN, MA, TN
38	1	TX	10	4	MD, MN, MO, WI
29	2	FL, NY	9	3	AL, CO, SC
20	2	IL, PA	8	2	KY, LA
18	1	OH	7	3	CT, OK, OR
16	2	GA, MI	6	6	AR, IA, KS, MS, NV, UT
15	1	NC	5	3	NE, NM, WV
14	1	NJ	4	5	HI, ID, ME, NH, RI
13	1	VA	3	8	AK, DE, DC, MT, ND, SD, VT, WY
12	1	WA			

The table above gives the numbers for the 2020 election; all of a state's representatives vote for the same person (we will ignore some fine points). The Partition Problem asks if there could be a tie.

2.32 **PROBLEM (Linear Programming)** Optimize a linear function $F(x_0, \dots, x_n) = c_0x_0 + \dots + c_nx_n$ subject to linear constraints, ones of the form $a_{i,0}x_0 + \dots + a_{i,n}x_n \leq b_i$ or $a_{i,0}x_0 + \dots + a_{i,n}x_n \geq b_i$.

2.33 **EXAMPLE** Maximize $F(x_0, x_1) = x_0 + 2x_1$ subject to $4x_0 + 3x_1 \leq 24$, $x_1 \leq 4$, $x_0 \geq 0$ and $x_1 \geq 0$. The shaded region has the points that satisfy all the inequalities; these are said to be ‘feasible’ points.



The level lines of F indicate that the maximum is at $(x_0, x_1) = (3, 4)$.

2.34 **PROBLEM (Crossword)** Given an $n \times n$ grid and a set of $2n$ -many strings, each of length n , decide if the words can be packed into the grid.

- 2.35 EXAMPLE Can we pack the words AGE, AGO, BEG, CAB, CAD, and DOG into a 3×3 grid?

2.36 ANIMATION: Instance of the Crossword problem

- 2.37 PROBLEM (Fifteen Game) Given an $n \times n$ grid holding tiles numbered $1, \dots, n^2 - 1$, and a blank, find the minimum number of moves that will put the tile numbers into ascending order. A move consists of switching a tile with an adjacent blank.

This game became popular with $n = 4$ as a toy.



The final three problems, about primes and divisibility, have an impeccable history. No less an authority than Gauss said, “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 that every possible means be explored for the solution of a problem so elegant and so celebrated.”

The three may be hard to tell apart at first glance. But as we understand them today, they differ in the Big- \mathcal{O} behavior of the algorithms to solve them.

- 2.38 PROBLEM (Divisor) Given a number $n \in \mathbb{N}$, find a nontrivial divisor.

We know of no efficient algorithm to find divisors.[†] However, as is so often the case, at this moment we also have no proof that no such algorithm exists.[‡] Not all numbers of a given length are equally hard to factor. The hardest numbers to factor are semiprimes, products of two prime numbers.

- 2.39 PROBLEM (Prime Factorization) Given a number $n \in \mathbb{N}$, produce its decomposition into a product of primes.

Factoring seems to be hard. But what if you only want to know whether a number is prime and don’t care about its factors?

- 2.40 PROBLEM (Primality) Given a number $n \in \mathbb{N}$, determine if it is prime; that is, decide if there are no numbers a that divide n and such that $1 < a < n$.

[†]No efficient algorithm is known on a non-quantum computer. [‡]The presumed difficulty of this problem is at the heart of widely used algorithms in cryptography.

For many years the consensus among experts was that finding a primality testing algorithm that was polytime in the number of digits of the input was very unlikely. After all, for centuries, many of the smartest people in the world had worked on composites and primes, and none of them had produced a fast test.[†]

However, in 2002 M Agrawal, N Kayal, and N Saxena produced such an algorithm, the AKS algorithm.[‡] Today, refinements of their technique run in $\mathcal{O}(n^6)$.

This dramatically illustrates that even though a problem is high profile, and even though many well-respected experts have worked on it, does not mean that the problem will never be solved.

Although opinions of experts have value, nonetheless they can be wrong. People producing a result that gainsays established orthodoxy has happened before and will happen again. One correct proof is all it takes.



Nitin Saxena (b 1981),
Neeraj Kayal (b 1979),
Manindra Agrawal
(b 1966)

V.2 Exercises

- 2.41 Name the prime numbers less than one hundred.
- 2.42 Decide if each is prime. (A) 5 477 (B) 6 165 (C) 6 863 (D) 4 207 (E) 7 689
- ✓ 2.43 Find a proper divisor of each. (A) 31 221 (B) 52 424 (C) 9 600 (D) 4 331 (E) 877
- 2.44 We can specify a propositional logic behavior in a truth table and then produce such a statement in conjunctive normal form.

P	Q	R	
F	F	F	T
F	F	T	T
F	T	F	F
F	T	T	F
T	F	F	T
T	F	T	F
T	T	F	T
T	T	T	F

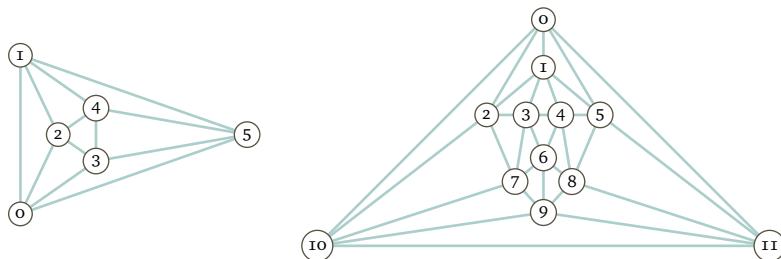
- (A) The two terms P and $\neg P$ are **atoms**. So are Q , $\neg Q$, R , and $\neg R$. Produce a three-atom clause that evaluates to F only on the F - T - F line.
- (B) Produce three-atom clauses for each of the other truth table lines having the value F on the right.
- (C) Take the conjunction of those four clauses and verify that it has the given behavior.

[†]There are a number of probabilistic algorithms that are often used in practice that can test primality very quickly, with an extremely small chance of error. [‡]At the time that they did most of this research, Kayal and Saxena were undergraduates.

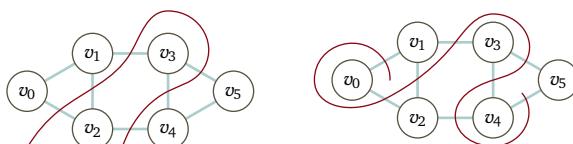
- ✓ 2.45 Decide if each formula is satisfiable.
 - (A) $(P \wedge Q) \vee (\neg Q \wedge R)$
 - (B) $(P \rightarrow Q) \wedge \neg((P \wedge Q) \vee \neg P)$
- ✓ 2.46 Each of the five Platonic solids has a Hamiltonian circuit, as shown.



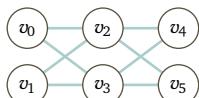
Hamilton used the fourth, the dodecahedron, for his game. Find a Hamiltonian circuit for the third and the fifth, the octahedron and the icosahedron. To make the connections easier to see, below we have grabbed a face in the back of each solid, and expanded it until we could squash the entire shape down into the plane without any edge crossings.



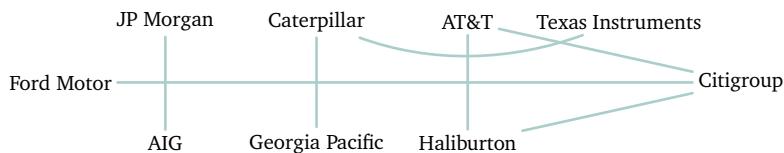
- 2.47 Give a planar map that requires four colors.
- 2.48 (A) The Four Color problem requires that the countries be contiguous, that they not consist of separated regions (that is, components). Give a planar map that consists of separated regions that requires five colors. (B) We also define adjacent to mean sharing a border that is an interval, not just a point. Give a planar map that, without that restriction, would require five colors.
- ✓ 2.49 Example 2.20 exhibits a cut set with six members, as shown on the left. But on the right there are eight cut edges; what's wrong with it?



- 2.50 Find the maximum cut set for this graph.



- ✓ 2.51 This shows interlocking corporate directorships. The vertices are corporations and they are connected if they share a member of their Board of Directors (the data is from 2004).



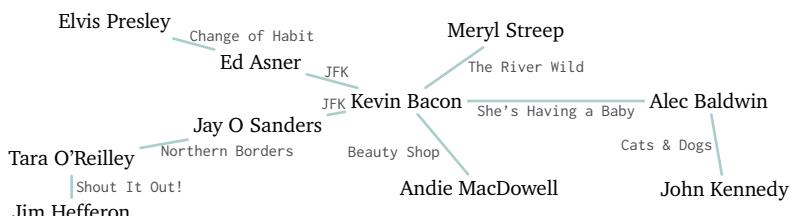
- (A) Is there a path from AT&T to Ford Motor? (B) Can you get from Haliburton to Ford Motor? (c) Can you get from Caterpillar to Ford Motor? (D) JP Morgan to Ford Motor?

2.52 How many edges are there in a Hamiltonian path?

2.53 On some Traveling Salesman problem graphs we can change the edge weights to ensure that an edge is used but on some we cannot.

- (A) A circuit for a Traveling Salesman problem instance is a Hamiltonian path. Produce an undirected graph without loops on which there is at least one Hamiltonian circuit, but containing an edge that belongs to no such circuit.
- (B) Consider an undirected graph with an edge e through which at least one Hamiltonian circuit runs. Fix edge weights for all other edges. Show that there is an edge weight for e such that any solution for the Traveling Salesman problem includes e .

- ✓ 2.54 A popular game extends the Vertex-to-Vertex Path problem by counting the degrees of separation. Below is a portion of the movie connection graph, where actors are connected if they have ever been together in a movie.



A person's **Bacon number** is the number of edges connecting them to Bacon, or infinity if they are not connected. The game *Six Degrees of Kevin Bacon* asks: is everyone connected to Kevin Bacon by at most six movies?

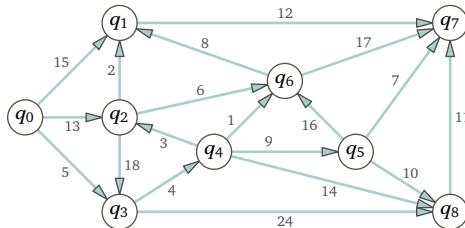
- (A) What is Elvis's Bacon number?
 (B) John Kennedy's (no, it is not that John Kennedy)?
 (C) Bacon's?
 (D) How many movies separate me from Meryl Streep?
- ✓ 2.55 This Knapsack instance has no solution when the weight bound is $B = 73$ and the value target is $T = 140$.

Item	a	b	c	d	e
Weight	21	33	49	42	19
Value	50	48	34	44	40

Verify that by brute force, by checking every possible packing attempt.

2.56 Using the data in Example 2.31, decide if there could be a tie in the 2020 Electoral College.

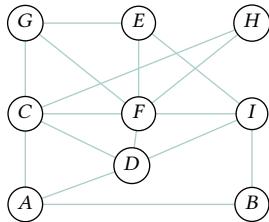
2.57 Find the shortest path in this graph



(A) from q_2 to q_7 , (B) from q_0 to q_8 , (C) from q_8 to q_0 .

2.58 The Subset Sum instance with $S = \{21, 33, 49, 42, 19\}$ and target $T = 114$ has no solution. Verify that by brute force, by checking every possible combination.

- ✓ 2.59 What shape is a 3-clique? A 2-clique?
- 2.60 How many edges does a k -clique have?
- ✓ 2.61 The **Course Scheduling** problem starts with a list of students and the classes that they wish to take, and then finds how many time slots are needed to schedule the classes. If there is a student taking two classes then those two will not be scheduled to meet at the same time. Here is an instance: a school has classes in Astronomy, Biology, Computing, Drama, English, French, Geography, History, and Italian. After students sign up, the graph below shows which classes have an overlap. For instance Astronomy and Biology share at least one student while Biology and Drama do not.



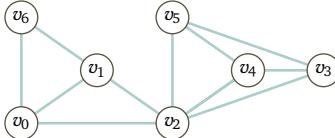
What is the minimum number of class times that we must use? In graph coloring terms, we define that classes meeting at the same time are the same color and we ask for the minimum number of colors needed so that no two same-colored vertices share an edge. (A) Show that no three-coloring suffices. (B) Produce a four-coloring.

2.62 If a Boolean expression F is satisfiable, does that imply that its negation $\neg F$ is not satisfiable?

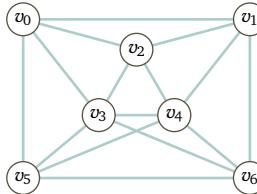
2.63 Some authors define the **Satisfiability** problem as: given a finite set of propositional logic statements, not just one statement, find if there is a single

input tuple b_0, \dots, b_{j-1} , where each b_i is either T or F , that satisfies them all. Show that this is equivalent to the definition given in Problem 2.9.

- ✓ 2.64 Find all 3-cliques in this graph.

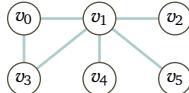


- 2.65 Is there a 3-clique in this graph? A 4-clique? A 5-clique?

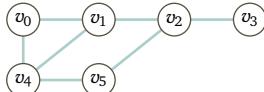


- 2.66 Recall that **Vertex Cover** inputs a graph $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ and a number $k \in \mathbb{N}$, and asks if there is a subset S of at most k vertices such that for each edge at least one endpoint is an element of S . The **Independent Set** problem inputs a graph and a number $\hat{k} \in \mathbb{N}$ and asks if there is a subset \hat{S} with at least \hat{k} vertices such that for each edge at most one endpoint is in \hat{S} . The two are obviously related.

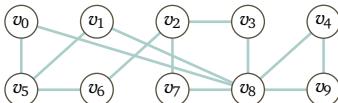
- (A) In this graph find a vertex cover S with $k = 2$ elements. Find an independent set with $\hat{k} = 4$ elements.



- (B) In this graph find a vertex cover with $k = 3$ elements, and an independent set with $\hat{k} = 3$ elements.



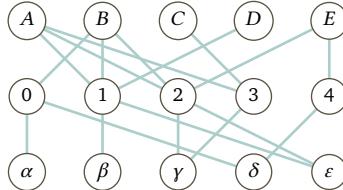
- (C) In this graph find a vertex cover S with $k = 4$ elements. Find an independent set \hat{S} with $\hat{k} = 6$ elements.



- (D) Prove that S is a vertex cover if and only if its complement $\hat{S} = \mathcal{N} - S$ is an independent set.

- ✓ 2.67 A college department has instructors A, B, C, D , and E . They need placing into courses 0, 1, 2, 3, and 4. The available time slots are $\alpha, \beta, \gamma, \delta$, and ε . This

shows which instructors can teach which courses, and which courses can run in which slots.



For example, instructor A can only teach courses 1, 2, and 3. And, course 0 can only run at time α or time δ . Verify that this is an instance of the Three-dimensional Matching problem and find a match.

- 2.68 Consider Three Dimensional Matching, Problem 2.22. Let $X = \{a, b, c\}$, $Y = \{b, c, d\}$, and $Z = \{a, d, e\}$. (A) List the elements of $M = X \times Y \times Z$. (B) Is there a three element subset \hat{M} whose triples have the property that no two of them agree on any coordinate?

SECTION

V.3 Problems, algorithms, and programs

Now, with many examples in hand, we will briefly reflect on problems and solutions. We will keep this discussion on an intuitive level only—indeed, many of these things have no widely accepted precise definition.

A problem is a job, a task. It is a usually uniform family of tasks, with an unbounded number of instances. For a sense of ‘family’, contrast the general **Shortest Path** problem with that of finding the shortest path between Los Angeles and New York. The first is a family while the second is an instance. We are more likely to talk about the family, both because any conclusion about the first subsumes the second and also because the first feels more natural.[†] We are focused on problems that can be solved with a mechanism, although we continue to be interested to learn that a problem cannot be solved mechanically at all.

An algorithm is an effective way to solve a problem.[‡] An algorithm is not a program, although it should be described in a way that is detailed enough that implementing it is routine for an experienced professional. The description should be complete enough to analyze its Big \mathcal{O} behavior.

One subtle point about algorithms is that while they are abstractions, they are nonetheless based on a computing model. An algorithm that is based on a

[†]There are interesting problems with only one task, such as computing the digits of π . [‡]There is no widely-accepted formal definition of ‘algorithm’. Whatever it is, it fits between ‘mathematical function’ and ‘computer program’. For example, a ‘sort’ routine takes in a set of items and returns the sorted sequence. This task, this input-output behavior, could be accomplished using different algorithms: merge sort, heap sort, etc. So the best handle that we have is informal—an ‘algorithm’ is an equivalence class of programs (i.e., Turing machines), where two programs are equivalent if they do a task in essentially the same way, whatever “essentially” means.

Turing machine model for adding one to an input would be very different than an algorithm to do the same task on a model that is like an everyday computer.

An example of an unusual computing model that an algorithm could target is distributed computation. For instance, Science United is a way for anyone with a computer and an Internet connection to help scientific projects by donating computing time. These projects do research in astronomy, physics, biomedicine, mathematics, and environmental science. Contributors install a free program that runs jobs in the background. This is massively parallel computation.[†]

A program is an implementation of an algorithm, expressed in a formal computer language and often designed to be executed on a specific computing platform.

Here is an illustration of the differences between the problems, algorithms, and programs. We've discussed the Primality problem. One algorithm is, given an input $n > 1$, to try every $k \in (1 .. n)$ to see if it divides n . We could implement that algorithm with a program written in Racket.

Problem types We have already seen **function problems**. These ask that an algorithm has a single output for each input. A example is the **Prime Factorization** problem, which takes in a natural number and returns its prime decomposition. Another example is the problem of finding the greatest common divisor, where the input is a pair of natural numbers and the output is a natural number.

Another problem type is the **optimization problem**. These ask for a solution that is best according to some metric. The **Shortest Path** problem is one of these, as is the **Minimal Spanning Tree** problem.

A perhaps less familiar problem type is the **search problem**. For these, while there may be many solutions in the search space, the algorithm can stop when it has found one. An example inputs a Propositional Logic statement and outputs any truth table line witnessing that the statement is satisfiable. A second is the problem that inputs a weighted graph, two vertices, and a bound $B \in \mathbb{R}$, and finds a path between the vertices that costs less than the bound. Another example is that of finding a B -coloring for a graph. Still another is the **Knapsack** problem. In all of these, we want to find if there is a way to solve the problem, such as a way to pack the knapsack, and if there is at least one then we are done.

A **decision problem** is one with a ‘Yes’ or ‘No’ answer.[‡] The first problem that we saw, the *Entscheidungsproblem*, is one of these.[#] We have also seen decision problems in conjunction with the **Halting** problem, such as the problem of determining, given an index e , whether there is an input such that ϕ_e will output a seven. In this chapter we saw the **Primality** problem, the problem of deciding whether a given natural number is prime, as well as the **Subset Sum** problem.

Often a decision problem is expressed as a **language decision problem**, where

[†]There are now coming up on a million volunteers offering computing time. To join them, visit <https://scienceunited.org/>. [‡]Although a decision problem calls for producing a function of a kind, a Boolean function, they are important enough to be a separate category. [#]Recall that the word is German for “decision problem” and that it asks for an algorithm to decide, given a mathematical statement, whether that statement is true or false.

we are given a language and asked for an algorithm to decide if the input is a member of that language. We will see many examples later but just to give one here: we can express the task of deciding the primality of a natural number, the **Primality problem**, as that of deciding membership in the set of bitstrings $\{\sigma \in \mathcal{P}(\mathbb{B}) \mid \sigma \text{ is the binary representation of a prime } n \in \mathbb{N}\}$.

This relates to the discussion from the Languages section, on page 141, about the distinction between deciding a language and recognizing it. We are ready for the following.

- 3.1 DEFINITION** A language \mathcal{L} is **decided** by a Turing machine, or is **Turing machine decidable**, if the function computed by that machine is the characteristic function of the language. The language is **recognized**, or **accepted**, by a machine when for each input $\sigma \in \mathbb{B}^*$, if $\sigma \in \mathcal{L}$ then the machine returns 1, while if $\sigma \notin \mathcal{L}$ then either the machine does not halt or it returns something other than 1.

Restated, the Turing machine \mathcal{P} decides the language \mathcal{L} if it has this input-output behavior.

$$\phi_{\mathcal{P}}(\sigma) = \mathbb{1}_{\mathcal{L}}(\sigma) = \begin{cases} 1 & - \text{if } \sigma \in \mathcal{L} \\ 0 & - \text{otherwise} \end{cases}$$

Thus, if \mathcal{P} decides the language then it halts for all inputs.

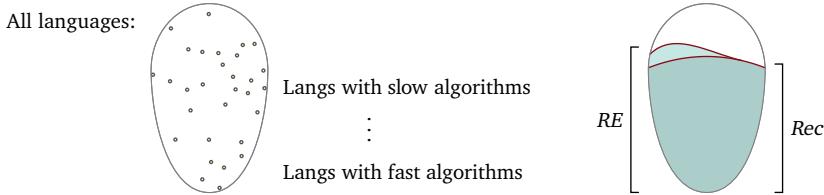
- 3.2 REMARK** One reason that we are interested in language membership decisions comes from practice. A language compiler must recognize whether a given source file is a member of the language.

Another reason is that Finite State machines decide languages. We did lots of those, such as producing a machine that decides if an input string is a member of $\mathcal{L} = \{\sigma \in \{a, b\}^* \mid \sigma \text{ contains at least two } b's\}$ or proving that no Finite State machine can determine membership in $\{a^n b^n \mid n \in \mathbb{N}\}$. Thus, if we want to compare Finite State machines with other machines then we must compare which languages they can decide, because that's all that they can do. Pushdown machines are the same.

Still another reason is that in many contexts stating a problem in this way is natural, as we saw with the **Halting problem**.

Distinctions between problem types can be fuzzy. In addition, often if we have a task then we could describe it with more than one type. An instance is the task of determining the evenness of a natural number. We could express it as the function problem ‘given n , return its remainder on division by 2’, or as the language decision problem of determining membership in $\mathcal{L} = \{2k \mid k \in \mathbb{N}\}$.

When we have a choice of problem types, we prefer language decision problems. It is our default interpretation of ‘problem’ and we will focus on them in the rest of the book. In addition, we will be sloppy about the distinction between the decision problem for a language and the language itself; for instance, we will write \mathcal{L} for a problem.



3.3 FIGURE: Both of these show the collection of languages, $\mathcal{P}(B^*)$, which we often call the ‘problems’. On the left, the dots in the blob emphasize that this is a collection of separate sets, not a continuum. It is drawn with quickly-solvable problems, those with a fast decider, at the bottom. But there is a catch. On the right the shaded collection *Rec* consists of the Turing computable languages. Similarly, *RE* consists of the languages that are computably enumerable. So this diagram makes the point that not all languages have a decider or a recognizer—some languages are perfectly good problems, but they are unsolvable.

Many problem types can be recast as decision problems.

- 3.4 EXAMPLE The **Satisfiability** problem, as we have stated it, is a decision problem. We can recast it as the problem of determining membership in the language $SAT = \{E \mid E \text{ is a satisfiable propositional logic statement}\}$. This recasting is trivial, suggesting that the language recognition problem form is a natural way to describe the underlying task.

A pattern that we shall see is to recast optimization problems as language decision problems, by parametrizing.

- 3.5 EXAMPLE The **Chromatic Number** problem inputs a graph and returns a minimal $k \in \mathbb{N}$ such that the graph is k -colorable. Recast it by considering the family of languages, $\mathcal{L}_B = \{\mathcal{G} \mid \mathcal{G} \text{ that has a } B\text{-coloring}\}$ for $B \in \mathbb{N}^+$. Here the parameter is B . If we could solve these language decision problems then we could compute the minimal chromatic number by testing $B = 1, B = 2$, etc., until we find the smallest one for which $\mathcal{G} \in \mathcal{L}_B$.

- 3.6 EXAMPLE The **Traveling Salesman** problem is an optimization problem. Recast it as a sequence of language decision problems as above: consider a parameter $B \in \mathbb{N}$ and define $\mathcal{T}\mathcal{S}_B = \{\mathcal{G} \mid \text{the graph } \mathcal{G} \text{ has a circuit of length no more than } B\}$.

What is important about these recastings of optimization problems is that they preserve polytime solvability. For instance, for the **Traveling Salesman** problem, if for each B we could solve member of the family $\mathcal{T}\mathcal{S}_B$ in time $\mathcal{O}(n^k)$ then looping through $B = 1, B = 2$, etc., will solve the optimization problem in polytime, namely time $\mathcal{O}(n^{k+1})$.

Our last example is a preview of the work that we will do later in the chapter to relate problems.

- 3.7 EXAMPLE Consider the **Satisfying Assignment** problem that takes in a boolean expression $E(P_0, P_1 \dots P_k)$ and returns an assignment of truth values for P_0, P_1, \dots that makes E evaluate to T , or a flag that there is no such assignment. This is a

search problem in that there may be many such assignments but we stop if we find one.

We will show that this problem is equivalent to the **Satisfiability** language decision problem in this sense: if we could solve one in polytime then we could also solve the other in polytime. The easy direction is that if we had an algorithm that solves **Satisfying Assignment** in polytime then given an expression E we can apply that algorithm to it and know in polytime whether E is satisfiable.

For the other direction suppose that we have a polytime algorithm for **SAT**. Given an expression E , apply **SAT**'s algorithm to determine whether E is satisfiable at all. If so then we need to return an assignment. First fix P_0 at F and run **SAT**'s algorithm. If $E(P_0 = F, P_1, P_2 \dots P_k)$ is not satisfiable then our assignment needs $P_0 = T$, and otherwise $P_0 = F$ suffices. Now with a suitable P_0 we iterate, next fixing $P_1 = F$, etc. This builds an assignment by wrapping a loop of length k around the polytime algorithm for **SAT**, and therefore the algorithm as a whole is polytime.

One more remark. Sometimes selecting the problem type requires judgment. Consider the task of finding rational number roots of a polynomial. As a function problem it is ‘given a polynomial p , return its rational roots’, but a language decision problem restatement is ‘decide if $\langle p, r \rangle$, belongs to the set of pairs consisting of a polynomial and one of its rational roots’. The second just ask us to plug r into p and does not seem to capture the essential difficulty.

Statements and representations To be complete, the description of a problem must include the form of the inputs and outputs. For instance, if we state a problem as: ‘input two numbers and output their midpoint’ then we have not fully specified what needs to be done. The input or output might use strings representing decimal numbers, or might be floating point representations, or even might be in unary.

This matters in that the input’s form can change the algorithm that we choose or its runtime behavior. Suppose that we must decide whether a number is divisible by four. If the input is in binary then the algorithm is immediate: a number is divisible by four if and only if in its final two bits are 00 .[†] In contrast, if it is in unary then we may scan the 1’s, keeping track of the current remainder modulo 4.

On the other hand, the representation doesn’t matter in the sense that if we have an algorithm for one representation then we can solve the problem for other representations by translating. For example, we could do the divisible-by-four problem with unary input by converting to binary and then applying the binary algorithm.[‡] Typically the costs of different representations don’t change the Big \mathcal{O} runtime behavior. For example we might have a graph algorithm whose run time is $\mathcal{O}(n \lg n)$. Even for this minimal time, we can find a representation for the input graphs, such as where inputting takes $\mathcal{O}(n)$ time, that leaves the algorithm analysis conclusion unchanged at $\mathcal{O}(n \lg n)$.

[†]Thus, on a Turing machine, if when the machine starts the head is under the final character, then the machine does not even need to read the entire input to decide the question. The algorithm runs in time independent of the input length. [‡]That is, the unary case reduces to the binary one.

With this in mind, we will take the view, which we call **Lipton's Thesis**, that everything of interest can be represented with reasonable efficiency by bitstrings.[†] This applies to all of the mathematical problems stated earlier. But it also applies to cases that may seem less natural, such as that we can use bitstrings to represent with sufficient fidelity Beethoven's 9th Symphony or an exquisite Old Master.[‡]



3.8 FIGURE: *Basket of Fruit* by Caravaggio (1571–1610)

Consequently, in practice researchers often do not mention representations. We may describe the Shortest Path problem as, “Given a weighted graph and two vertices . . .” in place of the more complete, “Given the following reasonably efficient bitstring representation of a weighted graph \mathcal{G} and vertices v_0 and v_1 , . . .” Outside of this section we also do this, leaving implementation details to a programmer. (When we do discuss representations, we use $\text{str}(x)$ to denote a convenient, reasonably efficient, bitstring representation of x .[#]) Basically, the representation details do not affect the outcome of our analysis, much.

- 3.9 **REMARK** There is a caveat. We have seen that conflating $\{n \in \mathbb{N} \mid n \text{ is prime}\}$ with $\{\sigma \in \mathbb{B}^* \mid \sigma \text{ represents a prime number}\}$ can cause confusion. The distinction between thinking of an algorithm as inputting a number and thinking of it as inputting the string representation of a number is the basis for describing the Big \mathcal{O} behavior of that algorithm as pseudopolynomial. This is because the binary representation of a number n takes $\mathcal{O}(\lg n)$ bits.

[†]‘Reasonable’ means that it is not so inefficient as to greatly change the big- \mathcal{O} behavior. [‡]This is in a way like Church’s Thesis. We cannot prove it but our experience with digital reproduction of music, movies, etc., argues that it is so. [#]Many authors use diamond brackets to stand for a representation, as in $\langle \mathcal{G}, v_0, v_1 \rangle$. Here, we reserve diamond brackets for sequences.

V.3 Exercises

- ✓ 3.10 For each of these, list three examples and then—speaking informally, since some of them do not have formal definitions—describe the difference between them and an algorithm. (A) a heuristic (B) pseudocode (C) a Turing machine (D) a flowchart (E) source code (F) an executable (G) a process
- 3.11 Your friend asks, “So, if a problem is essentially a set of strings, what constitutes a solution?” Answer them.
- 3.12 What is the difference between a decision problem and a language decision problem?
- 3.13 As an illustration of the thesis that even surprising things can be represented reasonably efficiently and with reasonable fidelity in binary, we can do a simple calculation. (A) At 30 cm, the resolution of the human eye is about 0.01 cm. How many such pixels are there in a photograph that is 21 cm by 30 cm? (B) We can see about a million colors. How many bits per pixel is that? (C) How many bits for the photo, in total?
- 3.14 Name something important that cannot be represented in binary.
- ✓ 3.15 True or false: any two programs that implement the same algorithm must compute the same function. What about the converse?
- 3.16 Some tasks are hard to express as a language decision problem. Consider sorting the characters of a string into ascending order. Briefly describe why each of these language decision problems fails to capture the task’s essential difficulty. (A) $\{\sigma \in \Sigma^* \mid \sigma \text{ is sorted}\}$ (B) $\{\langle\sigma, p\rangle \mid p \text{ is a permutation that orders } \sigma\}$
- ✓ 3.17 For each language decision problem, name three members of the set, if there are three, and then sketch an algorithm solving it.
- (A) $\mathcal{L}_0 = \{\langle n, m \rangle \in \mathbb{N}^2 \mid n + m \text{ is a square and one greater than a prime}\}$
 - (B) $\mathcal{L}_1 = \{\sigma \in \{0, \dots, 9\}^* \mid \sigma \text{ represents in decimal a multiple of 100}\}$
 - (C) $\mathcal{L}_2 = \{\sigma \in \mathbb{B}^* \mid \sigma \text{ has more 1's than 0's}\}$
 - (D) $\mathcal{L}_3 = \{\sigma \in \mathbb{B}^* \mid \sigma^R = \sigma\}$
- 3.18 Solve the language decision problem for (A) the empty language, (B) the language \mathbb{B} , and (C) the language \mathbb{B}^* .
- 3.19 For each language, sketch an algorithm that solves the language decision problem.
- (A) $\{\sigma \in \mathbb{B}^* \mid \sigma \text{ matches the regular expression } a^*ba^*\}$
 - (B) The language defined by this grammar
- $$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA \mid \epsilon \\ B &\rightarrow bB \mid \epsilon \end{aligned}$$
- 3.20 Solve each decision problem about Finite State machines, \mathcal{M} , by producing an algorithm. (A) Given \mathcal{M} , decide if the language accepted by \mathcal{M} is empty.

(B) Decide if the language accepted by \mathcal{M} is infinite. (C) Decide if $\mathcal{L}(\mathcal{M})$ is the set of all strings, Σ^* .

3.21 For each language decision problem, give an algorithm that runs in $\mathcal{O}(1)$.

(A) The language of minimal-length binary representations of numbers that are nonzero.

(B) The binary representations of numbers that exceed 1000.

3.22 In a graph, a **bridge edge** is one whose removal disconnects the graph. That is, there are two vertices that before the bridge is removed are connected by a path, but are not connected after it is removed. (More precisely, a **connected component** of a graph is a set of vertices that can be reached from each other by a path. A bridge edge is one whose removal increases the number of connected components.) The problem is: given a graph, find a bridge. Is this a function problem, a decision problem, a language decision problem, a search problem, or an optimization problem?

- ✓ 3.23 For each, give the categorization that best applies: a function problem, a decision problem, a language decision problem, a search problem, or an optimization problem. (A) The **Graph Connectedness** problem, which inputs a graph and decides whether for any two vertices there is a path between them. (B) The problem that inputs two natural numbers and returns their least common multiple. (C) The **Graph Isomorphism** problem that inputs two graphs and determines whether they are isomorphic. (D) The problem that takes in a propositional logic statement and returns an assignment of truth values to its inputs that makes the statement true, if there is such an assignment. (E) The **Nearest Neighbor** problem that inputs a weighted graph and a vertex, and returns a vertex nearest the given one that does not equal the given one. (F) The **Discrete Logarithm** problem: given a prime number p and two numbers $a, b \in \mathbb{N}$, determine if there is a power $k \in \mathbb{N}$ so that $a^k \equiv b \pmod{p}$. (G) The problem that inputs a bitstring and decides if the number that it represents in binary will, when converted to decimal, contain only odd digits.

- ✓ 3.24 For each, give the characterization that best applies: a function problem, a decision problem, a language decision problem, a search problem, or an optimization problem. (A) The **3-SAT** problem, Problem 2.10 (B) The **Divisor** problem, Problem 2.38 (C) The **Prime Factorization** problem, Problem 2.39 (D) The **F-SAT** problem, where the input is a propositional logic expression and the output is either an assignment of T and F to the expression's variables that makes it evaluate to T , or the string **None**. (E) The **Primality** problem, Problem 2.40

3.25 Express each task as a language decision problem. Include in the description explicit mention of the string representation. (A) Decide whether a number is a perfect square. (B) Decide whether a triple $\langle x, y, z \rangle \in \mathbb{N}^3$ is a Pythagorean triple, that is, whether $x^2 + y^2 = z^2$. (C) Decide whether a graph has an even number of edges. (D) Decide whether a path in a graph has any repeated vertices.

- ✓ 3.26 Recast each as a language decision problem. Include explicit mention of the string representation. (A) Given a natural number, do its factors add to more than twice the number? (B) Given a Turing machine and input, does the machine halt on the input in less than ten steps? (C) Given a propositional logic statement, are there three different assignments that evaluate to T ? That is, are there more than three lines in the truth table that end in T ? (D) Given a weighted graph and a bound $B \in \mathbb{R}$, for any two vertices is there a path from one to the other with total cost less than the bound?
- 3.27 Recast each in language decision terms. Include explicit mention of the string representation. (A) Graph Colorability, Problem 2.7, (B) Euler Circuit, Problem 2.4, (C) Shortest Path, Problem 2.5.
- 3.28 Restate the Halting problem as a language decision problem.
- ✓ 3.29 As stated, the Shortest Path problem, Problem 2.5, is an optimization problem. Convert it into a parametrized family of decision problems. *Hint:* use the technique outlined following the Traveling Salesman problem, Problem 2.3.
- ✓ 3.30 Express each optimization problem as a parametrized family of language decision problems. (A) Given a Fifteen Game board, find the least number of slides that will solve it. (B) Given a Rubik's cube configuration, find the least number of moves to solve it. (C) Given a list of jobs that must be accomplished to assemble a car, along with how long each job takes and which jobs must be done before other jobs, find the shortest time to finish the entire car.
- 3.31 As stated, the Hamiltonian Circuit problem is a decision problem. Give a function version of this problem. Also give an optimization version.
- 3.32 The different problem types are related. Each of these inputs a square matrix M with more than 3 rows, and relates to a 3×3 submatrix (form the submatrix by picking three rows and three columns, which need not be adjacent). Characterize each as a function problem, a decision problem, a search problem, or an optimization problem. (A) Find a submatrix that is invertible. (B) Decide if there is an invertible submatrix. (C) Return a submatrix that is invertible, or the string 'None'. (D) Return a submatrix whose determinant has the largest absolute value. Also give a language for an associated language decision problem.
- 3.33 Recast each function problem as a decision problem.
 - (A) The problem that inputs two natural numbers and returns their product.
 - (B) The Nearest Neighbor problem, that inputs a weighted graph and a vertex and returns the vertex nearest the given one, but not equal to it.
- 3.34 The Linear Programming problem is described on page 280. Give a version that is a (A) language decision problem, (B) search problem, (C) function problem, and (D) optimization problem. (For some parts there is more than one sensible answer.)
- 3.35 An **independent set** in a graph is a collection of vertices such that no two are connected by an edge. Give a version of the problem of finding an

independent set that is a (A) a decision problem, (B) language decision problem, (C) search problem, (D) function problem, and (E) optimization problem. (For some parts there is more than one reasonable answer.)

3.36 Give an example of a problem where the decision variant is solvable quickly but the search variant is not.

3.37 Let $\mathcal{L}_F = \{\langle n, B \rangle \in \mathbb{N}^2 \mid \text{there is an } m \in \{2, \dots, B\} \text{ that divides } n\}$ and consider its language decision problem. (A) Show that $\langle d, B \rangle \in \mathcal{L}_F$ if and only if B is greater than or equal to the least prime factor of d . (B) Conclude that you can use a solution to the language recognition problem to solve the search problem that is given a number and returns a prime factor of that number.

- ✓ 3.38 Show how to use an algorithm that solves the **Shortest Path** problem to solve the **Vertex-to-Vertex Path** problem. How to use it on graphs that are not weighted?
- ✓ 3.39 Show that with an algorithm that quickly solves the **Subset Sum** problem, Problem 2.25, we can quickly solve the associated function problem of finding the subset.
- 3.40 Show how to use an algorithm that solves **Vertex-to-Vertex Path** problem to solve the **Graph Connectedness** problem, which inputs a graph and decides whether that graph is connected, so that for any two vertices there is a path between them.

SECTION

V.4 **P**

We have said that we often blur the distinction between the problem of deciding membership in a language \mathcal{L} and the language itself. So to express that we are studying problems of a certain type we may say we are studying languages.

- 4.1 **DEFINITION** A **complexity class** is a collection of languages.

The term ‘complexity’ is there because these collections are often associated with some resource specification, so that for instance one class is the collection of languages that are accepted by a Turing machine in quadratic time.[†]

- 4.2 **EXAMPLE** One complexity class is the collection of languages for which there is a deciding Turing machine that runs in time $\mathcal{O}(n^2)$. Thus $\mathcal{C} = \{\mathcal{L}_0, \mathcal{L}_1, \dots\}$, where each \mathcal{L}_j is decided by some machine \mathcal{P}_{ij} , for which the function f relating the size of the machine’s input $|\sigma|$ to the number of steps that the machine takes to finish is quadratic, that is, f is $\mathcal{O}(n^2)$.

[†]There are other definitions of complexity class. Some authors require that in a class the characteristic function of each language can be computed under some resource specification. This has implications—if all of the members of a class must be computable by Turing machines then each class is countable. Here, we only say that it is a collection so that the definition is maximally general.

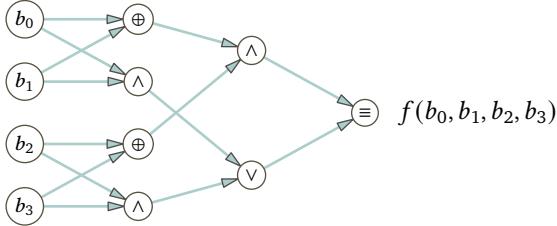
- 4.3 EXAMPLE Another is the collection of languages accepted by some Turing machine that uses only logarithmic space. That is, for such a machine, with input string σ the function f relating $|\sigma|$ to the maximum number of tape squares that the machine visits in deciding a string of that length is logarithmic, $f \in \mathcal{O}(\lg)$.

Two points bear explication. As to the computing machine, researchers study not just Turing machines but other types of machines as well, including nondeterministic Turing machines and Turing machines with access to an oracle for random numbers. And as for the resource specification, it often involve bounds on the time or space behavior. But a class could instead be, for instance, the complement of $\mathcal{O}(n^2)$, so the specification isn't always a bound.[†]

Definition The complexity class that we introduce now is the most important one. It is the collection of problems that under Cobham's Thesis we take to be tractable.

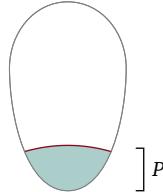
- 4.4 DEFINITION A language decision problem \mathcal{L} is a member of the class P if there is an algorithm to decide membership in \mathcal{L} that on a deterministic Turing machine runs in polynomial time.
- 4.5 EXAMPLE The problem $\mathcal{L} = \{\mathcal{G} \mid \text{there is a path between any two vertices}\}$ of deciding whether a given graph is connected is a member of P . To verify this, we must produce an algorithm that decides membership in this language, and that runs in polynomial time. One is to do a breadth first search of the graph, which has a runtime that is cubic in the number of nodes.
- 4.6 EXAMPLE Another member of P is the problem of deciding whether two numbers are relatively prime, $\{\langle n_0, n_1 \rangle \in \mathbb{N}^2 \mid \text{their greatest common divisor is } 1\}$. As before, to verify that this language is a member of P we produce an algorithm that determines membership and that runs in polytime. Euclid's algorithm fits the bill; it solves this problem and has runtime $\mathcal{O}(\lg(\max(n_0, n_1)))$.
- 4.7 EXAMPLE Still another member of P is the **String Search** problem of deciding substring-ness, $\{\langle \sigma, \tau \rangle \in \Sigma^* \mid \sigma \text{ is a substring of } \tau\}$. (Often in practice τ is very long and is called the **haystack** while σ is short and is the **needle**.) The algorithm that first tests σ at the initial character of τ , then at the next character, etc., has a runtime of $\mathcal{O}(|\sigma| \cdot |\tau|)$, which is $\mathcal{O}(\max(|\sigma|, |\tau|))^2$.
- 4.8 EXAMPLE A **circuit** is a directed acyclic graph. Below, each vertex, called a **gate**, acts as a two input/one output Boolean function. The only exception is that some vertices are **input gates** that provide source bits (below on the left they are $b_0, b_1, b_2, b_3 \in \mathbb{B}$). Edges are called **wires**, \wedge denotes the boolean function 'and', \vee is 'or', \oplus is 'exclusive or', and \equiv is the negation of 'exclusive or', which returns 1 if and only if the two inputs bits are the same.

[†]At this writing there are 546 studied classes but the number changes frequently; see the Complexity Zoo, <https://complexityzoo.net/>.



This circuit returns 1 if the sum of the input bits is a multiple of 3. The **Circuit Evaluation** problem inputs a circuit like this one and computes the output, $f(b_0, b_1, b_2, b_3)$. This problem is a member of P .

- 4.9 EXAMPLE Although polytime is a restriction, nonetheless P is a very large collection. More example members: (1) matrix multiplication, taken as a language decision problem for $\{ \langle \sigma_0, \sigma_1, \sigma_2 \rangle \mid \text{they represent matrices with } M_0 \cdot M_1 = M_2 \}$ (2) minimal spanning tree, $\{ \langle G, T \rangle \mid T \text{ is a minimal spanning tree in } G \}$ (3) edit distance, the number of single-character removals, insertions, or substitutions needed to transform between strings, $\{ \langle \sigma_0, \sigma_1, n \rangle \mid \sigma_0 \text{ transforms to } \sigma_1 \text{ in at most } n \text{ edits} \}$.



4.10 FIGURE: This blob contains all language decision problems, all $\mathcal{L} \subseteq \mathbb{B}^*$. Shaded is P .

Two final points. First, if a problem has an algorithm that is $\mathcal{O}(\lg n)$ then that problem is in P . Second, the members of P are problems (actually, languages that represent problems), so it is wrong to say that an algorithm is in P .

Effect of the model of computation A problem is in P if it has an algorithm that is polytime. But algorithms are based on an underlying computing model. Is membership in P dependent on the model that we use?

In particular, our experience with Turing machines gives the sense that they involve a lot of tape shuttling. So we may expect that algorithms directed at Turing machine hardware are slow. However, close analysis with a wide range of alternative computational models proposed over the years shows that while Turing machine algorithms are indeed often slower than related algorithms for other natural models, it is only by a factor of between n^2 and n^4 [†]. That is, if we have a problem for which there is a $\mathcal{O}(n)$ algorithm on another model then we may find that on a Turing machine model it is $\mathcal{O}(n^3)$, or $\mathcal{O}(n^4)$, or $\mathcal{O}(n^5)$. So it is still in P .

[†]We take a model to be ‘natural’ if it was not invented in order to be a counterexample to this.

A variation of Church's thesis, the **Extended Church's Thesis**, posits that not only are all reasonable models of mechanical computation of equal power, but in addition that they are of equivalent speed in that we can simulate any reasonable model of computation[†] in polytime on a probabilistic Turing machine.[‡] Under the extended thesis, a problem that falls in the class P using Turing machines also falls in that class using any other natural models. (Note, however, that this thesis does not enjoy anything like the support of the original Church's Thesis. Also, we know of several problems, including the familiar Prime Factorization problem, that under the Quantum Computing model have algorithms with polytime solutions, but for which we do not know of any polytime solution in a non-quantum model. So Quantum Computing could well provide a counterexample to the extended thesis, if we can produce physical devices matching that model.)

- 4.11 REMARK In recent years a number of researchers claimed to have built devices that achieved **quantum advantage**, to have solved a problem using an algorithm running on a physical quantum device that does not appear to be solvable on a Turing machine or RAM machine-based device in less than centuries.

The claim is the subject of scholarly reservations. For one thing, the advantage depends both on there being a quantum device that accomplishes the task and also on there not being a classical algorithm that is fast. In fact soon after the original claim was made other researchers produced an algorithm for a traditional device that is near parity. Another thing is that on its face, this is not about general purpose computing; the problem solved is exotic and especially suitable to the instruments that researchers have available. There are sound reasons to wonder whether quantum computers will ever be practical physical devices used for everyday problems, although scientists and engineers are making great progress. We will put this aside for being as-yet unsettled but it is worth monitoring developments closely.

Naturalness We give the class P our attention because there are reasons to suppose that it is the best candidate for the collection of problems that have a feasible solution. We close this section with a discussion of those.

The first reason echos the prior subsection. There are many models of computation, including Turing machines, RAM machines, and Racket programs. All of them compute the same set of functions as Turing machines. Further, while their speeds may differ, all of them run within polytime of each other.[#] That makes P invariant under the choice of computing model: if a problem is in P for any of these models then it is in P for all. The fact that Turing machines are our standard is in some ways a historical accident but differences between the runtime behavior of any of these models is lost in the general polynomial sloppiness.

Another reason that P is a natural class is that we'd like that if two things are easy to compute then a simple combination of the two is also easy. More precisely,

[†]One definition of 'reasonable' is "in principle physically realizable" (Bernstein and Vazirani 1997).

[‡]A Turing machine with access to an oracle of random bits. [#]All, that is, of the non-quantum natural models.

fix two total functions $f, g: \mathbb{N} \rightarrow \mathbb{N}$ and consider these.

$$\begin{aligned}\mathcal{L}_f &= \{\text{str}(\langle n, f(n) \rangle) \in \mathbb{B}^* \times \mathbb{B}^* \mid n \in \mathbb{N}\} \\ \mathcal{L}_g &= \{\text{str}(\langle n, g(n) \rangle) \in \mathbb{B}^* \times \mathbb{B}^* \mid n \in \mathbb{N}\}\end{aligned}$$

(Recall that $\text{str}(\dots)$ means that we represent the argument reasonably efficiently as a bitstring.) With that recasting of functions as languages, P is closed under function addition, scalar multiplication by an integer, subtraction, multiplication, and composition. It is also closed under language concatenation and the Kleene star operator. It is the smallest nontrivial class with these appealing properties.

But the main reason that P is our candidate is Cobham's Thesis, the contention that a problem is tractable if it has a solution algorithm that runs in polytime. Recall the counterargument that a problem whose solution algorithms cannot be improved below a runtime of $\mathcal{O}(n^{1000000})$ is not really tractable. We know such problems exist because we can produce them using diagonalization. But problems produced in that way are artificial. Empirical experience over close to a century of computing is that problems with solution algorithms of very large degree polynomial time complexity do not seem to arise often in practice. We see plenty of problems with solution algorithms that are $\mathcal{O}(n \lg n)$, or $\mathcal{O}(n^3)$, and we see plenty that are exponential, but we just do not see much of $\mathcal{O}(n^{1000000})$.

Moreover, often in the past when a researcher has produced an algorithm for a problem with a runtime that has even a moderately large degree then often, with this foot in the door, over the next few years the community brings to bear an array of mathematical and algorithmic techniques that lower the runtime degree to reasonable size.

Even if the objection to Cobham's Thesis is right and P is too broad, the class would still be useful because if we could show that a problem is not in P then we would have shown that it has no general solution algorithm that is practical.[†]

So Cobham's Thesis, to this point, has largely held up. Insofar as theory should be a guide for practice, this is a good reason to use P as a touchstone for other complexity classes.

V.4 Exercises

- ✓ 4.12 True or False: if the language is finite then the language decision problem is in P .
- ✓ 4.13 Your coworker says something mistaken, “I've got a problem whose algorithm is in P .” They are being a little sloppy with terms; how?
- ✓ 4.14 What is the difference between an order of growth and a complexity class?
- ✓ 4.15 Your friend says to you, “I think that the Circuit Evaluation problem takes exponential time. There is a final vertex. It takes two inputs, which come from

[†]This argument has lost some of its force in recent years with the rise of SAT solvers. These attack problems believed to not be in P and can solve instances of the problems of moderately large size, using only moderately large computing resources. See Extra C.

two vertices, and each of those take two inputs, etc., so that a five-deep circuit can have thirty two vertices.” Help them see where they are wrong.

4.16 In class, someone says to the professor, “Why aren’t all languages in P : I’ll design a Turing machine so that no matter what the input is, it outputs 1. That runs in polytime for sure.” Explain how this is mistaken.

4.17 True or false: if a problem has a logarithmic solution then it is in P .

4.18 True or false: if a language is decided by a machine then its complement is also accepted by some machine.

- ✓ 4.19 Show that the decision problem for $\{\sigma \in \mathbb{B}^* \mid \sigma = \tau^3 \text{ for some } \tau \in \mathbb{B}^*\}$ is in P .
- ✓ 4.20 Show that the language of palindromes, $\{\sigma \in \mathbb{B}^* \mid \sigma = \sigma^R\}$, is in P .
 - 4.21 Sketch a proof that each problem is in P .
 - (A) The τ^3 problem: given a bitstring σ , decide if it has the form $\sigma = \tau^3$.
 - (B) The problem of deciding which Turing machines halt within ten steps.
- ✓ 4.22 Consider the problem of **Triangle**: given an undirected graph, decide if it has a 3-clique, three vertices that are mutually connected.
 - (A) Why is this not the **Clique** problem, from page 278?
 - (B) Sketch a proof that this problem is in P .
- ✓ 4.23 Prove that each problem is in P by citing the runtime of an algorithm that suits.
 - (A) Deciding the language $\{\sigma \in \{a, \dots, z\}^* \mid \sigma \text{ is in alphabetical order}\}$.
 - (B) Deciding the language of correct sums, $\{\langle a, b, c \rangle \in \mathbb{N}^3 \mid a + b = c\}$.
 - (C) Analogous to the prior item, deciding this language of triples of matrices that give correct products, $\{\langle A, B, C \rangle \mid \text{the matrices are such that } AB = C\}$.
 - (D) Deciding the language of primes, $\{1^k \mid k \text{ is prime}\}$.
 - (E) **Vertex-to-Vertex Path**: $\{\langle G, v_0, v_1 \rangle \mid \text{the graph } G \text{ has a path from } v_0 \text{ to } v_1\}$.
- 4.24 Find which of these are currently known to be in P and which are not.
Hint: you may need to look up the fastest known algorithm.
 - (A) **Shortest Path**
 - (B) **Knapsack**
 - (C) **Euler Path**
 - (D) **Hamiltonian Circuit**
- 4.25 Is the empty language $\{\}$ a member of P ?
- 4.26 The problem of **Graph Connectedness** is: given a finite graph, decide if there is a path from any vertex to any other. Sketch an argument that this problem is in P .
- 4.27 Following the definition of complexity class, Definition 4.1, is a discussion of the additional condition of being computed by some machine under a resource specification, such as a Big \mathcal{O} constraint on time or space.
 - (A) Show that the set of regular languages forms a complexity class, and that it meets this additional constraint.
 - (B) The definition of P uses Turing machines. We can view a Finite State machine as a kind of Turing machine, one that consumes its input one character at a time, never writes to the tape, and, depending on the state that the machine

is in when the input is finished, prints 0 or 1. With that, argue that any regular language is an element of P .

- 4.28 We have already studied the collection RE of languages that are computably enumerable.
- (A) Recast RE as a class of language decision problems.
 - (B) Following Definition 4.1 is a discussion of the additional condition of being computed by a machine under a resource specification. Show that RE also satisfies this condition.
- 4.29 Is P countable or uncountable?
- 4.30 If $\mathcal{L}_0, \mathcal{L}_1 \in P$ and $\mathcal{L}_0 \subseteq \mathcal{L} \subseteq \mathcal{L}_1$, must \mathcal{L} be in P ?
- ✓ 4.31 Is the Halting problem in P ?
- 4.32 A common modification of the definition of Turing machine designates one state as an accepting state. Then the machine decides the language \mathcal{L} if it halts on all input strings, and \mathcal{L} is the set of strings that such that the machine ends in the accepting state. A language is decidable if it is decided by some machine. Prove that every language in P is decidable.
- ✓ 4.33 Draw a circuit that inputs three bits, $b_0, b_1, b_2 \in \mathbb{B}$, and outputs the value of $b_0 + b_1 + b_2 \pmod{2}$.
- 4.34 Prove that the union of two complexity classes is also a complexity class. What about the intersection? Complement?
- ✓ 4.35 Prove that P is closed under the union of two languages. That is, prove that if two languages are both in P then so is their union. Prove the same for the union of finitely many languages.
- 4.36 Prove that P is closed under complement. That is, prove that if a language is in P then so is its set complement.
- 4.37 Prove that the class of languages P is closed under reversal. That is, prove that if a language is an element of P then so is the reversal of that language (which is the language of string reversals).
- 4.38 Show that P is closed under the concatenation of two languages.
- 4.39 Show that P is closed under Kleene star, meaning that if $\mathcal{L} \in P$ then $\mathcal{L}^* \in P$. (*Hint:* $\sigma \in \mathcal{L}^*$ if $\sigma = \epsilon$, or $\sigma \in \mathcal{L}$, or $\sigma = \alpha \bar{\wedge} \beta$ for some $\alpha, \beta \in \mathcal{L}^*$)
- 4.40 Show that this problem is unsolvable: give a Turing machine \mathcal{P} , decide whether it runs in polytime on the empty input. *Hint:* if you could solve this problem then you could solve the Halting problem.
- 4.41 There are studied complexity classes besides those associated with language decision problems. The class FP consists of the binary relations $R \subseteq \mathbb{N}^2$ where there is a Turing machine that, given input $x \in \mathbb{N}$, can in polytime find a $y \in \mathbb{N}$ where $\langle x, y \rangle \in R$.
- (A) Prove that this class closed under function addition, multiplication by a scalar $r \in \mathbb{N}$, subtraction, multiplication, and function composition.

- (b) Where $f: \mathbb{N} \rightarrow \mathbb{N}$ is computable, consider this decision problem associated with the function, $\mathcal{L}_f = \{\text{str}(\langle n, f(n) \rangle) \in \mathbb{B}^* \mid n \in \mathbb{N}\}$ (where the numbers are represented in binary). Assume that we have two functions $f_0, f_1: \mathbb{N} \rightarrow \mathbb{N}$ such that $\mathcal{L}_{f_0}, \mathcal{L}_{f_1} \in P$. Show that the natural algorithm to check for closure under function addition is pseudopolynomial.

- 4.42 Where $\mathcal{L}_0, \mathcal{L}_1 \subseteq \mathbb{B}^*$ are languages, we say that $\mathcal{L}_1 \leq_p \mathcal{L}_0$ if there is a function $f: \mathbb{B}^* \rightarrow \mathbb{B}^*$ that is computable, total, that runs in polytime, and so that $\sigma \in \mathcal{L}_1$ if and only if $f(\sigma) \in \mathcal{L}_0$. Prove that if $\mathcal{L}_0 \in P$ and $\mathcal{L}_1 \leq_p \mathcal{L}_0$ then $\mathcal{L}_1 \in P$.

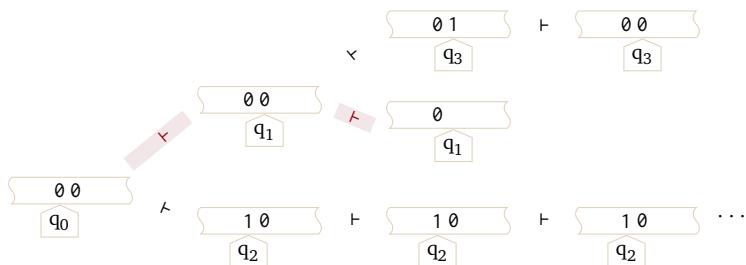
SECTION

V.5 NP

Recall that a Finite State machine is nondeterministic if from a present configuration and input it may pass to more than one next configuration, or one, or zero. We can make a nondeterministic Turing machine by doing the same. Here is one having two instructions starting with q_0 and 0 so if the machine is in q_0 and it reads a 0 on the tape then it is legal both to go to state q_2 and to state q_1 .

$$\mathcal{P} = \{q_001q_2, q_00Rq_1, q_10Bq_1, q_101q_3, q_211q_2, q_310q_3\}$$

For such a machine the computational history can be more than a line, it can be a tree. Below is part of the tree for machine \mathcal{P} and input 00, with the middle branch highlighted.



Nondeterministic Turing machines This modifies the definition of a Turing machine by changing the transition function so that it outputs sets.

- 5.1 **DEFINITION** A **nondeterministic Turing machine** \mathcal{P} is a finite set of instructions $q_p T_p T_n q_n \in Q \times \Sigma \times (\Sigma \cup \{L, R\}) \times Q$, where Q is a finite set of states and Σ is a finite set of tape alphabet characters, which contains at least two members, including blank, and does not contain the characters L or R. Some of the states, $F \subseteq Q$, are **accepting states**. The association of the present state and tape character with what happens next is given by the transition function, $\Delta: Q \times \Sigma \rightarrow \mathcal{P}((\Sigma \cup \{L, R\}) \times Q)$.

After the definition of the Turing machine and others, we described how they act as a sequence of ‘l’ steps from the initial configuration. Exercise 5.38 asks for a similar description for these machines.

Adding nondeterminism to Turing machines adds some wrinkles. The computation tree might have some branches that halt and some that don’t, or maybe some output 0 while some output 1. The simplest approach is not to define a function computed by a nondeterministic machine but instead to follow our strategy for Finite State machines and describe when the input string is accepted, so that these machines are language deciders.

Of course, we have two mental models for how nondeterministic machines act. One is that the machine is unboundedly parallel and simultaneously computes all of the branches. The other is that the machine guesses which branch to follow—or is told by some demon—and then can deterministically check that branch. For both, the machine accepts an input string if at least one branch ends in an accepting state and does not accept the input if no branch ends in an accepting state.

That final sentence raises two points. First, once some branch accepts the input we might as well stop the computation, so we can take that accepting computations always halt. Second, “no branch ends” seemingly could mean that in a non-accepting computation some branches do not accept because their computation fails to halt. But we are using these machines as language deciders and we shall want to time them, including how long they take to not accept. So we will only define language-deciding when all branches halt.

- 5.2 **DEFINITION** Let \mathcal{P} be a nondeterministic Turing machine such that every branch in the computation tree, every sequence of valid transitions from the starting configuration, is finite. Then \mathcal{P} **accepts** an input string if at least one branch ends in an accepting state and otherwise **rejects** it. The machine **decides a language** \mathcal{L} when for every input string σ , if $\sigma \in \mathcal{L}$ then \mathcal{P} accepts it while if $\sigma \notin \mathcal{L}$ then \mathcal{P} rejects it.

For Finite State machines, nondeterminism does not make any difference in that a language is decided by some nondeterministic Finite State machine if and only if it is decided by a deterministic Finite State machine. But Pushdown machines are different: there are languages that a nondeterministic Pushdown machine can decide but that cannot be decided by any deterministic one. Does adding nondeterminism to Turing machines add any new capabilities?

- 5.3 **LEMMA** For Turing machines, deterministic and nondeterministic machines decide the same languages.

Proof One direction is easy. A deterministic Turing machine is a special case of a nondeterministic one. So if a deterministic machine decides a language then a nondeterministic one does also.

Conversely, let the nondeterministic Turing machine \mathcal{P} decide the language \mathcal{L} . Consider a deterministic machine \mathcal{Q} that does a breadth-first search of \mathcal{P} ’s computation tree. We will show that it decides the same language. Fix an input σ .

If \mathcal{P} accepts σ then the search done by \mathcal{Q} will eventually find that node in the computation tree and then \mathcal{Q} accepts σ . If \mathcal{P} does not accept σ then every branch in its computation tree halts in a state that is not accepting. There is a longest such branch by König's lemma. So the breadth-first search will eventually exhaust all branches and then \mathcal{Q} rejects σ . \square

That proof is, basically, time-slicing. In everyday computing we simulate an unboundedly-parallel machine by having its CPU cycle among processes. For example, if on a machine's screen we have a window open to edit a program and another window showing a video, the computer may do the two tasks by updating the editor for a time, then updating the video, then back to the editor, etc. When we use a such a system we perceive that many things are happening at once although actually there is only one, or at least a limited number of simultaneous physical computations.[†] This is a kind of dovetailing, a way of doing a breadth-first traversal of the computation branches.

So adding nondeterminism doesn't expand what Turing machines can compute. Nonetheless, nondeterministic Turing machines are of interest. The reason is that they at least appear to be fast.

Speed The real excitement is that a nondeterministic Turing machine might be much faster than a deterministic one.

- 5.4 EXAMPLE Recall that the **Satisfiability** problem inputs a Propositional Logic expression and determines whether there are truth value that we can substitute for the variables to make the expression T .

Is this propositional logic expression satisfiable?

$$E = (P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (\neg P \vee \neg Q \vee \neg R) \wedge (Q \vee R) \quad (*)$$

The natural approach is to compute a truth table. The table below shows that it is satisfiable, because the TTF row ends in a T .

P	Q	R	$P \vee Q$	$P \vee \neg Q$	$\neg P \vee Q$	$\neg P \vee \neg Q \vee \neg R$	$Q \vee R$	(*)
F	F	F	F	T	T	T	F	F
F	F	T	F	T	T	T	T	F
F	T	F	T	F	T	T	T	F
F	T	T	T	F	T	T	T	F
T	F	F	T	T	F	T	F	F
T	F	T	T	T	F	T	T	F
T	T	F	T	T	T	T	T	T
T	T	T	T	T	T	F	T	F

As to runtime, the number of table rows grows exponentially: it is 2 raised to the number of input variables. Going through the rows one at a time would be very slow.

Each line of the truth table is easy; the issue is that there are lots of lines.

[†]Depending on how many cores are in the CPU.

This situation is perfectly suited for a machine that is unboundedly parallel. For each line we could fork a child process. Each of these children is done quickly, certainly in polytime, and if they are working in parallel then the whole thing is polytime. If at the end any child is holding a T then the expression as a whole is satisfiable. That is, while a serial machine appears to require exponential time, a nondeterministic machine does this job in polytime.

The same potential speedup happens with the Traveling Salesman problem. Checking every potential circuit one at a time to see if it is shorter than a specified bound would be slow. But we can think of a nondeterministic machine either as doing all circuits in parallel, or as just guessing the best circuit (or being given it by some oracular demon) and then checking it, which would be fast.

So while adding nondeterminism to Turing machines doesn't allow them to compute anything new, we might reasonably conjecture that it does allow them to compute those things faster.

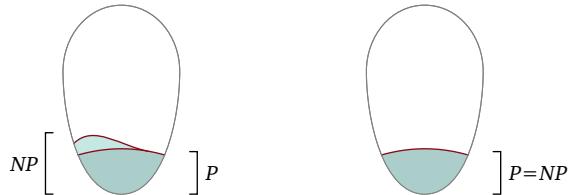
Definition Next we name a class of language decision problems associated with nondeterministic Turing machines.

5.5 **DEFINITION** The complexity class NP is the set of languages for which there is a nondeterministic Turing machine decider \mathcal{P} that runs in polytime, meaning that there is a polynomial p such that on input σ , all branches of \mathcal{P} 's computation halt in time $p(|\sigma|)$.

The following is immediate because a deterministic Turing machine is a special case of a nondeterministic one.

5.6 **LEMMA** $P \subseteq NP$

Very important: no one knows whether P is a strict subset. That is, no one knows whether $P \neq NP$ or $P = NP$. This is the biggest open problem in the Theory of Computing and we will say much more in the rest of this chapter.

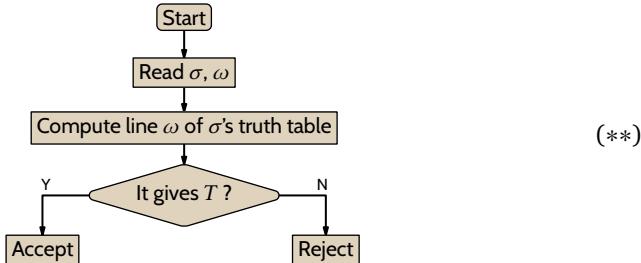


5.7 FIGURE: Which is it: $P \subset NP$ or $P = NP$?

A pattern in mathematical presentations is to have a definition that is conceptually clear, followed by a result that is what we use in practice to determine whether the definition applies. We now give that result. This is where we use the mental model of the machine guessing, or of being told an answer.

Consider Satisfiability. Imagine that in Example 5.4 above the demon whispers, "Psst! Try TTF." With that hint we can quickly verify with a deterministic machine.

For that example's language of satisfiable expressions, here is the verifier.



We start with the expression $\sigma = E$ from that example's (*), and also feed it the demon's hint $\omega = \text{TTF}$. It accepts, certainly in polytime, which verifies that E is satisfiable.

- 5.8 **DEFINITION** A **verifier** for a language \mathcal{L} is a deterministic Turing machine \mathcal{V} that inputs $\langle \sigma, \omega \rangle \in \mathbb{B}^2$ and is such that $\sigma \in \mathcal{L}$ if and only if there exists an ω so that \mathcal{V} accepts $\langle \sigma, \omega \rangle$.[†] The string ω is called the **witness** or **certificate**.
- 5.9 **LEMMA** A language is in NP if and only if it has a verifier that runs in time polynomial in $|\sigma|$. That is, $\mathcal{L} \in NP$ if and only if there is a polynomial p and a deterministic Turing machine \mathcal{V} that halts on all inputs $\langle \sigma, \omega \rangle$ in $p(|\sigma|)$ time, and is such that $\sigma \in \mathcal{L}$ exactly when there is a witness ω where \mathcal{V} accepts $\langle \sigma, \omega \rangle$.

Before the lemma's proof we will first discuss some aspects of both the definition and the lemma.

- 5.10 **EXAMPLE** Our touchstone is the **Satisfiability** problem. Using the lemma to show that this problem is in NP requires that we produce a deterministic Turing machine verifier. In the flowchart below the first input σ is a Boolean expression while the second, the ω , is a string that \mathcal{V} interprets as describing a line of σ 's truth table. If a candidate expression σ is satisfiable then there is a suitable witness, a line from the truth table, so that \mathcal{V} can check that the named line gives a result of T . As an example, for the expression (*) from above, take $\omega = \text{TTF}$. Clearly the verifier can do the checking in polytime. On the other hand, if a candidate σ is not satisfiable, for example with the expression $\sigma = P \wedge \neg P$, then no ω will cause \mathcal{V} to accept.

Before the next example, a few comments.

The most striking thing about the definition is that it says that 'there exists' a witness ω but it does not say where the witness comes from. A person with a computational mindset may well ask, "but how will we calculate the ω 's?" The point is not how to find them. The point is whether there exists a deterministic Turing machine \mathcal{V} that can leverage a given hint ω to verify in polytime that $\sigma \in \mathcal{L}$. That is, we don't find the ω 's, we just use them.

[†]While we have given a definition of a nondeterministic Turing machine accepting its input, we have not given one for deterministic machines. We could modify the machine definition to add accepting states but for simplicity we take it to mean that \mathcal{V} halts and outputs 1 or 'Accept'.

Second, if $\sigma \notin \mathcal{L}$ then the definition does not require a witness to that. Instead, what's required is that from among all possible strings ω there is none such that the verifier accepts $\langle \sigma, \omega \rangle$.[†]

The third comment relates to this asymmetry. Imagine that a demon hands you some papers and claims that they contain an unbeatable strategy for chess. Verifying requires stepping through the responses to each move, and responses to the responses, etc., so there is lots of branching. To prove that the strategy is unbeatable we appear to have to check all of the branches, not just find one good one. It seems that a deterministic verifier must take exponential time. That would make the demon's papers, in a sense, useless. So this chess strategy is not like the problems that we have been considering.

Also reflecting this asymmetry, it is not clear that $\mathcal{L} \in NP$ implies that its complement \mathcal{L}^c is a member of NP . Consider **Satisfiability**. If a propositional logic expression σ is satisfiable then a witness to that is a pointer to a line of the truth table. But for non-satisfiability there is no natural witness; instead, the natural thing is to check all lines. As far as we know today, verifying that a Boolean formula is not satisfiable takes more than polytime. Consequently, where the complexity class $co\text{-}NP$ contains the complements of languages from NP , we suspect that $NP \neq co\text{-}NP$.

Thus, the lemma explains something about the class NP : while P contains problems where we can find the answer in polytime, NP contains the problems whose answers are useful in that we can at least verify them in polytime.

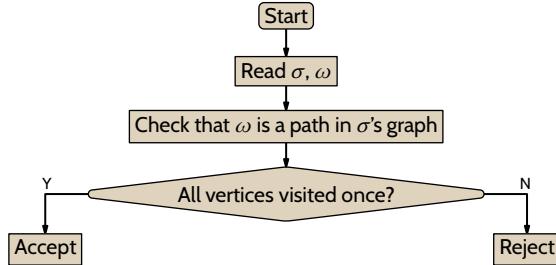
Finally, the lemma requires that the runtime of the verifier is polynomial in $|\sigma|$, not polynomial in the length of its input, $\langle \sigma, \omega \rangle$. If it said the latter then we could check the chess strategy just by using a witness that is exponentially long, which consequently makes $\langle \sigma, \omega \rangle$ exponentially long. Also observe that because \mathcal{V} runs in time polynomial in $|\sigma|$, for the verifier to accept there must exist a witness whose length is at most polynomial in $|\sigma|$, because with ω 's that are too long the verifier cannot even input them before its runtime bound expires.

- 5.11 EXAMPLE The **Hamiltonian Path** problem is like the **Hamiltonian Circuit** problem except that instead of requiring that the starting vertex equals the ending one, it inputs two vertices. It is the problem of determining membership in this set.

$$\mathcal{L} = \{ \langle \mathcal{G}, v, \hat{v} \rangle \mid \text{path in } \mathcal{G} \text{ between } v \text{ and } \hat{v} \text{ visits every vertex exactly once} \}$$

We will show that this problem is in the class NP . We must produce a deterministic Turing machine verifier \mathcal{V} . It is sketched below. It takes as input $\langle \sigma, \omega \rangle$, where the candidate for membership in \mathcal{L} is $\sigma = \langle \mathcal{G}, v, \hat{v} \rangle$. The verifier interprets the witness to be a path, $\omega = \langle v, v_1, \dots, \hat{v} \rangle$.

[†]With this in mind, perhaps a better term for ω is “potential witness” or “proposed witness.” But those are not standard terms.



If there is a Hamiltonian path then there exists a witness ω , and so there is input that \mathcal{V} will accept. Clearly, if given acceptable input then \mathcal{V} runs in polytime. On the other hand, if σ has no Hamiltonian path then for no ω will \mathcal{V} be able to verify that ω is such a path, and thus it will not accept any input pair starting with σ .

- 5.12 EXAMPLE The Primality problem asks whether a given number has a nontrivial factor.

$$\mathcal{L} = \{ n \in \mathbb{N}^+ \mid n \text{ has a divisor } a \text{ with } 1 < a < n \}$$

To show that $\mathcal{L} \in NP$ we construct a verifier. It inputs $\langle \sigma, \omega \rangle$ where σ represents a number $n \in \mathbb{N}^+$. For the witness ω we can use a string that \mathcal{V} interprets as a number a , and then it tests whether $1 < a < n$ and whether a divides n . If $\sigma \in \mathcal{L}$ then there is a suitable such witness, while if $\sigma \notin \mathcal{L}$ then no ω will make \mathcal{V} accept the input. We can write this \mathcal{V} to run in polytime. Therefore $\mathcal{L} \in NP$.

We are ready now for the promised proof of Lemma 5.9.

Proof Suppose first that the language \mathcal{L} is accepted by a nondeterministic Turing machine \mathcal{P} in polytime. We will construct a deterministic verifier \mathcal{V} that runs in polytime. Let $p: \mathbb{N} \rightarrow \mathbb{N}$ be the polynomial such that for any input $\sigma \in \mathcal{L}$, the machine \mathcal{P} has an accepting branch of length at most $p(|\sigma|)$. Use that branch to make a witness to σ 's acceptance, for instance as in the sequence $\omega = \langle 3, 2 \dots \rangle$ meaning, ‘At the first node take the third child, then take the second child of that, etc.’ Restated, \mathcal{P} has a finite number of states k so we can represent the accepting branch of its computation tree with a sequence ω of at most $p(|\sigma|)$ many numbers, each less than k . In total, ω 's length is polynomial in $|\sigma|$. With this ω , a deterministic machine \mathcal{V} can verify \mathcal{P} 's acceptance of σ , in polynomial time.

For the converse suppose that the language \mathcal{L} is accepted by a verifier $\hat{\mathcal{V}}$ that runs in time bounded by a polynomial q . We will construct a nondeterministic Turing machine $\hat{\mathcal{P}}$ that in polytime accepts an input bitstring τ if and only if $\tau \in \mathcal{L}$.

The key is that $\hat{\mathcal{P}}$ is nondeterministic. Given a candidate τ for membership in the language, (1) have $\hat{\mathcal{P}}$ nondeterministically produce a witness $\hat{\omega}$ of length less than $q(|\tau|)$, (2) have $\hat{\mathcal{P}}$ then run $\langle \tau, \hat{\omega} \rangle$ through $\hat{\mathcal{V}}$, and (3) if the verifier accepts its input then $\hat{\mathcal{P}}$ accepts τ , while if \mathcal{V} does not accept then $\hat{\mathcal{P}}$ rejects τ .

We must check that $\hat{\mathcal{P}}$ accepts τ if and only if $\tau \in \mathcal{L}$. A nondeterministic machine accepts a string if there exists a branch that accepts the string, and rejects the string if every branch rejects it. Suppose first that $\tau \in \mathcal{L}$. Because $\hat{\mathcal{V}}$ is a verifier,

in this case there exists a witness $\hat{\omega}$ (of length less than $q(|\tau|)$) that will result in \hat{V} accepting $\langle \tau, \hat{\omega} \rangle$, so there is a way for the prior paragraph to result in acceptance of τ , and so \hat{P} accepts τ . Conversely, suppose that $\tau \notin \mathcal{L}$. By the definition of a verifier, no witness $\hat{\omega}$ will result in \hat{V} accepting $\langle \tau, \hat{\omega} \rangle$, and thus \hat{P} rejects τ . \square

A common reaction to the second half of that proof is, “Wait, \hat{P} pulls $\hat{\omega}$ out of the air? How is that legal?” This reaction — about everyday experience versus abstraction — is both common and reasonable so we will address it.

The first response is purely formal. Definition 5.8, as written, states that the candidate τ is accepted if there exists an $\hat{\omega}$ and does not require us to be able to compute it. The proof’s final paragraph covers the two possibilities: if $\tau \in \mathcal{L}$ then there is such an $\hat{\omega}$ and otherwise there is not, so the definition is satisfied. True, the language “nondeterministically produces a witness” is provocative in that it tends to draw the objection that we are addressing, but this language is common in the literature. (In terms of the two mental models, we can take ‘ \hat{P} nondeterministically produces a witness’ to mean either that it uses unbounded parallelism to produce all possible $\hat{\omega}$ ’s, or that it guesses $\hat{\omega}$ or gets it from a demon.)

The second response is more broad. We today do not have physical devices bearing the same relationship to nondeterministic Turing machines that everyday computers bear to deterministic ones. (We can write a program to simulate nondeterministic behavior but no device does it in hardware.) When Turing formulated his definition there were no physical computers but they appeared soon after; will we someday have nondeterministic devices? Putting aside as too exotic proposals that involve things like time travel through wormholes, we don’t know of any candidates.[†] But that doesn’t mean that thinking about them is a purely academic exercise.[‡] The model of nondeterministic Turing machines has proven to be very fruitful.

As evidence of that, the problems that are associated with these machines, the members of NP , are eminently practical, as witnessed by the fact that computer scientists have been trying to find fast solutions to many members of this class since computers have existed. For another, Lemma 5.9 rephrases questions about nondeterministic machines as questions about deterministic ones, the verifiers.

We close with a reflection. In this section we have defined the class of problems NP for which there is a good way to verify the solution, in contrast with the problems in P for which there is a good way to find the solution. Just as computably enumerable sets seem to be the limit of what can be in theory be known, polytime verification seems to be the limit of what can feasibly be done.

In the next section we will consider whether the two classes P and NP differ.

[†]In order here is a caution about the machine types that seem likely to be coming, quantum computers. Well-established physical theory says that subatomic particles can be in a superposition of many states at once. Naively, it may seem that because of this essentially unbounded multi-way branching, if we could manipulate these particles then we would have nondeterministic computation. But, that we know of, this is false. That we know of, to get information out of a quantum system we must use interference. (Some popularizations wrongly suggest that quantum computers can try all potential solutions in parallel. That is, they miss the point about interference.) [‡]Not that there is anything wrong with academic exercises.

V.5 Exercises

- ✓ 5.13 Your study partner asks, “In Lemma 5.9, since the witness ω is not required to be effectively computable, why can’t I just take it to be the bit 1 if $\sigma \in \mathcal{L}$, and 0 if not? Then writing the verifier is easy: just ignore σ and follow the bit.” They are confused. Straighten them out.
- 5.14 Which is the negation of ‘at least one branch accepts’?
- (A) Every branch accepts.
 - (B) At least one branch rejects.
 - (C) Every branch rejects.
 - (D) At least one branch fails to reject.
 - (E) None of these.
- ✓ 5.15 Decide if it is satisfiable.
- (A) $(P \wedge Q) \vee (\neg Q \wedge R)$
 - (B) $(P \rightarrow Q) \wedge \neg((P \wedge Q) \vee \neg P)$
- 5.16 True or false? If a language is in P then it is in NP .
- 5.17 Uh-oh. You find yourself with a nondeterministic Turing machine where on input σ , one branch of the computation tree accepts and one rejects. Some branches don’t halt at all. What is the upshot?
- ✓ 5.18 You get an exercise, *Write a nondeterministic algorithm that inputs a maze and outputs 1 if there is a path from the start to the end.*
- (A) You hand in an algorithm that does backtracking to find any possible solution. Your professor sends it back, and says to try again. What was wrong?
 - (B) You hand in an algorithm that, each time it comes to a fork in the maze, chooses at random which way to go. Again you get it back with a note to work out another try. What is wrong with this one?
 - (C) Give a right answer.
- 5.19 Sketch a nondeterministic algorithm to search an unordered array of numbers for the number k . Describe it both in terms of unbounded parallelism and in terms of guessing.
- 5.20 A **simple substitution cipher** encrypts text by substituting one letter for another. Start by fixing a permutation of the letters, for example $\langle W, P, \dots \rangle$. Then the cipher is that any A is replaced by a W, any B is replaced by a P, etc. Sketch three algorithms for decoding a substitution cipher (assume that you can recognize a correctly decoded string): (A) one that is deterministic, (B) one expressed in terms of unbounded parallelism, and (C) one expressed in terms of guessing.
- ✓ 5.21 Outline a nondeterministic algorithm that inputs a finite planar graph and outputs Yes if and only if the graph has a four-coloring. Describe it both in terms of unbounded parallelism and in terms of a demon providing a witness.
- 5.22 The **Linear Programming** problem is described on page 280. The related problem **Integer Linear Programming** also seeks to maximize a linear objective

function $F(x_0, \dots, x_n) = d_0x_0 + \dots + d_nx_n$ subject to linear constraints $a_{i,0}x_0 + \dots + a_{i,n}x_n \leq b_i$ (or $\geq b_i$), but with the restriction that all of x_j , d_j , b_j , and $a_{i,j}$ are integers. Recast it as a family of language decision problems. Sketch a nondeterministic algorithm, giving both an unbounded parallelism formulation and a guessing formulation.

- ✓ 5.23 The **Semiprime** problem inputs a number $n \in \mathbb{N}$ and decides if its prime factorization has exactly two primes, $n = p_0^{e_0}p_1^{e_1}$ where $e_i > 0$. State it as a language decision problem. Sketch a nondeterministic algorithm that runs in polytime. Give both an unbounded parallelism formulation and a guessing formulation.

5.24 For each, give a language so that it is a language decision problem. Then give a polytime nondeterministic algorithm. State it in terms of guessing.

- (A) **Three Dimensional Matching:** where X, Y, Z are sets of integers having n elements, given as input a set of triples $M \subseteq X \times Y \times Z$, decide if there is an n -element subset $\hat{M} \subseteq M$ so that no two triples agree on their first coordinates, or second, or third.
- (B) **Partition:** given a finite multiset A of natural numbers, decide if A splits into multisets $\hat{A}, A - \hat{A}$ so the elements total to the same number, $\sum_{a \in \hat{A}} a = \sum_{a \notin \hat{A}} a$.

5.25 Sketch a nondeterministic algorithm that inputs a planar graph and a bound $B \in \mathbb{N}$ and decides whether the graph is B -colorable, described both in terms of unbounded parallelism and also in terms of guessing.

- ✓ 5.26 For each problem, cast it as a language decision problem and then prove that it is in NP by filling in the blanks in this argument.

Lemma 5.9 requires that we produce a deterministic Turing machine verifier, \mathcal{V} . It must input pairs of the form $\langle \sigma, \omega \rangle$, where σ is (1). It must have the property that if $\sigma \in \mathcal{L}$ then there is an ω such that \mathcal{V} accepts the input, while if $\sigma \notin \mathcal{L}$ then there is no such witness ω . And it must run in time polynomial in $|\sigma|$.

The verifier interprets the bitstring witness ω as (2), and checks that (3). Clearly that check can be done in polytime.

If $\sigma \in \mathcal{L}$ then by definition there is (4), and so a witness ω exists that will cause \mathcal{V} to accept the input pair $\langle \sigma, \omega \rangle$. If $\sigma \notin \mathcal{L}$ then there is no such (5), and therefore no witness ω will cause \mathcal{V} to accept the input pair.

- (A) The **Double-SAT** problem inputs a propositional logic statement and decides whether it has at least two different substitutions of Boolean values that make it true.
- (B) The **Subset Sum** problem inputs a set of numbers $S \subset \mathbb{N}$ and a target sum $T \in \mathbb{N}$, and decides whether least one subset of S adds to T .
- ✓ 5.27 In the game show *Countdown*, players get a target integer $T \in [100..999]$ and six numbers from $S = \{1, 2, \dots, 10, 25, 50, 75, 100\}$ (these can be repeated). They make an arithmetic expression that evaluates to the target, using each given number at most once. The expression can use addition, subtraction, multiplication, and division without remainder. Show that the decision problem

for **Countdown** = { $\langle s_0, \dots, s_5, T \rangle \in S^6 \times I \mid$ a combination of the s_i gives T } is in NP .

- ✓ 5.28 Recall that we recast **Traveling Salesman** optimization problem as a language decision problem for a family of languages. Show that each such language is in NP by applying Lemma 5.9, sketching a verifier that works with a suitable witness.
- 5.29 The problem of **Independent Sets** starts with a graph and a natural number n and decides whether in the graph there are n -many independent vertices, that is, vertices that are not connected. State it as a language decision problem, and use Lemma 5.9 to show that this problem is in NP .
- ✓ 5.30 Use Lemma 5.9 to show that the **Knapsack** problem is in NP .
- 5.31 True or false? For the language $\{ \langle a, b, c \rangle \in \mathbb{N}^3 \mid a + b = c \}$, the problem of deciding membership is in NP .
- ✓ 5.32 The **Longest Path** problem is to input a graph and a bound, $\langle G, B \rangle$, and determine whether the graph contains a simple path of length at least $B \in \mathbb{N}$. (A path is simple if no two of its vertices are equal). Show that this is in NP .
- 5.33 Recast each as a language decision problem and then show it is in NP .
 - (A) The **Linear Divisibility** problem inputs a pair of natural numbers $\sigma = \langle a, b \rangle$ and asks if there is an $x \in \mathbb{N}$ with $ax + 1 = b$.
 - (B) Given n points scattered on a line, how far they are from each other defines a multiset. (Recall that a multiset is like a set but element repeats don't collapse.) The reverse of this problem, starting with a multiset M of numbers and deciding whether there exist a set of points on a line whose pairwise distances defines M , is the **Turnpike** problem.
- 5.34 Is NP countable or uncountable?
- ✓ 5.35 Show that this problem is in NP . A company has two delivery trucks. They work with a weighted graph called the 'road map'. (Some vertex is distinguished as the start/finish.) Each morning the company gets a set of vertices, V . They must decide if there are two cycles such that every vertex in V is on at least one of the two cycles, and each cycle has length at most $B \in \mathbb{N}$.
- ✓ 5.36 Two graphs G_0, G_1 are isomorphic if there is a one-to-one and onto function $f: \mathcal{N}_0 \rightarrow \mathcal{N}_1$ such that $\{v, \hat{v}\}$ is an edge of G_0 if and only if $\{f(v), f(\hat{v})\}$ is an edge of G_1 . Consider the problem of computing whether two graphs are isomorphic. (A) Define the appropriate language. (B) Show that the language membership problem is in NP .
- 5.37 The definition of when a nondeterministic machine decides a language, Definition 5.2, requires that every branch in the computation tree is finite. For recognition of languages we can drop that. We say that a nondeterministic Turing machine P **recognizes** a language \mathcal{L} when if $\sigma \in \mathcal{L}$ then there is at least one branch in the computation tree that accepts σ , while if $\sigma \notin \mathcal{L}$ then no branch in the computation tree accepts (some branches may fail to accept because they

are infinite). Show that if there is a nondeterministic machine that recognizes a language then there is a deterministic machine that also recognizes it.

5.38 Following the definition of Turing machine, on page 8, we gave a formal description of how these machines act. We did the same for Finite State machines on page 180, and for nondeterministic Finite State machines on page 188. Give a formal description of the action of a nondeterministic Turing machine.

5.39 (A) Show that the Halting problem is not in NP . (B) What is wrong with this reasoning? The Halting problem is in NP because given $\langle \mathcal{P}, x \rangle$, we can take as the witness ω a number of steps for \mathcal{P} to halt on input x . If it halts in that number of steps then the verifier accepts, and if not then the verifier rejects.

SECTION

V.6 Reductions between problems

When we studied incomputability we considered a sense in which some problems are harder than others. Recall the **Halts on Three** problem to decide membership in the set $S = \{e \in \mathbb{N} \mid \phi_e(3) \downarrow\}$. We showed that if we could solve this then we could solve the Halting problem and we denoted this situation with $K \leq_T S$. In general, a set B Turing reduces to A , written $B \leq_T A$, if there is a Turing machine that computes B from an A oracle, so that $\phi_e^A = \mathbb{1}_B$.[†] Said another way, we can answer questions about membership in B by being given access to a routine that answers questions about membership in A .

There are reductions between problems other than Turing reductions. Consider these two problems, where the second is a translation or alternate presentation of the other: $S = \{e \mid \phi_e(3) \downarrow\}$ and $T = \{e^2 + 1 \mid \phi_e(3) \downarrow\}$. We can compute answers to questions about T from answers about S — we have $x \in T$ iff $\sqrt{x - 1} \in S$. We say that B is **mapping reducible** or **many-one reducible** to A , denoted $B \leq_m A$, if there is a total computable **translation function** f such that $x \in B$ if and only if $f(x) \in A$. Thus for this paragraph's example, $T \leq_m S$.

The difference between the two reducibilities is that with $B \leq_T A$, we answer questions about membership in B by asking a number of questions of an A oracle and then possibly doing more processing with that information. But with \leq_m we make only one oracle call, asking whether $f(x) \in A$, and then the reduction returns that call's result. So, many-one reductions are a special case of Turing reductions, meaning that if $B \leq_m A$ then $B \leq_T A$, but the converse does not hold; see Exercise 6.34.

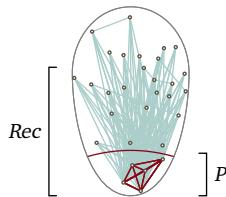
This chapter focuses on efficient use of resources so one natural thing to do is to adapt Turing reduction and define a **Cook reduction** or **polytime Turing reduction**, denoted $B \leq_T^p A$, if there is an oracle Turing machine giving $\phi_e^A = \mathbb{1}_B$ that runs in

[†]Often people get the phrase ‘reduces to’ the wrong way around. Perhaps they are misled by ‘ $B \leq_T A$ ’ into thinking that B is the reduced-to thing but the opposite is true. For example where A is the *Entscheidungsproblem* of answering all questions in Mathematics and B is Goldbach’s conjecture, the right terminology is that B reduces to A because a solution for A gives one for B as a side effect.

polytime. However, the more common reduction between problems is when we adapt many-one reduction by restricting the translation function to run in polytime.

- 6.1 **DEFINITION** Let $\mathcal{L}_0, \mathcal{L}_1$ be languages, subsets of \mathbb{B}^* . Then \mathcal{L}_1 is **polynomial time reducible to \mathcal{L}_0** , or **Karp reducible**, or **polynomial time mapping reducible**, or **polynomial time many-one reducible**, written $\mathcal{L}_1 \leq_p \mathcal{L}_0$ or $\mathcal{L}_1 \leq_m^p \mathcal{L}_0$, if there is a **reduction function** or **transformation function** $f: \mathbb{B}^* \rightarrow \mathbb{B}^*$ that is polynomial time computable and such that $\sigma \in \mathcal{L}_1$ if and only if $f(\sigma) \in \mathcal{L}_0$.[†]

The point of a polytime reduction $\mathcal{L}_1 \leq_p \mathcal{L}_0$ is that a fast way to determine membership in \mathcal{L}_0 gives a fast way to determine membership in \mathcal{L}_1 .



6.2 FIGURE: This is the collection of all problems, $\mathcal{L} \in \mathcal{P}(\mathbb{B}^*)$, with a few shown as dots.

Ones with fast algorithms are at the bottom. Problems are connected if there is a polytime reduction from one to the other. Highlighted are connections within P .

We gave the intuition that there is a reduction of this kind when one problem is a translation of the other, or at least a translation of a special case. The first example illustrates.

- 6.3 **EXAMPLE** The **Shortest Path** problem inputs a weighted graph, two vertices, and a bound, and decides if there is path between the vertices of length less than the bound.

$$\mathcal{L}_0 = \{ \langle \mathcal{G}, v_0, v_1, B \rangle \mid \text{there is path from } v_0 \text{ to } v_1 \text{ of length less than } B \}$$

The **Vertex-to-Vertex Path** problem inputs an unweighted graph and two vertices, and decides if there is a path between the two.

$$\mathcal{L}_1 = \{ \langle \mathcal{H}, w_0, w_1 \rangle \mid \text{there is path between } w_0 \text{ and } w_1 \}$$

We will prove that $\mathcal{L}_1 \leq_p \mathcal{L}_0$. To decide whether some $\langle \mathcal{H}, w_0, w_1 \rangle$ is a member of \mathcal{L}_1 , we will translate that question into one of membership in \mathcal{L}_0 . From \mathcal{H} , make a

[†]If between two problems there is a Karp reduction then there is a Cook reduction also, but the converse does not hold. Under Cook reduction a problem and its complement are equivalent but that's not true under Karp reduction. An important case is that the complexity classes *NP* and *co-NP* (the problems whose complement is in *NP*) are equal under Cook reduction but appear to be separate under Karp reduction. Since any Karp reduction result is automatically a Cook reduction result, Karp reduction gives a more fine-grained partition of the problems and so while it may at first glance seem a little less natural, it is the right one for us to study. In any event, Karp reduction is absolutely the standard in computational complexity.

weighted graph \mathcal{G} by and giving all its edges weight 1. Then $\langle \mathcal{H}, w_0, w_1 \rangle \in \mathcal{L}_1$ if and only if $f(\mathcal{H}) = \langle \mathcal{G}, w_0, w_1, |\mathcal{G}| \rangle \in \mathcal{L}_0$. Clearly f can be done in polytime.

In the next example at first glance the problems don't seem related—for one thing, one is a graph problem and the other is not—so we include some development suggesting how to see such a relationship.

- 6.4 REMARK Authors describing a reduction will often omit this kind of development from the write-up. It is perfectly standard to expect the reader to work out the motivation for the reduction function's definition.
- 6.5 EXAMPLE The **Clique** problem is the decision problem for the language $\mathcal{L}_B = \{ \langle \mathcal{G}, B \rangle \mid \mathcal{G} \text{ has a clique with } B \text{ vertices} \}$. We will sketch that **Satisfiability** \leq_p **Clique**, so that intuitively **Clique** is at least as hard as **Satisfiability**.

Consider how to satisfy this Boolean expression.

$$E = (x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2)$$

The \wedge 's make the statement as a whole T if and only if all of its clauses are T . The \vee 's mean that each clause is T if and only if any of its literals is T . So to satisfy the expression, select a literal from each clause and assign it the value T . For example, we can make E be T by selecting x_0 from the first clause, x_2 from the second, and $\neg x_1$ from the third, and making them T . Similarly, if $\neg x_1$ from the first and third clauses and x_3 from the second are T then E is T . What we cannot do is pick x_2 from the first and second and then $\neg x_2$ from the third, because we cannot set both of these literals to be T .

That is, we can think of **Satisfiability** as a combinatorial problem. The clauses are like buckets and we select one thing from each bucket, subject to the constraint that the things we select must be pairwise compatible.

This view of **Satisfiability** has a binary relation ‘can be compatibly picked’ between the literals. So, as below, let \mathcal{G}_E be a graph whose vertices are pairs $\langle c, \ell \rangle$ where c is the number of a clause and ℓ is a literal in that clause. Two vertices $v_0 = \langle c_0, \ell_0 \rangle$ and $v_1 = \langle c_1, \ell_1 \rangle$ are connected by an edge if they come from different clauses so $c_0 \neq c_1$, and if the literals are not negations of each other so $\ell_0 \neq \neg \ell_1$.

6.6 ANIMATION: Compatibility graph associated with E .

A choice of three mutually compatible vertices makes E evaluate to T . That is, the 3 clause expression E is satisfiable if and only if \mathcal{G}_E has a 3-clique.

More formally, the reduction function f inputs a propositional logic expression E

and outputs a pair $f(E) = \langle \mathcal{G}_E, B \rangle$ where \mathcal{G}_E is the compatibility graph associated with E defined in the prior paragraph and where B is the number of clauses in E . Then $E \in \text{SAT}$ if and only if $f(E) \in \mathcal{L}_B$. Clearly this function can be computed in polytime.

- 6.7 EXAMPLE Recall that a graph is k -colorable if we can partition the vertices into k many classes, called ‘colors’, so that two vertices can have the same color only when there is no edge between them.

6.8 ANIMATION: A 3-coloring of the Petersen graph.

We will illustrate that the Graph Colorability problem reduces to the Satisfiability problem, Graph Colorability \leq_p Satisfiability, by focusing on the $k = 3$ construction. (Larger k 's work much the same way, although the $k = 2$ case is different.)

Denote the set of satisfiable propositional logic statements as \mathcal{L}_0 and the set of 3-colorable graphs as \mathcal{L}_1 . To show that $\mathcal{L}_1 \leq_p \mathcal{L}_0$ we must produce a reduction function f . It inputs a graph \mathcal{G} and outputs a propositional logic expression $E = f(\mathcal{G})$ such that the graph is 3-colorable if and only if the expression is satisfiable.

The function builds E by including clauses that state, in the language of propositional logic, the constraints to be met for the graph to be 3-colorable. Let \mathcal{G} have n -many vertices $\{v_0, \dots, v_{n-1}\}$. Then E has $3n$ -many Boolean variables, a_0, \dots, a_{n-1} , and b_0, \dots, b_{n-1} , and c_0, \dots, c_{n-1} . The idea is that if the i -th vertex v_i gets the first color then E will be satisfied when the associated variables have $a_i = T, b_i = F, c_i = F$, while if v_i gets the second color then E will be satisfied when $a_i = F, b_i = T, c_i = F$, and if v_i gets the third color then E will be satisfied when $a_i = F, b_i = F, c_i = T$.

Specifically, the expression includes two kinds of clauses. For every vertex v_i , there is a clause saying that the vertex gets at least one color.

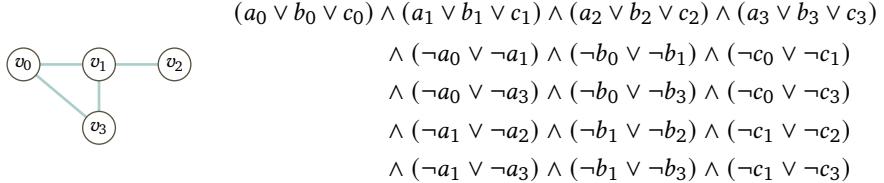
$$(a_i \vee b_i \vee c_i)$$

And for each edge $\{v_i, v_j\}$, there are three clauses which together ensure that the edge does not connect two same-color vertices.

$$(\neg a_i \vee \neg a_j) \quad (\neg b_i \vee \neg b_j) \quad (\neg c_i \vee \neg c_j)$$

The function's output E is the conjunction of all of these clauses.

This illustrates. The expression's top line has the clauses of the first kind while the remaining lines have the other kind.



By construction, there is an assignments of truth values for the variables to satisfy the expression if and only if the graph has a 3-coloring.

Completing the argument requires checking that the reduction function, which inputs a bitstring representation of the graph and outputs a bitstring representation of the expression, is polynomial. That's clear so we omit the details.

A reduction function is a kind of compiler. A programming language compiler inputs descriptions from one domain, such as a Racket program, and outputs a corresponding statement from another domain, such an executable in the machine's native format. Similarly, the function f above translates problem instances in the domain of graphs to those in the domain of propositional logic.

- 6.9 EXAMPLE We will show that **Subset Sum** reduces to **Knapsack**, that **Subset Sum** \leq_p **Knapsack**. The **Knapsack** problem starts with a multiset of objects $U = \{u_0, \dots, u_{n-1}\}$ whose elements each have a weight $w(u_i)$ and a value $v(u_i)$, along with an upper bound on the weights $W \in \mathbb{N}$ and a lower bound for the values $V \in \mathbb{N}$. It asks for a subset $A \subseteq U$ such that the weight total does not exceed W while the value total is at least as big as V .

$$\mathcal{L}_0 = \{\langle U, w, v, W, V \rangle \mid \text{some } A \subseteq U \text{ has } \sum_{a \in A} w(a) \leq W \text{ and } \sum_{a \in A} v(a) \geq V\}$$

The **Subset Sum** problem starts with a multiset $S = \{s_0, \dots, s_{k-1}\} \subset \mathbb{N}$ and a target $T \in \mathbb{N}$. It asks for a subset whose elements add to the target.

$$\mathcal{L}_1 = \{\langle S, T \rangle \mid \text{some } R \subseteq S \text{ has } \sum_{r \in R} r = T\}$$

The reduction function f must input pairs $\langle S, T \rangle$ and output five-tuples $\langle U, w, v, W, V \rangle$, and must be such that $\langle S, T \rangle \in \mathcal{L}_1$ if and only if $f(\langle S, T \rangle) \in \mathcal{L}_0$. And it must be polytime.

A numerical example gives the idea of how f proceeds. Imagine that we want to know if there is a subset of $S = \{18, 23, 31, 33, 72, 86, 94\}$ that adds to $T = 126$. If we had access to an oracle for **Knapsack** then we could set $U = S$, let w and v be the identity functions so that $w(18) = v(18) = 18$ and $w(23) = v(23) = 23$, etc., and then fix the weight and value targets as $W = V = T = 126$. Then $\langle S, T \rangle \in \mathcal{L}_1$ iff $\langle S, w, v, W, V \rangle \in \mathcal{L}_0$. In this way, we think of the **Subset Sum** problem as a special case of **Knapsack**.

More generally, let f take the input $\langle S, T \rangle$ to the output $\langle S, w, v, T, T \rangle$, where the functions w and v are given by $w(s_i) = v(s_i) = s_i$. Then $\langle S, T \rangle \in \mathcal{L}_0$ if and only if $f(\langle S, T \rangle) \in \mathcal{L}_1$. Clearly f can be done in polytime.

We close with some basic facts about polytime reduction.

- 6.10 LEMMA Polytime reduction is reflexive: $\mathcal{L} \leq_p \mathcal{L}$ for all languages. It is also transitive: $\mathcal{L}_2 \leq_p \mathcal{L}_1$ and $\mathcal{L}_1 \leq_p \mathcal{L}_0$ imply that $\mathcal{L}_2 \leq_p \mathcal{L}_0$. Every nontrivial computable language is **P hard**: for $\mathcal{L}_1 \in P$, every language \mathcal{L}_0 with $\mathcal{L}_0 \neq \emptyset$ and $\mathcal{L}_0 \neq \mathbb{N}$ satisfies that $\mathcal{L}_1 \leq_p \mathcal{L}_0$. The class P is closed downward: if $\mathcal{L}_0 \in P$ and $\mathcal{L}_1 \leq_p \mathcal{L}_0$ then $\mathcal{L}_1 \in P$. So also is the class NP .

Proof The first two sentences and the final sentence are Exercise 6.35.

For the third sentence fix a \mathcal{L}_0 that is nontrivial, so there is a member $\sigma \in \mathcal{L}_0$ and a nonmember $\tau \notin \mathcal{L}_0$. Let \mathcal{L}_1 be an element of P . We will specify a polytime reduction function f for $\mathcal{L}_1 \leq_p \mathcal{L}_0$. For $\alpha \in \mathbb{B}^*$, computing whether $\alpha \in \mathcal{L}_1$ can be done in polytime. If it is a member then let $f(\alpha) = \sigma$ while if not then let $f(\alpha) = \tau$.

For the downward closure of P , suppose that $\mathcal{L}_1 \leq_p \mathcal{L}_0$ via the polytime function g and also suppose that there is a polytime algorithm for determining membership in \mathcal{L}_0 . Determine membership in \mathcal{L}_1 by starting with an input σ , finding $g(\sigma)$, and applying the \mathcal{L}_0 algorithm to settle whether $g(\sigma) \in \mathcal{L}_0$. Where the \mathcal{L}_0 algorithm runs in time that is $\mathcal{O}(n^i)$ and where g runs in time that is $\mathcal{O}(n^j)$, determining \mathcal{L}_1 membership in this way runs in time that is $\mathcal{O}(n^{\max(i,j)})$, which is polynomial. \square

By the lemma, because Example 6.7 shows Graph Colorability \leq_p Satisfiability and Example 6.5 gives Satisfiability \leq_p Clique, we know that Graph Colorability \leq_p Clique.

Reiterating this section's main point, a reducibility such as $\mathcal{L}_1 \leq_p \mathcal{L}_0$ means that if we could solve the \mathcal{L}_0 problem in polynomial time then we could solve the \mathcal{L}_1 problem in polynomial time also.

Finally, more examples of problem reductions in Section 7.

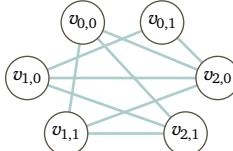
V.6 Exercises

- 6.11 If $\mathcal{L}_1 \leq_p \mathcal{L}_0$ then which is true?
- A fast algorithm for \mathcal{L}_0 would give a fast algorithm for \mathcal{L}_1 .
 - A fast algorithm for \mathcal{L}_1 would give a fast one for \mathcal{L}_0 .
- 6.12 Suppose that $\mathcal{L}_1 \leq_p \mathcal{L}_0$. Which is the right way to use the phrase ‘reduces to’: \mathcal{L}_1 reduces to \mathcal{L}_0 , or \mathcal{L}_0 reduces to \mathcal{L}_1 ?
- ✓ 6.13 Show that if $\mathcal{L}_0 \notin P$ and $\mathcal{L}_0 \leq_p \mathcal{L}_1$ then $\mathcal{L}_1 \notin P$ also. What about NP ?
- 6.14 Prove that $\mathcal{L} \leq_p \mathcal{L}^c$ if and only if $\mathcal{L}^c \leq_p \mathcal{L}$.
- 6.15 Your friend is confused. “Lemma 6.10 says that every language in P is \leq_p every other nontrivial language. But there are uncountably many languages and only countably many f 's because they each come from some Turing machine. So I'm not seeing how there are enough reduction functions for a given language to be related to all others.” Un-confuse them.

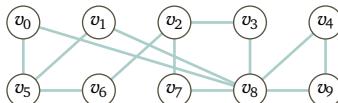
6.16 Example 6.9 includes as illustration a **Subset Sum** problem, where $S = \{18, 23, 31, 33, 72, 86, 94\}$ and $T = 126$. Solve it.

6.17 Produce the compatibility graph for $(x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_1) \wedge (x_0 \vee \neg x_1)$.

6.18 Following the method of Example 6.7 give the expression associated with the question of whether this graph is 3-colorable. Is that expression satisfiable?



- ✓ 6.19 Suppose that the language A is polynomial time reducible to the language B , $A \leq_p B$. Which of these are true?
 - (A) A tractable way to decide A can be used to tractably decide B .
 - (B) If A is tractably decidable then B is tractably decidable also.
 - (C) If A is not tractably decidable then B is not tractably decidable too.
- ✓ 6.20 The **Substring** problem inputs two strings and decides if the second is a substring of the first. The **Cyclic Shift** problem inputs two strings and decides if the second is a cyclic shift of the first. (Where $\alpha = abcde$, one cyclic shift is $\beta = deabc$. More precisely, if $\alpha = a_0a_1 \dots a_{n-1}$ and $\beta = b_0b_1 \dots b_{n-1}$ are length n strings, then β is a cyclic shift of α when there is an index $k \in \{0, \dots, n-1\}$ such that $a_i = b_{(k+i) \bmod n}$ for all $i < n$)
 - (A) Name three cyclic shifts of $\alpha = 0110010$.
 - (B) Decide whether $\beta = 101001101$ is a cyclic shift of $\alpha = 001101101$.
 - (C) State the **Substring** problem as a language decision problem.
 - (D) Also state the **Cyclic Shift** problem as a language decision problem.
 - (E) Show that **Cyclic Shift** \leq_p **Substring**. Hint: for same length strings, β is a cyclic shift of α if and only if β is a substring of $\alpha^\sim \alpha$.
- ✓ 6.21 The **Independent Set** problem inputs a graph and a bound, and decides if there is a set of vertices, of size at least equal to the bound, that are not connected by any edge. The **Vertex Cover** problem inputs a graph and a bound and decides if there is a vertex set, of size less than or equal to the bound, such that every edge contains at least one vertex in the set.
 - (A) State each as a language decision problem.
 - (B) Consider this graph. Find a vertex cover with four elements.



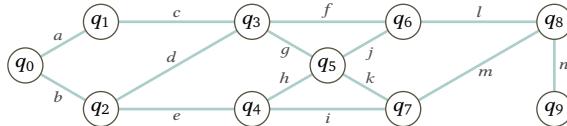
- (C) In that graph find an independent set with six elements.
- (D) Show that in a graph, S is an independent set if and only if $\mathcal{N} - S$ is a vertex cover, where \mathcal{N} is the set of vertices.
- (E) Conclude that **Vertex Cover** \leq_p **Independent Set**.

(F) Also conclude that **Independent Set** \leq_p **Vertex Cover**.

6.22 The **Vertex Cover** problem inputs a graph and a bound and decides if there is a vertex set, of size less than or equal to the bound, such that every edge contains at least one vertex in the set. The **Set Cover** problem inputs a set S , a collection of subsets $S_0 \subseteq S, \dots, S_n \subseteq S$, and a bound, and decides if there is a subcollection of the S_j , with a number of sets at most equal to the bound, whose union is S .

(A) State each as a language decision problem.

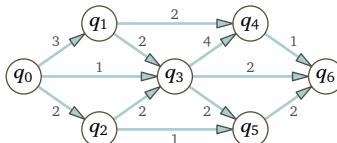
(B) Find a vertex cover for this graph.



(C) Make a set S consisting of all of that graph's edges, and for each v make a subset S_v of the edges incident on that vertex. Find a set cover.

(D) Show that **Vertex Cover** \leq_p **Set Cover**.

- ✓ 6.23 Show that **Hamiltonian Circuit** \leq_p **Traveling Salesman**. (A) State each as a language decision problem. (B) Produce the reduction function.
- ✓ 6.24 In this network, each edge is labeled with a capacity. (Imagine railroad lines going from q_0 to q_6 .)



The **Max-Flow** problem is to find the maximum total amount that can flow across the network, usually by using many paths at once. That is, we will find a **flow** F_{q_i, q_j} for each edge, subject to the constraints that the flow through an edge must not exceed its capacity and that the flow into a vertex must equal the flow out (except for the source q_0 and the sink q_6). The **Linear Programming** problem is described on page 280.

- (A) Express each as a language decision problem, remembering the technique of converting optimization problems using bounds.
- (B) By eye, find the maximum flow for the above network.
- (C) For each edge $v_i v_j$, define a variable $x_{i,j}$. Describe the constraints on that variable imposed by the edge's capacity. Also describe the constraints on the set of variables imposed by the limitation that for many vertices the flow in must equal the flow out. Finally, use the variables to give an expression to optimize in order to get maximum flow.
- (D) Show that **Max-Flow** \leq_p **Linear Programming**.

6.25 The **Max-Flow** problem inputs a directed graph where each edge is labeled with a capacity, and the task is to find the maximum amount that can flow from the source node to the sink node (for more, see Exercise 6.24). The

Drummer problem starts with two same-sized sets, the rock bands, B , and potential drummers, D . Each band $b \in B$ has a set $S_b \subseteq D$ of drummers that they would agree to take on. The goal is to make the most number of matches.

- (A) Consider four bands $B = \{b_0, b_1, b_2, b_3\}$ and drummers $D = \{d_0, d_1, d_2, d_3\}$.
Band b_0 likes drummers d_0 and d_2 . Band b_1 likes only drummer d_1 , and b_2 also likes only d_1 . Band b_3 like the sound of both d_2 and d_3 . What is the largest number of matches?
- (B) Express each as a language decision problem.
- (C) Draw a graph with the bands on the left and the drummers on the right.
Make an arrow from a band to a drummer if there is a connection. Now add a source and a sink node to make a flow diagram.
- (D) Show that Drummer \leq_p Max-Flow.

6.26 The **3-SAT** problem is to decide the satisfiability of CNF propositional logic expressions where every clause has at most three literals. The **Strict 3-Satisfiability problem** requires that each clause has exactly three unequal literals. We will show that the two are inter-reducible.

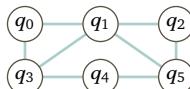
- (A) Show the easy half, that Strict 3-Satisfiability \leq_p 3-SAT.
- (B) Also show that we can go from clauses with two literals to clauses with three by introducing an irrelevant variable: $P \vee Q$ is equivalent to $(P \vee Q \vee R) \wedge (P \vee Q \vee \neg R)$. Along the same lines, show that P is equivalent to $(P \vee Q \vee R) \wedge (P \vee \neg Q \vee R) \wedge (P \vee Q \vee \neg R) \wedge (P \vee \neg Q \vee \neg R)$.
- (C) Show 3-SAT \leq_p Strict 3-Satisfiability.

6.27 We will show that the **3-SAT** problem, 3-SAT, is inter-reducible with SAT.
(We will assume that instances of SAT are in Conjunctive Normal form.)

- (A) Show the easy half, that 3-SAT \leq_p SAT.
- (B) As a preliminary for the other reduction, show that the propositional logic implication $P \rightarrow Q$ is equivalent to $\neg P \vee Q$.
- (C) To go from clauses with four literals to those with three, start with $P \vee Q \vee R \vee S$.
Introduce a variable A such that $A \leftrightarrow (P \vee Q)$, that is, $(A \rightarrow (P \vee Q)) \wedge (A \leftarrow (P \vee Q))$. Show that $(A \rightarrow (P \vee Q))$ is equivalent to $(P \vee Q \vee \neg A)$. Also verify that $(P \vee Q) \rightarrow A$ is equivalent to $(P \vee \neg Q \vee A) \wedge (\neg P \vee Q \vee A) \wedge (\neg P \vee \neg Q \vee A)$. Conclude that $P \vee Q \vee R \vee S$ is equivalent to $(A \vee R \vee S) \wedge (P \vee Q \vee \neg A) \wedge (P \vee \neg Q \vee A) \wedge (\neg P \vee Q \vee A) \wedge (\neg P \vee \neg Q \vee A)$.
- (D) For a five literal clause $P \vee Q \vee R \vee S \vee X$, find an equivalent propositional logic expression made of clauses having only three literals each.
- (E) Show that SAT \leq_p 3-SAT.

6.28 The **Independent Set** problem inputs a graph and a bound, and decides if there is a set of vertices, of size at least equal to the bound, that are not connected by any edge.

- (A) In this graph, find an independent set with at least $B = 3$ members.



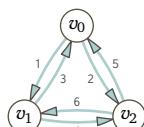
- (B) State Independent Set as a language decision problem.
- (C) Decide if $E = (P_0 \vee \neg P_1 \vee \neg P_2) \wedge (P_1 \vee P_2 \vee \neg P_3)$ is satisfiable.
- (D) State 3-SAT as a language decision problem.
- (E) With the expression E , make a triangle for each of the two clauses, where the vertices of the first are labeled v_0 , \bar{v}_1 , and \bar{v}_2 , while the vertices of the second are w_1 , w_2 , and \bar{w}_3 . In addition to the edges forming the triangles, also put one connecting \bar{v}_1 with w_1 , and one connecting \bar{v}_2 with w_2 .
- (F) Sketch an argument that $\text{3-SAT} \leq_p \text{Independent Set}$.
- ✓ 6.29 The decision problem Integer Linear Programming starts with a list of linear inequalities with variables x_0, \dots, x_n , such as $a_{i,0}x_0 + \dots + a_{i,n}x_n \leq b_i$ (or $\geq b$), and it looks for a sequence $\langle s_0, \dots, s_n \rangle$ that is feasible in that it satisfies all of the constraints, but with the restriction that the numbers must be integers, $a_{i,j}, b_i, s_i \in \mathbb{Z}$.
- (A) Consider the propositional logic clause $P_0 \vee \neg P_1 \vee \neg P_2$. Create variables z_0 , z_1 , and z_2 and list linear constraints such that each must be either 0 or 1. Also give a linear inequality that holds if and only if the clause is true.
- (B) Show that $\text{3-SAT} \leq_p \text{Integer Linear Programming}$.
- 6.30 Consider the problem D of deciding whether a multivariable polynomial has any integer roots. That is, it is the language decision problem for this set.

$$D = \{p \mid p \text{ is a polynomial and there is } \vec{c} \in \mathbb{Z}^n \text{ so that } p(\vec{c}) = 0\}$$

We will show that $\text{3-SAT} \leq_p \text{D}$.

- (A) Argue that a one-clause disjunction has a value of T if and only if any of its literals has a value of T. For instance, $E_0 = P_0 \vee \neg P_1$ is true if and only if $P_0 = T$ or $\neg P_1 = T$.
- (B) Associate E_0 with the set $S_{E_0} = \{x_0(1 - x_0), x_1(1 - x_1), x_0(1 - x_1)\}$ of three polynomials. Argue that all three have a value of 0 if and only if both variables have a value of either 0 or 1, and either $x_0 = 0$ or $x_1 = 1$.
- (C) For the expression $E_1 = (P_0 \vee \neg P_1 \vee \neg P_2) \wedge (P_1 \vee P_2 \vee \neg P_3)$, produce a set of polynomials S_{E_1} with the analogous properties.
- (D) Combine the polynomials from S_{E_1} in the prior item into a single polynomial in such a way that it has an overall value of 0 if and only if all the members of S have a value of 0.
- (E) Show that $\text{3-SAT} \leq_p \text{D}$.

- 6.31 Our definition of the Traveling Salesman problem is based on an undirected graph. The Asymmetric Traveling Salesman problem takes place on a directed graph. (The problem name comes from the fact that the graph's matrix is asymmetric.)



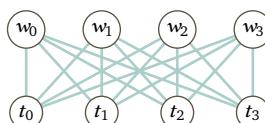
$$M = \begin{matrix} & v_0 & v_1 & v_2 \\ v_0 & - & 1 & 2 \\ v_1 & 3 & - & 4 \\ v_2 & 5 & 6 & - \end{matrix}$$

An example of such a problem is when nodes are cities and edges are flights, with the weight of an edge being the flight's cost.

- (A) Show that **Traveling Salesman** \leq_p **Asymmetric Traveling Salesman**.
 - (B) We will develop the converse. An asymmetric problem instance is a directed graph \mathcal{G} with vertices g_0, \dots, g_n . From it make a new graph \mathcal{H} with twice as many vertices, which come in two colors, $h_0, \dots, h_n, \hat{h}_0, \dots, \hat{h}_n$. This graph only has connections between different-colored vertices. Where $i \neq j$, if vertex g_i is connected to g_j by an edge with weight w then vertex \hat{h}_i is connected to h_j and vertex h_i is connected to \hat{h}_j , both by an edge with weight w . Using the graph \mathcal{G} pictured above, produce the graph matrix for \mathcal{H} and verify that it is symmetric.
 - (C) To finish, let W be the product of \mathcal{G} 's maximum edge weight with its number of vertices. In the graph H , for all indices i use an edge of weight $-W - 1$ to connect h_i with \hat{h}_i , in both directions. Modify the matrix from the prior part and verify that the modified matrix is symmetric.
 - (D) Find the connection between solution circuits in \mathcal{H} and those in \mathcal{G} . Argue that with a **Traveling Salesman** oracle we could, given an asymmetric instance \mathcal{G} , translate that to a symmetric instance \mathcal{H} and use the oracle to derive a solution to \mathcal{G} , all in polytime.
- ✓ 6.32 We can do reductions between problems of types other than language decision problems, such as optimizations. A **reduction between optimization problems** is a pair of polytime computable functions $\langle f, g \rangle$ mapping bitstrings to bitstrings, where f is a reduction function as in Definition 6.1, taking instances to instances, and g takes optimal solutions to optimal solutions.
- (A) The **Assignment problem** inputs a set of workers $W = \{w_0, \dots, w_{n-1}\}$ and a set of tasks $T = \{t_0, \dots, t_{n-1}\}$, of equal sizes. For each worker and task there is a cost $C(w_i, t_j)$. The goal is to assign each task, one per worker, for minimal total cost. By eye, solve this **Assignment** problem instance.

$\text{Cost } C(w_i, t_j)$	w_0	w_1	w_2	w_3
t_0	13	4	7	6
t_1	1	11	5	4
t_2	6	7	2	8
t_3	1	3	5	9

- (B) The **Asymmetric Traveling Salesman** problem asks for circuit of minimal total cost in a weighted graph that is directed. Show that **Assignment** \leq_p **Asymmetric Traveling Salesman**. Hint: the reduction function f inputs an assignment table. Have it output a bipartite graph as shown here, where each worker is connected to each task,

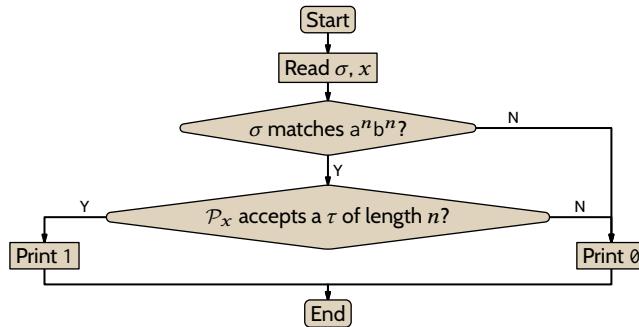


but make it a directed graph where between each worker and task there

is an edge in each direction. Use the given assignment cost table to make appropriate edge weights. Finish by verifying that there is a polytime computable function g that associates optimal assignments with optimal circuits.

6.33 We will show that $\text{Fin} \leq_p \text{Reg}$, where they are the decision problems for the language $R = \{x \in \mathbb{N} \mid \text{the language decided by } \mathcal{P}_x \text{ is regular}\}$ and also for the language $F = \{i \in \mathbb{N} \mid \text{the language decided by } \mathcal{P}_i \text{ is finite}\}$ (this means that \mathcal{P}_i halts on all inputs and acts as the characteristic function of a set that is finite).

- (A) Adapt Example 5.2 from Chapter Four to show that any infinite subset of $\{a^n b^n \mid n \in \mathbb{N}\}$ is not regular.
- (B) Argue that there is a Turing machine with the behavior below. Then apply the $s\text{-}m\text{-}n$ lemma to parameterize x .



- (C) Using the prior item, produce the reduction function.

6.34 As discussed at the start of the section, $B \leq_T A$ if there is an oracle A Turing machine that computes B , so that $\phi_e^A = \mathbb{1}_B$. And, $B \leq_m A$ if there is a total computable function so that $x \in B$ if and only if $f(x) \in A$. We will show that the two reductions \leq_T and \leq_m differ. (A) Let B be the nonempty computable set $\{2n \mid n \in \mathbb{N}\}$ of even numbers and let A be the empty set. Show that $B \leq_T A$ but it is not the case that $B \leq_m A$. (B) Observe that $A \leq_T A^c$ for all sets, and so $K \leq_T K^c$. Show that if $B \leq_m A$ and B is computably enumerable then so is A , and conclude that K is not many-one reducible to its complement.

- ✓ 6.35 Lemma 6.10 leaves a couple of points undone. (A) Show that \leq_p is reflexive and transitive. (B) It says that nontrivial languages are P hard. What about trivial ones? Which languages reduce to the empty set? To \mathbb{B}^* ? (C) Show that NP is downward closed, that if $\mathcal{L}_1 \leq_p \mathcal{L}_0$ and $\mathcal{L}_0 \in NP$ then $\mathcal{L}_1 \in NP$ also.

6.36 When $\mathcal{L}_i \leq_p \mathcal{L}_j$, does that mean that the best algorithm to decide \mathcal{L}_i takes time that is less than or equal to the amount taken by the best algorithm for \mathcal{L}_j ? Fix a language decision problem \mathcal{L}_0 whose fastest algorithm is $\mathcal{O}(n^3)$, an \mathcal{L}_1 whose best algorithm is $\mathcal{O}(n^2)$, a \mathcal{L}_2 whose best is $\mathcal{O}(2^n)$, and a \mathcal{L}_3 whose best is $\mathcal{O}(\lg n)$. In the array entry i, j below, put 'N' if $\mathcal{L}_i \leq_p \mathcal{L}_j$ is not possible.

	\mathcal{L}_0	\mathcal{L}_1	\mathcal{L}_2	\mathcal{L}_3
\mathcal{L}_0	(0,0)	(0,1)	(0,2)	(0,3)
\mathcal{L}_1	(1,0)	(1,1)	(1,2)	(1,3)
\mathcal{L}_2	(2,0)	(2,1)	(2,2)	(2,3)
\mathcal{L}_3	(3,0)	(3,1)	(3,2)	(3,3)

6.37 Is there a connection between subset and polytime reducibility? Find languages $\mathcal{L}_0, \mathcal{L}_1 \in \mathcal{P}(\mathbb{B}^*)$ for each: (A) $\mathcal{L}_0 \subset \mathcal{L}_1$ and $\mathcal{L}_0 \leq_p \mathcal{L}_1$, (B) $\mathcal{L}_0 \not\subset \mathcal{L}_1$ and $\mathcal{L}_0 \leq_p \mathcal{L}_1$, (C) $\mathcal{L}_0 \subset \mathcal{L}_1$ and $\mathcal{L}_0 \not\leq_p \mathcal{L}_1$, (D) $\mathcal{L}_0 \not\subset \mathcal{L}_1$ and $\mathcal{L}_0 \not\leq_p \mathcal{L}_1$.

SECTION

V.7 NP completeness

Because $P \subseteq NP$, the class NP contains lots of easy problems, ones with a fast algorithm. Nonetheless, the interest in the class is that it also contains lots of problems that seem to be hard. Can we prove that these problems are indeed hard?

This question was raised by S Cook in 1971. He noted that the idea of polynomial time reducibility gives us a way to make precise that an efficient solution for one problem implies an efficient solution for the other. He then showed that among the problems in NP , there are ones that are maximally hard. (This was also shown by L Levin but he was behind the Iron Curtain and knowledge of his work did not spread to the rest of the world for some time.)



Stephen Cook
b 1939

Here, ‘maximally hard’ means that these are NP problems and they are at least as hard as any NP problem, in that if we could solve one of these then we could solve any NP problem at all.

7.1 **THEOREM (COOK-LEVIN THEOREM)** The Satisfiability problem is in NP and has the property any problem in NP reduces to it: $\mathcal{L} \leq_p SAT$ for any $\mathcal{L} \in NP$.

First, we have already observed that $SAT \in NP$ because, given a Boolean expression, we can use as a witness ω an assignment of truth values that satisfies the expression.

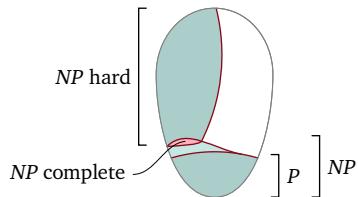


Leonid Levin
b 1948

Here is an outline of the proof’s other half. Given $\mathcal{L} \in NP$, we must show that $\mathcal{L} \leq_p SAT$. We produce a function $f_{\mathcal{L}}$ that translates membership questions for \mathcal{L} into Boolean expressions, such that the membership answer is ‘yes’ if and only if the expression is satisfiable. What we know about \mathcal{L} is that its member σ ’s are accepted by a nondeterministic machine \mathcal{P} in time given by a polynomial q . With that, from $\langle \mathcal{P}, \sigma, q \rangle$ the proof constructs a Boolean expression that yields T if and only if \mathcal{P} accepts σ . The Boolean expression encodes the constraints under which a Turing machine operates, such as that the only tape symbol that can be changed in the current step is the symbol under the machine’s head.

- 7.2 **DEFINITION** A problem is **NP hard** if every problem in **NP** reduces to it, that is, \mathcal{L} is **NP hard** if $\hat{\mathcal{L}} \in NP$ implies that $\hat{\mathcal{L}} \leq_p \mathcal{L}$. A problem is **NP complete** if, in addition to being **NP hard**, it is also a member of **NP**.[†]

So a problem is **NP complete** if it is, in a sense, at least as hard as any member of **NP**. The sketch below illustrates.



7.3 **FIGURE:** The blob contains all problems. In the bottom is **NP**, drawn with **P** as a proper subset. The top has the **NP-hard** problems. The highlighted intersection is the set of **NP complete** problems.

The Cook-Levin Theorem says that there is at least one **NP complete** problem, namely **SAT**. In fact, we shall see that there are many such problems.

The **NP complete** problems are to the class **NP** as the problems Turing-equivalent to the Halting problem set **K** are to the computably enumerable sets. If we could solve the one problem then we could solve every other problem in that class.

- 7.4 **LEMMA** If \mathcal{L}_0 is **NP complete**, and $\mathcal{L}_0 \leq_p \mathcal{L}_1$, and $\mathcal{L}_1 \in NP$ then \mathcal{L}_1 is **NP complete**.

Proof Exercise 7.29. □

Soon after Cook raised the question of **NP completeness**, R Karp brought it to widespread attention. Karp noted that there are clusters of problems: there is a collection of problems solvable in time $\mathcal{O}(\lg(n))$, problems of time $\mathcal{O}(n)$, those of time $\mathcal{O}(n \lg n)$, etc. There is also a cluster of problems that seem much tougher. He gave a list of twenty one of these, drawn from Computer Science, Mathematics, and the natural sciences, where lots of smart people had for years been unable find efficient algorithms. He showed that all of these problems are **NP complete**, so that if we could efficiently solve any then we could efficiently solve them all. Not every difficult problem is **NP complete** but many thousands of problems have been shown to be so and thus whatever it is that makes these problems hard, all of them share it.



Richard M Karp
b 1935

Typically we prove that a problem \mathcal{L} is **NP complete** in two halves. First we show that it is in **NP** by exhibiting a witness ω that a deterministic verifier can check in polytime. Second, we show that the problem is **NP hard** by showing that an **NP complete** problem reduces to it. The list below gives the **NP complete** problems most often used. For instance, we might show that $3\text{-SAT} \leq_p \mathcal{L}$.

[†]In general, for a complexity class C , a problem \mathcal{L} is **C hard** when all problems in that class reduce to it: if $\hat{\mathcal{L}} \in C$ then $\hat{\mathcal{L}} \leq_p \mathcal{L}$. A problem is **C complete** if it is hard for that class and also is a member of that class.

- 7.5 **THEOREM (BASIC NP COMPLETE PROBLEMS)** Each of these problems is *NP* complete.

3-Satisfiability, 3-SAT Given a propositional logic formula in conjunctive normal form in which each clause has at most 3 variables, decide if it is satisfiable.

3 Dimensional Matching Given as input a set $M \subseteq X \times Y \times Z$, where the sets X, Y, Z all have the same number of elements, n , decide if there is a matching, a set $\hat{M} \subseteq M$ containing n elements such that no two of the triples in \hat{M} agree on any of their coordinates.

Vertex cover Given a graph and a bound $B \in \mathbb{N}$, decide if the graph has a size B set of vertices C such that for any edge $v_i v_j$, at least one of its ends is a member of C .

Clique Given a graph and a bound $B \in \mathbb{N}$, decide if the graph has a set of B -many vertices where any two are connected.

Hamiltonian Circuit Given a graph, decide if it contains a cyclic path that includes each vertex.

Partition Given a finite multiset S of natural numbers, decide if there is a division of the set into the two parts \hat{S} and $S - \hat{S}$ so the total of their elements is the same, $\sum_{s \in \hat{S}} s = \sum_{s \notin \hat{S}} s$.

- 7.6 **EXAMPLE** We will show that the **Traveling Salesman** problem is *NP* complete. Recall that we have recast it as the decision problem for the language of pairs $\langle \mathcal{G}, B \rangle$, where B is a parameter bound, and that this problem is a member of *NP*. We will show that it is *NP* hard by proving that the **Hamiltonian Circuit** problem reduces to it, $\text{Hamiltonian Circuit} \leq_p \text{Traveling Salesman}$.

We need a reduction function f . It must input an instance of **Hamiltonian Circuit**, a graph $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ whose edges are unweighted. Define f to return the instance of **Traveling Salesman** that uses \mathcal{N} as cities, that takes the distances between cities to be $d(v_i, v_j) = 1$ if $v_i v_j \in \mathcal{E}$ and $d(v_i, v_j) = 2$ if $v_i v_j \notin \mathcal{E}$, and such that the bound is the number of vertices, $B = |\mathcal{N}|$.

This bound means that there will be a **Traveling Salesman** solution if and only if there is a **Hamiltonian Circuit** solution; namely, the salesman uses the edges that appear in the Hamiltonian circuit. All that remains is to argue that the reduction function runs in polytime. The number of edges in a graph is no more than twice the number of vertices so polytime in the input graph size is the same as polytime in the number of vertices. The reduction function's algorithm examines all pairs of vertices, which takes time that is quadratic in the number of vertices.

A common way to show that a given problem \mathcal{L} is *NP* hard is to show that a special case of \mathcal{L} is *NP* hard.

- 7.7 **EXAMPLE** The **Knapsack** problem starts with a multiset of objects $S = \{s_0, \dots, s_{k-1}\}$, each with a natural number weight $w(s_i)$ and a value $v(s_i)$, along with a weight bound B and value target T . We then look for a knapsack $K \subseteq S$ whose elements have total weight less than or equal to the bound and total value greater than or

equal to the target.

First we check that this problem is in NP . As the witness we can use the k -bit string ω such that $\omega[i] = 1$ if s_i is in the knapsack K , and $\omega[i] = 0$ if it is not. A deterministic machine can verify this witness in polynomial time since it only has to total the weights and values of the elements of K .

To finish we must show that Knapsack is NP hard. It is sufficient to show that a special case is NP hard. Consider the Knapsack instance where $w(s_i) = v(s_i)$ for all $s_i \in S$, and where the two criteria each equal half of the weight total, $B = T = 0.5 \cdot \sum_{0 \leq i < k} w(i)$. This shows that any instance of the Partition problem, which is in the above basic list, can be expressed as a Knapsack instance, so $\text{Partition} \leq_p \text{Knapsack}$, and consequently the latter is NP hard.

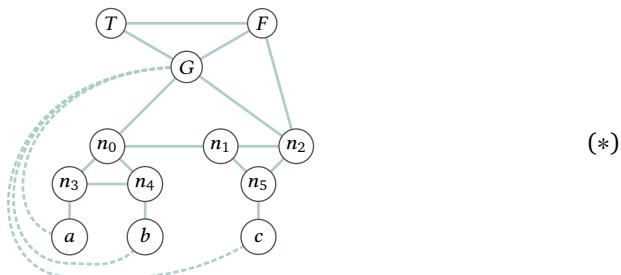
Another common strategy in these proofs is to build the reduction function so that the behavior of the input instance's fundamental units is simulated by subparts of the output instance. Such a construct is a **gadget**.

- 7.8 EXAMPLE Recall that a graph can be three-colored if we can partition its vertices into three categories, the colors, in such a way that no two same-colored vertices are connected by an edge. We will show that the **3-Coloring** problem, the decision problem for $\mathcal{L} = \{\mathcal{G} \mid \text{the graph } \mathcal{G} \text{ has a 3-coloring}\}$, is NP complete.

The easy half is $\mathcal{L} \in NP$. As a witness we use a 3-coloring $\omega = \{C_0, C_1, C_2\}$ where each C_i is a subset of \mathcal{G} 's vertices. Clearly we can produce a verifier that inputs the graph $\sigma = \mathcal{G}$ along with ω and certifies in polytime that ω partitions the vertices and that never do two same-color vertices lie on the same edge.

The other half is to show that \mathcal{L} is NP hard. We will sketch the proof that $3\text{-SAT} \leq_p 3\text{-Coloring}$. The reduction function inputs a propositional logic expression in Conjunctive Normal form. It outputs a graph, which is 3-colorable if and only if the input expression is satisfiable.

The gadget is below. It simulates one clause in the input propositional logic expression, $a \vee b \vee c$ (these are literals so that each could be x or $\neg x$). At the top are nodes labeled T , F , and G . If the graph is to be 3-colored then no two nodes in this triangle can have the same color. At the gadget's bottom are nodes labeled a , b , and c . They are connected to G and consequently each of these three must have either the color of T or the color of F . (The connections are shown dashed to fit with the drawing (**)) below.) Also, because n_2 is connected to both G and F , it must be the color of T .

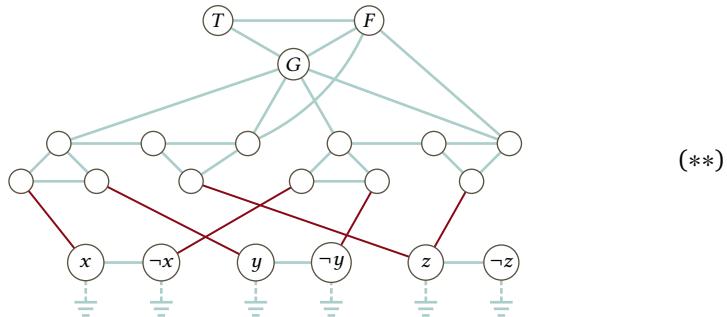


We will verify that this gadget is 3-colorable if and only if nodes a , b , and c are not all the color of F , matching the behavior of the clause $a \vee b \vee c$. For ‘only if’, assume that a , b , and c are the color of F . Then one of n_3 and n_4 is the color of T while the other is the color of G , and hence n_0 is the color of F . Since n_2 is the color of T this implies that n_1 is the color of G and this in turn gives that n_5 is the color of F . That violates 3-colorability because c is the color of F .

For ‘if’, we need only exhibit that a 3-coloring exists for each remaining case.

a	b	c	n_0	n_1	n_2	n_3	n_4	n_5
F	F	T	F	G	T	T	G	F
F	T	F	T	F	T	G	F	G
F	T	T	F	G	T	T	G	F
T	F	F	T	F	T	F	G	G
T	F	T	F	G	T	G	T	F
T	T	F	T	F	T	F	G	G
T	T	T	T	F	T	F	G	G

We finish by describing how to combine multiple gadgets for multiple clauses.



Above is the output graph for the two-clause input expression $E = (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee z)$. On the left is the gadget for the first clause and on the right is the gadget for the second. At the bottom, instead of (*)’s single node a , here there are two nodes, marked x and $\neg x$. Similarly, there are y , $\neg y$, z , and $\neg z$. All six share an edge with G (like (*), this drawing uses dashes for the edges but here they lead to an electrical ground symbol denoting the G connection). Because of this connection, in a 3-colored graph all six of these nodes must have the color of T or the color of F . What’s more, because x is connected to $\neg x$, in a 3-colored graph one of these has the color of T and the other has the color of F , which explains the node names. The same holds for the other bottom node pairs.

The first clause in E is $x \vee y \vee z$ so in the left gadget the reduction function outputs the graph with the highlighted edges lead to the nodes marked x , y , and z . The second clause is $\neg x \vee \neg y \vee z$ so in the right gadget the edges lead to $\neg x$, $\neg y$, and z .

Clearly this gives a 3-colorable graph if and only if the expression is satisfiable.

One of Karp's points was the practical importance of *NP* completeness. Many problems from applications fall into this class. The next example illustrates.

- 7.9 EXAMPLE Scheduling is a rich source of difficult combinatorial problems. One is the problem of scheduling college classes into time slots. Usually colleges start the process by assigning classes to slots and then students try to find classes that they need and that are offered at different times. Imagine if instead the process began with each student submitting their list of desired classes, and then the college tries to find a non-conflicting schedule of slots and rooms to accomodate the requests.

Specifically, consider a college with $t = 12$ available time slots, and that must work $n = 420$ classes into $r = 60$ classrooms. Since $12 \cdot 60 = 720$ and we need only accomodate 420 classes, this might seem easy. But this college has 1724 students and when each one submits their class requests, $S_i = \{c_{i_0}, c_{i_1}, \dots, c_{i_n}\}$, each pair of classes c_{i_p}, c_{i_q} puts the restriction on the college-wide schedule that those two cannot meet at the same time. Can the college find a schedule despite all these constraints?

The **Class Scheduling** problem inputs the number of time slots and rooms, along with the class requests from each student, and decides if there is a way to allocate classes so that there is no conflict.

$$\mathcal{L} = \{\langle t, r, \{S_0, S_1, \dots\} \rangle \mid \text{a nonconflicting schedule exists}\}$$

We will show that this problem is *NP* complete.

It is a member of *NP* because we can take as the witness ω a schedule, a nonconflicting assignment of classes to rooms and times. Writing a verifier that inputs $\sigma = \langle t, r, \{S_0, S_1, \dots\} \rangle$ and ω , and then checks in polytime that ω meets the restrictions is straightforward.

What remains is show that this problem is *NP* hard. This problem is a good fit with the **Graph Colorability** problem with the nodes of the graph as the classes, where two nodes are connected when some student has requested them both, and where nodes are colored the same if they are in the same time slot. (There is a further restriction about the number of available rooms that we will handle along the way.) The prior example proves that **Graph Colorability** is *NP* hard for $k = 3$ colors and an extension of that argument proves the same for any larger k . So we will show that **Class Scheduling** is *NP* hard by showing that **Graph Colorability** \leq_p **Class Scheduling**.

Assume that we are given an instance of **Graph Colorability**, a graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ and a natural number k . Here is the instance of the **Class Scheduling** problem that the reduction function associates with \mathcal{G} : the number of classes is $|\mathcal{V}|$, as is the number of rooms, and the number of time slots is k . Each student takes two classes and there is an edge if and only if it connects some student's two. Clearly from the input instance we can in polytime produce the output instance. Also clearly, the **Graph Colorability** instance has a solution if and only if the associated **Class Scheduling** instance has a nonconflicting schedule.

Before we leave this discussion, we address a natural question: we've seen that lots of problems are NP complete but which problems are not?

Trivially, the definition gives that the empty language and the language of all strings are not complete. Also trivially from the definition, a problem cannot be NP complete if it is not in NP . For instance, as with finding a chess strategy, a problem can be so hard that we cannot even check its solution in polytime.

The more interesting part of the answer is tied to whether $P = NP$ or $P \neq NP$. We will address it in the next subsection but just to not have brushed past the issue, first note that if $P = NP$ then every nontrivial problem is NP hard by Lemma 6.10. So assume that $P \neq NP$. Any algorithms text describes many problems with polytime algorithms and $P \neq NP$ implies that these are not NP complete. So if $P \neq NP$ then most problems from ordinary programming experience are not NP complete.

The most interesting questions concern the **NP intermediate** problems. These are in $NP - P$ but are not NP complete. If $P \neq NP$ we can prove that such problems exist. However, proving that a particular problem of interest is in this category is astonishingly difficult—at this moment the field just does not know how to do it. Two problems of great interest that many experts believe are NP intermediate, but no one can currently prove that, are Prime Factorization[†] and Graph Isomorphism (to decide whether two graphs are isomorphic).

$P = NP$? Every deterministic Turing machine is a nondeterministic machine and so $P \subseteq NP$. Thus every polytime problem is in NP . But what about the other direction—could it be that $P = NP$ and every NP problem is, in a sense, easy?

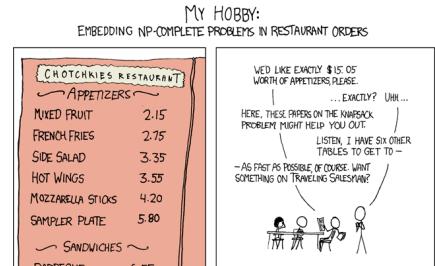
We have seen that one way to think of nondeterministic machines is that they are unboundedly parallel. So the P versus NP question asks: does adding parallelism add speed?

The short answer is that no one knows. There are a number of ways to potentially settle the question. For example, by Lemma 7.4 if there is even one NP complete problem that we can prove is a member of P , then $P = NP$. Conversely, if someone shows that there is an NP problem that is not a member of P then $P \neq NP$. However, despite nearly a half century of effort by many brilliant people, no one has accomplished either one.

To explain all the effort on the question, we first argue for its importance. As formulated in Karp's original paper, the question of whether P equals NP might seem of only technical interest.

A large class of computational problems involve the determination of properties

[†]In 1994, P Shor discovered an algorithm for a quantum computer that solves the Prime Factorization problem in polynomial time. This will have significant implications if we manage to build quantum computers. However, on classical computers most experts believe that this problem is in $NP - P$ but not complete.



Courtesy xkcd.com

of graphs, digraphs, integers, arrays of integers, finite families of finite sets, boolean formulas and elements of other countable domains. Through simple encodings ... these problems can be converted into language recognition problems, and we can inquire into their computational complexity. It is reasonable to consider such a problem satisfactorily solved when an algorithm for its solution is found which terminates within a number of steps bounded by a polynomial in the length of the input. We show that a large number of classic unsolved problems of covering, matching, packing, routing, assignment and sequencing are equivalent, in the sense that either each of them possesses a polynomial-bounded algorithm or none of them does.

These careful words mask the excitement. Karp demonstrated that many problems that people had been struggling with in practice—classic unsolved problems—fall in this category. Researchers who have been looking for an efficient solution to *Vertex Cover* and those who have been working on *Clique* find that they are working on the same problem, in that the two are inter-translatable. By now the list of *NP* complete problems includes determining the best layout of transistors on a chip, developing accurate financial-forecasting models, analyzing protein-folding behavior in a cell, or finding the most energy-efficient airplane wing. So the question of whether *P* equals *NP* is extremely practical, and extremely important.[†]

Researchers often take proving that a problem is *NP* complete to be an ending point; they may feel that continuing to look for an algorithm is a waste since many of the world's best minds have failed to find one. They may turn to finding approximations (see Extra B) or to probabilistic methods.

We next argue that among many similar questions, each of which is important, *P* versus *NP* suggests itself as especially significant. First, a philosophical take. At the start of this book we studied problems that are unsolvable. That is black and white—either a problem is mechanically solvable or it is not. In this chapter we find that many problems are solvable in principle but computing a solution seems to be infeasible. The set *P* consists of the problems that we can feasibly solve. But if $P \neq NP$ then the problems in $NP - P$, including the *NP* complete ones, are ones for which we can verify a correct answer but we cannot reliably find it. The poet R Browning wrote, “Ah, but a man’s reach should exceed his grasp, Or what’s a heaven for?” We can view these problems as a transition between the possible and the impossible.

The sense that the *P* versus *NP* question fits into a larger intellectual setting returns us to the book’s opening. Recall the *Entscheidungsproblem* that was a motivation behind the definition of a Turing machine. It asks for an algorithm that inputs a mathematical statement and decides whether it is true. It is perhaps a caricature, but imagine that the job of mathematicians is to prove theorems.



Robert
Browning,
1812–1889

[†]One indication of its importance is its inclusion on the Clay Mathematics Institute’s list of problems for which there is a one million dollar prize; see <http://www.claymath.org/millennium-problems>. Part of the introduction there says, “[O]ne of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems ... certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear.”

Then the *Entscheidungsproblem* asks if it is possible to replace mathematicians with mechanisms.

In the intervening century we have come to understand, through the work of Gödel and others, that there is a difference between a statement's being true and its being provable. Church and Turing expanded on this insight to show that the *Entscheidungsproblem* is unsolvable. Consequently, we change to asking for an algorithm that inputs statements and decides whether they are provable.

In principle this is simple. A proof is a sequence of statements, $\sigma_0, \sigma_1, \dots, \sigma_k$, where the final statement is the conclusion and where each statement either is an axiom or else follows from the statements before it by an application of a rule of deduction (a typical rule allows the simultaneous replacement of all x 's with $y + 4$'s). A computer could brute-force the question of whether a given statement is provable by doing a dovetail, a breadth-first search of all derivations. If a proof exists then it will appear, eventually.[†]

The difficulty is the ‘eventually’. This algorithm is very slow. Is there a tractable way? In the terminology that we now have, the modified *Entscheidungsproblem* is a decision problem: given a statement σ and a bound, we ask if there is a sequence ω of statements witnessing a proof that ends in σ and that is shorter than the bound. A computer can quickly check whether a given proof is valid—this problem is in NP . With the current status of the P versus NP problem, the answer to the question in the prior paragraph is that no one knows of a fast algorithm but no one can show that there isn't one either.

As far back as 1956, Gödel raised these issues in a letter to von Neumann (this letter did not become public until years later).[‡]

One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n , allows one to decide if there is a proof of F of length n (length = number of symbols). Let $\Psi(F, n)$ be the number of steps the machine requires for this and let $\phi(n) = \max_F \Psi(F, n)$. The question is how fast $\phi(n)$ grows for an optimal machine. One can show that $\phi(n) \geq k \cdot n$. If there really were a machine with $\phi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the *Entscheidungsproblem*, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $\phi(n)$ grows

[†]That is, in a particular subject such as elementary number theory, the set of theorems is computably enumerable. [‡]At the meeting where Gödel, as an unknown fresh PhD, announced his Incompleteness Theorem, the only person who approached him with interest was von Neumann, who was already well established. Later, when Gödel was trying to escape the Nazis, von Neumann wrote to the director of the Institute for Advanced Study, “Gödel is absolutely irreplaceable. He is the only mathematician . . . about whom I would dare to make this statement.” So they were professionally quite close. At the time of the letter, von Neumann had cancer, probably from his work on the Manhattan Project. Gödel was misinformed and wrote, “Since you now, as I hear, are feeling stronger, I would like to allow myself to write you about a mathematical problem, of which your opinion would very much interest me.” Within a year von Neumann had died. We don't know if he replied or even read the letter.

that slowly. . . It would be interesting to know, for instance, the situation concerning the determination of primality of a number and how strongly in general the number of steps in finite combinatorial problems can be reduced with respect to simple exhaustive search.

Again, this brings us back to this book's beginning. Again, we are asking, "What can be done?" But here we are asking about what can be done feasibly.

In summary, we can compare P versus NP with the Halting problem. The Halting problem and related results tell us, in the light of Church's Thesis, what is knowable in principle. The P versus NP question, in contrast, speaks to what we can know in practice.

Discussion Certainly the P versus NP question is the sexiest one in the Theory of Computing today. It has attracted a great deal of gossip. In 2018, a poll of experts found that out of 152 respondents, 88% thought that $P \neq NP$ while only 12% thought that $P = NP$. This subsection discusses some of the intuition involved in the question.



A Selman's plate,
courtesy S Selman

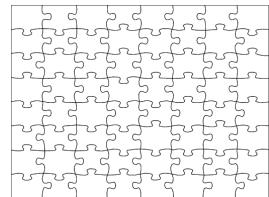
First we address the intuition around the conjecture that $P \neq NP$. One way to think about the question is that a problem is in P if finding a solution is fast, while a problem is in NP if verifying the correctness of a given witness is fast. Then the claim that $P \subseteq NP$ becomes the observation that if a problem is fast to solve then it must be fast to verify. But the other inclusion seems to most experts to be extremely unlikely. For example, speaking informally, S Aaronson has said, "I'd give it a 2 to 3 percent chance that P equals NP . Those are the betting odds that I'd take." Similarly, R Williams puts the chance that $P \neq NP$ at 80%.

As early as Karp's original paper there was a sense that $P \neq NP$ was the natural guess. Here is the first paragraph of that paper.

All the general methods presently known for computing the chromatic number of a graph, deciding whether a graph has a Hamiltonian circuit, or solving a system of linear inequalities in which the variables are constrained to be 0 or 1, require a combinatorial search for which the worst case time requirement grows exponentially with the length of the input. In this paper we give theorems which strongly suggest . . . that these problems, as well as many others, will remain intractable perpetually.

This intuition comes from a number of sources but an important one is the everyday experience that there is a genuine difference between the difficulty of finding a solution and that of verifying that an existing solution is correct.

Imagine a jigsaw puzzle. We perceive that if a demon gave us an assembled puzzle ω , then checking that it is correct is very much easier than it would have been to work out the solution from scratch. Checking for correctness is mechanical, tedious. But the finding of a solution, we perceive, is creative—we feel that solving a jigsaw puzzle by brute-force trying every possible piece against every other is too much computation to be practical.



Similarly, mathematicians find that verifying the correctness of a formally-described proof is routine. But finding that proof in the first place may be the work of a lifetime, or more.

Some commentators have extended this way of thinking beyond the narrow bounds of Theoretical Computer Science. One is A Wigderson, “[$P = NP$ would be] utopia for the quest for knowledge and technological development by humans. There would be a short program that, for every mathematical statement and given page limit, would quickly generate a proof of that length, if one exists! There would be a short program which, given detailed constraints on any engineering task, would quickly generate a design which meets the given criteria, if one exists. The design of new drugs, cheap energy, better strains of food, safer transportation, and robots that would release us from all unpleasant chores, would become a triviality.” He continues, “. . . most people revolt against the idea that such amazing discoveries like Wiles’s proof of Fermat, Einstein’s relativity, Darwin’s evolution, Edison’s inventions, as well as all the ones we are awaiting, could be produced in succession quickly by a mindless robot. . . . If $P = NP$, any human (or computer) would have the sort of reasoning power traditionally ascribed to deities, and this seems hard to accept.” Cook is of the same mind, “. . . Similar remarks apply to diverse creative human endeavors, such as designing airplane wings, creating physical theories, or even composing music. The question in each case is to what extent an efficient algorithm for recognizing a good result can be found.” Perhaps it is hyperbole to say that if $P = NP$ then writing great symphonies would be a job for computers, a job for mechanisms, but it is correct to say that if $P = NP$ and if we can write fast algorithms to recognize excellent music—and our everyday experience with Artificial Intelligence makes this seem more and more a possibility—then we could have fast mechanical writers of excellent music.

We finish with a taste of the intuition behind the contrarian view, the sense that perhaps $P = NP$ could be right.

Many observers have noted that there are cases where everyone “knew” that some algorithm was the fastest but in the end it proved not to be so. The section on Big- \mathcal{O} begins with one, the grade school algorithm for multiplication. Another is the problem of solving systems of linear equations. The Gauss’s Method algorithm, which runs in time $\mathcal{O}(n^3)$, is perfectly natural and had been known for centuries without anyone making improvements. However, while trying to prove that Gauss’s Method is optimal, V Strassen found a $\mathcal{O}(n^{\lg 7})$ method ($\lg 7 \approx 2.81$).[†]

A more dramatic speedup happens with the **Matching** problem. It starts with a graph whose vertices represent people and such that pairs of vertices are connected if the people are compatible. We want a set of edges that is maximal, and such that no two edges share a vertex. The naive algorithm tries all possible match

[†]Here is an analogy: consider the problem of evaluating $2p^3 + 3p^2 + 4p + 5$. Someone might claim that writing it as $2 \cdot p \cdot p \cdot p + 3 \cdot p \cdot p + 4 \cdot p + 5$ makes obvious that it requires six multiplications. But rewriting it as $p \cdot (p \cdot (2 \cdot p + 3) + 4) + 5$ shows that it can be done with just three. That is, naturalness and obviousness do not guarantee that something is correct. Without a proof, we must worry that someone will produce a clever way to do the job with less.

sets, which takes 2^m checks where m is the number of edges. Even with only a hundred people there are more things to try than atoms in the universe. But since the 1960's we have an algorithm that runs in polytime.

Every day on the Theory of Computing blog feed there are examples of researchers producing algorithms faster than the ones previously known. A person can certainly have the sense that we are only just starting to explore what is possible with algorithms. R J Lipton captured this feeling.

Since we are constantly discovering new ways to program our “machines,” why not a discovery that shows how to factor? or how to solve SAT? Why are we all so sure that there are no great new programming methods still to be discovered? . . . I am puzzled that so many are convinced that these problems could not fall to new programming tricks, yet that is what is done each and every day in their own research.

Knuth has a related but somewhat different take.

Some of my reasoning is admittedly naive: It’s hard to believe that $P \neq NP$ and that so many brilliant people have failed to discover why. On the other hand if you imagine a number M that’s finite but incredibly large . . . then there’s a humongous number of possible algorithms that do n^M bitwise addition or shift operations on n given bits, and it’s really hard to believe that all of those algorithms fail.

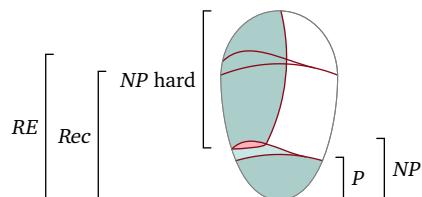
My main point, however, is that I don’t believe that the equality $P = NP$ will turn out to be helpful even if it is proved, because such a proof will almost surely be nonconstructive. Although I think M probably exists, I also think human beings will never know such a value. I even suspect that nobody will even know an upper bound on M .

Mathematics is full of examples where something is proved to exist, yet the proof tells us nothing about how to find it. Knowledge of the mere existence of an algorithm is completely different from the knowledge of an actual algorithm.

Of course, all this is speculation. Speculating is fun, and in order to make progress in their work, people must have some intuition. But in the end, we look to settle the question with proof.[†]

V.7 Exercises

7.10 This diagram is an extension of one we saw earlier. (It assumes that $P \neq NP$.)



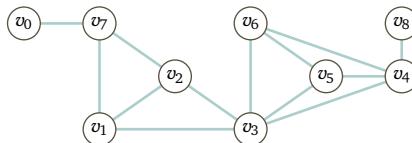
On that, locate these languages.

[†]There is also in the air what we could think of as a third side to this debate. Theory should be guide for practice. There is some recent evidence, as suggested by two Topics at the end of this chapter, that the fact that a problem is NP complete does not mean that in practice it is intractable, even for reasonably large-scale instances. So we can ask whether the attention given to the question puts the focus of the community in exactly the right place.

- (A) $K = \{\sigma \mid \sigma \text{ represents } x \in \mathbb{N} \text{ where } \phi_x(x) \downarrow\}$
 (B) \emptyset
 (C) $\mathcal{L}_B = \{\langle \mathcal{G}, v_0, v_1 \rangle \mid \text{there is a path from } v_0 \text{ to } v_1 \text{ of length at most } B\}$
 (D) SAT
- ✓ 7.11 You hear someone say, “The **Satisfiability** problem is *NP* because it is not computable in polynomial time, so far as we know.” It’s a short sentence but find three mistakes.
- ✓ 7.12 Someone in your class says, “I will show that the **Hamiltonian Circuit** problem is not in *P*, which will demonstrate that *P* \neq *NP*. The algorithm to solve a given instance \mathcal{G} of the **Hamiltonian Circuit** problem is: generate all permutations of \mathcal{G} ’s vertices, test each to find if it is a circuit, and if any circuits appear then accept the input, else reject the input. For sure that algorithm is not polynomial, since the first step is exponential.” Where is their mistake?
- ✓ 7.13 Your friend says, “The problem of recognizing when one string is a substring of another has a polytime algorithm, so it is not in *NP*.” They have misspoken; help them out.
- 7.14 Someone in your study group wants to ask your professor, “Is the brute force algorithm for solving the **Satisfiability** problem *NP* complete?” Explain to them that it isn’t a sensible question, that they are making a type error.
- 7.15 True or false?
- (A) The collection *NP* is a subset of the *NP* complete sets, which is a subset of *NP* hard.
 (B) The collection *NP* is a specialization of *P* to nondeterministic machines, so it is a subset of *P*.
- ✓ 7.16 Assume that *P* \neq *NP*. Which of these statements can we infer from the fact that the **Prime Factorization** problem is in *NP*, but is not known to be *NP* complete?
 (A) There exists an algorithm for arbitrary instances of the **Prime Factorization** problem.
 (B) There exists an algorithm that efficiently solves arbitrary instances of this problem.
 (C) If we found an efficient algorithm for the **Prime Factorization** problem then we could immediately use it to solve **Traveling Salesman**.
- ✓ 7.17 Suppose that $\mathcal{L}_1 \leq_p \mathcal{L}_0$. For each, decide if you can conclude it. (A) If \mathcal{L}_0 is *NP* complete then so is \mathcal{L}_1 . (B) If \mathcal{L}_1 is *NP* complete then so is \mathcal{L}_0 .
 (C) If \mathcal{L}_0 is *NP* complete and \mathcal{L}_1 is in *NP* then \mathcal{L}_1 is *NP* complete. (D) If \mathcal{L}_1 is *NP* complete and \mathcal{L}_0 is in *NP* then \mathcal{L}_0 is *NP* complete. (E) It cannot be the case that both \mathcal{L}_0 and \mathcal{L}_1 are *NP* complete (F) If \mathcal{L}_1 is in *P* then so is \mathcal{L}_0 .
 (G) If \mathcal{L}_0 is in *P* then so is \mathcal{L}_1 .
- 7.18 Show that these are in *NP* but are not *NP* complete, assuming that *P* \neq *NP*.
 (A) The language of even numbers.
 (B) The language $\{\mathcal{G} \mid \mathcal{G} \text{ has a vertex cover of size at most four}\}$.

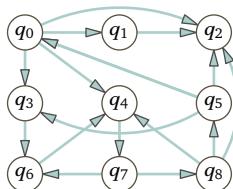
7.19 If $P = NP$ then what happens with NP complete sets? Show that if $P = NP$ then every nontrivial language in P is NP complete.

- ✓ 7.20 **Traveling Salesman** is NP complete. From $P \neq NP$ which of the following statements could we infer?
 - (A) No algorithm solves all instances of **Traveling Salesman**.
 - (B) No algorithm quickly solves all instances of **Traveling Salesman**.
 - (C) **Traveling Salesman** is in P .
 - (D) All algorithms for **Traveling Salesman** run in polynomial time.
- ✓ 7.21 Prove that the **4-Satisfiability** problem is NP hard.
- ✓ 7.22 The **Hamiltonian Path** problem inputs a graph and decides if there are two vertices in that graph such that there is a path between those two that contains all the vertices.
 - (A) Show that **Hamiltonian Path** is in NP .
 - (B) This graph has a Hamiltonian path from v_0 to v_8 . Find it.



Why must those two be the endpoints?

- (c) Show that **Hamiltonian Circuit** \leq_p **Hamiltonian Path**.
- (d) Conclude that the **Hamiltonian Path** problem is NP complete.
- ✓ 7.23 The **Longest Path** problem is to input a graph and find the longest simple path in that graph.
 - (A) Find the longest path in this graph.

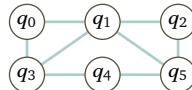


- (b) Remembering the technique for converting an optimization problem to a language decision problem by using bounds, state this as a language decision problem. Show that **Longest Path** $\in NP$.
- (c) Show that the **Hamiltonian Path** problem reduces to **Longest Path**. Hint: leverage the bound from the prior item.
- (d) Use the prior exercise to conclude that the **Longest Path** problem is NP complete.
- ✓ 7.24 The **Subset Sum** problem inputs a multiset T and a target $B \in \mathbb{N}$, and decides if there is a subset $\hat{T} \subseteq T$ whose elements add to the target. The **Partition** problem inputs a multiset S and decides whether or not it has a subset $\hat{S} \subset S$ so that the sum of elements of \hat{S} equals the sum of elements not in that subset.

- (A) Find a subset of $T = \{3, 4, 6, 7, 12, 13, 19\}$ that adds to $B = 30$.
- (B) Find a partition of $S = \{3, 4, 6, 7, 12, 13, 19\}$.
- (C) Show that if the sum of the elements in a set is odd then the set has no partition.
- (D) Express each problem as a language decision problem.
- (E) Prove that $\text{Partition} \leq_p \text{Subset Sum}$. (*Hint:* handle separately the case where the sum of elements in S is odd.)
- (F) Conclude that **Subset Sum** is NP complete.

7.25 The **Independent Set** problem inputs a graph and a bound, and decides if there is a set of vertices, of size at least equal to the bound, that are not connected to each other by an edge.

- (A) Find an independent set in this graph.



- (B) State **Independent Set** as a language decision problem.
- (C) Decide if $E = (P_0 \vee \neg P_1 \vee \neg P_2) \wedge (P_1 \vee P_2 \vee \neg P_3)$ is satisfiable.
- (D) State **3-Satisfiability** as a language decision problem.
- (E) With the expression E , make a triangle for each of the two clauses, where the vertices of the first are labeled v_0, \bar{v}_1 , and \bar{v}_2 , while the vertices of the second are labeled w_1, w_2 , and \bar{w}_3 . In addition to the edges forming the triangles, also put one connecting \bar{v}_1 with w_1 , and one connecting \bar{v}_2 with w_2 .
- (F) Sketch an argument that **3-Satisfiability** \leq_p **Independent Set**.

- ✓ 7.26 The difficulty in settling $P = NP$ is to prove lower bounds. That is, the trouble lies in showing, for a given problem, that any algorithm at all must use such-and-such many steps. One common mistake is to reason that any algorithm for the problem must take at least as many steps as the length of the input, thinking that to compute the output the algorithm must at least read all of the input. We will exhibit a familiar problem for which this isn't true.

Consider the successor function. Show that it can be computed on a Turing machine without reading all of the input. More, show how to compute it in constant time, that it has an algorithm whose running time when the input is large is the same as the running time when the input is small. Assume that the algorithm is given the input n in unary with the head under the leftmost 1, and that it ends with $n + 1$ -many 1's and with the head under the leftmost 1.

7.27 Do we know of any problems in NP and not in P , and that are not NP complete?

7.28 Find three languages so that $\mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$, and $\mathcal{L}_2, \mathcal{L}_0$ are NP complete, while $\mathcal{L}_1 \in P$.

7.29 Prove Lemma 7.4.

7.30 The class P has some nice closure properties, and so does NP . (A) Prove that NP is closed under union, so that if $\mathcal{L}, \hat{\mathcal{L}} \in NP$ then $\mathcal{L} \cup \hat{\mathcal{L}} \in NP$. (B) Prove that

NP is closed under concatenation. (c) Argue that no one can prove that NP is not closed under set complement.

7.31 Is the set of NP complete sets countable or uncountable?

7.32 We will sketch a proof that the Halting problem is NP hard but not NP . Consider the language $\mathcal{HP} = \{\langle \mathcal{P}_e, x \rangle \mid \phi_e(x) \downarrow\}$. (A) Show that $\mathcal{HP} \notin NP$. (B) Sketch an argument that for any problem $\mathcal{L} \in NP$, there is a polynomial time computable verifier, $f: \mathbb{B}^* \rightarrow \mathbb{B}^*$, such that $\sigma \in \mathcal{L}$ if and only if $f(\sigma) \in \mathcal{HP}$.

SECTION

V.8 Other classes

There are many other defined complexity classes. The next class is quite natural.

EXP The Satisfiability problem is a touchstone result among problems in NP . We have discussed computing it using a nondeterministic Turing machine that is unboundedly parallel, or alternatively using a witness and verifier. But, naively, in the familiar computational setting of a deterministic machine, it appears that to solve it, we must go through the truth table line by line. That is, SAT appears to take exponential time.

In this chapter's first section we included $\mathcal{O}(2^n)$ and $\mathcal{O}(3^n)$, and by extension other exponentials, in the list of common orders of growth.

8.1 **DEFINITION** A language decision problem is an element of the complexity class EXP if there is an algorithm for solving it that runs in time $\mathcal{O}(b^{p(n)})$ for some constant base b and polynomial p .

A first, informal, take is that EXP contains nearly every problem with which we concern ourselves in practice—it contains most problems that we seriously hope ever to attack. In contrast with polytime, where a rough summary is that its problems all have an algorithm that can conceivably be used, for the hardest problems in EXP , even the best algorithms are just too slow.

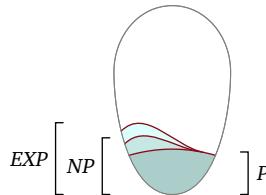
8.2 **LEMMA** $P \subseteq NP \subseteq EXP$

Proof Fix $\mathcal{L} \in NP$. We can verify \mathcal{L} on a deterministic Turing machine \mathcal{P} in polynomial time using a witness whose length is bounded by the same polynomial. Let this problem's bound be n^c .

We will decide \mathcal{L} in exponential time by brute-forcing it: we will use \mathcal{P} to run every possible verification. Trying any single witness requires polynomial time, n^c . Witnesses are in binary so for length ℓ there are $\sum_{0 \leq i \leq \ell} 2^i = 2^{\ell+1} - 1$ many possible ones; In total then, brute force requires $\mathcal{O}(n^c 2^{n^c})$ operations. Finish by observing that $n^c 2^{n^c}$ is in $\mathcal{O}(2^{n^c})$. \square

We know by a result called the Time Hierarchy Theorem that the three classes are not all equal. But where the division is, we don't know. Just as we don't today have a proof that P is a proper subset of NP , we also don't know whether or not there are NP complete problems that absolutely require exponential time. The

class NP could conceivably be contained in a smaller deterministic time complexity class—for instance, maybe **Satisfiability** can be solved in less than exponential time. But we just don't know.



8.3 FIGURE: The blob encloses all problems. Shaded are the three classes P , NP , and EXP . They are drawn with strict containment, which most experts guess is the true arrangement, but no one knows for sure.

Time Complexity Researchers have generalized to many more classes, trying to capture various aspects of computation. For instance, the impediment that a programmer runs across first is time.

- 8.4 **DEFINITION** Let $f: \mathbb{N} \rightarrow \mathbb{N}$. A decision problem for a language is an element of $DTIME(f)$ if it is decided by a deterministic Turing machine that runs in time $\mathcal{O}(f)$. A problem is an element of $NTIME(f)$ if it is decided by a nondeterministic Turing machine that runs in time $\mathcal{O}(f)$.
- 8.5 **LEMMA** A problem is polytime, P , if it is a member of $DTIME(n^c)$ for some power $c \in \mathbb{N}$.

$$P = \bigcup_{c \in \mathbb{N}} DTIME(n^c) = DTIME(n) \cup DTIME(n^2) \cup DTIME(n^3) \cup \dots$$

The matching statements hold for NP and EXP .

$$\begin{aligned} NP &= \bigcup_{c \in \mathbb{N}} NTIME(n^c) = NTIME(n) \cup NTIME(n^2) \cup NTIME(n^3) \cup \dots \\ EXP &= \bigcup_{c \in \mathbb{N}} DTIME(2^{n^c}) = DTIME(2^n) \cup DTIME(2^{n^2}) \cup DTIME(2^{n^3}) \cup \dots \end{aligned}$$

Proof The only equality that is not immediate is the last one. Recall that a problem is in EXP if an algorithm for it that runs in time $\mathcal{O}(b^{p(n)})$ for some constant base b and polynomial p . The equality above only uses the base 2. To cover the discrepancy, we will show that $3^n \in \mathcal{O}(2^{(n^2)})$. Consider $\lim_{x \rightarrow \infty} 2^{(x^2)} / 3^x$. Rewrite the fraction as $(2^x/3)^x$, which when $x > 2$ is larger than $(4/3)^x$, which goes to infinity. This argument works for any base, not just $b = 3$. \square

- 8.6 **REMARK** While the above description of NP reiterates its naturalness, as we saw earlier, the characterization that proves to be most useful in practice is that a problem \mathcal{L} is in NP if there is a deterministic Turing machine \mathcal{V} such that for each

input σ there is a polynomial length witness ω and the verification on \mathcal{V} for σ using ω takes polytime.

Space Complexity We can consider how much space is used in solving a problem.

- 8.7 **DEFINITION** A deterministic Turing machine **runs in space** $s: \mathbb{B}^* \rightarrow \mathbb{R}^+$ if for all but finitely many inputs σ , the computation on that input uses less than or equal to $s(|\sigma|)$ -many cells on the tape. A nondeterministic Turing machine **runs in space** s if for all but finitely many inputs σ , every computation path on that input takes less than or equal to $s(|\sigma|)$ -many cells.

The machine must use less than or equal to $s(|\sigma|)$ -many cells even on non-accepting computations.

- 8.8 **DEFINITION** Let $s: \mathbb{N} \rightarrow \mathbb{N}$. A language decision problem is an element of **$DSPACE(s)$** , or **$SPACE(s)$** , if that language is decided by a deterministic Turing machine that runs in space $O(s)$. A problem is an element of **$NSPACE(s)$** if the language is decided by a nondeterministic Turing machine that runs in space $O(s)$.

The definitions arise from a sense we have of a symmetry between time and space, that they are both examples of computational resources. (There are other resources; for instance we may want to minimize disk reading or writing, which may be quite different than space usage.) But space is not just like time. For one thing, while a program can take a long time but use only a little space, the opposite is not possible.

- 8.9 **LEMMA** Let $f: \mathbb{N} \rightarrow \mathbb{N}$. Then $DTIME(f) \subseteq DSPACE(f)$. As well, this holds for nondeterministic machines, $NTIME(f) \subseteq NSPACE(f)$.

Proof A machine can use at most one cell per step. □

- 8.10 **DEFINITION**

$$PSPACE = \bigcup_{c \in \mathbb{N}} DSPACE(n^c) = DSPACE(n) \cup DSPACE(n^2) \cup \dots$$

$$NPSPACE = \bigcup_{c \in \mathbb{N}} NSPACE(n^c) = NSPACE(n) \cup NSPACE(n^2) \cup \dots$$

So **PSPACE** is the class of problems that can be solved by a deterministic Turing machine using only a polynomially-bounded amount of space, regardless of how long the computation takes.

As even those preliminary results suggest, restricting by space instead of time allows for a lot more power.

- 8.11 **LEMMA** $P \subseteq NP \subseteq PSPACE$

Proof For any problem in **NP**, check all possible witness strings ω . These take at most polynomial space. If any proof string works then the answer to the problem is ‘yes’. Otherwise, the answer is ‘no’. □

Note that the method in the proof may take exponential time but it takes only polynomial space.

Here is a result whose proof is beyond our scope, but that serves as a caution that time and space are very different. We don't know whether deterministic polynomial time equals nondeterministic polynomial time, but we do know the answer for space.

8.12 **THEOREM (SAVITCH'S THEOREM)** $PSPACE = NPSPACE$

We finish with a list of the most natural complexity classes.

8.13 **DEFINITION** These are the **canonical complexity classes**

1. $L = DSPACE(\lg n)$, **deterministic log space** and $NL = NSPACE(\lg n)$, **nondeterministic log space**
2. P , **deterministic polynomial time** and NP , **nondeterministic polynomial time**
3. $E = \bigcup_{k=1,2,\dots} DTIME(k^n)$ and $NE = \bigcup_{k=1,2,\dots} NTIME(k^n)$
4. $EXP = \bigcup_{k=1,2,\dots} DTIME(2^{n^k})$, **deterministic exponential time** and $NEXP = \bigcup_{k=1,2,\dots} NTIME(2^{n^k})$, **nondeterministic exponential time**
5. $PSPACE$, **deterministic polynomial space**
6. $EXPSPACE = \bigcup_{k=1,2,\dots} DSPACE(2^{n^k})$, **deterministic exponential space**

The Zoo Researchers have studied a great many complexity classes. There are so many that they have been gathered into an online Complexity Zoo, at complexityzoo.uwaterloo.ca/.

One way to understand these classes is that defining a class asks a type of Theory of Computing question. For instance, we have already seen that asking whether NP equals P is a way of asking whether unbounded parallelism makes any essential difference—can a problem change from intractable to tractable if we switch from a deterministic to a nondeterministic machine? Similarly, we know that $P \subseteq PSPACE$. In thinking about whether the two are equal, researchers are considering the space-time tradeoff: if you can solve a problem without much memory does that mean you can solve it without using much time?

Here is one extra class, to give some flavor of the possibilities. For more, see the Zoo.

The class **BPP , Bounded-Error Probabilistic Polynomial Time**, contains the problems solvable by an nondeterministic polytime machine such that if the answer is ‘yes’ then at least two-thirds of the computation paths accept and if the answer is ‘no’ then at most one-third of the computation paths accept. (Here all computation paths have the same length.) This is often identified as the class of feasible problems for a computer with access to a genuine random-number source. Investigating whether BPP equals P is asking whether every efficient randomized algorithm can be made deterministic: are there some problems for which there are fast randomized algorithms but no fast deterministic ones?

On reading in the Zoo, a person is struck by two things. There are many, many results listed—we know a lot. But there also are many questions to be answered—breakthroughs are there waiting for a discoverer.

V.8 Exercises

- ✓ 8.14 Give a naive algorithm for each problem that is exponential. (A) Subset Sum problem (B) k Coloring problem
- 8.15 Show that $n!$ is $2^{\mathcal{O}(n^2)}$. Show that Traveling Salesman $\in EXP$.
- ✓ 8.16 This illustrates how large a problem can be and still be in EXP . Consider a game that has two possible moves at each step. The game tree is binary.
 - (A) How many elementary particles are there in the universe?
 - (B) At what level of the game tree will there be more possible branches than there are elementary particles?
 - (C) Is that longer than a chess game can reasonably run?
- 8.17 We will show that a polynomial time algorithm that calls a polynomial time subroutine can run, altogether, in exponential time.
 - (A) Verify that the grade school algorithm for multiplication gives that squaring an n -bit integer takes time $\mathcal{O}(n)$.
 - (B) Verify that repeated squaring of an n -bit integer gives a result that has length $2^i n$, where i is the number of squarings.
 - (C) Verify that if your polynomial time algorithm calls a squaring subroutine n times then the complexity is $\mathcal{O}(4^n n^2)$, which is exponential.

EXTRA

V.A RSA Encryption

In this chapter we have built up the sense that there are functions that are intractible to compute. Here we see that we can leverage this to engineering advantage. Here we describe the celebrated RSA encryption system that is used to protect Internet commerce.

One of the great things about the interwebs, besides that you can get free books, is that you can buy stuff. You send a credit card number and a couple of days later the stuff appears. For this to be practical, your credit card number must be encrypted.

When you visit a web site using a `https` address, that site sends you information, called a key, that your browser uses to encrypt your card number. The web site then uses a different key to decrypt. This is an important point: the decrypter must differ from the encrypter since people on the net can see the encrypter information that the site sent you. But the site keeps the decrypter information private. These two, encrypter and decrypter, form a matched pair. We will describe the mathematical technologies that make this work.

The arithmetic We can take the view that everything on a computer is numbers. Consider the message ‘send money’. Its ASCII encoding is 115 101 110 100 32 109 111 110 101 121. Converting to a bitstring gives 01110011 01100101 01101110 01100100 00100000 01101101 01101111 01101110 01100101 01111001. In decimal that’s 544 943 221 199 950 100 456 825. So there is no loss in generality in viewing everything we do, including encryption systems, as numerical operations.

To make encryption systems, mathematicians and computer scientists have leveraged that there are things we can do easily but that we do not know how to easily undo — there are operations we can use for encryption that are fast, but such that the operations needed to decrypt (without the decrypter) are believed to be so slow that they are completely impractical. So this is the engineering of Big- \mathcal{O} .

We will describe an algorithm based on the **Factoring** problem. We have algorithms for multiplying numbers that are fast. By comparison, the algorithms that we have for starting with a number and decomposing it into factors are quite slow. To illustrate this, you might contrast the time it takes you to multiply two four-digit numbers by hand with the time it takes you to factor an eight-digit number chosen at random. For that second job set aside an afternoon; it’ll take a while.

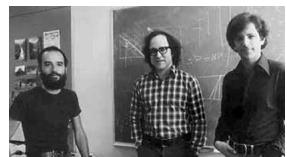
The algorithm that we shall describe exploits this difference. It was invented in 1976 by three graduate students, R Rivest, A Shamir, and L Adleman. Rivest read a paper proposing the idea of key pairs and decided to develop an implementation. Over a year, he and Shamir came up with a number of ideas and for each Adleman would then produce a way to break it. Finally they thought to use Fermat’s Little Theorem (see below). Adleman was unable to break it since, he said, it seemed that only solving the **Factoring** problem would break it and no one knew how to do that. Their algorithm, called RSA, was first announced in Martin Gardner’s *Mathematical Games* column in the August 1977 issue of *Scientific American*. It generated a tremendous amount of interest and excitement.

The basis of RSA is to find three numbers, a **modulus** n , an **encrypter** e , and a **decrypter** d , related by this equation (here m is the message, as a number).

$$(m^e)^d \equiv m \pmod{n}$$

The encrypted message is $m^e \pmod{n}$. To decrypt it, to recover m , calculate $(m^e)^d \pmod{n}$. These three are chosen so that knowing e and n , or even m , still leaves a potential secret-cracker who is looking for d with an extremely difficult job.

To choose them, first choose distinct prime numbers p and q . Pick these at random so they are of about equal bit-lengths. Compute $n = pq$ and $\varphi(n) = (p - 1) \cdot (q - 1)$. Next, choose e with $1 < e < \varphi(n)$ that is relatively prime to n .



Adi Shamir (b 1952), Ron Rivest (b 1947), Leonard Adleman (b 1945)

Finally, find d as the multiplicative inverse of e modulo n . (We shall show below that all these operations, including using the keys for encryption and decryption, can be done quickly.)

The pair $\langle n, e \rangle$ is the **public key** and the pair $\langle n, d \rangle$ is the **private key**. The length of d in bits is the **key length**. Most experts consider a key length of 2 048 bits to be secure for the mid-term future, until 2030 or so, when computers will have grown in power enough that they may be able to use an exhaustive brute-force search to find d . Quite cautious people might use 3 072 bits.

- A.1 EXAMPLE Alice chooses the primes $p = 101$ and $q = 113$ (these are too small to use in practice but are good for an illustration) and then calculates $n = pq = 11\,413$ and $\varphi(n) = (p - 1)(q - 1) = 11\,200$. To get the encrypter she randomly picks numbers $1 < e < 11\,200$ until she gets one that is relatively prime to 11 200, choosing $e = 3533$. She publishes her public key $\langle n, e \rangle = \langle 11\,413, 3\,533 \rangle$ on her home page. She computes the decrypter $d = e^{-1} \bmod 11\,200 = 6\,597$, and finds a safe place to store her private key $\langle n, d \rangle = \langle 11\,413, 6\,597 \rangle$.

Bob wants to say ‘Hi’. In ASCII that’s 01001000 01101001. If he converted that string into a single decimal number it would be bigger than n so he breaks it into two substrings, getting the decimals 72 and 105. Using her public key he computes

$$72^{3533} \bmod 11413 = 10496 \quad 105^{3533} \bmod 11413 = 4861$$

and sends Alice the sequence $\langle 10496, 4861 \rangle$. Alice recovers his message by using her private key.

$$10496^{6597} \bmod 11413 = 72 \quad 4861^{6597} \bmod 11413 = 105$$

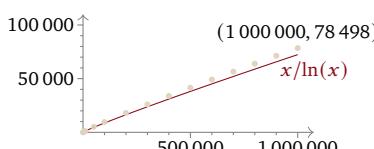
The arithmetic, fast We’ve just illustrated that RSA uses invertible operations. There are lots of ways to get invertible operations so our understanding of RSA is incomplete unless we know why it uses these particular operations. As discussed above, the important point is that they can be done quickly, but undoing them, finding the decrypter, is believed to take a very long time.

We start with a classic, beautiful, result from Number Theory.

- A.2 THEOREM (PRIME NUMBER THEOREM) The number of primes less than $n \in \mathbb{N}$ is approximately $n/\ln(n)$. That is, this limit is 1.

$$\lim_{n \rightarrow \infty} \frac{\text{number of primes less than } n}{(n/\ln n)}$$

This shows the number of primes less than n for some values up to a million.



This theorem says that primes are common. For example, the number of primes less than 2^{1024} is about $2^{1024}/\ln(2^{1024}) \approx 2^{1024}/709.78 \approx 2^{1024}/2^{9.47} \approx 2^{1015}$. Said another way, if we choose a number n at random then the probability that it is prime is about $1/\ln(n)$ and so a random number that is 1024 bits long will be a prime with probability about $1/(\ln(2^{1024})) \approx 1/710$. On average we need only select 355 odd numbers of about that size before we find a prime. Hence we can efficiently generate large primes by just picking random numbers, as long as we can efficiently test their primality.

On our way to giving an efficient way to test primality, we observe that the operations of multiplication and addition modulo m are efficient.

- A.3 **EXAMPLE** Multiplying 3 915 421 by 52 567 004 modulo 3 looks hard. The naive approach is to first take their product and then divide by 3 to find the remainder. But there is a more efficient way. Rather than multiply first and then reduce modulo m , reduce first and then multiply. That is, we know that if $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$ then $ac \equiv bd \pmod{m}$ and so since $3\ 915\ 421 \equiv 1 \pmod{3}$ and $52\ 567\ 004 \equiv 2 \pmod{3}$ we have this.

$$3\ 915\ 421 \cdot 52\ 567\ 004 \equiv 1 \cdot 2 \pmod{3} \equiv 2 \pmod{3}$$

Similarly, exponentiation modulo m is also efficient, both in time and in space.

- A.4 **EXAMPLE** Consider raising 4 to the 13-th power, modulo $m = 497$. The naive approach would be to raise 4 to the 13-th power, which is a very large number, and then reduce modulo 497. But there is a better way.

Start by expressing the power 13 in base 2 as $13 = 8 + 4 + 1 = 1101_2$. So, $4^{13} = 4^8 \cdot 4^4 \cdot 4^1$. If we can efficiently get those powers then the prior example says that we can multiply them modulo m efficiently, and we will be all set.

Get these powers by repeated squaring, modulo m . Start with $p = 1$. Squaring gives 4^2 , then squaring again gives 4^4 , and squaring again gives 4^8 . Squaring modulo m is just a multiplication, which we can do efficiently.

The last thing that we need for efficiently testing primality is to efficiently find the multiplicative inverse modulo m . Recall that two numbers are **relatively prime** or **coprime** if their greatest common divisor is 1. For example, $15 = 3 \cdot 5$ and $22 = 2 \cdot 11$ are relatively prime.

- A.5 **LEMMA** If a and m are relatively prime then there is an inverse for a modulo m , a number k such that $a \cdot k \equiv 1 \pmod{m}$

Proof Because the greatest common divisor of a and m is 1, Euclid's algorithm produces a linear combination of the two, a $sa + tm$ for some $s, t \in \mathbb{Z}$, that adds to 1. Doing the operations modulo m gives $sa + tm \equiv 1 \pmod{m}$. Since tm is a multiple of m , we have $tm \equiv 0 \pmod{m}$, leaving $sa \equiv 1 \pmod{m}$, and s is the inverse of a modulo m . \square

Euclid's algorithm is efficient, both in time and space, so finding an inverse modulo m is efficient.

With that we can efficiently test for primes. The simplest way to test whether a number is prime is to try dividing it by all possible factors. But that is very slow. There is a faster way, based on the next result.

- A.6 **THEOREM (FERMAT'S LITTLE THEOREM)** For a prime p , if $a \in \mathbb{Z}$ is not divisible by p then $a^{p-1} \equiv 1 \pmod{p}$.

Proof Let a be an integer not divisible by the prime p . Multiply a by each number $i \in \{1, \dots, p-1\}$ and reduce modulo p to get the numbers $r_i = ia \pmod{p}$.

We will show that the set $R = \{r_1, \dots, r_{p-1}\}$ equals the set $P = \{1, \dots, p-1\}$. First, $R \subseteq P$. Because p is prime and does not divide i or a , it does not divide their product ia . Thus $r_i \not\equiv 0 \pmod{p}$ and so all the r_i are members of the set $\{1, \dots, p-1\}$.

To get inclusion the other way, $P \subseteq R$, we show that if $i_0 \neq i_1$ then $r_{i_0} \neq r_{i_1}$. For, with $r_0 - r_1 = i_0a - i_1a = (i_0 - i_1)a$, because p is prime and does not divide $i_0 - i_1$ or a as each is smaller in absolute value than p , it does not divide their product. That means that the two sets have the same number of elements, so $P \subseteq R$.

To finish, multiply together the elements of that set.

$$\begin{aligned} a \cdot 2a \cdots (p-1)a &\equiv 1 \cdot 2 \cdots (p-1) \pmod{p} \\ (p-1)! \cdot a^{p-1} &\equiv (p-1)! \pmod{p} \end{aligned}$$

Cancelling the $(p-1)!$'s gives the result. □

- A.7 **EXAMPLE** Let the prime be $p = 7$. Of course, any natural number a with $0 < a < p$ is not divisible by p . This list verifies that $a^{p-1} - 1$ is divisible by p .

a	1	2	3	4	5	6
$a^{p-1} = a^6$	1	64	729	4096	15625	46656
$(a^6 - 1)/7$	0	9	104	585	2232	6665

By Fermat's Little Theorem, given n , if we find a base a with $0 < a < n$ so that $a^{n-1} \pmod{n}$ is not 1 then n is not prime.

- A.8 **EXAMPLE** Let $n = 415692$. If $a = 2$ then $2^{415692} \equiv 58346 \pmod{415693}$ so n is not prime.

So if we are given n and try to show it is not prime with a number of such a 's, and each time find that $a^{n-1} \pmod{n} = 1$, then we may start to think that n is prime after all. Now, there are composite numbers n where $a^{n-1} \equiv 1 \pmod{n}$ but n is not prime. Such a number is a **Fermat liar** or **Fermat pseudoprime** with base a . One is $n = 341 = 11 \cdot 31$ for base $a = 2$, since $2^{340} \equiv 1 \pmod{341}$. However, computer searches suggest that these are very rare.

The rarity of exceptions suggests that we can use a **probabilistic primality test**: given $n \in \mathbb{N}$ to test for primality, pick at random a base a with $0 < a < n$ and calculate whether $a^{n-1} \equiv 1 \pmod{n}$. If it is not true then n is not prime and we

stop testing. But if it is true then we have evidence that n is prime. Now we iterate until we reach the desired level of confidence.

As to the relationship between number of iterations and the confidence level, researchers have shown that if n is not prime then each choice of a has a less than $1/2$ chance of finding that $a^{n-1} \equiv 1 \pmod{n}$. So if n were not prime and we did the test with two different bases a_0, a_1 then there would be a less than $(1/2)^2$ chance of getting both $a_0^{n-1} \equiv 1 \pmod{n}$ and $a_1^{n-1} \equiv 1 \pmod{n}$. Restated, there is at least a $1 - (1/2)^2$ chance that n is prime. In general, after k -many iterations of choosing a base, doing the calculation, and never finding that that n is not prime, then we have a greater than $1 - (1/2)^k$ chance that n is prime.

In summary, if n passes k -many tests for any reasonable-sized k then we are quite confident that it is prime. Our interest in this test is that it is extremely fast; it runs in time $\mathcal{O}(k \cdot (\log n)^2 \cdot \log \log n \cdot \log \log \log n)$. So we can run it lots of times, becoming very confident, in not very much time.

- A.9 EXAMPLE We could test whether $n = 7$ is prime by computing, say, that $3^6 \equiv 1 \pmod{7}$, and $5^6 \equiv 1 \pmod{7}$, and $6^6 \equiv 1 \pmod{7}$. The fact that $n = 7$ does not fail makes us confident it is prime.

The RSA algorithm also uses this offshoot of Fermat's Little Theorem.

- A.10 COROLLARY Let p and q be unequal primes and suppose that a is not divisible by either. Then $a^{(p-1)(q-1)} \equiv 1 \pmod{n}$.

Proof By Fermat, $a^{p-1} \equiv 1 \pmod{p}$ and $a^{q-1} \equiv 1 \pmod{q}$. Raise the first to the $q-1$ power and the second to the $p-1$ power, giving $a^{(p-1)(q-1)} \equiv 1 \pmod{p}$ and $a^{(p-1)(q-1)} \equiv 1 \pmod{q}$. Since $a^{(p-1)(q-1)} - 1$ is divisible by both p and q , it is divisible by their product $pq = n$. \square

Experts think that the most likely attack on RSA encryption is by factoring the modulus n . Anyone who factors n can use the same method as the RSA key setup process to turn the encrypter e into the decrypter d . That's why n is taken to be the product of two large primes; it makes factoring as hard as possible.

There is a factoring algorithm that takes only $\mathcal{O}(b^3)$ time (and $\mathcal{O}(b)$ space), called Shor's algorithm. But it runs only on quantum computers. At this moment there are no such computers built, although there has been progress on that. For the moment, RSA seems safe. (There are schemes that could replace it, if needed.)

V.A Exercises

- ✓ A.11 There are twenty five primes less than or equal to 100. Find them.
- ✓ A.12 We can walk through an RSA calculation. (A) For the primes, take $p = 11$, $q = 13$. Find $n = pq$ and $\varphi(n) = (p-1) \cdot (q-1)$. (B) For the encoder e use the smallest prime $1 < e < \varphi(n)$ that is relatively prime with $\varphi(n)$. (C) Find the decoder d , the multiplicative inverse of e modulo n . (You can use Euclid's algorithm, or just test the candidates.) (D) Take the message to be represented as the number $m = 9$. Encrypt it and decrypt it.

A.13 To test whether a number n is prime, we could just try dividing it by all numbers less than it. (A) Show that we needn't try all numbers less than n , instead we can just try all k with $2 \leq k \leq \sqrt{n}$. (B) Show that we cannot lower that any further than \sqrt{n} . (c) For input $n = 10^{12}$ how many numbers would you need to test? (d) Show that this is a terrible algorithm since it is exponential in the size of the input.

A.14 Show that the probability that a random b -bit number is prime is about $1/b$.

EXTRA

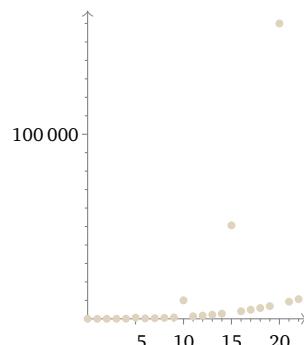
V.B Good-enoughness

A theory shapes the way that you look at the world, at how you see and address what comes before you in practice. For example, Newton's $F = ma$ points is a program for analyzing physical situations: if you see an acceleration then look around for a force. That approach has been fantastically successful, enabling us to build bridges, send people to the moon, etc. Likewise, Darwin's theory tells us that if you see a change in a species then look for a reproductive advantage.

Here we will point out a way in which a naive understanding of Complexity Theory can lead to a misunderstanding of what can be done in practice. Of course, the theorems are right—the proofs check out, the results stand up to formalization, etc. But in learning, we build mental models of what those formal statements mean and there is a common misperception about solving problems that our theory labels “hard.”

Cobham's Thesis identifies the problems having a tractable algorithm with P . However, we have noted that just because a problem is in P does not mean that it has an algorithm that we could use in practice. An example is a problem whose fastest algorithm is $\mathcal{O}(n^{1000})$ and another is a problem whose algorithm has a huge coefficient, such as $2^{1000} \cdot n^2$. The flip side of this is that just because a problem is NP hard does not mean that it is hard to solve on problems that we see in practice. It could be that an algorithm's runtime function takes a while to get big and the first 20 000 instances are quite doable. For another way, consider this function.

$$f(n) = \begin{cases} n^2 & - \text{if } n \text{ is not a multiple of five} \\ n^{\lg n} & - \text{otherwise} \end{cases}$$



Most of the time f grows slowly but occasionally it is super-polynomial. The

exceptions could be rarer than shown, such as every 10, or every 10^{10} , or 10^x . The definition of Big \mathcal{O} is such that as long as there are infinitely many super-polynomial exceptions then the growth of the function as a whole is superpoly.

Suppose that a problem's best algorithm is $\mathcal{O}(f)$. We classify that problem as hard. But if the exception comes once in 10^{10} times then for any single instance the chance of a fast runtime is awfully good.



London pubs, via Google Earth

In short, thinking that NP hard problems are sure to be too slow to solve except for extremely small inputs is an incomplete understanding.

An example of an NP complete problem for which there are available very capable algorithms is the **Traveling Salesman** problem. There are algorithms that can in a reasonable time find solutions for prob-

lem instances with millions of nodes, either giving the optimal solution or, with a high probability, even more quickly finding a path just two or three percent away from the optimal solution. Recently a group of applied mathematicians solved the minimal pub crawl, the shortest route to visit all 24 727 UK pubs. The optimal tour is 45 495 239 meters. The algorithm took 305.2 CPU days, running in parallel on up to 48 cores on Linux servers. That is a lot of computing but it is also a lot of pubs—this is not a toy example.

Another group solved the Traveling Salesman instance of visiting all 24 978 cities in Sweden, giving a tour of about 72 500 kilometers. The approach was to find a nearly-best solution and then use that to find the best one. The final stages, that improved the lower bound by 0.000 023 percent, required eight years of computation time running in parallel on a network of Linux workstations.

There are a number of systems for solving the Traveling Salesman problem that are widely available. An example is that the Free mathematics system **Sage** includes one. Here is a brief example. It uses a graph that is sure to have a solution, and then we display the solution as an adjacency matrix. For more, see the documentation.



Sweden tour

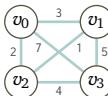
```
sage: g = graphs.HeawoodGraph()
sage: tsp = g.traveling_salesman_problem()
sage: tsp.adjacency_matrix()
[0 0 0 0 1 0 0 0 0 0 0 1]
[0 0 1 0 0 0 0 0 0 1 0 0 0]
[0 1 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 1 0]
[0 0 0 1 0 0 0 0 0 1 0 0 0]
[1 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 1 0]
[0 0 1 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 1]
[0 0 0 0 1 0 0 0 0 0 1 0 0]
[0 1 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 1 0]
```

In summary, even though the theory says that a problem is hard does not mean

it that it cannot be attacked for non-toy instances. See also the next Extra section.

V.B Exercises

- B.1 Critique this from social media: “The Traveling Salesman problem is *NP* hard. That means that algorithms exist that solve the problem but these algorithms are very slow. So for an input of size 100 you may already have to wait hundreds of years.”
- B.2 A naive algorithm for the Traveling Salesman problem is to try every possible circuit. If there are n -many cities then how many circuits are there?
- B.3 Use the exhaustive search of all possible circuits to find the shortest Travelling Salesman Problem solution for a circuit involving the graph below.



EXTRA

V.C SAT solvers

The prior Extra section gives an example of a problem where the best algorithm we know is super-polynomial, but in practice that problem is solvable. For instance, problems may have occasional hard instances—black holes where the computer goes in and does not come out—but on most instances we can find the answer in a reasonable time. This section demonstrates a program to solve **SAT**, a **SAT solver**, and shows how to use that as an oracle to solve others.

A problem reduction $\mathcal{L}_1 \leq_p \mathcal{L}_0$ gives a way to transfer problem domains, to change questions about \mathcal{L}_1 into questions about \mathcal{L}_0 . Here we take \mathcal{L}_0 to be **SAT** and for \mathcal{L}_1 we will use Sudoku.

	9		1	5
5		4	7	
4	7	3	5	6
		5	6	1
			9	
			7	4
				9
			4	
			8	6
				8
			4	3
			8	
				5
1	3	5	9	
		6	2	5
			5	7
				3
7	2		1	
				9

In case it is unfamiliar, the popular Sudoku puzzle starts with a 9-by-9 array, with some of the cells already filled in. An example is above. Players solve it by filling in the blanks while satisfying three restrictions: every row must contain each of the numbers 1–9, and the same holds for each column, as well as the nine subsquares.

We first argue that this is hard. The definition of the computational complexity of a problem requires that we describe how a solution algorithm's use of some resource grows with the problem's input size. For that, we must frame the problem to allow instances of different input sizes. The natural way to generalize the puzzle is: instead of eighty one variables $x_{1,1}, \dots x_{9,9}$ we could use an arbitrary number, $x_1, \dots x_n$. Instead of those variables taking on 1–9, they could take on a value in $1-k$ (and we could call these ‘colors’ instead of ‘values’). And, in place of rows, columns, and subsquares that are driven by the geometry, a problem instance could have arbitrary sets, $S \subseteq \{x_1, \dots x_n\}$ of size k . With that, it is more than a fixed-sized puzzle, it is a problem, and we can show that the Soduku problem is NP complete. Thus we can fairly consider this problem to be quite hard.

However, having noted this, in this section we will ignore the generalization and limit our attention to the traditional 9×9 board.

To solve puzzle instances using the SAT solver as an oracle we focus on the reduction $\text{Soduku} \leq_p \text{SAT}$. We must produce a function that inputs game boards and outputs Propositional Logic expressions. Recall that expressions such as $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$ or $(x_1 \vee x_2 \vee \neg x_3) \wedge x_4 \wedge (\neg x_2 \vee x_3)$ are in Conjunctive Normal form, CNF. These are the conjunction of clauses where each clause is the disjunction of literals (a literal is either a single atom such as x_i or the negation of an atom such as $\neg x_i$). The SAT solver can require that input be in this form because for any Boolean function there is a CNF expression giving that behavior; more is in Appendix C.

The SAT solver that we use needs its input file formatted to a standard called DIMACS. It starts with comment lines, beginning with the character c. Next comes a problem line, which starts with a p, followed by a space and the problem type cnf, then followed by a space and the number of variables, and then followed by a space and the number of clauses. After that line the rest of the file consists of the clause descriptions.

To describe a clause the file lists its indices. Thus we describe $x_2 \vee x_5 \vee x_6$ with the list 2 5 6. Negations are described with negatives, so that $\neg x_5 \vee x_7 \vee \neg x_9$ matches -5 7 -9. These clause descriptions are separated by 0's.[†]

We will have many variables. For instance, for the Soduku array's row 1, column 1 entry, we will have a Boolean variable $x_{1,1,1}$, another variable $x_{1,1,2}$, etc., up to $x_{1,1,9}$. Only one of these nine will be T and the rest will be F. The variable $x_{1,1,v}$ is T if in our solution the number in row 1 and column 1 is v . Otherwise this variable is F. Restated, if for instance in the first row and first column the puzzle has the value 5 then $x_{1,1,5}$ is T while all other $x_{1,1,i}$ are F.

Thus for each row, column, and value triple $r, c, v \in \{1, \dots 9\}$ we will have a variable $x_{r,c,v}$. That's $9^3 = 729$ variables.

The CNF expression that we will produce has clauses of two kinds. One describes the general rules of the game Soduku while the other is specific to the

[†] Since DIMACS uses 0 to separate clauses, in this section we don't follow our usual practice of starting with the first variable named x_0 . Rather, we start with x_1 .

particular starting board instance. It is as though we bought a puzzle book and opened first to the introduction describing the rules, and later opened to the page containing the specific partial board.

To describe the general rules we need lot of clauses describing relationships among the variables. An example of such a rule is that exactly one of $x_{1,1,1}$, $x_{1,1,2}$, ... $x_{1,1,9}$ is T . There are too many of these rules to write by hand so we will get the computer to do them.

The Racket file starts with some constants that make the code easier to read.

```
(define ONETONINE '(1 2 3 4 5 6 7 8 9))
(define ONETOTHREE '(1 2 3)) ; for boxes
(define FOURTOSIX '(4 5 6))
(define SEVENTONINE '(7 8 9))
(define BOX-INDICES (list ONETOTHREE FOURTOSIX SEVENTONINE))
```

Each row has nine restrictions. For instance, to express that the first row contains an entry with the value 2, we need this clause.

$$x_{1,1,2} \vee x_{1,2,2} \vee x_{1,3,2} \vee x_{1,4,2} \vee x_{1,5,2} \vee x_{1,6,2} \vee x_{1,7,2} \vee x_{1,8,2} \vee x_{1,9,2}$$

The Racket code below produces this corresponding list: ((1 1 2) (1 2 2) (1 3 2) (1 4 2) (1 5 2) (1 6 2) (1 7 2) (1 8 2) (1 9 2)). We express all of the row restrictions with one such list for each row and value.

```
;; row-restrictions  Return list of list of triples, each list of triples meaning
;;      that each row has to have each value 1-9.
(define (row-restrictions)
  (define (one-row-one-value row-number variable-value)
    (for/list ([column-number ONETONINE])
      (list row-number column-number variable-value)))

  (for*/fold ([accumulator '()])
            #:result (reverse accumulator)
            ([variable-value ONETONINE]
             [row-number ONETONINE])
    (cons (one-row-one-value row-number variable-value) accumulator)))
```

Running that routine produces a list of lists. This is a typical member, requiring that at least one entry in row 3 must be an 8.

```
((3 1 8) (3 2 8) (3 3 8) (3 4 8) (3 5 8) (3 6 8) (3 7 8) (3 8 8) (3 9 8))
```

The column restrictions are much like the row ones.

```
;; column-restrictions  Return list of list of triples, each list of triples meaning
;;      that each row has to have each value 1-9.
(define (column-restrictions)
  (define (one-column-one-value column-number variable-value)
    (for/list ([row-number ONETONINE])
      (list row-number column-number variable-value)))

  (for*/fold ([accumulator '()])
            #:result (reverse accumulator)
            ([variable-value ONETONINE]
             [column-number ONETONINE])
    (cons (one-column-one-value column-number variable-value) accumulator)))
```

Here is a typical line, requiring that at least one entry in column 7 must be an 8.

```
((1 7 8) (2 7 8) (3 7 8) (4 7 8) (5 7 8) (6 7 8) (7 7 8) (8 7 8) (9 7 8))
```

For the subsquares, the restrictions are the same in principle but the form of the code is a bit different.

```
;; box-restrictions  Return list of list of triples, each list of triples meaning
;;   that each 3x3 box has to have each value 1-9.
(define (box-restrictions)
  (define (one-box-one-value box-row-list box-column-list variable-value)
    (for*/list ([row-number box-row-list]
               [column-number box-column-list])
              (list row-number column-number variable-value)))

  (for*/fold ([accumulator '()])
            #:result (reverse accumulator))
            ([variable-value ONETONINE]
             [box-row-list BOX-INDICES]
             [box-column-list BOX-INDICES])
            (cons (one-box-one-value box-row-list box-column-list variable-value) accumulator)))
```

Here is one of the box restrictions produced by the Racket code below, saying that some entry in the upper-right box has the value 8.

```
((7 1 8) (7 2 8) (7 3 8) (8 1 8) (8 2 8) (8 3 8) (9 1 8) (9 2 8) (9 3 8))
```

Running the **SAT** solver with just the restrictions above finds that they can be satisfied. But there is a surprise. It finds a satisfying assignment by putting more than one value in some boxes and no value at all in some other boxes.

Thus we need one more set of restrictions, that no entry can contain two values. We add clauses like $\neg x_{3,4,1} \vee \neg x_{3,4,2}$, meaning that the row 3 and column 4 entry cannot be both a 1 and a 2 (it could of course be neither).

```
;; entry-restrictions  Return list of list of triples, each list of triples meaning
;;   that each entry cannot be two separate values 1-9.
(define (entry-restrictions)
  (for*/list ([row-number ONETONINE]
             [column-number ONETONINE]
             [variable-value1 ONETONINE]
             [variable-value2 ONETONINE]
             #:unless (>= variable-value1 variable-value2))
            (list (list row-number column-number (* -1 variable-value1)))
                  (list row-number column-number (* -1 variable-value2))))
```

This is one of the resulting lines, enforcing that the entry in row 8 and column 8 cannot be both 6 and 9 (again, the negative means logical negation).

```
((8 8 -6) (8 8 -9))
```

To finish we must specify the initial board. For example, to tell the **SAT** solver that there is a 9 in row 1 and column 3 we include the one-literal clause $x_{1,3,9}$. Here are the first few lines of that routine.

```
;; INITIAL-CLAUSES The given layout of the board. Each row is a list with a
;; triple: row number, column number, integer.
(define INITIAL-CLAUSES
  (list (list '(1 3 9)) ; there is a 9 in position (1,3)
        (list '(1 8 1))
        (list '(1 9 5)))
```

In this development, and in the Racket file, we work in $x_{r,c,v}$'s. But DIMACS wants a single index. So we need to convert each of our variables to some x_k and back again. The formula is $k = 1 + 81 \cdot (|v| - 1) + 9 \cdot (r - 1) + (c - 1)$.

```
;; triple->varnum Find the variable number associated with the row, column, and value
;; row-number column-number integers, counting starts at 1
;; variable-value integer value of the entry. If negative, then
;; the predicate is to be negated.
;; If variable-value < 0 then use the absolute value for the basic varnum,
;; but return the negative of the polynomial (indicating that the predicate is negated).
(define (triple->varnum row-number column-number variable-value)
  (let ([a-value (+ (* 81 (- (abs variable-value) 1))
                  (* 9 (- row-number 1))
                  (- column-number 1))
         1)]) ; add 1 because DIMACS uses 0 to terminate clauses
  (if (negative? variable-value)
      (* -1 a-value)
      a-value)))

;; varnum->triple From the variable number, return the associated row, column, and value
(define (varnum->triple v)
  (let* ([offset (- (abs v) 1)]
     [variable-value (quotient offset 81)]
     [vv-removed (- offset (* 81 variable-value))]
     [row-number (quotient vv-removed 9)]
     [column-number (remainder vv-removed 9)])
  (if (negative? v)
      (list (+ 1 row-number) (+ 1 column-number) (* -1 (+ 1 variable-value)))
      (list (+ 1 row-number) (+ 1 column-number) (+ 1 variable-value))))
```

Here are a couple of example conversions using these routines.

```
> (triple->varnum 3 4 5)
346
> (varnum->triple 346)
'(3 4 5)
```

Now we can write all of this to the DIMACS-format file. This will convert the clauses.

```
;; produce-clauses Given a list of lists of triples, produce the matching set of strings
;; for the DIMACS file
(define (produce-clauses list-of-lists)
  (define (one-line-of-one variable-in-clause) ; produce line from a list of one number
    (apply format "~a\n" variable-in-clause))
  (define (one-line-of-two variables-in-clause) ; produce line from a list of two numbers
    (apply format "a ~a 0\n" variables-in-clause))
  (define (one-line-of-nine variables-in-clause) ; produce line from a list of nine numbers
    (apply format "a ~a ~a ~a ~a ~a ~a ~a ~a 0\n" variables-in-clause))

  (for/list ([clause-list list-of-lists])
    (display clause-list)(newline)
    (cond [= 9 (length clause-list))
```

```
(one-line-of-nine (map (lambda (x) (triple->varnum (first x) (second x) (third x)))
                      clause-list)))
[ (= 1 (length clause-list))
  (one-line-of-one (map (lambda (x) (triple->varnum (first x) (second x) (third x)))
                        clause-list))]
[ (= 2 (length clause-list))
  (one-line-of-two (map (lambda (x) (triple->varnum (first x) (second x) (third x)))
                        clause-list)))]))
```

And this gathers the clauses together and then calls the above routine.

```
;; CLAUSES  The list of all clauses, including those auto generated.
(define CLAUSES
  (append INITIAL-CLAUSES (entry-restrictions)
          (row-restrictions) (column-restrictions) (box-restrictions)))

(define FILE-PREAMBLE
  (list (format "c ~a\n" FILENAME)
        "c DIMACS format file for SAT solver\n"
        (format "c ~a Jim Hefferon, hefferon.net. Public Domain.\n"
                (date->string (current-date)))
        (format "p cnf ~a ~a\n" (* 9 9 9) (length CLAUSES)))))

(define FILE-LINES
  (append
    FILE-PREAMBLE
    (produce-clauses CLAUSES)))

;; dump-to-file  Drop the clauses to the file
(define (dump-to-file)
  (define (dump-lines outfile)
    (for ([ln FILE-LINES])
      (display ln outfile)))

  (call-with-output-file* FILENAME dump-lines #:mode 'text #:exists 'replace))
```

The first few lines of the result `sudoku.cnf` look like this.

```
c sudoku.cnf
c DIMACS format file for SAT solver
c 2022-01-16 Jim Hefferon, hefferon.net. Public Domain.
p cnf 729 3197
651 0
8 0
333 0
334 0
256 0
663 0
502 0
262 0
506 0
183 0
```

Remember that the 0's separate clauses.

We can now run the **SAT** solver, MiniSat (Eén and Sörensson 2005).

```
ftpmain@millstone:~/Documents/computing/src/scheme/complexity$ minisat sudoku.cnf sudoku.out
WARNING: for repeatability, setting FPU to use double precision
===== [ Problem Statistics ] =====
```

```

| Number of variables:          729
| Number of clauses:           2044
| Parse time:                  0.00 s
| Eliminated clauses:          0.00 Mb
| Simplification time:         0.00 s
|
===== [ Search Statistics ] =====
| Conflicts |      ORIGINAL      |          LEARNED      | Progress |
|           | Vars Clauses Literals | Limit Clauses Lit/Cl | | | | | |
|---|---|---|---|---|---|---|---|
|    100 |     257    1209    4354 |     443      99    7 | 47.051 % |
|   250 |     256    1209    4354 |     487     248    8 | 47.188 % |
|   475 |     247    1190    4283 |     536     464    8 | 48.423 % |
|-----|-----|-----|-----|-----|-----|-----|-----|
restarts          : 6
conflicts        : 645          (88660 /sec)
decisions        : 1326          (0.00 % random) (182268 /sec)
propagations     : 15204          (2089897 /sec)
conflict literals : 4922          (18.52 % deleted)
Memory used      : 29.00 MB
CPU time         : 0.007275 s
SATISFIABLE

```

It took far less than a second, on an ordinary laptop. The algorithms for **SAT** solvers are exponential in the worst case but they seem to do very well in practice.

Below is the output. It contains 729 numbers but only a few fit on this page, and anyway that many numbers would not be more enlightening than just showing the first few.

```
ftpmaint@millstone:~/Documents/computing/src/scheme/complexity$ cat soduku.out
SAT
-1 -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
```

To see a solution entry, pick a positive number from the output.

```
> (varnum->triple 8)
'(1 8 1)
```

Here is a positive output number that doesn't show in the above line.

```
> (varnum->triple 107)
'(3 8 2)
```

Thus, row 3 and column 8 holds 2.

Some simple routines built on what is above show the entire board.

```
> (show-solved-board)
'#((2 6 9 3 7 8 4 1 5)
 #(5 8 1 4 2 9 7 6 3)
 #(4 7 3 5 6 1 9 2 8)
 #(8 1 2 7 4 5 3 9 6)
 #(3 5 7 1 9 6 2 8 4)
 #(6 9 4 8 3 2 1 5 7)
 #(1 3 5 9 8 4 6 7 2)
 #(9 4 6 2 5 7 8 3 1)
 #(7 2 8 6 1 3 5 4 9))
```

In summary, we can in reasonable time solve instances of **SAT** that are not toy exercises. They are large enough that to write the CNF form we had to resort to code.

V.C Exercises

C.1 This board is described online as an especially hard Sudoku. Use the routines from this section to solve it.

3	4	5	6	9
5	4	8		1
		8 2 3		7
1	8	7		3
		9		
8	7		8	7 2
4	9			

Part Four
Appendices

APPENDIX A STRINGS

An **alphabet** is a nonempty and finite set of **symbols** (sometimes called **tokens**). We write symbols in a distinct typeface, as in `1` or `a`, because the alternative of quoting them would be clunky.[†] A **string** or **word** over an alphabet is a finite sequence of elements from that alphabet. The string with no elements is the **empty string**, denoted ϵ , along with <https://www.youtube.com/watch?v=juXwu0Nqc3I>.

One potentially surprising aspect of a symbol is that it may contain more than one letter. For instance, a programming language may have `if` as a symbol, meaning that it is indecomposable into separate letters. Another example is that the Racket alphabet contains the symbols `or` and `car`, as well as allowing variable names such as `x`, or `lastname`. An example of a string is `(or a ready)`, which is a sequence of five alphabet elements, $\langle(), \text{or}, \text{a}, \text{ready}, \rangle$.

Traditionally, we denote an alphabet with the Greek letter Σ . In this book we will name strings with lower case Greek letters (except that we use ϕ for something else) and denote the items in the string with the associated lower case roman letter, as in $\sigma = \langle s_0, \dots, s_{n-1} \rangle$ and $\tau = \langle t_0, \dots, t_{m-1} \rangle$. The **length** of the string σ , $|\sigma|$, is the number of symbols that it contains, n . In particular, the length of the empty string is $|\epsilon| = 0$.

In place of s_i we sometimes write $\sigma[i]$. One convenience of this form is that we use $\sigma[-1]$ for the final character, $\sigma[-2]$ for the one before it, etc. We also write $\sigma[i:j]$ for the substring between terms i and j , including the i -th term but not the j -th, and we write $\sigma[i:]$ for the tail substring that starts with term i as well as $\sigma[:j]$ for $\sigma[0:j]$.

The notations such as diamond brackets and commas are ungainly. We usually work with alphabets having single-character symbols and then we write strings by omitting the brackets and commas. That is, we write $\sigma = abc$ instead of $\langle a, b, c \rangle$.[‡] This convenience comes with the disadvantage that without the diamond brackets the empty string is just nothing, which is why we use the separate symbol ϵ .[#]

The alphabet consisting of the bit characters is $\mathbb{B} = \{0, 1\}$. Strings over this alphabet are **bitstrings** or **bit strings**.[§]

Where Σ is an alphabet, for $k \in \mathbb{N}$ the set of length k strings over that alphabet is Σ^k . The set of strings over Σ of any finite length is $\Sigma^* = \cup_{k \in \mathbb{N}} \Sigma^k$. The asterisk symbol is the **Kleene star**, read aloud as “star.”

Strings are simple so there are only a few operations. Let $\sigma = \langle s_0 \dots s_{n-1} \rangle$ and $\tau = \langle t_0, \dots, t_{m-1} \rangle$ be strings over an alphabet Σ . The **concatenation** $\sigma \cdot \tau$ or $\sigma \tau$ appends the second string to the first, $\sigma \cdot \tau = \langle s_0 \dots s_{n-1}, t_0, \dots, t_{m-1} \rangle$. Where

[†]We give them a distinct look to distinguish the symbol ‘`a`’ from the variable ‘`a`’, so that we can tell “let `x = a`” apart from “let `x = a`.” Symbols are not variables—they don’t hold a value, they are themselves a value.

[‡]To see why when we drop the commas we want the alphabet to consist of single-character symbols, consider $\Sigma = \{a, aa\}$ and the string `aaa`. Without the commas this string is ambiguous: it could mean $\langle a, aa \rangle$, or $\langle aa, a \rangle$, or $\langle a, a, a \rangle$.

[#]Omitting the diamond brackets and commas also blurs the distinction between a symbol and a one-symbol string, between `a` and $\langle a \rangle$. However, dropping the brackets it is so convenient that we accept this disadvantage.

[§]Some authors consider infinite bitstrings but ours will always be finite.

$\sigma = \tau_0 \hat{\cdot} \dots \hat{\cdot} \tau_{k-1}$, we say that σ **decomposes** into the τ 's, and that each τ_i is a **substring** of σ . The first substring, τ_0 , is a **prefix** of σ . The last, τ_{k-1} , is a **suffix**.

A **power** or **replication** of a string is an iterated concatenation with itself, so that $\sigma^2 = \sigma \hat{\cdot} \sigma$ and $\sigma^3 = \sigma \hat{\cdot} \sigma \hat{\cdot} \sigma$, etc. We write $\sigma^1 = \sigma$ and $\sigma^0 = \varepsilon$. The **reversal** σ^R of a string takes the symbols in reverse order: $\sigma^R = \langle s_{n-1}, \dots, s_0 \rangle$. The empty string's reversal is $\varepsilon^R = \varepsilon$.

For example, let $\Sigma = \{a, b, c\}$ and let $\sigma = abc$ and $\tau = bbaac$. Then the concatenation $\sigma\tau$ is $abcbbaac$. The third power σ^3 is $abcabcabc$, and the reversal τ^R is $caabb$. A **palindrome** is a string that equals its own reversal. Examples are $\alpha = abba$, $\beta = cdc$, and ε .

Exercises

- A.1 Let $\sigma = 10110$ and $\tau = 110111$ be bit strings. Find each. (A) $\sigma \hat{\cdot} \tau$ (B) $\sigma \hat{\cdot} \tau \hat{\cdot} \sigma$ (C) σ^R (D) σ^3 (E) $\theta^3 \hat{\cdot} \sigma$
- A.2 Let the alphabet be $\Sigma = \{a, b, c\}$. Suppose that $\sigma = ab$ and $\tau = bca$. Find each. (A) $\sigma \hat{\cdot} \tau$ (B) $\sigma^2 \hat{\cdot} \tau^2$ (C) $\sigma^R \hat{\cdot} \tau^R$ (D) σ^3
- A.3 Let $\mathcal{L} = \{\sigma \in \mathbb{B}^* \mid |\sigma| = 4 \text{ and } \sigma \text{ starts with } 0\}$. How many elements are in that language?
- A.4 Suppose that $\Sigma = \{a, b, c\}$ and that $\sigma = abcccbba$. (A) Is $abcb$ a prefix of σ ? (B) Is ba a suffix? (C) Is bab a substring? (D) Is ε a suffix?
- A.5 What is the relation between $|\sigma|$, $|\tau|$, and $|\sigma \hat{\cdot} \tau|$? You must justify your answer.
- A.6 The operation of string concatenation follows a simple algebra. For each of these, decide if it is true. If so, prove it. If not, give a counterexample. (A) $\alpha \hat{\cdot} \varepsilon = \alpha$ and $\varepsilon \hat{\cdot} \alpha = \alpha$ (B) $\alpha \hat{\cdot} \beta = \beta \hat{\cdot} \alpha$ (C) $\alpha \hat{\cdot} \beta^R = \beta^R \hat{\cdot} \alpha^R$ (D) $\alpha^{RR} = \alpha$ (E) $\alpha^{iR} = \alpha^i$
- A.7 Show that string concatenation is not commutative, that there are strings σ and τ so that $\sigma \hat{\cdot} \tau \neq \tau \hat{\cdot} \sigma$.
- A.8 In defining decomposition above we have ' $\sigma = \tau_0 \hat{\cdot} \dots \hat{\cdot} \tau_{n-1}$ ', without parentheses on the right side. This takes for granted that the concatenation operation is associative, that no matter how we parenthesize it we get the same string. Prove this. Hint: use induction on the number of substrings, n .
- A.9 Prove that this constructive definition of string power is equivalent to the one above.

$$\sigma^n = \begin{cases} \varepsilon & - \text{if } n = 0 \\ \sigma^{n-1} \hat{\cdot} \sigma & - \text{if } n > 0 \end{cases}$$

APPENDIX B FUNCTIONS

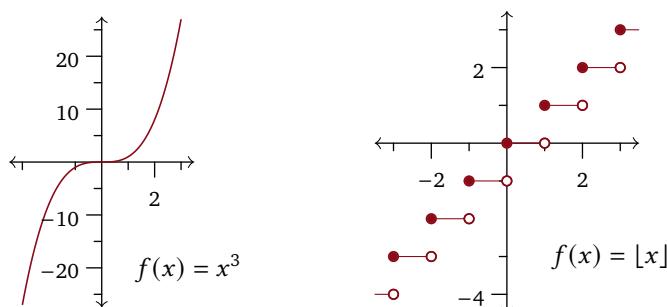
A function is an input-output relationship: each input is associated with a unique output. An example is the association of each input natural number with the output number that is its square. Another is the association of each string of characters with the length of that string. A third is the association of each polynomial $a_nx^n + \dots + a_1x + a_0$ with a Boolean value T or F , depending on whether 1 is a root of that polynomial.

An important point is that, contrary to what is said in most introductions, a function isn't a 'rule'. The function that associates a year with that year's winners of the US baseball World Series isn't given by any rule simpler than an exhaustive listing of all cases. Nor is the kind of association that a database might have, such as linking the government ID of US citizens to their income in the most recent tax year. True, many functions in science are described by a formula, such as $E(m) = mc^2$, and as well many functions are computed by a program. But what makes something a function is that for each input there is exactly one associated output. If we can go from the inputs to the outputs with a calculation then that's great but that is not the point.

For a precise definition fix two sets, a **domain** D and a **codomain** C . A **function** or **map**, $f: D \rightarrow C$, is a set of pairs $(x, y) \in D \times C$, subject to the restriction of being **well-defined**, that every $x \in D$ appears as the first entry in one and only one pair (more on well-definedness is below). We write $f(x) = y$ or $x \mapsto y$ and say that 'x maps to y'. Note the difference between the arrow symbols used in $f: D \rightarrow C$ and $x \mapsto y$. We say that x is an **input** or **argument** to the function, and that y is an **output** or **value** of the function.

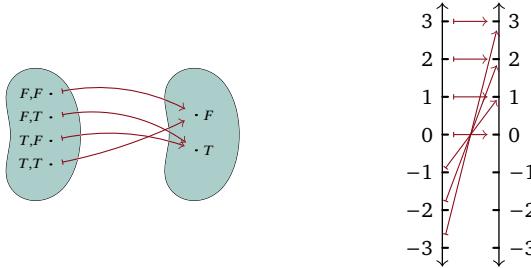
Some functions take more than one input, such as $\text{dist}(x, y) = \sqrt{x^2 + y^2}$. We say that this function is 2-ary, while some other functions are 3-ary, etc. The number of inputs is the function's **arity**. If the function takes only one input but that input is a tuple then we often drop the parentheses, so we write $f(3, 5)$ instead of $f((3, 5))$.

Pictures We often illustrate functions using the familiar xy axes.



We also illustrate functions with a bean diagram, which separates the domain and the codomain sets. Below on the left is the action of the exclusive or operator while

on the right is a variant, showing the absolute value function mapping integers to integers.



Codomain and range Where $S \subseteq D$ is a subset of the domain, its **image** is the set $f(S) = \{f(s) \mid s \in S\}$. Thus, under the squaring function the image of $S = \{0, 1, 2\}$ is $f(S) = \{0, 1, 4\}$. Under the floor function $g: \mathbb{R} \rightarrow \mathbb{R}$ given by $g(x) = \lfloor x \rfloor$, the image of the positive reals is the set of natural numbers.

The image of the entire domain D is the function's **range**, $\text{ran}(f) = f(D) = \{f(d) \mid d \in D\}$. For instance, the range of the function $f: \mathbb{Z} \rightarrow \mathbb{Z}$ given by $x \mapsto 2x$ is the set of even integers.

Note the difference between a function's range and its codomain; the codomain is a convenient superset. An example is that for the function given by $f(x) = \sqrt{2x^4 + 2x^2 + 15}$, we will usually be content to note that the polynomial is always nonnegative and so the output is real, writing $f: \mathbb{R} \rightarrow \mathbb{R}$ where the second \mathbb{R} is the codomain, rather than troubling to find its exact range.

Domain Sometimes a function's domain requires attention. Examples are that $f(x) = 1/x$ is undefined at $x = 0$ and that the function defined by the infinite series $g(r) = 1 + r + r^2 + \dots$ diverges when r is outside the interval $(-1..1)$. Formally, when we define the function we must specify the domain to eliminate such problems, for instance by defining the domain of f as $\mathbb{R} - \{0\}$. However, we are usually casual about this, expecting that a reader will understand to omit inputs that are an issue.

We sometimes have a function $f: D \rightarrow C$ and want to reduce the domain to some subset $S \subseteq D$. The **restriction** $f \upharpoonright_S$ is the function with domain S and codomain C defined by $f \upharpoonright_S(x) = f(x)$.

In the Theory of Computation We sometimes use these terms in a way that conflicts with their traditional definition. When we study a function often we will fix a convenient set of inputs D , such as the set of all strings Σ^* or all natural numbers \mathbb{N} , and refer to D as the function's domain although that function may be undefined on some of D 's elements. In this case we say that f is a **partial function**. If f is defined on all inputs then it is a **total function**. Strictly speaking, 'partial' is redundant since any function is partial, including a total function, but often 'partial' connotes that f is not defined on some $d \in D$.

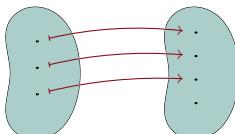
Well-defined The definition of a function contains the condition that for each domain element there cannot be two associated codomain elements. We say that functions are **well-defined**.

When we are considering a relationship between x 's and y 's and asking if it is a function, typically it is well-definedness that is at issue.[†] For instance, consider the set of ordered pairs (x, y) where $y^2 = x$. If $x = 9$ then both $y = 3$ and $y = -3$ are related to x so this is not a functional relationship—it is not well-defined—because $x = 9$ does not have only one associated y . Another example is that when setting up a company's email we may decide to use each person's first initial and last name, but there could be more than one, say, lwainwright, making the relationship email \mapsto person be not well-defined.

For a function $f : \mathbb{R} \rightarrow \mathbb{R}$ that is suitable for graphing on xy axes, visual proof of well-definedness is that for any x in the domain, the vertical line through x intercepts f 's graph in exactly one point.

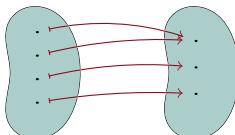
One-to-one and onto The definition of function has an asymmetry: among the ordered pairs (x, y) , it requires that each domain element x be in one pair and only one pair, but it does not require the same of the codomain elements.

A function is **one-to-one** (or **1-1** or an **injection**) if each codomain element y is in at most one pair. The function below is one-to-one because for every element y in the codomain, the bean on the right, there is at most one arrow ending at y .



The most common way to prove that a function f is one-to-one is to assume that $f(x_0) = f(x_1)$ and then argue that therefore $x_0 = x_1$. If a function is suitable for graphing on xy axes then visual proof that it is one-to-one is that for any y in the codomain, the horizontal line at y intersects the graph in at most one point.

A function is **onto** (or a **surjection**) if each codomain element y is in at least one pair. Thus, a function is onto if its codomain equals its range. The function below is onto because every element in the codomain bean has at least one arrow ending at it.



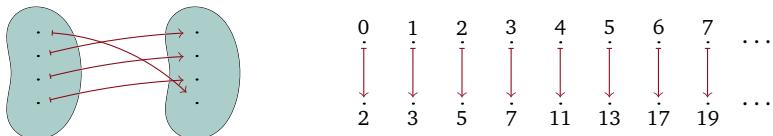
The most common way to verify that a function is onto is to start with a generic (that is, arbitrary) codomain element y and then exhibit a domain element x that

[†]Sometimes people say that they are, “checking that the function is well-defined.” In a strict sense this is confused, because if it is a function then it is by definition well-defined. However, while all tigers have stripes, we do sometimes say “striped tiger.” Natural language is funny that way.

maps to it. If a function is suitable for graphing on xy axes then visual proof that it is onto is that for any y in the codomain, the horizontal line at y intercepts the graph in at least one point.

As the above pictures suggest, where the domain and codomain are finite, when there is a function $f: D \rightarrow C$ then we can conclude that the number of elements in the domain is less than or equal to the number in the codomain. Further, if the function is onto then the number of elements in the domain equals the number in the codomain if and only if the function is one-to-one.

Correspondence A function is a **correspondence** (or **bijection**) if it is both one-to-one and onto. The picture on the left shows a correspondence between two finite sets, both with four elements, and the picture on the right shows a correspondence between the natural numbers and the primes.



The most common way to verify that a function is a correspondence is to separately verify that it is one-to-one and that it is onto. Where the function is $f: \mathbb{R} \rightarrow \mathbb{R}$, so it can be graphed on xy axes, visual proof that it is a correspondence is that for any y in the codomain, the horizontal line at y intercepts the graph in exactly one point.

As the picture above on the left suggests, where the domain and codomain are finite, if a function is a correspondence then its domain has the same number of elements as its codomain.

Composition and inverse If $f: D \rightarrow C$ and $g: C \rightarrow B$ then their **composition** $g \circ f: D \rightarrow B$ is defined by $g \circ f(d) = g(f(d))$. For instance, the real functions $f(x) = x^2$ and $g(x) = \sin(x)$ combine to give $g \circ f = \sin(x^2)$.

Composition does not commute. Using the functions from the prior paragraph, $f \circ g = \sin(x^2)$ and $f \circ g = (\sin x)^2$ are different; for instance they are unequal when $x = \pi$. Composition can fail to commute more dramatically: if $f: \mathbb{R}^2 \rightarrow \mathbb{R}$ is given by $f(x_0, x_1) = x_0$, and $g: \mathbb{R} \rightarrow \mathbb{R}$ is $g(x) = x$, then $g \circ f(x_0, x_1) = x_0$ is perfectly sensible but composition in the other order is not even defined.

The composition of one-to-one functions is one-to-one, and the composition of onto functions is onto; see the exercises. It follows then that the composition of correspondences is a correspondence.

An **identity function** $\text{id}: D \rightarrow D$ is given by $\text{id}(d) = d$ for all $d \in D$. It acts as the identity element in function composition, so that if $f: D \rightarrow C$ then $f \circ \text{id} = f$ and if $g: C \rightarrow D$ then $\text{id} \circ g = g$. As well, if $h: D \rightarrow D$ then $h \circ \text{id} = \text{id} \circ h = h$.

Given $f: D \rightarrow C$, if $g \circ f$ is the identity function then g is a **left inverse** function of f , or what is the same thing, f is a **right inverse** of g . If g is both a left and right inverse of f then we simply say that it is an **inverse** (or **two-sided inverse**) of f

and denoted it as f^{-1} . If a function has an inverse then that inverse is unique. A function has a two-sided inverse if and only if it is a correspondence.

Exercises

B.1 Let $f, g: \mathbb{R} \rightarrow \mathbb{R}$ be $f(x) = 3x + 1$ and $g(x) = x^2 + 1$. (A) Show that f is one-to-one and onto. (B) Show that g is not one-to-one and not onto.

B.2 Show each of these.

(A) Let $g: \mathbb{R}^3 \rightarrow \mathbb{R}^2$ be the projection map $(x, y, z) \mapsto (x, y)$ and let $f: \mathbb{R}^2 \rightarrow \mathbb{R}^3$ be $(x, y) \mapsto (x, y, 0)$. Then g is a left inverse of f but not a right inverse.

(B) The function $f: \mathbb{Z} \rightarrow \mathbb{Z}$ given by $f(n) = n^2$ has no left inverse.

(C) Where $D = \{0, 1, 2, 3\}$ and $C = \{10, 11\}$, the function $f: D \rightarrow C$ given by $0 \mapsto 10, 1 \mapsto 11, 2 \mapsto 10, 3 \mapsto 11$ has more than one right inverse.

B.3 (A) Where $f: \mathbb{Z} \rightarrow \mathbb{Z}$ is $f(a) = a + 3$ and $g: \mathbb{Z} \rightarrow \mathbb{Z}$ is $g(a) = a - 3$, show that g is inverse to f .

(B) Where $h: \mathbb{Z} \rightarrow \mathbb{Z}$ is the function that returns $n + 1$ if n is even and returns $n - 1$ if n is odd, find a function inverse to h .

(C) If $s: \mathbb{R}^+ \rightarrow \mathbb{R}^+$ is $s(x) = x^2$, find its inverse.

B.4 Fix $D = \{0, 1, 2\}$ and $C = \{10, 11, 12\}$. Let $f, g: D \rightarrow C$ be $f(0) = 10, f(1) = 11, f(2) = 12$, and $g(0) = 10, g(1) = 10, g(2) = 12$. Then: (A) verify that f is a correspondence (B) construct an inverse for f (C) verify that g is not a correspondence (D) show that g has no inverse.

B.5 (A) Prove that a composition of one-to-one functions is one-to-one. (B) Prove that a composition of onto functions is onto. With the prior item, this gives that a composition of correspondences is a correspondence. (C) Prove that if $g \circ f$ is one-to-one then f is one-to-one. (D) Prove that if $g \circ f$ is onto then g is onto. (E) If $g \circ f$ is onto, must f be onto? If it is one-to-one, must g be one-to-one?

B.6 Prove.

(A) A function f has an inverse if and only if f is a correspondence.

(B) If a function has an inverse then that inverse is unique.

(C) The inverse of a correspondence is a correspondence.

(D) If f and g are each invertible then so is $g \circ f$, and $(g \circ f)^{-1} = f^{-1} \circ g^{-1}$.

B.7 Prove these for a function f with a finite domain D . They imply that corresponding finite sets have the same size. Hint: for each, you can do induction on either $|D|$ or $|\text{ran}(f)|$.

(A) $|\text{ran}(f)| \leq |D|$

(B) If f is one-to-one then $|\text{ran}(f)| = |D|$.

APPENDIX C PROPOSITIONAL LOGIC

A **proposition** is a statement that has a Boolean value, that is, it is either true or false, which we write T or F . For instance, ‘7 is odd’ and ‘ $8^2 - 1 = 127$ ’ are

propositions, with values T and F . In contrast, ‘ x is a perfect square’ is not a proposition because for some x it is T while for others it is not.

We can operate on propositions, including negating as with ‘it is not the case that 8 is prime’, or taking the conjunction of two propositions as with ‘5 is prime and 7 is prime’. The **truth tables** below define the behavior of **not** (also called **negation**), **and** (also called **conjunction**), and **or** (also called **disjunction**).

<i>not P</i>		<i>P and Q</i> $P \wedge Q$		<i>P or Q</i> $P \vee Q$	
<i>P</i>	$\neg P$	<i>P</i>	<i>Q</i>	<i>P</i>	$\neg P$
<i>F</i>	<i>T</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>
<i>T</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>T</i>	
		<i>T</i>	<i>F</i>	<i>F</i>	<i>T</i>
		<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>

Thus where ‘7 is odd’ is P , and ‘8 is prime’ is Q , get the value of ‘7 is odd and 8 is prime’ from the right-hand table’s third column, third row: F .

In some fields the practice is to write 0 where we write F and 1 in place of T .

The advantage of using symbols over writing the sentences out is that we can express more things. For instance, if P stands for ‘7 is odd’, Q stands for ‘9 is a perfect square’, and R means ‘11 is prime’ then $(P \vee Q) \wedge \neg(P \vee (R \wedge Q))$ is too complex to comfortably state in everyday language. We call that a propositional logic **expression** and denote it with a capital Roman letter such as E .

Truth tables help in working out the behavior of the complex statements by building them up from their components. The table below shows the input/output behavior of $(P \vee Q) \wedge \neg(P \vee (R \wedge Q))$.

<i>P</i>	<i>Q</i>	<i>R</i>	$P \vee Q$	$R \wedge Q$	$P \vee (R \wedge Q)$	$\neg(P \vee (R \wedge Q))$	<i>expression</i>
<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>F</i>
<i>F</i>	<i>F</i>	<i>T</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>F</i>
<i>F</i>	<i>T</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>F</i>
<i>F</i>	<i>T</i>	<i>T</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>F</i>
<i>T</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>F</i>
<i>T</i>	<i>F</i>	<i>T</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>F</i>
<i>T</i>	<i>T</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>F</i>
<i>T</i>	<i>T</i>	<i>T</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>F</i>

The three ‘ \neg ’, ‘ \wedge ’, and ‘ \vee ’ are **operators** (or **connectives**). There are other operators; here are two common ones.

<i>P</i>	<i>Q</i>	<i>P implies Q</i> $P \rightarrow Q$	<i>P if and only if Q</i> $P \leftrightarrow Q$
		<i>P</i>	$\neg P$
<i>F</i>	<i>F</i>	<i>T</i>	<i>T</i>
<i>F</i>	<i>T</i>	<i>T</i>	<i>F</i>
<i>T</i>	<i>F</i>	<i>F</i>	<i>F</i>
<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>

Two statements are **equivalent** (or **logically equivalent**) if they have equal output values whenever we give the same values to the variables. For instance,

$P \rightarrow Q$ is equivalent to $\neg P \vee Q$, because if we assign $P = F, Q = F$ then they both give the value T , if we assign $P = F, Q = T$ then they also give the same value, etc. That is, the statements are equivalent when their truth tables have the same final column. We denote equivalence using \equiv , as with $P \rightarrow Q \equiv \neg P \vee Q$

The set of formulas describing when statements are equivalent is **Boolean algebra**. For instance, these are the distributive laws

$$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R) \quad P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$$

and these are **DeMorgan's laws**.

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q \quad \neg(P \vee Q) \equiv \neg P \wedge \neg Q$$

The three operators ' \neg ', ' \wedge ', and ' \vee ' form a complete set in that we can reverse the activity above: for any truth table we can use the three to produce an expression whose input/output behavior is that table. In short, we can produce expressions with any desired behavior. Here are two examples.

P	Q	E_0	P	Q	R	E_1
F	F	T	F	F	F	F
F	T	F	F	F	T	T
T	F	F	F	T	F	T
T	T	F	F	T	T	F
			T	F	F	T
			T	F	T	F
			T	T	F	F
			T	T	T	F

To produce E_0 , on the left, focus on the T row. There, $P = F$ and $Q = F$ and so for E_0 we can use the expression $\neg P \wedge \neg Q$. For E_1 , on the right, focus on all of the T rows. Target the second row with $\neg P \wedge \neg Q \wedge R$. Target the third row with $\neg P \wedge Q \wedge \neg R$. For the fifth use $P \wedge \neg Q \wedge \neg R$. Then E_1 joins these clauses with \vee 's.

$$(\neg P \wedge \neg Q \wedge R) \vee (\neg P \wedge Q \wedge \neg R) \vee (P \wedge \neg Q \wedge \neg R) \quad (*)$$

In a propositional logic expression, a single variable such as P or Q is an **atom**. An atom or its negation, such as P or $\neg P$, is a **literal**. A **clause** is a number of literals joined by a connective (we stick to either \wedge or \vee), so that $\neg P \wedge \neg Q \wedge R$ is a clause.

The form of $(*)$ above is important. A propositional logic expression is in **Disjunctive Normal form** or **DNF** if it is a disjunction of clauses, where each clause is a conjunction of literals.

Intuition requires that there be a matching approach that starts with the F rows. We illustrate with E_0 . The second, third, and fourth rows are F . So $E_0 \equiv \neg((\neg P \wedge Q) \vee (P \wedge \neg Q) \vee (P \wedge Q))$. DeMorgan's second law gives $\neg(\neg P \wedge \neg Q) \wedge \neg(P \wedge \neg Q) \wedge \neg(P \wedge Q)$. Next DeMorgan's first law gives $E_0 \equiv (P \vee Q) \wedge (\neg P \vee Q) \wedge (\neg P \vee \neg Q)$. **Conjunctive Normal form** or **CNF** is a conjunction of clauses, where each clause is

a disjunction of literals. We use CNF more than DNF. With this form an expression evaluates to T if and only if all of its clauses evaluate to T . And each clause evaluates to T if and only if at least one of its literals evaluates to T .

A **Boolean function** has Boolean inputs, that is, they are either T or F , and Boolean outputs. By the prior paragraphs about DNF and CNF, each single-output Boolean function is determined by some Boolean expression.

Exercises

C.1 Make a truth table for each of these propositions. (A) $(P \wedge Q) \wedge R$
 (B) $P \wedge (Q \wedge R)$ (C) $P \wedge (Q \vee R)$ (D) $(P \wedge Q) \vee (P \wedge R)$

C.2 Make a truth table for these. (A) $\neg(P \vee Q)$ (B) $\neg P \wedge \neg Q$ (C) $\neg(P \wedge Q)$
 (D) $\neg P \vee \neg Q$

C.3 For the tables below, construct a DNF propositional logic expression: (A) the table on the left, (B) the one on the right.

P	Q	R		P	Q	R	
F	T						
F	F	T	T	F	F	T	F
F	T	F	T	F	T	F	T
F	T	T	F	F	T	T	F
T	F	F	F	T	F	F	F
T	F	T	T	T	F	T	F
T	T	F	F	T	T	F	T
T	T	T	F	T	T	T	T

C.4 For the tables in the prior exercise, construct a CNF propositional logic expression: (A) the table on the left, (B) the one on the right.

C.5 There are sixteen binary logical operators. Give all sixteen truth tables, and give the operator's name, such as ' $P \rightarrow Q$ ' or ' $Q \rightarrow P$ '.

Part Five
Notes

Endnotes

These are citations, sources, or discussions that supplement the text body. Each refers to a word or phrase from that text body, in italics, and then the note is in plain text. Many of the entries include links to more detail.

Cover

Calculating the bonus <http://www.loc.gov/pictures/item/np007012636/>

Preface

in addition to technical detail, also attends to a breadth of knowledge S Pinker emphasizes that a liberal approach involves making connections and understanding in a context (Pinker 2014). “It seems to me that educated people should know something about the 13-billion-year prehistory of our species and the basic laws governing the physical and living world, including our bodies and brains. They should grasp the timeline of human history from the dawn of agriculture to the present. They should be exposed to the diversity of human cultures, and the major systems of belief and value with which they have made sense of their lives. They should know about the formative events in human history, including the blunders we can hope not to repeat. They should understand the principles behind democratic governance and the rule of law. They should know how to appreciate works of fiction and art as sources of aesthetic pleasure and as impetuses to reflect on the human condition. On top of this knowledge, a liberal education should make certain habits of rationality second nature. Educated people should be able to express complex ideas in clear writing and speech. They should appreciate that objective knowledge is a precious commodity, and know how to distinguish vetted fact from superstition, rumor, and unexamined conventional wisdom. They should know how to reason logically and statistically, avoiding the fallacies and biases to which the untutored human mind is vulnerable. They should think causally rather than magically, and know what it takes to distinguish causation from correlation and coincidence. They should be acutely aware of human fallibility, most notably their own, and appreciate that people who disagree with them are not stupid or evil. Accordingly, they should appreciate the value of trying to change minds by persuasion rather than intimidation or demagoguery.” See also <https://www.aacu.org/leap/what-is-a-liberal-education>.

computational thinking <http://www.cs.cmu.edu/afs/cs/usr/wing/www/publications/Wing06.pdf>

Prologue

D Hilbert and W Ackermann Hilbert was a very prominent mathematician, perhaps the world’s most prominent mathematician, and Ackermann was his student. So they made an impression when they wrote, “[This] must be considered the main problem of mathematical logic” (Hilbert and Ackermann 1950), p 73.

mathematical statement Specifically, the statement as discussed by Hilbert and Ackermann comes from a first-order logic (versions of the *Entscheidungsproblem* for other systems had been proposed by other mathematicians). First-order logic differs from propositional logic, the logic of truth

tables, in that it allows variables. Thus for instance if you are studying the natural numbers then you can have a Boolean function $\text{Prime}(x)$. (In this context a Boolean function is traditionally called a ‘predicate’.) To make a statement that is either true or false we must then quantify statements, as in the (false) statement “for all $x \in \mathbb{N}$, $\text{Prime}(x)$ implies $\text{PerfectSquare}(x)$.” The modifier “first-order” means that the variables used by the Boolean functions are members of the domain of discourse (for Prime above it is \mathbb{N}), but we cannot have that variables themselves are Boolean functions. (Allowing Boolean functions to take Boolean functions as input is possible, but would make this a second-order, or even higher-order, logic.)

after a run He was 22 years old at the time. (Hodges 1983), p 96. This book is the authoritative source for Turing’s fascinating life. During the Second World War, he led a group of British cryptanalysts at Bletchley Park, Britain’s code breaking center, where his section was responsible for German naval codes. He devised a number of techniques for breaking German ciphers, including an electromechanical machine that could find settings for the German coding machine, the Enigma. Because the Battle of the Atlantic was critical to the Allied war effort, and because cracking the codes was critical to defeating the German submarine effort, Turing’s work was very important. (The major motion picture on this *The Imitation Game* (Wikipedia contributors 2016e) is a fun watch but is not a slave to historical accuracy.) After the war, at the National Physical Laboratory he made one of the first designs for a stored-program computer. In 1952, when it was a crime in the UK, Turing was prosecuted for homosexual acts. He was given chemical castration as an alternative to prison. He died in 1954 from cyanide poisoning which an inquest determined was suicide. In 2009, following an Internet campaign, British Prime Minister G Brown made an official public apology on behalf of the British government for “the appalling way he was treated.”

Olympic marathon His time at the qualifying event was only ten minutes behind what was later the winning time in the 1948 Olympic marathon. For more, see <https://www.turing.org.uk/book/update/part6.html> and http://www-groups.dcs.st-and.ac.uk/~history/Extras/Turing_running.html.

clerk Before the engineering of computing machines had advanced enough to make capable machines widely available, much of what we would today do with a program was done by people, then called “computers.” This book’s cover shows such computers at work.



Katherine Johnson, b 1918

Another example, as told in the film *Hidden Figures*, is that the trajectory for US astronaut John Glenn’s pioneering orbit of Earth was found by the human computer Katherine Johnson and her colleagues, African American women whose accomplishments are all the more impressive because they occurred despite appalling discrimination.

don’t involve random methods We can build things that return completely random results; one example is a device that registers consecutive clicks on a Geiger counter and if the second gap between clicks is longer than the first it returns 1, else it returns 0. See also <https://blog.cloudflare.com/randomness-101-lavarand-in-production/>.

continuous methods Before there were computers, engineers worked with analog models that were sometimes quite large; see (Wikipedia contributors 2021). In these models there is no sense of step one, step two.

analog devices See (A/V Geeks 2013) about slide rules, (Wikipedia contributors 2016c) about nomograms, (YouTube user navyreviewer 2010) about a naval firing computer, and (Gizmodo 1948) about a more general-purpose machine. See also <https://www.youtube.com/watch?v=qqlJ50zDgeA> about the Antikythera mechanism. For a more recent take, see <https://www.youtube.com/watch?v=GVsUOuSjvcg>.

reading results off of a slide rule or an instrument dial Suppose that an intermediate result of a calculation is 1.23. If we read it off the slide rule with the convention that the resolution accuracy is only one decimal place then we write down 1.2. Doubling that gives 2.4. But doubling the original number $2 \cdot 1.23 = 2.46$ and then rounding to one place gives 2.5.

no upper bound This explication is derived from (Rogers 1987), p 1–5.

more is provided Perhaps the clerk has a helper or the mechanism has a person attending it.

A reader may object that this violates the goal of the definition, to model in-principle-physically-realizable computations We often describe computations that do not have a natural bounds. The long division algorithm that we learn in grade school has no inherent bounds on the lengths of either inputs or outputs, or on the amount of available scratch paper.

are so elementary that we cannot easily imagine them further divided (Turing 1937), (Turing 1938a)

LEGO's See for instance <https://www.youtube.com/watch?v=RLPVCJjTNgk&t=114s>.

Finally, it trims off a 1 The instruction $q_4 q_1 q_5$ won't ever be reached, but it does no harm. It is there for the definition of a Turing machine, to make Δ defined on all $q_p T_p$. See also the note to that definition.

transition function The definition describes Δ as a function $\Delta: Q \times \Sigma \rightarrow (\Sigma \cup \{L, R\}) \times Q$. That is a fudge. In P_{pred} , the state q_3 is used only for the purpose of halting the machine and so there is no defined next state. In P_{add} , the state q_5 plays the same role. So, strictly speaking, the transition function is a partial function, one where for some members of the domain there is no associated value; see page 365. (Alternatively, we could write the set of states as $Q \cup \hat{Q}$ where the states in \hat{Q} are there only for halting, and the transition function's definition is $\Delta: Q \times \Sigma \rightarrow (\Sigma \cup \{L, R\}) \times (Q \cup \hat{Q})$.) We have left this point out of the main presentation since it doesn't cause confusion and the discussion can be a distraction.)

a complete description of a machine's action It is reasonable to ask why our standard model, the Turing machine, is one that is so basic that programming it can be annoying. Why not choose a real world machine? The reason is that, as here, we can completely describe the actions of the Turing machine model, or of any of the other simple model that are sometimes used, in only a few paragraphs. A real machine would take a full book, and a full semester. We do Turing machines because they are simple to describe (they are also historically important, and the work in Chapter Five needs them).

q is a state, a member of Q We are vague about what 'states' are but we assume that whatever they are, the set of states Q is disjoint from the set $\Sigma \cup \{L, R\}$.

a snapshot, an instant in a computation So the configuration along with the Turing machine is all the information that you need to continue a computation—it encapsulates the future history of that computation.

A state machine is a device that stores the status of something at a given time. On input it can change the status (it can also cause an action or output to take place). Mathematically, it is a finite set $Q = \{q_0, \dots, q_n\}$ along with a function $\Delta: Q \times \Sigma \rightarrow Q$.

rather than, “this shows that ϕ takes a string representing 3 to a string representing 5.” That is, we do this for the same reason that we would say, “This is me when I was ten.” instead of, “This is a picture of me when I was ten.”

a physical system evolves through a sequence of discrete steps that are local, meaning that all the action takes place within one cell of the head Adapted from (Wigderson 2017).

constructed the first machine See (Leupold 1725).

A number of mathematicians See also (Wikipedia contributors 2014).

Church suggested to the most prominent expert in the area, Gödel (Soare 1999)

established beyond any doubt (Gödel 1995)

This is central to the Theory of Computation Some authors have claimed that neither Church nor Turing stated anything as strong as is given here but instead that they proposed that the set of things that can be done by a Turing machine is the same as the set of things that are computable by a human computer (see for instance (Copeland and Proudfoot 1999)). But the thesis as stated here, that what can be done by a Turing machine is what can be done by any physical mechanism that is discrete and deterministic, is certainly the thesis as it is taken in the field today. And besides, Church and Turing did not in fact distinguish between the two cases; (Hodges 2016) points to Church’s review of Turing’s paper in the *Journal of Symbolic Logic*: “The author [i.e. Turing] proposes as a criterion that an infinite sequence of digits 0 and 1 be ‘computable’ that it shall be possible to devise a computing machine, occupying a finite space and with working parts of finite size, which will write down the sequence to any desired number of terms if allowed to run for a sufficiently long time. As a matter of convenience, certain further restrictions are imposed on the character of the machine, but these are of such a nature as obviously to cause no loss of generality—in particular, a human calculator, provided with pencil and paper and explicit instructions, can be regarded as a kind of Turing machine.” This has Church referring to the human calculator not as the prototype but instead as a special case of the class of defined machines.

We cannot give a mathematical proof of Church’s Thesis We cannot give a proof that starts from axioms whose justification is on firmer footing than the thesis itself. R Williams has commented, “[T]he Church-Turing thesis is not a formal proposition that can be proved. It is a scientific hypothesis, so it can be ‘disproved’ in the sense that it is falsifiable. Any ‘proof’ must provide a definition of computability with it, and the proof is only as good as that definition.” (Stack Exchange author Ryan Williams 2010)

formalizes ‘intuitively mechanically computable’ Kleene wrote that “its role is to delimit precisely an hitherto vaguely conceived totality.” (Kleene 1952), p 318.

Turing wrote (Turing 1937)

systematic error (Dershowitz and Gurevich 2008) p 304.

it may be the right answer Gödel wrote, “the great importance . . . [of] Turing’s computability [is] largely due to the fact that with this concept one has for the first time succeeded in giving an absolute definition of an interesting epistemological notion, i.e., one not depending on the formalism chosen.” (Gödel 1995), pages 150–153.

can compute all of the functions that can be computed by machines with two or more tapes For instance, we can simulate a two-tape machine \mathcal{P}_2 on a one-tape machine \mathcal{P}_1 . One way to do this is by having \mathcal{P}_1 use its even-numbered tape positions for \mathcal{P}_2 's first tape and using its odd tape positions for \mathcal{P}_2 's second tape. (A more hand-wavy explanation is: a modern computer can clearly simulate a two-tape Turing machine but a modern computer has sequential memory, which is like the one-tape machine's sequential tape.)

compute the same set of functions We must adjust the convention for what is the output of a function. *evident immediately* (Church 1937)

S Aaronson has made this point From his blog *Shtetl-Optimized*, (Aaronson 2012b).

supply a stream of random bits Some CPU's come with that capability built in; see for instance <https://en.wikipedia.org/wiki/RdRand>.

beyond discrete and deterministic From (Stack Exchange author Andrej Bauer 2016): “Turing machines are described concretely in terms of states, a head, and a working tape. It is far from obvious that this exhausts the computing possibilities of the universe we live in. Could we not make a more powerful machine using electricity, or water, or quantum phenomena? What if we fly a Turing machine into a black hole at just the right speed and direction, so that it can perform infinitely many steps in what appears finite time to us? You cannot just say ‘obviously not’ — you need to do some calculations in general relativity first. And what if physicists find out a way to communicate and control parallel universes, so that we can run infinitely many Turing machines in parallel time?”

everything that experiments with reality would ever find to be possible Modern Physics is a sophisticated and advanced field of study so we could doubt that anything large has been overlooked. However, there is historical reason for supposing that such a thing is possible. The physicists H von Helmholtz in 1856 and S Newcomb in 1892 calculated that the Sun is about 20 million years old (they assumed that the Sun glowed from the energy provided by its gravitational contraction in condensing from a nebula of gas and dust to its current state). Consistently with that, one of the world’s most reputable physicists, W Kelvin, estimated in 1897 that the Earth was, “more than 20 and less than 40 million year old, and probably much nearer 20 than 40” (he calculated how long it would take the Earth to cool from a completely molten object to its present temperature). He said, “unless sources now unknown to us are prepared in the great storehouse of creation” then there was not enough energy in the system to justify a longer estimate. One person very troubled by this was Darwin, having himself found that a valley in England took 300 million years to erode, and consequently that there was enough time, called “deep time,” for the slow but steady process of evolution of species to happen. Then, in 1896, A Becquerel discovered radiation. Everything changed. All of the prior calculations did not account for it and the apparent discrepancy vanished. (Wikipedia contributors 2016a)

the solution is not computable See (Pour-El and Richards 1981).

compute an exact solution See <http://www.smbc-comics.com/?id=3054>.

Three-Body Problem See https://en.wikipedia.org/wiki/Three-body_problem.

we can still wonder See (Piccinini 2017).

This big question remains open A sample of readings: frequently cited is (Black 2000), which takes the thesis to be about what is humanly computable, and (Copeland 1996), (Copeland 1999), and (Copeland 2002) argue that computations can be done that are beyond the capabilities of Turing machines. Against that are (Davis 2004), (Davis 2006), and (Gandy 1980), which give

arguments that many Theory of Computing researchers consider conclusive.

Often when we want to show that something is computable The same point stated another way, from (Stack Exchange author Andrej Bauer 2018): In books on computability theory it is common for the text to skip details on how a particular machine is to be constructed. The author of the computability book will mumble something about the Turing-Church thesis somewhere in the beginning. This is to be read as “you will have to do the missing parts yourself, or equip yourself with the same sense of inner feeling about computation as I did”. Often the author will give you hints on how to construct a machine, and call them “pseudo-code”, “effective procedure”, “idea”, or some such. The Church-Turing thesis is the social convention that such descriptions of machines suffice. (Of course, the social convention is not arbitrary but rather based on many years of experience on what is and is not computable.) . . . I am not saying that this is a bad idea, I am just telling you honestly what is going on. . . . So what are we supposed to do? We certainly do not want to write out detailed constructions of machines, because then students will end up thinking that’s what computability theory is about. It isn’t. Computability theory is about contemplating what machines we could construct if we wanted to, but we don’t. As usual, the best path to wisdom is to pass through a phase of confusion.

Suppose that you have infinitely many dollars. (MathOverflow user Joel David Hamkins 2010)

H Grassmann produced a more elegant definition In 1888 Dedekind used this definition to give the first rigorous proof of the laws of elementary school arithmetic.

it specifies the meaning, the semantics, of the operation A Perl’s epigram, “Recursion is the root of computation since it trades description for time” expresses this idea. The recursive definition includes steps implicitly, and with them time, in that you need to keep expanding the recursive calls. But it does not include them in preference to what they are about.

logically problematic The sense that there is something perplexing about recursion is often expressed with a story.

W James gave a public lecture on cosmology, and was approached by an older woman from the audience. “Your idea that the sun is the center of the solar system and the earth orbits around it has a good ring Mr James, but it’s wrong.” she said. “Our crust of earth lies on the back of a giant turtle.” James gently asked, “What does this turtle stand on?” “You’re very clever, Mr James,” she replied, “but the first turtle stands on the back of a second, far larger, turtle.” James persisted, “And the second turtle, Madam?” Immediately she crowed, “It’s no use Mr James—it’s turtles all the way down!” (Wikipedia contributors 2016f)

See <https://xkcd.com/1416>.

Another widely known reference is that with the invention of better microscopes, scientists studying fleas came to see that the fleas themselves had parasites. The Victorian mathematician Augustus De Morgan wrote a poem (derived from one of Jonathan Swift) called *Siphonaptera*, which is the biological order of fleas.

Great fleas have little fleas upon their backs to bite 'em,
And little fleas have lesser fleas, and so *ad infinitum*.

See also *Room 8*, winner of the 2014 short film award from the British Academy of Film and Television Arts.

define the function on higher-numbered inputs using only its values on lower-numbered ones. For the function specified by $f(0) = 1$ and $f(n) = n \cdot f(f(n - 1) - 1)$, try computing the values $f(0)$ through $f(5)$.

the first sequence of numbers ever computed on an electronic computer It was computed on EDSAC, on 1949-May-06. See (N. J. A. Sloane 2019) and (Renwick 1949).

Towers of Hanoi The puzzle was invented by E Lucas in 1883 but the next year H De Parville made of it quite a great problem with the delightful problem statement.

hyperoperation (Goodstein 1947)

$\mathcal{H}_3(4, 4)$ is much greater than the number of elementary particles in the universe The radius of the universe is about 45×10^9 light years. That's about 10^{62} Plank units. A system of much more than $r^{1.5}$ particles packed in r Plank units will collapse rapidly. So the number of particles is less than 10^{92} , which is about 2^{305} , which is much less than $\mathcal{H}_3(4, 4)$. (Levin 2016)

a programming language having only bounded loops computes all of the primitive recursive functions (Meyer and Ritchie 1966)

output only primes In fact, there is no one-input polynomial with integer coefficients that outputs a prime for all integer inputs, except if the polynomial is constant. This was shown in 1752 by C Goldbach. The proof is so simple and delightful, and not widely known, that we will give it here. Suppose p is a polynomial with integer coefficients that on integer inputs returns only primes. Fix some $\hat{n} \in \mathbb{N}$, and then $p(\hat{n}) = \hat{m}$ is a prime. Into the polynomial plug $\hat{n} + k \cdot \hat{m}$, where $k \in \mathbb{Z}$. Expanding gives lots of terms with \hat{m} in them, and gathering together like terms shows $p(\hat{n} + k \cdot \hat{m}) \equiv p(\hat{n}) \pmod{\hat{m}}$. Because $p(\hat{n}) = \hat{m}$, this gives that $p(\hat{n} + k \cdot \hat{m}) = \hat{m}$ since that is the only prime number that is a multiple of \hat{m} , and p outputs only primes. But with that, $p(n) = \hat{m}$ has infinitely many roots, and is therefore the constant polynomial. \square

Collatz conjecture See (Wikipedia contributors 2019a).

$\sin(x)$ may be calculated via its Taylor polynomial The Taylor series is $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$. We might do a practical calculation by deciding that a sufficiently good approximation is to terminate that series at the x^5 term, giving a Taylor polynomial.

C Shannon See this profile of him: <http://www.newyorker.com/tech/elements/claudeshannon-the-father-of-the-information-age-turns-1100100>.

master's thesis His paper on the subject was his master's thesis, https://en.wikipedia.org/wiki/A_Symbolic_Analysis_of_Relay_and_Switching_Circuits.

type of NOT gate This shows an N-type Metal Oxide Semiconductor Transistor. There are many other types.

problem of humans on Mars To get there the idea was to use a rocket ship impelled by dropping a sequence of atom bombs out the bottom; the energy would let the ship move rapidly around the solar system. This sounds like a crank plan but it is perfectly feasible (Brower 1983). Having been a key person in the development of the atomic bomb, von Neumann was keenly aware of their capabilities.

J Conway Conway was a magnetic person and extraordinarily creative. Sadly, Covid-19 killed him. See an excerpt from the excellent biography at <https://www.ias.edu/ideas/2015/roberts-john-horton-conway>.

M Gardner's celebrated Mathematical Games column of Scientific American in October 1970 (Gardner 1970)

computer craze (Bellos 2014)

Start A good way to experiment is the Free program Golly; see <http://golly.sourceforge.net/>.

zero-player game See <https://www.youtube.com/watch?v=R9Plq-D1gEk>.

a rabbit Discovered by A Trevororow in 1986.

We will prove that here This presentation is based on that of (Hennie 1977), (Smoryński 1991), and (Robinson 1948).

Ackermann function There are many different Ackermann functions in the literature. A common one is the function of one variable $\mathcal{A}(k, k)$.

a function is primitive recursive See the history at (Brock 2020).

LOOP (Meyer and Ritchie 1966)

Background

Deep Field movie <https://www.youtube.com/watch?v=yDiD8F9ItXo>

two paradoxes These are what Quine calls veridical paradoxes: they may at first seem absurd but we will demonstrate that they are nonetheless true. (Wikipedia contributors 2018)

Galileo's Paradox He did not invent it but he gave it prominence in his celebrated *Discourses and Mathematical Demonstrations Relating to Two New Sciences*.

same cardinality Numbers have two natures. First, in referring to the set of stars known as the Pleiades as the “Seven Sisters” we mean to take them as a set, not ordered in any way. In contrast, second, in referring to the “Seven Deadly Sins,” well, clearly some of them score higher than others. The first reference speaks to the cardinal nature of numbers and the second to their ordinal nature. For finite numbers the two are bound together, as Lemma 1.5 says, but for infinite numbers they differ.

was proposed by G Cantor in the 1870's For his discoveries, Cantor was reviled by a prominent mathematician and former professor L Kronecker as a “corrupter of youth.” That was pre-Elvis.

which is Cantor's definition (Gödel 1964)

the most important infinite set is the natural numbers, $\mathbb{N} = \{0, 1, 2, \dots\}$ Its existence is guaranteed by the Axiom of Infinity, one of the standard axioms of Mathematics, the Zermelo-Frankel axioms.

due to Zeno Zeno gave a number of related paradoxes of motion. See (Wikipedia contributors 2016g) (Huggett 2010), (Bragg 2016), as well as <http://www.smbc-comics.com/comic/zeno> and this xkcd.



Courtesy xkcd.com

the distances $x_{i+1} - x_i$ shrink toward zero, there is always further to go because of the open-endedness at the left of the interval $(0 .. \infty)$ A modern paradox that like this one uses the open-endedness of the numbers is Thomson's Lamp Paradox: a person turns on the room lights and then a minute later turns them off, a half minute later turns them on again, and a quarter minute later turns them off, etc. After two minutes, are the lights on or off? This paradox was devised in 1954 by J F Thomson to analyze the possibility of a supertask, the completion of an infinite number of tasks. Thomson's answer was that it creates a contradiction: "It cannot be on, because I did not ever turn it on without at once turning it off. It cannot be off, because I did in the first place turn it on, and thereafter I never turned it off without at once turning it on. But the lamp must be either on or off" (Thomson 1954). See also the discussion of the Littlewood Paradox (Wikipedia contributors 2016d).

Give the diagonals numbers Really, these are the anti-diagonals, since the diagonal is composed of the pairs $\langle n, n \rangle$.

arithmetic series with total $d(d + 1)/2$ It is called the d -th triangular number

$\text{cantor}(x, y) = x + [(x + y)(x + y + 1)/2]$ The Fueter-Pólya Theorem says that this is essentially the only quadratic function that serves as a pairing; see (Smoryński 1991). No one knows whether there are pairing functions that are any other kind of polynomial.

memoization The term was invented by Donald Michie (Wikipedia contributors 2016b), who among other accomplishments was a coworker of Turing's in the World War II effort to break the German secret codes.

assume that we have a family of correspondences $g_j: N \rightarrow \hat{S}_j$ To pass from the original collection of infinitely many onto functions $g_i: \mathbb{N} \rightarrow \hat{S}_i$ to a single, uniform, family of onto functions $g_j(i) = G(j, y)$ we need some version of the Axiom of Choice, perhaps Countable Choice. We omit discussion of that because it would take us far afield.

doesn't matter much For more on "much" see (Rogers 1958).

adding the instruction $q_{j+k} \text{BB} q_{j+k}$

This is essentially what a compiler calls 'unreachable code' in that it is not a state that the machine will ever be in.

central to the entire Theory of Computation The classic text (Rogers 1987) says, "It is not inaccurate to say that our theory is, in large part, a 'theory of diagonalization'."

This technique is diagonalization The argument just sketched is often called Cantor's diagonal proof, although it was not Cantor's original argument for the result, and although the argument style is not due to Cantor but instead to Paul du Bois-Reymond. The fact that scientific results are often attributed to people who are not their inventor is *Stigler's law of eponymy*. Naturally it wasn't invented by Stigler (who attributes it to Merton). In mathematics this is called *Boyer's Law*, who didn't invent it either. (Wikipedia contributors 2015).

Musical Chairs It starts with more children than chairs. Some music plays and the children walk around the chairs. But the music stops suddenly and each child tries to sit, leaving someone without a chair. That child has to leave the game, a chair is removed, and the game proceeds.

so many reals This is a Pigeonhole Principle argument.

one of them has cardinality less than or equal to the other This is equivalent to the Axiom of Choice.

consider this element of $\mathcal{P}(S)$ This is sometimes called the Russell set because of its relation to Russell's paradox. See also this XKCD.

AUTHOR KATHARINE GATES RECENTLY ATTEMPTED
TO MAKE A CHART OF ALL SEXUAL FETISHES.
LITTLE DID SHE KNOW THAT RUSSELL AND WHITEHEAD
HAD ALREADY FAILED AT THIS SAME TASK.

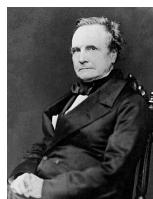


Courtesy XKCD

Your study partner is confused about the diagonal argument From (Stack Exchange author Kaktus and various others 2019).

ENIAC, reconfigure by rewiring. Jean Jennings (left), Marlyn Wescoff (center), and Ruth Licherman program the ENIAC, circa 1946. US Army Photo.

A pattern in technology is for jobs done in hardware to migrate to software One story that illustrates the naturalness of this involves the English mathematician C Babbage, and his protogee A Lovelace. In 1812 Babbage was developing tables of logarithms. These were calculated by computers—the word then current for the people who computed them by hand. To check the accuracy he had two people do the same table and compared. He was annoyed at the number of discrepancies and had the idea to build a machine to do the computing. He got a government grant to design and construct a machine called the difference engine, which he started in 1822. This was a single-purpose device, what we today would call a calculator. One person who became interested in the computations was an acquaintance of his, Lovelace (who at the time was named Byron, as she was the daughter of the poet Lord Byron).



Charles Babbage, 1791–1871



Ada Lovelace (nee Byron), 1815–1852

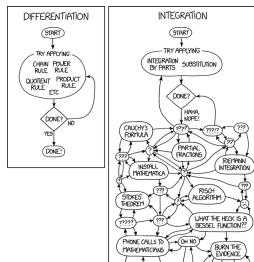
However, this machine was never finished because Babbage had the thought to make a device that would be programmable, and that was too much of a temptation. Lovelace contributed an extensive set of notes on a proposed new machine, the analytical engine, and has become known as the first programmer.

controlled by cards It weaves with hooks whose positions, raised or lowered, are determined by holes punched in the cards

have the same output behavior A technical point: Turing machines have a tape alphabet. So a universal machine's input or output can only involve symbols that it is defined as able to use. If

another machine has a different tape alphabet then how can the universal machine simulate it? As usual, we define things so that the universal machine manipulates representations of the other machine's alphabet. This is similar to the way that an everyday computer represents decimals using binary.

flowchart Flowcharts are widely used to sketch algorithms; here is one from XKCD.



Courtesy xkcd

See also http://archive.computerhistory.org/resources/text/Remington_Rand/Univ_ac.Flowmatic.1957.102646140.pdf.

the interpreter evaluate the expression that is input Writing a program that allows general users to evaluate arbitrary code is powerful but not safe, especially if these users are just surfing in from the Internet. Restricting which commands the user can evaluate, known as sandboxing, forms part of being careful with that power. For us, however, the software engineering issues are not relevant.

consecutive nines At the 762-nd decimal point there are six nines in a row. This is call the Feynman point; see https://en.wikipedia.org/wiki/Feynman_point. Most experts guess that for any n the decimal expansion contains a sequence of n consecutive nines but no one has proved or disproved that.

there is a difference between showing that this function is computable This is a little like Schrödinger's cat paradox (see https://en.wikipedia.org/wiki/Schr%C3%B6dinger%27s_cat) in that it seems that one of the two is right but we just don't know which.

"something is computable if you can write a program for it" From (Stack Exchange author JohnL 2020): "Most people, I believe, felt a bit disoriented the first time when this kind of proof/conclusion was encountered. . . We do not have to understand fully what is [the problem]. All we need is there exists an algorithm that decides [it], whatever [the answer] turns out to be. This deviates from . . . the naive sense of decidability . . . that you might have even before you encountered the theory of computation."

the i -th decimal place of π As we have noted, some real numbers have two decimal representations, one ending in 0's and one ending in 9's. But every such number is rational (as "ending in 0's" implies) and π is not rational, so π is not one of these numbers.

partial application See (Wikipedia contributors 2019d).

parametrizing A parameter is a constant that varies across equations of the same form. For instance, someone studying quadratics may consider the family of equations $y = ax^2$; here, a is a parameter. So a parameter is a kind of fixed variable. (Memorable in this context is one of A Perlis's epigrams, "One man's constant is another man's variable." (Perlis 1982))

it must be effective In fact, careful analysis shows that it is primitive recursive.

In the top case the particular output value 42 doesn't matter We often use 42 when we need a nominal output value because of its connection with *The Hitchhiker's Guide to the Galaxy*, (Adams 1979). See also (Wikipedia contributors 2020a).

undecidable The word 'undecidable' is used in mathematics in two different ways. The definition here of course applies to the Theory of Computation. In relation to results such as Gödel's theorems, it means that a statement is cannot be proved true or proved false within a given formal system.

We are using 'reduces to' An engineer, a physicist and a mathematician are asleep in a conference hotel when their coffee machine breaks into fire. The engineer grabs the fire extinguisher, sprays like crazy, and when the fire is out they all go back to sleep.

Then the room refrigerator breaks out. The physicist remembers that the fire extinguisher is now empty, quickly estimates the fire's temperature and energy output, and fills the trash can with the right amount of water, putting the fire out. They all go back to sleep.

Incredibly now the TV starts. The mathematician wakes up, remembers what happened, and hands the trash can to the physicist, thus reducing to the prior solution.

not effectively computable (Wikipedia contributors 2017h)

one of these two programs produces the right answer See [https://en.wikipedia.org/wiki/Yes_\(Unix\)](https://en.wikipedia.org/wiki/Yes_(Unix)).

halt on a proper subset of inputs but not on the rest A Turing machine could fail to halt because it has an infinite loop. The Turing machine $\mathcal{P}_0 = \{q_0Bq_0, q_011q_0\}$ never halts, cycling forever in state q_0 . We could patch this problem; we could write a program `inf_loop_decider` that at each step checks whether a machine has ever before in this computation had the same configuration as it has now. This program will detect infinite loops like the prior one.

However, note that there are machines that fail to halt but do not have loops, in that they never repeat a configuration. One is $\mathcal{P}_1 = \{q_0B1q_1, q_11Rq_0\}$ which when started on a blank tape will endlessly move to the right, writing 1's.

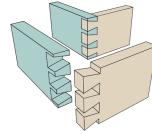
496 and 8128 The divisors of 496 are 1, 2, 4, 8, 16, 31, 62, 124, 248, and 496. The divisors of 8128 are 1, 2, 4, 8, 16, 32, 64, 127, 254, 508, 1016, 2032, 4064, and 8128.

understand the form of all even perfect numbers A number is an even perfect number if and only if it has the form $(2^p - 1) \cdot 2^{p-1}$ where $2^p - 1$ is prime.

fall to this approach A computer program that solved the Halting Problem, if one existed, could be very slow. So this might not be a feasible way to settle this question. But we are studying what can be done in principle.

any $n > 4$ such that $2^{(2^n)} + 1$ is prime A number of this form that is prime is a Fermat prime. For $n = 0$ through 4 they are 3, 5, 17, 257, and 65537, all of which are prime. Computer searches up to 30 have not found any more.

dovetailing A dovetail joint is used by carpenters or woodworkers for building strong wood drawers. It weaves the two sides in alternately, as shown here, an interlocking way.



"We shall not go any further into the nature of this oracle apart from saying that it cannot be a machine."
(Turing 1938b)

magic smoke See (Wikipedia contributors 2017f).

we will instead describe it conceptually For a full treatment see (Rogers 1987).

the notion of partial computable function seems to have an in-built defense against diagonalization (Odifreddi 1992), p 152.

this machine's name is its behavior Nominative determinism is the theory that a person's name has some influence over what they do with their life. Examples are: the sprinter Usain Bolt, the US weatherman Storm Fields, the baseball player Prince Fielder, and the Lord Chief Justice of England and Wales named Igor Judge, I Judge. See https://en.wikipedia.org/wiki/Nomative_determinism.

considered mysterious, or at any rate obscure For example, "The recursion theorem . . . has one of the most unintuitive proofs where I cannot explain why it works, only that it does." (Fortnow and Gasarch 2002)

we say that it is mentioned We can have a lot of fun with the use-mention distinction. One example is the old wisecrack that answers the statement, "Nothing rhymes with orange" with "No it doesn't," that turns on the distinction between nothing and 'nothing'. Another example is the conundrum that we all agree that $1/2 = 3/6$, but one of them involves a 3 and the other does not—how can different things be equal? The resolution is that the assertion that they are equal refers to the number that they represent, not to the representation itself. That is, in mention ' $1/2$ ' and ' $3/6$ ' are different strings but in use, they point to the same number.

mathematical fable This mathematical fable came from David Hilbert in 1924. It was popularized by George Gamow in *One, Two, Three . . . Infinity*. (Kragh 2014).

Napoleon's downfall in the early 1800's See (Wikipedia contributors 2017d).

period of prosperity and peace See (Wikipedia contributors 2017i).

A A Michelson, who wrote in 1899, "The more important fundamental laws and facts of physical science have all been discovered, and these are now so firmly established that the possibility of their ever being supplanted in consequence of new discoveries is exceedingly remote." Michelson was a major figure, whose opinions carried weight. From 1901 to 1903 he was president of the American Physical Society. In 1910–1911 he was president of the American Association for the Advancement of Science and from 1923–1927 he was president of the National Academy of Sciences. In 1907 he received the Copley Medal from the Royal Society in London, and the Nobel Prize. He remains well known today for the Michelson–Morley experiment that tried to detect the presence of aether, the hypothesized medium through which light waves travel.

working out a game by watching it being played See <https://www.youtube.com/watch?v=o1dgrylwML4>.

many observers thought that we basically had got the rules An example is that Max Planck was advised not to go into physics by his professor, who said, "in this field, almost everything is already

discovered, and all that remains is to fill a few unimportant holes.” (Wikipedia contributors 2017j)

the discovery of radiation This happened in 1896, before Michaelson’s statement. Often the significance of things takes time to be apparent

Einstein became an overnight celebrity “Einstein Theory Triumphs” was the headline in *The New York Times*. JJ Thomson, president of the Royal Society, referred to the experiment’s success as “one of the momentous, if not the most momentous, pronouncements in the history of human thought.” And, when Einstein arrived in New York by boat in 1921, reporters were delighted to find not a stuffy academic but instead someone who was very endearing, quotable, and photogenic. The hair, the scruffy clothes, and the violin, all made him seem the definition of a genius, which of course continues today.

“*everything is relative.*” Of course, the history around Einstein’s work is vastly more complex and subtle. But we are speaking of the broad understanding, not of the truth.

loss of certainty This phrase is the title of a famous popular book on mathematics, by M Klein. The book is fun and a thought-provoking read. Also thought-provoking are some criticisms of the book. (Wikipedia contributors 2019b) is good introduction to both.

from a proof of the Halting problem, we can get to a proof of Gödel’s Theorem See (Aaronson 2011). See also <https://math.stackexchange.com/a/53324/12012>.

the development of a fetus is that it basically just expands The issue was whether the fetus began preformed or as a homogeneous mass; see (Maienschein 2017). Today we have similar questions about the Big Bang—we are puzzled to explain how a mathematical point, which is without internal structure and entirely homogeneous, could develop into the very non-homogeneous universe that we see today.

infinite regress This line of thinking often depends on the suggestion that all organisms were created at the same time, that they have existed since the beginning of the posited creation.

discovery by Darwin and Wallace of descent with modification through natural selection Darwin wrote in his autobiography, “The old argument of design in nature, as given by Paley, which formerly seemed to me so conclusive, fails, now that the law of natural selection has been discovered. We can no longer argue that, for instance, the beautiful hinge of a bivalve shell must have been made by an intelligent being, like the hinge of a door by man. There seems to be no more design in the variability of organic beings and in the action of natural selection, than in the course which the wind blows. Everything in nature is the result of fixed laws.”

the car is in some way less complex than the robot This is an information theoretic analog of the Second Law of Thermodynamics. E Musk has expressed something of the same sentiment, “The extreme difficulty of scaling production of new technology is not well understood. It’s 1000% to 10,000% harder than making a few prototypes. The machine that makes the machine is vastly harder than the machine itself.” See <https://twitter.com/elonmusk/status/1308284091142266881>.

self-reference ‘Self-reference’ describes something that refers to itself. The classic example is the Liar paradox, the statement attributed to the Cretian Epimenides, “All Cretans are liars.” Because he is Cretian we take the statement to be an utterance about utterances by him, that is, to be about itself. If we suppose that the statement is true then it asserts that anything he says is false, so the statement is false. But if we suppose that it is false then we take that he is saying the truth, that all his statements are false. Its a paradox, meaning that the reasoning seems locally sound but it leads to a global impossibility.

This is related to Russell's paradox, which lies at the heart of the diagonalization technique, that if we define the collection of sets $R = \{S \mid S \notin S\}$ then $R \in R$ holds if and only if $R \notin R$ holds.

Self-reference is obviously related to recurrence. You see it sometimes pictured as an infinite recurrence, as here on the front of a chocolate product.



Because of this product, having a picture contain itself is sometimes known as the Droste effect. Besides the Liar paradox there are many others. One is Quine's paradox, a sentence that asserts its own falsehood.

“Yields falsehood when preceded by its quotation”
yields falsehood when preceded by its quotation.

If this sentence were false then it would be saying something that is true. If this sentence were true then what it says would hold and it would be not true.

A wonderful popular book exploring these topics and many others is (Hofstadter 1979).

quine Named for the philosopher Willard Van Orman Quine.

The verb ‘to quine’ Invented by D Hofstadter. It follows the statement due to the philosopher W Quine: “yields falsehood when preceded by its quotation” yields falsehood when preceded by its quotation which has the paradoxical quality that if true it asserts its own falsehood, and if false it must be true. And it accomplishes that without direct self-reference.

which n-state Turing Machine leaves the most 1's after halting R H Bruck famously wrote (Bruck 1953), “I might compare the high-speed computing machine to a remarkably large and awkward pencil which takes a long time to sharpen and cannot be held in the fingers in the usual manner so that it gives the illusion of responding to my thoughts, but is fitted with a rather delicate engine and will write like a mad thing provided I am willing to let it dictate pretty much the subjects on which it writes.” The Busy Beaver machine is the maddest writer possible.

who can write the busiest machine Two very nice videos on this subject are *The Boundary of Computation* and *What happens at the Boundary of Computation?* from YouTube contributor Mutual Information.

Radó noted in his 1962 paper This paper (Radó 1962) is exceptionally clear and interesting.

a number of websites that cover the topic The canonical one is <https://bbchallenge.org>.

$\Sigma(n)$ is unknowable See (Aaronson 2012a). See also <https://www.quantamagazine.org/the-busy-beaver-game-illuminates-the-fundamental-limits-of-math-20201210/>.

a 7918-state Turing machine The number of states needed has since been reduced. As of this writing it is 748. See the wonderful bachelor's degree thesis by J Riebel at <https://www.ingo-blechschmidt.eu/assets/bachelor-thesis-undecidability-bb748.pdf>.

the standard axioms for Mathematics This is ZFC, the Zermelo–Fraenkel axioms with the Axiom of Choice. (In addition, they also took the hypothesis of the Stationary Ramsey Property.)

take the floor Let the n -th triangle number be $t(n) = 0 + 1 + \dots + n = n(n + 1)/2$. The function t is monotonically increasing and there are infinitely many triangle numbers. Thus for every natural number c there is a unique triangle number $t(n)$ that is maximal so that $c = t(n) + k$ for some $k \in \mathbb{N}$. Because $t(n+1) = t(n) + n + 1$, we see that $k < n + 1$, that is, $k \leq n$. Thus, to compute the diagonal number d from the Cantor number c of a pair, we have $(1/2)d(d+1) \leq c < (1/2)(d+1)(d+2)$. Applying the quadratic formula to the left half and right halves gives $(1/2)(-3 + \sqrt{1 + 8c}) < d \leq (1/2)(-1 + \sqrt{1 + 8c})$. Taking $(1/2)(-1 + \sqrt{1 + 8c})$ to be α gives that $c \in (\alpha - 1 .. \alpha]$ so that $d = \lfloor \alpha \rfloor$. (SE author Brian M. Scott 2020)

let's extend to tuples of any size See https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it.

Languages

having elephants move to the left side of a road or to the right Less fancifully, we could be making a Turing machine out of LEGO's and want to keep track by sliding a block from one side of a column to the other. Or, we could use an abacus.

we could translate any such procedure While a person may quite sensibly worry that elephants could be not just on the left side or the right, but in any of the continuum of points in between, we will make this assertion without more philosophical analysis than by just referring to the discrete nature of our mechanisms (as Turing basically did). That is, we take it as an axiom.

finite set {1000001, 1100001} Although it looks like two strings plucked from the air, the language is not without sense. The bitstring 1000001 represents capital A in the ASCII encoding, while 1100001 is lower case a. The American Standard Code for Information Interchange, ASCII, is a widely used, albeit quite old, way of encoding character information in computers. The most common modern character encoding is UTF-8, which extends ASCII. For the history see <https://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt>.

palindrome Sometimes people call Psychology the study of college freshmen because so many studies start, roughly, “we put a bunch of college freshmen in a room, lied to them about what we were doing, and . . . ” In the same way, Theory of Computing can sometimes seem like the study of palindromes.

English palindromes Some people like to move beyond single word palindromes to make sentence-length palindromes that make some sense. Some of the more famous are: (1) supposedly the first sentence ever uttered, “Madam, I’m Adam” (2) Napoleon’s lament, “Able was I ere I saw Elba” and (3) “A man, a plan, a canal: Panama”, about Theodore Roosevelt.

In practice a language is usually given by rules Linguists started formalizing the description of language, including phrase structure, at the start of the 1900’s. Meanwhile, string rewriting rules as formal, abstract systems were introduced and studied by mathematicians including Axel Thue in 1914, Emil Post from the 1920’s through the 1940’s and Turing in 1936. Noam Chomsky, while teaching linguistics to students of information theory at MIT, combined linguistics and mathematics by taking Thue’s formalism as the basis for the description of the syntax of natural language. (Wikipedia contributors 2017e)

“the red big barn” sounds wrong. Experts vary on the exact rules but one source gives (article) + number + judgment/attitude + size, length, height + age + color + origin + material +

purpose + (noun), so that “big red barn” is size + color + noun, as is “little green men.” This is called the Royal Order of Adjectives; see <http://english.stackexchange.com/a/1159>. A person may object by citing “big bad wolf” but it turns out there is another, stronger, rule that if there are three words then they have to go I-A-O and if there are two words then the order has to be I followed by either A or O. Thus we have tick tock but not tock tick. Similarly for tic-tac-toe, mishmash, King Kong, or dilly dally.

very strict rules Everyone who has programmed has had a compiler chide them about a syntax violation.

grammars are the language of languages. From Matt Swift, <http://matt.might.net/articles/grammars-bnf-ebnf/>.

this grammar Taken from https://en.wikipedia.org/wiki/Formal_grammar.

dangling else See https://en.wikipedia.org/wiki/Dangling_else.

postal addresses. Adapted from https://en.wikipedia.org/wiki/BackusNaur_Form.

Recall Turing’s prototype computer In this book we stick to grammars where each rule head is a single nonterminal. That greatly restricts the languages that we can compute. More general grammars can compute more, including every set that can be decided by a Turing machine.

we often state problems For instance, see the blogfeed for Theoretical Computer Science <http://cstheory-feed.org/> (Various authors 2017)

represent graphs Example 3.2 make the point that a graph is about the connections between vertices, not about how it is drawn. This graph representation via a matrix also illustrates that point because it is, after all, not drawn.

the most common way to express grammars One factor influencing its adoption was a letter that D Knuth wrote to the *Communications of the ACM* (Knuth 1964). He listed some advantages over the grammar-specification methods that were then widely used. Most importantly, he contrasted BNF’s more descriptive elements such as using ‘<addition operator>’ instead of ‘A’, saying that the difference is a great addition to “the explanatory power of a syntax.” He also proposed the name Backus Naur Form. (Now a hyphen is most common.)

some extensions for grouping and replication The best current standard is <https://www.w3.org/TR/xml/>.

Time is a difficult engineering problem One complication of time, among many, is leap seconds. The Earth is constantly undergoing deceleration caused by the braking action of the tides. The average deceleration of the Earth is roughly 1.4 milliseconds per day per century, although the exact number varies from year to year depending on many factors, including major earthquakes and volcanic eruptions. To ensure that atomic clocks and the Earth’s rotational time do not differ by more than 0.9 seconds, occasionally an extra second is added to civil time. This leap second can be either positive or negative depending on the Earth’s rotation—on occasion there are minutes with only 58 seconds, and on occasion minutes with 60.

Adding to the confusion is that the changes in rotation are uneven and we cannot predict leap seconds far into the future. The International Earth Rotation Service publishes bulletins that announce leap seconds with a few weeks warning. Thus, there is no way to determine how many seconds there will be between the current instant and, say, ten years from now. (This can cause trouble in area such as navigation and high-frequency trading and there are proposals to eliminate leap seconds or replace them with leap hours.) Since the first leap second in 1972, all

leap seconds have been positive and there were 23 leap seconds in the 34 years to January 2006.
(U.S. Naval Observatory 2017)

RFC 3339 (Klyne and Newman 2002)

strings such as 1958-10-12T23:20:50.52Z This format has a number of advantages including human readability, that if you sort a collection of these strings then earlier times will come earlier, simplicity (there is only one format), and that they include the time zone information.

a BNF grammar Some notes: (1) Coordinated Universal Time, the basis for civil time, is often called UTC, but is sometimes abbreviated Z and read aloud as “Zulu,” (2) years are four digits to prevent the Y2K problem (Encyclopædia Britannica Editors 2017), (3) the only month numbers allowed are 01–12 and in each month only some day numbers are allowed, and (4) the only time hours allowed are 00–23, minutes must be in the range 00–59, etc. (Klyne and Newman 2002)

Automata

what can be done by a machine having a number of possible configurations that is bounded From Rabin, Scott, Finite Automata and Their Decision Problems, 1959: *Turing machines are widely considered to be the abstract prototype of digital computers; workers in the field, however, have felt more and more that the notion of a Turing machine is too general to serve as an accurate model of actual computers. It is well known that even for simple calculations it is impossible to give an a priori upper bound on the amount of tape a Turing machine will need for any given computation. It is precisely this feature that renders Turing's concept unrealistic. In the last few years the idea of a finite automaton has appeared in the literature. These are machines having only a finite number of internal states that can be used for memory and computation. The restriction on finiteness appears to give a better approximation to the idea of a physical machine. Of course, such machines cannot do as much as Turing machines, but the advantage of being able to compute an arbitrary general recursive function is questionable, since very few of these functions come up in practical applications.*

transition function $\Delta: Q \times \Sigma \rightarrow Q$ Some authors allow the transition function to be partial. That is, some authors allow that for some state-symbol pairs there is no next state. This choice by an author is a matter of convenience, as for any such machine you can create an error state q_{error} or dead state, that is not an accepting state and that transitions only to itself, and send all such pairs there. This transition function is total, and the new machine has the same collection of accepted strings as the old.

Unicode While in the early days of computers characters could be encoded with standards such as ASCII, which includes only upper and lower case unaccented letters, digits, a few punctuation marks, and a few control characters, today's global interconnected world needs more. The Unicode standard assigns a unique number called a code point to every character in every language (to a fair approximation). See (Wikipedia contributors 2017l).

if a language is finite then there is a Finite State machine that accepts a string if and only if it is a member of that language In practice the suggestion that for any finite set of strings there is a Finite State machine that accepts it, simply by listing all of the cases may not be reasonable. For example, there are finitely many people and each has finitely many active phone numbers so the set of all currently-active phone numbers is a finite language. But constructing a machine for it would be silly. In addition, a finite language doesn't have to be large for it to be difficult, in a sense. Take Goldbach's conjecture, that every even number greater than 2 is the sum of two primes, as in $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5$, ... Computer testing shows that this pattern continues to hold up to very large numbers but no one knows if it is true for all evens. Now

consider the set consisting of the string $\sigma \in \{0, \dots, 9\}^*$ representing in decimal the smallest even number that is not the sum of two primes. This set is finite since it has either one member or none. But while that set is tiny, we don't know what it contains.

this was how phone dialing was handled in North America See the description of the North America Numbering Plan (Wikipedia contributors 2017g).

physical devices The devices to do the switching were invented in 1889 by an undertaker whose competitor's wife was the local telephone operator and routed calls to her husband's business. (Wikipedia contributors 2017b)

same-area local exchange Initially, large states, those divided into multiple numbering plan areas, were assigned area codes with a 1 in the second position. Areas that covered entire states or provinces got codes with 0 as the middle digit. This was abandoned by the early 1950s. (Wikipedia contributors 2017g).

Alcuin of York (735–804) See <https://www.bbc.co.uk/programmes/m000dqy8>.

a wolf, a goat, and a bundle of cabbages This translation is from A Raymond, from the University of Washington.

that of finding the shortest circuit visiting every city in a list See <https://nbviewer.jupyter.org/url/norvig.com/ipython/TSP.ipynb>.

US lower forty eight states See <https://wiki.openstreetmap.org/wiki/TIGER>.

no-state A person can wonder about no-state. Where is it, exactly? We can think that it is like what happens if you write a program with a sequence of if-then statements and forget to include an else. Obviously a computer goes somewhere, the instruction pointer points to some next address, but what happens is not sensible in terms of the model you've intended.

Alternatively, the wonderful book (Hofstadter 1979) describes a place named Tumbolia, which is where holes go when they are filled, and also where your lap goes when you stand up. Perhaps the machines go there.

amb(...) The name *amb* abbreviates ‘ambiguous function’. Here is a small example. Essentially *Amb*(*x*, *y*, *z*) splits the computation into three possible futures: a future in which the value *x* is yielded, a future in which the value *y* is yielded, and a future in which the value *z* is yielded. The future which leads to a successful subsequent computation is chosen. The other “parallel universes” somehow go away. (*Amb* called with no arguments fails.) See <https://rosettacode.org/wiki/Amb>. These were described by John McCarthy in (McCarthy 1963). “Ambiguous functions are not really functions. For each prescription of values to the arguments the ambiguous function has a collection of possible values. An example of an ambiguous function is *less*(*n*) defined for all positive integer values of *n*. Every non-negative integer less than *n* is a possible value of *less*(*n*). First we define a basic ambiguity operator *amb*(*x*, *y*) whose possible values are *x* and *y* when both are defined: otherwise, whichever is defined. Now we can define *less*(*n*) by *less*(*n*) = *amb*(*n* – 1, *less*(*n* – 1)).”

demonic The term ‘demon’ arose from Maxwell’s demon. This is a thought experiment created in 1867 by the physicist J C Maxwell about the second law of thermodynamics, which says that it takes energy to raise the temperature of a sealed system. Maxwell imagined a chamber of gas with a door controlled by an all-knowing demon. When the demon sees a gas molecule of gas approaching that is slow-moving, it opens the door and lets that molecule out of the chamber, thereby raising the chamber’s temperature without any external heat. See (Wikipedia contributors 2019c).

$c \in \Sigma$ Here we see that to denote a ϵ transition we use the character ‘ ϵ ’, where it does not represent the empty string. A fine point, to be sure.

Pronounced KLAY-nee His son Ken Kleene, wrote, “As far as I am aware this pronunciation is incorrect in all known languages. I believe that this novel pronunciation was invented by my father.” (Free Online Dictionary of Computing (Denis Howe) 2017)

mathematical model of neurons (Wikipedia contributors 2017c)

have a vowel in the middle Most speakers of American English cite the vowels as ‘a’, ‘e’, ‘i’, ‘o’, and ‘u’. See (Bigham 2014).

before and after pictures This diagram is derived from (Hopcroft, Motwani, and Ullman 2001).

The fact that we can describe these languages in so many different ways (Stack Exchange author David Richerby 2018).

performing that operation on its members always yields another member Familiar examples are that adding two integers always gives an integer so the integers are closed under the operation of addition, and that squaring an integer always results in an integer so that the integers are closed under squaring.

the machine accepts at least one string of length k, where $n \leq k < 2n$ This gives an algorithm that inputs a Finite State machine and determines, in a finite time, if it recognizes an infinite language.

Moore’s algorithm It is easy and suitable for small calculations but if you are writing code then be aware that another algorithm, Hopcroft’s algorithm, is more efficient. See (Knuutila 2001).

In a formula The technical term is that p is a homomorphism between the automata.

The last verification is that M_O has a minimal number of states This is derived from the presentation in (Hopcroft, Motwani, and Ullman 2001).

reserved a character We use it only to mark the stack bottom, never in the middle of the stack.

We are ready for the definition There are a variety of definitions for Pushdown machines. For instance, here we have the machine accepts if its tape is empty and it is in an accepting state, but a variant requires that its stack be empty. However, all of these variants that extend nondeterministic machines accept the same set of languages.

$\Delta: Q \times (\Sigma \cup \{B, \epsilon\}) \times (\Gamma \cup \{\perp\}) \rightarrow \mathcal{P}(Q \times (\Gamma \cup \{\perp\})^*)$ Tape outputs consist of sequences of elements of Γ that optionally end in a \perp . So a more precise codomain is $\mathcal{P}(Q \times S)$ for $S = \Gamma^* \cup (\Gamma^* \cap \{\perp\})$.

without proof An excellent source for more is (Hopcroft, Motwani, and Ullman 2001).

including C, Java, Python, and Racket This is a good approximation but the full story is more complicated. Usually the set of programs accepted by the parser is a subset of a context free language, conditioned on some additional rules that the parser enforces. For example, in C every variable must be appear in a declaration inside an enclosing scope, which is clearly a context-sensitive constraint. Another example is that in Python all the whitespace prefixes inside a block have to be the same length, which again is a context-sensitive constraint.

\d We shall ignore cases of non-ASCII digits, that is, cases outside 0–9. Unicode includes many different sets of graphemes for the decimal digits, along with non-decimal numerals such as Roman numerals. There are also a number of typographical variations of the ASCII numerals

provided for specialized mathematical use and for compatibility with earlier character sets, such as circled digits sometimes used for itemization.

ZIP codes ZIP stands for Zone Improvement Plan. The system has been in place since 1963 so it, like the music movement called ‘New Wave’, is an example of the danger of naming your project something that will become obsolete if that project succeeds.

a colon and two forward slashes The inventor of the World Wide Web, T Berners Lee, has admitted that the two slashes don’t have a purpose (Firth 2009).

more power than the theoretical regular expressions that we studied earlier Omitting this power, and keeping the implementation in sync with the theory, has the advantage of speed. See (Cox 2007).

It is described by the regex It is credited to the Perl hacker Abigail, from <http://abigail.be/>.

valid email addresses This expression follows the RFC 822 standard. The full listing is at <http://www.ex-parrot.com/pdw/Mail-RFC822-Address.html>. It is due to Paul Warren who did not write it by hand but instead used a Perl program to concatenate a simpler set of regular expressions that relate directly to the grammar defined in the RFC. To use the regular expression, should you be so reckless, you would need to remove the formatting newlines.

J Zawinski The post is from alt.religion.emacs on 1997-Aug-12. For some reason it keeps disappearing from the online archive. The full discussion reveals that the quote is more dogmatic than the complete assertion. One response to the quote is, “Some people, when confronted with a problem, think ‘I know, I’ll quote Jamie Zawinski.’ Now they have two problems.” (Martin Liebach, 2009-Mar-04, <https://m.lieba.ch/2009/03/04/regex-humor/>).

Now they have two problems. A classic example is trying to use regular expressions to parse an HTML document. Sometimes scraping a fixed document to get some needed data by using regexes is just what you need, quick and not too hard. But to parse significant parts of an HTML document, or to try to anticipate possible changes, just leads to horrors. See (Stack Exchange author bobnica 2009).

regex golf See <https://alf.nu/RegexGolf>, and <https://nbviewer.jupyter.org/url/norvig.com/ipython/xkcd1313.ipynb>.

Complexity

mirrors the subject’s history This is like the slogan “ontogeny recapitulates phylogeny” for the now-discredited biological theory that the development of an embryo, which is called ontogeny, goes through same stages as the adult stages in the evolution of the animal’s ancestors, which is phylogeny.

A natural next step is to look to do jobs efficiently S Aaronson states it more provocatively as, “[A]s computers became widely available starting in the 1960s, computer scientists increasingly came to see computability theory as not asking quite the right questions. For, almost all the problems we actually want to solve turn out to be computable in Turing’s sense; the real question is which problems are *efficiently* or *feasibly* computable.” (Aaronson 2011)

A Karatsuba See https://en.wikipedia.org/wiki/Anatoly_Karatsuba.

clever algorithm The idea is: let $k = \lceil n/2 \rceil$ and write $x = x_1 2^k + x_0$ and $y = y_1 2^k + y_0$ (so for instance, $678 = 21 \cdot 2^5 + 6$ and $42 = 1 \cdot 2^5 + 10$). Then $xy = A \cdot 2^{2k} + B \cdot 2^k + C$ where $A = x_1 y_1$, and $B = x_1 y_0 + x_0 y_1$, and $C = x_0 y_0$ (for example, $28476 = 21 \cdot 2^{10} + 216 \cdot 2^5 + 60$). The multiplications by 2^{2k} and 2^k are just bit-shifts to known locations independent of the values of x

and y , so they don't affect the time much. But the two multiplications for B seem remove all the advantage and still give n^2 time. However, Karatsuba noted that $B = (x_0 + x_1) \cdot (y_0 + y_1) - A - C$. Boom: done. Just one multiplication.

The ' $f = \mathcal{O}(g)$ ' notation is very common, but awkward See also https://whystartat.xyz/wiki/Big_O_notation.

are most common in practice Sometimes in practice intermediate powers are notable. For instance, at this moment the complexity of matrix multiplication is $\mathcal{O}(n^{2.373})$, approximately. But most often we work with natural number expressions.

next table shows why This table is adapted from (Garey and Johnson 1979).

there are 3.16×10^7 seconds in a year The easy way to remember this is the bumper sticker slogan by Tom Duff from Bell Labs: “ π seconds is a nanocentury.”

very, very much larger than polynomial growth According to an old tale from India, the Grand Vizier Sissa Ben Dahir invented chess. For it, the delighted Indian King granted him a wish. Sissa said, “Majesty, give me a grain of wheat to place on the first square of the board, and two grains of wheat to place on the second square, and four grains of wheat to place on the third, and eight grains of wheat to place on the fourth, and so on. Oh, King, let me cover each of the 64 squares of the board.”

“And is that all you wish, Sissa, you fool?” exclaimed the astonished King.

“Oh, Sire,” Sissa replied, “I have asked for more wheat than you have in your entire kingdom. Nay, for more wheat than there is in the whole world, truly, for enough to cover the whole surface of the earth to the depth of the twentieth part of a cubit.”

Sissa has the right idea but his arithmetic is slightly off. A cubit is the length of a forearm, from the tip of the middle finger to the bottom of the elbow, so perhaps twenty inches. The geometric series formula gives $1 + 2 + 4 + \dots + 2^{63} = 2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615 \approx 1.84 \times 10^{19}$ grains of rice. The surface area of the earth, including oceans, is 510 072 000 square kilometers. There are 10^{10} square centimeters in each square kilometer so the surface of the earth is 5.10×10^{18} square centimeters. That's between three and four grains of rice on every square centimeter of the earth. Not rice an inch thick, but still a lot.

Another way to get a sense of the amount of rice is: there are about 7.5 billion people on earth so it is on the order of 10^8 grains of rice for each person in the world. There are about $1\,000\,000 = 10^7$ grains of rice in a bushel. In sum, ten bushels for each person.

Cobham's thesis Credit for this goes to both A Cobham and J Edmonds, separately; see (Cobham 1965) and (Edmonds 1965).



Jack Edmonds, b 1934



Alan Cobham, 1927–2011

Cobham's paper starts by asking whether “is it harder to multiply than to add?” a question that we still cannot answer. Clearly we can add two n -bit numbers in $\mathcal{O}(n)$ time, but we don't know whether we can multiply in linear time.

Cobham then goes on to point out the distinction between the complexity of a problem and the running time of a particular algorithm to solve that problem, and notes that many familiar functions, such as addition, multiplication, division, and square roots, can all be computed in time “bounded by a polynomial in the lengths of the numbers involved.” He suggests we consider the class of all functions having this property.

As for Edmunds, in a “Digression” he writes: ‘An explanation is due on the use of the words ‘efficient algorithm.’ According to the dictionary, ‘efficient’ means ‘adequate in operation or performance.’ This is roughly the meaning I want—in the sense that it is conceivable for [this problem] to have no efficient algorithm. . . . There is an obvious finite algorithm, but that algorithm increases in difficulty exponentially with the size of the graph. It is by no means obvious whether or not there exists an algorithm whose difficulty increases only algebraically with the size of the graph . . . If only to motivate the search for good, practical algorithms, it is important to realize that it is mathematically sensible even to question their existence.”

(It is worth noting that Cobham and Edwards were not the first to talk about polynomial and other time behaviors. For instance, in 1910 HC Pocklington discussed it in exploring the behavior of algorithms for solving quadratic congruences. But Cobham and Edwards were the ones who started the current interest.)

tractable Another word that you can see in this context is ‘feasible’. Some authors use them to mean the same thing, roughly that we can solve reasonably-sized problem instances using reasonable resources. But some authors use ‘feasible’ to have a different connotation, for instance explicitly disallowing inputs are too large, such as having too many bits to fit in the physical universe. The word ‘tractable’ is more standard and works better with the definition that includes the limit as the input size goes to infinity, so here we stick with it.

slower than the right by four calculations We won’t consider whether the compiler optimizes it out of the loop.

if the algorithm is $\mathcal{O}(n^2)$ on the RAM then on the Turing machine it can be $\mathcal{O}(n^5)$ A more extreme example of a model-based difference is that addition of two $n \times n$ matrices on a RAM model takes time that is $\mathcal{O}(n^2)$, but on an unboundedly parallel machine model it takes constant time, $\mathcal{O}(1)$.

Its definition ignores constant factors This discussion originated as (Stack Exchange author babou and various others 2015).

could that not make the second algorithm more useful in practice? A great writeup of the details of an algorithm for small values is the description of the sorting algorithm used by Python in (Peters 2023).

the order of magnitude of these constants For a rough idea of what these may be, here are some numbers that every programmer should know.

Operation	Cost in nanoseconds
Cache reference	0.5–7
Branch mispredict	5
Main memory reference	100
Send 1K bytes over 1 Gbps network	10 000
Read 1 MB sequentially from disk	20 000 000
Send packet CA to Netherlands to CA	150 000 000

A nanosecond is 10^{-9} seconds. For more, see <https://www.youtube.com/watch?v=JEpsKnWZ>

rJ8&app=desktop.

update that standard Even Knuth had to update standards, from his machine model MIX to MMIX.
an important part of the culture That is, these are storied problems.

inventor of the quaternion number system See https://en.wikipedia.org/wiki/History_of_quaternions.

Around the World Another version was called *The Icosian Game*. See <http://puzzlemuseum.com/month/picm02/200207icosian.htm>.

This is the solution given by L Euler The figure is from (Euler 1766).

find the shortest-distance circuit that visits every city Traveling Salesman was first posed by K Menger in an article that appeared in the same journal and issue as Gödel's Incompleteness Theorem. The two were close friends.

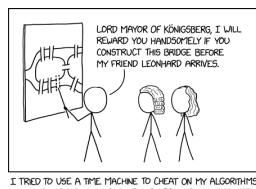
Kaliningrad is a Russian enclave between Poland and Lithuania



Kaliningrad, on the Baltic Sea between Poland and Lithuania

Königsberg This happens to be the hometown of David Hilbert.

A local mayor wrote to Euler XKCD, as usual, is on it.



Courtesy xkcd.com

no circuit is possible Consider a land mass. For each bridge in there must be an associated bridge out. So an at least necessary condition is that the land masses have an even number of associated edges. But that is not true for this city.

the countries must be contiguous A notable example of a non-contiguous country in the world today is that Russia is separated from Kaliningrad, the city that used to be known as Königsberg.

we can draw it in the plane This is because the graph comes from a planar map.

start with a planar graph The graph is undirected and without loops.

Counties of England and the derived planar graph This is today's map. At the time, some counties were not contiguous.

it was controversial

See https://www.maa.org/sites/default/files/pdf/upload_library/22/Ford/Swart69-7-707.pdf.

Given a graph and a number $k \in \mathbb{N}$ In the name of the problem we often omit the k .

Conjunctive Normal form Any Boolean function can be expressed in that form; see the Appendix.

The table above gives the numbers for the 2020 election Here are the abbreviations for states and the District of Columbia: Alabama AL, Alaska AK, Arizona AZ, Arkansas AR, California CA, Colorado CO, Connecticut CT, Delaware DE, District of Columbia DC, Florida FL, Georgia GA, Hawaii HI, Idaho ID, Illinois IL, Indiana IN, Iowa IA, Kansas KS, Kentucky KY, Louisiana LA, Maine ME, Maryland MD, Massachusetts MA, Michigan MI, Minnesota MN, Mississippi MS, Missouri MO, Montana MT, Nebraska NE, Nevada NV, New Hampshire NH, New Jersey NJ, New Mexico NM, New York NY, North Carolina NC, North Dakota ND, Ohio OH, Oklahoma OK, Oregon OR, Pennsylvania PA, Rhode Island RI, South Carolina SC, South Dakota SD, Oklahoma OK, Tennessee TN, Texas TX, Utah UT, Vermont VT, Virginia VA, Washington WA, Wisconsin WI, Wyoming WY

ignore some fine points Both Maine and Nebraska have two districts, and each elects their own representative to the Electoral College, rather than having two state-wide electors who vote the same way.

words can be packed into the grid The earliest known example is the Sator square, five Latin words that pack into a grid.

SATOR	S	A	T	O	R
AREPO	A	R	E	P	O
FENET	T	E	N	E	T
OPERA	O	P	E	R	A
ROTAS	R	O	T	A	S

It appears in many places in the Roman Empire, often as graffiti. For instance, it was found in the ruins of Pompeii. Like many word game solutions it sacrifices comprehension for form but it is a perfectly grammatical sentence that translates as something like, “The farmer Arepo works the wheel with effort.”

popular with $n = 4$ as a toy It was invented by Noyes Palmer Chapman, a postmaster in Canastota, New York. As early as 1874 he showed friends a precursor puzzle. By December 1879 copies of the improved puzzle were circulating in the northeast and students in the American School for the Deaf and other started manufacturing it. They became popular as the “Gem Puzzle.” Noyes Chapman had applied for a patent in February, 1880. By that time the game had become a craze in the US, somewhat like Rubik’s Cube a century later. It was also popular in Canada and Europe. See (Wikipedia contributors 2017a).

We know of no efficient algorithm to find divisors An effort in 2009 to factor a 768-bit number (232-digits) used hundreds of machines and took two years. The researchers estimated that a 1024-bit number would take about a thousand times as long.

Factoring seems to be hard Finding factors has for many years been thought hard. For instance, a number is called a Mersenne prime if it is a prime number of the form $2^n - 1$. They are named after M Mersenne, a French friar and important figure in the early sharing of scientific results, who studied them in the early 1600’s. He observed that if n is prime then $2^n - 1$ may be prime,

for instance with $n = 3$, $n = 7$, $n = 31$, and $n = 127$. He suspected that others of that form were also prime, in particular $n = 67$.

On 1903-Oct-31 F N Cole, then Secretary of the American Mathematical Society, made a presentation at a math meeting. When introduced, he went to the chalkboard and in complete silence computed $2^{67} - 1 = 147\,573\,952\,589\,676\,412\,927$. He then moved to the other side of the board, wrote 193 707 721 times 761 838 257 287, and worked through the calculation, finally finding equality. When he was done Cole returned to his seat, having not uttered a word in the hour-long presentation. His audience gave him a standing ovation.

Cole later said that finding the factors had been a significant effort, taking “three years of Sundays.”

Platonic solids See (Wikipedia contributors 2017k).

as shown Some PDF readers cannot do opacity, so you may not see the entire Hamiltonian path.

Six Degrees of Kevin Bacon One night, three college friends, Brian Turtle, Mike Ginelli, and Craig Fass, were watching movies. *Footloose* was followed by *Quicksilver*, and between was a commercial for a third Kevin Bacon movie. It seemed like Kevin Bacon was in everything! This prompted the question of whether Bacon had ever worked with De Niro? The answer at that time was no, but De Niro was in *The Untouchables* with Kevin Costner, who was in *JFK* with Bacon. The game was born. It became popular when they wrote to Jon Stewart about it and appeared on his show. (From (Blanda 2013).) See <https://oracleofbacon.org/>.

uniform family of tasks From (Jones 1997).

There is no widely-accepted formal definition of ‘algorithm’ This discussion derives from (Pseudonym 2014).

we prefer language decision problems Because of this, some authors modify the definition of a Turing machine to have it come with a subset of accepting states. Such a machine solves a problem if it halts on all input strings, and when it halts it is in an accepting state exactly when that string is in the language.

default interpretation of ‘problem’ Not every computational problem is naturally expressible as a language decision problem Consider the task of sorting the characters of strings into ascending order. We could try to express it as the language of sorted strings $\{\sigma \in \Sigma^* \mid \sigma \text{ is sorted}\}$. But recognizing a correctly-sorted string does not require that we find a good way to sort an unsorted input. Another thought is to consider the language of pairs $\langle\sigma, p\rangle$ where p is a permutation of the numbers $0, \dots, |\sigma| - 1$ that brings the string into ascending order. Here also the formulation seems to not capture the sorting problem, in that recognizing a correct permutation feels different than generating one from scratch.

Both of these show the collection of languages One misleading aspect of this picture is that there are uncountably many languages but only countably many Turing machines, and hence only countably many computable or computably enumerable languages. So, shown to scale, the computably enumerable area of the blob would be an infinitesimally small speck at the very bottom. But such a picture would not show the features we want to illustrate, so these drawings take a graphical license.

the shaded collection Rec consists of the Turing computable languages The name **Rec** is because these used to be known as the ‘recursive’ languages.

input two numbers and output their midpoint See <https://hal.archives-ouvertes.fr/file/index/docid/576641/filename/computing-midpoint.pdf>.

final two bits are 00 Decimal representation is not much harder since a decimal number is divisible by four if and only if the final two digits are in the set {00, 04, ... 96}.

everything of interest can be represented with reasonable efficiency by bitstrings See <https://rjlipton.wordpress.com/2010/11/07/what-is-a-complexity-class/>. Of course, a wag may say that if it cannot be represented by bitstrings then it isn't of interest. But we mean something less tautological: we mean that if we could want to compute with it then it can be put in bitstrings. For example, we find that we can process speech, adjust colors on an image, or regulate pressure in a rocket fuel tank, all in bitstrings, despite what may at first encounter seem to be the inherently analog nature of these things.

Beethoven's 9th Symphony The official story is that CD's are 72 minutes long so that they can hold this piece.

researchers often do not mention representations This is like a programmer saying, "My program inputs a number" rather than, "My program inputs the binary representation of a number." It is also like a person saying, "That's me on the card" rather than "That's a picture of me."

leaving implementation details to a programmer (Grossman 2010)

the time or space behavior We will concentrate our attention resource bounds in the range from logarithmic and exponential, because these are the most useful for understanding problems that arise in practice.

less than centuries See the video from Google at <https://www.youtube.com/watch?v=-ZNEzzDcllU> and S Aaronson's Quantum Supremacy FAQ at <https://www.scottaaronson.com/blog/?p=4317>.

The claim is the subject of scholarly reservations See the posting from IBM Research at <https://www.ibm.com/blogs/research/2019/10/on-quantum-supremacy/> and G Kalai's Quantum Supremacy Skepticism FAQ at <https://gilkalai.wordpress.com/2019/11/13/gils-collegial-quantum-supremacy-skepticism-faq/>.

We give the class P our attention This discussion gained much from the material in (Allender, Loui, and Regan 1997). This includes several direct quotations.

RE Recall that 'recursively enumerable' is an older term for 'computably enumerable'.

adds some wrinkles But it avoids a wrinkle that we needed for Finite State machines and Pushdown machines, ϵ transitions, since Turing machines are not required to consume their input one character at a time.

function computed by a nondeterministic machine One thing that we can do is to define that the nondeterministic machine computes $f: \mathbb{B}^* \rightarrow \mathbb{B}^*$ is that if on an input σ , all branches halt and they all leave the same value on the tape, which we call $f(\sigma)$. Otherwise, the value is undefined, $f(\sigma) \uparrow$.

might be much faster R Hamming gives this example to demonstrate that an order of magnitude change in speed can change the world, can change what can be done: we walk at 4 mph, a car goes at 40 mph, and an airplane goes at 400 mph. This relates to the bug picture that opens this chapter.

we don't find the ω 's, we just use them This is like the Mechanical Turk https://en.wikipedia.org/wiki/Mechanical_Turk in that the machine \mathcal{V} does not need the smarts, it is the person, or the demon, who provides that.

strategy for chess Chess is known to be a solvable game. This is Zermelo's Theorem (Wikipedia

contributors 2017m) — there is a strategy for one of the two players that forces a win or a draw, no matter how the opponent plays

a deterministic verifier must take exponential time In fact, in the terminology of a later section, chess is known to be *EXP* complete. See (Fraenkel and Lichtenstein 1981).

in a sense, useless Being given an answer with no accompanying justification is a problem. This is like the Feynman algorithm for doing Physics: “The student asks . . . what are Feynman’s methods? [M] Gell-Mann leans coyly against the blackboard and says: Dick’s method is this. You write down the problem. You think very hard. (He shuts his eyes and presses his knuckles parodically to his forehead.) Then you write down the answer.” (Gleick 1992) It is also like the mathematician S Ramanujan, who relayed that the advanced formulas that he produced came in dreams from the god Narasimha. Some of these formulas were startling and amazing, but some of them were wrong. (Chakrabarty 2017) Another such story has to do with G Hardy about to board a ferry crossing rough seas from Denmark to Britain. He sent a postcard to another mathematician stating that he had solved the Riemann hypothesis (this is still one of the most famous unproven hypothesis in mathematics). And of course the most famous example of a failure to provide backing is Fermat writing in a book he was reading that that there are no nontrivial instances of $x^n + y^n = z^n$ for $n > 2$ and then saying, “I have discovered a truly marvelous proof of this, which this margin is too narrow to contain.”

the verifier cannot even input them before its runtime bound expires Some authors instead define that the verifier runs in time polynomial in its input, $\langle \sigma, \omega \rangle$, and add the explicit restriction that $|\omega|$ must be polynomial in $|\sigma|$.

How is that legal? This is reminiscent of Quantum Bogosort, a facetious sorting algorithm. Given an unordered list of length n , it uses a quantum source of randomness to generate a permutation of n . It reorders the input according to that permutation. If the list is now sorted then good. If not then the algorithm destroys the entire universe. Assuming the Many World’s Hypothesis, the result is that in any surviving universe the list has been sorted in linear time.

Countdown For a brief description see [https://en.wikipedia.org/wiki/Countdown_\(game_show\)](https://en.wikipedia.org/wiki/Countdown_(game_show)). A version that you may find fun, worth searching for the videos, is https://en.wikipedia.org/wiki/8_Out_of_10_Cats_Does_Countdown.

many-one reducible The name comes from the fact that still another reducibility is one-one reducibility, where the function must be one-to-one.

that’s not true under Karp reduction The Halting problem set K and its complement are not Karp reducible. For, we already know that K is computably enumerable. If $K^c \leq_p K$ then $x \in K^c$ implies that $f(x) \in K$, and we can enumerate $f(0), f(1), \dots$ and check those against the values enumerated into K , so we would have that K^c is also computably enumerable. That would imply that K is computable, which it is not.

the Petersen graph The Petersen graph is a rich source of counterexamples for conjectures in Graph Theory

Drummer problem This is often called the *Marriage* problem, where the men pick suitable women. But perhaps it is time for a new paradigm.

Asymmetric Traveling Salesman (Jonker and Volgenent 1983)

NP complete The name is from a survey created by Knuth. See blog.computationalcomplexity.org/2010/11/by-any-other-name-would-be-just-as-hard.html.

there are many such problems The “at least as hard” is true in the sense that such problems can answer questions about any other problem in that class. However, note that it might be that one *NP* complete problem runs in nondeterministic time that is $\mathcal{O}(n)$ while another runs in $\mathcal{O}(n^{1000000})$ time. So this sense is at odds with our earlier characterization of problems that are harder to solve.

*The list below gives the *NP* complete problems most often used* These are from the classic standard reference (Garey and Johnson 1979).

a gadget See <https://cs.stackexchange.com/a/1249/50343> from the Computer Science Stack Exchange user JeffE.

tied to whether $P = NP$ or $P \neq NP$ Ladner’s theorem is that if $P \neq NP$ then there is a problem in $NP - P$ that is not *NP* complete.

A large class See (Karp 1972).

an ending point That is, as Pudlák observes, we treat $P \neq NP$ as an informal axiom. (Pudlák 2013)

caricature Paul Erdős joked that a mathematician is a machine for turning coffee into theorems.

completely within the realm of possibility that $\phi(n)$ grows that slowly Hartmanis observes (Hartmanis 2017) that it is interesting that Gödel, the person who destroyed Hilbert’s program of automating mathematics, seemed to think that these problems quite possibly are solvable in linear or quadratic time.

In 2018, a poll The poll was conducted by W Gasarch, a prominent researcher and blogger in Computational Complexity. There were 124 respondents. For the description see <https://www.cs.umd.edu/users/gasarch/BLOGPAPERS/pollpaper3.pdf>. Note the suggestions that both respondents and even the surveyor took the enterprise in a light-hearted way.

88% thought that $P \neq NP$ Gasarch divided respondents into experts, the people who are known to have seriously thought about the problem, and the masses. The experts were 99% for $P \neq NP$.

S Aaronson has said See (Roberts 2021) for both the Aaronson and Williams estimates.

A Wigderson See (Wigderson 2009).

Cook is of the same mind See (S. Cook 2000).

Many observers For example, (Viola 2018).

$\mathcal{O}(n^{\lg 7})$ method ($\lg 7 \approx 2.81$) Strassen’s algorithm is used in practice. The current record is $\mathcal{O}(n^{2.37})$ but it is not practical. It is a galactic algorithm because while runs faster than any other known algorithm when the problem is sufficiently large, but the first such problem is so big that we never use the algorithm. For other examples see (Wikipedia contributors 2020b).

Matching problem The Drummer problem described earlier is a special case of this for bipartite graphs.

more things to try than atoms in the universe There are about 10^{80} atoms in the universe. A graph with 100 vertices has the potential for $\binom{100}{2}$ edges, which is about 100^2 . Trying every edge would be $2^{10000} \approx 10^{10000/3.32}$ cases, which is much greater than 10^{80} .

since the 1960’s we have an algorithm Due to J Edmonds.

Theory of Computing blog feed (Various authors 2017)

R J Lipton captured this feeling (Lipton 2009)

Knuth has a related but somewhat different take (Knuth 2014)

all this is speculation Arthur C Clarke's celebrated First Law is, "When a distinguished but elderly scientist states that something is possible, he is almost certainly right. When he states that something is impossible, he is very probably wrong." (Wikipedia contributors 2023)

exploits this difference Recent versions of the algorithm used in practice incorporate refinements that we shall not discuss. The core idea is unchanged.

Their algorithm, called RSA Originally the authors were listed in the standard alphabetic order: Adleman, Rivest, and Shamir. Adleman objected that he had not done enough work to be listed first and insisted on being listed last. He said later, "I remember thinking that this is probably the least interesting paper I will ever write."

tremendous amount of interest and excitement In his 1977 column, Martin Gardner posed a \$100 challenge, to crack this message: 9686 9613 7546 2206 1477 1409 2225 4355 8829 0575 9991 1245 7431 9874 6951 2093 0816 2982 2514 5708 3569 3147 6622 8839 8962 8013 3919 9055 1829 9451 5781 5254 The ciphertext was generated by the MIT team from a plaintext (English) message using $e = 9007$ and this number n (which is too long to fit on one line).

114, 381, 625, 757, 888, 867, 669, 235, 779, 976, 146, 612, 010, 218, 296, 721, 242,
362, 562, 561, 842, 935, 706, 935, 245, 733, 897, 830, 597, 123, 563, 958, 705,
058, 989, 075, 147, 599, 290, 026, 879, 543, 541

In 1994, a team of about 600 volunteers announced that they had factored n .

$p = 3, 490, 529, 510, 847, 650, 949, 147, 849, 619, 903, 898, 133, 417, 764,$
 $638, 493, 387, 843, 990, 820, 577$

and

$q = 32, 769, 132, 993, 266, 709, 549, 961, 988, 190, 834, 461, 413, 177, 642, 967,$
992, 942, 539, 798, 288, 533

That enabled them to decrypt the message: *the magic words are squeamish ossifage*.

based on the next result It is called Fermat's Little Theorem in contrast with his celebrated assertion that $a^n + b^n = c^n$ for $n > 2$.

computer searches suggest that these are very rare Among the numbers less than 2.5×10^{10} there are only 21 853 $\approx 2.2 \times 10^4$ pseudoprimes base 2. That's six orders of magnitude less.

a greater than $1 - (1/2)^k$ chance that n is prime Here is the probability $1 - (1/2)^k$ for the first few k 's.

k	Chance n is prime
1	0.500 000 000
2	0.750 000 000
3	0.875 000 000
4	0.937 500 000
5	0.968 750 000
6	0.984 375 000
7	0.992 187 500
8	0.996 093 750
9	0.998 046 875

We get an extra decimal place of certainty about every $3\frac{1}{3}$ iterations because $\lg(10) \approx 3.32$. So if you want, say, five decimal places, so that you have at least a probability of 0.999 99, then it is safe to iterate $4 \cdot 5 = 20$ times.

any reasonable-sized k Selecting an appropriate k is an engineering choice between the cost of extra iterations and the gain in confidence.

we are quite confident that it is prime We are confident, but not certain. There are numbers, called Carmichael numbers, that are pseudoprime for every base a relatively prime to n . The smallest example is $n = 561 = 3 \cdot 11 \cdot 17$, and the next two are 1 105 and 1 729. Like pseudoprimes, these seem to be very rare. Among the numbers less than 10^{16} there are 279 238 341 033 922 primes, about 2.7×10^{14} , but only 246 683 $\approx 2.4 \times 10^5$ -many Carmichael numbers.

the minimal pub crawl See (W. Cook et al. 2017).

An example is that the Free mathematics system Sage includes one See also <https://www.youtube.com/watch?v=q8nQTNvCrjE> about the Concorde TSP solver.

the Sudoku problem is NP complete First proved in the MS thesis of Takayuki Yato, from the Department of Information Science at the University of Tokyo in 1987. That document seems to have disappeared from the web; for a place to start see the Sudoku Wikipedia page.

Appendices

empty string, denoted ϵ Possibly ϵ came as an abbreviation for ‘empty’. Some authors use λ , possibly from the German word for ‘empty’, *leer*. Or it might just be that someone used the symbols just because one was needed; the story goes that when asked why he used the λ symbol for his λ calculus, Church replied, “enie, meenie, meinie, mo” (Stack Exchange author Jouni Sirén 2016)

reversal σ^R of a string The most practical current notion of a string, the Unicode standard, does not have string reversal. All of the naive ways to reverse a string run into problems for arbitrary Unicode strings which may contain non-ASCII characters, combining characters, ligatures, bidirectional text in multiple languages, and so on. For example, merely reversing the chars (the Unicode scalar values) in a string can cause combining marks to become attached to the wrong characters. Another example is: how to reverse ab<backspace>ab? The Unicode Consortium has not gone through the effort to define the reverse of a string because there is no real-world need for it. (From <https://qntm.org/trick>.)

Credits

Prologue

- I.1.11 SE user Shuzheng, <https://cs.stackexchange.com/q/45589/50343>
- I.1.12 Question by SE user Arsalan MGR, <https://cs.stackexchange.com/q/135343/50343>
- I.2.9 SE user Yuval Filmus, <https://cs.stackexchange.com/a/135170/50343>
- I.2.13 <http://www.ivanociardelli.altervista.org/wp-content/uploads/2016/09/Solutions-to-exercises.pdf>

Background

- II.2 Image credit: Robert Williams and the Hubble Deep Field Team (STScI) and NASA.
- II. Image credit File:Galilee.jpg. (2018, September 27). *Wikimedia Commons, the free media repository*. Retrieved 22:19, January 26, 2020 from <https://commons.wikimedia.org/w/index.php?title=File:Galilee.jpg&oldid=322065651>.
- II.3.17 User scherk at pbworks.com.
- II.3.27 Michael J Neely
- II.3.29 Answer from Stack Exchange member Alex Becker.
- II.4.1 ENIAC Programmers, 1946 U. S. Army Photo from Army Research Labs Technical Library
- II.4.6 Started on Stack Exchange
- II.4.9 From a Stack Exchange question.
- II.5.12 CS SE user Kyle Strand <https://cs.stackexchange.com/q/11645/50343>.
- II.5.13 SE user npostavs, <https://cs.stackexchange.com/a/44875/50343>
- II.5.33 SE user Raphael <https://cs.stackexchange.com/a/44901/50343>
- II.6.10 Question by SE user MathematicalOrchid, <https://cs.stackexchange.com/q/2811/67754>, and answer by SE user Andrej Bauer.
- II.6.29 SE user Rajesh R
- II.8.12 <http://people.cs.aau.dk/~srba/courses/tutorials-CC-10/t5-sol.pdf>
- II.8.14 SE user Karolis Juodelė
- II.8.17 SE user Noah Schweber
- II.9.9 (Rogers 1987), p 214.
- II.9.11 (Rogers 1987), p 214.
- II.9.14 (Rogers 1987), p 214.
- II.A.1 <https://www.ias.edu/ideas/2016/pires-hilbert-hotel>
- II.C.1 <https://research.swtch.com/zip> and Kevin Matulef
- II.C.2 http://en.wikipedia.org/wiki/Quine_%28computing%29

Languages

- III.1.25 F Stephan, <https://www.comp.nus.edu.sg/~fstephan/toc01slides.pdf>
- III.1.36 SE user babou
- III.2.9 SE user Rick Decker
- III.2.16 <http://www.cs.utsa.edu/~wagner/CS3723/grammar/examples.html>

- III.2.19 (Hopcroft, Motwani, and Ullman 2001), exercise 5.1.2.
- III.2.32 Wikipedia contributors, https://en.wikipedia.org/wiki/Buffalo_buffalo_Buffalo_buffalo_buffalo_Buffalo_buffalo, William J. Rapaport, <https://cse.buffalo.edu/~rapaport/BuffaloBuffalo/buffalobuffalo.html>
- III.2.36 <http://www.cs.utsa.edu/~wagner/CS3723/grammar/examples.html>
- III.3.17 SE user DollarAkshay
- III.3.24 T Zaremba, <http://www.geom.uiuc.edu/~zaremba/graph3.html>.
- III.A.9 <http://people.cs.ksu.edu/~schmidt/300s05/Lectures/GrammarNotes/bnf.html>

Automata

- IV.1.44 From *Introduction to Languages* by Martin, edition four, p 77.
- IV.4.23 (Rich 2008), <https://math.stackexchange.com/a/1102627>
- IV.4.25 SE user jmite, <https://cs.stackexchange.com/a/67317/50343>.
- IV.4.27 (Rich 2008)
- IV.5.19 SE user David Richerby, <https://cs.stackexchange.com/a/97885/67754>
- IV.5.23 (Rich 2008)
- IV.5.30 SE user Brian M Scott, <https://math.stackexchange.com/a/1508488>
- IV.5.31 <https://cs.stackexchange.com/a/30726>
- IV.6.16 <https://www.eecs.wsu.edu/~cook/tcs/l10.html>

Complexity

- V. Some of the discussion is from <https://softwareengineering.stackexchange.com/a/20833>.
- V. Discussion of the third point started as <https://cs.stackexchange.com/questions/9957/justification-for-neglecting-constants-in-big-o>.
- V. The fourth point derives from <https://stackoverflow.com/a/19647659>.
- V. This discussion originated as (Stack Exchange author templatetypedef 2013).
- V.1.55 Stack Exchange user Daniel Fischer, <https://math.stackexchange.com/a/674039>, and Stack Exchange user anon, <https://math.stackexchange.com/a/61741>
- V.1.60 Stack Exchange user Ilmari Karonen, <https://math.stackexchange.com/questions/925053/using-limits-to-determine-big-o-big-omega-and-big-theta>
- V.2.24 Sean McCulloch, <https://npcomplete.owu.edu/2014/06/03/3-dimensional-matching/>
- V.2.53 Private communication from Puck Rombach.
- V.2.67 Jan Verschelde, <http://homepages.math.uic.edu/~jan/mcs401/partitioning.pdf>
- V.3.11 A.A. at <https://rjlipton.wordpress.com/2010/11/07/what-is-a-complexity-class/#comment-8872>
- V.4.16 <https://cs.stackexchange.com/q/57518>
- V.5.19 Paul Black, <https://xlinux.nist.gov/dads/HTML/nondetermAlgo.html>
- V.6.27 SE user JesusIsLord at <https://cstheory.stackexchange.com/a/47031/4731>
- V.6.29 SE user user326210, <https://math.stackexchange.com/a/2564255>
- V.6.33 Neal E Young, University of California Riverside
- V. By Psyon (Own work) CC BY-SA 3.0 https://commons.wikimedia.org/wiki/File:Jigsaw_Puzzle.svg
- V.7.12 William Gasarch, <https://www.cs.umd.edu/~gasarch/COURSES/452/F14/poly.pdf>
- V.7.16 <http://www.cs.princeton.edu/courses/archive/fall02/cos126/exercises/np.html>

V.7.17 Kevin Wayne. <http://www.cs.princeton.edu/courses/archive/fall02/cos126/exercises/np-sol.html>

V.7.20 <http://www.cs.princeton.edu/courses/archive/fall02/cos126/exercises/np.html>

V.7.23 Y Lyuu, <https://www.csie.ntu.edu.tw/~lyuu/complexity/2016/20161129s.pdf>

V.7.28 SE user Yuval Filmus <https://cs.stackexchange.com/a/132902/50343>

V.8.17 SE user Yuval Filmus <https://cs.stackexchange.com/a/54452/50343>

Bibliography

- A/V Geeks, Y. user, ed. (2013). *Slide Rule - Proportion, Percentage, Squares And Square Roots (1944)*. Division of Visual Aids, US Office of Education. URL: <https://www.youtube.com/watch?v=dT7bSn03lx0> (visited on 08/09/2015).
- Aaronson, S. (July 21, 2011). *Rosser's Theorem via Turing machines*. URL: <http://www.scottaaronson.com/blog/?p=710> (visited on 12/31/2023).
- (May 3, 2012a). *The 8000th Busy Beaver number eludes ZF set theory: new paper by Adam Yedidia and me*. URL: <http://www.scottaaronson.com/blog/?p=2725>.
- (Aug. 30, 2012b). *The Toaster-Enhanced Turing Machine*. URL: <http://www.scottaaronson.com/blog/?p=1121> (visited on 05/28/2015).
- Adams, D. (1979). *The Hitchhiker's Guide to the Galaxy*. Harmony Books. ISBN: 9780345391803.
- Allender, E., M. C. Loui, and K. W. Regan (1997). "Complexity Classes". In: ed. by M. J. Atallah and M. Blanton. Boca Raton, Florida: CRC Press. Chap. 27.
- Bellos, A. (Dec. 15, 2014). "The Game of Life: a beginner's guide". In: *The Guardian*. URL: <http://www.theguardian.com/science/alexs-adventures-in-numberland/2014/dec/15/the-game-of-life-a-beginners-guide> (visited on 07/14/2015).
- Bernstein, E. and U. Vazirani (1997). "Quantum Complexity Theory". In: *SIAM Journal of Computing* 26.5, pp. 1411–1473.
- Bigham, D. S. (Aug. 19, 2014). *How Many Vowels Are There in English? (Hint: It's More Than AEIOU.)*. Slate. URL: http://www.slate.com/blogs/lexicon_valley/2014/08/19/aeiou_and_sometimes_y_how_many_english_vowels_and_what_is_a_vowel_anyway.html (visited on 06/12/2017).
- Black, R. (2000). "Proving Church's Thesis". In: *Philosophia Mathematica* 8, pp. 244–258.
- Blanda, S. (2013). *The Six Degrees of Kevin Bacon*. [Online; accessed 2019-Apr-01]. URL: <https://blogs.ams.org/mathgradblog/2013/11/22/degrees-kevin-bacon/>.
- Bragg, M. (Sept. 2016). *Zeno's Paradoxes*. Podcast. Guests: Marcus du Sautoy, Barbara Sattler, and James Warren. British Broadcasting Corporation. URL: <https://www.bbc.co.uk/programmes/b07vs3v1>.
- Brock, D. C. (2020). *Discovering Dennis Ritchie's Lost Dissertation*. [Online; accessed 2020-Jun-20]. URL: <https://computerhistory.org/blog/discovering-dennis-ritchie-s-lost-dissertation/>.
- Brower, K. (1983). *The Starship and the Canoe*. Harper Perennial; Reprint edition. ISBN: 978-0060910303.
- Bruck, R. H. (1953). "Computational Aspects of Certain Combinatorial Problems". In: *AMS Symposium in Applied Mathematics* 6, p. 31.
- Chakrabarty, R. (Apr. 26, 2017). "Srinivasa Ramanujan: The mathematical genius who credited his 3900 formulae to visions from Goddess Mahalakshmi". In: *India Today*. URL: <https://www.indiatoday.in/education-today/gk-current-affairs/story/srinivasa-ramanujan-life-story-973662-2017-04-26> (visited on 11/27/2020).
- Church, A. (1937). "Review of Alan M. Turing, On computable numbers, with an application to the Entscheidungsproblem". In: *Journal of Symbolic Logic* 2, pp. 42–43.
- Cobham, A. (1965). "The intrinsic computational difficulty of functions". In: *Logic, Methodology and*

- Philosophy of Science: Proceedings of the 1964 International Congress*. Ed. by Y. Bar-Hillel. North-Holland Publishing Company, pp. 24–30.
- Cook, S. (2000). *The P vs NP Problem*. Official problem description. Clay Mathematics Institute. URL: <https://www.claymath.org/sites/default/files/pvsn.pdf> (visited on 01/11/2018).
- Cook, W. et al. (2017). *UK Pubs Travelling Salesman Problem*. URL: <http://www.math.uwaterloo.ca/tsp/pubs/index.html> (visited on 12/16/2017).
- Copeland, B. J. and D. Proudfoot (1999). “Alan Turing’s Forgotten Ideas in Computer Science”. In: *Scientific American* 280.4, pp. 99–103.
- Copeland, B. J. (Sept. 1996). “What is Computation?” In: *Computation, Cognition and AI*, pp. 335–359.
- (1999). “Beyond the universal Turing machine”. In: *Australasian Journal of Philosophy* 77.1, pp. 46–67.
 - (Aug. 19, 2002). *The Church-Turing Thesis; Misunderstandings of the Thesis*. URL: <http://plato.stanford.edu/entries/church-turing/#Bloopers> (visited on 01/07/2016).
- Cox, R. (2007). *Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)*. URL: <https://swtch.com/~rsc/regexp/regexp1.html> (visited on 06/29/2019).
- Davis, M. (2004). “The Myth of Hypercomputation”. In: *Alan Turing: Life and Legacy of a Great Thinker*. Ed. by C. Teuscher. Springer, pp. 195–211. ISBN: ISBN 978-3-662-05642-4.
- (2006). “Why there is no such discipline as hypercomputation”. In: *Applied Mathematics and Computation* 178, pp. 4–7.
- Dershowitz, N. and Y. Gurevich (Sept. 2008). “A Natural Axiomatization of Computability and Proof of Church’s Thesis”. In: *Bulletin of Symbolic Logic* 14.3, pp. 299–350.
- Edmunds, J. (1965). “Paths, trees, and flowers”. In: *Canadian Journal of Mathematics* 17, pp. 449–467.
- Eén, N. and N. Sörensson (2005). *MiniSat*. URL: <http://minisat.se/> (visited on 05/16/2022).
- Encyclopædia Britannica Editors (2017). *Y2K bug*. URL: <https://www.britannica.com/technology/Y2K-bug> (visited on 05/10/2017).
- Euler, L. (1766). “Solution d’une question curieuse que ne paroît soumise a aucune analyse (Solution of a curious question which does not seem to have been subjected to any analysis)”. In: *Mémoires de l’Academie Royale des Sciences et Belles Lettres, Année 1759* 15. [Online; accessed 2017-Sep-23, article 309], pp. 310–337. URL: <http://eulerarchive.maa.org/>.
- Firth, N. (Oct. 14, 2009). “Sir Tim Berners-Lee admits the forward slashes in every web address ‘were a mistake’”. In: *Daily Mail*. URL: <https://www.dailymail.co.uk/sciencetech/article-1220286/Sir-Tim-Berners-Lee-admits-forward-slashes-web-address-mistake.html> (visited on 11/29/2018).
- Fortnow, L. and B. Gasarch (2002). *Computational Complexity Blog*. [Online; accessed 2017-Nov-13]. URL: <http://blog.computationalcomplexity.org/2002/11/foundations-of-complexitylesson-7.html>.
- Fraenkel, A. S. and D. Lichtenstein (1981). “Computing a Perfect Strategy for $n \times n$ Chess Requires Time Exponential in n ”. In: *Journal Of Combinatorial Theory, Series A*, pp. 199–214.
- Free Online Dictionary of Computing (Denis Howe) (2017). *Stephen Kleene*. [Online; accessed 21-June-2017]. URL: <http://foldoc.org/Stephen%20Kleene>.
- Gandy, R. (1980). “Church’s Thesis and Principles for Mechanisms”. In: *The Kleene Symposium*. Ed. by J. Barwise, H. J. Keisler, and K. Kunen. North-Holland Amsterdam, pp. 123–148. ISBN: 978-0-444-85345-5.
- Gardner, M. (Oct. 1970). “Mathematical Games: The fantastic combinations of John Conway’s new solitaire game ‘life’”. In: *Scientific American* 223, pp. 120–123. URL:

- <http://www.ibiblio.org/lifePatterns/october1970.html>.
- Garey, M. and D. Johnson (1979). *Computers and Intractability, A Guide to the Theory of NP Completeness*. W. H. Freeman.
- Gizmodo (1948). *UCLA's 1948 Mechanical Computer*. Accessed 2019-September-18. URL: <https://vimeo.com/70589461>.
- Gleick, J. (Sept. 20, 1992). "Part Showman, All Genius". In: *New York Times Magazine*. URL: <https://www.nytimes.com/1992/09/20/magazine/part-showman-all-genius.html> (visited on 11/27/2020).
- Gödel, K. (1964). "What is Cantor's Continuum Problem?" In: *Philosophy of Mathematics: Selected Readings*. Ed. by P. Benacerraf and H. Putnam. Cambridge University Press, pp. 470–494.
- (1995). "Undecidable diophantine propositions". In: *Collected works Volume III: Unpublished essays and lectures*. Ed. by S. F. et al. Oxford University Press.
- Goodstein, R. L. (Dec. 1947). "Transfinite Ordinals in Recursive Number Theory". In: *Journal of Symbolic Logic* 12.4, pp. 123–129.
- Grossman, L. (2010). *Metric Math Mistake Muffed Mars Mereorology Mission*. [Online; accessed 2017-May-25]. URL: <https://www.wired.com/2010/11/1110mars-climate-observer-report/>.
- Hartmanis, J. (2017). *Gödel, von Neumann and the P=?NP Problem*. URL: <http://www.cs.cmu.edu/~15455/hartmanis-on-godel-von-neumann.pdf> (visited on 12/25/2017).
- Hennie, F. (1977). *Introduction to Computability*. Addison-Wesley. ISBN: 978-0201028485.
- Hilbert, D. and W. Ackermann (1950). *Principles of Mathematical Logic*. Trans. by R. E. Luce. AMS Chelsea Publishing. ISBN: 978-0821820247.
- Hodges, A. (1983). *Alan Turing: the enigma*. Simon and Schuster. ISBN: 0-671-49207-1.
- (2016). *Alan Turing in the Stanford Encyclopedia of Philosophy*. URL: <http://www.turing.org.uk/publications/stanford.html> (visited on 04/06/2016).
- Hofstadter, D. R. (1979). *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books. ISBN: 978-0465026562.
- Hopcroft, J. E., R. Motwani, and J. D. Ullman (2001). *Introduction to Automata Theory, Languages, and Computation*. 2nd ed. Pearson Education. ISBN: 0201441241.
- Huggett, N. (2010). *Zeno's Paradoxes — Stanford Encyclopedia of Philosophy*. [Online; accessed 23-Dec-2016]. URL: <https://plato.stanford.edu/entries/paradox-zeno/#ParMot>.
- Jones, N. D. (1997). *Computability and Complexity From a Programming Perspective*. 1st ed. MIT Press. ISBN: 978-0262100649.
- Jonker, R. and T. Volgenent (Nov. 1, 1983). "Transforming Asymmetric into Symmetric Traveling Salesman Problems". In: *Operations Research Letters* 2.4, pp. 161–163.
- Karp, R. M. (1972). "Reducibility Among Combinatorial Problems". In: ed. by R. E. Miller and J. W. Thatcher. New York: Plenum, pp. 85–103. URL: <https://www.loc.gov/resource/cph.3c10471/> (visited on 12/21/2017).
- Kleene, S. (1952). *Introduction to Metamathematics*. North-Holland Amsterdam. ISBN: 978-0923891572.
- Klyne, G. and C. Newman (July 2002). *Date and Time on the Internet: Timestamps*. RFC 3339. RFC Editor, pp. 1–18. URL: <https://www.ietf.org/rfc/rfc3339.txt>.
- Knuth, D. E. (Dec. 1964). "Backus Normal Form vs. Backus Naur Form". In: *Communications of the ACM* 7.12, pp. 735–736.
- (May 20, 2014). *Twenty Questions for Donald Knuth*. URL: <http://www.informit.com/articles/article.aspx?p=2213858> (visited on 02/17/2018).
- Knuutila, T. (2001). "Redescribing an algorithm by Hopcroft". In: *Theoretical Computer Science* 250,

- pp. 333–363.
- Kragh, H. (Mar. 27, 2014). *The True (?) Story of Hilbert's Infinite Hotel*. URL: <http://arxiv.org/abs/1403.0059>.
- Leupold, J. (1725). “Details of the mechanisms of the Leibniz calculator, the most advanced of its time”. In: Illustration in: Theatrum arithmeticо-geometricum, das ist . . . [bound with Theatrum machinarium, oder, Schau-Platz der Heb-Zeuge/Jacob Leupold. Leipzig, 1725]. Leipzig: Zufinden bey dem Autore und Joh. Friedr. Gleditschens seel. Sohn: Gedruckt bey Christoph Zunkel, 1727. URL: <https://www.loc.gov/resource/cph.3c10471/> (visited on 11/14/2016).
- Levin, L. A. (Dec. 7, 2016). *Fundamentals of Computing*. URL: <https://www.cs.bu.edu/fac/lnd/toc/>.
- Lipton, R. J. (Sept. 22, 2009). *It's All Algorithms, Algorithms and Algorithms*. URL: <https://rjlipton.wordpress.com/2009/09/22/its-all-algorithms-algorithms-and-algorithms/> (visited on 02/17/2018).
- Maienschein, J. (2017). “Epigenesis and Preformationism”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by E. N. Zalta. Spring 2017. Metaphysics Research Lab, Stanford University.
- MathOverflow user Joel David Hamkins (2010). *Answer to: Infinite CPU clock rate and hotel Hilbert*. URL: <https://mathoverflow.net/a/22038> (visited on 04/19/2017).
- McCarthy, J. (1963). *A Basis for a Mathematical Theory of Computation*. URL: <http://www-formal.stanford.edu/jmc/basis1.pdf> (visited on 06/15/2017).
- Meyer, A. R. and D. M. Ritchie (1966). *Research report: The complexity of loop programs*. Tech. rep. 1817. IBM.
- N. J. A. Sloane, e. (2019). *The On-Line Encyclopedia of Integer Sequences*, A000290. URL: <https://oeis.org/A000290> (visited on 03/02/2019).
- Odifreddi, P. (1992). *Clasical Recursion Theory*. Elsevier Science. ISBN: 0-444-87295-7.
- Perlis, A. J. (Sept. 1, 1982). “Epigrams on Programming”. In: *SIGPLAN Notices* 17.9, pp. 7–13. URL: <https://web.archive.org/web/19990117034445/http://www-pu.informatik.uni-tuebingen.de/users/klaeren/epigrams.html> (visited on 12/23/2023).
- Peters, T. (2023). *Timsort*. URL: <https://bugs.python.org/file4451/timsort.txt> (visited on 01/14/2023).
- Piccinini, G. (2017). “Computation in Physical Systems”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by E. N. Zalta. Summer 2017. Metaphysics Research Lab, Stanford University.
- Pinker, S. (Sept. 4, 2014). *The Trouble With Harvard*. URL: <https://newrepublic.com/article/119321/harvard-ivy-league-should-judge-students-standardized-tests> (visited on 12/23/2020).
- Pour-El, M. B. and I. Richards (1981). “The wave equation with computable initial data such that its unique solution is not computable”. In: *Adv. in Math* 39, pp. 215–239.
- Pseudonym, S. E. author (2014). *Answer to: What exactly is an algorithm?* URL: <https://cs.stackexchange.com/a/31953> (visited on 12/27/2018).
- Pudlák, P. (2013). *Logical Foundations of Mathematics and Computational Complexity*. Springer. ISBN: 978-3-319-34268-9.
- Radó, T. (May 1962). “On Non-computable Functions”. In: *Bell Systems Technical Journal*, pp. 877–884. URL: <https://ia601900.us.archive.org/0/items/bstj41-3-877/bstj41-3-877.pdf>.
- Rendell, P. (2011). <http://rendell-attic.org/gol/tm.htm>. URL: <http://rendell-attic.org/gol/tm.htm> (visited on 07/21/2015).
- Renwick, W. S. (May 6, 1949). *The start of the EDSAC log*. [Online; accessed 2019-Mar-02]. URL: <https://www.cl.cam.ac.uk/relics/elog.html>.
- Rich, E. (2008). *Automata, Computability, and Complexity*. Pearson. ISBN: 978-0-13-228806-4.

- Roberts, S. (Oct. 27, 2021). “The 50-year-old problem that eludes theoretical computer science”. In: *MIT Technology Review*.
- Robinson, R. (1948). “Recursion and Double Recursion”. In: *Bulletin of the American Mathematical Society* 10, pp. 987–993.
- Rogers Jr., H. (Sept. 1958). “Gödel numberings of partial recursive functions”. In: *Journal of Symbolic Logic* 23.3, pp. 331–341.
- (1987). *Theory of Recursive Functions and Effective Computability*. MIT Press. ISBN: 0-262-68052-1.
- SE author Brian M. Scott (Feb. 14, 2020). *Inverting the Cantor pairing function*. URL: <http://math.stackexchange.com/q/222835> (visited on 10/28/2012).
- Smoryński, C. (1991). *Logical Number Theory I*. Springer-Verlag. ISBN: 978-3540522362.
- Soare, R. I. (1999). “Computability and Incomputability”. In: *Handbook of Computability Theory*. Ed. by E. R. Griffor. North-Holland, Amsterdam, pp. 3–36.
- Stack Exchange author Andrej Bauer (2016). *Answer to: Is a Turing Machine “by definition” the most powerful machine?* [Online; accessed 2017-Nov-05]. Stack Overflow discussion board. URL: <https://cs.stackexchange.com/a/66753/78536>.
- (2018). *Answer to: Problems understanding proof of smn theorem using Church-Turing thesis*. [Online; accessed 2020-Feb-13]. Stack Overflow discussion board. URL: <https://cs.stackexchange.com/a/97946/67754>.
- Stack Exchange author babou and various others (2015). *Justification for neglecting constants in Big O*. [Online; accessed 2017-Oct-29]. Computer Science Stack Exchange discussion board. URL: <https://cs.stackexchange.com/a/41000/78536>.
- Stack Exchange author bobnice (2009). *Answer to: RegEx match open tags except XHTML self-contained tags*. URL: <https://stackoverflow.com/a/1732454/7168267> (visited on 01/27/2019).
- Stack Exchange author David Richerby (2018). *Why is there no permutation in Regexes? (Even if regular languages seem to be able to do this)*. [Online; accessed 2020-Jan-01]. Stack Overflow discussion board. URL: <https://cs.stackexchange.com/a/100215/67754>.
- Stack Exchange author JohnL (2020). *How to decide whether a language is decidable when not involving turing machines?* [Online; accessed 2020-Jun-11]. Computer Science Stack Exchange discussion board. URL: <https://cs.stackexchange.com/a/127035/67754>.
- Stack Exchange author Jouni Sirén (2016). *Answer to: What is the origin of λ for empty string?* Accessed 2016-October-20. URL: <http://cs.stackexchange.com/a/64850/50343>.
- Stack Exchange author Kaktus and various others (2019). *Georg Cantor’s diagonal argument, what exactly does it prove?* [Online; accessed 2019-Dec-25]. Computer Science Stack Exchange discussion board. URL: <https://math.stackexchange.com/q/2176304>.
- Stack Exchange author Ryan Williams (Sept. 2, 2010). *Comment to answer for What would it mean to disprove Church-Turing thesis?* URL: <https://cstheory.stackexchange.com/a/105/4731> (visited on 06/24/2019).
- Stack Exchange author templatetypedef (2013). *What is pseudopolynomial time? How does it differ from polynomial time?* [Online; accessed 2017-Oct-29]. Stack Overflow discussion board. URL: <https://stackoverflow.com/a/19647659>.
- Thompson, K. (Aug. 1984). “Reflections on trusting trust”. In: *Communications of the ACM* 27 (8), pp. 761–763.
- Thomson, J. F. (Oct. 1954). “Tasks and Super-Tasks”. In: *Analysis* 15.1, pp. 1–13.
- Turing, A. M. (1937). “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society*. 2nd ser. 42, pp. 230–265.
- (1938a). “On Computable Numbers, with an Application to the Entscheidungsproblem. A

- Correction.” In: *Proceedings of the London Mathematical Society*. 6th ser. 43, pp. 544–546.
- Turing, A. M. (1938b). “Systems of Logic Based on Ordinals”. PhD. Princeton University.
- U.S. Naval Observatory, T. S. D. (2017). *Leap Seconds*. [Online; accessed 10-May-2017]. URL: <http://tycho.usno.navy.mil/leapsec.html>.
- Various authors (2017). *Theory of Computing Blog Aggregator*. [Online; accessed 17-May-2017]. URL: <http://cstheory-feed.org/>.
- Viola, E. (Feb. 16, 2018). *I believe P=NP*. URL: <https://emanueleviola.wordpress.com/2018/02/16/i-believe-pnp/> (visited on 02/16/2018).
- Wigderson, A. (2009). “Knowledge, Creativity and P versus NP”. in: URL: <https://www.math.ias.edu/~avi/PUBLICATIONS/MYPAPERS/AW09/AW09.pdf> (visited on 06/10/2023).
- (2017). *Mathematics and Computation*. [Draft of a to-be-published book; accessed 2017-Oct-27]. URL: <https://www.math.ias.edu/avi/book>.
- Wikipedia contributors (2014). *History of the Church-Turing thesis — Wikipedia, The Free Encyclopedia*. [Online; accessed 2-October-2016]. URL: https://en.wikipedia.org/w/index.php?title=History_of_the_Church%20%93Turing_thesis&oldid=618643863.
- (2015). *Stigler's law of eponymy — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/w/index.php?title=Stigler%27s_law_of_eponymy&oldid=691378684 (visited on 02/14/2016).
 - (2016a). *Age of the Earth — Wikipedia, The Free Encyclopedia*. [Online; accessed 13-June-2016]. URL: https://en.wikipedia.org/w/index.php?title=Age_of_the_Earth&oldid=724796250.
 - (2016b). *Donald Michie — Wikipedia, The Free Encyclopedia*. [Online; accessed 24-March-2016]. URL: https://en.wikipedia.org/w/index.php?title=Donald_Michie&oldid=708156000 (visited on 03/24/2016).
 - (2016c). *Nomogram — Wikipedia, The Free Encyclopedia*. [Online; accessed 6-October-2016]. URL: <https://en.wikipedia.org/w/index.php?title=Nomogram&oldid=742964268>.
 - (2016d). *Ross-Littlewood paradox — Wikipedia, The Free Encyclopedia*. [Online; accessed 9-February-2017]. URL: https://en.wikipedia.org/w/index.php?title=Ross%20%93Littlewood_paradox&oldid=739534216.
 - (2016e). *The Imitation Game — Wikipedia, The Free Encyclopedia*. [Online; accessed 28-June-2016]. URL: https://en.wikipedia.org/w/index.php?title=The_Imitation_Game&oldid=723336480.
 - (2016f). *Turtles all the way down — Wikipedia, The Free Encyclopedia*. [Online; accessed 2016-September-04]. URL: https://en.wikipedia.org/w/index.php?title=Turtles_all_the_way_down&oldid=736001775.
 - (2016g). *Zeno's paradoxes — Wikipedia, The Free Encyclopedia*. [Online; accessed 23-December-2016]. URL: https://en.wikipedia.org/w/index.php?title=Zeno%27s_paradoxes&oldid=752685211%7D.
 - (2017a). *15 puzzle — Wikipedia, The Free Encyclopedia*. [Online; accessed 16-September-2017]. URL: https://en.wikipedia.org/w/index.php?title=15_puzzle&oldid=789930961.
 - (2017b). *Almon Brown Strowger — Wikipedia, The Free Encyclopedia*. [Online; accessed 9-June-2017]. URL: https://en.wikipedia.org/w/index.php?title=Almon_Brown_Strowger&oldid=783883144.
 - (2017c). *Artificial neuron — Wikipedia, The Free Encyclopedia*. [Online; accessed 21-June-2017]. URL: https://en.wikipedia.org/w/index.php?title=Artificial_neuron&oldid=780239713.

- (2017d). *Aubrey–Maturin series* — Wikipedia, The Free Encyclopedia. [Online; accessed 28-March-2017]. URL: https://en.wikipedia.org/w/index.php?title=Aubrey%20%93_Maturin_series&oldid=771937634.
- (2017e). *Backus–Naur form* — Wikipedia, The Free Encyclopedia. [Online; accessed 7-May-2017]. URL: https://en.wikipedia.org/w/index.php?title=Backus%20%93Naur_form&oldid=778354081.
- (2017f). *Magic smoke* — Wikipedia, The Free Encyclopedia. [Online; accessed 2017-October-11]. URL: https://en.wikipedia.org/w/index.php?title=Magic_smoke&oldid=785207817.
- (2017g). *North American Numbering Plan* — Wikipedia, The Free Encyclopedia. [Online; accessed 9-June-2017]. URL: https://en.wikipedia.org/w/index.php?title=North_American_Numbering_Plan&oldid=780178791.
- (2017h). *Ouija* — Wikipedia, The Free Encyclopedia. [Online; accessed 14-May-2017]. URL: <https://en.wikipedia.org/w/index.php?title=Ouija&oldid=776109372>.
- (2017i). *Pax Britannica* — Wikipedia, The Free Encyclopedia. [Online; accessed 14-May-2017]. URL: https://en.wikipedia.org/w/index.php?title=Pax_Britannica&oldid=775067301.
- (2017j). *Philipp von Jolly* — Wikipedia, The Free Encyclopedia. [Online; accessed 30-January-2019]. URL: https://en.wikipedia.org/w/index.php?title=Philipp_von_Jolly&oldid=764485788.
- (2017k). *Platonic solid* — Wikipedia, The Free Encyclopedia. [Online; accessed 2017-October-22]. URL: https://en.wikipedia.org/w/index.php?title=Platonic_solid&oldid=801264236.
- (2017l). *Unicode* — Wikipedia, The Free Encyclopedia. URL: <https://en.wikipedia.org/w/index.php?title=Unicode&oldid=784443067>.
- (2017m). *Zermelo's theorem (game theory)* — Wikipedia, The Free Encyclopedia. [Online; accessed 2017-Nov-26]. URL: [https://en.wikipedia.org/w/index.php?title=Zermelo%27s_theorem_\(game_theory\)&oldid=806070716](https://en.wikipedia.org/w/index.php?title=Zermelo%27s_theorem_(game_theory)&oldid=806070716).
- (2018). *Paradox* — Wikipedia, The Free Encyclopedia. [Online; accessed 14-December-2018]. URL: <https://en.wikipedia.org/w/index.php?title=Paradox&oldid=871193884>.
- (2019a). *Collatz conjecture* — Wikipedia, The Free Encyclopedia. [Online; accessed 15-February-2019].
- (2019b). *Mathematics: The Loss of Certainty* — Wikipedia, The Free Encyclopedia. [Online; accessed 30-January-2019]. URL: https://en.wikipedia.org/w/index.php?title=Mathematics:_The_Loss_of_Certainty&oldid=879406248.
- (2019c). *Maxwell's demon* — Wikipedia, The Free Encyclopedia. [Online; accessed 1-January-2020]. URL: https://en.wikipedia.org/w/index.php?title=Maxwell%27s_demon&oldid=930445803%7D.
- (2019d). *Partial application* — Wikipedia, The Free Encyclopedia. [Online; accessed 26-December-2019].
- (2020a). *Foobar* — Wikipedia, The Free Encyclopedia. [Online; accessed 2020-Feb-14]. URL: <https://en.wikipedia.org/w/index.php?title=Foobar&oldid=934819128>.
- (2020b). *Galactic algorithm* — Wikipedia, The Free Encyclopedia. [Online; accessed 2020-Jun-17]. URL: https://en.wikipedia.org/w/index.php?title=Galactic_algorithm&oldid=957279293.
- (2021). *Mississippi River Basin Model* — Wikipedia, The Free Encyclopedia. [Online; accessed 25-September-2022]. URL: https://en.wikipedia.org/w/index.php?title=Mississippi_River_Basin_Model&oldid=1041334010.

Wikipedia contributors (2023). *Clarke's three laws* — Wikipedia, The Free Encyclopedia. [Online; accessed 27-June-2023]. URL: https://en.wikipedia.org/w/index.php?title=Clarke%27s_three_laws&oldid=1156462008. YouTube user navyreviewer (2010). *Mechanical computer part 1*. URL: <https://www.youtube.com/watch?v=mpkTHyfr0pM> (visited on 08/09/2015).

Index

- +
 - in transition tables, 177
 - operation on a language, 214
- 3 Dimensional Matching problem, 328
- 3-Coloring problem, 329
- 3-SAT, *see* 3-Satisfiability problem
- 3-SAT, *see* 3-Satisfiability problem, *see also* 3-SAT problem, *see also* 3-SAT problem, *see also* 3-SAT problem, *see also* 3-SAT problem
- 3-Satisfiability problem, 276, 294, 328
- 3-SATproblem, 323, 340
- 3-SAT problem, 322, 323
- 3-Satisfiability
 - Strict variant, 322
- 4-Satisfiability problem, 339
- accept a language, 141
- accept an input, 181, 189, 194
- acceptable numbering, 69
- accepted language, *see* recognized of a Turing machine, 289
- accepting state, 13, 175, 177, 303
 - in transition tables, 177
- nondeterministic Finite State machine, 188
- Pushdown machine, 236
- accepts, 177, 181, 189, 194
- Ackermann function, 30–33, 36, 46–50
- Ackermann, W, 3
 - picture, 32
- action set, 8
- action symbol, 8
- addition, 6
- adjacency matrix, 158
- adjacent, 155, 156
- Adleman, L
 - picture, 346
- Agrawal, M
 - picture, 281
- AKS primality test, 282
- algorithm, 287
 - definition, 287
 - reliance on model, 288
- alphabet, 139, 362
 - input, 177
- Pushdown machine, 236
- tape, 8
- amb* function, 392
- ambiguous grammar, 148
- argument, to a function, 364
- Aristotle's Paradox, 57, 59
- Assignment problem, 324
- Asymmetric Traveling Salesman problem, 323, 324
- asymptotically equivalent, 263, 272
- atom, 282
- Backus, J
 - picture, 165
- Backus-Naur form, BNF, 165
- Bacon, Kevin, 399
- balanced parentheses, 234
- Berra, Y
 - picture, 186
- Big \mathcal{O} , 261
- Big Θ , 263
- bijection, 367
- binary sequence, 70
- binary tree, 173
- bit string, *see* bitstring
- bitstring, 362
- blank, 8, 303
- blank, B, 5
- BNF, 165–169
- body of a production, 144
- Bogosort, 401
- Boole, G
 - picture, 276
- boolean, 276
 - expression, 276
 - function, 276
 - variable, 276
- Boolean algebra, 370
- Boolean function, 371
- bottom, \perp , 234
- BPP, *Bounded-Error Probabilistic Polynomial Time* problem, 344
- branch
 - of the computation tree, 189
- breadth first traversal, 163
- breadth-first traversal, 170
- bridge, 294
- Brocard's problem, 97
- Brzozowski's algorithm, 232

Busy Beaver, 128–132
button
 start, 5

c.e. set, *see* computably enumerable
caching, 67
Cantor’s correspondence, 64–71
Cantor’s pairing function, 65
Cantor’s Theorem, 74
Cantor, G
 and diagonalization, 382
 picture, 59

cardinality, 57–78
 less than or equal to, 73

cellular automaton, 43
certificate, 307
Chromatic Number problem, 276, 290
chromatic number, 158
Church’s Thesis, 14–21
 and uncountability, 75
 argument by, 19
 clarity, 17
 consistency, 16
 convergence, 16
 coverage, 15
 Extended, 299

Church, A
 picture, 15
 Thesis, 15

circuit, 156, 297
 Euler, 156
 gate, 297
 Hamiltonian, 156
 wire, 297

Circuit Evaluation problem, 298

class, 140, 296
 complexity, 296

Class Scheduling problem, 331

Clique problem, 278, 301, 316, 328
clique, in a graph, 278
closed walk, 156
closure under an operation, 211
CNF, 152, 276, 322, 329, 354, 370
co-computably enumerable, 108
co-NP, 308
Cobham’s Thesis, 266
Cobham, A
 picture, 395

Thesis, 266
codomain, 364
codomain versus range, 365
Collatz conjecture, 95
coloring of a graph, 158
colors, 275
compiler-compiler, 167
complete, 112
 for a class, 327
 NP, 327

complete graph, 160

complexity class, 296
 canonical, 344
 EXP, 341
 NP, 306
 P, 297
 polytime, 297
complexity function, 261
Complexity Zoo, 297, 344
composition, 367
computable
 from a set, 110
 relative to a set, 110
 set, 104

computable function, 11

computable functions, 9–11

computable relation, 11

computable set, 11

computably enumerable, 103–108
 in an oracle, 116
 K is complete, 112

computably enumerable set, 104
 co-computably enumerable, 108
 collection of, *RE*, 302
 in increasing order, 108
 without repetition, 108

computation
 distributed, 288
 Finite State machine, 181
 nondeterministic Finite State machine, 189,
 194
 relative to an oracle, 109
 step, 8
 Turing machine, 9

computation tree
 branch, 189

concatenation of languages, 140
concatenation of strings, 362

configuration, 8, 180, 188, 193
 halting, 181, 189, 194
 initial, 8
conjunction, 276, 369
Conjunctive Normal form, 152, 276, 329, 354, 370
connected component, 294
connected graph, 156
connectives
 logical, 369
context free
 grammar, 145
 language, 240
context sensitive grammar, 240
control, of a machine, 4
converge, 10
Conway, J
 picture, 43
Cook reducibility, 314
Cook, S
 picture, 326
Cook-Levin theorem, 326
correspondence, 58, 367
 Cantor's, 65
countable, 60
countably infinite, 60
Course Scheduling problem, 285
CPU of Turing machine, 4
Crossword problem, 280
current symbol, 8
CW, 161
cycle, 156
Cyclic Shift problem, 320

daemon, *see* demon
DAG, directed acyclic graph, 156
dangling else, 149
De Morgan, A
 picture, 275
decidable, 289
 language, 98
decidable language, 302
decide a language, 141
decided, 13
decided language, 181, 289
 of a nondeterministic Turing machine, 304
decider, 11
decides
 language, 302

set, 11
decision problem, 3, 11, 35, 288
decrypter, 346
degree of a vertex, 159
degree sequence, 159
demon, or daemon, 188
DeMorgan's laws
 for logic expressions, 370
depth first traversal, 163
depth-first traversal, 170
derivation, 144, 145, 147
derivation tree, 145
determinism, 8, 16
diagonalization, 72–78, 116, 117
 effectivized, 87
digraph, 156
directed acyclic graph, 156
directed graph, 156
Discrete Logarithm problem, 294
disjunction, 276, 369
Disjunctive Normal form, DNF, 370
distinguishable classes, 224
distinguishable states, 223
distributed computation, 288
distributive laws
 for logic expressions, 370
diverge, 10
Divisor problem, 281, 294
DNF, 370
domain, 364, 365
 in the Theory of Computation, 365
Double-SAT problem, 312
doubler function, 3, 12
dovetailing, 104
Droste effect, 388
Drummer problem, 322
DSPACE, 343
DTIME, 342

edge, 155
edge weight, 156
Edmunds, J
 picture, 395
effective, 3, 14
Electoral College, 280
empty language
 decision problem, 293
empty string, ϵ , 8, 60, 362

encrypter, 346
Entscheidungsproblem, 3, 11, 14, 35, 288, 333, 334
 unsolvability, 94
 enumerate, 60
 ϵ closure, 193
 ϵ moves, 190
 ϵ transitions, 190–194
 equinumerous sets, 59
 equivalent growth rates, 263
 equivalent propositional logic statements, 369
 Euler Circuit problem, 274, 295
 Euler circuit, 156
 Euler, L
 picture, 274
 eval, 80
 exclusive or, 42
 EXP, 341–342
 expansion of a production, 144
 Ext, extensible functions, 115
 Extended Church’s Thesis, 299
 extended regular expression, 242
 extended transition function, 181
 for nondeterministic Finite State machines, 194
 nondeterministic Finite State machine, 189
 extensible, 115

F-SAT problem, 294
 Factoring problem, 346
 Prime Factorization problem, 294
 Fermat number, 36
 Fibonacci numbers, 29
 Fifteen Game problem, 281, 295
 Fin problem, 325
 final state, 175, 177
 in transition tables, 177
 nondeterministic Finite State machine, 188
 finite set, 60
 Finite State automata, *see* Finite State machine
 Finite State machine, 175–185
 accept string, 181
 accepting state, 177
 alphabet, 177
 computation, 181
 configuration, 180
 final state, 177
 halting configuration, 181
 initial configuration, 181
 input string, 181
 language of, 181
 minimization, 222–233
 next-state function, 177
 nondeterminism, 303
 nondeterministic, 188
 powerset construction, 194
 product, 211
 reject string, 181
 state, 177
 step, 181
 transition function, 177
 Fixed point theorem, 116–122
 discussion, 119–121
 Flauros, Duke of Hell
 picture, 188
 flow, 321
 flow chart, 80
 Four Color problem, 275
 function, 364–368
 91 (McCarthy), 30
 Ackermann, 47
 argument, 364
 Big \mathcal{O} , 261
 Big Θ , 263
 Boolean, 371
 boolean, 276
 codomain, 364
 composition, 367
 computable, 11
 computed by a Turing machine, 9
 converge, 10
 correspondence, 58, 367
 definition, 364
 diverge, 10
 domain, 364
 doubler, 3, 12
 effective, 3
 enumeration, 60
 exponential growth, 265
 extended transition, 181
 extensible, 115
 general recursive, 35
 identity, 367
 image under, 365
 index, 364
 injection, 366
 inverse, 367

left inverse, 367
logarithmic growth, 265
 μ recursive function (mu recursive), 35
next-state, 8, 177
one-to-one, 58, 366
onto, 58, 366
order of growth, 261
output, 364
pairing, 65, 132
partial, 10, 365
partial recursive, 35
polynomial growth, 265
predecessor, 6
projection, 24, 35
range, 365
recursive, 11, 35
reduction, 315
restriction, 365
right inverse, 367
successor, 12, 21, 24, 35
surjection, 366
total, 10, 115, 365
transition, 8, 177, 303
translation, 314
unpairing, 65, 132
value, 364
well-defined, 364, 366
zero, 24, 35
function problem, 288
functions
 same behavior, 98
 same order of growth, 263
gadget
 example of, 329
 in complexity arguments, 329
Galilei, Galileo
 picture, 57
Galileo, *see* Galilei, Galileo
Galileo's Paradox, 57, 59, 60
Game of Life, 43–46
gate, 40, 297
general recursion, 30–37
general recursive function, 35
general unsolvability, 89–92
Gödel number, 69
Gödel, K, 14
 letter to von Neumann, 334
picture, 15
picture with Einstein, 125
Gödel's theorem, 14
Goldbach's conjecture, 34, 97, 104
grammar, 143–154
 ambiguous, 148
 Backus-Naur form, BNF, 165
body of a production, 144
context free, 145
context sensitive, 240
derivation, 144
expansion of a production, 144
head, 144
linear, 200
nonterminal, 144
production, 144, 145
rewrite rule, 144, 145
right linear, 200
start symbol, 145
syntactic category, 145
terminal, 144
graph, 155–165
 adjacent, 155
 adjacent edges, 156
 bridge edge, 294
 chromatic number, 158
 circuit, 156
 clique, 278
 closed walk, 156
 coloring, 158
 colors, 275
 complete, 160
 connected, 156
 connected component, 294
 cycle, 156
 degree sequence, 159
 digraph, 156
 directed, 156
 directed acyclic, 156
 edge, 155
 edge weight, 156
 Euler circuit, 156
 Hamiltonian circuit, 156
 induced subgraph, 157
 isomorphism, 159–160
 loop, 156
 matrix representation, 158
 multigraph, 156

neighbors, 155
 node, 155

- rank, 170
- open walk, 156
- path, 156
- planar, 161, 275
- representation, 157–158
- simple, 155
- spanning subgraph, 277
- subgraph, 157
- trail, 156
- transition, 7
- traversal, 156–157, 163
 - breadth-first, 170
 - depth-first, 170
- tree, 156, 277
- vertex, 155
- vertex cover, 277
- vertex degree, 159
- walk, 156
- walk length, 156
- weighted, 156

Graph Colorability problem, 275, 295, 317, 331
Graph Connectedness problem, 294, 296
Graph Isomorphism problem, 294, 332
 Grassmann, H, 21

- picture, 21

 guessing

- by a machine, 188

 hailstone function, 95
Halt light, 5
 halting configuration, 181, 189, 194
 Halting problem, 87–89, 98

- as a decision problem, 295
- discussion, 92–94
- in intellectual culture, 123–126
- reduction to another problem, 92
- relativized, 113
- significance, 93
- unsolvability, 88

 halting state, 12
Halts On Three problem, 89, 110, 314
 Hamilton, W R

- picture, 272

 Hamiltonian circuit, 156
Hamiltonian Circuit problem, 273, 295, 308, 321, 328

- isomorphic graphs, 159
- isomorphism, 159

 Hamiltonian Path problem, 308, 339
 hard

- for a class, 327
- NP*, 327

 haystack, 297
 head

- read/write, 4

 head of a production, 144
 Hilbert’s Hotel, 122–123
 Hilbert, D, 3

- picture, 124

 Hofstadter, D, 388
 hyperoperation, 31
 I/O head, *see* read/write head
 identity function, 367
Ignorabimus, 124
 image under a function, 365
 implies, 42
 Incompleteness Theorem, 14
Independent Set problem, 286, 296, 320, 322, 340
 index number, 69
 index set, 98
 indistinguishable states, 222
 induced subgraph, 157
 infinite set, 60
 infinity, 57–64, 78
 initial configuration, 8, 181, 188, 193
 initial state, 181, 193
 injection, 366
 input

- loading, 9
- input alphabet, 177
- input string, 181, 188, 193
- input symbol, 8
- input, to a function, 364
- instruction, 5, 8, 303
 - stack machine, 236

Integer Linear Programming problem, 311

- decision problem, 323
- inverse of a function, 367
 - left, 367
 - right, 367
 - two-sided, 367

k Coloring problem, 158, 275
 K , the Halting problem set, 88, 106
 complete among c.e. sets, 112
 K_0 , set of halting pairs, 97, 107, 111
 Karatsuba, A, 258
 Karp reducible, 315
 Karp, R
 picture, 327
 Kayal, N
 picture, 281
 Kleene star, 60, 139, 140, 362
 regular expression, 202
 Kleene's fixed point theorem, 117
 Kleene's theorem, 203–207
 Kleene, S, 35
 picture, 200
 K_n , complete graph on n vertices, 160
 Knapsack problem, 279, 288, 313, 318, 328
 Knight's Tour problem, 273
 Knuth, D
 picture, 268
 Kolmogorov, A
 picture, 257
 König's lemma, 157, 305
 Königsberg, 274
 L'Hôpital's Rule, 263
 \mathcal{L} -distinguishable, 250
 \mathcal{L} -indistinguishable, 250
 \mathcal{L} -related, 250
 lambda calculus, λ calculus, 14
 language, 139–143
 + operation, 214
 accept, 141
 accepted by a Finite State machine, *see* language, recognized by a Finite State machine
 accepted by Turing machine, 102, 289
 class, 140
 concatenation, 140
 context free, 240
 decidable, 98, 302
 decide, 141
 decided, 289
 decided by a Finite State machine, 181
 decided by a Turing machine, 13, 302
 decision problem, 288
 derived from a grammar, 147
 grammar, 145
 Kleene star, 140
 non-regular, 216–222
 of a Finite State machine, 181
 of a nondeterministic Finite State machine, 189
 operations on, 140
 power, 140
 recognize, 141
 recognized, 289
 recognized by a Finite State machine, 181
 recognized by Turing machine, 289
 regular, 210–216
 reversal, 140
 verifier, 307
 language decision problem, 288
 last in, first out (LIFO) stack, 233
 left inverse, 367
 leftmost derivation, 145
 LEGO, 5
 length, 156
 length of a string, 362
 lexicographic order, 60
 Life, Game of, 43–46
 LIFO stack, 233
 light
 Halt, 5
 Linear Divisibility problem, 313
 Linear Programming language decision problem, 280, 295, 311, 321
 Lipton's Thesis, 292
 loading, 9
 logic gate, 40
 logical connectives, 369
 logical operator
 and, 276, 369
 not, 276, 369
 or, 276, 369
 logical operators, 369
 Longest Path problem, 313, 339
 LOOP
 language, 51
 program, 51
 loop, 156
 LOOP program, 50–55
 \mathcal{M} -related, 250
 machine

state, 9
 many-one reducible, 314
 map, *see* function
 mapping reducible, 314
Marriage problem, *see* Drummer problem or Matching problem
Matching problem, 336
 matching, three dimensional, 279
Max Cut problem, 278
Max-Flow problem, 321
 McCarthy's 91 function, 30
 memoization, 67
 memory, 4
 metacharacter, 144, 165, 201
 Meyer, A
 picture, 51
Minimal Spanning Tree problem, 288
 minimization, 33
 minimization of a Finite State machine, 222–233
 Brzozowski's algorithm, 232
 Moore's algorithm, 224
 minimization, unbounded, 35
Minimum Spanning Tree problem, 277
 modulus, 346
 Moore's algorithm, 224
 Morse code, 161
 μ -recursion (mu recursion), 33
 μ recursive function, 35
 multigraph, 156
 multiset, 279
 Musical Chairs, 73

n-distinguishable states, 223
n-indistinguishable states, 223
n-distinguishable classes, 224
 Naur, P
 picture, 165
Nearest Neighbor problem, 294, 295
 needle, 297
 negation, 276, 369
 neighbors, 155
 next state, 5, 8
 next tape action, 5
 next-state function, 8, 177
 nondeterministic Finite State machine, 188
NFSM, *see* nondeterministic Finite State machine
 node, 155
 rank, 163

 nondeterminism, 185–200
 for Finite State machines, 188, 303
 for Turing machines, 303
 nondeterministic Finite State machine, 188
 accept string, 189, 194
 computation, 189, 194
 configuration, 188, 193
 convert to a deterministic machine, 194
 ϵ moves, 190
 ϵ transitions, 190
 halting configuration, 189, 194
 initial configuration, 188, 193
 initial state, 193
 input string, 188, 193
 language of, 189
 language recognized, 189
 reject string, 189, 194
 nondeterministic machine
 recognizes a language, 313
 nondeterministic Pushdown machine, 233–241
 nondeterministic Turing machine
 accepting state, 303
 decided language, 304
 definition, 303
 instruction, 303
 transition function, 303
 nonterminal, 144, 145
NP, 303–314
 NP complete, 326–332
 basic problems, 328
 NP hard, 327
 NP intermediate problems, 332
NSPACE, 343
NTIME, 342
 numbering, 69
 acceptable, 69

 Ω , Big Omega, 263
 σ , omicron, 263
 one-to-one function, 58, 366
 onto function, 58, 366
 open walk, 156
 operators
 logical, 369
 optimization problem, 288
 optimization problem reducibility, 324
 oracle, 108–116
 computably enumerable in, 116

oracle Turing machine, 109
order of growth, 257–272
 function, 261
 hierarchy, 265
ouroboros, 79
output, from a function, 364

P, 296–303
P hard, 319
P versus *NP*, 306, 332–337
pairing function, 65, 132
Paley, W
 picture, 126
palindrome, 14, 140, 238, 363
paradox
 Aristotle’s, 57
 Galileo’s, 57
 Zeno’s, 61
parameter, 83
Parameter theorem, 83
parametrization, 82–85
parametrizing, 83
parentheses
 balanced, 234
parse tree, 145
parser-generator, 167
partial function, 10, 365
partial recursive function, 35
Partition problem, 280, 312, 328, 329, 339
path, 156
perfect number, 93
Péter, R
 picture, 47
Petersen graph, 161, 163
pipe symbol, | , 144
planar graph, 161, 275
pointer, in C, 120
polynomial time, 297
polynomial time reducibility, 315
polytime, 297
power of a language, 140
power of a string, 363
powerset construction, 194
predecessor function, 6
prefix of a string, 363
present state, 5, 8
present tape symbol, 5
primality, 282

Primality problem, 281, 288, 289, 294, 309
Prime Factorization problem, 281, 288, 332, 338
primitive recursion, 21–30, 35
 arity, 23
primitive recursive functions, 24
private key, 347
problem, 287
 decision, 288
 function, 288
 Halting, 88, 89
 language decision, 288
 optimization, 288
 search, 288
 unsolvable, 89
problem miscellany, 272–287
problems
 tractable, 266
 unsolvable, 103
product construction, 211
production, 145
production in a grammar, 144
program, 288
projection function, 24, 35
property
 semantic, 98
 syntactic, 98
Propositional logic, 368–371
 atom, 282
 Boolean algebra, 370
 Boolean function, 371
 Conjunctive Normal form, 152, 276, 322,
 354, 370
 DeMorgan’s laws, 370
 Disjunctive Normal form, 370
 distributive laws, 370
 exclusive or, 42
 implication, 42
 operators, 369
pseudopolynomial, 269
public key, 347
Pumping lemma, 216
pumping length, 216
Pushdown automata, *see* pushdown machine
Pushdown machine, 233–241
 input alphabet, 236
 nondeterministic, 233–241
 stack alphabet, 236
 transition function, 236

pushdown stack, 233

quantum advantage, 299

Quantum Bogosort, 401

Quantum Computing, 299

quantum computing

- quantum advantage, 299

quantum supremacy, *see also* quantum advantage

quine, 127

Quine's paradox, 388

r.e. set, *see* computably enumerable set

Radó, T

- picture, 129

RAM, *see* Random Access machine

Random Access machine, 267

range of a function, 365

rank, 163, 170

RE, computably enumerable sets, 290

reachable vertex, 156, 276

read/write head, 4

REC, computable sets, 290, 399

recognize a language, 141

recognized language

- of a Finite State machine, 181
- of a nondeterministic Finite State machine, 189
- of a Turing machine, 289

recursion, 21–37

Recursion theorem, 117

recursive function, 11, 35

recursive set, 11

recursively enumerable set, *see* computably enumerable set

reduces to, 110

reducibility

- between optimization problems, 324
- Cook, 314
- Karp, 315
- polynomial time, 315
- polytime, 315
- polytime many-one, 315
- polytime mapping, 315
- polytime Turing, 314

reducible

- many-one, 314
- mapping, 314

reduction from the Halting problem to another, 92

reduction function, 315

reductions between problems, 92, 314–326

Reflections on Trusting Trust, 128

Reg problem, 325

regex, 242

regular expression, 200–210

- extended, 242
- in practice, 241–249
- operator precedence, 201
- regex, 242
- semantics, 202
- syntax, 201

regular language, 210–216

reject an input, 181, 189, 194

rejecting state, 13

rejects, 177

relation, computable, 11

relativized Halting problem, 113

replication of a string, 363

representation, of a problem, 292

restriction of a function, 365

reversal of a language, 140

reversal of a string, 363

rewrite rule, 144, 145

Rice's theorem, 98–103

right inverse, 367

right linear, 200

Ritchie, D

- picture, 51

Rivest, R

- picture, 346

RSA Encryption, 345–351

Russell set, 382

s-m-n theorem, 83

same behavior, functions with, 98

same order of growth, 263

SAT, *see* Satisfiability problem

SAT solver, 353

Satisfiability problem, 276, 285, 291, 307, 316, 317, 322, 326

- as a language recognition problem, 290
- on a nondeterministic Turing machine, 305

satisfiable Propositional logic expression, 276

Satisfying Assignment problem, 290

Saxena, N

picture, 281
 schema of primitive recursion, 23
 Science United, 288
 search problem, 288
 self reproducing program, 127
 self reproduction, 126–128
 semantic property, 98
 semicomputable set, 104
 semidecidable set, 104
 semidecide a language, 141
 semiprime, 281
Semiprime problem, 312
 set
 c.e., 104
 cardinality, 59
 computable, 11, 104
 computably enumerable, 103–108
 countable, 60
 countably infinite, 60
 decider, 11
 equinumerous, 59
 finite, 60
 index, 98
 infinite, 60
 oracle, 108–116
 r.e., *see* computably enumerable set
 recursive, 11
 recursively enumerable, *see* computably enumerable set
 reduces to, 110
 semicomputable, 104
 semidecidable, 104
 T equivalent, 111
 Turing equivalent, 111
 uncountable, 73
 undecidable, 89
Set Cover problem, 321
 Shamir, A
 picture, 346
 Shannon, C
 picture, 40
Shortest Path problem, 274, 288, 295, 296, 315
 \sim , asymptotically equivalent, 272
 simple graph, 155
SPACE, 343
 span a graph, 277
 spanning subgraph, 277
***st*-Connectivity** problem, *see* Vertex-to-Vertex Path problem
***st*-Path** problem, *see* Vertex-to-Vertex Path problem
 stack, 233
 alphabet, 236
 bottom, \perp , 234
 LIFO, Last-In, First-Out, 233
 pop, 233
 push, 233
 Start button, 5, 177
 start state, 5, 177
 Pushdown machine, 236
 start symbol, 145
 state, 177
 accept, 13
 accepting, 175, 177, 303
 final, 175, 177
 halting, 12
 next, 5
 present, 5
 reject, 13
 start, 5
 unreachable, 102
 working, 12
 state machine, 9, 377
 states, 4
 distinguishable, 223
 indistinguishable, 222
 n-distinguishable, 223
 n-indistinguishable, 223
 set of, 8
 Stator square, 398
STCON problem, *see* Vertex-to-Vertex Path problem
 step of a computation, 8, 181
 store, of a machine, 4
 str function, 292
Strict 3-Satisfiability, 322
 string, 139, 362–363
 concatenation, 362
 decomposition, 363
 empty, 8, 60, 362
 length, 362
 power, 363
 prefix, 363
 replication, 363
 reversal, 363

substring, 363
suffix, 363
string accepted
 by deterministic Finite State machine, 177, 181
 by nondeterministic Finite State machine, 189, 194
string rejected, 177
String Search problem, 297
subgraph, 157
 induced, 157
Subset Sum problem, 279, 288, 296, 312, 318, 339
substring, 363
Substring problem, 320
successor function, 12, 21, 24, 35
suffix of a string, 363
surjection, 366
symbol, 8, 139, 362
 action, 8
 current, 8
 input, 8
syntactic category, 145
syntactic property, 98

T equivalent, 111
T reducible, 110
table, transition, 7
tail recursion, 172
tape, 4
tape alphabet, 8
tape symbol, 8
 blank, 5
terminal, 144, 145
tetration, 32
Thompson, K
 picture, 128
Three Dimensional Matching problem, 279, 312
time taken by a machine, 267
token, 139, 362
Tot, set of total computable functions, 107, 115
total function, 10, 115, 365
Towers of Hanoi, 26
tractable, 265–266
trail, 156
transformation function, *see* reduction function
transition function, 8, 177, 303
 extended, 181, 194
graph of, 7
Pushdown machine, 236
table of, 7
transition graph, 7
transition table, 7
translation function, 314
Traveling Salesman problem, 186, 273, 290, 306, 321, 323, 328, 339, 352
Asymmetric, 323, 324
traversal, 163
tree, 156, 277
 binary, 173
 rank, 163
 traversal, 169–173
Triangle problem, 301
triangular numbers, 25
truth table, 276, 369
Turing equivalent, 111
Turing machine, 3–14
 accept a language, 102
 accepting a language, 302
 accepting state, 13
 action set, 8
 action symbol, 8
 computation, 9
 configuration, 8
 control, 4
 CPU, 4
 current symbol, 8
 decidable, 289
 decides a set, 11
 deciding a language, 302
 definition, 8
 deterministic, 8
 for addition, 6
 function computed, 9
 Gödel number, 69
 index number, 69
 input symbol, 8
 instruction, 5, 8
 language accepted, 289
 language decided, 13, 289
 language recognized, 289
 multitape, 20
 next action, 5
 next state, 5, 8
 next-state function, 8
nondeterminism, 303

numbering, 69
palindrome, 14
present state, 5, 8
present symbol, 5
rejecting state, 13
simulator, 37–39
tape alphabet, 8
transition function, 8
universal, 79–81
with oracle, 109
Turing reducibility, 314
Turing reducible, 110, 314
Turing, A
picture, 3
Turnpike problem, 313
two-sided inverse, 367

unbounded minimization, 33
unbounded search, 33
uncountable, 73
undecidable, 89
Unicode, 178, 391
uniformity, 81–82
Universal Turing machine, 79–81
universality, 78–87
unpairing function, 65, 132
unreachable state, 102
unsolvability, 103
unsolvable problem, 89, 103
use-mention distinction, 120

value, of a function, 364
verifier, 307
 polytime, 307
vertex, 155
 rank, 163
 reachable, 156, 276
vertex cover, 277
Vertex Cover problem, 277, 286, 320, 321
Vertex cover problem, 328
Vertex-to-Vertex Path problem, 276, 296, 301,
 315
von Neumann, J
picture, 43

walk, 156
walk length, 156
weight, 156

weighted graph, 156
well-defined, 364, 366
wire, 297
witness, 307
word, *see* string
working state, 12

XOR, 42

\vdash , yields
 for Finite State machines, 181, 189
 for nondeterministic Finite State machines,
 189, 193
 for Turing machines, 9

Zeno’s Paradox, 61
zero function, 24, 35
Zoo, Complexity, 344