Theory of Computation

Making Connections

Second edition

Jim Hefferon

https://hefferon.net/computation

Notation summary

Notation	*Description*
$\mathscr{P}(S)$	power set, collection of all subsets of S
S^{c}	complement of the set S
$\mathbb{1}_S$	characteristic function of the set S
$\langle a_0, a_1, \ldots \rangle$	sequence
$\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$	natural numbers $\{0, 1, \ldots\}$, integers, rationals, reals
a, b, . . . 0, 1	character (note the typeface)
Σ	alphabet, set of characters
$\mathbb{B}$	alphabet of bits characters $\{\mathtt{0}, \mathtt{1}\}$, or set of bits $\{0, 1\}$
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
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*	o	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 Divison, 1924-Nov-24.
Calculating the bonus owed to each US WW I veteran.
(Auto-generated cropping decoration added.)

This PDF was compiled 2025-Apr-26.

Preface

The Theory of Computation is a wonderful thing. It is beautiful. It has deep connections with other areas in mathematics and computer science, 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 also.

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 a pleasure.

For students While you were learning to instruct computers to do your bidding, have you ever wondered what cannot be done? And what can be done in principle but not in practice? In this course you will see signpost results in the study of these questions and begin to understand how they affect your work.

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

We won't stint on precision — why would we want to? — but we will approach the ideas liberally, in a way that attends to a breadth of knowledge in addition to technical detail. 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 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 many notes, many of which contain links that will take you even deeper. There are also Extra sections at the end of each chapter to explore further. Whether or not your instructor covers them in class, these 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 an introduction to 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, 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 on the first page, where we begin with Turing machines. The alternative of starting with Finite State machines is mathematically slicker but for 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 in outline or as working code in the Racket language, which better communicates the ideas than code for a theoretical model or intricate recursions.

A second example of choosing naturalness and making connections happens with 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 is the discussion of the current state of *P* versus *NP*. Taken together, these and many more encourage students to develop the habit of inquiry. They should expect that stuff makes sense.

Exploration and Enrichment The Theory of Computation is fascinating. This text aims to showcase that, to draw readers in, to be absorbing, including by using lively language and many illustrations.

One way to encourage readers is to make the material explorable. Where practical, references are clickable. This makes them very much more likely to be the subject of further reading than is the same content in a physical library.

Another example of encouraging engagement is the many notes in the back that fill out the core presentation. Still another example is the inclusion of 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 has not had an Ah-ha! moment in a hand-wavy hallway conversation?

Finally, chapters end in a number of additional topic sections. They are suitable as one-day lectures, or for assigning for group presentations or extra credit or honors credit, or just to read for pleasure.

Schedule I have used the semester schedule below. The classes were four credits and met three times a week. I used the slides available from the home page.

Chapter I defines computation, Chapter II covers unsolvability, Chapter III does

languages and graphs, Chapter IV is automata, and Chapter V is computational complexity. (Some instructors will replace Section IV.5 with Extra IV.B.) I assign the readings as homework and quiz on them.

	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.4, V.5	V.3	
14	V.6, V.6	V.B	
15	V.7	Discussion	Review for final

For those reading this text independently, I have marked a selection of exercises with ✓.

One reason for this license is the culture in which I have had the pleasure to work. The book is written using the LaTeX tools, which are Free, as is Asymptote that drew the illustrations, along with Emacs and all of GNU software, as well as the entire Linux platform on which this book was developed. And anyway, the research present here was 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 following a trail that is both well-marked and productive seems only right.

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

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

I must honor my teachers. First among them is M Lerman. Thank you, Manny. They also include H Abelson, GJ Sussman, and J Sussman, 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

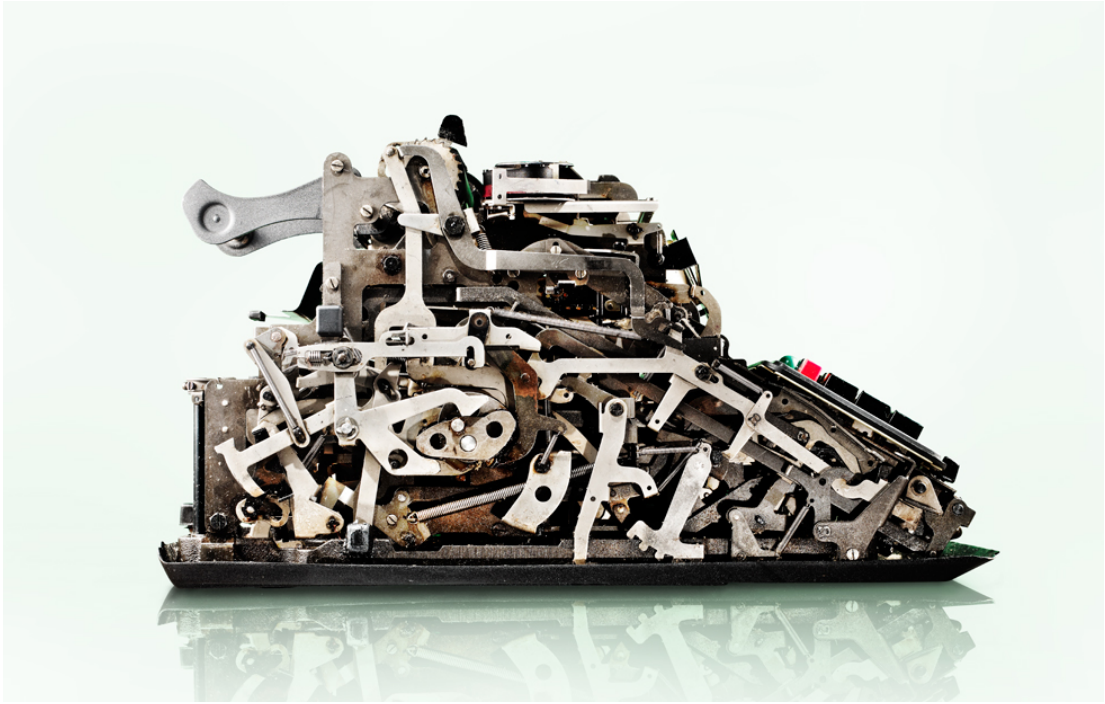
Of what use are computers? They can only give answers.
– P Picasso, THE PARIS REVIEW, SUMMER-FALL 1964

Jim Hefferon
Jericho, VT USA
University of Vermont
`hefferon.net`
Version 1.99, 2025-Mar-17

Contents

Part One

Classical Computability



Chapter I Mechanical Computation

What can be computed? For instance, the doubler function that inputs a natural number x and outputs $2x$ is intuitively mechanically computable. We shall call these 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.

Alan Turing 1912–1954

The context of Turing's thinking was the *Entscheidungsproblem*,[†] proposed in 1928 by D Hilbert and W Ackermann, which asks for an algorithm that takes as input a mathematical statement and decides 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. They follow a step-wise procedure and use 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.

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 an

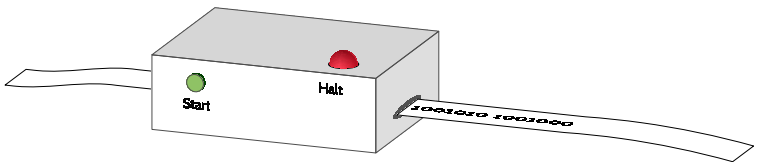
Image: copyright Kevin Twomey, `http://kevintwomey.com/lowtech.html` [†] German for "decision problem." Pronounced *en-SHY-duns-pob-lem*. [‡] He was a serious candidate for the 1948 British Olympic marathon team.

instrument dial or a slide rule. In line with this, the agent works step by step. 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, ...

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 beyond that for inputs and outputs.[†] He similarly put no upper bound on the number of instructions. He also 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 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. In both cases the point is that if something cannot be computed when there are no bounds then it cannot be computed on any physical 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 like the machines that we are used to it has integrated circuits, or perhaps it has gears and levers, or perhaps LEGO's — what matters is that it has finitely many parts, each of which 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 constructed, in total the box has only finitely many states.

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

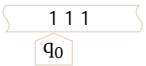
Consequently, machine steps involve four pieces of information. Call the present state q_p and the next state q_n. The symbol that the read/write head is presently pointing to is T_p. Finally, the next tape action is T_n. Possible actions are: moving the tape head left or right without writing, which we denote with $T_n = \mathtt{L}$ or $T_n = \mathtt{R}$,[†] 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 $q_pT_pT_nq_n$ is an instruction. For example, the instruction $q_3\mathtt{1B}q_5$ 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 $\Sigma = \{\mathtt{B}, \mathtt{1}\}$ has six instructions.

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

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 $q_0\mathtt{1R}q_0$ 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 after later steps. Briefly, the head slides to the right, blanks out the final 1, and slides back to the start.

[†] Whether we move the tape or the head doesn't matter, what matters is their relative motion. Thus $T_n = \mathtt{L}$ 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 the graphics are easier to read.

Step	Configuration
1	1 1 1 q_0
2	1 1 1 q_0
3	1 1 1 q_0
4	1 1 1 q_1
5	1 1 q_1
6	1 1 q_2
7	1 1 q_2
8	1 1 q_2
9	1 1 q_3

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

If this machine starts with an initial tape that is entirely blank except for n-many consecutive 1's with $n > 0$, and the read/write head points to the leftmost of those 1's, then when the machine halts, the tape will have $n - 1$-many 1's. If the tape starts with 0-many 1's then when the machine halts, the tape will have 0 many 1's. That is, this machine computes the predecessor function.

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

Other examples continue the convention of starting the machine with the head pointing to the leftmost symbol of the input.

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

$$\mathcal{P}_{\text{add}} = \{ q_0\mathtt{BB}q_1,\ q_0\mathtt{1R}q_0,\ q_1\mathtt{B1}q_1,\ q_1\mathtt{11}q_2,\ q_2\mathtt{BB}q_3,\ q_2\mathtt{1L}q_2,\\ q_3\mathtt{BR}q_3,\ q_3\mathtt{1B}q_4,\ q_4\mathtt{BR}q_5,\ q_4\mathtt{11}q_5 \}$$

The input numbers are represented by two strings of 1's, separated with a blank. This shows the machine ready to compute 2 + 3.

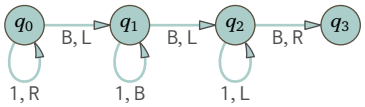
1 1 1 1 1
q_0

Once we press `Start`, the machine scans right, looking for the blank separator. It changes that blank into a 1. It then reverses to scan left until it finds the beginning of the input. Finally, it trims off a 1 and halts with the read/write head pointing to the start of the non-blank part of the tape. Here are the steps.

Step	Configuration
1	1 1 1 1 1 / q_0
2	1 1 1 1 1 / q_0
3	1 1 1 1 1 / q_1
4	1 1 1 1 1 1 / q_1
5	1 1 1 1 1 1 / q_2
6	1 1 1 1 1 1 / q_2
7	1 1 1 1 1 1 / q_2
8	1 1 1 1 1 1 / q_2
9	1 1 1 1 1 1 / q_3
10	1 1 1 1 1 1 / q_3
11	1 1 1 1 1 / q_4
12	1 1 1 1 1 / q_5

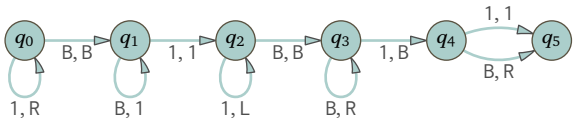
Instead of giving a machine's instructions as a list, we can use a table or a diagram. Here is the transition table for $\mathcal{P}_{\text{pred}}$ and its transition graph.

Δ_{pred}	B	1
q_0	Lq_1	Rq_0
q_1	Lq_2	Bq_1
q_2	Rq_3	Lq_2
q_3	–	–



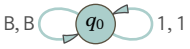
And here is the corresponding table and graph for $\mathcal{P}_{\text{add}}$.

Δ_{add}	B	1
q_0	Bq_1	Rq_0
q_1	1q_1	1q_2
q_2	Bq_3	Lq_2
q_3	Rq_3	Bq_4
q_4	Rq_5	1q_5
q_5	–	–



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

1.3 EXAMPLE The machine $\mathcal{P}_{\text{inf loop}} = \{q_0\text{B}\text{B}q_0,\ q_0\text{1}\text{1}q_0\}$ 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.[†]

1.4 DEFINITION A Turing machine $\mathcal{P}$ is a finite set of four-tuple instructions $q_pT_pT_nq_n$.[‡] 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 $\Sigma \cup \{\mathtt{L},\mathtt{R}\}$.

The set $\mathcal{P}$ must be deterministic: different four-tuples cannot begin with the same q_pT_p. Thus, over the set of instructions $q_pT_pT_nq_n \in \mathcal{P}$, the association of present pair q_pT_p with next pair T_nq_n defines a function, the transition function or next-state function $\Delta\colon Q \times \Sigma \to (\Sigma \cup \{\mathtt{L},\mathtt{R}\}) \times Q$.

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.

Tracing through Example 1.1 and Example 1.2 highlights that Turing machines act by governing transitions between machine arrangements. A configuration of a Turing machine is a four-tuple $\langle q, s, \tau_L, \tau_R\rangle$, where q is a state, 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 $s = \mathtt{B}$, 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 $\langle q_0, \mathtt{B}, 11, 111\rangle$. Restated, 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 $q_pT_pT_nq_n \in \mathcal{P}$ 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 configuration is $\mathcal{C}(t+1) = \langle q_n, T_n, \tau_L, \tau_R\rangle$, defined as follows. (2) If $T_n = \mathtt{L}$ then the machine moves the tape head to the left. More precisely, the next configuration is $\mathcal{C}(t+1) = \langle q_n, \hat{s}, \hat{\tau}_L, \hat{\tau}_R\rangle$. Its new right tape $\hat{\tau}_R$ is the concatenation of the

[†] 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 requires 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.

one-character string $\langle s \rangle$ with the prior configuration's τ_R. As to $\mathcal{C}(t+1)$'s new left tape and new present character, there are two cases: when $\tau_L = \varepsilon$ then $\hat{s}$ is the blank and $\hat{\tau}_L = \varepsilon$, and when $\tau_L \neq \varepsilon$ then the new present character $\hat{s}$ is τ_L's rightmost character, $\hat{s} = \tau_L[-1]$, while the new left tape $\hat{\tau}_L$ is τ_L with that rightmost character omitted, $\hat{\tau}_L = \tau_L[:-1]$. (3) If $T_n = \mathtt{R}$ then the machine moves the tape head to the right. This is so like (2) that 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 $\mathcal{C}(0) \vdash \mathcal{C}(1) \vdash \mathcal{C}(2) \vdash \cdots$. 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 $\mathcal{C}(0) \vdash^* \mathcal{C}(h)$.

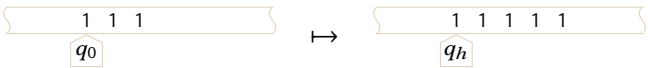
1.5 Example In Example 1.1's table tracing the machine's steps, the graphics illustrate the successive configurations. Here is the same sequence as a computation.

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

One final point about these details. We have defined computation as a process whereby a physical system evolves through a sequence of discrete steps. Note that this evolution is local, in that all of the action takes place within one cell of the head.

Computable functions In this chapter's opening we expressed more interest in the things that the machines compute than in how the computation proceeds.[#] We close 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 connects the string on the tape when the machine starts with the one on the tape when it stops. So we have 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." [#] This contrasts with a programming class, where a great deal of effort goes into learning how to decompose a complex task into simpler ones, how to develop reliable interfaces between those subtasks, etc.

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 build machines so that they also finish with the head under the first character of the output string, which isn't strictly necessary but makes it easier to compose machines. (3) In most fields of mathematics a function comes with a domain, the set of inputs on which it is defined. In this field a key property is that a machine may take in an input string but then never return an output, because it does not halt. So here we often write $\phi\colon \Sigma^* \to \Sigma^*$ 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.)[†]

We will often consider functions that are not maps from strings to strings and describe them as computed by a machine. For this, we must impose an interpretation on the input and output. An example is the predecessor machine in Example 1.1, where we took the strings to represent natural numbers in unary. 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 quartet, or however we please.[‡]

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 sometimes worry about 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 power of 5 in that number's 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

[†] For more see Appendix B. [‡] 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.

exclusively as a begin/end marker. Or we could enforce that strings come in the form $\sigma = \alpha \mathtt{B} \tau$ where τ consists of $|\alpha|$ many 1's. 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 typically don't insist on this level of detail. Our everyday experience convinces us that machines can use their alphabets to reasonably represent anything computable. Besides, spending a great deal of time on the details risks hiding the underlying ideas, and we want to get to more interesting material. The next section will say 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 relation is computable if it is computable as a set.[‡]

1.10 DEFINITION A Turing machine decides a set if it computes the characteristic function of that set. That is, for all inputs that are in the set the machine halts and accepts (perhaps signaled by ending with just a 1 on the tape), while for all inputs not in the set it halts and rejects (perhaps ending with an all-blank tape). A Turing machine recognizes a set if for all set member inputs it halts and accepts, while for nonmembers it never halts and accepts (but it might fail to halt).

We close with a summary. We have given a precise definition of what is intuitively mechanically computable. This gives us a precise characterization 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 $\Sigma = \{\mathtt{B}, \mathtt{1}\}$. *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.*

1.11 How is a Turing machine like a program? How is it unlike a program? How is it like the kind of computer that we have on our desks? Unlike?

1.12 Why does the definition of a Turing machine, Definition 1.4, not include a definition of the tape contents?

1.13 Your study partner asks, "The opening paragraphs talk about the *Entscheidungsproblem*, to mechanically determine whether a mathematical statement is true or false. I write programs with Boolean decision clauses like "`if (x>3)`" all the time. What's the problem?" Help them out.

✓ 1.14 Trace each computation, as in Example 1.5. (A) The machine $\mathcal{P}_{\text{pred}}$ from Example 1.1 when starting on a tape with two 1's. (B) The machine $\mathcal{P}_{\text{add}}$ from Example 1.2 where the addends are 2 and 2. (C) Give the two computations as configuration sequences, as on page 8.

[†] The term 'recursive' used to be universal but is now old-fashioned. [‡] For instance, the relation 'less than' is computable because there is a computable function that inputs two integers a and b and returns 1 if $a < b$ but otherwise returns 0.

✓ 1.15 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 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.

1.16 Some authors explicitly define Turing machines to have halting states, where we send the machine solely to make it halt. In this case the others are working states. Our definition doesn't require that but many of our examples use the idea. 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.

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

1.18 Trace the execution on each given input of this Turing machine for ten steps or fewer if it halts.

$$\{\, q_0\texttt{B}\texttt{B}q_4,\ q_0\texttt{0}\texttt{R}q_0,\ q_0\texttt{1}\texttt{R}q_1,\ q_1\texttt{B}\texttt{B}q_4,\ q_1\texttt{0}\texttt{R}q_2,\ q_1\texttt{1}\texttt{R}q_0,\ q_2\texttt{B}\texttt{B}q_4,\ q_2\texttt{0}\texttt{R}q_0,\ q_2\texttt{1}\texttt{R}q_3 \,\}$$

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

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

✓ 1.20 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.

✓ 1.21 Produce Turing machines to perform these Boolean operations, using $\Sigma = \{\texttt{B}, \texttt{0}, \texttt{1}\}$ and $\Sigma_0 = \{\texttt{0}, \texttt{1}\}$. (A) Take the 'not' of a bit $b \in$. That is, convert the input $b = \texttt{0}$ into the output 1, and convert 1 into `0`. (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 `0`, while if the input is 11 then the output should be 1. (C) Do the same for 'or'.

1.22 Give a Turing machine that takes as input a bit string, using the alphabet $\{\texttt{B}, \texttt{0}, \texttt{1}\}$, and appends `01` at the back.

1.23 Produce a Turing machine over $\Sigma = \{\texttt{B}, \texttt{1}\}$ 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.

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

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

(A) Take the machine to be over $\Sigma = \{\texttt{B}, \texttt{1}\}$ and the input and output to be in unary. *Hint:* erase the first 1, move to the end of the 1's, past a blank, and put down two 1's. Then move left to the start of the first sequence of 1's. Repeat.

(B) Instead assume that the alphabet is $\Sigma = \{\texttt{B}, \texttt{0}, \texttt{1}\}$ and the input is represented in binary.

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

1.27 Write a machine $\mathcal{P}$ with tape alphabet Σ consisting of blank B, stroke 1, and the comma ',' character. Where $\Sigma_0 = \Sigma - \{\mathtt{B}\}$, 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, $\phi_{\mathcal{P}}(1111,111,1) = 11111111$.

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

1.29 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 $\Sigma = \{0, 1, \mathtt{B}\}$. (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 $b_3b_2b_1b_0$ you end with $b_2b_1b_0b_3$. (C) Take as input two 4-bit strings and perform a bitwise AND.

✓ 1.30 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.

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 that 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 instructions 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 *See the instructions before Exercise 1.31.* 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 instructions 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 $\Sigma = \{0, 1, \mathtt{B}\}$ 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 $\Sigma = \{\mathtt{B}, \mathtt{a}, \mathtt{b}, 0, 1\}$. Have the machine end with a single 1 on the tape if the input is a palindrome over $\{\mathtt{a}, \mathtt{b}\}$, and with a blank tape if not.

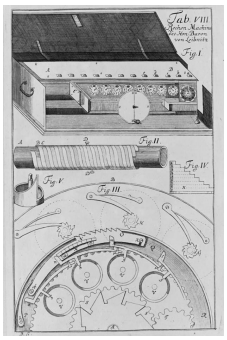
1.36 Turing machines tend to have many instructions and to be hard to understand. So rather than exhibit a machine, people often give an overview. Do that for a machine that replicates the input: if it is started with the tape blank except for a contiguous sequence of n-many 1's, then it will halt with 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 a person is told that the machine was brought to that configuration and was asked what was the prior configuration, then they 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, $h(t) = -4.9t^2 + 58.4$. This is a kind of program: get the height output by squaring the time input, multiplying by -4.9, and adding 58.4.



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. Leibniz wrote that to settle any dispute 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 'mechanically computable' 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 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.

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

Alonzo Church 1903–1995

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 without any need for intuition or insight. 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: how far does this collection extend? Arguing that 'derivable with the λ calculus' implies effective does not give the converse.

Everything changed with Turing's masterful analysis, outlined in the prior section. Gödel 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 asserts that our theoretical results have a larger importance, that they describe the physical 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 works within this formalization they are then free to reason 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

Even though it is not 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 scholars 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 in the 1930's but also everything ever computed by every program written for every existing computer, because all of them can be rewritten as 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 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

[‡] In recent years this has come to be often called the Church-Turing Thesis. Here we figure that because Turing has the machine, we can give Church the thesis.

of another influential approach. However, despite this wide variation, 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 we find that they have taken many different approaches but they all got the same answer then we could reasonably suppose that it is 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 $\Sigma = \{\mathtt{B}, \mathtt{1}\}$ 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 does not allow a machine to have, say, both of the instructions $q_5\mathtt{1R}q_6$ and $q_5\mathtt{1L}q_4$, because they both begin with $q_5\mathtt{1}$. 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 if given access to 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? 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?

[†] 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.

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 reliably computes a function that no Turing machine can do. But there is also not yet an analysis of physically-possible mechanical computation in the non-discrete case that has the support enjoyed by Turing's analysis in its more narrow domain.

We will not pursue this further, instead only observing that the mainstream community of researchers takes 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 all reasonable models of computation, including 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 — the set of functions that each model computes equals the set of functions that are computable by in-principle physically realizable mechanisms. 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 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 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 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

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

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 it 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 persons 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 $f, g\colon \mathbb{N} \to \mathbb{N}$. (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 $f_0, f_1 : \mathbb{N} \to \mathbb{N}$ are computable partial functions. Show that $h : \mathbb{N} \to \mathbb{N}$ 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 $f : \mathbb{N} \to \mathbb{N}$ is a computable function that is total. Show that $h : \mathbb{N} \to \mathbb{N}$ is a computable partial function, where $h(x) = 1$ if x is in the range of f and and $h(x)\uparrow$ otherwise. (D) Suppose $f_0, f_1 : \mathbb{N} \to \mathbb{N}$ are computable total functions. Show that their composition $h = f_1 \circ f_0$ is a computable function $h : \mathbb{N} \to \mathbb{N}$. (E) Suppose $f_0, f_1 : \mathbb{N} \to \mathbb{N}$ are computable partial functions. Show that their composition is a computable partial function $f_1 \circ f_0 : \mathbb{N} \to \mathbb{N}$.

✓ 2.12 Suppose that $f : \mathbb{N} \to \mathbb{N}$ 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 $f, g : \mathbb{N} \to \mathbb{N}$ 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 $p(x_0, \ldots x_n)$, we can determine whether or not it has natural number roots by trying all possible settings of the input $(x_0, \ldots x_n)$ to $n+1$-tuples of integers."

The next three exercises involve multitape machines. Definition 1.4's transition function for a single tape is $\Delta : Q \times \Sigma \to (\Sigma \cup \{\mathtt{L}, \mathtt{R}\}) \times Q$. *Define a machine with k-many tapes by extending to* $\Delta : Q \times \Sigma^k \to (\Sigma \cup \{\mathtt{L}, \mathtt{R}\})^k \times Q$. *Thus, a typical four-tuple for a* $k = 2$ *tape machine with alphabet* $\Sigma = \{\mathtt{0}, \mathtt{1}, \mathtt{B}\}$ *is* $q_4\langle \mathtt{1}, \mathtt{B}\rangle\langle \mathtt{0}, \mathtt{L}\rangle q_3$. *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.

2.15 Someone online asks, "How could a multitape Turing machine be equivalent to a single tape when single tape can loop forever? It seems like the multi tape one has a harder time looping forever than the single tape, because all tapes would have to loop." What are they missing?

2.16 Write the transition table of a two-tape machine to complement a bitstring. The machine has alphabet $\{\mathtt{0}, \mathtt{1}, \mathtt{B}\}$. 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 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).

✓ 2.18 If we allow processes to take infinitely many steps then we can have all kinds of fun. Suppose that we have infinitely many dollars. We run into the Devil. He proposes an infinite sequence of transactions, in each of which he will hand us two dollars and take from us one dollar. (The first transaction will take $1/2$ hour, the second $1/4$ hour, etc.) We figure we can't lose. But he proves to be particular about the order in which we exchange bills. First he numbers the bills as 1, 3, 5, ... At each step he buys our lowest-numbered bill and pays us with two higher-numbered bills. Thus, he first accepts from us bill number 1 and pays us with his own bills, numbered 2 and 4. Next he buys from us bill number 2 and pays us with his bills numbered 6 and 8. How much do we end with?

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 way to do this and because it is useful.[†] We will list some initial functions that are intuitively computable. We will also describe 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 $\mathcal{S}\colon \mathbb{N} \to \mathbb{N}$, described by $\mathcal{S}(x) = x + 1$. An example of a combiner that preserves computability is function composition. Using those, the plus-two operation $\mathcal{S} \circ \mathcal{S}\,(x) = x + 2$ is also intuitively mechanically computable.

Primitive recursion We now introduce another effectiveness-preserving combiner.

Hermann Grassmann 1809-1877

Grade school students learn addition and multiplication as mildly involved algorithms. They multiply, for example, by arranging the digits into a table, doing partial products, and then adding. In 1861, H Grassmann produced a more elegant definition. Here is the formula for addition, $\text{plus}\colon \mathbb{N}^2 \to \mathbb{N}$, which takes as given the successor map.

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

3.1 Example This finds the sum of 3 and 2.

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

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

This approach has an 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 is descriptive because it specifies the meaning, the semantics, of the operation.

On first seeing recursion, many people wonder whether it might be logically problematic — isn't defining something in terms of itself a fallacy? However, in the example above plus$(3, 2)$ is not defined in terms of itself, it is defined in terms of plus$(3, 1)$ (and the successor function). Similarly, plus$(3, 1)$ is defined in terms of plus$(3, 0)$. And, clearly the definition of 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 product$(2, 3)$ gives a sum of three 2's.

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

Exponentiation works the same way.

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

3.3 Example Similarly, the expansion of power$(2, 3)$ gives a product of three 2's.

$$\begin{aligned} \text{power}(2, 3) &= \text{product}(\text{power}(2, 2), 2) = \text{product}(\text{product}(\text{power}(2, 1), 2), 2) \\ &= \text{product}(\text{product}(\text{product}(\text{power}(2, 0), 2), 2), 2) \\ &= \text{product}(\text{product}(\text{product}(1, 2), 2), 2) = 8 \end{aligned}$$

We are interested in Grassmann's definition because it is effective — it translates directly 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 many other 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 and sets it to $y - 1$.) 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))))
```

3.4 DEFINITION A function f is defined by the schema[†] of primitive recursion from the functions g and h when this holds.

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

Here 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.

3.5 EXAMPLE The function plus is defined by primitive recursion from $g(x_0) = x_0$ and $h(w, x_0, z) = \mathcal{S}(w)$. 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 $h(w, x_0, z) = \text{product}(w, x_0)$.

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

3.6 EXAMPLE The predecessor function is like an inverse to successor except that we are using the natural numbers and so we can't allow the predecessor of zero to be negative. We instead take the special case that if the input is zero then the output is zero also. We can define this function pred: $\mathbb{N} \to \mathbb{N}$ using the primitive recursive schema.

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

Comparing this with Definition 3.4, pred has no x_i's. Thus the bookkeeping is that g has an arity of zero, and with no inputs it is therefore the constant function $g() = 0$. As to h, its arity is two although it ignores its first input, $h(w, z) = z$.

3.7 EXAMPLE For subtraction we must also special-case negatives. We take proper subtraction, denoted $x \dot{-} y$, to equal $x - y$ unless it is negative, in which case it

[†] A schema is an underlying organizational pattern or structure.

equals 0. This defines that function via primitive recursion.

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

The function f has arity two. That makes g of arity one, $g(x_0)=x_0$. And the arity of h is three so $h(w,x_0,z)=\text{pred}(w)$, with two dummy inputs.

Here is the promised collection of initial functions and function combiners.

3.8 Definition The set of primitive recursive functions[†] has the initial operations of the zero function $\mathcal{Z}(\vec{x})=0$, the successor function $\mathcal{S}(\vec{x})=x+1$, and the projection functions $\mathcal{I}_i(\vec{x})=\mathcal{I}_i(x_0,\ldots x_{k-1})=x_i$. This set consists of the functions that can be obtained from the initial ones by a finite number of applications of function composition and primitive recursion.

The initial functions are all clearly effective. Note also that the combiners are such that if the parts are effective then so is their combination. In particular, the computer code above makes evident that primitive recursion preserves effectiveness. Hence every function in that set is of interest to us, as it is intuitively mechanically computable.

Function composition covers not just the simple case of two functions f and g that combine as $f\circ g\,(\vec{x})=f(g(\vec{x}))$. It also covers simultaneous substitution, where from $f(x_0,\ldots x_n)$ and $h_0(y_{0,0},\ldots y_{0,m_0})$, ... and $h_n(y_{n,0},\ldots y_{n,m_n})$ we get $f(\,h_0(y_{0,0},\ldots y_{0,m_0}),\ldots h_n(y_{n,0},\ldots y_{n,m_n})\,)$.

3.9 Example The function defined by the recurrence

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

has a first few values of $f(0)=2$, $f(1)=f(0)+3\cdot 1+2=7$, and $f(2)=f(1)+3\cdot 2+2=15$. We will show that it is primitive recursive by verifying that it is built from the initial functions using only the two combiners.

On the outermost level it looks like a primitive recursion. Because f is a function of one input, the bookkeeping says that g has no inputs and h has two. We want this.

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

As a function of no arguments, g is constant. So we use $g(\,)=\mathcal{S}(\mathcal{S}(Z(\,)))=2$. For h we need that $h(f(z),z)=f(y-1)+3y+2=f(z)+3(z+1)+2$. We can use this.

$$h(a,b)=\text{plus}(\text{plus}(a,\text{product}(\mathcal{S}(\mathcal{S}(\mathcal{S}(Z(a))))),\text{plus}(b,1))),\mathcal{S}(\mathcal{S}(Z(a))))$$

[†] 'Primitive' distinguishes these from another set of functions that we will see in the next section. The vector $\vec{x}$ abbreviates $x_0,\ldots x_{k-1}$. There are actually infinitely many zero functions, one for each natural number arity k, despite that we called it "the" zero function. The same holds for successor. For projection, for each k there is a projection for each $i\in\{0,\ldots k-1\}$. Also, projection generalizes the identity function, which is why we use the letter $\mathcal{I}$.

Besides the functions in this section's examples, many other familiar mathematical operations are in the set of primitive recursive functions. They include the boolean function that tests whether one number is less than another, the elementary arithmetic function that finds the remainder left when one number is divided by another, and the number-theoretic function that inputs a number and a prime and returns the largest power of the prime that divides that number.

We have noted that every primitive recursive function is mechanically computable. The list of primitive recursive functions given above and in the exercises is so extensive that we may wonder whether every mechanically computable functions is in the set of primitive recursive functions. The next section shows that the answer is no — although primitive recursion is powerful, nonetheless there are intuitively mechanically computable functions that are not primitive recursive.

I.3 Exercises

✓ 3.10 What is the difference between primitive recursion and primitive recursive?

3.11 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?

✓ 3.12 As the section body describes, recursion doesn't have to be logically problematic. But some recursions are ill-defined; 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 $f(2)$.

3.13 Consider this function.

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

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

(B) Show that F is primitive recursive by describing it in the form of Definition 3.4, giving suitable functions g and h You can use functions already shown in this section to be primitive recursive.

3.14 The function plus_three: $\mathbb{N} \to \mathbb{N}$ adds three to its input. Show that it is a primitive recursive function.

3.15 The Boolean function is_zero inputs a natural number and returns T if the input is zero, and F otherwise. Give a definition by primitive recursion, representing T with 1 and F with 0.

✓ 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 $s(0), \ldots s(10)$.
(B) Verify that s is primitive recursive by putting it in the form given in Definition 3.4, giving suitable functions g and h. You can use functions already shown in this section to be primitive recursive.

✓ 3.17 Start with a square array of dots that is n dots on a side. Consider those dots that are below or on the diagonal (the upper left to lower right diagonal). This triangle has one dot in row 1, two in row 2, etc. The total number of dots in n rows is the n-th triangular number $t(n)$.
(A) Find $t(0), \ldots t(10)$.
(B) Show that t is primitive recursive by describing it in the form given in Definition 3.4. For g and h you can use functions already verified in this section to be primitive recursive.

3.18 Consider this recurrence.

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

(A) Find $d(0), \ldots d(5)$.
(B) Verify that d is primitive recursive by putting it in the form given in Definition 3.4. You can use functions already shown in this section to be primitive recursive.

✓ 3.19 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 $n = 1, \ldots, 10$.

(B) Verify that H is primitive recursive by putting it in the form given in Definition 3.4. You can use functions already shown in this section to be primitive recursive.

3.20 Define the factorial function $\text{fact}(y) = y \cdot (y-1) \cdots 1$ by primitive recursion, using functions that this section has already shown are primitive recursive.

✓ 3.21 Recall that a natural number d is a divisor of the natural number a if there is a natural k so that $k \cdot d = a$. A natural number is common divisor of $a, b \in \mathbb{N}$ if it divides them both. The greatest common divisor of the two is the largest of their common divisors.

(A) Euclid observed that if $a > b$ then any common divisor of the two is also a common divisor of $a - b$ and b. Thus, to find $\gcd(a, b)$ we can reduce to finding $\gcd(a - b, b)$ (this is a reduction because the first number is smaller). Use that to produce a recursion that computes $\gcd(a, b)$ for any $a, b \in \mathbb{N}$.

(B) Compute $\gcd(28, 12)$, $\gcd(104, 20)$, and $\gcd(300009, 25)$ using that method.

(C) Euclid also produced this reduction.

$$\gcd(a, b) = \begin{cases} a & \text{– if } b = 0 \\ \gcd(b, \text{rem}(b, a)) & \text{– if } b > 0 \end{cases}$$

where $\text{rem}(b, a)$ is the remainder when b is divided by a. Note that it has the form of the schema of primitive recursion (however, this does not show that it is primitive recursive because we have not yet verified that the remainder function is primitive recursive). Use this method to compute $\gcd(28, 12)$, $\gcd(104, 20)$, and $\gcd(300009, 25)$.

The following three 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 instructions above.*

(A) Constant function: $C_k(\vec{x}) = C_k(x_0, \ldots x_{n-1}) = k$ for a fixed $k \in \mathbb{N}$.

(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 instructions before Exercise 3.22.*

(A) Sign predicate: $\text{sgn}(y)$, which gives 0 if $y = 0$ and gives 1 if y is greater than zero.

(B) Negation of the sign predicate: $\text{negsign}(y)$, which gives 0 if y is greater than zero and 1 if $y = 0$.

(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 instructions before Exercise 3.22.*

(A) Boolean functions: we have the convention that we represent 'true' with 1 and 'false' with 0, and that holds for the outputs here. But for inputs, while we

still take 0 for 'false', we take any positive input to mean 'true'. There is the standard one-input function

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

and the two-input functions.

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

(To avoid being tedious, in the output of the clauses we write 0 to abbreviate $Z(\,)$ while 1 abbreviates $\mathcal{S}(Z(\,))$.)

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

✓ 3.25 *See the instructions 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.

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

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 where the terms $g(i)$ are specified by a single primitive recursive function g, so that $S_g(y) = \sum_{0\leq i<y} g(i) = g(0) + g(1) + \cdots + g(y-1)$ (the sum of zero-many terms is $S_g(0) = 0$). In comparison with the final item of the prior question, while the number of summands is also finite, here it varies with y.

(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 for all $i < m$. The minimization operator $M(\vec{x}, i)$, often written $\min_{i<m}[p(\vec{x}, i)]$ or $\mu i_{i<m}[p(\vec{x}, i)]$, 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 shown to be primitive recursive in this section, or in a prior exercise, or a prior item.

(A) Bounded universal quantification: where $m \in \mathbb{N}$, for each $i < m$ let $p(\vec{x}, i)$ be a predicate. Then $U(\vec{x}, m)$, typically written $\forall i < m\,[p(\vec{x}, i)]$, has value 1 if $p(\vec{x}, 0) = 1$ and ... and $p(\vec{x}, m-1) = 1$. Otherwise, if even one $p(\vec{x}, i)$ is non-1 for $0 \leq i < m$ then $U(\vec{x}, m) = 0$.

(B) Bounded existential quantification: where $m \in \mathbb{N}$, for each $i < m$ let $p(\vec{x}, i)$ be a predicate. Then $E(\vec{x}, m)$, typically written $\exists i \leq m\, [p(\vec{x}, i)]$, has value 1 if $p(\vec{x}, 0) = 1$ or ... or $p(\vec{x}, m-1) = 1$, and has value 0 otherwise.

(C) Divides predicate: where $x, y \in \mathbb{N}$ we have divides(x, y) if there is some $k \in \mathbb{N}$ with $y = x \cdot k$.

(D) Primality predicate: prime(y) if y has no nontrivial divisor.

3.28 We will show that the function 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
rem$(a, 3)$								

(B) Observe that rem$(a+1, 3) =$ rem$(a) + 1$ for many of the entries. When is this relationship not true?

(C) Fill in the blanks.

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

(D) Show that 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 3.4, here the two arguments are switched, which is only a typographic difference.)

(E) Extend the prior item to show that rem(a, b) is primitive recursive.

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

(A) Fill in this table.

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

(B) Much of the time div$(a+1, 3) =$ div$(a, 3)$. Under what circumstance does it not happen?

(C) Show that 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, or a prior exercise such as Exercise 3.22 or Exercise 3.24. (Compared with Definition 3.4, here the two arguments are switched, which is only a difference of appearance.)

(D) Show that 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:* bounded minimization from Exercise 3.26 is a good place to start.

3.31 The examples of primitive recursion in this section and earlier exercises all have $f(y)$ use only one prior value, $f(z) = f(y-1)$. But some recursions use more

than one, such as the Fibonacci recursion $F(y) = F(y-1) + F(y-2)$ that uses two (for Fibonacci, we get the recursion started by defining $F(0) = 1$ and $F(1) = 1$). In a 'course-of-values recursion', the next value $f(y)$ depends on some or all of the prior values $f(y-1), \dots f(0)$. To do these in a primitive recursive way, we get access to the sequence of all prior values by encoding them into a single number. Consider a finite sequence of natural numbers $A = \langle a_0, \dots a_{k-1} \rangle$. Gödel's multiplicative encoding of A is the natural number computed by multiplying factors, where each factor is the i-th prime number raised to the successor of the i-th sequence element.

$$G(A) = 2^{\mathcal{S}(a_0)} \cdot 3^{\mathcal{S}(a_1)} \cdot 5^{\mathcal{S}(a_2)} \cdots p_{k-1}^{\mathcal{S}(a_{k-1})}$$

For the empty sequence, $G(\langle\,\rangle) = 1$. We will sketch how to include all the prior values.

(A) Find $G(A)$ for $A_0 = \langle 3, 1 \rangle$ and $A_1 = \langle 2, 2, 2 \rangle$.

(B) For each number n, find the sequence A where $n = G(A)$, or find that no such sequence exists: $n_0 = 10800$, $n_1 = 12$, and $n_2 = 343$.

(C) Why does the encoding use the successor function?

(D) Where $f\colon \mathbb{N}^{k+1} \to \mathbb{N}$ the course of values function $\bar{f}\colon \mathbb{N}^{k+1} \to \mathbb{N}$ is defined as here for any $\vec{x} \in \mathbb{N}^k$.

$$\bar{f}(\vec{x}, y) = \begin{cases} G(\langle\,\rangle) & \text{– if } y = 0 \\ G(\langle f(\vec{x}, 0), \dots f(\vec{x}, z) \rangle) & \text{– if } z = \mathcal{S}(y) \end{cases}$$

Let $F(n)$ be the n-th Fibonacci number. Find $\bar{F}(0)$, $\bar{F}(1)$, $\bar{F}(2)$, and $\bar{F}(3)$.

✓ 3.32 This is McCarthy's 91 function.

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

(A) What is the output for inputs $x \in \{0, \dots 101\}$? For larger inputs? (You may want to write a small script.) (B) Show that this function is primitive recursive. You may cite the results from this section or from prior exercises.

3.33 Show that every primitive recursive function is total.

Section I.4 General recursion

Every primitive recursive function is intuitively mechanically computable. What about the converse: is every function that is intuitively mechanically computable a primitive recursive function? Here we will answer 'no'.

Ackermann functions We will give a function that is intuitively mechanically computable but that is not primitive recursive. 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.

$$x+y=\underbrace{\mathcal{S}(\,\mathcal{S}(\cdots\mathcal{S}(x)))}_{y\text{ many}} \qquad x\cdot y=\underbrace{x+x+\cdots+x}_{y\text{ many}} \qquad x^y=\underbrace{x\cdot x\cdot\cdots\cdot x}_{y\text{ many}}$$

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, $\mathcal{H}_0=\mathcal{S}$, and then use it as an initial function for these other $\mathcal{H}_i$'s.

$$\text{plus}(x,y)=\mathcal{H}_1(x,y)=\begin{cases}x & \text{– if } y=0\\ \mathcal{H}_0(x,\mathcal{H}_1(x,z)) & \text{– if } y=\mathcal{S}(z)\end{cases}$$

$$\text{product}(x,y)=\mathcal{H}_2(x,y)=\begin{cases}0 & \text{– if } y=0\\ \mathcal{H}_1(x,\mathcal{H}_2(x,z)) & \text{– if } y=\mathcal{S}(z)\end{cases}$$

$$\text{power}(x,y)=\mathcal{H}_3(x,y)=\begin{cases}1 & \text{– if } y=0\\ \mathcal{H}_2(x,\mathcal{H}_3(x,z)) & \text{– if } y=\mathcal{S}(z)\end{cases}$$

The pattern is in the 'otherwise' lines. Each one is $\mathcal{H}_n(x,y)=\mathcal{H}_{n-1}(x,\mathcal{H}_n(x,y-1))$.

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

4.1 DEFINITION This is the hyperoperation $\mathcal{H}\colon\mathbb{N}^3\to\mathbb{N}$.

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

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

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

We prove the level 1 statement $\mathcal{H}_1(x,y)=\mathcal{H}(1,x,y)=x+y$ 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 $y=0,\ldots y=k$ and consider the $y=k+1$ case. The 'otherwise' line says $\mathcal{H}_1(x,k+1)=\mathcal{H}(1,x,k+1)=\mathcal{H}(0,x,\mathcal{H}_1(x,k))=\mathcal{H}_0(x,\mathcal{H}_1(x,k))$. The inductive hypothesis gives $\mathcal{H}_0(x,\mathcal{H}_1(x,k))=\mathcal{H}_0(x,x+k)$. By the prior paragraph this equals $x+k+1=x+y$.

The other two, $\mathcal{H}_2$ and $\mathcal{H}_3$, are Exercise 4.23. □

4.3 Remark Level four 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: $(3^3)^3 = 27^3 = 3^9 = 19\,683$ while $3^{(3^3)} = 3^{27} = 7\,625\,597\,484\,987$. Tetration does it in the second, larger, way. The rapid growth of the output values is a striking aspect of tetration. For instance, $\mathcal{H}_4(4,3) = 4^{4^4}$ 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.

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

The problem is not that the arguments are in a different order; that is cosmetic. The problem is that the definition of primitive recursive function, Definition 3.4, requires that h be a function for which we already have a primitive recursive derivation but here h is the function that we are defining, H.

Of course, just because one definition has the wrong form doesn't mean that no definition with the right form exists. However, Ackermann[†] proved that there is no such definition,

4.4 Theorem The hyperoperation $\mathcal{H}$ is not primitive recursive.

The full details of the proof would a detour for us (see Extra D). In outline, it shows that $\mathcal{H}$ grows faster than any primitive recursive function. More precisely, 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 $\mathcal{H}(n,x,y) > f(n,x,y)$. This proof is about uniformity, or rather lack of it: while at each level n the function $\mathcal{H}_n$ is primitive recursive, no primitive recursive function encompasses all levels at once — there is no primitive recursive way to uniformly compute all primitive recursive functions.

[†] We have seen Ackermann already as one of the people who stated the *Entscheidungsproblem*. A function having a recursion similar to that of $\mathcal{H}$ is an Ackermann function.

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 primitive recursive. So 'primitive recursive' seems to have many of the same characteristics as 'Turing machine computable'. However, we now have an effective function that is not primitive recursive. So the set of primitive recursive functions fails the 'coverage' test from the Church's Thesis discussion.

Wilhelm Ackermann 1896–1962

Consequently we next will expand from the set of primitive recursive functions to a larger collection, which proves to be the same as the set of Turing-computable functions.

μ recursion The prior section's Exercise 3.26 suggests the right direction. Primitive recursion does operations that are bounded, such as bounded sum $\sum_{0\leq i<y} g(i) = g(0)+\cdots+g(y-1)$ and bounded minimization $\min_{i<m}[p(\vec{x},i)] = \mu i_{i<m}[p(\vec{x},i)]$, which returns the smallest $i < m$ such that $p(\vec{x},i) = 0$. We can show that a programming language having only bounded loops computes 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 $g\colon \mathbb{N}^{n+1} \to \mathbb{N}$ is total, so that for every input tuple there is an output number. Then $f\colon \mathbb{N}^n \to \mathbb{N}$ is defined from g by minimization or μ-recursion, written $f(\vec{x}) = \mu y\,\big[g(\vec{x},y) = 0\big]$, if $f(\vec{x})$ is the minimum number y such that $g(\vec{x},y) = 0$.

This is unbounded search. Think of it as examining $g(\vec{x},0)$, then $g(\vec{x},1)$, etc., looking for one of them to give the output 0. If that ever happens, so that $g(\vec{x},y) = 0$ for some least y, then $f(\vec{x}) = y$. If there is no such number then $f(\vec{x})$ is undefined.

4.6 Example Euler noticed that the polynomial $p(y) = y^2 + y + 41$ at least at first output only primes. Does the pattern continue forever?

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

Here is a way to do an unbounded search for non-prime output on a quadratic. Start with this (Euler's p will use $x_0 = 1$, $x_1 = 1$, and $x_2 = 41$).

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

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

This code illustrates. Racket comes with a test for primality.

```
(require math/number-theory)  ;; provides predicate: prime?
```

Use it to define g, which tests whether the quadratic is prime.

```
(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)   ; x0, x1, x2 inherited from enclosing def of f
    (if (= 0 (g x0 x1 x2 y))
        y
        (f-helper (add1 y))))

  (f-helper 0))
```

calls `(f-helper 0)`, then `(f-helper 1)`, etc. It finds an input for which Euler's quadratic p returns a non-prime.

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

All primitive recursive functions are total. But by using the minimization operator we can get functions whose output value is undefined for some or all inputs. For instance, 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 the next example no one currently knows whether the search will end.

4.7 Example Goldbach's conjecture is that every even number greater than two is the sum of two primes. The first few instances are $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5$, and $10 = 3 + 7$. This conjecture is not known to be true, although researchers have confirmed it for all evens up to $y = 4 \times 10^{18}$.

Here we do an unbounded search for a counterexample. This auxiliary function

```
;; 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))
```

helps us get the definition's boolean function g.

```
(define (gb-g y)
  (if (or (odd? y)
          (< y 3))
      1
      (if (gb-check y)
          1
          0)))
```

It returns 0 if the input is even and greater than 2 but there is no a pair of primes. Then here is the unbounded search gb-f() $= \mu y \left[\text{gb-g}(y) = 0 \right]$.

```
(define (gb-f)
  (define (gb-f-helper y)
    (if (= 0 (gb-g y))
        y
        (gb-f-helper (add1 y))))

  (gb-f-helper 0))
```

4.8 EXAMPLE We can expand on that approach. Suppose that we want to settle the open problem Legendre's conjecture, that for every natural number $n > 0$ there is a prime number p with $n^2 < p < (n+1)^2$. Start an unbounded search for a counterexample, but at the same time also run an unbounded search for a proof. After all, a proof is a sequence of statements in a suitable formal language where each statement is either an axiom or follows logically from the prior statements, and where the final statement is the desired theorem. We could use a computer to search for a proof as: for each n, interpret it as a string (perhaps convert n to binary and interpret that binary as a string) and check whether that string is a proof of the theorem. Now wait for one or the other of these searches to halt. Obviously this relates unbounded search to the *Entscheidungsproblem*.

The above discussion makes clear that unbounded search via the μ operator is intuitively mechanically computable. We now define a superset of the primitive recursive functions by adding this function operation.

4.9 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 $\mathcal{Z}(\vec{x}) = 0$, the successor function $\mathcal{S}(x) = x + 1$, and the projection functions $\mathcal{I}_i(\vec{x}) = x_i$ by a finite number of applications of function composition, the schema of primitive recursion, and minimization.

S Kleene showed that this set of functions is the same as the Turing machine-based set of computable functions.

We have seen 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 this is as a search for a halting step.

I.4 Exercises

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.

4.10 What is the difference between total recursive and primitive recursive?

✓ 4.11 Find each: $\mathcal{H}_4(2,0)$, $\mathcal{H}_4(2,1)$, $\mathcal{H}_4(2,2)$, $\mathcal{H}_4(2,3)$, and $\mathcal{H}_4(2,4)$.

✓ 4.12 How many years is $\mathcal{H}_4(3,3)$ seconds?

4.13 What is the ratio $\mathcal{H}_3(3,3)/\mathcal{H}_2(2,2)$?

4.14 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.

4.15 This variant of $\mathcal{H}$ is often called "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 much the same recursion. (Extra D has more about this variant.) Compute $\mathcal{A}(k, y)$ for $0 \leq k < 4$ and $0 \leq y < 7$.

✓ 4.16 Let $g(x, y) = 0$ if $x + y = 100$ and let $g(x, y) = 1$ otherwise. Now let $f(x) = \mu y \left[g(x, y) = 0 \right]$. 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.

4.17 Let $g(x, y) = 0$ if $x \cdot y = 100$ and $g(x, y) = 1$ otherwise. Also let $f(x) = \mu y \left[g(x, y) = 0 \right]$. 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) $f(101)$

✓ 4.18 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 $F_4 = 65\,537$, are prime; these are Fermat primes. But F_5 is not prime, nor are $F_6, \ldots F_{32}$. (We don't know of any primes for higher n, and F_{32} is the highest that researchers have checked.) 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 $f(x) = \mu y \left[g(x, y) = 0 \right]$. For each, what can you say? (A) $f(0)$ (B) $f(1)$ (C) $f(50)$ (D) $f(F_4)$

4.19 Let $g(x, y) = 0$ if y^3 is greater than or equal to x^2, and let $g(x, y) = 1$ otherwise. Also let $f(x) = \mu y \left[g(x, y) = 0 \right]$. Find each, or state 'undefined'. (A) $f(0)$ (B) $f(1)$ (C) $f(50)$ (D) $f(100)$ (E) $f(x)$

✓ 4.20 Two natural numbers are relatively prime if the largest natural number that divides them both is 1. Let notrelprime$(x, y) = 0$ if the two are not relatively prime and let notrelprime$(x, y) = 1$ otherwise. Find each $f(x) = \mu y \left[\text{notrelprime}(x, y) = 0 \right]$. (A) $f(0)$ (B) $f(1)$ (C) $f(2)$ (D) $f(3)$ (E) $f(4)$ (F) $f(42)$ (G) $f(x)$

4.21 Where $x \in \mathbb{R}$ the notation $\lceil x \rceil$ means the least integer that is at least as big as x. Let $g(x, y) = \lceil ((x + 1)/(y + 1)) - 1 \rceil$ and let $f(x) = \mu y \left[g(x, y) = 0 \right]$.

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

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

4.22 In defining general recursive functions, Definition 4.9, 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?

4.23 Finish the proof of Lemma 4.2 by verifying that $\mathcal{H}_2(x, y) = x \cdot y$ and $\mathcal{H}_3(x, y) = x^y$.

4.24 Prove that the computation of $\mathcal{H}(n, x, y)$ always terminates.

✓ 4.25

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

(B) Use that to prove that this function is μ-recursive: $f(n) = 0$ if n is even, and $f(n)\uparrow$ if n is odd.

✓ 4.26 Consider the Turing machine $\mathcal{P} = \{q_0\mathsf{B}1q_1,\ q_0\mathsf{1R}q_0,\ q_1\mathsf{BR}q_2,\ q_1\mathsf{1L}q_1\}$. 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 $f(x) = \mu y\,\big[g(x, y) = 0\big]$ for these. (A) $f(0)$ (B) $f(1)$ (C) $f(2)$ (D) $f(3)$ (E) $f(4)$ (F) $f(5)$

4.27 Define $g(x, y)$ by: start $\mathcal{P} = \{q_0\mathsf{B}1q_2,\ q_0\mathsf{1L}q_1,\ q_1\mathsf{B}1q_2,\ q_1\mathsf{1}1q_2\}$ 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 $f(x) = \mu y\,\big[g(x, y) = 0\big]$. Find $f(x)$ for these. (A) $f(0)$ (B) $f(1)$ (C) $f(2)$ (D) $f(3)$ (E) $f(4)$ (F) $f(5)$

4.28 Consider this Turing machine.

$$\{q_0\mathsf{BR}q_1,\ q_0\mathsf{1R}q_1,\ q_1\mathsf{BR}q_2,\ q_1\mathsf{1R}q_2,\ q_2\mathsf{BL}q_3,\ q_2\mathsf{1L}q_3,\ q_3\mathsf{BL}q_4,\ q_3\mathsf{1L}q_4\}$$

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 $f(x) = \mu y\,\big[g(x, y) = 0\big]$. Find: (A) $f(0)$ (B) $f(1)$ (C) $f(2)$ (D) $f(x)$.

✓ 4.29 Define $h\colon \mathbb{N}^+ \to \mathbb{N}$ 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 $C(n) = \mu k\big[H(n, k) = 1\big]$. (A) Compute $H(4, 1)$, $H(4, 2)$, and $H(4, 3)$. (B) Find $C(4)$, if it is defined. (C) Find $C(5)$, if it is defined. (D) Find $C(11)$, if 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.

4.30 The Ackermann function is intuitively mechanically computable (and total) but is not primitive recursive. Here is an alternative such function. Assume that all partial recursive functions take one natural number input and yield one natural numbers output. (We can simulate input pairs, etc, with Gödel's multiplicative encoding; see Exercise 3.31.) Let f_0, f_1, ... be an effective list of all primitive recursive functions. That is, there is a primitive recursive function that inputs the index i and returns some way of computing f_i. (*Remark:* this is an interpreter for the primitive recursive functions, which given the specification i of the function, can do the computation of $f_i(x)$ for any input x.) Now consider $D(n) = f_n(n) + 1$. Show that D, while intuitively computable, is not primitive recursive.

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.

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

That translates to the input file `pred.tm`.

```
% pred.tm
% Compute predecessor fcn: pred(0)=0 and pred(n)=n-1
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, including its command line invocation. The machine starts with a current symbol of 1 and the tape to the right of the current symbol is 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.

```
computing/src/scheme/prologue$ ./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` gives 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 $\mathcal{P}_{\text{add}}$ 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 $\Sigma = \{\mathtt{0}, \mathtt{1}, \mathtt{B}\}$. 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 machine. However, can we be sure that there is an associated actual mechanism for each transition table, a physical implementation with that behavior?

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

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

Below 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.

P	*not* P $\neg P$
0	1
1	0

P	Q	P *and* Q $P \wedge Q$	P *or* Q $P \vee Q$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Those three logic operators are all we need. We will show how to go from a specified input-output behavior, a desired truth table, to a propositional logic expression having that behavior that uses only '$\neg$', '$\wedge$', and '$\vee$'. Then we will sketch how to implement that with electronic components.

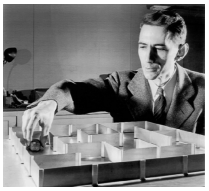
The two tables below show how. Start with the one on the left and focus on the row with output 1. The expression $\neg P \wedge \neg Q$ makes this row take on value 1 and every other row take on value 0.

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

P	Q	R	
0	0	0	0
0	0	1	1
0	1	0	1
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 ending in 1's. For the second row the clause is $\neg P \wedge \neg Q \wedge R$. Target the third row with $\neg P \wedge Q \wedge \neg R$ and the fifth row with $P \wedge \neg Q \wedge \neg R$. Now put these clauses together with $\vee$'s to get the statement with the given table. (A statement consisting of clauses using $\wedge$'s that are joined with $\vee$'s is in Disjunctive Normal Form, DNF. See Section C.)

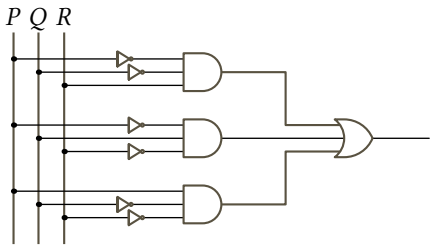
$$(\neg P \wedge \neg Q \wedge R) \vee (\neg P \wedge Q \wedge \neg R) \vee (P \wedge \neg Q \wedge \neg R) \qquad (*)$$



Claude Shannon
1916–2001

Next we translate those expressions into devices. The observation that we can use this form of a propositional logic expression to systematically design logic circuits was made by C Shannon in his 1937 master's thesis. We can get electronic devices, called gates, that perform logical operations on signals. (For this discussion we take a 1 to be signaled by the presence of 5 volts and a 0 to be 0 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.

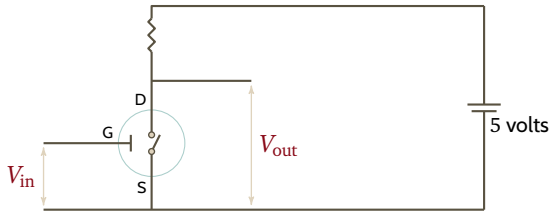
A schematic of a circuit that implements expression $(*)$, given below, shows three input signals on the three wires at left. For instance, to implement the first clause, the top AND gate is fed the not P, the not Q, and the R signals. The second and third clauses are implemented in the other two AND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



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

We will close with an aside. A person can wonder how these gates are constructed, and especially 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. 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 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 $V_{\text{in}} = 0$. Then there is no current flow between D and S. With no current the resistor provides no voltage drop and consequently the output voltage V_{out} across the gap is all of the voltage supplied by the battery, 5 volts. So $V_{\text{in}} = 0$ results in $V_{\text{out}} = 5$.

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

Thus, for this device the voltage out V_{out} is the opposite of the voltage in V_{in}.

I.B Exercises

B.1 A propositional logic operator that is often used is Exclusive Or, XOR. It is defined by: P XOR $Q = 1$ if and only if $P \neq Q$.

(A) Specify a truth table and from it construct a DNF propositional logic expression.

(B) Use that to make a circuit.

B.2 The propositional logic operator Implication, $\rightarrow$, is given by: $P \rightarrow Q$ is 1 except when P is 1 and Q is 0.

(A) Make a truth table and from it construct a Disjunctive Normal Form expression.

(B) Use that to make a circuit.

B.3 For the table below, construct a DNF propositional logic expression.

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

Use that expression to design a circuit.

B.4 For each table construct a propositional logic expression in Disjunctive Normal Form.

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

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

Use the expressions to make a circuit
(A) for the table on the left,
(B) for the one on the right.

B.5 Make a table with inputs P, Q, and R for the behavior that the output is 1 if P equals R. Produce the associated DNF expression. Draw the circuit.

B.6 Make a three-input table for the behavior: the output is 1 if a majority of the inputs are 1's. Produce the associated DNF expression. Draw the circuit.

B.7 Consider the input/output behavior that the output is 1 if a majority of the inputs are 1's (this does not allow ties).
(A) Make a four-input table for the behavior. Produce the associated Disjunctive Normal Form expression.
(B) Also produce the DNF expression for this behavior with five inputs.

B.8 To add two binary numbers the most natural approach works like the grade school decimal addition algorithm. Start at the right with the one's column. Add those two bits 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 1001.
(B) Make a truth table giving the desired behavior in adding the numbers in one column. It must have three inputs because of the possibility of a carry. It must also have two output columns, one for the least significant bit of the sum along with one for any carry.
(C) Draw the circuits.

EXTRA I.C

Game of Life

J von Neumann was one of the twentieth century's most prolific and influential mathematicians. Just in computing, his contributions to developments in hardware are significant enough that the single-memory stored-program architecture is commonly called the von Neumann architecture, and in software he was also an important innovator including inventing merge sort.

John von Neumann 1903-1957

One of the many things he studied was the problem of humans living on Mars. He thought that to colonize Mars we should first terraform it with 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 of that iron, the robots could make more robots. So von Neumann was thinking about making machines that could self-reproduce.[†] A suggestion from his best friend S Ulam led him to explore the topic by computing on a grid, a cellular automaton.

John Conway 1937–2020

Widespread 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.

Start by drawing a two-dimensional grid of square cells, as with graph paper. Each cell has eight neighbors, four that are horizontally or vertically adjacent and four more that are 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 for (1) live cells will die if they are either isolated or overcrowded while for (2), if the environment is just right then the neighbors can reproduce to spread life into this cell.) We begin by seeding the board with some initial pattern, and then watch what develops.

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

> 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.

[†] There is a later Extra on self-reproduction.

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, as Conway says, is a mathematical recreation that is a "zero-player game."

The simplest nontrivial pattern, a single cell, immediately dies.[†]

Generation 0

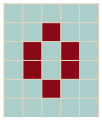
Generation 1

Some other patterns don't die but don't do anything else, either. This 2×2 collection is a block. It is stable from generation to generation.

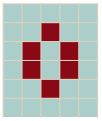
Generation 0

Generation 1

Because it doesn't change, a block is a 'still life'. Another still life is the beehive.

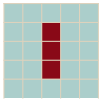


Generation 0

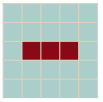


Generation 1

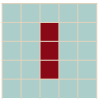
But many patterns are not still. This three-cell pattern, the blinker, does a simple oscillation.



Generation 0

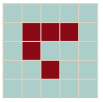


Generation 1



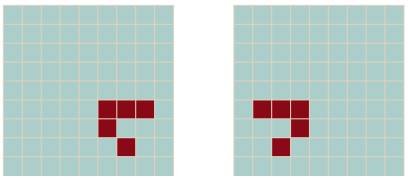
Generation 2

There are other patterns that 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.

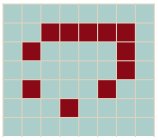
[†] These pictures show the part of the game board containing the cells that are alive.



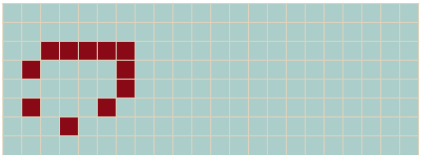
C.1 Animation: Gliding left and gliding right.

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. B Gosper showed that there is, by building the glider gun, which produces a new glider every thirty generations.

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

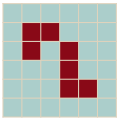


It also crawls across the screen.

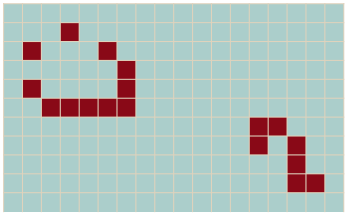


C.2 Animation: Moving across space.

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

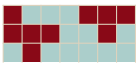


Here it eats a medium weight spaceship.



C.3 Animation: Eating a spaceship.

The behavior of some initial gameboards is extraordinarily complex. For example, a methuselah is a small pattern that stabilizes only after a long time. This pattern is a rabbit. It takes 17 331 turns to stabilize.



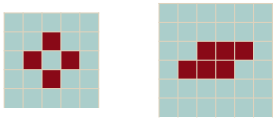
How powerful is the Game of Life as a computational system? Although exhibiting this is beyond our scope, we can build Turing machines in the game and so it is able to compute anything that can be mechanically computed.

To further explore this topic a great site to visit is `conwaylife.com`.

I.C Exercises

For some of these a program to simulate the game will be a help. This book's source has a Life simulator written in Racket under the `src/scheme` *directory. You can also find simulators using a search engine.*

C.4 On the left is the tub and on the right is the toad. One is a still life and one an oscillator. Which is which?



C.5 It is easy to run the clock forward. Can you run the clock back?

C.6 We can ask how rare various behaviors are.

(A) How many 3×3 grids are there? $n \times n$?

(B) How many of the 3×3 patterns will result in any cells on the board that survive into the next generation?

(C) Ten 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 this function is not primitive recursive. Here we will produce a simplified variant and then show that it is not primitive recursive.

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

Rózsa Péter
1905–1977

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

This variant, which is an Ackermann function, is often what authors mean when they discuss 'the' Ackermann function.[†]

This function has only two variables so we can list its first few values with a table.

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

Including the next two entries gives the sense that this function grows very fast indeed.

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

We will prove that $\mathcal{A}$ is not primitive recursive. The intuition is that for any $f\colon \mathbb{N}^n \to \mathbb{N}$ that is primitive recursive, $\mathcal{A}$ grows faster than f.

Recall that if a function has multiple inputs $x_0, \ldots x_{n-1}$ then we may abbreviate that sequence with the vector $\vec{x}$. Also, we will write $\max(\vec{x})$ for $\max(\{x_0, \ldots x_{n-1}\})$. To compare the growth of the two-input function $\mathcal{A}$ with an $f(x_0, \ldots x_{n-1})$, we will look at $\mathcal{A}(k, \max(\vec{x}))$.

The proof's strategy is to show that each primitive recursive function has a natural number level but $\mathcal{A}$ does not — it grows faster than any fixed-level function.

4.1 DEFINITION Where $k \in \mathbb{N}$, a function f is level k if $\mathcal{A}(k, \max(\vec{x})) > f(\vec{x})$ for all $\vec{x}$.

By item E of the following result, if a function is level k then it is also level $\hat{k}$ for any $\hat{k} > k$.

4.2 LEMMA (MONOTONICITY PROPERTIES) (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) $\mathcal{A}(k+2,y) > \mathcal{A}(k,2y)$

[†] Although some authors mean the one-input version $f(x) = \mathcal{A}(x,x)$.

Proof Here we will verify the first item, that $\mathcal{A}(k,y) > y$ for all k and for all y, leaving the others as Exercise D.12. We will do induction on k. The $k = 0$ base step holds because $\mathcal{A}(0,y) = y+1$, and so $\mathcal{A}(0,y) > y$.

For the inductive step, assume that this holds for $k = 0, \ldots n$.

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

We must do the $k = n+1$ case.

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

We will verify ($**$) with an additional induction, this time on y.

For the $y = 0$ base step of this induction inside an induction, the second clause in the definition of $\mathcal{A}$ is that $\mathcal{A}(n+1,0) = \mathcal{A}(n,1)$. By the hypothesis of the outer induction, that statement ($*$) is true when $k = n$, we have $\mathcal{A}(n,1) > 1 > y = 0$.

Still doing the inside induction, for the inductive step assume that statement ($**$) holds for $y = 0, \ldots m$ and consider $y = m+1$. The definition's third clause gives $\mathcal{A}(n+1,m+1) = \mathcal{A}(n,\mathcal{A}(n+1,m))$. The inductive hypothesis ($**$) gives that $\mathcal{A}(n+1,m) > m$. The inductive hypothesis of the outer induction, ($*$), gives that because $\mathcal{A}(n,\mathcal{A}(n+1,m))$ has a second argument of at least $m+1$ then $\mathcal{A}(n+1,m+1) > m+1$, as required. □

D.3 **Theorem (Ackermann, 1925)** For each primitive recursive function f there is a $k \in \mathbb{N}$ such that f is level k.

We shall prove this by structural induction. That is, every such f is derived from a finite number of initial functions via a finite number of applications of the operations of composition and primitive recursion, and we will show the result by induction on this construction. The proof involves three lemmas. We will first show that there is such a k for each initial function. Then we will show that if there is a level number for each function in a composition then the result also has a level number. Finally, we will also show that if there is a level number for the functions used in a primitive recursion then the result also has a level number.

4.4 **Lemma** Each of these initial functions has a level: (1) the zero functions $\mathcal{Z}(\vec{x}) = 0$, (2) the successor functions $\mathcal{S}(\vec{x}) = x+1$, and (3) the projection functions $\mathcal{I}_i(\vec{x}) = \mathcal{I}_i(x_0, \ldots x_{k-1}) = x_i$.

Proof For the first, $k = 0$ suffices by the first clause of the definition of $\mathcal{A}$ since $\mathcal{A}(0,y) = y+1 > \mathcal{Z}(y) = 0$. For item (2), $k = 1$ works because by Lemma 4.2.E $\mathcal{A}(1,y) > \mathcal{A}(0,y) = y+1$. For (3) take $k = 0$ because by the definition's first clause $\mathcal{A}(0,\max(\vec{x})) = \max(\vec{x})+1$ and that is larger than the projection $\mathcal{I}_i(\vec{x})$, as we are taking a maximum. □

4.5 **Lemma** Let each primitive recursive function $g_0, \ldots g_{m-1}, h$ have a level, $k_0, \ldots k_{m-1}, k_m$. Let f be the composition $f(\vec{x}) = h(g_0(\vec{x}), \ldots g_{m-1}(\vec{x}))$. Then f is level $\max(\{k_0, \ldots k_{m-1}, k_m\}) + 2$.

Proof Take $k = \max(\{k_0, \ldots k_{m-1}, k_m\})$. Then all of the functions $g_0, \ldots g_{m-1}, h$ are level k by Lemma 4.2.E.

Lemma 4.2's item C and then the third clause in $\mathcal{A}$'s definition gives this.

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

Focusing on the second argument of the right-hand expression, Lemma 4.2.E and the assumption that each function $g_0,\ldots g_{m-1}$ is level k show that for each function index $i \in \{0,\ldots m-1\}$ we have $\mathcal{A}(k+1,\max(\vec{x})) > \mathcal{A}(k,\max(\vec{x})) > g_i(\vec{x})$. Hence $\mathcal{A}(k+1,\max(\vec{x})) > \max(\{\, g_0(\vec{x}),\ldots g_{m-1}(\vec{x}) \,\})$.

Lemma 4.2.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.

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

The second inequality holds because the function h is level k. □

4.6 LEMMA Assume that the function f is obtained by the schema of primitive recursion

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

where g has level k_g and h has level k_h. Then f has level $\max(\{k_g,k_h\})+3$.

Proof As in the prior argument it will simplify things to take $k=\max(\{k_g,k_n\})$ so that both of the functions g and h are level k.

We will first show this.

$$\mathcal{A}(k+1,\max(\vec{x})+y) > f(\vec{x},y) \qquad (*)$$

We will use induction on y. The $y=0$ base step is that $\mathcal{A}(k+1,\max(\vec{x})+0) = \mathcal{A}(k+1,\max(\vec{x}))$ is greater than $f(\vec{x},0)=g(\vec{x})$, because g is level k.

For the inductive step assume that $(*)$ holds for $y=0,\ldots n$ and consider $y=n+1$. The third clause of A's definition is that $\mathcal{A}(k+1,\max(\vec{x})+n+1) = \mathcal{A}(k,\,\mathcal{A}(k+1,\max(\vec{x})+n))$. The second argument, $\mathcal{A}(k+1,\max(\vec{x})+n)$, is larger than $\max(\vec{x})+n$ by Lemma 4.2.A, and so is larger than any x_i and larger than n. It is also larger than $f(\vec{x},n)$ by the inductive hypothesis.

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

With that, the first inequality below follows from Lemma 4.2.B, monotonicity of $\mathcal{A}$ in its second argument. The second holds because h is a level k function.

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

That ends the verification of $(*)$.

To finish the proof, Lemma 4.2.F gives the first inequality below.

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

The second inequality follows from $2\cdot\max(\{x_0,\ldots x_{m-1},y\}) \geq \max(\vec{x})+y$, and the third is $(*)$. □

4.7 COROLLARY The function $\mathcal{A}$ is not primitive recursive.

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

I.D Exercises

D.8 In base 10, how many digits are in $\mathcal{A}(4,2) = 2^{65536}-3$?

D.9 A classmate asks you, "How does it work that all the levels of $\mathcal{A}$ are primitive recursive but as a whole it is not? Isn't that like saying you have a cake and all the parts are delicious but the cake as a whole is not?"

D.10 Trace through the argument to find a level number k for these primitive recursive functions (it needn't be the least level).
(A) $f(y) = y+2$
(B) $\text{pred}(y) = y-1$ if $y>0$ and $\text{pred}(0)=0$.

D.11 Show that for any k,y the evaluation of $\mathcal{A}(k,y)$ terminates.

D.12 Verify these parts of Lemma 4.2. (A) Item B, $\mathcal{A}(k,y+1) > \mathcal{A}(k,y)$ and in general if $\hat{y}>y$ then $\mathcal{A}(k,\hat{y}) > \mathcal{A}(k,y)$ (B) Item C, $\mathcal{A}(k+1,y) \geq \mathcal{A}(k,y+1)$ (C) Item D, $\mathcal{A}(k,y) > k$ (D) Item E, $\mathcal{A}(k+1,y) > \mathcal{A}(k,y)$ and in general if $\hat{k}>k$ then $\mathcal{A}(\hat{k},y) > \mathcal{A}(k,y)$ (E) Item F, $\mathcal{A}(k+2,y) > \mathcal{A}(k,2y)$

EXTRA

I.E LOOP programs

The primitive recursive functions are a proper subset of the general recursive functions. The latter set 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 is that in a `for` loop we know in advance the number of times that the machine will go through the code inside the loop (above it is three times) — as long as we don't change the value of the loop variable — but a do allows the machine to go through its code an unbounded number of times.

```
> (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 `r0`, `r1`, ... that hold natural numbers. There are four kinds of instructions, which we describe using `r0` and `r1`: (i) `r0 = 0` sets the contents of the register to zero, (ii) `r0 = r0 + 1` increments the contents of the register, (iii) `r1 = r0` copies the contents of `r0` into `r1`, leaving `r0` unchanged, and (iv) `loop r0 ... end` executes a sequence of instructions repeatedly, with the number of repetitions given by the value of the register. For the last, the '...' 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 LOOP program below finishes with the register `r0` holding a value of 4, and with `r1` 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 before running the program.

```
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.

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

Instead, when the loop ends the value in `r0` will be twice what it was when the loop began.

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 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/scheme/prologue$ ./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/scheme/prologue$ ./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
```

[†] Racket version 8.2.

```
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 pred(x),

```
# 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:src/scheme/prologue$ ./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 $x \dot{-} y$,

```
#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:src/scheme/prologue$ ./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 the initial functions 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 $\mathcal{I}_i(x_0, \ldots x_i, \ldots x_{n-1}) = x_i$ is computed by `r0 = ri`.

Composition of two functions is easy. Let $g(x_0, \ldots x_n)$ and $f(y_0, \ldots y_m)$ 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

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

and produces $f(h_0(y_{0,0}, \ldots y_{0,m_0}), \ldots h_n(y_{n,0}, \ldots y_{n,m_n}))$. This needs a little more thought than the two-function case. The issue is that were we to load the inputs $y_{0,0}, \ldots y_{n,m_n}$ into the registers `r0`, `r1`, ... and then immediately begin computing h_0, there would be a danger of overwriting the inputs for later functions such as h_1. For instance, `rotate-shift-right.loop` above used 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}, ... 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 $j + m_0 + \cdots + m_n + 1$. 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.

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

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 $f(x_0, \ldots\ x_{k-1}, t + 1) = h(f(x_0, \ldots x_{k-1}, t),\ x_0, \ldots\ x_{k-1}, t)$ 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`.

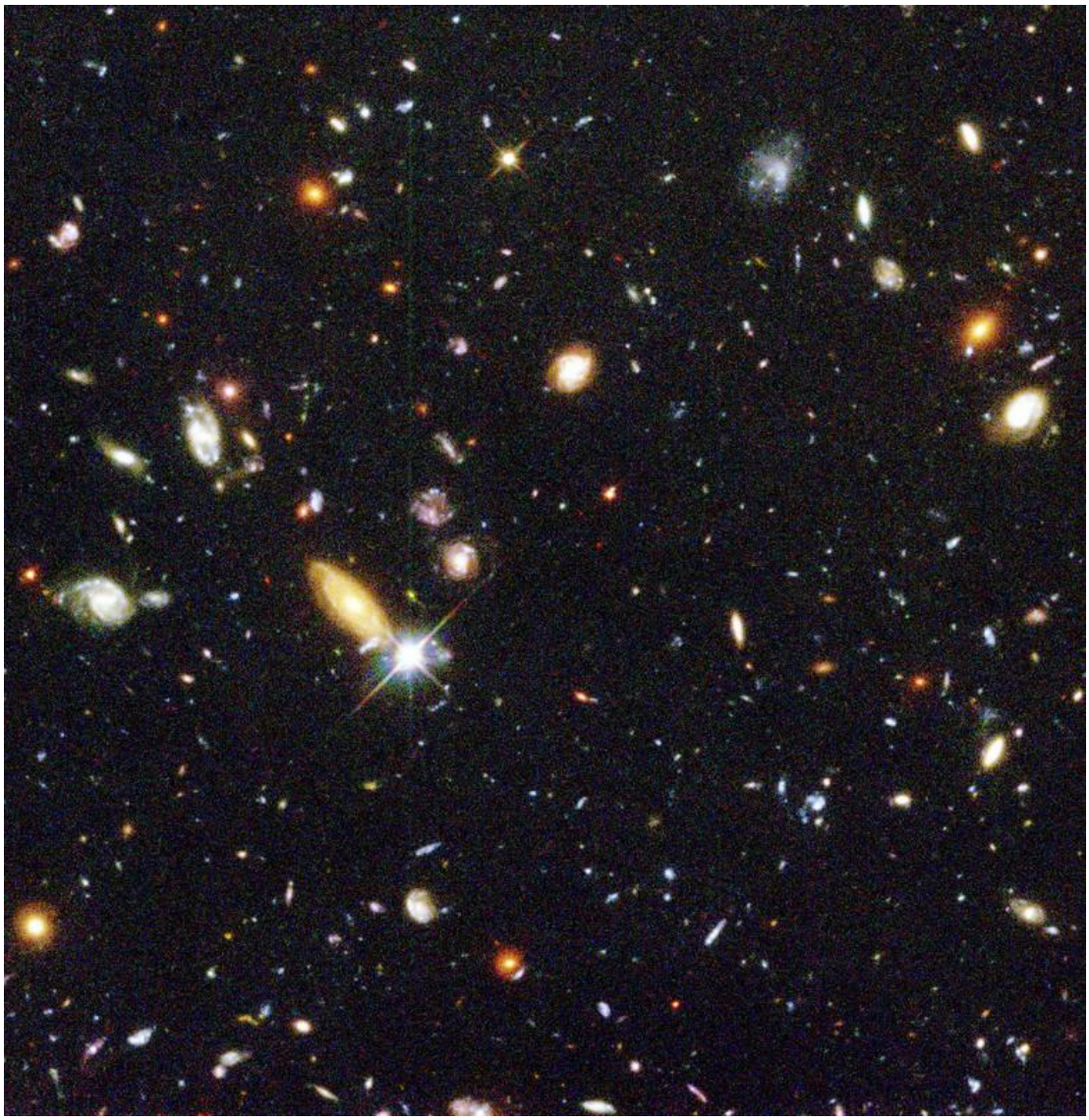
E.4 The program `rotate-shift-right.loop` inputs three numbers, outputs three, and shifts the inputs right (with the third number ending in the first register). Write an three input/three output program that does a rotate shift left. Also write the program that composes the two. What does it compute?

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

E.6 What happens when you try to change a Racket loop variable inside of the loop? For example, what is the behavior of these two?

```
(for ([i (in-range 5)])
  (set! i 1)
  (displayln (~a "i=" i)))
```

```
(for ([i (in-range 5)])
  (displayln (~a "i=" i))
  (set! i 1))
```



Chapter II Background

We want to understand the set of functions that are effective, that are mechanically computable, which we have 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 not computed by any machine — there are jobs that no machine can do. We will first prove this with a counting argument, and later in the chapter we will give specific problems that are unsolvable.

Section II.1 Infinity

We will show that there are more functions $f : \mathbb{N} \to \mathbb{N}$ than Turing machines and that 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.

Galileo Galilei
1564–1642

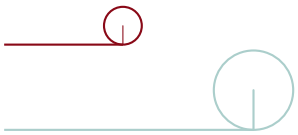
The first puzzle is Galileo's Paradox. It compares the size of the set of perfect squares with the size of the set of natural numbers. The first is a proper subset of the second and so it may seem somehow smaller. However, the figure below shows that the two sets can be made to correspond element-by-element, so in this sense there are exactly as many squares as there are natural numbers.

0	1	2	3	4	5	6	7	8	9	10	11	⋯
↕												
0	1	4	9	16	25	36	49	64	81	100	121	⋯

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 circle inside the larger and roll them, as with 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 at the darkest part of the sky, the very background, for eleven days. It covers an area of the sky about the same width as a dime viewed seventy five feet away. Every speck is a galaxy. There are thousand of them — there is a lot in the background. Credit: Robert Williams and the Hubble Deep Field Team (STScI) and NASA. (Also see the Deep Field movie.)

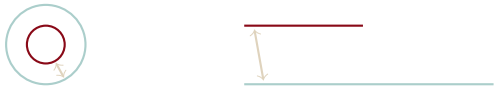


1.2 Animation: Circles of different radiuses have different circumferences.

1.3 Animation: Embedded circles rolling together.

As with Galileo's Paradox, a person might think that the smaller circle's points make a set that is in some way smaller. But point-by-point, the smaller circle corresponds to the larger. The correct view is that the two sets of points have the same number of elements.

The animations below illustrate matching the points in two ways. On the left they are shown as nested circles, with points on the inside corresponding to points on the outside. The second animation straightens that out so that the circumferences make segments, and there 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 the definition of a correspondence as a function that is both one-to-one and onto. A function $f : D \to C$ 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 $y = f(x)$. (Appendix B is a review.) Below, the left map is one-to-one but it is not onto because there is a codomain element with no associated domain element. The right map is onto but not one-to-one since 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. 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.

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

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

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 $\mathbb{N}$, written $|S| = |\mathbb{N}|$. The function $f: \mathbb{N} \to S$ 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 nonnegative, $x_0 = x_1$. 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, $|S_0| = |S_1|$. One correspondence, from S_0 to S_1, is $x \mapsto x + 10$.

1.11 EXAMPLE The set of natural numbers greater than zero, $\mathbb{N}^+ = \{1, 2, \ldots\}$, has the same cardinality as $\mathbb{N}$. A correspondence is $f: \mathbb{N} \to \mathbb{N}^+$ 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 justifies 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 $\{0, 1, \ldots n\}$ for some $n \in \mathbb{N}$, or if it is empty. Otherwise it is infinite.

For us the most important infinite set is the natural numbers, $\mathbb{N} = \{0, 1, 2, \ldots\}$.

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 $f : \mathbb{N} \to S$ 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 $g : \mathbb{N} \to 3\mathbb{N}$ is $g(n) = 3n$.

1.15 EXAMPLE The set $\mathbb{N} - \{2, 5\} = \{0, 1, 3, 4, 6, 7, \ldots\}$ 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	0	1	2	3	4	5	6	...
$f(n)$	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 $p : \mathbb{N} \to P$ 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 $\Sigma = \{\mathtt{a}, \ldots \mathtt{z}\}$. 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 one correspondence, the one that puts the strings 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, ε = ''.)

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

1.18 EXAMPLE The set of integers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ 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	...

Example 1.18 circles back to the paradoxes of infinity. We might naively expect that the positives and the negatives combine make $\mathbb{Z}$ somehow twice as big as $\mathbb{N}$. But this is Galileo's Paradox: the correct way to measure how many elements a set has is not through superset and subset, but through cardinality.

We close by mentioning one more paradox, due to Zeno (c 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

[†] 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.

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 in this chapter we 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 $g\colon \mathbb{N} \to \{3k \mid k \in \mathbb{N}\}$ given by $n \mapsto 3n$ is both one-to-one and onto.

1.21 A friend says, "The perfect squares and the perfect cubes have the same number of elements because these sets are both one-to-one and onto." That's not right; straighten them out.

1.22 Let $f, g\colon \mathbb{Z} \to \mathbb{Z}$ 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) $f\colon \mathbb{N} \to \mathbb{N}$ given by $f(n) = n + 1$
(B) $f\colon \mathbb{Z} \to \mathbb{Z}$ given by $f(n) = n + 1$
(C) $f\colon \mathbb{N} \to \mathbb{N}$ given by $f(n) = 2n$
(D) $f\colon \mathbb{Z} \to \mathbb{Z}$ given by $f(n) = 2n$
(E) $f\colon \mathbb{Z} \to \mathbb{N}$ given by $f(n) = |n|$.

1.24 Decide if each is a correspondence (you must also verify): (A) $f\colon \mathbb{Q} \to \mathbb{Q}$ given by $f(n) = n + 3$ (B) $f\colon \mathbb{Z} \to \mathbb{Q}$ given by $f(n) = n + 3$ (C) $f\colon \mathbb{Q} \to \mathbb{N}$ 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) $\{0, 1, 4, 9, 16, \ldots\}$ (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 $\{1, 2, 3, 4, \ldots\}$ and $\{7, 10, 13, 16 \ldots\}$.

✓ 1.30 Consider the set of characters $C = \{\texttt{0}, \texttt{1}, \ldots \texttt{9}\}$ and the set of integers $A = \{48, 49, \ldots 57\}$.
(A) Produce a correspondence $f : C \to A$.
(B) Verify that the inverse $f^{-1} : A \to C$ 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 $f : \mathbb{R} \to (0 \,..\, \infty)$ 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.

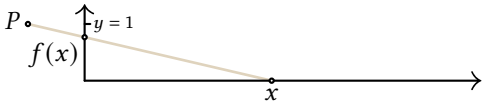
1.36 For each, give functions from $\mathbb{N}$ to itself. You must justify your claims. (A) Give two examples of functions that are one-to-one but not onto. (B) Give two examples of functions that are onto but not one-to-one. (C) Give two that are neither. (D) Give two that are both.

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) $\{0, 1, \ldots 99\}$, $\{m \in \mathbb{N} \mid m^2 < 10\,000\}$ (C) $\{0, 1, 3, 6, 10, 15, \ldots\}$, $\mathbb{N}$

✓ 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, $f\colon (0\,..\,1) \to (0\,..\,\infty)$.



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 $S = \{\, \sqrt[n]{2} \;\big|\; n \in \mathbb{N} \text{ and } n \geq 2 \,\}$ 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 $g\colon I_0 \to I_1$ 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 $f\colon D \to \mathbb{R}$ 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 Galieo'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) $\mathbb{N}$, $\{\, 2n \;\big|\; n \in \mathbb{N} \,\}$ (B) $\mathbb{N}$, $\{\, n \in \mathbb{N} \;\big|\; n > 4 \,\}$

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 $C = \{\text{Spades}, \text{Hearts}, \text{Clubs}, \text{Diamonds}\}$, and let $f\colon D \to C$ be given by $f(0) = \text{Spades}$, $f(1) = \text{Hearts}$, $f(2) = \text{Clubs}$, $f(3) = \text{Diamonds}$. Find the inverse function $f^{-1}\colon C \to D$ and verify that it is a correspondence.

(B) Let $f\colon D \to C$ 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 $f^{-1}\colon C \to D$.

(C) Show that 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.

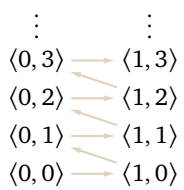
(D) Two finite sets have the same number of elements if and only if there is a correspondence from one to the other.

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 Cartesian product operation, in part because we will want to count Turing machines, which are sets of four-tuples.

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 with 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, 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 association mapping the table's top row and its 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.

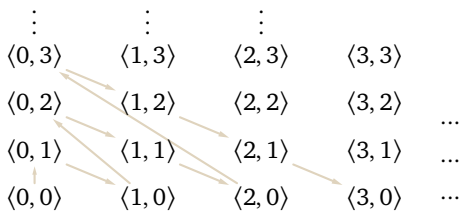
2.3 LEMMA The Cartesian product of two finite sets is finite, and therefore countable. The Cartesian product of a finite set and a countably infinite set, or of a countably infinite set and a finite set, is countably infinite.

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

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

$$\begin{array}{cccccc}
\vdots & \vdots & \vdots & \vdots & \\
\langle 0,3\rangle & \langle 1,3\rangle & \langle 2,3\rangle & \langle 3,3\rangle & \cdots \\
\langle 0,2\rangle & \langle 1,2\rangle & \langle 2,2\rangle & \langle 3,2\rangle & \cdots \\
\langle 0,1\rangle & \langle 1,1\rangle & \langle 2,1\rangle & \langle 3,1\rangle & \cdots \\
\langle 0,0\rangle & \langle 1,0\rangle & \langle 2,0\rangle & \langle 3,0\rangle & \cdots
\end{array}$$

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 $\mathbb{N}\times\mathbb{N}$.

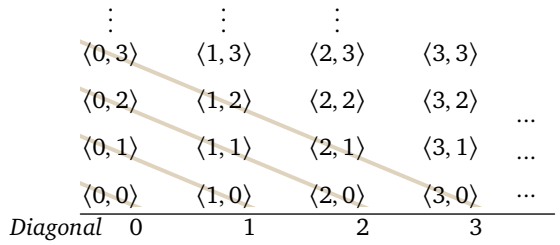
Here is the correspondence as a table.

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

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

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

There is in fact a simple formula for the unpairing function. It is amusing so we will produce it. We first walk through calculating cantor$(\langle 1,2\rangle)$. Start by numbering the diagonals.



The pair $\langle 1,2\rangle$ 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 in Cantor's correspondence. With that, $\langle 1,2\rangle$ is number 7.

To find the number corresponding to $\langle x,y\rangle$, observe that it lies on diagonal $d=x+y$. Prior to diagonal d comes $1+2+\cdots d$ 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 $(x+y)(x+y+1)/2$ in Cantor's correspondence. Next on that diagonal, $\langle 1,x+y-1\rangle$ gets the number $1+[(x+y)(x+y+1)/2]$, etc. In general, $\text{cantor}(x,y)=x+[(x+y)(x+y+1)/2]$.

2.7 Example Two early examples are $\text{cantor}(2,0)=5$ and $\text{cantor}(6,2)=42$. A later one is $\text{cantor}(0,36)=666$.

2.8 Lemma The Cartesian product $\mathbb{N}\times\mathbb{N}$ is countable, for instance under Cantor's correspondence, $\text{cantor}\colon \mathbb{N}^2\to\mathbb{N}$. As well, the sets $\mathbb{N}^3=\mathbb{N}\times\mathbb{N}\times\mathbb{N}$, and $\mathbb{N}^4,\ldots$ are all countable.

Proof The function $\text{cantor}\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ is one-to-one and onto by construction, meaning that 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 $\text{cantor}_3\colon \mathbb{N}^3\to\mathbb{N}$ by $\text{cantor}_3(x,y,z)=\text{cantor}(\text{cantor}(x,y),z)$. Exercise 2.34 shows that this function is a correspondence. With that, the details of the full induction are routine. ∎

2.9 Corollary The Cartesian product of finitely many countable sets is countable.

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

2.10 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 $f\colon \mathbb{N}\to\mathbb{Q}^+\cup\{0\}$. A nonnegative rational number is a numerator-denominator pair $\langle n,d\rangle\in\mathbb{N}\times\mathbb{N}^+$. The complication is that some pairs collapse, such as that $n=10$ and $d=5$ is the

same rational as $n = 2$ and $d = 1$. Similarly, $n = 0$ and $d = 1$ matches $n = 0$, $d = 2$.

We will describe f with an algorithm rather than a formula. Suppose that it is given input i. It will use its prior values $f(0)$, $f(1)$, ... $f(i-1)$. The algorithm loops, applying cantor^{-1} to generate pairs: first $\text{cantor}^{-1}(0) = \langle 0, 0\rangle$, then $\text{cantor}^{-1}(1) = \langle 0, 1\rangle$ ... For each such $\langle a, b\rangle$, if the rational number a/b is among the function's prior values or if the second entry is $b = 0$ (which would give an invalid fraction) then the algorithm skips this pair, going on to the next loop iteration, to generate a new candidate pair. It finds an acceptable $\langle a, b\rangle$ eventually because there are infinitely many rationals, and it outputs the rational $f(i) = a/b$.

2.11 Remark Having a routine save prior values to use later is memoization or caching. It is widely used. For example, when your browser visits a web site it saves any images to your disk so that the next time it visits 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 $f : \mathbb{N} \to S$.

Proof Assume first that S is countable. If it is empty then we are done. If it is finite but nonempty, $S = \{s_0, \ldots s_{n-1}\}$, 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 $f : \mathbb{N} \to S$. 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 $f : \mathbb{N} \to S$. A finite set is countable so we are down to the case where S is infinite. We must produce a correspondence between $\mathbb{N}$ and S so define $\hat{f} : \mathbb{N} \to S$ by $\hat{f}(n) = f(k)$ where k is the least number such that $f(k) \notin \{\hat{f}(0), \ldots \hat{f}(n-1)\}$. Such a k exists because S is infinite and f is onto. This $\hat{f}$ is both one-to-one and onto, by construction. □

2.13 Corollary (1) Any subset of a countable set is countable. (2) The intersection of two countable sets is countable. More generally, 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 For (1), suppose that S is countable and $\hat{S} \subseteq S$. If S is empty then so is $\hat{S}$, and thus it is countable. Otherwise there is an onto $f : \mathbb{N} \to S$. If $\hat{S}$ is empty then it is countable, and if not fix some $\hat{s} \in \hat{S}$. Then this $\hat{f} : \mathbb{N} \to \hat{S}$ 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 $S_0 \cup \varnothing = S_0$. Otherwise, suppose that $f_0 : \mathbb{N} \to S_0$ and $f_1 : \mathbb{N} \to S_1$ are onto. Then count by alternating between the two sets. More precisely, Lemma 2.3 gives a correspondence $g : \mathbb{N} \to \{0, 1\} \times \mathbb{N}$ 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 $S_0 \cup S_1 \cup \cdots$ 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 $g_j : N \to \hat{S}_j$. Then use Cantor's pairing function: the desired onto map from $\mathbb{N}$ to $S_0 \cup S_1 \cup \cdots$ is $\hat{g}(n) = g_j(k)$ where $\text{cantor}^{-1}(n) = \langle j, k \rangle$. □

2.14 Corollary For any countable set, the collection of its finite subsets is countable.

Proof If the set S is empty then the statement is trivial. If not, use an onto map $f : \mathbb{N} \to S$ to put the elements of S into a sequence $\langle s_0, s_1, \ldots \rangle$ with no repeats: take $s_0 = f(0)$, and for each j the number $f(j)$ gets appended to the sequence only if it is not already a member. With that, any finite subsequence $\langle s_{i_0}, s_{i_1}, \ldots s_{i_k} \rangle$ corresponds to the natural number $2^{i_0} + 2^{i_1} + \cdots 2^{i_k}$; for instance, $\langle s_2, s_3, s_7 \rangle$ corresponds to $2^2 + 2^3 + 2^7 = 140$. □

2.15 Corollary Fix an alphabet Σ. There are countably many Turing machines over that alphabet.

Proof The alphabet Σ is finite by Appendix A, and $Q = \{q_0, q_1, \ldots\}$ is countable. So by Corollary 2.9 the set $Q \times \Sigma \times (\Sigma \cup \{\mathtt{L}, \mathtt{R}\}) \times Q$ is countable. Each Turing machine is a finite set of instructions, of members that set. Then the prior result gives that the collection of Turing machines is countable. □

We can do that result one better. Observe that Corollary 2.9 on the Cartesian product of countable sets is effectivizable — if we can count the sets via some effective function then we can count their Cartesian product via an effective function — because the cantor function and its inverse are effective. Therefore there is an effective function that takes in a natural number and outputs the corresponding instruction, and there is also an effective function that takes in an instruction and outputs the corresponding number.

Observe as well that Corollary 2.14 is effectivizable. So we can improve Corollary 2.15 by numbering the Turing machines effectively: there is a program that inputs a Turing machine and outputs a number, as well as a program that inputs a number and outputs a machine, and these two are inverse in that we can

round-trip from the number to the machine and back to the number.

The exact numbering scheme that we use doesn't matter much as long as it is has the properties in the definition below. But for illustration here is an outline of a specific way: starting with a Turing machine $\mathcal{P}$, effectively convert each of its instructions to a number, giving a set $\{i_0, i_1, \ldots i_n\}$. Then define the number e associated with $\mathcal{P}$ to be the one that when written in binary has 1 in bits $i_0, \ldots i_n$, that is, $e = 2^{i_0} + 2^{i_1} + \cdots + 2^{i_n}$. For the inverse, given $e \in \mathbb{N}$, expand it into binary as $e = 2^{j_0} + \cdots + 2^{j_k}$ and the set of instructions corresponding to the numbers j_0, $\ldots$ j_k is the Turing machine. (Except that we must check that the instruction set is deterministic, that no two instructions begin with the same $q_p T_p$. If this is not true then let the machine associated with e be the empty machine, $\mathcal{P} = \{\}$.)

2.16 DEFINITION A numbering is a function that assigns to each Turing machine a natural number. A numbering is acceptable if: (1) there is an effective function 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 there is a program to go from the machine's index to its source, the set of four-tuple instructions, and a program going from the source to the index. Briefy, the index is computationally equivalent to the source.[†]

2.17 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 $f = \phi_{e_0} = \phi_{e_1} = \cdots$. We can effectively produce a list of such indices.

2.18 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 that contains the number 0, or the number 1, etc.

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 $q_{j+k}\mathtt{BB}q_{j+k}$, This gives an effective sequence of Turing machines $\mathcal{P}_{e_1}$, $\mathcal{P}_{e_2}$, $\ldots$ with distinct indices, all having the same behavior, $\phi_{e_k} = f$. □

[†] 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, which we can interpret as a binary number. In the other direction, given a number we take it to be a bitstring, and dissasemble it into machine code source. (One problem with this approach is that if the first character in the source is represented by binary 0 then in passing to a binary number that information is lost. There are patches for the problems but they reduce the intuitive appeal so while this idea is helpful, it is best left informal.)

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

II.2 Exercises

✓ 2.19 Extend the table of Example 2.1 through $n = 12$. Where $f(n) = \langle x, y \rangle$, give formulas for x and y.

✓ 2.20 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) $\langle 200, 0 \rangle$

✓ 2.21 Corollary 2.13 says that the union of two countable sets is countable.

(A) For the sets $T = \{ 2k \mid k \in \mathbb{N} \}$ and $F = \{ 5m \mid m \in \mathbb{N} \}$ produce a correspondence $f_T : \mathbb{N} \to T$ and $f_F : \mathbb{N} \to F$. Give a table listing the values of $f_T(0), \ldots f_T(9)$ and give another table listing $f_F(0), \ldots f_F(9)$.

(B) Give a table listing the first ten values for a correspondence $f : \mathbb{N} \to T \cup F$.

2.22 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 $\langle 200, 1 \rangle$.

✓ 2.23 Example 2.1 says that the method for two columns extends to three. Give a function enumerating $\{0, 1, 2\} \times \mathbb{N}$. That is, where $f(n) = \langle x, y \rangle$ give a formula for x and y as functions of n. Find the pair corresponding to 0, 10, 100, and 1 000. Find the number corresponding to $\langle 1, 2, 3 \rangle$, $\langle 1, 20, 300 \rangle$, and $\langle 1, 200, 3000 \rangle$.

2.24 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 the formula the general case of an enumeration of $\{0, 1, 2, \ldots k\} \times \mathbb{N}$.

✓ 2.25 Extend the table of Example 2.4 to cover correspondences up to 16.

✓ 2.26 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, $\text{pair} : \mathbb{N} \to \mathbb{N}^2$, is also effective by sketching a way to compute it with a program.

2.27 Prove that if A and B are countable sets then their symmetric difference $A \Delta B = (A - B) \cup (B - A)$ is countable.

2.28 Show that the subset $S = \{ a + bi \mid a, b \in \mathbb{Z} \}$ of the complex numbers is countable.

2.29 List the first dozen nonnegative rational numbers enumerated by the method described in Example 2.10.

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 subset.

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

(A) Fix a natural number n. Prove that the set of degree n polynomials $\mathbb{Z}_n[x] = \{ a_n x^n + \cdots + a_0 \mid a_n, \ldots a_0 \in \mathbb{Z} \}$ is countable.

(B) Finish the argument.

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

✓ 2.34 The proof of Lemma 2.8 says that the function $\text{cantor}_3 : \mathbb{N}^3 \to \mathbb{N}$ given by $\text{cantor}_3(a, b, c) = \text{cantor}(\text{cantor}(a, b), c)$ is a correspondence. Verify that.

2.35 Define $c_3 : \mathbb{N}^3 \to \mathbb{N}$ 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, 4, and 5. (C) Give a formula.

2.36 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.

2.37 A binary sequence is an infinite bitstring, that is, we can think of it as a list $b = \langle b_0, b_1, \ldots \rangle$ or as a function $b : \mathbb{N} \to \mathbb{B}$. Suppose that we consider two binary sequences equivalent if they have the same tail, so that $b \equiv \hat{b}$ if there is an N so that $i \geq N$ implies $b(i) = \hat{b}$. Show that for any b, the number of equivalent binary sequences is countably infinite.

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 $f : S \to T$ 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 $g : \mathbb{N}^2 \to \mathbb{N}$ 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. Produce a table of $B(x, y)$ for $0 \leq x, y \leq 4$. To what value does $(3, 4)$ correspond?

2.44 Use Lemma 2.12 to give a much slicker, and shorter, proof that the rational numbers are countable than the one in Example 2.10.

2.45 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. There are pairs of infinite sets that do not correspond. To demonstrate this we now introduce a very powerful technique. Our interest in this technique goes far beyond this result — it is central to the subject.

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 $\mathbb{R}$.

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

We start by illustrating the proof's technique. The table below shows a function $f : \mathbb{N} \to \mathbb{R}$, listing some inputs and outputs, with the outputs aligned on the decimal point.

Input n	*Decimal expansion of output* $f(n)$
0	42.3127704...
1	2.0100000...
2	1.4141592...
3	−20.9195919...
4	0.1010010...
5	−0.6255418...
⋮	⋮

We will show that this function is not onto by producing a number $z \in \mathbb{R}$ that does not equal 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 in that decimal place z gets a 2, while otherwise z gets a 1 there. Thus, in this example $z = 0.121112\ldots$

By construction, z differs from what's 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 $f(n)$. Therefore f is not onto. This technique is diagonalization.

Proof We will show that no map $f\colon \mathbb{N} \to \mathbb{R}$ 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 $\{0, \ldots, 9\}$ 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. □

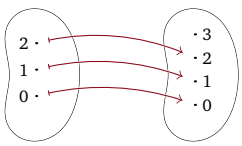
(We have skirted a technicality, that some real numbers have two different decimal representations. For instance, $1.000\ldots = 0.999\ldots$ because the two differ by less than 0.1, less than 0.01, etc. This is a potential snag to the argument because it means that even though we have constructed a representation that is different than all the representations on the list, it still might not be that the number z is different than all the numbers $f(n)$ 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.)

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 other infinite sets. In particular, the reals have more elements than the naturals. As with Galileo's Paradox, it 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.

There is an analogy with the children's game of Musical Chairs. We have countably many chairs $P_0, P_1, \ldots$ but there are so many children — so many reals — that at least one 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 function 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 $\iota\colon \mathbb{N} \to \mathbb{R}$ 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.

3.6 Remark 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 3.32.

For the next result, recall that for a set S, the 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 characters $S = \{\mathsf{a}, \mathsf{c}\}$, the characteristic function with domain $\Sigma = \{\mathsf{a}, \ldots, \mathsf{z}\}$ is $\mathbb{1}_S(\mathsf{a}) = T$, $\mathbb{1}_S(\mathsf{b}) = F$, $\mathbb{1}_S(\mathsf{c}) = T$, $\mathbb{1}_S(\mathsf{d}) = F$, ... $\mathbb{1}_S(\mathsf{z}) = F$.

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

3.7 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 $\mathscr{P}(S)$ 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 $\mathscr{P}(S)$ is a correspondence. For an example of this half consider the set $S = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. We will walk through how we prove that the function $f : S \to \mathscr{P}(S)$ below is not onto.

$$\mathsf{a} \overset{f}{\longmapsto} \{\mathsf{b}, \mathsf{c}\} \qquad \mathsf{b} \overset{f}{\longmapsto} \{\mathsf{b}\} \qquad \mathsf{c} \overset{f}{\longmapsto} \{\mathsf{a}, \mathsf{b}, \mathsf{c}\} \tag{$*$}$$

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

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

We will show that f is not onto by producing a member of $\mathscr{P}(S)$ that is not any of the three output sets in $(*)$. For that, diagonalize. Take the table's diagonal FTT and flip the values to get TFF. That describes the characteristic function of the set $R = \{\mathsf{a}\}$. The set R is not equal to the set $f(\mathsf{a})$ because their characteristic functions differ on a. Similarly, R is not the set $f(\mathsf{b})$ because they differ on b, and R is not $f(\mathsf{c})$ because they differ on c. So R is not in the range of f and consequently f is not onto.

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

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

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 R, 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. □

3.8 COROLLARY The cardinality of the set $\mathbb{N}$ is strictly less than the cardinality of the set of natural number functions $f : \mathbb{N} \to \mathbb{N}$.

Proof Let the set of functions be G. There is a correspondence between $\mathscr{P}(\mathbb{N})$ and G, namely the one that associates each subset $S \subseteq \mathbb{N}$ with its characteristic function, $\mathbb{1}_S : \mathbb{N} \to \mathbb{N}$. Therefore $|\mathbb{N}| < |\mathscr{P}(\mathbb{N})| \leq |G|$. □

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

Proof Lemma 2.8 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 — no association of Turing machines with natural number functions is onto. In particular, when we associate each Turing machine with the function that it computes, that association is not onto. There is a natural number function that is without a Turing machine to compute it. □

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 who has taken programming class, where they repeatedly go from a task to the program that does the task, the existence of things that cannot be done may be a surprise, perhaps even a shock. One point that these results make is that while the work here on sizes of infinities could at first seem impracticably abstract, it nevertheless leads to interesting and useful conclusions.

II.3 Exercises

3.10 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.

3.11 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?

3.12 Verify Cantor's Theorem, Theorem 3.7, for these finite sets. (A) $\{0, 1, 2\}$ (B) $\{0, 1\}$ (C) $\{0\}$ (D) $\{\}$

✓ 3.13 Use Definition 3.4 to prove that the first set has cardinality less than or equal to the second.
(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

3.14 One set is countable and the other is uncountable. Which is which? (A) $\{n \in \mathbb{Z} \mid n + 3 < 5\}$ (B) $\{x \in \mathbb{R} \mid x + 3 < 5\}$

✓ 3.15 Characterize each set as countable or uncountable. You need only give a one-word answer. (A) $(5 \,..\, \infty) \subset \mathbb{R}$ (B) $[1 \,..\, 4) \subset \mathbb{N}$ (C) $[1 \,..\, 4) \subset \mathbb{R}$ (D) $[5 \,..\, \infty) \subset \mathbb{N}$

3.16 List all of the functions with domain $A_2 = \{0, 1\}$ and codomain $\mathscr{P}(A_2)$. How many functions are there for a set A_3 with three elements? n elements?

3.17 List all of the functions from S to T. Which are one-to-one? (A) $S = \{0, 1\}$, $T = \{10, 11\}$ (B) $S = \{0, 1\}$, $T = \{10, 11, 12\}$

✓ 3.18 Short answer: fill each blank, giving the sharpest conclusion possible, 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 one blank or not at all). 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 ____.

3.19 Short answer: Consider $f : S \to T$ where S is countable. For the two 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: (A) the map is onto, (B) the map is one-to-one.

3.20 Let $S = [0 \,..\, 1]$ and $T = (0 \,..\, 1)$ be intervals of reals. Give a correspondence. *Hint:* $f(x) = x$ works for most inputs but the endpoints have to go somewhere, so start with $f(0) = 1/2$ and $f(1) = 1/4$ and follow the consequences.

✓ 3.21 Give a set with a larger cardinality than $\mathbb{R}$.

✓ 3.22 Recall that we write the set of bits as $\mathbb{B} = \{0, 1\}$.
(A) Show that the set of finite bit strings, $\langle b_0 b_1 \ldots b_{k-1} \rangle$ where $b_i \in \mathbb{B}$ and $k \in \mathbb{N}$, is countable.
(B) We can think of an infinite bit string $f = \langle b_0, b_1, \ldots \rangle$ as a function $f : \mathbb{N} \to \mathbb{B}$. Show that the set of infinite bit strings is uncountable, using diagonalization.

3.23 Prove that for two sets, $S \subseteq T$ implies $|S| \leq |T|$.

3.24 Use diagonalization to show that this is false: all functions $f : \mathbb{N} \to \mathbb{N}$ with finite range are computable. *Hint:* it is enough to show that the collection of such functions is not countable.

3.25 In mathematics classes we mostly work with rational numbers, perhaps leaving the impression that irrational numbers are uncommon. But actually, there are more irrational numbers than rationals. Prove that while the set of rational numbers is countable by Example 2.10, the set of irrational numbers is uncountable.

✓ 3.26 Example 2.10 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 $q_0, q_1, \ldots$ that contains all of the rationals. Represent each of those numbers with a decimal expansion $q_i = d_i.d_{i,0}d_{i,1}d_{i,2}\ldots$ (where $d_i \in \mathbb{Z}$ and $d_{i,j} \in \{0, \ldots 9\}$) 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 $0, 1, \ldots 9$ given by $g(1) = 2$, and $g(i) = 1$ if $i \neq 2$. Consider the number down the diagonal, $d = \sum_{n\in\mathbb{N}} d_{n,n} \cdot 10^{-(n+1)}$. Transform its digits using g, that is, 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 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?

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

3.28 The key to the proof of Cantor's Theorem, Theorem 3.1 is the definition of $R = \{s \mid s \notin f(s)\}$. 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 consist of those who are not on their own list. Show that no student's list equals H.

3.29 Show that there is no set of all sets. *Hint:* use Theorem 3.7.

3.30 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 $0.01000\ldots$ and $0.00111\ldots$. But in a base two argument, when building z there is no way to avoid the digits 0 and 1. How could you make the argument work in base two?

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

(A) Verify this equality by using the formula for an infinite geometric series, $a + ar + ar^2 + ar^3 + \cdots = a/(1-r)$.

(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.

3.32 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

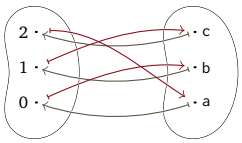
by iterating application of the two functions, both to the right and the left, as here.

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

(Starting with s the chain to the right is s, $f(s)$, $g(f(s))$, $f(g(f(s)))$, ... while the chain stretching to the left is ... $f^{-1}(g^{-1}(s))$, $g^{-1}(s)$, 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 $T = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, and consider the two one-to-one functions $f : S \to T$ and $g : T \to S$ shown here.

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



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

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

(A) Consider $S = \{0, 1, 2, 3\}$ and $T = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$. Let f associate $0 \mapsto \mathsf{a}$, $1 \mapsto \mathsf{b}$, $2 \mapsto \mathsf{d}$ and $3 \mapsto \mathsf{c}$. Let g associate $\mathsf{a} \mapsto 0$, $\mathsf{b} \mapsto 1$, $\mathsf{c} \mapsto 2$ and $\mathsf{d} \mapsto 3$. 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 $f : S \to T$ be $f(x) = x + 1$ and let $g : T \to S$ 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 $f : S \to T$ and $g : T \to S$ 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.

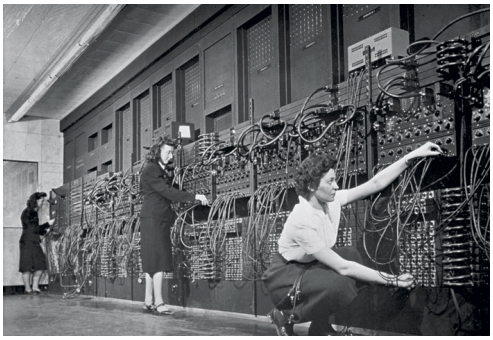
(E) Show that for any chain the function below is a correspondence between the chain's elements from S and its elements from T.

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

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



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 card decks.

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

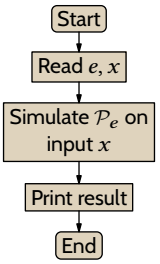
An ouroboros, a snake swallowing its own tail

Before stating Turing's theorem, we address an often-asked question. The machine $\mathcal{UP}$ 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 $\mathcal{UP}$ 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 to machines; we feed them symbols, representations of things. True, we can feed $\mathcal{UP}$ a specification of itself, such as a pair e, x where e is the index number of $\mathcal{UP}$ 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 $\mathcal{UP}$ its own index number. Lots of interesting things happen as a result, but for a start 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.[†] This figure[‡] outlines the action of such a machine.



Before our argument for their existence we first observe that universal machines are familiar from everyday computing. For one thing, we can compare this flow chart with the behavior of a computer operating system. An operating system is given a program to run and some data to feed to that program. Think of the program as $\mathcal{P}_e$ and the data as a bitstring that we can interpret as a number, x. The operating system arranges that the underlying hardware will behave like machine e, with input x. In short, as with an operating system, Universal Turing machines change their behavior in software. No patch chords.

Another everyday computing experience that is like a universal machine is a language interpreter. Below is an interaction with the Racket interpreter. At the first prompt we type in a routine that takes `x` and returns the sum of the first `x` numbers. At the second prompt we specify the input to that routine, $x = 4$.

```
$ racket
Welcome to Racket v8.2 [cs].
> (define (triangular x)
      (if (= x 0)
          0
          (+ x (triangular (sub1 x)))))
> (triangular 4)
10
```

The most direct example of computing systems that act as universal machines is

[†] We could also define a Universal Turing machine to take the single-number input cantor(e, x). [‡] This is a flowchart, which gives a high level sketch of a routine. We use three types of boxes. Boxes with rounded corners are for `Start` and `End`. Rectangles are for ordinary operations on data. In later charts we will also see diamond boxes, which are for decisions, `if` statements.

a language's `eval` statement. At the first prompt below we define a routine that 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 `lambda (i) ...` describes a function of one input.[†] In the third and fourth prompts, the interpreter evaluates the routine that is described in that list and applies it to the numbers 5 and 0. That is, as with the loom's punched cards, we can make `utm` behave differently by giving it a description of whatever routine is desired.

```
> (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 used a Turing machine simulator program, 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 output its digits. More precisely, produce a Turing machine $\mathcal{P}_r$ 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 while there are uncountably many real numbers, there are 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 — why can't write a routine that will output whatever digits we like?

There certainly are real numbers for which there is such a routine. One is 11.25.

```
(define (one-quarter-decimal-places n)
  (cond
      [(= n 0) 11]
      [(= n 1) 2]
      [(= n 2) 5]
      [else 0]))
```

For a more generic number, say, some $r = 0.703\ldots$, 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]
    ...
  ))
```

[†] It uses 'lambda' to start the definition of a function because that's what Church used.

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 $\pi = 3.14159\ldots$ there are n consecutive nines.

There are two possibilities. Either for all n such a sequence exists, or else there is some N where a sequence of nines exists for lengths less than N and no sequence exists when $n \geq N$. Consequently the problem is solved: one of the two below is the right program (for illustration here we take $N = 1234$).

```
(define (sequence-of-nines-0 n)
  1)
```

```
(define (sequence-of-nines-1 n)
  (if (< n 1234)
      1
      0))
```

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 shows that the statement "something is computable if you can write a program for it" might be a useful first take but at the least omits some important subtleties.

In contrast, imagine that we have a routine `pi_decimals` 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 π's digits, looking for n consecutive nines. This approach has the advantage that it doesn't just say whether the answer exists, it constructs that answer. This approach is also uniform in the sense that we could modify it to use other routines such as `e_decimals` and so look for strings of nines in other numbers. However this approach has the disadvantage that if there is an N where π does not have n consecutive nines for $n \geq N$ then this program will search without bound, never 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 $\phi\colon \mathbb{N}^2 \to \mathbb{N}$ 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 so that $\phi(e,x)$ becomes $\phi_e(x)$. 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 then 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.

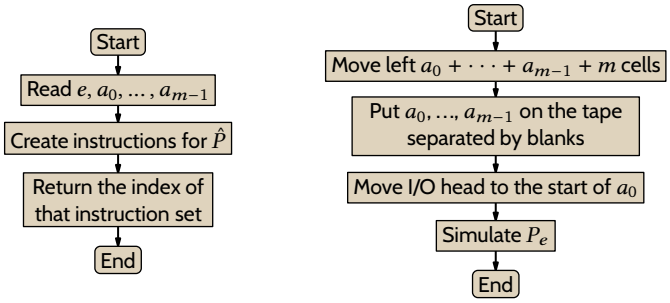
The programs in the family are related to the starting one, obviously. Denoting the function computed by the above starting program `P` as $\psi(x,y) = x + y$, partial application gives the family of functions: $\psi_0(y) = y$, $\psi_1(y) = 1 + y$, $\psi_2(y) = 2 + y$, ... The next result says that in general, from the index of a starting Turing machine or computable function and from the values that are frozen, we can compute the indices of 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 $s_{m,n}\colon \mathbb{N}^{1+m} \to \mathbb{N}$ such that for an $m+n$-ary function $\phi_e(x_0, \ldots x_{m-1}, x_m, \ldots x_{m+n-1})$, freezing the initial m variables at $a_0, \ldots a_{m-1} \in \mathbb{N}$ gives the n-ary computable function $\phi_{s(e,a_0,\ldots a_{m-1})}(x_m, \ldots x_{m+n-1})$.

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 $a_0, \ldots a_{m-1}$, and it must output the index of a machine $\hat{\mathcal{P}}$ that, when given the input $x_m, \ldots x_{m+n-1}$, will return the value $\phi_e(a_0, \ldots a_{m-1}, x_m, \ldots x_{m+n-1})$, or fail to halt if that function diverges.

The idea is that the machine that computes s will construct the instructions for $\hat{\mathcal{P}}$. We can get effectively from the instruction set to the index, 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, $\hat{\mathcal{P}}$, sketched on the right. The machine on the left needs $a_0, \ldots a_{m-1}$ for the right side's second, third, and fourth boxes, and it needs e for $\hat{\mathcal{P}}$'s fifth box. (In this book we try to avoid getting entangled in the detail of the convention for representations for inputs and outputs of Turing machines. However to be as clear as possible in the right side's flowchart, here we assume that its 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.)

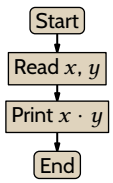


Finally, observe that the routine on the left does not run the routine on the right. Instead, it returns the index of a Turing machine, $\hat{\mathcal{P}}$, implementing that right chart. It can produce the source code for the right chart without knowing whether that routine will halt, and from that source it can compute the index. So the function computed by the left chart, s, is total. □

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 usually omit them.

The key point about the *s-m-n* theorem is that it gives not just one computable function but instead gives a family of related functions.

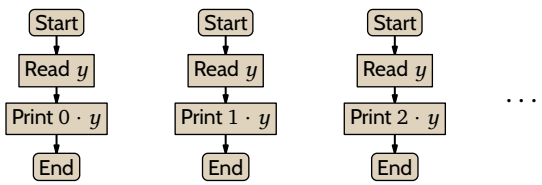
4.5 Example Consider the two-input routine sketched by this flowchart.



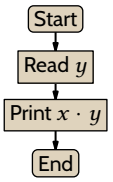
(∗)

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 on e_0 emphasizes that this index is fixed.)

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 family member, $\mathcal{P}_{s(e_0,x)}$.



(**)

Compare (**) to (*). The difference is that the machine in (**) does not read x; rather, thinking of these as programs instead of Turing machines, x is hard-coded into the source body.

That example shows the *s-m-n* Theorem taking us from the single computable function ϕ_{e_0} to the sequence of functions $\phi_{s(e_0,x)}$. This is a family of related functions that is is parametrized by x, since e_0 is fixed.

Restated, this family is uniformly computable — the single computable function s (more precisely, $s_{1,1}$) goes from the index e_0 and the parameter value x to the index of the result in (**). So the *s-m-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?

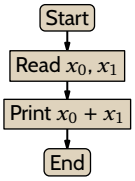
✓ 4.11 Consider the function $f(x_0, x_1) = 3x_0 + x_0 \cdot x_1$.

(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?

4.12 Consider $f(x_0, x_1, x_2) = x_0 + 2x_1 + 3x_2$.

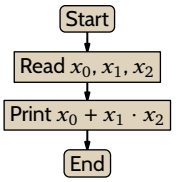
(A) Freeze x_0 to have the value 1. What is the resulting two-variable function?
(B) What two-variable function results from fixing x_0 to be 2?
(C) Let a be a natural number. What two-variable function results from fixing x_0 to be a?
(D) Freeze x_0 at 5 and x_1 at 3. What is the resulting one-variable function?
(E) What one-variable function results from fixing x_0 to be a and x_1 to be b, for $a, b \in \mathbb{N}$?

✓ 4.13 Suppose that the Turing machine sketched by this flowchart has index e_0.



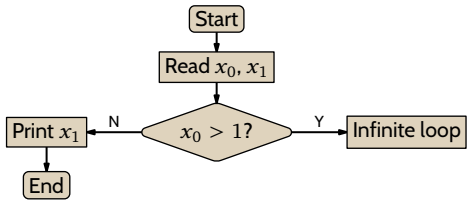
(A) Describe the function $\phi_{s_{1,1}(e_0,1)}$.
(B) What are the values of $\phi_{s_{1,1}(e_0,1)}(0)$, $\phi_{s_{1,1}(e_0,1)}(1)$, and $\phi_{s_{1,1}(e_0,1)}(2)$?
(C) Describe the function $\phi_{s_{1,1}(e_0,0)}$.
(D) What are the values of $\phi_{s_{1,1}(e_0,0)}(0)$, $\phi_{s_{1,1}(e_0,0)}(1)$, and $\phi_{s_{1,0}(e_0,0)}(2)$?

4.14 Let the Turing machine sketched by this flowchart have index e_0.



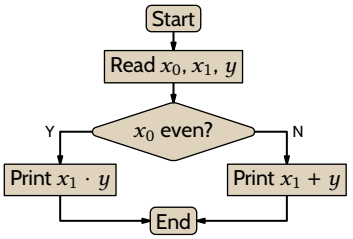
(A) Describe the function $\phi_{s_{1,2}(e_0,1)}$.
(B) Find $\phi_{s_{1,2}(e_0,1)}(0,1)$, $\phi_{s_{1,2}(e_0,1)}(1,0)$, and $\phi_{s_{1,2}(e_0,1)}(2,3)$
(C) Describe the function $\phi_{s_{2,1}(e_0,1,2)}$.
(D) Find $\phi_{s_{2,1}(e_0,1,2)}(0)$, $\phi_{s_{2,1}(e_0,1,2)}(1)$, and $\phi_{s_{2,1}(e_0,1,2)}(2)$.

✓ 4.15 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 $\phi_{s_{1,1}(e_0,2)}(5)$.

✓ 4.16 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) As in the *s-m-n* theorem, 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) Describe $\phi_{s(e_0,1,3)}$. What is $\phi_{s(e_0,1,3)}(5)$?

(C) Describe $\phi_{s(e_0,a,b)}$.

✓ 4.17 Show that there is a total computable function $g\colon \mathbb{N} \to \mathbb{N}$ such that Turing machine $\mathcal{P}_{g(n)}$ computes the function $y \mapsto y + n^2$.

✓ 4.18 Show that there is a total computable function $g\colon \mathbb{N}^2 \to \mathbb{N}$ such that Turing machine $\mathcal{P}_{g(m,b)}$ computes $x \mapsto mx + b$.

✓ 4.19 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 cantor(e, x) then it returns the same value as $\mathcal{P}_e(x)$. 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) $\phi_{e_0}(\text{cantor}(e_0, \text{cantor}(e_1, 5)))$

4.20 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, briefly say why not).

(A) $\phi_{e_0}(4)$

(B) $\phi_{e_0}(\text{cantor}(e_1, 4))$

(C) $\phi_{e_0}(\text{cantor}(4, e_1))$

(D) $\phi_{e_1}(\text{cantor}(e_0, 4))$

4.21 Write a program that reads a file and returns its first character. Apply that program to its own source.

4.22 Write a self-modifying program. Specifically, write a program that sets a variable `COUNTER`, and is such that when run the program will increment that variable in the source, and also print out the new value.

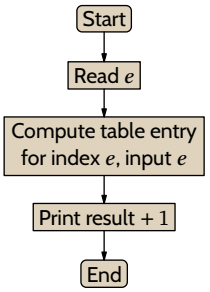
Section II.5 The Halting problem

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 74. Imagine that this table's rows are the computable functions and its columns are the inputs. For instance, this table lists $\phi_2(3) = 5$.

		Input							
		0	1	2	3	4	5	6	...
Function	ϕ_0	3	1	2	7	7	0	4	...
	ϕ_1	0	5	0	0	0	0	0	...
	ϕ_2	1	4	1	5	9	2	6	...
	ϕ_3	9	1	9	1	9	1	9	...
	ϕ_4	1	0	1	0	0	1	0	...
	ϕ_5	6	2	5	5	4	1	8	...
	⋮				⋮				



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 this machine.

But that's a puzzle. The flowchart outlines an effective procedure — we can implement this using a Universal Turing machine in its third box — and thus it seems that its output should be one of the rows.

What's the puzzle's resolution? The flowchart's first, second, fourth, and fifth boxes are trivial so the answer must involve the third one. There must be an $e \in \mathbb{N}$ so that $\phi_e(e)\uparrow$, so that for that number the machine in the flowchart never gets through its middle box, and consequently never gives any output. That is, to avoid a contradiction the above table must contain $\uparrow$'s.

So this puzzle has led to a key insight: the fact that some computations fail to halt on some inputs is critical to the entire subject.

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.

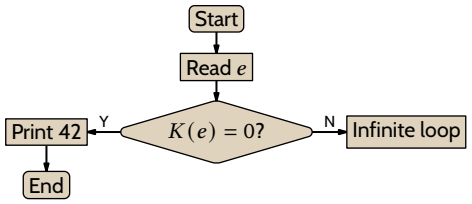
Proof Assume otherwise, that there exists a Turing machine with this behavior.

$$\mathbb{1}_K(e) = K(e) = \texttt{halt\_decider}(e) = \begin{cases} 1 & \text{– if } \phi_e(e)\downarrow \\ 0 & \text{– if } \phi_e(e)\uparrow \end{cases}$$

That assumption implies that the function f below is also mechanically computable. (Below, the first case's particular output value 42 doesn't matter, all that matters is that f converges.) The flowchart illustrates how f is constructed; it uses the above function `halt_decider` in its decision box.

[†] We use a distinct typeface for problem names, as in 'Halting'.

$$f(e) = \begin{cases} 42 & \text{– if } \phi_e(e)\uparrow \\ \uparrow & \text{– if } \phi_e(e)\downarrow \end{cases}$$



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. There are two possibilities and both lead to a contradiction. Since assuming that `halt_decider` is mechanically computable gives a contradiction, that function is not mechanically computable. □

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. We will next leverage that one to produce many jobs that cannot be done. So the Halting problem is part of a larger phenomenon of unsolvability.

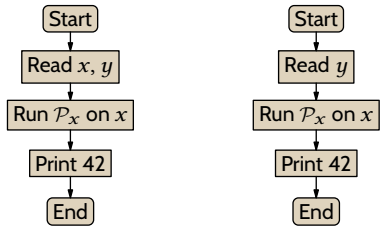
5.4 Example Consider this problem: we want an algorithm that tells us whether a given Turing machine halts on the input 3. That is: given e, does $\phi_e(3)\downarrow$?

We will show that if this

$$\texttt{halts\_on\_three\_decider}(e) = \begin{cases} 1 & \text{– if } \phi_e(3)\downarrow \\ 0 & \text{– otherwise} \end{cases}$$

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 an x and want to know whether $\phi_x(x)\downarrow$. Consider the machine outlined on the right below. It reads the input y and ignores it, and also gives a generic 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 it never gets through its middle box and so the machine as a whole does not halt.



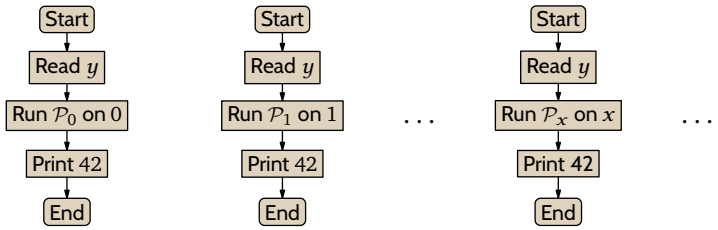
In particular, the machine on the right halts on input $y = 3$ if and only if $\mathcal{P}_x$ halts on x (it is also true that $\mathcal{P}_x$ halting on x implies the same for all other input y's but that is not relevant to our strategy). So with machine on the right, if we were able to answer questions about halting on 3 then we could leverage that ability to make ourselves able to determine whether $\mathcal{P}_x$ halts on x.

Now for the argument: consider this function.

$$\psi(x,y) = \begin{cases} 42 & \text{– if } \phi_x(x)\downarrow \\ \uparrow & \text{– otherwise} \end{cases}$$

It is computed by the machine described by the flowchart above on the left. 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-m-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 each of these machines the value used in its middle box is hard-coded into its 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 $\mathcal{P}_{s(e_0,x)}$ halts on input $y = 3$ if and only if $\mathcal{P}_x$ halts on input x.



Therefore, for all $x \in \mathbb{N}$ we have this.

$$\phi_x(x)\downarrow \quad \text{if and only if} \quad \texttt{halts\_on\_three\_decider}(\, s(e_0,x)\,) = 1 \qquad (*)$$

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.

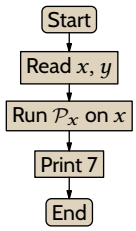
5.5 Example We will show that this problem is not mechanically solvable: given e, determine whether $\mathcal{P}_e$ outputs 7 for some input.

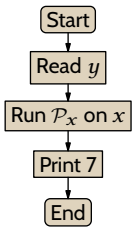
$$\texttt{outputs\_seven\_decider}(e) = \begin{cases} 1 & \text{– if } \phi_e(y) = 7 \text{ for some } y \\ 0 & \text{– otherwise} \end{cases}$$

The argument is much like the one in the prior example. 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 ψ is intuitively mechanically computable and by Church's Thesis 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, the functions computed by the machines $\mathcal{P}_{s(e_0,0)}$, $\mathcal{P}_{s(e_0,0)}$, ... On the right is the flowchart for the machine $\mathcal{P}_{s(e_0,x)}$. As in the prior example note that x is hard-coded into its source, so this is a single-input machine.

This machine $\mathcal{P}_{s(e_0,x)}$ has the property that $\mathcal{P}_x$ halts on x if and only if there exists an input y such that the machine outputs a 7. (Not only is it true if and only if there exist such an input, but in fact it is true if and only if this happens for every input. But that is not relevant to the argument.) Thus $\phi_x(x)\downarrow$ if and only if $\texttt{outputs\_seven\_decider}(s(e_0,x)) = 1$. If the function `outputs_seven_decider` were computable then because the composition of two computable functions, $\texttt{outputs\_seven\_decider} \circ s$, is computable, we would have that the Halting problem is computably solvable, which is not right. Therefore `outputs_seven_decider` is not computable.

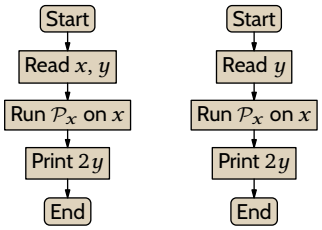
5.6 Example We next show that this problem is unsolvable: given e, decide whether ϕ_e is the doubler function, that is, whether $\phi_e(y) = 2y$ for all y.

We will show that this function is not computable.

$$\texttt{doubler\_decider}(e) = \begin{cases} 1 & \text{– if } \phi_e(y) = 2y \text{ for all } y \\ 0 & \text{– otherwise} \end{cases}$$

The function on the left below is intuitively computable by the flowchart in the middle.

$$\psi(x,y)=\begin{cases}2y & \text{– if } \phi_x(x)\downarrow\\ \uparrow & \text{– otherwise}\end{cases}$$



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-m-n* theorem to get a family of functions $\phi_{s(e_0,0)}$, $\phi_{s(e_0,1)}, \ldots$ A general member $\mathcal{P}_{s(e_0,x)}$ of this family is sketched by the flowchart on the right. It illustrates that $\phi_x(x)\downarrow$ if and only if `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 false.

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. We will close with a few points.

First, to reiterate, saying that a problem is unsolvable means that it is unsolvable by a mechanism, that no Turing machine computes the solution to the problem. There is a function that solves it, but that function is not effectively computable.

Second, the fact that the Halting problem is unsolvable does not mean that for all computations, we cannot tell if that computation halts. Here is a computation that obviously halts for every input.

```
> (define (successor i)
    (+ 1 i))
```

Nor does it mean that for all computations, we cannot tell whether it does not halt. This one, once started, just keeps going (below, control-C interrupted the run).

```
> (define (f x)
    (displayln x)
    (f (+ 1 x)))
> (f 0)
0
1
  ...
97806
97807
; user break [,bt for context]
```

[†] 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 function reduces to antidifferentiating that function." If we can antidifferentiate then we can find the area. Similarly here, if we can solve the Halts On Three problem then we can solve the Halting problem.

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. Consequently, 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 of the two applies is not what we have in mind when we think about solving the Halting problem. We want uniformity. We want a single effective procedure, one program, that inputs e and that outputs the right answer.

That same sentence 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 halt 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. Theorem 5.3 speaks to uniformity — specifically, it says that uniformity is impossible.

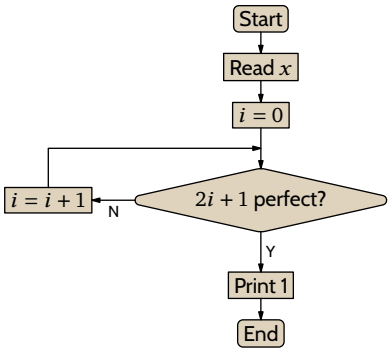
Our third point is about why unsolvability of the Halting problem is so important in the subject. A beginning programming class could leave the impression that if a program doesn't halt then it just has a bug, something fixable. So it could seem to a student in that course that the Halting problem is not deeply interesting.

That impression is wrong. Imagine that we could somehow write a utility `always_halts` 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 generic 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 90, 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 our fourth point is that there is another reason for our interest. With a computable `halt_decider` in hand, 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.[†]

[†] People have done computer checks up to 10^{1500} and not found any.



With a solution to the Halting Problem we could settle the question. The program sketched here 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. Also, 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* inputs a statement and then starts with the axioms and does a breadth-first enumeration of the theorems, looking for the given one. Turing and Church independently used an approach like the one in this section to show that the *Entscheidungsproblem* is unsolvable: if there were a computable way to answer all mathematical questions then we could use it to answer questions about Turing machines, including whether they halt.

Our final point in this discussion starts by noting 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 one of the instructions in $\mathcal{P}_e$ is $q_0\mathsf{BL}q_1$.

The difference between this problem and the ones earlier in this section is that this solvable one is about the machine's source, while the unsolvable ones that we have seen are about the machine's behavior. We seem to run into trouble when we try to mechanically analyze what a machine will do, rather than sticking to what it is.

This point brings us back to the opening of the first chapter where we said that most of our interest in the input-output behavior of the machines, in what they do, instead of in their internal construction.

II.5 Exercises

5.7 Someone 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?

5.8 Is this statement right or wrong: 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 $\phi_e(e)\uparrow$?

[†] This program takes an input x but ignores it; in this book we like to have the machines that we use take an input and (if ti halts) also give an output.

✓ 5.9 Which of these statements is a correct use of 'reduces to'?

(A) "The problem of getting bread reduces to the problem of finding the store" or "The problem of finding the store reduces to the problem of getting bread."

(B) "The problem of showing a number is nonprime reduces to the problem of exhibiting a one of its factors" or "The problem of exhibiting a factor of a number reduces to the problem of showing that the number is nonprime."

(C) "The problem of getting the most votes reduces to the problem of winning the election" or "The problem of winning the election reduces to the problem of getting the most votes."

5.10 Your study partner asks you, "The Turing machine $\mathcal{P} = \{q_0\mathsf{BB}q_0, q_0\mathsf{1}\mathsf{1}q_0\}$ 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? To not halt the machine must have an infinite loop, right? Any Turing machine has a finite number of states and the tape alphabet is finite, right? So there are only finitely many state-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." 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 For each of these, is it 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 the single number 1 if in 1924 G Mallory reached the summit of Everest, 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 e, 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 a total function, that is, if it converges on every input.

✓ 5.19 *See the instructions before Exercise 5.18.* Given an index e, decide if the Turing machine $\mathcal{P}_e$ squares its input. That is, decide if ϕ_e associates $y \mapsto y^2$.

5.20 *See the instructions before Exercise 5.18.* Given e, determine if the function ϕ_e halts and 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 before Exercise 5.18.* Given e, decide whether ϕ_e fails to converge on input 5.

5.22 *See the instructions before Exercise 5.18.* Given an index, determine if the computable function with that index fails to converge on all odd numbers.

5.23 *See the instructions before Exercise 5.18.* Given e, decide if the function ϕ_e has the action $x \mapsto x + 1$.

5.24 *See the instructions before Exercise 5.18.* Given e, decide if the function ϕ_e fails to converge on both inputs x and $2x$, for some x.

5.25 One of these problems is solvable and one is not. Which is which?
(A) Given an index e, decide if $\mathcal{P}_e$ halts on input 153.
(B) Given an index e, decide if $\mathcal{P}_e$ halts in sooner than 1000 steps on input 153.

5.26 Fix integers $a, b, c \in \mathbb{N}$ and consider the problem $\mathcal{L}_{a,b,c}$ of determining whether there is a single-number input $\text{cantor}(x, y)$ such that $ax + by = c$. Is this 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 $\mathcal{P}_4$ halts on input e. (D) Given e, decide if $\mathcal{P}_e$ contains an instruction with state q_e.

✓ 5.28 For each problem, fill in the blanks to show that the problem is unsolvable.

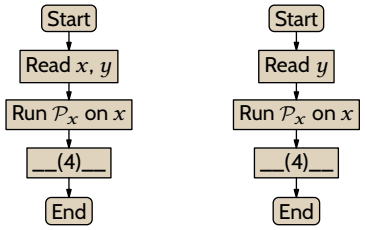
We will show that this is not mechanically computable.

$$\underline{\quad(1)\quad}\texttt{\_decider}(e) = \begin{cases} 1 & \text{– if } \underline{\quad(2)\quad} \\ 0 & \text{– otherwise} \end{cases}$$

For that, consider this function.

$$\psi(x,y) = \begin{cases} \underline{\quad(3)\quad} & \text{– if } \phi_x(x)\downarrow \\ 0 & \text{– otherwise} \end{cases}$$

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 $\phi_x(x)\downarrow$ if and only if `__(1)__decider`$(s(e_0,x)) = 1$. *Because the function s is mechanically computable, if* `__(1)__decider` *were mechanically computable then the* Halting *problem would be mechanically solvable. But the* Halting *problem is not mechanically solvable. Therefore* `__(1)__decider` *is not mechanically computable.*

(A) Given machine index e, decide if there is a $y \in \mathbb{N}$ so that $\mathcal{P}_e$ outputs y on input 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 $\phi_e(y) = y + 2$.

5.29 In some ways a more natural set than $K = \{e \in \mathbb{N} \mid \phi_e(e)\downarrow\}$ 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 also not computable.

5.30 The Halting problem of determining membership in the set $K = \{e \mid \phi_e(e)\downarrow\}$ appears to be an aggregate, or to cut across all Turing machines, in that for every Turing machine a piece of information about that machine forms part of K.

(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 $\mathcal{P}_e$ halts on y is decidable.

✓ 5.31 For each, if it is mechanically solvable then sketch a algorithm to solve it. If it is unsolvable then show that.

(A) Given $e \in \mathbb{N}$, 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.

5.32 Is K infinite?

5.33 True or false: the number of unsolvable problems is countably infinite.

5.34 Show that for any Turing machine, the problem of determining whether it halts on all inputs is solvable.

5.35 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.

5.36 Brocard's problem asks whether there are any numbers besides 4, 5, and 7 for which $n! + 1$ is a perfect square (computer searches up to a quadrillion, 1×10^{15}, have not found any other solutions). Show that if we could solve the Halting problem then we could in principle settle this problem.

5.37 Show that most problems are unsolvable by showing that there are uncountably many functions $f : \mathbb{N} \to \mathbb{N}$ 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 bitstrings 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.

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.[‡]

[‡] 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.

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 $\hat{e} \in \mathcal{I}$ also.

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 $\hat{e} \in \mathbb{N}$ are 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 $\phi_e(x) = 3x$ for all x or $\phi_e(x) = x^3$ for all x, and consequently $\hat{e} \in \mathcal{J}$.

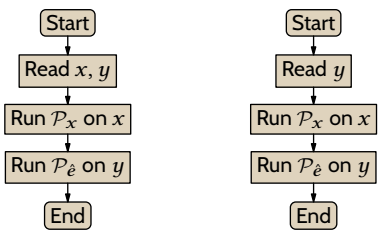
6.5 EXAMPLE The set $\{ e \in \mathbb{N} \mid \mathcal{P}_e \text{ contains an instruction starting with } q_{10} \}$ is not an index set. We can easily produce two Turing machines having the same behavior where one machine contains such an instruction while the other does not.

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 a nontrivial 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 6.36.

So assume $e \notin \mathcal{I}$. Since $\mathcal{I}$ is not empty it contains an index $\hat{e} \in \mathcal{I}$. Because $\mathcal{I}$ is an index set, $\phi_e \neq \phi_{\hat{e}}$. Thus there is a y such that $\phi_{\hat{e}}(y)\downarrow$.

Consider the flowchart on the left below. 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.

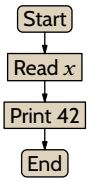


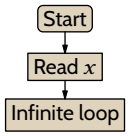
We've constructed the machine on the right so that if $\phi_x(x)\uparrow$ then $\phi_{s(e_0,x)} \simeq \phi_e$ and thus $s(e_0, x) \notin \mathcal{I}$. As well, if $\phi_x(x)\downarrow$ then $\phi_{s(e_0,x)} \simeq \phi_{\hat{e}}$, and thus $s(e_0, x) \in \mathcal{I}$. It follows that if $\mathcal{I}$ were mechanically computable, so that we could effectively check whether $s(e_0, x) \in \mathcal{I}$, then we could solve the Halting problem. □

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.

[†] It is called an index set because it is a set of indices.

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 $\hat{e} \in \mathcal{I}$. Therefore $\mathcal{I}$ is an index set.

The above example is the same problem as in the first example of the prior subsection. Rice's Theorem considerably simplifies the answer. (Of course, our development of the theorem relies 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. We will produce an appropriate $\mathcal{I}$ and verify that it is a nontrivial 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 Here is another unsolvable problem: given e, decide whether ϕ_e equals this.

$$f(x) = \begin{cases} 4 & \text{– if } x \text{ is prime} \\ x+1 & \text{– otherwise} \end{cases}$$

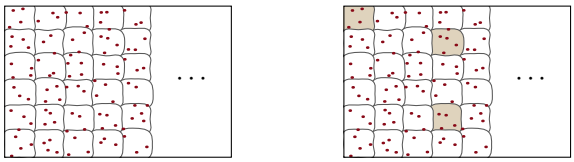
Let $\mathcal{I} = \{ j \in \mathbb{N} \mid \phi_j = f \}$. The set $\mathcal{I}$ is not empty because we can write a program with this behavior and so by Church's Thesis there is a Turing machine with this behavior, and its index is a member of $\mathcal{I}$. Also, $\mathcal{I} \neq \mathbb{N}$ because 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. Suppose that $e \in \mathcal{I}$ and also that $\hat{e}$ is such 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 $\phi_{\hat{e}}(x) = \phi_e(x)$ for all x, and so $\hat{e}$ is also a member of $\mathcal{I}$. Hence $\mathcal{I}$ is an index set.

We close by revisiting the theorem at a somewhat higher level. This chapter started by showing that unsolvable problems exist. However, that counting argument proof did not yield natural examples. In the prior section we got many examples of interesting and practical unsolvable problems, and these led to the intuition that we cannot mechanically analyze the behavior of machines. In this section we formally defined 'behavior'. With that, Rice's Theorem says that every interesting set of behaviors, 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 a casual reading might make it seem that every problem is unsolvable.

But that can't be right. We've all written programs that do real world tasks, with behaviors that we would at least informally characterize as interesting. The resolution of this dissonance is of course that Rice's Theorem says that no problem is solvable if it is of a certain kind. We now expand on what that kind is.

In the drawing on the left below, the red dots represent Turing machines. (Of course there are infinitely many machines, but it is just a sketch.) There are some pairs of machines that have the same input/output behavior, that is, they yield the same computable function.[†] So there are multiple machines whose input/output behavior is the doubling function $f(x) = 2x$, multiple machines that halt on no input, etc. The tiles group together machines with the same behavior, ones that give the same computable functions.



We say that a property of a machine is 'extensional' if whenever two machines compute the same function — whenever they are shown in the same tile — then either they both have that property or they both do not. Rice's Theorem asserts that an extensional property of machines that is effective must be trivial, meaning that it either holds of all machines or of none.

Here are some extensional properties of machines: (1) halting on input 3, (2) having an input for which the output is 7, (3) having an input on which the machine fails to halt. For example for (1), as it is an extensional property and nontrivial, Rice's Theorem says that it cannot be effective, that we cannot computably decide it.

Here are some non-extensional properties, ones to which Rice's Theorem does not apply: (1) halting within 100 steps on every input, (2) visiting fewer than 100 tape cells on input 0, and (3) containing the state q_{10}.

So if each tile is a behavior, what is an index set? As we saw earlier, for instance in Example 6.4, an index set is the union of the indices from a number of tiles; the

[†] In fact, the Padding Lemma on page 71 says that for every computable function there are infinitely many machines.

picture above on the right joins three of them. The property of being in this $\mathcal{I}$ is extensional because we are taking entire tiles.

In summary, although Rice's Theorem does not apply to all problems, nevertheless it is especially significant for understanding what can be done through mechanical analysis alone. Rice's Theorem is about those properties of machines that extend to be properties of the functions that those machines compute. For instance, the problem of deciding whether a program computes the squaring function is unsolvable, but the problem of deciding whether its source code contains the letter 'k' is not.

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 $\varnothing \subseteq \{ x \mid \phi_e(x)\downarrow \}$?

6.13 True or false: the given property of machines is extensional. Briefly justify. (A) The machine halts on input 5. (B) It has exactly four instructions. (C) It computes twice its input.

6.14 Briefly describe why these machine properties, listed in the section, are non-extensional. (A) The property of halting within 100 steps on every input, (B) of visiting fewer than 100 tape cells on input 0, and (C) of containing the state q_{10}.

6.15 Give a trivial index set: fill in the blanks $\mathcal{I} = \{ e \mid \_\_\_\_\_\mathcal{P}_e\_\_\_\_\_ \}$ so that the set $\mathcal{I}$ is empty.

6.16 Give a trivial index set: fill in the blanks $\mathcal{I} = \{ e \mid \_\_\_\_\_\mathcal{P}_e\_\_\_\_\_ \}$ so that the set $\mathcal{I}$ is all of $\mathbb{N}$.

6.17 For each problem, produce an index file suitable for applying Rice's Theorem. You needn't give the entire argument, just produce a file.

(A) Given e, determine if $\mathcal{P}_e$ halts on input 7 with output 7.
(B) Given e, determine if $\mathcal{P}_e$ halts on input e and returns output e.
(C) Given e, determine if $\mathcal{P}_{2e}$ returns output 7 for any input y.
(D) Given e, determine if $\mathcal{P}_e$ halts on 7 and gives a prime number.

For each of the problems from Exercise 6.18 to Exercise 6.24, show that it is unsolvable by applying Rice's theorem. (These repeat the problems from Exercise 5.18 to Exercise 5.24.)

✓ 6.18 *See the instructions above.* Given an index e, determine if ϕ_e is total, that is, if it converges on every input.

✓ 6.19 *See the instructions before Exercise 6.18.* Given an index e, decide if the Turing machine $\mathcal{P}_e$ squares its input. That is, decide if ϕ_e performs $y \mapsto y^2$.

6.20 *See the instructions before Exercise 6.18.* 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 $y \in \mathbb{N}$.

6.21 *See the instructions before Exercise 6.18.* Given an index e, determine whether ϕ_e fails to converge on input 5.

6.22 *See the instructions before Exercise 6.18.* Given an index, determine whether the Turing machine $\mathcal{P}_e$ fails to halt on all odd numbers.

6.23 *See the instructions before Exercise 6.18.* Given an index e, decide if the function ϕ_e computed by machine $\mathcal{P}_e$ performs $x \mapsto x+1$.

6.24 *See the instructions before Exercise 6.18.* Given an index e, decide if the function ϕ_e fails to converge on both inputs x and $2x$, for some x.

✓ 6.25 Show that each of these is an unsolvable problem by applying Rice's Theorem.

(A) The problem of determining whether a function is partial, that is, whether it fails to converge on some input.

(B) The problem of deciding whether a function ever converges, on any input.

✓ 6.26 For each problem, fill in the blanks to prove that it is unsolvable.

We will show that $\mathcal{I} = \{e \in \mathbb{N} \mid \underline{\quad (1) \quad}\}$ *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:* ___(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 other sketch:* ___(3)___ *is intuitively computable so by Church's Thesis there is such a Turing machine. Its index is not an element of* $\mathcal{I}$.

Finally, 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* $e \in \mathcal{I}$, ___(4)___. *Because* $\phi_e \simeq \phi_{\hat{e}}$ *we have that* ___(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.

6.27 Prove that any set that is not computable is infinite.

6.28 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`.

6.29 As in the prior exercise, a Turing machine accepts a set of bit strings $\mathcal{L} \subseteq \mathbb{B}^*$ if it inputs bit strings, halts on all inputs, and it outputs 1 if input is a member of $\mathcal{L}$, and 0 otherwise. Show that this problem is unsolvable: given e, determine if $\mathcal{P}_e$ accepts $\mathbb{B}^*$ itself. Show that this problem is mechanically unsolvable: give e, determine whether there is an input x so that $\phi_e(x)\!\downarrow$.

6.30 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 $\mathcal{I} = \{e \mid \mathcal{P}_e \text{ has an unreachable state}\}$ is not an index set.

6.31 Your classmate says, "The section ends by saying that Rice's Theorem is about those properties of machines that extend to be properties of the functions that those machines compute. But here is a problem that is about the properties 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 halt or it goes on." Where are they mistaken?

6.32 Show that no finite set that is nonempty is an index set.

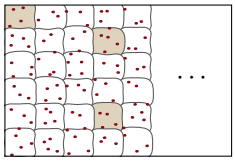
6.33 Show that each of these is an index set.

(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) $\{e \in \mathbb{N} \mid \text{the function } \phi_e \text{ is either total or else } \phi_e(3)\!\uparrow\}$

6.34 In the section we characterized of index sets as in the picture below. We start with the set of all integers, which is the rectangular box, and group them together when they are indices of equal computable functions. Then to get an index set, select a few parts such as the three shown, and take their union.



Here we justify that picture. (A) Consider the relation $\simeq$ between natural numbers given by $e \simeq \hat{e}$ if $\phi_e \simeq \phi_{\hat{e}}$. 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.

6.35 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.

6.36 Do the $e_0 \in \mathcal{I}$ case in the proof of Rice's Theorem, Theorem 6.6.

Section II.7 Computably enumerable sets

The natural way to attack the Halting problem is to start by simulating $\mathcal{P}_0$ on input 0 for one step. Next, 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, and $\mathcal{P}_1$ on 1 for a second step, and then $\mathcal{P}_2$ on 2 for one step. In this way, cycle among the $\mathcal{P}_e$ on e simulations, running each for a step.[†] Eventually some of these halt and

[†] That is, run a loop that at iteration i runs the s-th step of simulating $\mathcal{P}_e$ on input e, where $i = e + s$.

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 $f(0), f(1), \ldots$ gives the elements of K.

Why won't this process solve the Halting problem? If $e \in K$ then dovetailing will eventually find that out. But if $e \notin K$ then it will never reveal the non-membership.

Recall that a set of numbers is computable if its characteristic function is computable. We are seeing another way to describe a set, by listing its members. Definition 1.13 gives the terminology: a function f with domain $\mathbb{N}$ 'enumerates' its range.

7.1 Definition A set of natural numbers is computably enumerable (or c.e.) if it is effectively listable, that is, if it is the range of a computable function. (That function may be partial.) Alternate terms are: recursively enumerable (or r.e.), or semicomputable, or semidecidable. The set of computably enumerable sets is denoted *RE*.

Picture the stream $\phi(0), \phi(1), \phi(2), \ldots$ gradually filling out the set. (It may be that the stream contains repeats or that the numbers may appear in some willy-nilly order, not necessarily ascending, or that the $\phi(i)$'s diverge from some point on.)

7.2 Remark Here is a particularly interesting stream. 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 avoid any ambiguity of leading 0's in the binary). 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. It might first combine axiom 0 with axiom 1, and then combine axiom 0 with axiom 2, etc. In this way it generates a list of all of this theory'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 are interested in a statement from this topic, such as Goldbach's conjecture that every even number is the sum of at most two primes. We could watch the process as it enumerates the theorems. If the statement is provable then its number will eventually appear.

7.3 Lemma The following are equivalent for a set of natural numbers.
(A) It is computably enumerable, that is, it is the range of a computable function.
(B) It is the range of a total computable function, or it is empty.
(C) It is the domain of a computable function.

Proof We will show that the first and second are equivalent. That the second and third are equivalent is Exercise 7.32.

The (B) implies (A) direction is easy. Where the set S is the range of a total computable function, it is the range of a computable function. If S is empty then it is the range of the computable function that never converges.

Now for (A) implies (B). Assume that S is computably enumerable so that it is the range of a computable function ϕ_e (which may be non-total). If ϕ_e diverges for all inputs then $S = \varnothing$, which is one of the cases in (B).

The other case is where ϕ_e does not diverge for all inputs and we will produce a total computable f whose range is S. In this case there is an input y where $\phi_e(y)\!\downarrow$. Let $f(0) = \phi_e(y)$. As to the other values of the function: given $n \in \mathbb{N}$, run the computations of $\mathcal{P}_e$ on inputs 0, 1, ... n, each for n-many steps. Possibly some of these halt. Let $f(n)$ be the least k where $\mathcal{P}_e$ halts on some input i within n steps and outputs k, and also where $k \notin \{f(0), f(1), \ldots f(n-1)\}$. If no such k exists then let $f(n) = s_0$.

By the prior paragraph's final sentence, f is total. We must verify that the range of f is S. For $t \in \mathbb{N}$, if $t \notin S$ then $\mathcal{P}_e$ never outputs it and so t is never defined as $f(n)$. If $t \in S$ then there must be an n large enough that $\mathcal{P}_e$ halts on some input $i \leq n$ within n steps and outputs t. The number t is then queued for output by f in the sense that it will be enumerated as, at most, $f(n+t)$. □

Thus, the collection of effectively listable sets is the same as the collection of domains of computable functions. There is a standard notation for the latter.

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 its 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 it might not ever discover 'no'. 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 For (A) let $S \subseteq \mathbb{N}$ be computable so that its characteristic function is a computable function ϕ. We will enumerate the elements of S. Begin by using ϕ to test whether $0 \in S$, that is, whether $\phi(0) = 1$. Then test whether $1 \in S$, etc. If this sequence of tests ever finds a k_0 so that $\phi(k_0) = 1$, then set $f(0) = k_0$. After that, iterate: find the next element of S by testing whether $k_0 + 1 \in S$, $k_0 + 2 \in S$, ... and if this testing sequence ever halts with a k_1 then set $f(1) = k_1$. Clearly f is a computable function whose range is S.

As to (B), suppose first that S is computable. The complement S^{c} is also computable because $\mathbb{1}_{S^{\mathsf{c}}}(x) = 1 - \mathbb{1}_S(x)$ for all x. With that, item (A) gives that both S and S^{c} are computably enumerable.

For the converse 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 $\hat{g}$. To show that S is computable we will give an effective procedure that acts as its characteristic

procedure, that is, so that given $x \in \mathbb{N}$, the procedure determines whether or not $x \in S$. The idea is to 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. Continuing in this way, eventually we will find that x has been enumerated into one of S or S^{c}. □

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. So K is computably enumerable. If the complement K^{c} were also computably enumerable then Lemma 7.5 would imply that K is computable, which it isn't. □

That result gives one reason to be interested in computably enumerable sets, namely that the Halting problem set K falls into that class of sets, as do such sets as $\{e \mid \phi_e(3)\!\downarrow\}$ and $\{e \mid$ there is an x 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 in mind we can perceive that in a sense computable sets are the only ones that we will ever fully know. Consonant with that perception, computably enumerable sets are exactly those about which we have at least partial information: we can tell what is in but we cannot necessarily tell what is out.

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 Your study partner asks the group, "One computably enumerable set is the empty set. But the empty set is not effectively listable, because you can't list nothing." Where are they mis-thinking?

✓ 7.9 For each set, produce a function that enumerates it (A) $\mathbb{N}$ (B) the even numbers (C) the perfect squares (D) the set $\{5, 7, 11\}$.

7.10 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).

7.11 Are there computably enumerable sets that are infinite? Finite? Empty? All of the natural numbers?

7.12 One of these two is computable and the other is computably enumerable but not computable. Which is which?

(A) $\{e \mid \mathcal{P}_e$ halts on input 4 in less than twenty steps$\}$

(B) $\{e \mid \mathcal{P}_e$ halts on input 4 in more than twenty steps$\}$

7.13 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.

7.14 (IIS, IIT 2022) One of these statements is true. Which? (A) Every proper subset of a computably enumerable set is computable. (B) If a set and its complement are both computably enumerable then both are computable.

✓ 7.15 Someone online says, "every countable set S is computably enumerable because if $f\colon \mathbb{N} \to \mathbb{N}$ has range S then you have the enumeration S as $f(0)$, $f(1)$, ..." Explain why this is wrong.

✓ 7.16 The set $A_5 = \{ e \mid \phi_e(5)\!\downarrow \}$ is not computable. Show that it is computably enumerable.

7.17 Show that the collection of computably enumerable sets is countable.

7.18 Every uncomputable set is infinite, since every finite set is computable. Is every computably enumerable set infinite?

7.19 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 starting with state q_4. (B) The set of indices of Turing machines that halt on input 4. (C) The set of indices of Turing machines that halt on input 4 in fewer than 100 steps.

7.20 Show that the set $\{ e \mid \phi_e(2) = 4 \}$ is computably enumerable.

7.21 Name three sets that are computably enumerable but not computable.

✓ 7.22 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.

7.23 We know that there are subsets of $\mathbb{N}$ that are not computable. Do the computably enumerable sets make up the subsets that are not computable?

✓ 7.24 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.

7.25 Prove that the set $\{ e \mid \phi_e(3)\!\uparrow \}$ is not computably enumerable.

✓ 7.26 Prove that every infinite computably enumerable set has an infinite computable subset.

7.27 Define the function steps by: steps(e) is the minimal number of steps so that Turing machine $\mathcal{P}_e$ halts if started with input e on its 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 computable extension, no total computable $f\colon \mathbb{N} \to \mathbb{N}$ so that if steps$(e)\!\downarrow$ then steps$(e) = f(e)$.

7.28 A set is computable enumerable in increasing order if there is a computable f that is increasing, so that $n < m$ implies $f(n) < f(m)$, and whose range is the

set. Assume that the set S is infinite. Prove that S is computable if and only if it is computably enumerable in increasing order.

7.29 A set is co-computably enumerable if its complement is computably enumerable. Produce a set that is neither computably enumerable nor co-computably enumerable.

7.30 *(Compare this with 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?

7.31 *(Compare this with 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?

7.32 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 given machine halts is so hard that it is unsolvable. Is this the absolutely hardest problem or are there ones that are even harder?

Priestess of Delphi (Collier 1891)

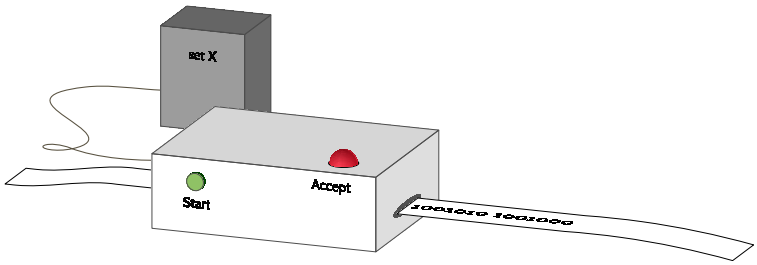
To answer we must define what it means to say that one problem is harder than another. Recall that we have already compared problem hardness, for instance when we considered the problem of whether a Turing machine halts on input 3. There we proved that if we could solve the halts-on-3 problem then we 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 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 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 we could computed if we somehow were able to solve the Halting problem, we attach

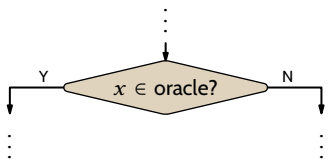
[†] We can instead conceptualize that the first problem is at least as general as the second. An example is that the problem of inputting a natural number and outputting its prime factors is at least as general as the problem of inputting a natural and determining whether it is divisible by seven. [‡] 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."

a K-oracle that answers questions of the form "Is $x \in K$?" This oracle is a black box, meaning that we can't open it to see how it works.



We could formally define computation relative to an oracle $X \subseteq \mathbb{N}$ by extending the Turing machine definition. However, the low-level details are less helpful in this section so we will instead describe it conceptually. Imagine starting with a programming language and adding a Boolean function `oracle ?`, or allowing flowcharts to have queries.[†]

```
(if (oracle? x)
    (displayln "It is in the set")
    (displayln "It is not in the set"))
```



We can change the oracle without changing the program code: in the picture if we swap out black boxes, exchanging an X oracle for a Y oracle, then the white CPU box is unchanged. Of course, the values returned by the oracle queries may change, which may change the tape output when we run the two-box system. But such a swap leaves the white hardware unchanged.

The rest of what we have already developed about Turing machines carries over. In particular, each such machine — each CPU box — 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 such a computation, we must specify which machine we are using along with which oracle. That explains the notation for the the oracle Turing machine, $\mathcal{P}_e^X$ and for the associated functions, ϕ_e^X.

8.1 Definition Let X be a set. If the characteristic function of a set S can be computed from X, that is, if $\mathbb{1}_S = \phi_e^X$ for some e, then we say that S is computable from the oracle X, or is X-computable, or that S reduces to X, or is Turing reducible to X, denoted $S \leq_T X$.[‡]

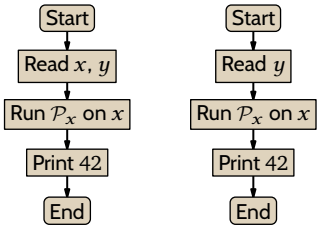
[†] A particular computation relative to an oracle might use one such query, or more than one, or none at all. [‡] We have already discussed, in the Halting problem section on page 94, that a common mistake is to read $S \leq_T X$ as 'X reduces to S' when what's right is 'S reduces to X'.

The intuition is that X "knows" at least as much as S, or has at least as much computational content as S, in that answers to questions about X suffice to give answers to questions about S.

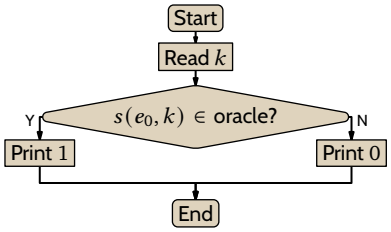
8.2 Example Recall the problem of determining, given e, whether $\mathcal{P}_e$ halts on input 3. That problem asks for a machine that acts as the characteristic function of the set $A = \{\, e \mid \mathcal{P}_e \text{ halts on } 3 \,\}$. We will show that $K \leq_T A$.

To produce an oracle machine $\mathcal{P}_e^X$ we first reprise Example 5.4. Start with the function $\psi\colon \mathbb{N}^2 \to \mathbb{N}$ on the left below. The flowchart in the middle along with 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-m-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 $k \in \mathbb{N}$.

$$\psi(x,y) = \begin{cases} 42 & \text{– if } \phi_x(x)\!\downarrow \\ \uparrow & \text{– otherwise} \end{cases}$$



With that we can build the oracle machine. The machine charted below is $\mathcal{P}_e^X$. It uses e_0 from the prior paragraph. If we connect it to an A oracle then it computes the characteristic function of K, by the prior paragraph's final sentence.



8.3 Example Let $B = \{\, e \mid \phi(y) = \phi(y+1) \text{ for some } y \,\}$. We will show that K reduces to B, that $K \leq_T B$.

We can reuse the function $\psi(x,y)$ from the prior example. As there, ψ is intuitively computable so Church's Thesis says that there is a Turing machine whose behavior is ψ. Let that machine have index e_0. Again apply the *s-m-n* Theorem to parametrize x, giving the family $\phi_{s(e_0,x)}$. Observe that $k \in K$ if and only if $s(e_0, k) \in B$. With that, the same oracle machine shows that $K \leq_T B$.

Turing equivalence As the less-than-or-equal symbol $\leq_T$ suggests, we have a kind of ordering, where some sets precede others.

8.4 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. Given a number x, we must decide whether $x \in A$ by using the 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 $\mathcal{P}_{\hat{e}}^C$ computes the characteristic function of B. Then compute the characteristic function of A directly from C by starting with the computation of A from B and replacing the B-oracle calls with calls to $\mathcal{P}_{\hat{e}}^C$. □

We next show that there are sets that come at the very beginning of this ordering.

8.5 Lemma If a set $Y \subseteq \mathbb{N}$ is computable then it is reducible to any set at all, $Y \leq_T X$ for any $X \subseteq \mathbb{N}$. Further, Y is computable if and only if it is reducible to any computable set, $Y \leq_T S$ where S is computable. In particular, a set is computable if and only if it is reducible to $\emptyset$.

Proof Assume that Y is computable, meaning that its characteristic function is computable. Then that characteristic function can be computed relative to an X oracle simply by never asking the oracle any questions.

As to the 'further', the prior paragraph proves that if Y is computable then it is reducible to a computable S because it is reducible to any S at all. For the other direction, suppose that the characteristic function of Y can be computed by reference to a computable oracle, so that $\mathbb{1}_Y = \phi_e^S$ where $\mathbb{1}_S$ is computable. Then replacing oracle calls in the machine $\mathcal{P}_e^S$ with direct computations of $\mathbb{1}_S$ will compute $\mathbb{1}_Y$ without reference to an oracle. Thus Y is computable. □

8.6 Definition Two sets A, B are Turing equivalent, $A \equiv_T B$, if $A \leq_T B$ and $B \leq_T A$.

8.7 Example Lemma 8.5 shows that any two computable sets are Turing equivalent.

Proving that two sets are T-equivalent, that they are inter-computable, shows that there is an underlying unity between the two. In a way, they are the same problem.

8.8 Example Example 8.2 proves that $K \leq_T A$, where $A = \{e \mid \mathcal{P}_e \text{ halts on } 3\}$. Exercise 8.27 proves that $A \leq_T K$. Thus $A \equiv_T K$.

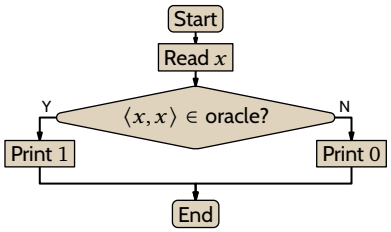
Of course, the Halting problem asks whether $\mathcal{P}_e$ halts on input e. A person may perceive that a more natural problem is deciding whether $\mathcal{P}_e$ halts on input x.

8.9 Definition $K_0 = \{\langle e, x\rangle \mid \mathcal{P}_e \text{ halts on input } x\}$

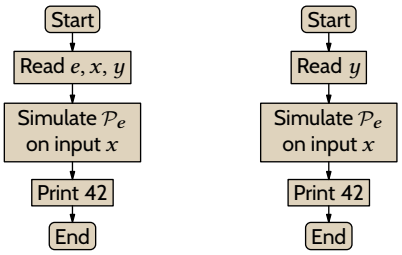
8.10 Theorem The sets K and K_0 are Turing equivalent, $K \equiv_T K_0$.[†]

Proof For $K \leq_T K_0$, suppose that we have access to a K_0-oracle. Then connecting the macine below to that oracle will give it the input/output behavior that is the characteristic function of K.

[†] Thus the choice of K as our touchstone is just a matter of convenience and convention. We use it because it has some technical advantages, including that it falls out of the diagonalization development that we did at the start of the Halting problem section, and because it is the standard in the literature.

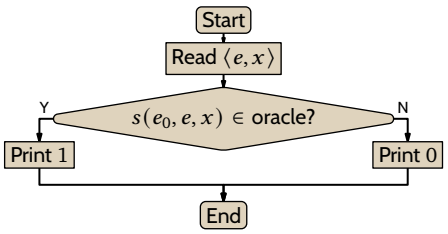


Next, the $K_0 \leq_T K$ half. Consider the flowchart on the left below. It halts for the input triple $\langle e, x, y\rangle$ if and only if $\langle e, x\rangle \in K_0$. By Church's Thesis there is a Turing machine implementing it; let that machine have index e_0.



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 for the oracle Turing machine. For a pair $\langle e, x\rangle$, the right-side machine above behaves the same on all inputs y. It either halts on all inputs 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 $y = s(e_0, e, x)$, so that $s(e_0, e, x) \in K$, if and only if $\phi_e(x)\!\downarrow$.



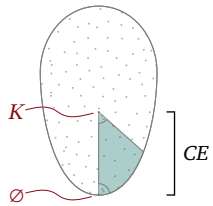
Thus given oracle K, the machine above acts as the characteristic function of K_0. □

8.11 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 The computably enumerable sets are the columns of K_0 since $W_e = \{y \mid \phi_e(y)\!\downarrow\} = \{y \mid \langle e, y\rangle \in K_0\}$. So computing W_e from a K_0 oracle is easy, and $W_e \leq_T K_0 \equiv_T K$. □

Because every computably enumerable set is Turing-computable from K, we say that K is complete among the computably enumerable sets.

Jumping The $\leq_T$ ordering ranks sets by how hard they are to compute. This illustrates. The dots are sets $S \subseteq \mathbb{N}$. If $S \leq_T X$ then S is drawn lower than X. The computable sets are at the bottom, grouped in with the empty set. The sets that are Turing equivalent to K are grouped together at the top of the computably enumerable sets.



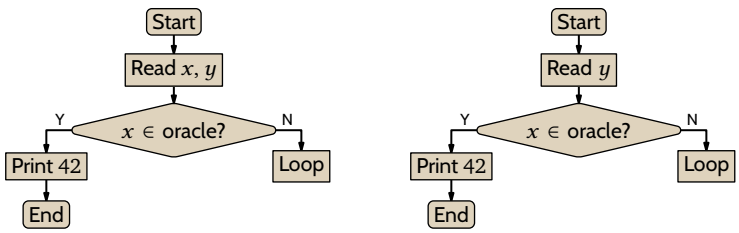
8.12 FIGURE: Subsets of $\mathbb{N}$ ranked by $\leq_T$.

We finish this section by describing a way to start with a set S, and then jump further up the order. Our prototype is to start with $\emptyset$ and jump to K.

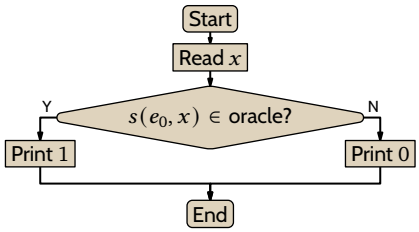
8.13 DEFINITION For any set S, the Relativized Halting problem for S is to determine membership in $K^S = \{x \mid \phi_x^S(x)\!\downarrow\}$.

8.14 THEOREM Every 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 says that it has an index, making it $\mathcal{P}_{e_0}^X$ for some e_0. Use the *s-m-n* theorem to parametrize x, giving the uniformly computable family of machines $\mathcal{P}_{s(e_0,x)}^X$ charted on the right.



Taking the oracle to be S and y to be $s(e_0, x)$ gives that $x \in S$ if and only if $\phi_{s(e_0,x)}^S(s(e_0,x))\!\downarrow$, which holds if and only if $s(e_0,x) \in K^S$. So if K^S is the oracle for the following machine



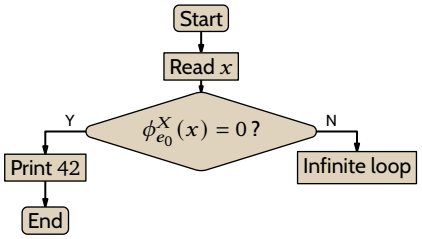
then that machine acts as the characteristic function of S. □

8.15 **Theorem** For no set S is K^S reducible to S. That is, $K^S \not\leq_T S$.

Proof Assume otherwise, that there is a computation relative to the S oracle that acts as the characteristic function of K^S.[†] Let that machine have index e_0.

$$\mathcal{P}^S_{e_0}(x) = \mathbb{1}_{K^S}(x) = \begin{cases} 1 & \text{– if } \phi^S_x(x)\downarrow \\ 0 & \text{– otherwise} \end{cases} \qquad (*)$$

Consider the oracle machine below. (Note that the decision box uses the machine with index e_0.) By Church's Thesis this machine has an index; let it be e_1.



Plug an S oracle into that machine and consider what happens when $x = e_1$.

By $(*)$ the function $\phi^S_{e_0}(x)$ converges for all x. If in the decision box $\phi^S_{e_0}(e_1) = 0$ then this machine halts, which is a contradiction because by $(*)$ the computation $\mathcal{P}^S_{e_0}(e_1)$ outputs 0 only when $\phi^S_{e_1}(e_1)$ diverges. If instead $\phi^S_{e_0}(e_1) = 1$ then this machine does not halt, also a contradiction because $(*)$ says that the computation $\mathcal{P}^S_{e_0}(e_1)$ outputs 1 only when $\phi^S_{e_1}(e_1)$ converges. Either way, assuming $(*)$ yields an impossibility. □

8.16 **Corollary** In particular, $K \leq_T K^K$ but $K^K \not\leq_T K$

Proof This is immediate from from the prior two results. □

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 that a Turing machine is a decider for a set if it computes the characteristic function of that set.

8.17 How to answer to your friend? "As the oracle you can use undecidable problems, such as the Halting problem. But isn't assuming the existence of a machine which can decide the Halting problem . . . problematic?"

8.18 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?

[†] This adaptats the proof from page 90 of the unsolvability of the Halting problem to computations relative to an oracle. Such an adaptation is a relativization.

✓ 8.19 Your friend says the professor, "Oracle machines are not real so why talk about them?" What should the professor say?

8.20 Your classmate says they answered a quiz question to define an oracle with, "A set to solve unsolvable problems." Give them a gentle critique.

✓ 8.21 Is there an oracle for every problem? For every problem is there an oracle?

8.22 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.

8.23 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 then there are only finitely many of them. But a finite set is computable and so by Lemma 8.5 it is reducible to any set." Give them a hand here.

8.24 To the drawing in Figure 8.12 add K^K.

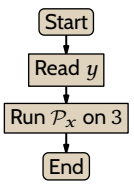
✓ 8.25 Suppose that the set A is Turing-reducible to the set B. Which of these are true? (A) A decider for A can be used to decide B. (B) If A is computable then B is computable also. (C) If A is uncomputable then B is uncomputable too.

8.26 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 $2B \leq_T B$.

✓ 8.27 Where $A = \{x \mid \mathcal{P}_x \text{ halts on } 3\}$, show that $A \leq_T K$. *Hint:* this machine $\mathcal{P}_e$ satisfies that $\phi_e(e)\!\downarrow$ if and only if $\phi_x(3)\!\downarrow$.



✓ 8.28 The set $S = \{e \mid \phi_e(3)\!\downarrow \text{ and } \phi_e(4)\!\downarrow\}$ is not computable. Outline an oracle machine showing that $S \leq_T K$. *Hint:* mimic Example 8.2.

✓ 8.29 For the set $S = \{e \mid \phi_e(3)\!\downarrow\}$, show that $S \leq_T K_0$.

✓ 8.30 Show that $K \leq_T \{e \mid \phi_e(y) = 2y \text{ for all input } y\}$.

8.31 Consider the set $\{e \mid \phi_e(j) = 7 \text{ for some } j\}$.

(A) Show that it is not computable using Rice's theorem.

(B) Sketch how to compute it using a K oracle.

8.32 Let $S = \{e \in \mathbb{N} \mid \phi_e(3)\!\downarrow \text{ and } \phi_{2e}(3)\!\downarrow \text{ and } \phi_e(3) = \phi_{2e}(3)\}$. Show $S \leq_T K$.

8.33 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 $K \leq_T$ Tot.

8.34 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 steps(x) is the smallest step number s where $\mathcal{P}_x$ halts on input x by step s, and steps$(x)\!\uparrow$ otherwise.

(B) Prove that $K \leq_T$ Ext.

8.35 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 $\phi^B(x)$ gives the same result as $\phi^A(x)$.

✓ 8.36 Prove that $A \leq_T A^{\mathsf{c}}$ for all $A \subseteq \mathbb{N}$.

8.37 Show that $K \not\leq_T \emptyset$.

8.38 Is the number of oracles countable or uncountable?

8.39 Fix an oracle. How many sets are computable from that oracle?.

8.40 Let A and B be sets. Produce a set C so that $A \leq_T C$ and $B \leq_T C$.

8.41 The relation $\leq_T$ is between 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 $A \leq_T A^{\mathsf{c}}$?

8.42 Assume $A \leq_T B$. Decide whether each is True or False, and sketch the argument. (A) $A^{\mathsf{c}} \leq_T B$ (B) $A \leq_T B^{\mathsf{c}}$ (C) $A^{\mathsf{c}} \leq_T B^{\mathsf{c}}$

8.43 Let $A \subseteq \mathbb{N}$. (A) Produce a definition of 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 74. We assumed that there is an onto function $f : \mathbb{N} \to \mathbb{R}$ and considered its inputs and outputs, as illustrated in this table.

n	$f(n)$'s decimal expansion
0	42.3 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 ...
⋮	⋮

We went down the diagonal to the right of the decimal point to get the sequence of digits $d_{0,0}, d_{1,1}, d_{2,2}, \ldots$, which in the illustration above is 3, 1, 4, 5, ... Using that, we constructed a number $z = 0.z_0 z_1 z_2 \ldots$ by making its n-th decimal place z_n be something other than $d_{n,n}$. Specifically, we took the 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 $z = 0.1211\ldots$ Then our argument culminated in verifying that z is not any of the rows. We say that we got z by 'diagonalizing out' of the list of 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 observation to sequences of computable functions, $\phi_{i_0}, \phi_{i_1}, \phi_{i_2}\ldots$ We are interested in effectiveness so we take the indices $i_0, i_1, i_2\ldots$ 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 computable sequence of computable functions has this form.

$$\phi_{\phi_e(0)},\ \phi_{\phi_e(1)},\ \phi_{\phi_e(2)}\ \cdots$$

Below is a table with all such sequences.

		Sequence term				
		$n=0$	$n=1$	$n=2$	$n=3$	$\ldots$
Sequence	$e=0$	$\phi_{\phi_0(0)}$	$\phi_{\phi_0(1)}$	$\phi_{\phi_0(2)}$	$\phi_{\phi_0(3)}$	$\cdots$
	$e=1$	$\phi_{\phi_1(0)}$	$\phi_{\phi_1(1)}$	$\phi_{\phi_1(2)}$	$\phi_{\phi_1(3)}$	$\cdots$
	$e=2$	$\phi_{\phi_2(0)}$	$\phi_{\phi_2(1)}$	$\phi_{\phi_2(2)}$	$\phi_{\phi_2(3)}$	$\cdots$
	$e=3$	$\phi_{\phi_3(0)}$	$\phi_{\phi_3(1)}$	$\phi_{\phi_3(2)}$	$\phi_{\phi_3(3)}$	
	$\vdots$	$\vdots$	$\vdots$	$\vdots$		

(*)

Each entry $\phi_{\phi_e(n)}$ is a computable function. If the index $\phi_e(n)$ diverges then the function as whole diverges.

As to the transformation, the natural one is this, for the computable function f.

$$\phi_x \overset{t_f}{\longmapsto} \phi_{f(x)} \qquad \text{so that} \qquad \phi_{\phi_i(j)} \overset{t_f}{\longmapsto} \phi_{f(\phi_i(j))}$$

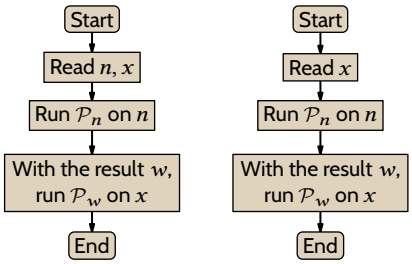
We next show that under this transformation, diagonalization fails. So t_f has a fixed point.

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 machine have index e_0. Apply the *s-m-n* theorem to parametrize n, giving the right chart, which describes a family of machines. The n-th member of that family,

[†] This is also known as the Recursion Theorem.

$\phi_{s(e_0,n)}$, computes the n-th function on the diagonal of the array $(*)$ above, $\phi_{\phi_n(n)}$.

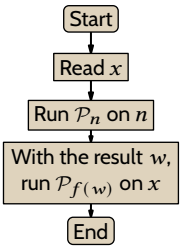


$$\phi_{s(e_0,n)}(x) = \begin{cases} \phi_{\phi_n(n)}(x) & \text{– if } \phi_n(n)\downarrow \\ \uparrow & \text{– otherwise} \end{cases}$$

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 functions on the diagonal are $\phi_{\phi_0(0)} = \phi_{g(0)}$, $\phi_{\phi_1(1)} = \phi_{g(1)}, \ldots$ The function g is computable and total because s is computable and total.

Under t_f those functions are transformed to $\phi_{fg(0)}$, $\phi_{fg(1)}$, $\phi_{fg(2)}$, ... The composition $f \circ g$ is computable and total since f is specified as computable and total in the theorem statement.

$$\phi_{fg(n)}(x) = \begin{cases} \phi_{f\phi_n(n)}(x) & \text{– if } \phi_n(n)\downarrow \\ \uparrow & \text{– otherwise} \end{cases}$$



As the flowchart underlines, $\phi_{fg(0)}$, $\phi_{fg(1)}$, $\phi_{fg(2)}$, ... is a computable sequence of computable functions. Hence it is one of table $(*)$'s rows. Let it be row v, making $\phi_{fg(m)} = \phi_{\phi_v(m)}$ for all m. Consider where the diagonal sequence $\phi_{\phi_n(n)} = \phi_{g(n)}$ intersects that row: $\phi_{g(v)} = \phi_{\phi_v(v)} = \phi_{fg(v)}$. The fixed point for f is $k = g(v)$. ∎

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 built-in defense against diagonalization.

The Fixed Point Theorem applies to any total computable function. It consequently it leads to many results, many of them surprising.

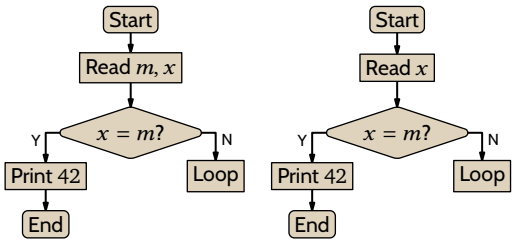
9.2 COROLLARY There is an index e so that $\phi_e = \phi_{e+1}$.

Proof The function $f(x) = x + 1$ is computable and total. So there is an $e \in \mathbb{N}$ such that $\phi_e = \phi_{f(e)}$. ∎

9.3 COROLLARY There is an index e such that $\mathcal{P}_e$ halts only on e.

Proof Consider the machine 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, $\mathcal{P}_{s(e_0,m)}$.



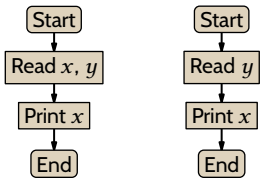
$$\phi_{s(e_o,m)}(x) = \begin{cases} 42 & \text{– if } x = m \\ \uparrow & \text{– otherwise} \end{cases}$$

Since e_0 is fixed, $s(e_0, x)$ is a total computable function of one variable, $f(m) = s(e_0, m)$, where the associated Turing machine halts only on input m. The function f has a fixed point, $\phi_{f(e)} = \phi_e$, and the associated Turing machine $\mathcal{P}_e$ halts only on e. □

We have previously informally referred to a machine's index as its "name." Rephrased for rhetorical effect, that result says the machine's name is how it behaves.

9.4 COROLLARY There is an $m \in \mathbb{N}$ such that $\phi_m(x) = m$ for all 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-m-n* theorem to get a uniformly computable family of functions parametrized by x, given by $\phi_{s(e_0,x)}(y) = x$. Let $f\colon \mathbb{N} \to \mathbb{N}$ be $f(x) = s(e_0, x)$. Because e_0 is fixed this is a total computable function of one input. Therefore there is a fixed point $m \in \mathbb{N}$ with $\phi_m(y) = \phi_{f(m)}(y) = \phi_{s(e_0,m)}(y) = m$ for all y. □

Here also, this machine's behavior is the same as its index. Imagine that we looked at the source of $\mathcal{P}_7$ and realized that it outputs 7 on all inputs. We might think that this is a crazy accident of our choice of numbering scheme for Turing machines. But the corollary says that any acceptable numbering must for some machine have this coincidence between index and behavior.

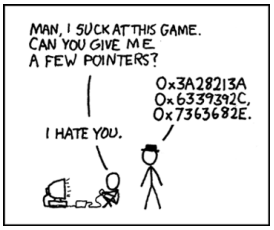
Further, since a Turing machine's index is source-equivalent, this result suggests that there is a program that outputs its own source, that self-reproduces. This is a form of self-reference; for more see Extra C.

Discussion The Fixed Point Theorem and its proof are often considered mysterious, or at any rate obscure. Here we will develop a point about the role of naming in the result.

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 points to something, 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, that we are using the word on two different levels.

A version of this happens in computer programming. See the C language code fragmant below. There, `x` and `y` are variables. If these were ordinary variables then the compiler would associate them with a particular memory cell. For instance, if an ordinary variable `a` were associated with cell 122 then the statement `a = 5` would result in the value 5 being stored in that cell. This is a name for the cell.

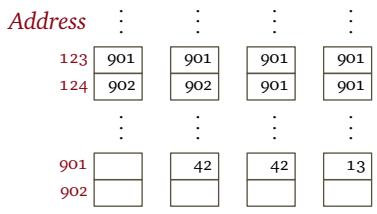
But the second line's asterisk means that `x` and `y` are not ordinary variables, they are pointers, which are associated with a cell but have some additional implications. The four vertical arrays illustrate by showing a machine's memory cells over time. They imagine that the compiler associates `x` with register 123 and `y` with 124. The first array shows that cell 123 happens to holds the number 901 and cell 124 holds 902.



Courtesy `xkcd.com`

Because these are pointers, we have declared to the compiler that we are interested 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.

```
void main() {
  int* x,y;
  x = malloc(sizeof(int));
  y = malloc(sizeof(int));
  *x = 42;
  y = x;
  *y = 13;
}
```



9.5 Animation: Pointers in a C program.

Here, as with 'Atlantis', `x` and `y` are being used on two different levels. One is that `x` refers to the contents of register 123, so it names 123. 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, recall the Padding

[†] This distinction is familiar from 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 italic for variables such as *a*, which have a value, and typewriter for characters such as `a`, which are a value.

Lemma, Lemma 2.17, that every computable function has infinitely many indices. So it is easy for a computable function to have two different names. We see this in Theorem 9.1, where the conclusion that $\phi_k = \phi_{f(k)}$ does not say that the two indices are equal. Rather it says that they describe machines that give rise to the same input/output relationship.

Another example is that in the proof $g(n)$ is this.

$$\phi_{g(n)}(x) = \phi_{s(e_0,n)}(x) = \begin{cases} \phi_{\phi_n(n)}(x) & \text{– if } \phi_n(n)\downarrow \\ \uparrow & \text{– otherwise} \end{cases} \qquad (*)$$

So $g(n)$, $s(e_0, n)$, and $\phi_n(n)$ are names for the same function. Again, equality of the named functions does not imply that the names are equal.

Informally, what $g(n)$ names is the procedure, "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 $\phi_n(n)$." So here also we see the use-mention distinction.

One way in which this distinction between what is named and the name itself is important is that regardless of whether $\phi_n(n)$ converges, we can nonetheless compute the index $g(n)$ and from it the instruction set $\mathcal{P}_{g(n)}$. 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 Your friend asks you about the proof of the Fixed Point Theorem, Theorem 9.1. "The last line says $\phi_{g(v)} = \phi_{\phi_v(v)}$; isn't this just saying that $g(v) = \phi_v(v)$? Why the circumlocution?" What can you say?

✓ 9.7 Show each. (A) There is an index e such that $\phi_e = \phi_{e+7}$. (B) There is an e such that $\phi_e = \phi_{2e}$.

9.8 What conclusion can you draw by applying the Fixed Point Theorem to the adds-five function $x \mapsto x + 5$? Generalize.

9.9 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 $x \mapsto 42$.

✓ 9.10 We will prove that there is an m such that $W_m = \{ x \mid \phi_m(x)\downarrow \} = \{ m^2 \}$.
(A) Produce this uniformly computable family of functions.

$$\phi_{s(e_0,x)}(y) = \begin{cases} 42 & \text{– if } y = x^2 \\ \uparrow & \text{– otherwise} \end{cases}$$

(B) Observe that e_0 is fixed so that $s(e_0, x)$ is a function of one variable only, and call that function $g\colon \mathbb{N} \to \mathbb{N}$.

(c) Apply the Fixed Point Theorem to get the desired m.

9.11 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 the function $p\colon \mathbb{N} \to \mathbb{N}$ such that $p(x)$ is the x-th prime is computable. (b) Use p and the *s-m-n* Theorem to get that this family of functions is uniformly computable: $\phi_{s(e_0,x)}(y)$ is 42 if $y = p(x)$, and $\phi_{s(e_0,x)}(y)$ diverges otherwise. (c) Draw the desired conclusion.

✓ 9.12 Prove that there exists $m \in \mathbb{N}$ such that $W_m = \{ y \mid \phi_m(y)\downarrow \} = \{ 10^m \}$.

9.13 Show that there must be a pair of Turing machines $\mathcal{P}_e$ and $\mathcal{P}_{\hat{e}}$ with the same input-output behavior, such that all of the digits in $\hat{e}$ are one larger than the matching digits in e, except that a 9 in e changes to a 0 in $\hat{e}$. For instance, we might have $e = 35$ and $\hat{e} = 46$, or we might have $e = 29$ and $\hat{e} = 30$.

9.14 Show there is an index n so that $W_n = \{ x \mid \phi_n(x)\downarrow \} = \{ 0, 1, \ldots n \}$.

✓ 9.15 Show that there is a total and computable function ϕ that names itself in the sense that it is constant, meaning that there is $k \in \mathbb{N}$ with $\phi(x) = k$ for all inputs, and its index equals that constant, $\phi = \phi_k$.

9.16 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 $f\colon \mathbb{N} \to \mathbb{N}$ so that $\phi_n = \phi_{f(n)}$?

9.17 Prove or disprove the existence of each. (a) $W_m = \{ y \mid \phi_m(y)\downarrow \} = \mathbb{N} - \{ m \}$ (b) $W_m = \{ x \mid \phi_m(x)\uparrow \}$

9.18 Corollary 9.3 shows that there is a computable function ϕ_m with domain $\{ m \}$.
(a) Show that there is a computable function ϕ_m with domain $\{ m + 1 \}$.
(b) Is there a computable function ϕ_m with range $\{ 2m \}$?

9.19 Prove that K is not an index set. *Hint:* use Corollary 9.3 and the Padding Lemma, Lemma 2.17.

9.20 We can extend the Fixed Point Theorem to show that not only does any computable and total $f\colon \mathbb{N} \to \mathbb{N}$ have a fixed point, it has infinitely many distinct ones. *Hint:* prove that assuming that f's collection of fixed points $F = \{ k \in \mathbb{N} \mid \phi_k = \phi_{f(k)} \}$ is finite leads to a contradiction.
(a) Show that there is a partial recursive function $g\colon \mathbb{N} \to \mathbb{N}$ whose indices are not members of F, i.e., where $g = \phi_e$ implies that $e \notin F$.
(b) Suppose that such a g has index e_0, so that $g = \phi_{e_0}$. Consider this function, which is clearly total and computable.

$$h(x) = \begin{cases} e_0 & \text{– if } x \in F \\ f(x) & \text{– otherwise} \end{cases}$$

Show that h has no fixed point, contradicting the Fixed Point theorem.

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. Naturally, the rooms were numbered 0, 1, ... One day when every room was occupied, someone new came to the front desk; could the hotel accommodate? The clerk hit on an idea. They moved each guest up a room, so the guest in room n moved to room $n+1$, leaving room 0 empty. Thus this hotel always has space for a new guest, or a finite number of new guests.

Plenty of empty rooms.

Next a bus rolled in with infinitely many people $p_0, p_1, \ldots$ The clerk had the brainstorm to move each guest to the room with twice their current number, putting the guest from room n into room $2n$. Now, with the odd-numbered rooms empty, p_i can go in room $2i+1$, and everyone has a room.

Then in rolled a convoy of buses, infinitely many of them, each with infinitely many people: $B_0 = \{p_{0,0}, p_{0,1}, \ldots\}$, and $B_1 = \{p_{1,0}, p_{1,1}, \ldots\}$, etc. By now the spirit was 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 for any number of guests there is a sufficiently clever method to have them all fit. 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 $f_{i,0}$, $f_{i,1}$, ... And, no surprise, there are infinitely many floors F_0, F_1, ... One day when the hotel is empty a fleet of 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?

[†] 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.

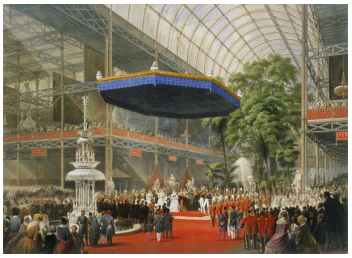
EXTRA

II.B Unsolvability in intellectual culture

Unsolvability results such as the Halting problem 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 an impact on the wider intellectual world.

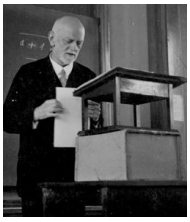
The discussion here is in the context of the history of European intellectual culture, the context in which early Theory of Computation results appeared. A broader view is beyond our scope.

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 Europe, many people 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.[†]



Queen Victoria opens the Great Exhibition, 1851

In science this optimism was captured by the physicist 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."



David Hilbert 1862–1943

The twentieth century physicist R Feynman likened science to working out nature's rules, "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, soon enough those would be done also.

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

[†] This is not to say that the perception was 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.

must know — we will know." ('*Ignorabimus*' means 'that which we must be forever ignorant of' or 'that thing which 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.†

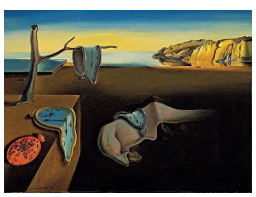
But starting in the early 1900's, that changed. Exhibit A is the picture to the right. That the modern mastery of mechanisms can have terrible effect on human bodies became apparent to everyone during World War I, 1914–1918. Ten million military men died. Overall, seventeen million people died. With universal conscription, probably the men in this picture did not want to be here. Probably they were killed by someone who also did not want to be there, who never knew that he killed them, and who simply entered coordinates into a firing mechanism. For people at those coordinates, it didn't matter how brave they were, or how strong, or how right was their cause — they died. The zeitgeist shifted: Pandora's box was now opened and the world had become not at all ordered, reasoned, or sensible.

World War I trench dead

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 the cultural shift directly. After experiments during a solar eclipse in 1919 provided strong support for his theories, he 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.

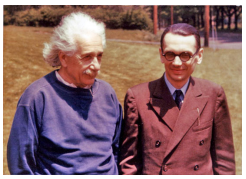


S Dali's 1931 *Persistence of Memory*.

In mathematics, much the same inversion of the standing order happened in 1930 with K Gödel's announcement of the Incompleteness

† Below we will cite some things as turning points that occur before 1930; how can that be? For one thing, it is typical for cultural shifts to have muddled timelines. Another thing is that 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 it. Gödel was in the audience for Hilbert's address and during it whispered to O Taussky-Todd, "He doesn't get it."

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.[†]



Gödel and friend, 1947

This statement of hard limits seemed to many to be especially striking in mathematics, which traditionally held the 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." This is all the more impactful as Gödel's results are not about a specialized area of only technical interest but instead are about statements in the natural numbers and about proof itself, and so the hole that Gödel finds lies at the very foundation of rational thought.

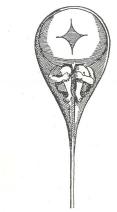
To be a mathematical proof, each step in an argument must be verifiable as either an axiom or as a deduction that is valid from the prior steps. So proving a mathematical theorem is a kind of computation.[‡] Thus, Gödel's Theorem and other uncomputability results are in the same vein. In fact, from a proof of the Halting problem we can get to a proof of Gödel's 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 with crude microscopes, speculated that the development of a fetus is that it basically just expands while retaining the essential features of one head, two arms, etc. They posited a 'homunculus', a small human-like figure that when given life swells to become a person.



Sperm, scientific illustration, 1695

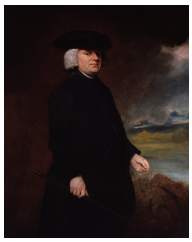
One issue with this theory is that the person may become a parent. So each homunculus contains its children? And grandchildren? This is the problem of infinite regress. Of course today we know that sperm and egg don't contain bodies, they contain DNA, which we may think of as code to create bodies.

[†] 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 because it is true it therefore 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.

Paley's watch In 1802, W Paley argued for the existence of a god from a perception of otherwise unexplainable 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.

This essay was very influential before the development by Darwin and Wallace of the theory of differential reproduction through natural selection.



William Paley
1743–1805

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 of all the rest.

This captures that for many thinkers before the theory of evolution, from among all the things in the world to marvel at — the graceful shell of a nautilus, the precision of an eagle's eye, or consciousness — the greatest wonder was self-reproduction. It may seem, for example, that making a machine to weave a rug is possible only because the rug is less complex than the machine. In this mindset having something that assembles a copy of itself appears to be a kind of magic, because a thing cannot be less complex than itself. But that's 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, $\mathcal{P}_m$'s name is its behavior.

Courtesy xkcd.com

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

[†] The easiest such program finds its source file on the disk and prints it. That is cheating.

language since it is low-level and so the details are not hidden.

We might think to include the source as a string within the source. Below is a try at that,[†] which we can call `try0.c`. But this is naive. The string would have to itself 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);}
```

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);}
```

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'." Another is "quine 'quine'." This is a linguistic analog of the self-reproducing programs where the second word plays the part of the data in a traditional program/data split, the same part as is played by `try3.c`'s first line string. That part is also played by 'produce' in "Produce the machine, and then do the machine."

We can express that in code. First consider quoting. To perform some action we ordinarily define a function and then call it as with `(f 1 2)`. As a consequence, if we want to produce a list of three strings then this

```
> (Boro is reading)
```

gives the error `Boro: undefined`. We must tell Racket not to evaluate these things, in this case not to evaluate the list in the usual way of taking the first entry to be a function and then applying it to the evaluation of the other entries.

```
> (quote (Boro is reading))
'(Boro is reading)
```

[†] 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. [#] The 10 is the ASCII encoding for newline and 34 is ASCII for a double quotation mark.

There is a single-character convenient shortcut.

```
> '(Boro is reading)
'(Boro is reading)
```

We can make this a function.

```
(define (P x)
  (list 'Boro 'is x))
```

```
> (P 'reading)
'(Boro is reading)
```

Using it gives the diagonalization routine.

```
(define (Diag p x)
  (p (list 'quote (p x))))
```

```
> (Diag P 'reading)
'(Boro is '(Boro is reading))
```

And here is a quine.

```
(define (Quine-1 x)
  (list x (list 'quote x)))
```

```
> (Quine-1 'reading)
'(reading 'reading)
> (Quine-1 'Quine-1)
'(Quine-1 'Quine-1)
```

Reflections on Trusting Trust K Thompson is one of the two main creators of the UNIX operating system. He worked at Bell Labs for much of his career and was central to the development of the C language. He pioneered the use of regular expressions, and sketched the definition of the UTF-8 encoding for Unicode. More recently was one of the creators of the Go language at Google.

Ken Thompson
b 1943

He has 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. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular. . . .
>
> 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.

This celebrated essay develops a quine and shows 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.

EXTRA

II.D Busy Beaver

For any $n \in \mathbb{N}$, the set of Turing machines having no more than n states is finite. So there are finitely machines $\mathcal{P}_e$ in this set that halt (on a blank tape). We can wonder if there are insights about computation in general and Turing machines in particular that we can get by understanding how small sized machines behave — for instance, do most n-state Turing machines halt, or only a few?

Tibor Radó
1895–1965

Define the function BB: $\mathbb{N} \to \mathbb{N}$ to give the minimal number of steps after which all of the size n machines that will ever halt on a blank tape have done so. Also let Σ: $\mathbb{N} \to \mathbb{N}$ be the largest number of 1's left on the tape by any n-state Turing machine, when started on a blank tape, after halting.

D.1 THEOREM (RADÓ, 1962) The functions BB and Σ are not computable.

Proof If BB were a computable function then to find whether some $\mathcal{P}_e$ halts on input e, we could run it for BB(n)-many steps. If $\mathcal{P}_e(e)$ has not halted by then, it never will. So computability of BB would contradict the unsolvability of the Halting problem. The function Σ is similar and is Exercise D.7. □

This BB may seem to be just one more uncomputable function among many However, it has the interesting property that any function f that grows faster than it — where $f(n) \geq \text{BB}(n)$ for all sufficiently large n — is also not computable, by the same argument as given in the proof. So we have an insight about what makes a function uncomputable: one way is to grow faster than any computable function.[†]

The Busy Beaver problem is: which n-state Turing Machine does the most computational work before halting?

Rare moment of rest

Think of this as a competition to produce the machine that sets the limit BB(n) or $\Sigma(n)$.[‡] A computation needs rules and here tradition fixes 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, there is a separate halt state that is not counted in the number of machine states, the machine is started on an empty tape, and where transitions are of the form $\Delta(\text{state, tape symbol}) = \langle \text{state, tape symbol, head shift} \rangle$.

What is known In the 1962 paper Radó covered the $n = 0$, $n = 1$, and $n = 2$ cases ($n = 0$ is trivial since it refers to a machine consisting only of a halting state). In 1964 Radó and Lin showed that $\Sigma(3) = 6$.

4.2 EXAMPLE This is the three state Busy Beaver machine, with halting state q_3.

[†] Note the connection with the Ackermann function, which we showed is not primitive recursive because it grows faster than any primitive recursive function. [‡] For many years after the problem was originally stated by T Radó, the competition was centered on Σ. However, recently it has become more common to discuss BB. In any event, the two are very closely related.

Δ	B	1
q_0	$q_1, 1, R$	$q_3, 1, R$
q_1	q_2, B, R	$q_1, 1, R$
q_2	$q_2, 1, L$	$q_0, 1, L$

In 1983 A Brady showed that $\Sigma(4) = 13$ and $\mathrm{BB}(4) = 107$.

In 2024, a team of researchers working together over the Internet and using the Coq proof verification system showed that $\Sigma(5) = 4\,098$ and $\mathrm{BB}(5) = 47\,176\,870$. Their machine computes $g(0)$, $g(g(0))$, ... until it halts, where g is this function.

$$g(n) = \begin{cases} (5n+18)/3 & \text{– if } n \text{ leaves a remainder of 0 on division by 3} \\ (5n+22)/3 & \text{– if } n \text{ leaves a remainder of 1} \\ \text{Halt} & \text{– otherwise} \end{cases}$$

This summarizes the current records.

n	1	2	3	4	5	6
$\mathrm{BB}(n)$	1	6	21	107	47 176 870	$\geq 10 \uparrow\uparrow 15$
$\Sigma(n)$	1	4	6	13	4 098	$\geq 10 \uparrow\uparrow 15$

The notation $10 \uparrow\uparrow 15$ means $10\hat{}(10\hat{}(10\hat{}\cdots)))$ with fifteen 10's.

How we find these After $n = 2$ the obvious place to start an attack on this problem is with a breadth-first search: there are finitely many n-state machines so run them all on a blank tape, dovetail, and await developments. That will quickly settle the question for a large number of machines. Of course, some of them won't halt or will run longer than our patience lasts. For some of these their action will be easy to determine from the source and we can hope to quickly reduce to a relatively few machines which we can study in depth, and so by exhaustion find the answer for this n.[†]

But what if for some n we find a machine that computes something that we don't know? For instance, what if n is big enough to allow a machine that halts if and only if it finds an odd perfect number? The $n = 6$ case seems to have a machine similar to this.

For more, there are a number of websites that cover the topic, including the latest results. Besides the Wikipedia page, the canonical site is `bbchallenge.org`. Some cover variations on machine standards such as considering machines with three or more symbols.

Not only are Busy Beaver numbers very hard to find, at some point they become impossible. In 2016, A Yedida and S Aaronson obtained an n for which $\mathrm{BB}(n)$ 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

[†] Brady (Brady 1983) reports that for $n = 4$ there are 5 280 such machines.

these axioms are consistent, so we believe that this machine doesn't halt. However, Gödel's Second Incompleteness Theorem shows that there is no way to prove that the axioms are consistent using the axioms themselves. So in this case the solution to the Busy Beaver problem is unknowable in that even if we were given the number $\mathrm{BB}(n)$ we could not use our axioms to prove that it is right, to prove that this machine halts.

In summary, 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 How many Turing machines are there of the style used in this discussion?

✓ D.4 Write and run a routine to compute $g(0), g(g(0)), \ldots$.

✓ D.5 Give a diagonal construction of a function that is eventually greater than any computable function.

✓ D.6 Show that there are uncomputable functions with the property that they grow no faster than the computable function $f(x) = 1$. *Hint:* An argument by countability works.

D.7 This is a proof that Σ is not computable. Let $f\colon \mathbb{N} \to \mathbb{N}$ be any total computable function. We will show that $\Sigma(n) > f(n)$ for infinitely many n, and so $\Sigma \neq f$.

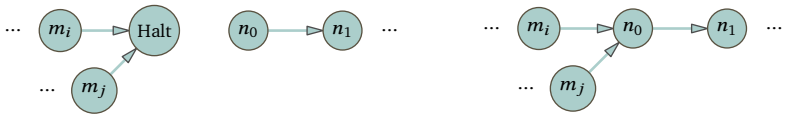
(A) Show that there is a Turing Machine $\mathcal{M}_j$ having j many states that writes j-many 1's to a blank tape.

(B) Let $F\colon \mathbb{N} \to \mathbb{N}$ be this function.

$$F(m) = (f(0) + 0^2) + (f(1) + 1^2) + (f(2) + 2^2) + \cdots + (f(m) + m^2)$$

Argue that it has the three properties: if $0 < m$ then $f(m) < F(m)$, and $m^2 \leq F(m)$, and $F(m) < F(m+1)$.

(C) The illustration below shows the composition of two Turing machines. On the right, we have combined the final states of the first machine from the left with the start state of the second.



Consider the Turing machine $\mathcal{P}$ that performs $\mathcal{M}_j$ and followed by the machine $\mathcal{M}_F$, and then follows by another copy of the machine $\mathcal{M}_F$. Show that its productivity is $F(F(j))$ and that it has $j + 2n_F$ many states.

(D) Finish by comparing 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 since 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. Argue that $f(j + 2n_F) \leq \Sigma(j + 2n_F)$.

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 in the most straightforward way: we 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 the top row to the bottom is Cantor's pairing function because it outputs pairs, while its inverse from the bottom to the top is the unpairing function.

First, unpairing. Given $\langle x, y \rangle$, we determine the diagonal that it lies on with $1 + 2 + \cdots n = n(n+1)/2$

```
;; 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 $(1 + 2 + \cdots x) + y$.

```
;; 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
>
```

We might wonder whether the arguments to `cantor-unpairing` should be the two numbers x and y as we've got above, or the pair $\langle x, y \rangle$. But it doesn't matter much because if we want the pair then we can instead use the `apply` operator.

```
> (cantor-unpairing 10 12)
263
> (apply cantor-unpairing '(10 12))
263
```

Next, the pairing function. 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.

$$d = \frac{-1+\sqrt{1-4\cdot 1\cdot(-2t)}}{2\cdot 1} = \frac{-1+\sqrt{1+8t}}{2}$$

(We kept only the '+' of the '±' because the other root is negative.) Given a pairing function input c, to find the number of the diagonal containing the $\langle x,y\rangle$ 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 we get $\langle x,y\rangle$ by seeing how far c is along that diagonal.

```
;; 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)
```

With those we can reproduce the table from the section's start.

```
> (for ([i '(0 1 2 3 4 5)])
     (displayln (cantor-pairing i)))
(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)])
    (displayln (cantor-pairing-3 i)))
(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)])
    (displayln (cantor-pairing-4 i)))
(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. What the heck, just for fun we can extend to tuples of any size.

For the function unpair: $\mathbb{N}^k \to \mathbb{N}$, 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-unparing-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 : $\mathbb{N} \to \mathbb{N}^k$, the awkwardness is that the routine can't know the intended 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)])
    (displayln (cantor-pairing-arity 4 i)))
(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 because 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)])
    (displayln (cantor-pairing-omega i)))
()
(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 then 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 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 first number matched by `cantor-pairing-omega` with a four-tuple?

Part Two

Automata

Chapter III Languages, Grammars, and Graphs

This chapter covers three topics we will use as a foundation for later work.

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, $\mathbb{B} = \{\texttt{0}, \texttt{1}\}$. 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 $\mathcal{L} = \{\texttt{1}, \texttt{10}, \texttt{11}, \texttt{100}, \ldots\}$.

1.3 Example Another language over $\mathbb{B}$ is the finite set $\{\texttt{1000001}, \texttt{1100001}\}$.

1.4 Example Let $\Sigma = \{\texttt{a}, \texttt{b}\}$. The language consisting of strings where the number of a's is twice the number of b's is $\mathcal{L} = \{\varepsilon, \texttt{aab}, \texttt{aba}, \texttt{baa}, \texttt{aaaabb}, \ldots\}$.

1.5 Example Let $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. The language of length-two strings over that alphabet is $\mathcal{L}_2 = \Sigma^2 = \{\texttt{aa}, \texttt{ab}, \texttt{ba} \ldots, \texttt{cc}\}$. Over the same alphabet, this language consists of length-three strings whose characters are in ascending order.

$$\mathcal{L}_3 = \{\texttt{aaa}, \texttt{bbb}, \texttt{ccc}, \texttt{aab}, \texttt{aac}, \texttt{abb}, \texttt{abc}, \texttt{acc}, \texttt{bbc}, \texttt{bcc}\}$$

1.6 Definition A palindrome is a string that reads the same forwards as backwards.

Some palindromes in English are `kayak`, `noon`, and `racecar`.

1.7 Example The language of palindromes over $\Sigma = \{\texttt{a}, \texttt{b}\}$ is $\mathcal{L} = \{\sigma \in \Sigma^* \mid \sigma = \sigma^{\mathsf{R}}\}$. A few members are `abba`, `aaabaaa`, `a`, and ε.

1.8 Example Let $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. 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

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 368.

language $\mathcal{L} = \{ \mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \in \Sigma^* \mid i,j,k \in \mathbb{N} \text{ and } i^2+j^2=k^2 \}$. Some members are $\mathtt{aaabbbbccccc} = \mathtt{a}^3\mathtt{b}^4\mathtt{c}^5$, and $\mathtt{a}^5\mathtt{b}^{12}\mathtt{c}^{13}$, and $\mathtt{a}^8\mathtt{b}^{15}\mathtt{c}^{17}$.

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 $\hat{\mathcal{L}} = \{\varepsilon\}$. These two languages are different, because the first has no members while the second has one.

The motivation for taking 'language' to be a set of strings is that we can imagine that Σ is the set of words in a dictionary and a sentence is a string of words, $\sigma \in \Sigma^*$. However, this thinking allows a language to be any string of words at all, while a natural language such as English we must follow 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 $\Sigma = \{\mathtt{B}, \mathtt{1}\}$. 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, $\mathcal{L}_0{}^\frown\mathcal{L}_1$ or $\mathcal{L}_0\mathcal{L}_1$, is the language of concatenations, $\{\sigma_0{}^\frown\sigma_1 \mid \sigma_0 \in \mathcal{L}_0 \text{ and } \sigma_1 \in \mathcal{L}_1\}$.

For any language $\mathcal{L}$, when $k > 0$ the power $\mathcal{L}^k$ is the language consisting of the concatenation of k-many members, $\mathcal{L}^k = \{\sigma_0{}^\frown\cdots{}^\frown\sigma_{k-1} \mid \sigma_i \in \mathcal{L}\}$.[†] We take $\mathcal{L}^0 = \{\varepsilon\}$.[‡] The Kleene star of a language $\mathcal{L}^*$ is the language consisting of the concatenation of any number of strings.

$$\mathcal{L}^* = \{\sigma_0{}^\frown\cdots{}^\frown\sigma_{k-1} \mid k \in \mathbb{N} \text{ and } \sigma_0,\ldots,\sigma_{k-1} \in \mathcal{L}\} = \mathcal{L}^0 \cup \mathcal{L}^1 \cup \mathcal{L}^2 \cup \cdots$$

This includes the concatenation of 0-many strings so $\varepsilon \in \mathcal{L}^*$ even if $\mathcal{L} = \varnothing$. The reversal of a language is the language of reversals, $\mathcal{L}^{\mathsf{R}} = \{\sigma^{\mathsf{R}} \mid \sigma \in \mathcal{L}\}$.

As described in Section A, we extend the star notation beyond languages to alphabets Σ, defining Σ^* to be the set of strings of characters from that alphabet.

1.14 Example Where the language is the set of bitstrings $\mathcal{L} = \{\mathtt{1000001}, \mathtt{1100001}\}$ then the reversal is $\mathcal{L}^{\mathsf{R}} = \{\mathtt{1000001}, \mathtt{10000011}\}$.

1.15 Example If the language $\mathcal{L}$ consists of two strings $\{\mathtt{a}, \mathtt{bc}\}$ then its second power is $\mathcal{L}^2 = \{\mathtt{aa}, \mathtt{abc}, \mathtt{bca}, \mathtt{bcbc}\}$. Its Kleene star is the union of the powers.

$$\mathcal{L}^* = \{\varepsilon, \mathtt{a}, \mathtt{bc}, \mathtt{aa}, \mathtt{abc}, \mathtt{bca}, \mathtt{bcbc}, \mathtt{aaa}, \ldots\}$$

1.16 Remark For the above definition of the operation $\mathcal{L}^k$ of repeatedly choosing strings, there are two ways that we could go. We could choose a string σ and then

[†] Don't confuse this with the Cartesian product operation for sets. [‡] We take $\sigma^0 = \varepsilon$ since ε is the identity element for string concatenation. (We saw the same reasoning when we defined the sum of zero-many numbers to be 0 and the product of zero-many numbers to be 1, on page 21.)

repeat it, and so get the set of all σ^k. Or we could repeatedly choose strings, getting the set of all $\sigma_0 \frown \sigma_1 \frown \cdots \frown \sigma_{k-1}$. The second is more useful so that's what we use.

We finish by describing two ways that a machine can relate to a language. 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. The other way relates to languages that are computably enumerable but not computable. For these 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 further, if the input is not an element of the language then the machine will never incorrectly report that it is an element. (The machine may determine that it is not, or it may simply not report a conclusion 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) $\{\sigma \in \mathbb{B}^* \mid \text{the number of } 0\text{'s plus the number of } 1\text{'s equals } 3\}$
(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 $\beta \frown \beta^{\mathsf{R}}$ is a palindrome. Do all palindromes have that form?

✓ 1.21 Let $\mathcal{L}_0 = \{\varepsilon, \mathsf{a}, \mathsf{aa}, \mathsf{aaa}\}$ and $\mathcal{L}_1 = \{\varepsilon, \mathsf{b}, \mathsf{bb}, \mathsf{bbb}\}$. (A) List all the members of $\mathcal{L}_0 \frown \mathcal{L}_1$. (B) List all the members of $\mathcal{L}_1 \frown \mathcal{L}_0$. (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 then list all of them.
(A) $\{\sigma \in \{a, b\}^* \mid \sigma = \mathsf{a}^n\mathsf{b} \text{ for } n \in \mathbb{N}\}$
(B) $\{\sigma \in \{a, b\}^* \mid \sigma = \mathsf{a}^n\mathsf{b}^n \text{ for } n \in \mathbb{N}\}$
(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 $\mathcal{L} = \{\mathsf{a}, \mathsf{ab}\}$, list each. (A) $\mathcal{L}^2$ (B) $\mathcal{L}^3$ (C) $\mathcal{L}^1$ (D) $\mathcal{L}^0$

1.24 Where $\mathcal{L}_0 = \{\mathsf{a}, \mathsf{ab}\}$ and $\mathcal{L}_1 = \{\mathsf{b}, \mathsf{bb}\}$ find each. (A) $\mathcal{L}_0 \frown \mathcal{L}_1$ (B) $\mathcal{L}_1 \frown \mathcal{L}_0$ (C) $\mathcal{L}_0^2$ (D) $\mathcal{L}_1^2$ (E) $\mathcal{L}_0^2 \frown \mathcal{L}_1^2$

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 find the least number of elements possible and the greatest number possible? (A) $\mathcal{L}_0 \cup \mathcal{L}_1$ (B) $\mathcal{L}_0 \cap \mathcal{L}_1$ (C) $\mathcal{L}_0 \frown \mathcal{L}_1$ (D) $\mathcal{L}_1^2$ (E) ${\mathcal{L}_1}^{\mathsf{R}}$ (F) ${\mathcal{L}_0}^* \cap {\mathcal{L}_1}^*$

1.26 What is the language that is the Kleene star of the empty set, $\varnothing^*$?

✓ 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 $\mathcal{L}^* = \mathcal{L}^0 \cup \mathcal{L}^1 \cup \mathcal{L}^2 \cup \cdots$.

1.31 Consider the empty language $\mathcal{L}_0 = \varnothing$. For any language $\mathcal{L}_1$, describe $\mathcal{L}_1 \frown \mathcal{L}_0$.

1.32 Languages are sets and so the operations of union and intersection apply.

(A) Name the shortest five strings in the union of $A = \{\mathtt{a}\}^*$ with $B = \{\mathtt{b}\}^*$.

(B) Suppose that Σ_0 and Σ_1 are disjoint, and that $\mathcal{L}_0$ and $\mathcal{L}_1$ are finite languages over those alphabets respectively. What is the number of elements in their union?

(C) Fill in the blank: the union of a language over Σ_0 with a language over Σ_1 is a language over ____ .

(D) Formulate 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) With the language 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 $\mathcal{L}_0, \ldots \mathcal{L}_{k-1}$ 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. (For an alphabet Σ, a language is a subset, $\mathcal{L} \subseteq \Sigma^*$. So its complement is $\mathcal{L}^{\mathsf{c}} = \Sigma^* - \mathcal{L}$, also a language over Σ.)

1.34 What is the difference between the languages $\mathcal{L} = \{\sigma \in \Sigma^* \;\big|\; \sigma = \sigma^{\mathsf{R}}\}$ and $\hat{\mathcal{L}} = \{\sigma \frown \sigma^{\mathsf{R}} \;\big|\; \sigma \in \Sigma^*\}$?

1.35 For any language $\mathcal{L} \subseteq \Sigma^*$ we can form the set of prefixes.

$$\text{Pref}(\mathcal{L}) = \{\tau \in \Sigma^* \;\big|\; \sigma \in \mathcal{L} \text{ and } \tau \text{ is a prefix of } \sigma\}$$

Where $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $\mathcal{L} = \{\mathtt{abaaba}, \mathtt{bba}\}$, find $\text{Pref}(\mathcal{L})$.

1.36 This explains why we define $\mathcal{L}^0 = \{\varepsilon\}$ even when $\mathcal{L} = \emptyset$.
(A) Show that $\mathcal{L}^m \frown \mathcal{L}^n = \mathcal{L}^{m+n}$ for any $m, n \in \mathbb{N}^+$.
(B) Show that if $\mathcal{L}_0 = \emptyset$ then $\mathcal{L}_0 \frown \mathcal{L}_1 = \mathcal{L}_1 \frown \mathcal{L}_0 = \emptyset$.
(C) Argue that if $\mathcal{L} \neq \emptyset$ then the only sensible definition for $\mathcal{L}^0$ is $\{\varepsilon\}$.
(D) Why would $\mathcal{L} = \emptyset$ throw a monkey wrench if the works unless we define $\mathcal{L}^0 = \{\varepsilon\}$?

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 The description of the powers of a language in Definition 1.13 writes $\sigma_0 \frown \cdots \frown \sigma_{k-1}$, without parentheses, and so doesn't specify a construction for the set. For that construction the natural approach is: $\mathcal{L}^0 = \{\varepsilon\}$, and $\mathcal{L}^{k+1} = \mathcal{L}^k \frown \mathcal{L}$. Verify that this gives the same set as in the definition.

1.39 True or false: if $\mathcal{L} \frown \mathcal{L} = \mathcal{L}$ then either $\mathcal{L} = \emptyset$ or $\varepsilon \in \mathcal{L}$. 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 $\mathcal{L} \frown \{\varepsilon\} = \mathcal{L}$ and $\{\varepsilon\} \frown \mathcal{L} = \mathcal{L}$.
(C) Language concatenation need not be commutative; there are languages such that $\mathcal{L}_0 \frown \mathcal{L}_1 \neq \mathcal{L}_1 \frown \mathcal{L}_0$.
(D) Language concatenation is associative, $(\mathcal{L}_0 \frown \mathcal{L}_1) \frown \mathcal{L}_2 = \mathcal{L}_0 \frown (\mathcal{L}_1 \frown \mathcal{L}_2)$.
(E) $(\mathcal{L}_0 \frown \mathcal{L}_1)^R = \mathcal{L}_1{}^R \frown \mathcal{L}_0{}^R$.
(F) Concatenation is left distributive over union, $(\mathcal{L}_0 \cup \mathcal{L}_1) \frown \mathcal{L}_2 = (\mathcal{L}_0 \frown \mathcal{L}_2) \cup (\mathcal{L}_1 \frown \mathcal{L}_2)$, and also right distributive.
(G) The empty language is an annihilator for concatenation, $\emptyset \frown \mathcal{L} = \mathcal{L} \frown \emptyset = \emptyset$.
(H) The Kleene star operation is idempotent, $(\mathcal{L}^*)^* = \mathcal{L}^*$.

Section III.2 Grammars

We have defined a 'language' as a set of strings. But this allows for any willy-nilly set. In practice usually a language is governed 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 pattern for English. Artificial languages such as programming languages also have syntax rules, usually very strict rules.

A grammar is a set of rules for the formation of strings in a language. In an aphorism, grammars are the language of languages.

Definition Before the formal definition we'll first see an example.

2.1 Example A full set of rules for a natural language such as English would be quite large. But here is a subset that gives a sense of what a set of rules would look like: (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 those rules.

⟨*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, →.[†] To the left of the arrow is the rule's head while to the right is its body or expansion. Sometimes two rules have the same head, as with ⟨*noun phrase*⟩. There are also two rules for ⟨*noun*⟩ but we have abbreviated by combining the bodies using the '|' pipe symbol.[‡]

The rules use two different kinds of components. The ones written in typewriter type, such as `the`, are elements of Σ, the alphabet of the language. These components are terminals.[#] The ones with angle brackets and italics, such as ⟨*sentence*⟩, are nonterminals. These are like variables for intermediate steps and do not appear in the language's strings.

The two symbols '→' 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 above we shall have that every derivation starts with ⟨*sentence*⟩. During a derivation, intermediate strings contain a mix of nonterminals and terminals. In our grammars every rule has a head with a single nonterminal. To get the next string, pick a nonterminal in the present string, find a rule where that nonterminal is a head, and then substitute that rule's body.

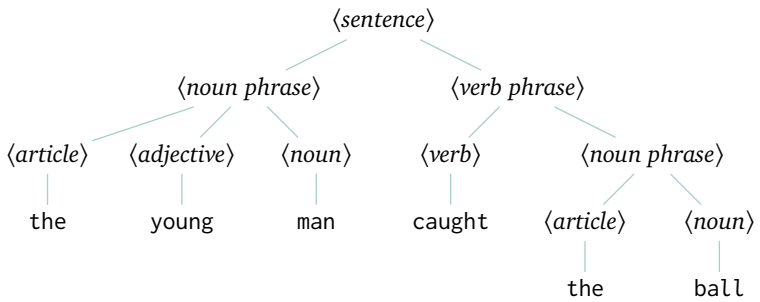
Below is one such derivation. Note that while the single line arrow → is for rules, we use the double line arrow ⇒ for derivations.[§]

[†] 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. [§] Read '⇒' aloud as "expands to."

$$\begin{aligned}
\langle \textit{sentence} \rangle &\Rightarrow \langle \textit{noun phrase} \rangle \ \langle \textit{verb phrase} \rangle \\
&\Rightarrow \langle \textit{article} \rangle \ \langle \textit{adjective} \rangle \ \langle \textit{noun} \rangle \ \langle \textit{verb phrase} \rangle \\
&\Rightarrow \texttt{the} \ \langle \textit{adjective} \rangle \ \langle \textit{noun} \rangle \ \langle \textit{verb phrase} \rangle \\
&\Rightarrow \texttt{the young} \ \langle \textit{noun} \rangle \ \langle \textit{verb phrase} \rangle \\
&\Rightarrow \texttt{the young man} \ \langle \textit{verb phrase} \rangle \\
&\Rightarrow \texttt{the young man} \ \langle \textit{verb} \rangle \ \langle \textit{noun phrase} \rangle \\
&\Rightarrow \texttt{the young man caught} \ \langle \textit{noun phrase} \rangle \\
&\Rightarrow \texttt{the young man caught} \ \langle \textit{article} \rangle \ \langle \textit{noun} \rangle \\
&\Rightarrow \texttt{the young man caught the} \ \langle \textit{noun} \rangle \\
&\Rightarrow \texttt{the young man caught the ball}
\end{aligned}$$

This is a leftmost derivation, which always substitutes for the leftmost nonterminal in that it first substitutes for the leftmost nonterminal on the right side of the first line, ⟨*noun phrase*⟩, then substitutes for the leftmost nonterminal on the second line, ⟨*article*⟩, etc. However, in general we could substitute for any nonterminal.

An alternative representation is the derivation tree or parse tree.[†]



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 take Σ and N to be 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.

We will use the convention that the start symbol is the head of the first rule.

2.3 Example This grammar describes algebraic expressions that involve only addition, multiplication, and parentheses.

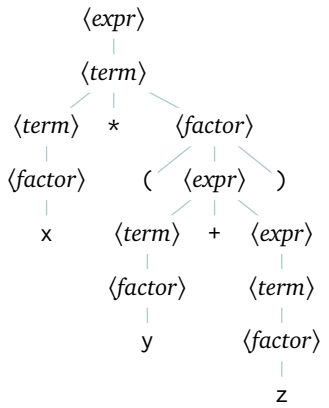
[†] The words 'terminal' and 'nonterminal' come from the position of those components in this tree.

[#] 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 → aBSc | abc, (ii) Ba → aB, (iii) Bb → 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. For instance, in the derivation S ⇒ aBSc ⇒ aBabcc ⇒ aaBbcc ⇒ aabbcc the third step uses (ii) and the fourth step uses (iii). Such grammars 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 appear most often in practice.

$\langle expr\rangle \;\to\; \langle term\rangle$ + $\langle expr\rangle \;\mid\; \langle term\rangle$
$\langle term\rangle \;\to\; \langle term\rangle$ * $\langle factor\rangle \;\mid\; \langle factor\rangle$
$\langle factor\rangle \;\to\; (\; \langle expr\rangle\;) \;\mid$ a $\mid$ b $\mid \ldots \mid$ z

Here is a derivation of the string `x*(y+z)`, along with its tree.

$$\begin{aligned}
\langle expr\rangle &\Rightarrow \langle term\rangle \\
&\Rightarrow \langle term\rangle * \langle factor\rangle \\
&\Rightarrow \langle factor\rangle * \langle factor\rangle \\
&\Rightarrow \mathtt{x} * \langle factor\rangle \\
&\Rightarrow \mathtt{x} * (\; \langle expr\rangle\;) \\
&\Rightarrow \mathtt{x} * (\; \langle term\rangle + \langle expr\rangle\;) \\
&\Rightarrow \mathtt{x} * (\; \langle term\rangle + \langle term\rangle\;) \\
&\Rightarrow \mathtt{x} * (\; \langle factor\rangle + \langle term\rangle\;) \\
&\Rightarrow \mathtt{x} * (\; \langle factor\rangle + \langle factor\rangle\;) \\
&\Rightarrow \mathtt{x} * (\; \mathtt{y} + \langle factor\rangle\;) \\
&\Rightarrow \mathtt{x} * (\; \mathtt{y} + \mathtt{z}\;)
\end{aligned}$$



In that example the rules for $\langle expr\rangle$ and $\langle term\rangle$ are recursive. But we don't get stuck in an infinite regress because the question is not whether we could perversely keep expanding $\langle expr\rangle$ forever. Instead, the question is whether, given a string such as `x*(y+z)`, we can find a terminating derivation.

In the prior example the nonterminals such as $\langle expr\rangle$ or $\langle term\rangle$ describe the role of those components in the language, as did the English grammar fragment's $\langle noun\ phrase\rangle$ and $\langle article\rangle$. That is why nonterminals are sometimes called 'syntactic categories'. But for examples and exercises we often use small grammars whose terminals and nonterminals do not have any particular meaning. For these, a common convention is to 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 $\to$ aSb $\mid \varepsilon$

Here is a derivation of the string $\mathtt{a}^2\mathtt{b}^2 = \mathtt{aabb}$.

$$\mathtt{S} \;\Rightarrow\; \mathtt{aSb} \;\Rightarrow\; \mathtt{aaSbb} \;\Rightarrow\; \mathtt{aa}\varepsilon\mathtt{bb} = \mathtt{aabb}$$

Similarly, $S \Rightarrow \mathtt{a}S\mathtt{b} \Rightarrow \mathtt{aa}S\mathtt{bb} \Rightarrow \mathtt{aaa}S\mathtt{bbb} \Rightarrow \mathtt{aaa}\varepsilon\mathtt{bbb} = \mathtt{aaabbb}$ is a derivation of $\mathtt{a}^3\mathtt{b}^3$. With this grammar, derivable strings have the form $\mathtt{a}^n\mathtt{b}^n$ for $n \in \mathbb{N}$.

We now give a complete description of how the production rules govern the derivations. Each rule in a context free grammar has the form 'head $\to$ 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 strings of terminals and non-terminals.

$$\tau_0 \frown \text{head} \frown \tau_1 \quad\Rightarrow\quad \tau_0 \frown \text{body} \frown \tau_1$$

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 strings of terminals and nonterminals, if they are related by a sequence of derivation steps then we 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 detailing how a Turing machine's instructions determine the evolution of the sequence of configurations that is a computation, on page 8. That is, production rules are like a program, directing a derivation. However, one difference is that Turing machines are deterministic, so that from a given input string there is a determined sequence of configurations. However here the sequence of derivation steps is nondeterministic in that from a given start symbol a derivation can branch out to go to many different ending strings.

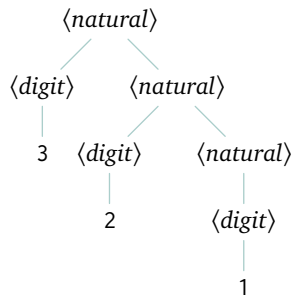
2.5 DEFINITION The language derived from a grammar is the set of strings of terminals having derivations that begin with the start symbol.

2.6 EXAMPLE This grammar's language is the set of representations of natural numbers.

$$\langle natural \rangle \;\to\; \langle digit \rangle \mid \langle digit \rangle \langle natural \rangle$$
$$\langle digit \rangle \;\to\; 0 \mid \ldots \mid 9$$

This is a derivation for the string 321, along with its parse tree.

$$\begin{aligned} \langle natural \rangle &\Rightarrow \langle digit \rangle \langle natural \rangle \\ &\Rightarrow 3\langle natural \rangle \\ &\Rightarrow 3\langle digit \rangle \langle natural \rangle \\ &\Rightarrow 32\langle natural \rangle \\ &\Rightarrow 32\langle digit \rangle \\ &\Rightarrow 321 \end{aligned}$$



2.7 EXAMPLE The language of this grammar

$$\langle natural \rangle \;\to\; \varepsilon \mid 1\langle natural \rangle$$

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.

$$S \;\to\; 00 \mid 01 \mid 10 \mid 11$$

This gives the length 3 bitstrings by using the nonterminals to keep count.

$$A \;\to\; 0B \mid 1B$$
$$B \;\to\; 0C \mid 1C$$
$$C \;\to\; 0 \mid 1$$

A derivation of 101 is $A \Rightarrow 1B \Rightarrow 10C \Rightarrow 101$.

2.9 EXAMPLE This grammar

$$\mathtt{S} \;\longrightarrow\; \mathtt{aSb} \;\mid \mathtt{aS} \;\mid \mathtt{a} \;\mid \mathtt{Sb} \;\mid \mathtt{b}$$

generates the language $\mathcal{L} = \{\, \mathtt{a}^i\mathtt{b}^j \in \{\mathtt{a},\mathtt{b}\}^* \;\big|\; i \neq 0 \text{ or } j \neq 0 \,\}$.

This is the first grammar that we have seen where the generated language is not clear, so we will do a 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 show that any derivation step $\tau_0 \frown \text{head} \frown \tau_1 \;\Rightarrow\; \tau_0 \frown \text{body} \frown \tau_1$ only adds a's on the left and b's on the right, so every string in the language has the form $\mathtt{a}^i\mathtt{b}^j$. 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 $\sigma = \mathtt{a}$ or $\sigma = \mathtt{b}$, each of which obviously has a derivation.

For the inductive step, fix $n \geq 1$ where every string from $\mathcal{L}$ of length $k = 1, \ldots k = n$ has a derivation, and let σ have length $n+1$. By the definition of $\mathcal{L}$ it has the form $\sigma = \mathtt{a}^i\mathtt{b}^j$. There are three cases: either or $i = j = 1$, or $i > 1$, or $j > 1$. The $\sigma = \mathtt{a}^1\mathtt{b}^1$ case is easy. For the $i > 1$ case, $\hat{\sigma} = \mathtt{a}^{i-1}\mathtt{b}^j$ is a string of length n, so by the inductive hypothesis it has a derivation $\mathtt{S} \Rightarrow \cdots \Rightarrow \hat{\sigma}$. Prefixing that derivation with a $\mathtt{S} \Rightarrow \mathtt{aS}$ 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

$$\begin{aligned} \langle \mathit{stmt} \rangle \;&\longrightarrow\; \texttt{if}\ \langle \mathit{bool} \rangle\ \langle \mathit{stmt} \rangle \\ \langle \mathit{stmt} \rangle \;&\longrightarrow\; \texttt{if}\ \langle \mathit{bool} \rangle\ \langle \mathit{stmt} \rangle\ \texttt{else}\ \langle \mathit{stmt} \rangle \end{aligned}$$

and this code string.

```
if enrolled(s) if studied(s) grade='P' else grade='F'
```

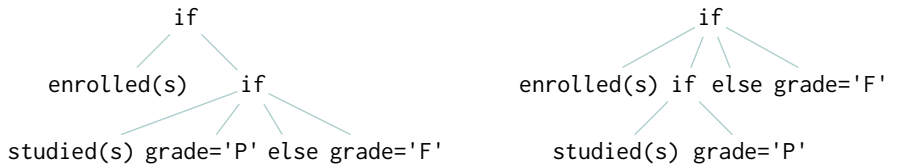
Here are the first two lines of one derivation

$$\begin{aligned} \langle \mathit{stmt} \rangle \;&\Rightarrow\; \texttt{if}\ \langle \mathit{bool} \rangle\ \langle \mathit{stmt} \rangle \\ &\Rightarrow\; \texttt{if}\ \langle \mathit{bool} \rangle\ \texttt{if}\ \langle \mathit{bool} \rangle\ \langle \mathit{stmt} \rangle\ \texttt{else}\ \langle \mathit{stmt} \rangle \end{aligned}$$

and here are the first two of another.

$$\begin{aligned} \langle \mathit{stmt} \rangle \;&\Rightarrow\; \texttt{if}\ \langle \mathit{bool} \rangle\ \langle \mathit{stmt} \rangle\ \texttt{else}\ \langle \mathit{stmt} \rangle \\ &\Rightarrow\; \texttt{if}\ \langle \mathit{bool} \rangle\ \texttt{if}\ \langle \mathit{bool} \rangle\ \langle \mathit{stmt} \rangle\ \texttt{else}\ \langle \mathit{stmt} \rangle \end{aligned}$$

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 C-like language code string that has been 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 makes clear which of the two possibilities is the intended one by using curly braces.)

A grammar is ambiguous if there is a string in its language with more than one leftmost derivation.

2.11 EXAMPLE This grammar for elementary algebra expressions

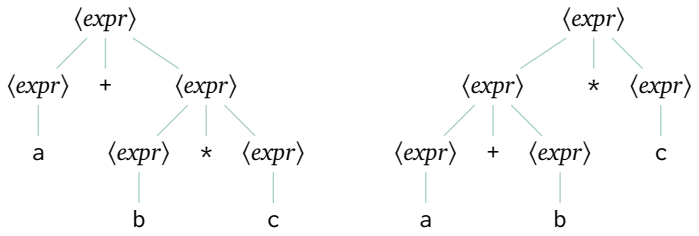
$$\begin{aligned}\langle expr\rangle &\to \langle expr\rangle + \langle expr\rangle \\ &\mid \langle expr\rangle * \langle expr>\rangle \\ &\mid (\ \langle expr\rangle\ )\ \mid \mathsf{a}\ \mid \mathsf{b}\ \mid \ldots \mathsf{z}\end{aligned}$$

is ambiguous because a+b*c has two leftmost derivations.

$$\begin{aligned}\langle expr\rangle &\Rightarrow \langle expr\rangle{+}\langle expr\rangle \Rightarrow \mathsf{a} + \langle expr\rangle \\ &\Rightarrow \mathsf{a} + \langle expr\rangle * \langle expr\rangle \Rightarrow \mathsf{a} + \mathsf{b} * \langle expr\rangle \Rightarrow \mathsf{a} + \mathsf{b} * \mathsf{c}\end{aligned}$$

$$\begin{aligned}\langle expr\rangle &\Rightarrow \langle expr\rangle * \langle expr\rangle \Rightarrow \langle expr\rangle + \langle expr\rangle * \langle expr\rangle \\ &\Rightarrow \mathsf{a} + \langle expr\rangle * \langle expr\rangle \Rightarrow \mathsf{a} + \mathsf{b} * \langle expr\rangle \Rightarrow \mathsf{a} + \mathsf{b} * \mathsf{c}\end{aligned}$$

The difference is reflected in different parse trees.



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 $(1 + 2) \cdot 3 = 9$.

In contrast, this grammar for the same language is unambiguous.

$$\begin{aligned}\langle expr\rangle &\to \langle expr\rangle + \langle term\rangle \\ &\mid \langle term\rangle \\ \langle term\rangle &\to \langle term\rangle * \langle factor\rangle \\ &\mid \langle factor\rangle \\ \langle factor\rangle &\to (\ \langle expr\rangle\ ) \\ &\mid \mathsf{a}\ \mid \mathsf{b}\ \mid \ldots\ \mid \mathsf{z}\end{aligned}$$

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 $\{\texttt{+}, \texttt{*}, \texttt{)}, \texttt{(}, \texttt{a} \ldots, \texttt{z}\}^*$ that cannot be derived.

2.13 Use the grammar of Example 2.1. (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 that cannot be derived from this grammar, or show that there are not three such strings.

2.14 Use this grammar.

⟨*natural*⟩ → ⟨*digit*⟩ | ⟨*digit*⟩⟨*natural*⟩
⟨*digit*⟩ → `0` | `1` | `...` | `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 metacharacters are used? (D) Derive `42`. Also give the associated parse tree. (E) Derive `993` and give its parse tree. (F) How can ⟨*natural*⟩ be defined in terms of ⟨*natural*⟩? 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

⟨*sentence*⟩ → ⟨*subject*⟩ ⟨*predicate*⟩
⟨*subject*⟩ → ⟨*article*⟩ ⟨*noun*⟩
⟨*predicate*⟩ → ⟨*verb*⟩ ⟨*direct object*⟩
⟨*direct object*⟩ → ⟨*article*⟩ ⟨*noun*⟩
⟨*article*⟩ → `the` | `a`
⟨*noun*⟩ → `car` | `wall`
⟨*verb*⟩ → `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.

⟨*sentence*⟩ → ⟨*subject*⟩ ⟨*predicate*⟩
⟨*subject*⟩ → ⟨*article*⟩ ⟨*noun1*⟩
⟨*predicate*⟩ → ⟨*verb*⟩ ⟨*direct-object*⟩
⟨*direct-object*⟩ → ⟨*article*⟩ ⟨*noun2*⟩
⟨*article*⟩ → `the` | `a` | ε
⟨*noun1*⟩ → `dog` | `flea`
⟨*noun2*⟩ → `man` | `dog`
⟨*verb*⟩ → `bites` | `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.

$$\begin{aligned}\langle sentence\rangle &\Rightarrow \langle subject\rangle\ \langle predicate\rangle\\ &\Rightarrow \langle article\rangle\ \langle noun1\rangle\ \langle predicate\rangle\\ &\Rightarrow \langle article\rangle\ \langle noun1\rangle\ \langle verb\rangle\ \langle direct\ object\rangle\\ &\Rightarrow \langle article\rangle\ \langle dog|flea\rangle\ \langle verb\rangle\ \langle article\rangle\ \langle noun2\rangle\\ &\Rightarrow \langle article\rangle\ \langle dog|flea\rangle\ \langle verb\rangle\ \langle article\rangle\ \langle man|dog\rangle\end{aligned}$$

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

$$\begin{aligned}S &\to TbU\\ T &\to aT \mid \varepsilon\\ U &\to aU \mid bU \mid \varepsilon\end{aligned}$$

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.

$$\begin{aligned}S &\to aABb\\ A &\to aA \mid a\\ B &\to Bb \mid b\end{aligned}$$

(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?

$$\begin{aligned}A &\to aA \mid B\\ B &\to bB \mid cA\end{aligned}$$

✓ 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.

$$\langle CNF\rangle \to (\ \langle Disjunction\rangle\ ) \wedge \langle CNF\rangle \mid (\ \langle Disjunction\rangle\ )$$

⟨*Disjunction*⟩ → ⟨*Literal*⟩ ∨ ⟨*Disjunction*⟩ | ⟨*Literal*⟩
⟨*Literal*⟩ → ¬ ⟨*Variable*⟩ | ⟨*Variable*⟩
⟨*Variable*⟩ → `x0` | `x1` | ...

For more, see Section 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 $\mathcal{L} = \{\, \mathtt{a}, \ldots, \mathtt{z} \,\}^*$

(B) the language of strings of at least one digit, $\{\, \sigma \in \{\, \mathtt{0}, \ldots, \mathtt{9} \,\}^* \;\big|\; |\sigma| \geq 1 \,\}$

✓ 2.30 This is a grammar for postal addresses. Note the use of the empty string ε to make some components optional, such as ⟨*opt suffix*⟩ and ⟨*apt num*⟩.

⟨*postal address*⟩ → ⟨*name*⟩ ⟨*EOL*⟩ ⟨*street address*⟩ ⟨*EOL*⟩ ⟨*town*⟩
⟨*name*⟩ → ⟨*personal part*⟩ ⟨*last name*⟩ ⟨*opt suffix*⟩
⟨*street address*⟩ → ⟨*house num*⟩ ⟨*street name*⟩ ⟨*apt num*⟩
⟨*town*⟩ → ⟨*town name*⟩ , ⟨*state or region*⟩
⟨*personal part*⟩ → ⟨*initial*⟩ . | ⟨*first name*⟩
⟨*last name*⟩ → ⟨*char string*⟩
⟨*opt suffix*⟩ → `Sr.` | `Jr.` | ε
⟨*house num*⟩ → ⟨*digit string*⟩
⟨*street name*⟩ → ⟨*char string*⟩
⟨*apt num*⟩ → ⟨*char string*⟩ | ε
⟨*town name*⟩ → ⟨*char string*⟩
⟨*state or region*⟩ → ⟨*char string*⟩
⟨*initial*⟩ → ⟨*char*⟩
⟨*first name*⟩ → ⟨*char string*⟩ | ε
⟨*char string*⟩ → ⟨*char*⟩ | ⟨*char*⟩ ⟨*char string*⟩ | ε
⟨*char*⟩ → `A` | `B` | ... `z` | `0` | ... `9` | `(space)`
⟨*digit string*⟩ → ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*digit string*⟩ | ε
⟨*digit*⟩ → `0` | ... `9`

The nonterminal ⟨*EOL*⟩ 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.

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 that is notable for having a small alphabet, while producing an infinite set of valid English sentences.

⟨*sentence*⟩ → `buffalo` ⟨*sentence*⟩ | ε

(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.

2.33 Here is a grammar for LISP.

⟨*s expression*⟩ → ⟨*atomic symbol*⟩
| `(` ⟨*s expression*⟩ `.` ⟨*s expression*⟩ `)`
| ⟨*list*⟩
⟨*list*⟩ → `(` ⟨*list-entries*⟩ `)`
⟨*list-entries*⟩ → ⟨*s expression*⟩
| ⟨*s expression*⟩ ⟨*list-entries*⟩
⟨*atomic symbol*⟩ → ⟨*letter*⟩ ⟨*atom part*⟩
⟨*atom part*⟩ → ε
| ⟨*letter*⟩ ⟨*atom part*⟩
| ⟨*number*⟩ ⟨*atom part*⟩
⟨*letter*⟩ → `a` | ... `z`
⟨*number*⟩ → `0` | ... `9`

Give a derivation for each string. (A) `(a . b)` (B) `(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 → S | ε

(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 $\mathcal{L} = \mathbb{B}^*$.

⟨*bit-string*⟩ → `0` | `1` | ⟨*bit-string*⟩ ⟨*bit-string*⟩

Show that it is ambiguous.

2.37

(A) Show that this grammar is ambiguous by producing two different leftmost derivations for `a-b-a`.

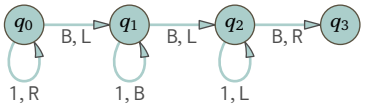
E → E `-` E | `a` | `b`

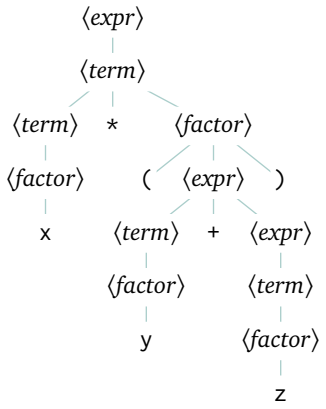
(B) Derive `a-b-a` from this grammar, which is unambiguous.

E → E `-` T | T
T → `a` | `b`

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.



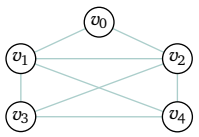


Definition We start with the basics.

3.1 Definition A simple graph is an ordered pair $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ where $\mathcal{N}$ is a 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.

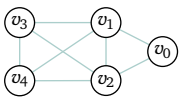
A graph is finite if it has finitely many vertices and infinite if it has infinitely many.

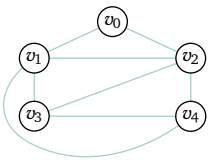
3.2 Example This simple graph $\mathcal{G}$ has five vertices and eight edges.



$$\mathcal{N} = \{v_0, \ldots v_4\}$$
$$\mathcal{E} = \{\{v_0, v_1\}, \{v_0, v_2\}, \ldots \{v_3, v_4\}\}$$

Important: a graph is not its picture. Both of the pictures below show the same graph as above because they show the same vertices connected with the same edges.





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 $e = \hat{v}v$.

There are many extensions of that definition for modeling different circumstances. One is to 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.

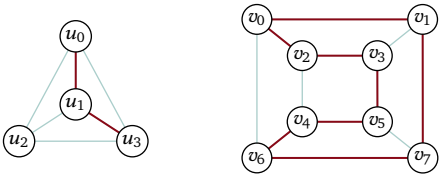
A very often-used variation is a directed graph or digraph, where edges have a direction, as in a road map that includes one-way streets. If an edge is directed from v to $\hat{v}$ then we can write it as $v\hat{v}$ but not in the other order. The Turing machine graph above is a digraph and also has 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 (often one vertex is singled out as the tree's root). A directed acyclic graph or DAG is a directed graph with no directed cycles.

Paths Many problems that we shall consider 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 $\langle v_0v_1,\ v_1v_2,\ \ldots\ v_{n-1}v_n\rangle$. 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 have that its first and last vertices are equal. A closed path with at least one edge is a cycle.[‡]

3.4 Example On the left, highlighted is a path from u_0 to u_3, $p = \langle u_0u_1, u_1u_3\rangle$. On the right the highlighted walk is a cycle.



3.5 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.

3.6 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.

3.7 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, so that $v_i, v_j \in \hat{\mathcal{N}}$ and $e = v_iv_j \in \mathcal{E}$ implies that $e \in \hat{\mathcal{E}}$ also, is an induced subgraph.

[†] Formally, we might extend the definition to allow some edges in $\mathcal{E}$ to be single-element sets. We will not specify how each variant is described. [‡] These terms are not completely standardized so you may see them used in other ways, especially in older work.

3.8 Example In the graph $\mathcal{G}$ on the left of Example 3.4, consider the edges in the highlighted path, $\hat{\mathcal{E}} = \{u_0u_1, u_1u_3\}$. 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, $\hat{\mathcal{N}} = \{u_0, u_1, u_3\}$, the induced subgraph is the triangle that adds the outer edge, $\mathcal{E} \cup \{u_0u_3\}$.

3.9 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.

In Chapter Five we will consider the graph of the possible branchings of a computation by a machine. Such a graph may have infinitely many nodes, as when there is a branch that does not halt. There, we will need the next result.

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, one with infinitely many vertices.

Proof Fix a vertex v_0. The graph is connected, so for every other vertex there is a path starting at v_0 that reaches it. For each of v_0's neighbors, there is a set of vertices that can be reached from v_0 via a path through that neighbor. There are infinitely many vertices so there must be a neighbor (unequal to v_0) where the set of vertices that are reachable in that way 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 the edge v_0v_1. Because v_1 has finitely many neighbors, there is a v_2 adjacent to v_1 (and unequal to either v_0 or v_1), through which there are paths to infinitely many of the graph's vertices. In this way we get a path containing infinitely many vertices □

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.

$$\mathcal{M}(\mathcal{G}) = \begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{ccccc} v_0 & v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

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 but none from v_j to v_i then the matrix is not symmetric because the i, j entry will be 1 but the j, i entry

will be 0. For 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 a natural number larger than zero.

3.11 DEFINITION For a graph $\mathcal{G}$, the adjacency matrix $\mathcal{M}(\mathcal{G})$ has that the i, j entry equals the number of edges from v_i to v_j.

3.12 LEMMA Let the matrix $\mathcal{M}(\mathcal{G})$ 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.

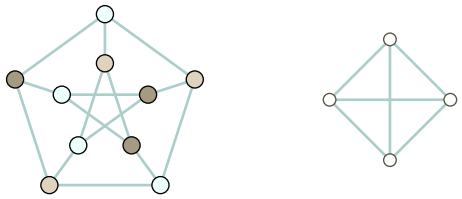
Proof Exercise 3.41. □

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 its vertices into k-many classes such that adjacent vertices are in different classes.

The name comes from the convention of showing the classes by drawing the vertices in different colors.

On the left is a graph that is 3-colored.

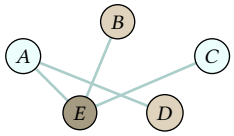


In contrast, the graph on the right has no 3-coloring. The four vertices are completely connected so if two got the same color then they would be adjacent.

3.14 EXAMPLE This table gives five committees. How many time slots must we use to so that no one has two meetings at once?

A	*B*	*C*	*D*	*E*
Armis	Crump	Burke	India	Burke
Jones	Edwards	Frank	Harris	Jones
Smith	Robinson	Ke	Smith	Robinson

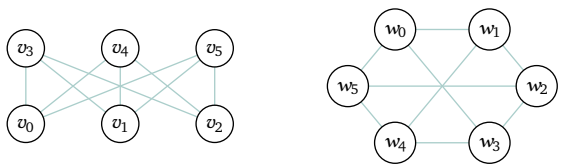
Model this with a graph by taking each vertex to be a committee and if committees are related by sharing a member then put an edge between them.



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 the even- and odd-numbered vertices. 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 define this correspondence between the vertices

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	$\{w_0, w_1\}$	$\{w_0, w_3\}$	$\{w_0, w_5\}$	$\{w_2, w_1\}$	$\{w_2, w_3\}$	$\{w_2, w_5\}$

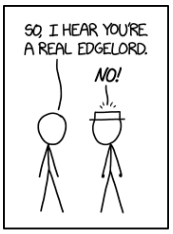
(Cont.) Edge on left	$\{v_2, v_3\}$	$\{v_2, v_4\}$	$\{v_2, v_5\}$
Edge on right	$\{w_2, w_1\}$	$\{w_2, w_3\}$	$\{w_2, w_5\}$

3.15 DEFINITION Two graphs $\mathcal{G}$ and $\hat{\mathcal{G}}$ are isomorphic if there is a one-to-one and onto map $f : \mathcal{N} \to \hat{\mathcal{N}}$ 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 useful such property to consider is 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 3.39 shows that if graphs are isomorphic then associated vertices have the same degree and thus graphs with different degree



Courtesy xkcd .com

sequences are not isomorphic. Also, if we have two isomorphic graphs then we can use the degrees of the vertices to help us construct an isomorphism, if there is one; examples are in the exercises. (Note, though, that there are graphs with the same degree sequence that are not isomorphic.)

Determining whether two given graphs are isomorphic is in general a hard problem. We could use brute force, checking every possible correspondence between the two sets of vertices, but that would be slow. We do not currently know whether there is a quick way. 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 they are unequal and if there is a $k \in \mathbb{N}$ with $m \cdot k = n$. Give the graph describing the divisor relation among positive natural numbers less than or equal to 12 (it is a digraph).

(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).

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 If a graph has many edges then from a visual design standpoint it can be confusing. Sometimes in a directed graph we can take advantage of a 'precedes' relation being transitive to draw it with the minimum number of edges that conveys all of the information. Suppose that in a Mathematics program students must take

Calculus II before Calculus III, and must take Calculus I before II. They must also take Calculus II before Linear Algebra, and to take Real Analysis they must have both Linear Algebra and Calculus III. Draw the digraph with a minimum number of edges.

3.19 Let a simple graph $\mathcal{G}$ have vertices $\{v_0, \ldots v_5\}$ 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.

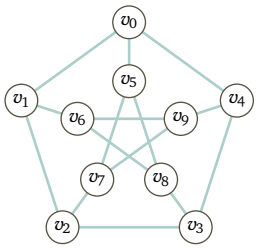
3.20 The complete graph on n vertices, K_n is the simple graph with all possible edges. (A) Draw K_4, K_3, K_2, and K_1. (B) Draw K_5. (C) How many edges does K_n have?

✓ 3.21 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.

3.22 This is the Petersen graph, often used for examples in Graph Theory.

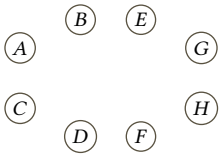


(A) List the vertices and edges. (B) Give two walks from v_0 to v_7. What is the length of each? (C) List both a closed walk and an open walk of length five, starting at v_4. (D) Give a cycle starting at v_5. (E) Is this graph connected?

3.23 A graph is a set of vertices and edges, not a drawing. So a single graph may be drawn with quite different pictures. Consider a graph $\mathcal{G}$ with the vertices $\mathcal{N} = \{A, \ldots H\}$ and these edges.

$$\mathcal{E} = \{AB, AC, AG, AH, BC, BD, BF, CD, CE, DE, DF, EF, EG, FH, GH\}$$

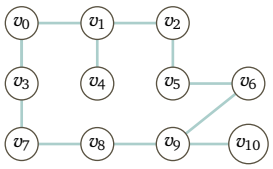
(A) Connect the dots below to get one drawing.



(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.

3.24 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 D. Species C cannot be with A, B, D, or E. Species D cannot be with B, C or F. Species E cannot be together with 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 be assigned 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.

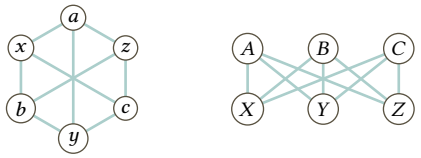
3.29 Give the array representation, like that in equation (∗), for the graphs of Example 3.4.

3.30 Draw a graph for this adjacency matrix.

$$\begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{cccc} v_0 & v_1 & v_2 & v_3 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

3.31 Show that every tree has a 2-coloring.

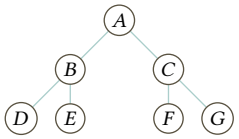
✓ 3.32 These two graphs are isomorphic.



(A) Define the function giving the correspondence.
(B) Verify that under that function the edges then also correspond.

3.33 For the two graphs in the prior exercise, give the degree sequences. Are they the same?

✓ 3.34 Consider this tree.

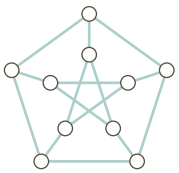


(A) Verify that $\langle BA, AC\rangle$ is a path from B to C.
(B) Why is $\langle BD, DB, BA, BC\rangle$ not also a path from B to C?
(C) Show that in any tree, for any two vertices there is a unique path from one to the other.

3.35 For the tree in the prior exercise, give the degree sequence.

✓ 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. (Also see **??**.)
(A) In a connected tree with a root, the rank of a vertex is the number of edges in the shortest path between that vertex and the root. 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 3.34's tree.
(B) A depth first traversal visits child vertices before sibling vertices. Give a depth first traversal of the same tree.

3.37 This is the Petersen graph.



(A) Show that it has no 2-coloring.
(B) Give a 3-coloring.

3.38 Consider building a simple graph by starting with 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 $n = 6$.

3.39 We can use degrees and degree sequences to to show that graphs are not isomorphic, or to help construct isomorphisms if they exist. (In this question graphs can have loops and multiple edges between vertices, but not directed edges or edges with weights.)

(A) Show that if two graphs are isomorphic then they have the same number of vertices. Thus graphs with different numbers of vertices are not isomorphic.

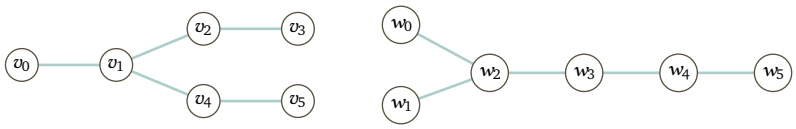
(B) Show that if two graphs are isomorphic then they have the same number of edges. Thus graphs with different numbers of edges are not isomorphic.

(C) Show that if two graphs are isomorphic and one has a vertex of degree k then so does the other. Thus two graphs where one has a degree k vertex and the other does not are not isomorphic.

(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 graphs with different degree sequences are not isomorphic.

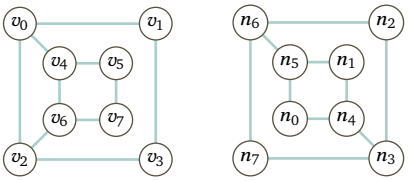
(E) Use the prior result to show that the two graphs of Example 3.4 are not isomorphic.

(F) 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.



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.

✓ 3.40 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 These two are the base and inductive steps for a proof of Lemma 3.12.

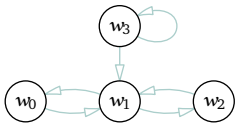
(A) An edge is a length-one walk. Show that in the product of the matrix with itself $\big(\mathcal{M}(\mathcal{G})\big)^2$ the entry i, j is the number of length-two walks.

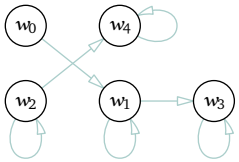
(B) Show that for $n > 2$, the i, j entry of the power $\big(\mathcal{M}(\mathcal{G})\big)^n$ equals the number of length n walks from v_i to v_j.

3.42 In a finite 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 graphs, starting with w_0.





EXTRA

III.A BNF

We shall introduce some grammar notation conveniences that are widely used. Together they are called Backus-Naur form, BNF.


John Backus 1924–2007 and Peter Naur 1928–2016

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 in the late 1950's by J Backus with contributions from P Naur 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 Section 2 is a minor typographical change. Metacharacters including '$\longrightarrow$' were at the time not typeable with a standard keyboard. In its place BNF uses '::='.[†]

BNF is both clear and concise. It can express the range of languages that we ordinarily want to express (context free grammars) 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 include some extensions for grouping and replication that are like what you typically see in the wild.[‡]

1.1 EXAMPLE This is a BNF grammar for real numbers with a finite decimal part. To the rules for ⟨*natural*⟩ from Example 2.6, add these.

[†] There are other typographical issues that arise with grammars. While many authors write nonterminals with diamond brackets, as we do, others use 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 single standard.

⟨*start*⟩ ::= `-`⟨*fraction*⟩ | `+`⟨*fraction*⟩ | ⟨*fraction*⟩
⟨*fraction*⟩ ::= ⟨*natural*⟩ | ⟨*natural*⟩`.`⟨*natural*⟩

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`

1.2 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. (It includes a metacharacter extension 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 extended BNF notations that that are more than simple character substitutions. One is shown above in the ⟨*partial-time*⟩ rule, which includes square brackets as metacharacters to denote that ⟨*time-secfrac*⟩ is optional. This is a very common construct: another example of it is in this syntax description for `if ... then ...` with an optional `else ...`

⟨*if-stmt*⟩ ::= `if` ⟨*boolean-expr*⟩ `then` ⟨*stmt-sequence*⟩
[`else` ⟨*stmt-sequence*⟩] ⟨*end if*⟩ `;`

To show repetition, BNF uses a Kleene star * for 'zero or more', as here.

⟨*identifier*⟩ ::= ⟨*letter*⟩ (⟨*letter*⟩ | ⟨*digit*⟩)*

To express the repetition construct 'one or more', BNF uses a plus sign, +.

1.3 Example This grammar for Python floating point numbers shows both square brackets and the plus sign.

⟨*floatnumber*⟩ ::= ⟨*pointfloat*⟩ | ⟨*exponentfloat*⟩
⟨*pointfloat*⟩ ::= [⟨*intpart*⟩] ⟨*fraction*⟩ | ⟨*intpart*⟩ .
⟨*exponentfloat*⟩ ::= (⟨*intpart*⟩ | ⟨*pointfloat*⟩) ⟨*exponent*⟩
⟨*intpart*⟩ ::= ⟨*digit*⟩+
⟨*fraction*⟩ ::= . ⟨*digit*⟩+
⟨*exponent*⟩ ::= (e | E) [+ | -] ⟨*digit*⟩+

In the ⟨*pointfloat*⟩ rule the first ⟨*intpart*⟩ is optional. And, an ⟨*intpart*⟩ 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

⟨*identifier*⟩ ::= ⟨*letter*⟩ (⟨*letter*⟩ | ⟨*digit*⟩)*

with this.

⟨*identifier*⟩ ::= ⟨*letter*⟩ | ⟨*letter*⟩ ⟨*atoms*⟩
⟨*atoms*⟩ ::= ⟨*letter*⟩ ⟨*atoms*⟩ | ⟨*digit*⟩ ⟨*atoms*⟩ | ε

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, using $\Sigma = \{\mathtt{a}, \ldots \mathtt{z}\}$.

✓ 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 1.3 uses some BNF convenience abbreviations. (A) Give a rule (or rules) equivalent to ⟨*pointfloat*⟩ but that doesn't use square brackets. (B) Similarly replace the repetition operator in ⟨*intpart*⟩'s rule, as well as the square brackets and repetition for ⟨*exponent*⟩.

✓ 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 represent natural numbers by writing these letters from left to right in descending order of value, so that XVI represents the number that in decimal notation is 16, while MDCCCCLVIII 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 the face of a watch or clock. In this notation 9 is IX, 40 is XL, 90 is XC, 400 is CD, and 900 is CM. Give an extended BNF 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`

There is also a rule that from ⟨*empty*⟩ produces a blank space `" "` (this rule is clearer to state in words than it is to show). 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`.

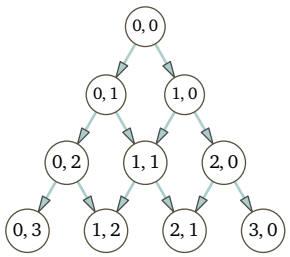
EXTRA

III.B Graph traversal

In a number of places in this book we describe traversing a tree or other graph. For example, when we described Cantor's correspondence enumerating the set $\mathbb{N} \times \mathbb{N}$, we drew this array.

$$\begin{array}{cccccc}
\vdots & \vdots & \vdots & \vdots & \\
\langle 0,3\rangle & \langle 1,3\rangle & \langle 2,3\rangle & \langle 3,3\rangle & \cdots \\
\langle 0,2\rangle & \langle 1,2\rangle & \langle 2,2\rangle & \langle 3,2\rangle & \cdots \\
\langle 0,1\rangle & \langle 1,1\rangle & \langle 2,1\rangle & \langle 3,1\rangle & \cdots \\
\langle 0,0\rangle & \langle 1,0\rangle & \langle 2,0\rangle & \langle 3,0\rangle & \cdots
\end{array}$$

We can make a graph by connecting each pair in the array to its neighbor above, and the one to the right. This shows the result with the lower left rotated to the top.



This graph isn't a tree because there are vertices that are connected by more than one path, for instance $\langle 0,0\rangle$ and $\langle 1,1\rangle$. Instead it is a directed acyclic graph, a DAG.

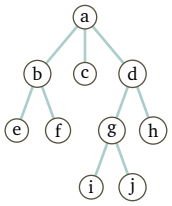
The enumeration starts like this.

Number	0	1	2	3	4	5	6	...
Pair	$\langle 0,0\rangle$	$\langle 0,1\rangle$	$\langle 1,0\rangle$	$\langle 0,2\rangle$	$\langle 1,1\rangle$	$\langle 2,0\rangle$	$\langle 0,3\rangle$	...

Cantor's enumeration is a breadth first traversal of the DAG.

Here we will show Racket code to traverse trees and DAG's (they are alike in that they have no cycles).

We will show two ways to do that. Below, on the left is a tree with ten nodes. On the right the table illustrates visiting the nodes in depth-first order, where we visit a node's children before going on to visit its siblings. It also illustrates breadth-first, 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 root has k edges).



Traversal	*Node order*
Depth first	*a-b-e-f-c-d-g-i-j-h*
Breadth first	*a-b-c-d-e-f-g-h-i-j*

The code first defines a `node`, essentially as an ordered pair.

```
(struct node (name children))
```

The Racket `struct` construct automatically defines a number of useful functions. One is a creator function `(node n s)`, which brings nodes into existence. It also makes the accessor functions `node-name` and `node-children` that we will see below.

We use that creator to make the next function.

```
(define (node-create name)
  (node name (mutable-set)))
```

Note that `children` is created as a set so that we can quickly access its members. This set is mutable because as we create the tree we will add children to that set, so we must be able to change it.

We use that routine to create new trees and DAG's.

```
(define (graph-create first-node-name)
  (node-create first-node-name))
```

The next routine inputs a node and adds a child to its set of children. (LISP-derived languages have a convention of using an exclamation mark for the names of procedures whose main role is not to return something but instead to cause side effects such as altering a data structure.)[†]

```
(define (node-add-child! parent child-name)
  (let ([n (node-create child-name)])
    (set-add! (node-children parent) n)
    n))
```

[†] The code here has a way to tie from parent to child but no direct way to tie back. So this tree is directed. We can of course write code for breadth-first traversals of undirected trees but for our purposes this suffices.

With those, we can construct the ten-node tree shown above.

```
(define (sample-tree-make)
  (let* ([t (graph-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))
```

And this returns the finite portion of Cantor's array shown earlier.

```
(define (cantor-DAG-make)
  (let* ([t (graph-create "0,0")]
         [nb (node-add-child! t "0,1")]
         [nc (node-add-child! t "1,0")]
         [nd (node-add-child! nb "0,2")]
         [ne (node-add-child! nb "1,1")]
         [v0 (set-add! (node-children nc) ne)]
         [nf (node-add-child! nb "2,0")]
         [ng (node-add-child! nd "0,3")]
         [nh (node-add-child! nd "1,2")]
         [v1 (set-add! (node-children ne) nh)]
         [ni (node-add-child! ne "2,1")]
         [v2 (set-add! (node-children nf) ni)]
         [nj (node-add-child! nf "3,0")]
         )
    t))
```

To demonstrate the traversal code, at each node we will just print out the name,

```
(define (show-node-name n r)
  (printf "~a~a\n" (string-pad r) (node-name n)))
```

indented by $2 \cdot k$ spaces where k is the rank.

```
(define (string-pad n)
  (apply string-append (build-list n (lambda (x) "  "))))
```

It is simpler so we first go through the routine to traverse depth-first.

```
(define (traverse-dfs node rank fcn #:maxrank [maximumrank MAXIMUM-RANK])
  (fcn node rank)
  (when (< rank maximumrank)
    (let ([children (node-children node)])
      (for ([child children])
        (traverse-dfs child (+ rank 1) fcn #:maxrank maximumrank))
      )))
```

The parameter `MAXIMUM-RANK` is a convenience for testing and development, to keep the routine from running away.

```
(define MAXIMUM-RANK 100)
```

The definition of `traverse-bfs` makes an optional keyword argument with it as the default.

This routine has a `fcn` input to describe what to do when we visit each node. Racket and other LISP-derived languages allow us to pass functions. We will pass `show-node-name` and it gets called on the `(fcn node rank)` line.

Here is the depth-first routine in action.

```
> (define t (sample-tree-make))
> (traverse-dfs t 0 show-node-name)
a
  b
    e
    f
  c
  d
    h
    g
      i
      j
```

Mathematical sets are unordered, so when we show elements it can be that the order out differs from the order in.

Now for breadth-first traversal. It comes in two functions, `traverse-bfs` and `traverse-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.

The strategy of `traverse-bfs-helper` is that at each level, when `rank` is k, the routine traverses all nodes at that rank by moving through the members of `level`. As it does so, it stores all of the children of those nodes in the list `next-level`.

```
(define (traverse-bfs node fcn #:maxrank [maxrank MAXIMUM-RANK])
  (traverse-bfs-helper (mutable-set node) 0 fcn #:maxrank maxrank))

(define (traverse-bfs-helper level rank fcn #:maxrank [maxrank MAXIMUM-RANK])
  (when (< rank maxrank)
    (let ([next-level (mutable-set)])
      (for ([node level])
        (fcn node rank)
        (for ([child-node (node-children node)])
          (set-add! next-level child-node)
          ))
      (when (not (set-empty? next-level))
        (traverse-bfs-helper next-level (+ 1 rank) fcn)))))
```

This strategy will not just take the routine around a cycle because both trees and DAGs are acyclic.

Here is the result of running the routine on the sample tree.

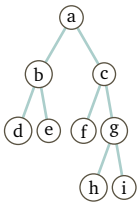
```
> (define t (sample-tree-make))
> (traverse-bfs t show-node-name)
a
  b
  c
  d
    h
    e
    f
    g
      i
      j
```

Finally, here is the result from the Cantor DAG.

```
> (define t (cantor-DAG-make))
> (traverse-bfs t show-node-name)
0,0
  0,1
  1,0
    2,0
    1,1
    0,2
      3,0
      2,1
      1,2
      0,3
```

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 depth-first traversal.
(C) Do breadth-first.


COMPUT ECCLESIASTIQUE
POUR L'AN DE GRACE
2009
CYCLE SOLAIRE
NOMBRE D'OR
INDICTION
LETTRE DOMINICALE
EPACTE

Chapter IV Automata

Our touchstone model of computation is the Turing machine. It has two components, the box and the tape. In this chapter we will focus on the box: we 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 that computes, 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.

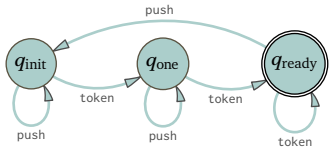
Definition We begin with some examples.

1.1 Example This power switch has two states, q_{off} and q_{on}, and its input alphabet has one token, `toggle`. (Its standard symbol is on the right.)

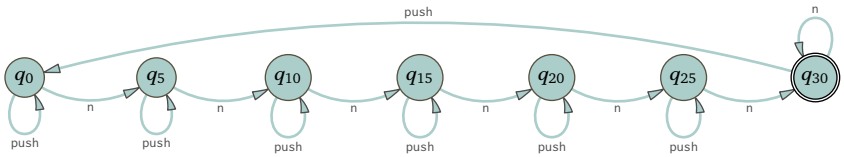
The state q_{on} is drawn with a double circle, denoting that it is a different kind of state than q_{off}. Finite State machines can't write to the tape so they need some other way to declare the computation's outcome. We say that q_{on} is an accepting state or final state. A computation accepts its 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 $\Sigma = \{\texttt{token}, \texttt{push}\}$. 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.

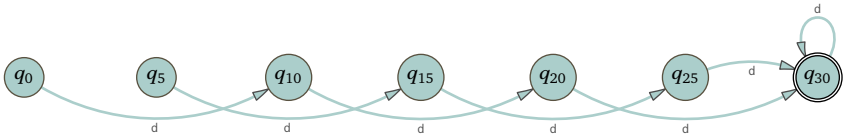
Image: The astronomical clock in Notre-Dame-de-Strasbourg Cathedral, for computing the date of Easter. Easter falls on the first Sunday after the first full moon on or after the nominal spring equinox of March 21. Calculation of this date was a great challenge for mechanisms of that time, 1843. [†] Studying the parts of the machine is natural but there is another motivation. A person could object to Turing's model by observing that there is a machine that iterates writing a character and then moving right, and thereby goes through unboundedly many configurations, while our understanding of the physical universe is that no device can do that. A rejoinder is that when we define a 'book' as 'a set if sheets' we don't worry whether physics limits the number of possible pages. Happily, we don't need to go further into this discussion this to justify our interest. These machines appear often in everyday computing, which is justification enough.



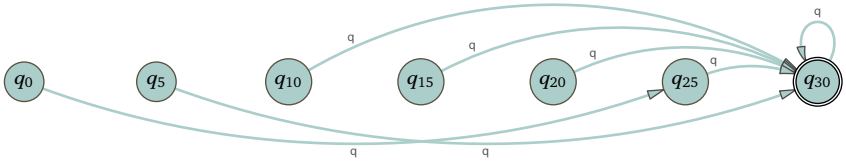
1.3 Example This vending machine dispenses items that cost 30 cents.[†] The picture is complex so we will show it in three layers. First are the arrows for nickels.



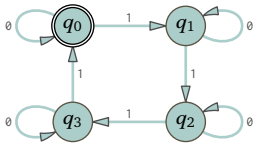
After receiving 30 cents and getting another nickel, this machine does something not very sensible: it stays in q_{30}. In practice a machine would have further states to keep track of overages so that it could give change but here we ignore that. Next comes the arrows for dimes



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.



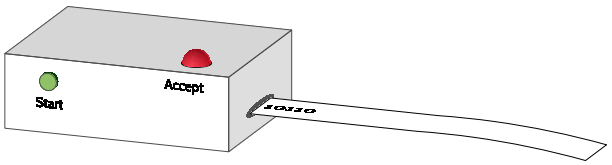
This machine accepts a bitstring if the number of 1's in its input is a multiple of four.

[†] US coins are: 1 cent coins not used here, nickels are 5 cents, dimes are 10 cents, and quarters are 25.

We defined Turing machines as sets of instructions. Instructions have the 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 $\Delta : Q \times \Sigma \to Q$.

A full description of the action of these machines comes after a few more examples. But basically, to work a machine, load a string input on the tape and press `Start`. At each step the machine consumes one tape token: it reads, acts on, and deletes it, and then moves so that the head points to the next tape cell.



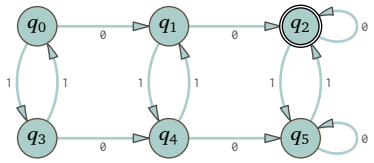
The picture shows a light labeled 'Accept'. When the machine stops, when the input string is fully consumed, if the current state is an accepting state then the light comes on. In this case we say that the machine accepts the input string, otherwise it rejects that string.

Here is a trace of the steps when we start Example 1.4's modulo 4 machine with the input string $\tau =$ `10110`. Since the ending state q_3 is not accepting the machine rejects τ.

Step	*Configuration*
0	`10110` q_0
1	`0110` q_1
2	`110` q_1
3	`10` q_2
4	`0` q_3
5	q_3

In contrast with the traces in the first chapter, here we hold the head still and move the tape. This emphasizes that Finite State machines consume one character per step. They stop once all the characters are gone so 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 mark accepting states with '+').



Δ	0	1
q_0	q_1	q_3
q_1	q_2	q_4
+ q_2	q_2	q_5
q_3	q_4	q_0
q_4	q_5	q_1
q_5	q_5	q_2

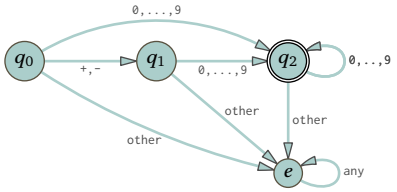
1.7 Remark We pause to briefly address the key to designing Finite State machines. Often people new to them put down a q_0, think of some input strings and then add states accounting for those inputs. This can give haphazard results.

Proceeding in this way is thinking of a state as about what happened to get there. Better is to think of states as about the future. The prior example brings this out: articulating the role of state q_1 gives something like, "waiting for a 0" or possibly "waiting for at least one 0". Similarly q_5 is "waiting for a 1." Another example is that state q_4 is looking for a 0 followed by a 1.

Finite State machine descriptions may 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 -7 and +37 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.

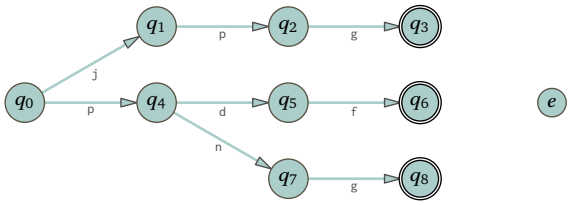


Δ	+, -	0, ... 9	–other–
q_0	q_1	q_2	e
q_1	e	q_2	e
+ q_2	e	q_2	e
e	e	e	e

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 accepting state.

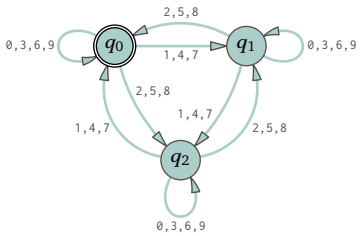
[†] We often use the characters a, b, c, etc., because something like 'b^2' is clearer than something like '1^2'.



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 while pictures are better for simple machines, past some point of complexity, 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))]))
```

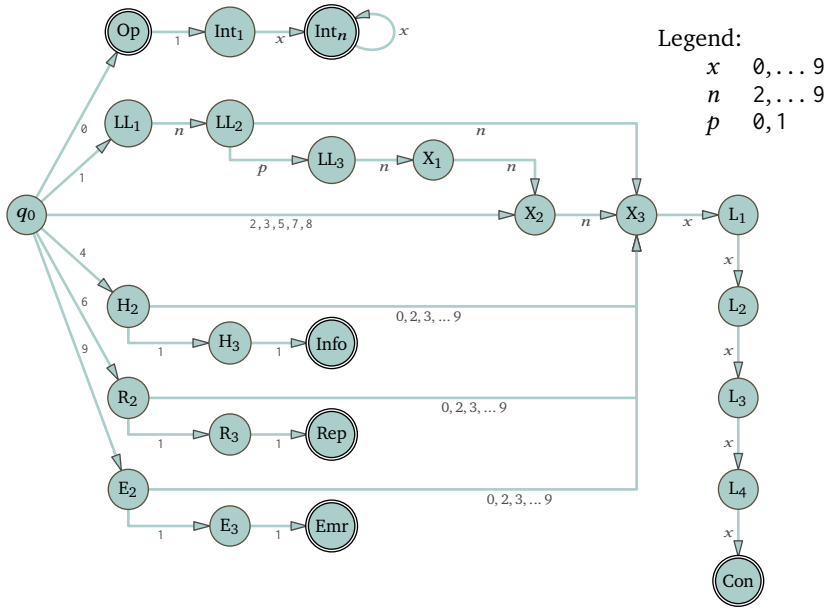
(In Racket, a character such as '`0`' is denoted `#\0`. The routine `memv` decides if the character `ch` is in the list, that is, it is a boolean function.) All that's left is to supply a calling function

```
(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 In the 1940's, phone call connections were handled by simple devices for local calls but required operator intervention for long distance. That changed with the adoption of the Finite State machine here, which allowed users to directly dial long distance in North America. Consider dialing `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's device 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 complete description of the action of those machines. We now do the same for Finite State machines. A configuration of a Finite State machine is a pair $\mathcal{C} = \langle q, \tau \rangle$ where q is a state, $q \in Q$, and τ is a (possibly empty) string, $\tau \in \Sigma^*$. A machine starts in an initial configuration $\mathcal{C}_0 = \langle q_0, \tau_0 \rangle$, so that τ_0 is the input and q_0 is the initial state.

A Finite State machine acts by making a sequence of transitions between configurations. A machine's configuration after the s-th transition (for $s \in \mathbb{N}^+$) is its configuration at step s. Fitting with $\mathcal{C}_0$, we write $\mathcal{C}_1$, $\mathcal{C}_2$, and so forth.

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, $\tau_s = c^\frown \tau_{s+1}$. Then 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. No transitions follow. Every Finite State machine eventually reaches a halting configuration because at each transition the tape string loses a character. A computation for a Finite State machine is a sequence $\mathcal{C}_0 \vdash \mathcal{C}_1 \vdash \cdots \mathcal{C}_h = \langle q, \varepsilon \rangle$, which we abbreviate $\mathcal{C}_0 \vdash^* \mathcal{C}_h$.[‡] If $\mathcal{C}_h$'s state is a final state, $q \in F$, then the machine accepts the input string τ_0. Otherwise it rejects τ_0.

1.13 Example The multiple of three machine of Example 1.10 with input `8013` gives the computation $\langle q_0, \texttt{8013} \rangle \vdash \langle q_2, 013 \rangle \vdash \langle q_2, 13 \rangle \vdash \langle q_0, 3 \rangle \vdash \langle q_0, \varepsilon \rangle$. 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 recognized, or decided, by the machine,[#] or just simply the language of that machine, $\mathcal{L}(\mathcal{M})$.

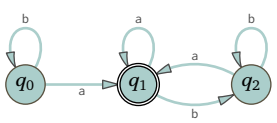
1.15 Example The language of the power switch machine from Example 1.1 is the set of strings consisting of the token `toggle` given an even number of times, $\mathcal{L} = \{ \texttt{toggle}^{2k} \mid k \in \mathbb{N} \}$. The language of the turnstile machine, Example 1.2 is the set of strings from the alphabet $\Sigma = \{ \texttt{token}, \texttt{push} \}$ of the form

$$(\texttt{push})^{k_0}\ \texttt{token}\ (\texttt{push})^{k_1}\ (\texttt{token})^{k_2}\ \big(\ (\texttt{push})^{k_3}\ \texttt{token}\ (\texttt{push})^{k_4}\ (\texttt{token})^{k_5}\ \big)^n$$

for $k_2, k_3, k_5 \in \mathbb{N}^+$ and $n, k_0, k_1, k_4 \in \mathbb{N}$.

1.16 Definition For any Finite State machine, the extended transition function $\hat{\Delta} \colon \Sigma^* \to Q$ 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.



Δ	a	b
q_0	q_1	q_0
+ q_1	q_1	q_2
q_2	q_1	q_2

Its extended transition function $\hat{\Delta}$ extends Δ in that it repeats the first row of Δ's

[†] As earlier, read $\vdash$ aloud as "yields." [‡] Read $\vdash^*$ as "yields eventually." [#] Finite State machines must halt and so there is no notion like computably enumerable. Thus the languages that such a machines can decide is the same as the languages that it can recognize (in contrast with the case for Turing machines, as defined on page 11). For these machines, 'recognized' is the more common term.

table.

$$\hat{\Delta}(\mathtt{a}) = q_1 \qquad \hat{\Delta}(\mathtt{b}) = q_0$$

(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.

$$\hat{\Delta}(\mathtt{aa}) = q_1 \quad \hat{\Delta}(\mathtt{ab}) = q_2 \quad \hat{\Delta}(\mathtt{ba}) = q_1 \quad \hat{\Delta}(\mathtt{bb}) = q_0$$

Observe that a string σ is accepted by the machine if and only if $\hat{\Delta}(\sigma)$ is an accepting state. For instance, the string aa is in the language of this machine as $\hat{\Delta}(\mathtt{aa}) = q_1$, which is accepting.

We can give a constructive definition of $\hat{\Delta}$. Fix a Finite State machine $\mathcal{M}$ with transition function $\Delta\colon Q \times \Sigma \to Q$. Begin with $\hat{\Delta}(\varepsilon) = \{q_0\}$. Then for $\tau \in \Sigma^*$ and $t \in \Sigma$, define $\hat{\Delta}(\tau^\frown t)$ to be $\Delta(q, t)$, where $\hat{\Delta}(\tau) = q$.

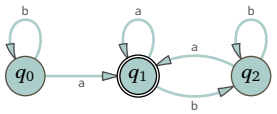
One way in which $\hat{\Delta}$ is handy is that it gives us a nice alternative definition of when an input string is accepted by a machine: τ_0 is accepted if $\hat{\Delta}(\tau_0) \in F$.

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, $\hat{\Delta}$ has one and only one output state for all input strings.

IV.1 Exercises

For the exercises that give a language description, a useful practice 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) bbaabbaa. Does the machine accept the string?



1.19 True or false: because a Finite State machine is finite, its language must be finite.

1.20 Your classmate says, "I have a language $\mathcal{L}$ that recognizes the empty string ε." Explain to them the mistake.

1.21 Rebut "no Finite State machine can recognize the language $\{\mathtt{a}^n\mathtt{b} \mid n \in \mathbb{N}\}$ because n is infinite."

✓ 1.22 How many transitions does an input string of length n cause a Finite State machine to undergo? n many? $n + 1$? $n - 1$? 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) $\mathcal{L} = \{\alpha \in \{\mathtt{a}, \mathtt{b}\}^* \mid \alpha = \mathtt{a}^n\mathtt{b}\mathtt{a}^n \text{ for } n \in \mathbb{N}\}$

(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) $\{a^n b a^{n+2} \in \{a, b\}^* \mid n \in \mathbb{N}\}$
(E) $\{\gamma \in \{a, b\}^* \mid \gamma \text{ has the form } \gamma = \alpha^\frown \alpha \text{ for } \alpha \in \{a, b\}^*\}$

✓ 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 accept the empty string ε? (C) What is the shortest string that each accepts? (D) Is the language of accepted strings infinite?

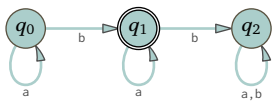
1.25 As in Example 1.13, give the computation for the multiple of three machine with the initial string 2332.

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, name five strings in the language and five that are not (if there are not five, name as many as there are). Then 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) $\mathcal{L}_1 = \{\sigma \in \Sigma^* \mid \sigma \text{ has at least one a and at least one b}\}$
(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 the states.

✓ 1.31 For each language 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) $\{\sigma \in \Sigma^* \mid \sigma \text{ has two or fewer a's}\}$
(D) $\{\sigma \in \Sigma^* \mid \sigma \text{ has at least one a followed by at least one b}\}$

✓ 1.32 Give a Finite State machine over $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ 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 $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.
(A) $\mathcal{L} = \{\sigma \in \{\mathtt{a}, \mathtt{b}\}^* \mid \sigma \text{ ends in } \mathtt{aa}\}$
(B) $\{\sigma \in \{\mathtt{a}, \mathtt{b}\}^* \mid \sigma = \varepsilon\}$
(C) $\{\sigma \in \{\mathtt{a}, \mathtt{b}\}^* \mid \sigma = \mathtt{a}^3\mathtt{b} \text{ or } \sigma = \mathtt{b}\mathtt{a}^3\}$
(D) $\{\sigma \in \{\mathtt{a}, \mathtt{b}\}^* \mid \sigma = \mathtt{a}^n \text{ or } \sigma = \mathtt{b}^n \text{ for } n \in \mathbb{N}\}$

1.34 Produce a Finite State machine over the alphabet $\Sigma = \{\mathtt{A}, \ldots \mathtt{Z}, \mathtt{0}, \ldots \mathtt{9}\}$ 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 that begin with the two-character string /#, end with the two-character string #/, and have no #/ substrings in the middle. Give a Finite State machine to recognize that language.

✓ 1.38 Produce a Finite State machine that recognizes each.
(A) $\{\sigma \in \{\mathtt{0}, \ldots \mathtt{9}\}^* \mid \sigma \text{ has either no 0's or no 2's}\}$
(B) $\{\sigma \in \{\mathtt{0}, \ldots \mathtt{9}\}^* \mid \sigma \text{ is the decimal representation of a multiple of 5}\}$

✓ 1.39 Give a Finite State machine over the alphabet $\Sigma = \{\mathtt{A}, \ldots \mathtt{Z}\}$ 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.

$\langle real \rangle \;\to\; \langle posreal \rangle \mid \mathtt{+}\ \langle posreal \rangle \mid \mathtt{-}\ \langle posreal \rangle$
$\langle posreal \rangle \;\to\; \langle natural \rangle \mid \langle natural \rangle\ \mathtt{.} \mid \langle natural \rangle\ \mathtt{.}\ \langle natural \rangle$
$\langle natural \rangle \;\to\; \langle digit \rangle \mid \langle digit \rangle\ \langle natural \rangle$
$\langle digit \rangle \;\to\; \mathtt{0} \mid \ldots \mathtt{9}$

(A) Give five strings of terminals that are in its language and five that are not. (B) Does the language contain the string .12? (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) $\{\sigma \in \mathbb{B}^* \mid \text{every 1 in } \sigma \text{ has a 0 just before it and just after}\}$
(B) $\{\sigma \in \mathbb{B}^* \mid \sigma \text{ represents in binary a number divisible by 4}\}$
(C) $\{\sigma \in \{\mathtt{0}, \ldots \mathtt{9}\}^* \mid \sigma \text{ represents in decimal an even number}\}$
(D) $\{\sigma \in \{\mathtt{0}, \ldots \mathtt{9}\}^* \mid \sigma \text{ represents in decimal a multiple of 100}\}$

1.42 Consider $\{\sigma \in \{0, \ldots 9\}^* \mid \sigma$ represents in decimal a multiple of $4\}$. Briefly describe a Finite State machine. You need not give the full graph or table.

1.43 Following Definition 1.16 is a constructive definition of the extended transition function. We will apply that 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 all length two 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 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 one of the oldest collection of mathematical problems. It is by Alcuin of York (735–804), royal advisor to Charlemagne and head of the Frankish court school, at Aachen. One problem, *Propositio de lupo et capra etfasciculo 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.* Readers of the 700's would have known that 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.

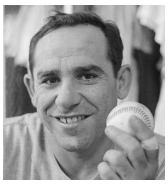
1.47 Fix an alphabet with at least two members. We will show that there are languages over that alphabet not recognized by any Finite State machine. (A) Show that the number of Finite State machines with that alphabet is infinite. (B) Show that it is countable. (C) Show that the number of languages over 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 we lay out an initial tape and push `Start` then the machine just walks through a fixed succession of step/next step calculations. We now consider machines that are nondeterministic, ones for which there may be configurations where there is more than one next state, or configurations where there is just one, or even configurations without any next state at all.

Motivation Imagine a grammar with some rules and a start symbol. We get a string and are asked to find a derivation of it. The challenge is that we sometimes don't know which rules the derivation should follow. For instance, if we have S $\to$ BaS | AbA then from S we can do two different things: which will work?

In the Grammar section's exercises we expected that an intelligent person would have the insight to guess the right way. If instead we were writing a program then we might have it try every case — we might do a breadth-first traversal of the directed acyclic graph of all derivations — until the program finds 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.

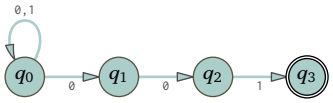
The prototypical example for this strategy is the celebrated Traveling Salesman problem, that of finding the shortest circuit visiting every city in a list. For instance, suppose that we want to know if there is a trip that visits each state capital in the US lower forty eight states and returns back to where it began, in less than 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 after Montpelier goes next to 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 finds it then we consider the overall search a success.

That computation is nondeterministic in that while it is happening the machine is simultaneously in many different states. Restated, the computation happens on an unboundedly-parallel machine, where whenever we need an additional computing agent, another CPU plus tape, one is available.[†]

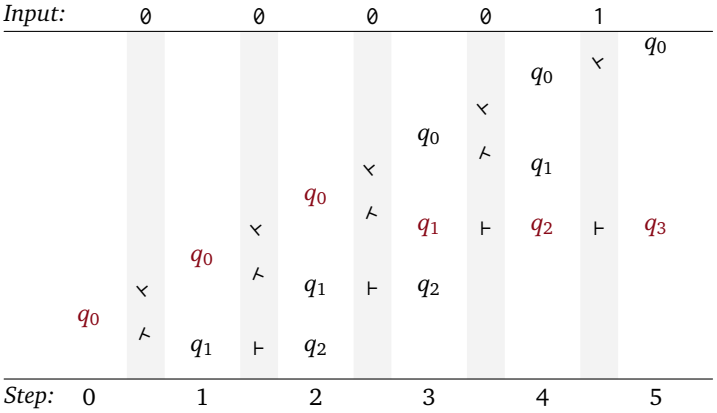
We will have two ways to think about nondeterminism, two mental models.[‡] The first is the one introduced above: when a machine is presented with multiple possible next states then it forks, so that it is in all of them simultaneously. The next example illustrates.

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.

[†] 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.



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 is at least one child process that finds it. The same happens here: there is a maximal path[†] of the computation tree that accepts the input string. There are also maximal paths that are not successful. The one at the bottom dies after step 2 because when the present state is q_2 and the input is `0` then this machine passes to no-state.[‡] The maximal path at the top does not die early but it also does not accept the input. However, we don't care if there are dozens of unsuccessful maximal paths, we only care that there is at least one success. Consequently, we will define that a nondeterministic machine accepts an input if the computation tree has at least one maximal path that accepts.

The machine in the above example accepts any string that ends in two `0`'s and a 1. As it is consuming the input $\sigma =$ `00001`, the problem that the machine faces is: when should it 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 what to do.

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 one to guess.[#]

[†] It is tempting to call this a 'branch' but that term has a technical meaning of maximal subtree, so that for instance a branch containing the state q_0 in the column for Step 1 must contain all of the maximal paths above the bottom one. [‡] No-state cannot be an accepting state, since it isn't a state at all.
[#] The name `amb` abbreviates 'ambiguous function'.

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.

When we talk about this way of expressing the second mental model our convention is to call the furnisher a demon, because they somehow know answers that cannot otherwise be found but also because 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 maximal path of the computation tree that a deterministic machine, if told what path to take, could verify.

Flauros, Duke of Hell

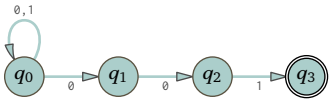
Below we shall describe nondeterminism using both paradigms: as a machine being in multiple states at once, and also as a machine guessing (or being told and verifying). In this chapter we will do that for Finite State machines and in the fifth chapter we will return to it for Turing machines.

Definition A nondeterministic Finite State machine's next-state function does not output single states, it outputs sets of states, members of the power set $\mathscr{P}(Q)$.

2.3 Definition A nondeterministic Finite State machine $\mathcal{M} = \langle Q, q_0, F, \Sigma, \Delta \rangle$ consists of a finite set of states Q, one of which is the start state q_0, a subset $F \subseteq Q$ of accepting states or final states, a finite input alphabet set Σ, and a next-state function $\Delta : Q \times \Sigma \to \mathscr{P}(Q)$.

We will use these machines in three ways. They are useful in practice; many tasks are more easily solved with nondeterministic machines than with deterministic ones. We will also use them to prove Kleene's Theorem, Theorem 3.11. And, they give us an initial encounter with nondeterminism, which is a critical concept for our final chapter.

2.4 Example This is Example 2.1's nondeterministic Finite State machine, along with its transition function. That function does not output states, it outputs sets of states.



	Δ	0	1
	q_0	$\{q_0, q_1\}$	$\{q_0\}$
	q_1	$\{q_2\}$	$\{\}$
	q_2	$\{\}$	$\{q_3\}$
+	q_3	$\{\}$	$\{\}$

The imagery in informal terms such as "guess" and "demon" helps introduce the ideas but may also give an impression that those ideas are fuzzy. So when we next step through the description of the action of these machines, note that it is precise.

2.5 Remark When we described the action of deterministic Finite State machines on page 184, we laid out how to construct the sequence of configurations by transitioning from each to its successor until the tape is empty. But for nondeterministic

machines there needn't be one and only one sequence. So the description is more complex.

As with deterministic machines, a configuration is a pair $\mathcal{C} = \langle q, \tau \rangle$ where $q \in Q$ and $\tau \in \Sigma^*$. A machine starts in an initial configuration $\mathcal{C}_0 = \langle q_0, \tau_0 \rangle$, so that τ_0 is the input. A configuration with an empty tape, $\mathcal{C}_h = \langle q_h, \varepsilon \rangle$, is a halting configuration.

We next describe when two configurations are related by a transition, denoted $\vdash$. In the first configuration $\mathcal{C}_s = \langle q, \tau_s \rangle$, let the string τ_s be not empty. Then where $\hat{\mathcal{C}} = \langle \hat{q}, \hat{\tau} \rangle$, we write $\mathcal{C}_s \vdash \hat{\mathcal{C}}$ or say that $\hat{\mathcal{C}}$ succeeds $\mathcal{C}_s$ if two conditions hold. First, it must be that $\hat{\tau}$ comes from removing τ_s's leading character, so that $\tau_s = c \frown \hat{\tau}$ for $c \in \Sigma$. Second, using that leading character c, it must also be that $\hat{q}$ is a member of the set $\Delta(q, c)$.

So far the definitions apply to any Finite State machine, deterministic or not. But for the machines in this section a configuration may have more than one configuration as its $\vdash$ child, or only one, or none at all. Consider all of the sequences of configurations that are related by $\vdash$ transitions and that start with $\mathcal{C}_0$. Put together, these form a directed graph that is a tree rooted at $\mathcal{C}_0$. This is the computation tree or just computation.

In that tree a maximal path is a sequence of '$\vdash$' edges that is not contained in any longer path. (In this machine type a halting configuration has no configuration succeeding it, so if a path has one then it comes last in that path. But maximal paths can also terminate in a final configuration that is not a halting configuration.) If the computation tree contains at least one maximal path ending in a halting configuration, $\mathcal{C}_0 = \langle q_0, \tau_0 \rangle \vdash \mathcal{C}_1 \vdash \cdots \mathcal{C}_h = \langle q_h, \varepsilon \rangle$, where the last state is accepting, $q_h \in F$, then $\mathcal{M}$ accepts the input τ_0. Otherwise, it rejects τ_0.

2.6 Example Example 2.1 shows a number of maximal paths. Since this one

$$\langle q_0, \texttt{00001} \rangle \vdash \langle q_0, \texttt{0001} \rangle \vdash \langle q_0, \texttt{001} \rangle \vdash \langle q_1, \texttt{01} \rangle \vdash \langle q_2, \texttt{1} \rangle \vdash \langle q_3, \varepsilon \rangle$$

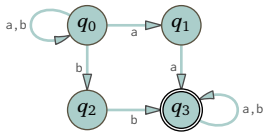
ends in a halting configuration whose state is accepting, the machine accepts `00001`.

As with deterministic Finite State machines, at each step the machine consumes one character. There is no Halting problem for these machines.

2.7 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 by that machine.

We can also adapt the definition of the extended transition function so that it outputs sets, $\hat{\Delta} \colon \Sigma^* \to \mathscr{P}(Q)$. Fix a nondeterministic $\mathcal{M}$ with transition function $\Delta \colon Q \times \Sigma \to Q$. Start with $\hat{\Delta}(\varepsilon) = \{ q_0 \}$. For $\tau \in \Sigma^*$, where $\hat{\Delta}(\tau) = \{ q_{i_0}, q_{i_1}, \ldots q_{i_k} \}$ and $t \in \Sigma$, define $\hat{\Delta}(\tau \frown t)$ to be $\Delta(q_{i_0}, t) \cup \Delta(q_{i_1}, t) \cup \cdots \cup \Delta(q_{i_k}, t)$. Then the machine accepts $\sigma \in \Sigma^*$ if and only if any element of $\hat{\Delta}(\sigma)$ is a final state.

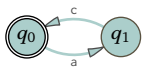
2.8 Example The language recognized by this nondeterministic machine



is the set of strings containing the substring aa or bb. For instance, the machine accepts abaaba because there is a sequence of transitions ending in an accepting state.

$$\langle q_0, \text{abaaba}\rangle \vdash \langle q_0, \text{baaba}\rangle \vdash \langle q_0, \text{aaba}\rangle \vdash \langle q_1, \text{aba}\rangle \vdash \langle q_3, \text{ba}\rangle \vdash \langle q_3, \text{a}\rangle \vdash \langle q_3, \varepsilon\rangle$$

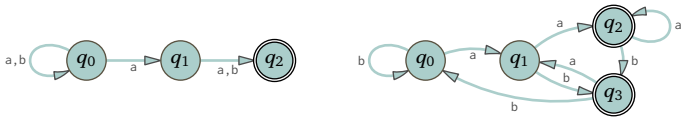
2.9 Example With $\Sigma = \{\,\text{a}, \text{b}, \text{c}\,\}$, this nondeterministic machine



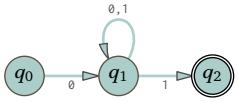
recognizes the language $\{\,(\text{ac})^n \;\big|\; n \in \mathbb{N}\,\} = \{\,\varepsilon, \text{ac}, \text{acac}, \ldots\,\}$. The symbol b isn't attached to any arrow so it won't play a part in any accepting string.

Often a nondeterministic Finite State machines is easier to write than a deterministic machine that does the same job.

2.10 Example Both of these machines accept any string whose next to last character is a. The nondeterministic one on the left is simpler than the deterministic one.



2.11 Example This machine accepts $\{\,\sigma \in \mathbb{B}^* \;\big|\; \sigma = 0^\frown \tau^\frown 1 \text{ where } \tau \in \mathbb{B}^*\,\}$.



2.12 Example This is a remote control listener that waits to hear the signal 0101110. That is, it recognizes the language $\{\,\sigma^\frown 0101110 \;\big|\; \sigma \in \mathbb{B}^*\,\}$.

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 $\Delta\colon Q \times \Sigma \to \mathscr{P}(Q)$ the transition function's signature is $\Delta\colon Q \times (\Sigma \cup \{\,\varepsilon\,\}) \to \mathscr{P}(Q)$.[†] The associated behavior is that the machine can transition spontaneously, without consuming any input. We start with a number of examples.

[†] For this purpose the 'ε' is a character, not a representation of the empty string. Assume that it is not an element of Σ.

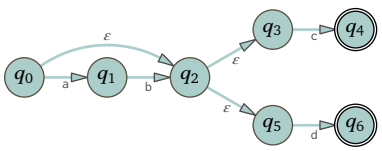
2.13 Example This machine recognizes valid integer representations. The ε on the first arrow means that the machine can jump from q_0 to q_1 without reading the tape.

For instance, with input 123 the machine can begin as below by following the ε transition to state q_1 without reading and deleting the leading 1. It next reads that 1 and transitions to q_2, and then stays there while processing the 2 and 3. This maximal path of the machine's computation tree accepting its input and so 123 is in the machine's language.

$$\langle q_0, 123\rangle \vdash \langle q_1, 123\rangle \vdash \langle q_2, 23\rangle \vdash \langle q_2, 3\rangle \vdash \langle q_2, \varepsilon\rangle$$

The practical effect of the ε is that this machine can accept strings that do not start with a + or − sign.

2.14 Example This machine has a number of ε transitions.



Here it accepts abc by following the ε transition between q_2 and q_3.

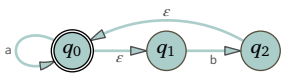
$$\langle q_0, \mathsf{abc}\rangle \vdash \langle q_1, \mathsf{bc}\rangle \vdash \langle q_2, \mathsf{c}\rangle \vdash \langle q_3, \mathsf{c}\rangle \vdash \langle q_4, \varepsilon\rangle$$

A machine may also, in a single step, follow two or more ε transitions in succession. Here, it accepts d by transitioning from q_0 to q_5 without consuming any input.

$$\langle q_0, \mathsf{d}\rangle \vdash \langle q_5, \mathsf{d}\rangle \vdash \langle q_6, \varepsilon\rangle$$

The language is $\mathcal{L} = \{\,\mathsf{abc}, \mathsf{abd}, \mathsf{c}, \mathsf{d}\,\}$.

2.15 Example Below is a machine that is nondeterministic and with ε moves, along with its computation tree on input aab. The ε moves are inside the white stripes.



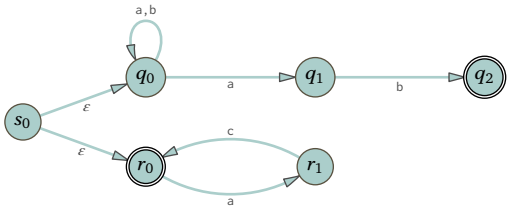
Input:	a		a		b	
						q_1
						⊥
						q_0
						⊥
q_1		q_1		q_1	⊢	q_2
⊥		⊥		⊥		
q_0	⊢	q_0	⊢	q_0		
Step: 0		1		2		3

2.16 Animation: Computation tree for a nondeterministic machine with ε moves.

At each step, the machine is in all of the states that are inside of the step's stripe. For instance, at step 0 the machine is in both q_0 and q_1. After exhausting the tape, at step 3 it is in both q_0 and q_2 and because q_0 is an accepting state, it accepts the input aab.

The ε transitions simplify building Finite State machines.

2.17 Example An ε transition can put two machines together with a parallel connection. Here is a machine whose states are named with q's combined in parallel with one whose states are named with r's.

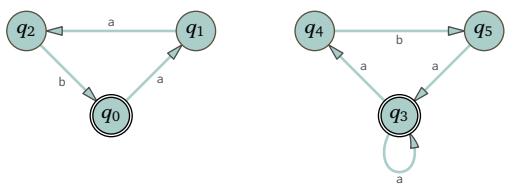


The top machine's language is $\{\sigma \in \{\mathsf{a},\mathsf{b}\}^* \mid \sigma \text{ ends in } \mathsf{ab}\}$ and the bottom's is $\{\sigma \in \{\mathsf{a},\mathsf{c}\}^* \mid \sigma = (\mathsf{ac})^n \text{ for some } n \in \mathbb{N}\}$. This is the language for the entire machine.

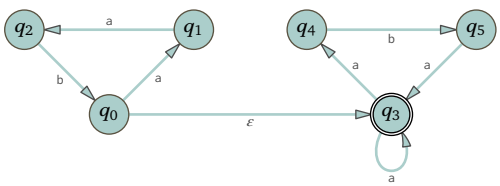
$$\mathcal{L} = \{\sigma \in \Sigma^* \mid \text{either } \sigma \text{ ends in ab or } \sigma = (\mathsf{ac})^n \text{ for } n \in \mathbb{N}\}$$

We can take the alphabet for the entire machine to be the union, $\Sigma = \{\mathsf{a},\mathsf{b},\mathsf{c}\}$.

2.18 Example An ε transition can also connect machines serially. The machine on the left below recognizes $\mathcal{L}_0 = \{(\mathsf{aab})^i \mid i \in \mathbb{N}\}$. The one on the right recognizes $\mathcal{L}_1 = \{\sigma_0 \frown \cdots \sigma_{j-1} \mid j \in \mathbb{N} \text{ and } \sigma_k = \mathsf{a} \text{ or } \sigma_k = \mathsf{aba} \text{ for } 0 \le k \le j-1\}$.

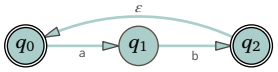


If we insert an ε bridge to the right side's initial state from each of the left side's final states (here there is only one such state), and de-finalize those states on the left,



then the combined machine accepts strings in the concatenation of those languages, $\mathcal{L}(\mathcal{M}) = \mathcal{L}_0 \frown \mathcal{L}_1$. For example, it accepts aabaababa, and aabaa, as well as abaa.

2.19 EXAMPLE We can also use ε transitions to get the Kleene star of a language. Without the ε edge this machine's language is $\mathcal{L} = \{\varepsilon, \mathsf{ab}\}$,

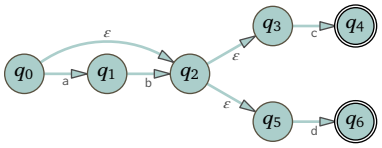


but with that edge the language is $\mathcal{L}^* = \{(\mathsf{ab})^n \mid n \in \mathbb{N}\}$.

We next describe the action of these machines. For that we need a function. The ε closure $\hat{E}\colon Q \to \mathscr{P}(Q)$ inputs a state q and returns the set of states that are reachable from q in some number of ε moves.

We will build it by iteratively, by constructing a function $E\colon Q \times \mathbb{N} \to \mathscr{P}(Q)$ such that $E(q,m)$ is the set of states reachable from q in at most m-many ε moves. To define this function, for $m = 0$ take $E(q,m) = \{q\}$. For $m > 0$, where $E(q,m) = \{q_{m_0}, \ldots q_{m_k}\}$, take $E(q,m+1) = E(q,m) \cup \Delta(q_{m_0},\varepsilon) \cup \cdots \Delta(q_{m_k},\varepsilon)$. The resulting sets are nested, $E(q,0) = \{q\} \subseteq E(q,1) \subseteq \cdots$. There are only finitely many states so for any initial q there must be an $m_q \in \mathbb{N}$ where the sequence of sets stops growing, $E(q,m_q) = E(q,m_q+1) = \cdots$. Define the ε closure as the limit, $\hat{E}(q) = E(q,m_q)$.

2.20 EXAMPLE Recall the machine from Example 2.14.



This table finds the epsilon closure for each state.

$E(q,m)$	$m=0$	1	2	3	$\hat{E}(q)$
q_0	$\{q_0\}$	$\{q_0,q_2\}$	$\{q_0,q_2,q_3,q_5\}$	$\{q_0,q_2,q_3,q_5\}$	$\{q_0,q_2,q_3,q_5\}$
q_1	$\{q_1\}$	$\{q_1\}$	$\{q_1\}$	$\{q_1\}$	$\{q_1\}$
q_2	$\{q_2\}$	$\{q_2,q_3,q_5\}$	$\{q_2,q_3,q_5\}$	$\{q_2,q_3,q_5\}$	$\{q_2,q_3,q_5\}$
q_3	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$
q_4	$\{q_4\}$	$\{q_4\}$	$\{q_4\}$	$\{q_4\}$	$\{q_4\}$
q_5	$\{q_5\}$	$\{q_5\}$	$\{q_5\}$	$\{q_5\}$	$\{q_5\}$

We can now describe the action. As earlier, a configuration is a pair $\mathcal{C} = \langle q, \tau\rangle \in Q \times \Sigma^*$. A machine starts in an initial configuration $\mathcal{C}_0 = \langle q_0, \tau_0\rangle$, with input τ_0.

Fix $\mathcal{C}_s = \langle q, \tau_s\rangle$ to describe under what circumstances $\mathcal{C}_s \vdash \hat{\mathcal{C}}$ for $\hat{\mathcal{C}} = \langle \hat{q}, \hat{\tau}\rangle$. There are two possibilities. The first is the same as earlier: τ is not the empty string and $\hat{\tau}$ results from popping τ's leading character, $\tau = c^\frown \hat{\tau}$, and also $\hat{q}$ is a member of the set $\Delta(q,c)$. The second possibility is new for these new machines: $\hat{\tau} = \tau$, so the machine doesn't consume any input, and also $\hat{q}$ is a member of the ε closure $\hat{E}(q)$.

Consider all of the sequences of configurations that are related by $\vdash$ transitions and that start with $\mathcal{C}_0$. Put together, these form a directed graph that is a tree rooted at $\mathcal{C}_0$. This is the computation tree or just computation.

In that tree a maximal path is a sequence that is not properly contained in any other path. (In this machine type there can be cycles of ε moves and so paths can be infinite.) If there exists at least one maximal path containing a halting configuration, $C_h = \langle q_h, \varepsilon \rangle$, and its state is accepting, $q_h \in F$, then $\mathcal{M}$ accepts the input τ_0. Otherwise $\mathcal{M}$ rejects τ_0.

We finish by defining the extended transition function $\hat{\Delta}\colon Q \times (\Sigma \cup \{\varepsilon\})^* \to Q$ to suit this new machine type. Fix $q \in Q$. Begin with $\hat{\Delta}(q, \varepsilon) = \hat{E}(q)$. For the iteration, suppose that $\hat{\Delta}(q, \tau) = \{q_{i_0}, q_{i_1}, \ldots q_{i_k}\}$ for $\tau \in \Sigma^*$. Where $c \in \Sigma$, to define $\hat{\Delta}(q, \tau^\frown c)$ first make the set of all states reachable without ε moves from a state in $\hat{\Delta}(q, \tau)$ with $S = \Delta(q_{i_0}, c) \cup \cdots \Delta(q_{i_k}, c)$. Then include ε moves: $\hat{\Delta}(q, \tau^\frown c) = \hat{E}(s_0) \cup \hat{E}(s_1) \cup \cdots \hat{E}(s_n)$ for $s_i \in S$. Observe that the machine $\mathcal{M}$ accepts τ_0 if $\hat{\Delta}(q_0, \tau_0) \cap F$ is not empty and otherwise $\mathcal{M}$ rejects τ_0.

Equivalence of the machine types In this section we have apparently extended the capabilities of Finite State machines, including by adding nondeterminism. This can be puzzling. For instance, if we feed the input $\tau =$ `010101110` to the machine from Example 2.12 then it accepts, but that machine is set up to accept strings that begin with two blocks of `01`'s, while τ begins with three. How can it know, without peeking ahead, that it should ignore the first block but transition on the second? How can it guess?

In mathematics we can consider whatever we can make precise but we have so far stuck to models of machines that are in principle physically realizable. So this may seem to be a shift.

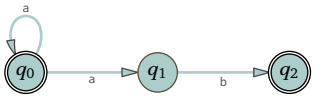
It is not a shift. We finish this section by showing how to convert any nondeterministic Finite State machine into a deterministic one that does the same job. So we can take nondeterminism to be an abbreviation, a shorthand, a convenience, a way of alternatively specifying a deterministic machine. This obviates at least some of the paradox of guessing.

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. 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 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.

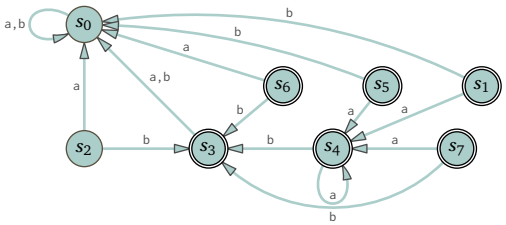
2.22 EXAMPLE This nondeterministic machine $\mathcal{M}_N$ has no ε transitions. Its language is $\mathcal{L} = \{\sigma \in \{\mathtt{a}, \mathtt{b}\}^* \bigm| \sigma = \mathtt{a}^n \text{ or } \sigma = \mathtt{a}^n\mathtt{ab} \text{ for } n \in \mathbb{N}\}$.



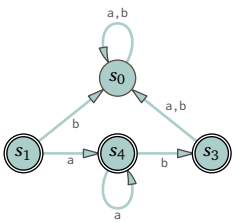
The associated deterministic machine $\mathcal{M}_D$ is below. Each member of this machine is a set of $\mathcal{M}_N$'s states. The start state of $\mathcal{M}_D$ is the one-element set $\{q_0\} = s_1$ containing the start state of $\mathcal{M}_N$, and a state of $\mathcal{M}_D$ is accepting if any of its elements are accepting states in $\mathcal{M}_N$.

As an illustration of constructing the transition table, suppose that we are in $s_5 = \{q_0, q_2\}$ and are reading a. Combine the above machine's next states due to q_0 with those due to q_2, that is, $\Delta_D(s_5, \mathtt{a}) = \{q_0, q_1\} \cup \{\} = \{q_0, q_1\}$. That's 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



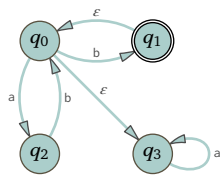
The machine's table, and its transition graph, make clear that $\mathcal{M}_D$ is deterministic. Many of the states are unreachable; for example, s_6 has only outgoing arrows. Below is the machine with those states removed. Again, the start state is s_1.



Now we give the algorithm, the powerset construction. States in $\mathcal{M}_D$ are sets of states from $\mathcal{M}_N$. The start state of $\mathcal{M}_D$ is the ε closure $\hat{E}(q_0)$ (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.

The transition function Δ_D inputs a state $s_i \in \mathcal{M}_D$, that is, $s_i = \{q_{k_0}, \ldots\ q_{k_i}\}$, along with a character $c \in \Sigma$. First apply $\mathcal{M}_N$'s next state function to s_i's elements to get a set $S_{i,c} = \Delta_N(q_{k_0}, c) \cup \cdots\ \Delta_N(q_{k_i}, c)$ (if s_i is empty then $S_{i,c}$ is also empty). Then include ε moves: where $S_{i,c} = \{q_{j_0}, \ldots\ q_{j_i}\}$, let $\Delta_D(s_i, c) = \hat{E}(q_{j_0}) \cup \cdots\ \hat{E}(q_{j_i})$. (For machines without ε transitions this second part has no effect.)

2.23 Example We next do a nondeterministic machine with ε transitions.



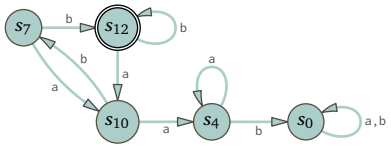
The table below computes the associated deterministic machine. The start state is $\hat{E}(q_0) = \{q_0, q_3\} = s_7$. A state is accepting if it contains q_1.

Here is an example walking through the powerset algorithm. First, let the machine be in state $s_7 = \{q_0, q_3\}$ and reading b. In the terms of the algorithm's description, applying Δ_N to each element of s_7 gives $S_{7,\mathsf{b}} = \Delta_N(q_0, \mathsf{b}) \cup \Delta_N(q_3, \mathsf{b}) = \{q_1\} \cup \{\} = \{q_1\}$. Taking the ε closure gives $\Delta_D(s_7, \mathsf{b}) = \{q_0, q_1, q_3\} = s_{12}$.

Finding the ε closures in advance is a help in constructing the table. We have $\hat{E}(q_0) = \{q_0, q_3\} = s_7$, and $\hat{E}(q_1) = \{q_0, q_1, q_3\} = s_{12}$, and $\hat{E}(q_2) = \{q_2\} = s_3$, and $\hat{E}(q_3) = \{q_3\} = s_4$.

	$S_{i,\mathsf{a}}$	$\Delta_D(s_i, \mathsf{a})$	$S_{i,\mathsf{b}}$	$\Delta_D(s_i, \mathsf{b})$
$s_0 = \{\}$	$\{\}$	s_0	$\{\}$	s_0
$s_1 = \{q_0\}$	$\{q_2\}$	s_3	$\{q_1\}$	s_{12}
+ $s_2 = \{q_1\}$	$\{\}$	s_0	$\{\}$	s_0
$s_3 = \{q_2\}$	$\{\}$	s_0	$\{q_0\}$	s_7
$s_4 = \{q_3\}$	$\{q_3\}$	s_4	$\{\}$	s_0
+ $s_5 = \{q_0, q_1\}$	$\{q_2\}$	s_3	$\{q_1\}$	s_{12}
$s_6 = \{q_0, q_2\}$	$\{q_2\}$	s_3	$\{q_0, q_1\}$	s_{12}
$s_7 = \{q_0, q_3\}$	$\{q_2, q_3\}$	s_{10}	$\{q_1\}$	s_{12}
+ $s_8 = \{q_1, q_2\}$	$\{\}$	s_0	$\{q_0\}$	s_7
+ $s_9 = \{q_1, q_3\}$	$\{q_3\}$	s_4	$\{\}$	s_0
$s_{10} = \{q_2, q_3\}$	$\{q_3\}$	s_4	$\{q_0\}$	s_7
+ $s_{11} = \{q_0, q_1, q_2\}$	$\{q_2\}$	s_3	$\{q_0, q_1\}$	s_{12}
+ $s_{12} = \{q_0, q_1, q_3\}$	$\{q_2, q_3\}$	s_{10}	$\{q_1\}$	s_{12}
$s_{13} = \{q_0, q_2, q_3\}$	$\{q_2, q_3\}$	s_{10}	$\{q_0, q_1\}$	s_{12}
+ $s_{14} = \{q_1, q_2, q_3\}$	$\{q_3\}$	s_4	$\{q_0\}$	s_7
+ $s_{15} = \{q_0, q_1, q_2, q_3\}$	$\{q_2, q_3\}$	s_{10}	$\{q_0, q_1\}$	s_{12}

The transition diagram is below. Many of the machine's table's sixteen states, are unreachable from the starting state $s_7 = \hat{E}(q_0)$. We can see that by starting at s_7 and tracing through the states. The diagram omits unreachable states.



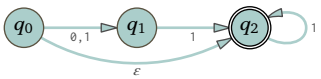
The powerset construction shows that for any nondeterministic machine there is a deterministic machine that recognizes the same language.

We can say more: if the nondeterministic machine has n states then the deterministic machine has at most 2^n states. (It turns out that 2^n is the best that we can do in that for any n there is an n state nondeterministic machine requiring a deterministic machine with 2^n states. However, in practice usually the deterministic machine is not too big once we minimize the number of states. Extra C shows how to minimize.)

IV.2 Exercises

2.24 Give the transition function for the machine of Example 2.8, and of Example 2.9.

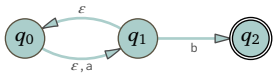
✓ 2.25 Consider this machine.



(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?

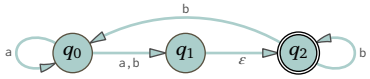
✓ 2.27 Your friend objects, "Epsilon transitions don't make any sense because the machine below will never get its first step done; it just endlessly follows the epsilons." Correct their misimpression.



✓ 2.28 Using the nondeterministic machine from Example 2.23, give a computation tree table like Example 2.15's for each input string. (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 2.12's machine accepts $\tau = 010101110$.

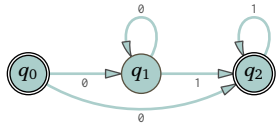
2.30 This machine has $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.



(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.

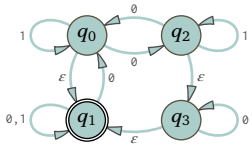
2.31 Produce the table description of the next-state function Δ for the machine in the prior exercise. It should have three columns, for a, b, and ε.

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 $\{\, \sigma = \tau_0 \frown \tau_1 \frown \tau_2 \in \mathbb{B}^* \;\big|\; \tau_0 = 1,\ \tau_1 = 1, \text{ and } \tau_2 = (00)^k \text{ for some } k \in \mathbb{N} \,\}$.

✓ 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) $\mathcal{L}_0 = \{\, \sigma \;\big|\; \sigma \text{ ends in } 00 \,\}$, having three states

(B) $\mathcal{L}_1 = \{\, \sigma \;\big|\; \sigma \text{ has the substring } 0110 \,\}$, with five states

(C) $\mathcal{L}_2 = \{\, \sigma \;\big|\; \sigma \text{ contains an even number of 0's or exactly two 1's} \,\}$, 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: $\mathcal{L} = \{\mathtt{cat}, \mathtt{cap}, \mathtt{carumba}\}$.

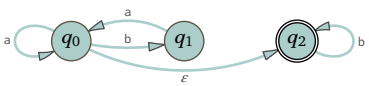
2.38 Give a nondeterministic Finite State machine over $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ recognizing the language of strings that omit at least one of the characters in the alphabet.

✓ 2.39 For each, draw the transition graph for a Finite State machine, which may be nondeterministic, that accepts the given strings from $\{\mathsf{a}, \mathsf{b}\}^*$.

(A) Accepted strings have a second character of a and next to last character of b.

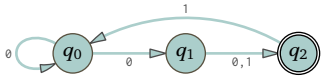
(B) Accepted strings have second character a and the next to last character is also a.

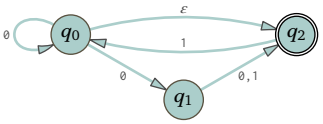
2.40 What is the language of this nondeterministic machine with ε transitions?



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 For each, follow the powerset construction to make a deterministic machine recognizing the same language.





2.43 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.

✓ 2.44 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.

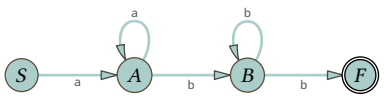
✓ 2.45 For each give a nondeterministic Finite State machine without ε transitions over $\Sigma = \{0, 1, 2\}$. (A) The machine recognizing the language of strings whose final character appears exactly twice in the string. (B) The machine recognizing the language of strings whose final character appears 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, possibly with ε transitions, that recognizes the language with alphabet $\mathbb{B}$. (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 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.

S → a A
A → a A | b B
B → b B | b



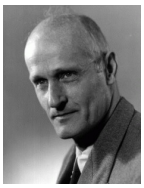
(A) Give three strings from the language of the grammar and show that they are accepted by the machine. (B) Describe that language.

2.49 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) $\mathcal{L}_{NFA} = \{\langle \mathcal{M}, \sigma \rangle \mid \text{the nondeterministic machine } \mathcal{M} \text{ accepts } \sigma\}$

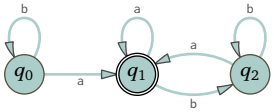
2.50 (A) For the machine of Example 2.23, for each $q \in Q$ produce $E(q, 0)$, $E(q, 1)$, $E(q, 2)$, and $E(q, 3)$. List $\hat{E}(q)$ for each $q \in Q$. (B) Do the same for Exercise 2.30's machine.

Section IV.3 Regular expressions

In 1951, S Kleene[†] was studying a mathematical model of neurons. These are like a Finite State machine in that they do not have scratch memory. Because of that connection, his work has made its way over to here. He noted patterns, that is, regularities, in the languages that they recognize. For instance, the Finite State machine below accepts strings that have any number of b's (perhaps zero many) followed by at least one a, optionally 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, for constructs such as 'any number of'.



Stephen Kleene 1909–1994



Definition A regular expression is a string that describes a language. We will introduce these with a few examples, using the alphabet $\Sigma = \{\mathtt{a}, \ldots \mathtt{z}\}$.

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, $\mathcal{L} = \{\mathtt{pat}, \mathtt{pet}, \mathtt{pit}, \mathtt{pot}, \mathtt{put}\}$.

The alternation operator, '|',[‡] denoting 'or', and the parentheses, which provide grouping, are not part of the strings being described; they are metacharacters.

Besides alternation and parentheses, the regular expression in that example also includes 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,

[†] Pronounced KLAY-nee. He was a PhD student of Church's. [‡] This symbol is called 'pipe'. Other notations for alternation are '+' and '∪'.

including possibly zero-many times'. This regular expression describes the language $\mathcal{L} = \{\texttt{ac}, \texttt{abc}, \texttt{abbc}, \ldots\}$.

3.3 DEFINITION Let $\Sigma = \{x_0, x_1, \ldots\ x_n\}$ be an alphabet not containing the metacharacters '(', ')', '|', or '*'.[†] A regular expression over Σ is a string in the language of this grammar.

$$\begin{array}{lcl}
\langle \textit{reg-exp} \rangle & \to & \langle \textit{char} \rangle \mid \langle \textit{alter} \rangle \mid \langle \textit{concat} \rangle \mid \langle \textit{star} \rangle \\
\langle \textit{char} \rangle & \to & \varnothing \mid \varepsilon \mid x_0 \mid x_1 \mid \ldots\ x_n \\
\langle \textit{alter} \rangle & \to & (\ \langle \textit{reg-exp} \rangle\ \text{'|'}\ \langle \textit{reg-exp} \rangle\ ) \\
\langle \textit{concat} \rangle & \to & (\ \langle \textit{reg-exp} \rangle\ \langle \textit{reg-exp} \rangle\ ) \\
\langle \textit{star} \rangle & \to & (\ \langle \textit{reg-exp} \rangle\ )*
\end{array}$$

In words, any of the single character strings $\varnothing$ or ε or x_0, ... or x_n are regular expressions. And, where R_0 and R_1 are regular expressions then so also are $(R_0|R_1)$, and (R_0R_1), and $(R_0)*$. Thus these four strings are regular expressions over $\Sigma = \{\texttt{a}, \texttt{b}\}$: `(ab)`, and `(a|b)`, and `b`, and `(a(b)*)`. (Below we will discuss conventions for omitting some parentheses.)

That definition gives the syntax of regular expressions, the rules for the structure of valid strings. As to their semantics, what they mean, a regular expression describes a language. The language described by the one-character regular expression $\varnothing$ is the empty language, $\mathcal{L}(\varnothing) = \varnothing$. The language described by the regular expression consisting of only the character ε is the one-element language containing only the empty string, $\mathcal{L}(\varepsilon) = \{\varepsilon\} = \{\text{''}\}$. If the regular expression consists of a single character from the alphabet Σ then the language that it describes contains only one string, which consists of only that single character, as in $\mathcal{L}(\texttt{a}) = \{\texttt{a}\}$.

We finish the semantics with the operations. Start with regular expressions R_0 and R_1 describing languages $\mathcal{L}(R_0)$ and $\mathcal{L}(R_1)$. The alternation of two regular expressions $(R_0|R_1)$ describes the union of their languages, $\mathcal{L}(R_0) \cup \mathcal{L}(R_1)$. Concatenation $({}_0R_1)$ describes concatenation of the languages, $\mathcal{L}(R_0)^\frown\mathcal{L}(R_1)$. And, the Kleene star $(R_0)*$ describes the star of the language $\mathcal{L}(R)^*$.

3.4 EXAMPLE Consider the regular expression `(ab)` over $\Sigma = \{\texttt{a}, \texttt{b}\}$. It is the concatenation of the regular expression `a` with the regular expression `b`. The first describes a single-element language $\mathcal{L}(\texttt{a}) = \{\texttt{a}\}$ and likewise the second describes $\mathcal{L}(\texttt{b}) = \{\texttt{b}\}$. Thus $\mathcal{L}(\texttt{(ab)}) = \mathcal{L}(\texttt{a})^\frown\mathcal{L}(\texttt{b}) = \{\texttt{ab}\}$, another language with only one element.

As formally defined, the syntax rules call for lots of parentheses. We cut down on the annoyance by adopting a convention for operator precedence: star binds most tightly, then concatenation, and then the alternation operator, `|`, binds least tightly.

3.5 EXAMPLE Instead of `(a(b(a)*))` we write `aba*`. The precedence rules give that the star applies only to the `a` before it, as in `ab(a*)`. (If concatenation bound more

[†] 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 this symbol also appears in regular expressions. For that usage the grammar wraps it in single quotes, as '|'.

tightly than star then aba* would instead mean (aba)*.) The described language is $\mathcal{L}(\texttt{aba*}) = \{\texttt{ab}, \texttt{aba}, \texttt{abaa}, \texttt{aba}^3, \ldots\} = \{\texttt{aba}^n \mid n \in \mathbb{N}\}$.

3.6 Remark There is an interaction between alternation and star. Consider the regular expression (b|c)*. It could mean either 'any number of repetitions of picking b or c' or 'pick b or c and repeat that character any number of times'. The first of those two is more convenient. Thus the language described by (b|c)* is $\mathcal{L} = \{\varepsilon, \texttt{b}, \texttt{c}, \texttt{bb}, \texttt{bc}, \texttt{cb}, \texttt{cc}, \ldots\}$. To describe the language whose members consist of any number of b's or any number of c's, $\hat{\mathcal{L}} = \{\varepsilon, \texttt{b}, \texttt{bb}, \ldots, \texttt{c}, \texttt{cc}, \ldots\}$, use the regular expression b*|c*.

We next see some common constructs. These examples use $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$.

3.7 Example The language consisting of strings of a's whose length is a multiple of three, $\mathcal{L} = \{a^{3k} \mid k \in \mathbb{N}\} = \{\varepsilon, \texttt{aaa}, \texttt{aaaaaa}, \ldots\}$, is described by (aaa)*.

Note that the empty string is a member of that language. A common mistake is to forget that star includes the possibility of zero-many repetitions.

3.8 Example To match any character we can list them all. The language over $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ of three-letter words ending in bc is $\{\texttt{abc}, \texttt{bbc}, \texttt{cbc}\}$. The regular expression (a|b|c)bc describes it. (Another is (abc)|(bbc)|(cbc).)

3.9 Example Use ε to mark things as optional. Thus a*(ε|b) describes the language of strings that have any number of a's and optionally end in one b, $\mathcal{L} = \{\varepsilon, \texttt{b}, \texttt{a}, \texttt{ab}, \texttt{aa}, \texttt{aab}, \ldots\}$. Similarly, to describe the language consisting of words with between three and five a's, $\mathcal{L} = \{\texttt{aaa}, \texttt{aaaa}, \texttt{aaaaa}\}$, we can use aaa($\varepsilon$|a|aa).

3.10 Example The language $\{\texttt{b}, \texttt{bc}, \texttt{bcc}, \texttt{ab}, \texttt{abc}, \texttt{abcc}, \texttt{aab}, \ldots\}$ 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(ε|c|cc).

Also see Extra A for extensions that are widely used in practice.

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 Fix an alphabet Σ. We will show that for any regular expression R over Σ there is a machine with alphabet Σ accepting exactly the strings matching the expression. We use induction on the structure of regular expressions.

Start with regular expressions consisting of a single character. If $R = \varnothing$ then $\mathcal{L}(R) = \{\,\}$ and the machine on the left below recognizes this language. If $R = \varepsilon$ then $\mathcal{L}(R) = \{\varepsilon\}$ and the machine in the middle recognizes it. If the regular

expression is a character from the alphabet such as $R = \mathtt{a}$ then the machine on the right works.

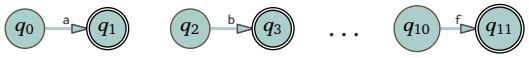
We finish by handling the three operations. Let R_0 and R_1 be regular expressions. The inductive hypothesis gives a machine $\mathcal{M}_0$ whose language is described by R_0 and a machine $\mathcal{M}_1$ whose language is described by R_1.

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.

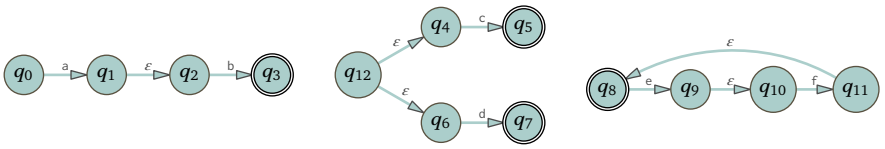
Next consider concatenation, $R = R_0 \frown R_1$. Join the two machines serially: for each accepting state in $\mathcal{M}_0$, make an ε transition to the start state of $\mathcal{M}_1$ and then convert all of the accepting states of $\mathcal{M}_0$ to be non-accepting states. See Example 2.18.

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. □

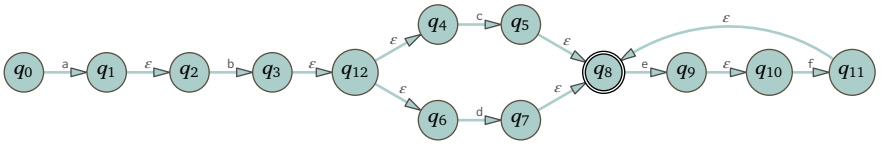
3.13 Example Building a machine for the regular expression `ab(c|d)(ef)*` starts with machines for each of the single characters.



Put these atomic components together



to get the complete machine.



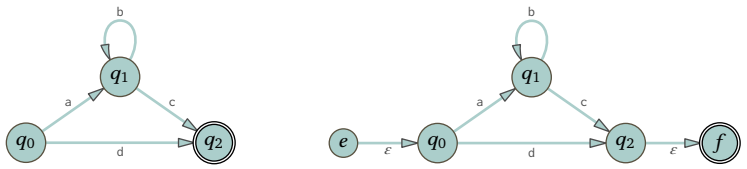
This is a nondeterministic machine with ε transitions. To get a deterministic machine use the powerset construction.

3.14 Lemma Any language recognized by a Finite State machine is described by a regular expression.

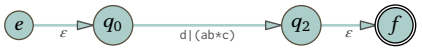
Our strategy starts with a Finite State machine and eliminates its states one at a time. Below is an illustration, before and after pictures of part of a larger machine, where we eliminate the state q.

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 $\varnothing$ 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 $\hat{M}$). 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. Connect it to q_0 with an ε transition, 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.")

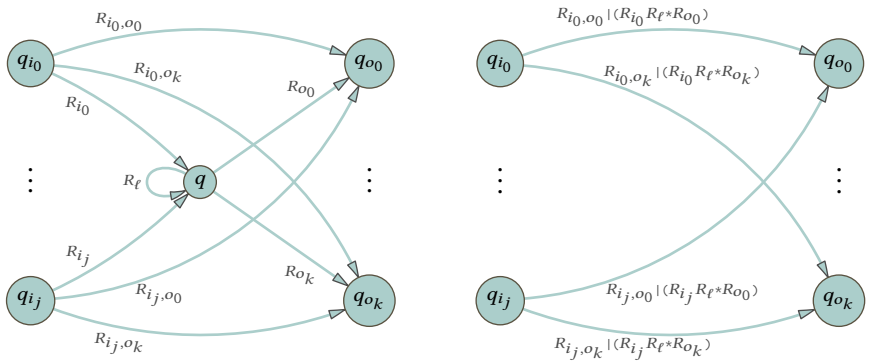
Because the edge labels are regular expressions, we can arrange that from any q_i to any q_j there is at most one edge, since if $\mathcal{M}$ has more than one edge then in $\hat{\mathcal{M}}$ we can use alternation, '|', to combine the labels.

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 states that are useless in that they don't affect which strings are accepted. 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 unreachable from the start e then drop that node along with its incoming and outgoing edges. (The idea behind useless states has some technical aspects. For instance, 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 machine have only finitely many nodes and so this omitting process must eventually finish. For a definition of unreachability see Exercise 3.34.)

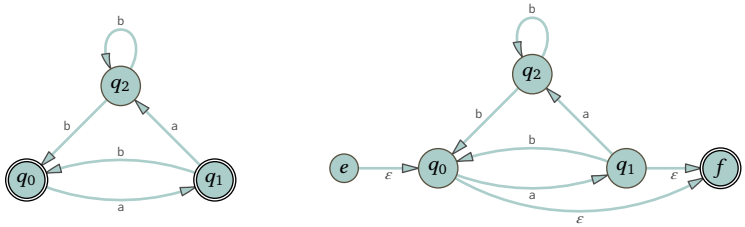
With that, $\hat{\mathcal{M}}$ is ready for state elimination. Pick any q not equal to e or f. Below are before and after diagrams. By the setup work above, q has at least one incoming and at least one outgoing edge. So there are states $q_{i_0}, \ldots q_{i_j}$ with an edge leading into q, and states $q_{o_0}, \ldots q_{o_k}$ that receive an edge leading out of q. In addition, q may have a loop. (A fine point is that possibly some of the states shown on the left of each diagram equal some shown on the right. For example, possibly q_{i_0} equals q_{o_0}, and the edge on the top of each diagram is a loop.)



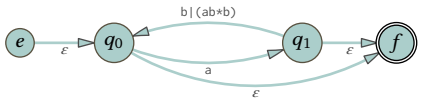
Eliminate q and the associated edges by making the replacements shown on the after diagram. (If an edge is not present then don't include any regular expression in the replacement. For instance, if there is no R_ℓ edge then the right's top edge should be $R_{i_0,o_0} \mid R_{i_0} R_{o_0}$.) By construction for any two states in the after machine, q_i and q_o, in passing from the before diagram to the after, the set of strings taking the machine from the first to the second is unchanged. Thus languages of the before and after machines are equal.

Repeat this procedure until the only states left are e and f. The desired regular expression is on the sole remaining edge. ∎

3.15 Example We want a regular expression describing the language of the machine $\mathcal{M}$ on the left below. Introduce e and f as on the right. There are no useless states so this is $\hat{\mathcal{M}}$.

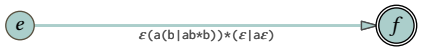


We will eliminate q_2, then q_1, then q_0 (this choice is arbitrary; we could have instead done any other order). For the first, in the notation used in the proof's diagram, $q_1 = q_{i_0}$ and $q_0 = q_{o_0}$. The regular expressions are $R_{i_0,o_0} = \mathtt{b}$, $R_{i_0} = \mathtt{a}$, $R_\ell = \mathtt{b}$, and $R_{o_0} = \mathtt{b}$. Elimination leaves the resulting machine with an arrow from $q_{i_0} = q_1$ to $q_{o_0} = q_0$ labeled $R_{i_0,o_0} \mathtt{|} (R_{i_0} R_\ell \mathtt{*} R_{o_0})$, which is $\mathtt{b|(ab*b)}$.



Next q_1. There is one node giving an incoming arrow, $q_0 = q_{i_0}$, and two nodes associated with outgoing arrows, $q_0 = q_{o_0}$ 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: there is no arrow for R_{i_0,o_0}, $R_{i_0,o_1} = \varepsilon$, $R_{i_0} = \mathtt{a}$, $R_{o_0} = \mathtt{b|(ab*b)}$, there is also no arrow for R_ℓ, and $R_{o_1} = \varepsilon$. Eliminating q_1 means that the next machine has an arrow from $q_{i_0} = q_0$ to $q_{o_0} = q_0$ labeled $R_{i_0,o_0} \mathtt{|} (R_{i_0} R_\ell \mathtt{*} R_{o_0})$, which is $\mathtt{a(b|ab*b)}$. It also means that the machine has an arrow from $q_{i_0} = q_0$ to $q_{o_1} = q_f$ labeled $R_{i_0,o_1} \mathtt{|} (R_{i_0} R_\ell \mathtt{*} R_{o_1})$, which is $\varepsilon \mathtt{|(a}\varepsilon\mathtt{)}$.

Final step. The sole incoming node is $e = q_{i_0}$ and the sole outgoing node is $f = q_{o_0}$. As well, $R_{i_0} = \varepsilon$, $R_{o_0} = \varepsilon \mathtt{|a}\varepsilon$, and $R_\ell = \mathtt{a(b|ab*b)}$.



This regular expression describes the language of the starting machine (we can simplify it; for instance, in the final parenthesis we can replace $\mathtt{a}\varepsilon$ with $\mathtt{a}$).

IV.3 Exercises

3.16 Decide if the string σ matches the regular expression R. (A) $\sigma = \mathtt{0010}$, $R = \mathtt{0*10}$ (B) $\sigma = \mathtt{101}$, $R = \mathtt{1*01}$ (C) $\sigma = \mathtt{101}$, $R = \mathtt{1*(0|1)}$ (D) $\sigma = \mathtt{101}$, $R = \mathtt{1*(0|1)*}$ (E) $\sigma = \mathtt{01}$, $R = \mathtt{1*01*}$

✓ 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) $\mathtt{01*}$ (B) $\mathtt{(01)*}$ (C) $\mathtt{1(0|1)1}$ (D) $\mathtt{(0|1)(}\varepsilon\mathtt{|1)0*}$ (E) $\varnothing$

3.18 Give a brief plain English description of the language for each regular expression. (A) `a*cb*` (B) `aa*` (C) `a(a|b)*bb`

✓ 3.19 For each string in $\{\mathtt{a}, \mathtt{b}\}^*$ that is of length less than or equal to 3, decide if the string is a match. (A) `a*b` (B) `a*` (C) $\varnothing$ (D) ε (E) `b(a|b)a` (F) `(a|b)(`ε`|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) $\varnothing$ (D) ε (E) `0(0|1)*` (F) `(100)(`ε`|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 must mean 'match the inside and repeat'. So I think that `(0*1)*` should match the strings `001001` and `010101`, but not the strings `01001` and `00000101`, because you can't write those two as σ^n for any substring σ." Straighten them out.

3.22 The person behind you in class says, "I don't get it. I got a regular expression that I am sure is right. But I looked in the answers and the book got a different one." Explain what is up.

3.23 Produce a regular expression for the language of bitstrings that have a substring consisting of at least three consecutive 1's.

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 nondeterministic machine is acceptable). (A) $\mathcal{L}_0 = \{ \mathtt{a}^n\mathtt{b}^{2m} \mid m, n \geq 1 \}$ (B) $\mathcal{L}_1 = \{ \mathtt{a}^n\mathtt{b}^{3m} \mid m, n \geq 1 \}$

3.25 Give a regular expression for the language over $\Sigma = \{ \mathtt{a}, \mathtt{b}, \mathtt{c} \}$ whose strings are missing at least one letter, that is, whose strings 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 $\Sigma = \{ \mathtt{a}, \mathtt{b} \}$. (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 $\Sigma = \{ \mathtt{a}, \mathtt{b}, \mathtt{c} \}$. (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 $\Sigma = \{ \mathtt{a}, \mathtt{b} \}$. (A) Every a is both immediately preceded and immediately followed by a b. (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) $\{\sigma \in \{\mathtt{a},\mathtt{b}\}^* \mid \sigma \text{ ends with the same symbol that it began with, and } \sigma \neq \varepsilon\}$

(B) $\{\mathtt{a}^i\mathtt{b}\mathtt{a}^j \mid i \text{ and } j \text{ leave the same remainder on division by three}\}$

✓ 3.32 Give a regular expression describing each language over $\mathbb{B}^*$.

(A) The strings representing a binary number that is a multiple of two.

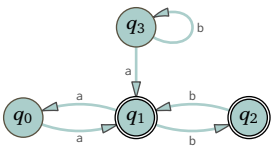
(B) The bitstrings where the first character differs from the final one.

(C) The bitstrings where no two adjacent characters are equal.

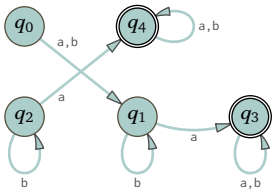
✓ 3.33 Produce a Finite State machine whose language equals the language described by each regular expression. (A) `a*ba` (B) `ab*(a|b)*a`

3.34 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.

(A)



(B)



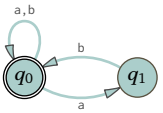
3.35 Here is a grammar for regular expressions that reflects the operator precedence rules.

$$\langle \textit{reg-exp} \rangle \;\to\; \langle \textit{concat} \rangle \mid \langle \textit{reg-exp} \rangle \text{ `|' } \langle \textit{concat} \rangle$$
$$\langle \textit{concat} \rangle \;\to\; \langle \textit{simple} \rangle \mid \langle \textit{concat} \rangle \langle \textit{simple} \rangle$$
$$\langle \textit{simple} \rangle \;\to\; (\ \langle \textit{reg-exp} \rangle\ ) \mid \langle \textit{simple} \rangle * \mid \varnothing \mid \varepsilon \mid x_0 \mid \ldots \mid x_n$$

Derive and construct the parse tree for each regular expressions over $\Sigma = \{\mathtt{a},\mathtt{b},\mathtt{c}\}$. (A) `a(b|c)` (B) `ab*(a|c)`

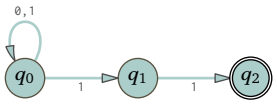
3.36 Use the grammar in the prior exercise to give the parse trees for Remark 3.6's `a(b|c)*` and `a(b*|c*)`.

3.37 Apply the method of Lemma 3.14's proof to this machine to eliminate q_0.



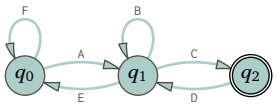
(A) Get $\hat{\mathcal{M}}$ by introducing e and f. (B) Where $q = q_0$, describe which state from the machine is playing the diagram's before picture role of q_{i_0}, which edge is R_{i_0}, etc. (C) Eliminate q_0.

✓ 3.38 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.

3.39 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 as described in the proof's comment on the fine point.



3.40 (IIS, IIT 2021) Let $\mathcal{L} \subseteq \mathbb{B}^*$ be recognized by a deterministic Finite State machine having exactly k states. One of these must also be accepted by such a machine with k states, while for the other that is not necessarily right. Which is which? (A) The complement, $\mathcal{L}^c$ (B) $\mathcal{L} \cup \{\texttt{01}\}$

3.41 Fix a Finite State machine $\mathcal{M}$. Kleene's Theorem shows that the set of strings taking $\mathcal{M}$ from the start state q_0 to the set of final states is regular.

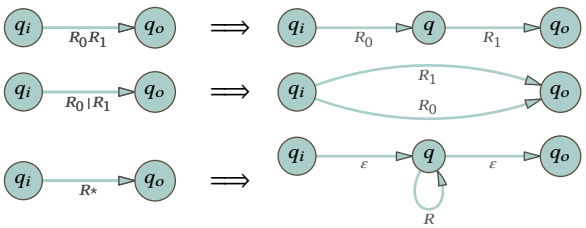
(A) Show that for any set of states $S \subseteq Q_{\mathcal{M}}$, final or not, the set of strings taking $\mathcal{M}$ from q_0 to one of the states in S is regular.

(B) Show that the set of strings taking $\mathcal{M}$ from any single state to any other single state is regular.

3.42 Fix an alphabet Σ. Show that the set of languages over Σ that are described by a regular expression is countably infinite. Conclude that there are languages over Σ not recognized by any Finite State machine.

3.43 An alternative proof of Lemma 3.12, the subset method, goes from a given regular expression to an associated machine by reversing the steps of Lemma 3.14. 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*)`

3.44 Nondeterministic Finite State machines can always be made to have a single accepting state. For deterministic machines that is not so.

(A) Show that any deterministic Finite State machine that recognizes the finite language $\mathcal{L}_1 = \{\varepsilon, \mathtt{a}\}$ must have at least two accepting states.

(B) Show that any deterministic Finite State machine that recognizes $\mathcal{L}_2 = \{\varepsilon, \mathtt{a}, \mathtt{aa}\}$ 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.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 now 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. There are languages that are not regular.

Proof Call the alphabet Σ. We first show that the set of regular languages over Σ is infinite. Section A specifies that any alphabet is nonempty and finite. 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 show that the set of regular languages over Σ is at most countable. There is one language for each regular expression so we can do that by showing that there are countably many regular expressions. There are finitely many regular expressions of length 1, finitely many of length 2, etc. The union of them all is a countable union of countable sets, and so is countable.

To finish we show that the set of all languages over Σ is uncountable, from which it follows that there are languages that are not regular. First, the collection of strings Σ^* is infinite because where $y \in \Sigma$, it contains infinitely the many different elements, y, yy, ... In addition, that collection contains finitely many strings of length zero, finitely many of length one, etc. and so is a countable union of countable sets, and is therefore countably infinite. In contrast, the set of all languages $\mathcal{L} \subseteq \Sigma^*$ is the power set of Σ^*, and so has a greater cardinality, which makes it uncountable. □

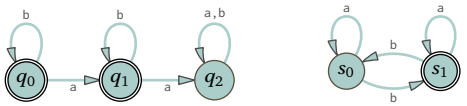
[†] This is just like how the fact that Turing machines, general recursive functions, and many other models all compute the same sets says that these computable sets are a natural and important collection. This collection is not just a historical artifact of what happened to be first proposed.

Closure properties In proving the first half of Kleene's Theorem, Lemma 3.12, 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 $\mathcal{L}_0 \frown \mathcal{L}_1$, and the Kleene star $\mathcal{L}_0{}^*$. A set is 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
+	q_0	q_1	q_0
+	q_1	q_2	q_1
	q_2	q_2	q_2

	Δ_1	a	b
	s_0	s_0	s_1
+	s_1	s_1	s_0

The product machine $\mathcal{M}$ has states that are the members of the cross product $Q_0 \times Q_1$ and transitions that are given by $\Delta\big((q_i, s_j)\big) = \big(\Delta_0(q_i),\ \Delta_1(s_j)\big)$. Its start state is (q_0, s_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)

As an example, with aba on the tape, $\mathcal{M}$'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 to q_2, while $\mathcal{M}_1$ does s_0 to s_0, to s_1, and to s_1. That is, the product machine $\mathcal{M}$ runs $\mathcal{M}_0$ and $\mathcal{M}_1$ in parallel.

We have not yet fully specified the machine because we have not said which states are accepting. On the left below, (q_i, s_j) is accepting if both q_i and s_j are accepting. With this, $\mathcal{M}$ accepts a string if and only if both $\mathcal{M}_0$ and $\mathcal{M}_1$ accept it, so $\mathcal{M}$ recognizes $\{\sigma \in \Sigma \mid \sigma \text{ has fewer than two a's and an odd number of b's}\}$.

[†] If the two languages have different alphabets Σ_0 and Σ_1 then the two languages as well as their union are regular over the alphabet $\Sigma_0 \cup \Sigma_1$.

	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)

	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)

On the right, accepting states (q_i, s_j) are the ones where q_i is accepting and s_j is not. Then the machine accepts strings that are in the language of $\mathcal{M}_0$ but not that of $\mathcal{M}_1$, so $\mathcal{M}$ recognizes $\{\sigma \in \Sigma \mid \sigma \text{ has fewer than two a's and an even number of b's}\}$.

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 Fix an alphabet Σ and consider languages $\mathcal{L}_0$ and $\mathcal{L}_1$. Let them be recognized by the Finite State machines $\mathcal{M}_0$ and $\mathcal{M}_1$. 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 then $\mathcal{M}$ recognizes 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 $\mathcal{M}$ recognizes the set difference of the languages, $\mathcal{L}_0 - \mathcal{L}_1$. A special case of this is when $\mathcal{L}_0$ is the set of all strings, Σ^*, so that $\mathcal{M}$ recognizes the complement, $\mathcal{L}_1{}^{\mathsf{c}}$. □

These closure properties often simplify showing that a language is regular.

4.6 EXAMPLE To show that the language

$$\mathcal{L} = \{\sigma \in \mathbb{B}^* \mid \sigma \text{ has an even number of } \mathtt{0}\text{'s and more than two 1's}\}$$

is regular, we could produce a machine that recognizes it or exhibit a regular expression. Instead, following the above result we note that $\mathcal{L}$ is the intersection of $\{\sigma \in \mathbb{B}^* \mid \sigma \text{ has an even number of } \mathtt{0}\text{'s}\}$ and $\{\sigma \in \mathbb{B}^* \mid \sigma \text{ has more than two 1's}\}$. Showing that those two are regular by producing machines or regular expressions is easy.

IV.4 Exercises

4.7 Someone in class says, "I know that regular languages are closed under closure properties. For example, we know if $\mathcal{L}_0$ and $\mathcal{L}_1$ are regular then their intersection $\mathcal{L}_0 \cap \mathcal{L}_1$ is also regular. But when $\mathcal{L}_0$ and $\mathcal{L}_1$ are not regular then $\mathcal{L}_1 0 \cap \mathcal{L}_1 = \mathcal{L}_2$ doesn't make $\mathcal{L}_2$ not regular, why? Doesn't being closed mean for non-regularity too?" Explain it to them.

4.8 Is English a regular language?

4.9 Name a class of languages that are closed under intersection and union but not under complement.

✓ 4.10 True or false? Justify each answer.
(A) The empty language is not regular.
(B) The intersection of two languages is regular.
(C) The language of all bitstrings, $\mathbb{B}^*$, is not regular.
(D) In every infinite regular language there are two strings where no character from the alphabet is in the same place in both.

4.11 One of these is true and one is false. Which is which? (A) Any finite language is regular. (B) Any regular language is finite.

4.12 Is $\{\sigma \in \mathbb{B}^* \mid \sigma \text{ represents in binary a power of } 2\}$ a regular language?

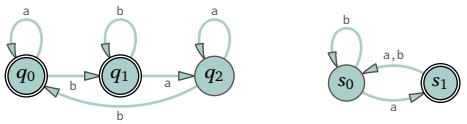
✓ 4.13 Show that each language over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ is regular.
(A) $\{\sigma \in \Sigma^* \mid \sigma \text{ starts and ends with } \mathtt{a}\}$
(B) $\{\sigma \in \Sigma^* \mid \text{the number of a's is even}\}$

4.14 Is the set of strings 9^n that occur in the decimal expansion of π a regular language?

✓ 4.15 Suppose that the language $\mathcal{L}$ over $\mathbb{B}$ is regular. Show that the language $\hat{\mathcal{L}} = \{1^\frown\sigma \mid \sigma \in \mathcal{L}\}$, also over $\mathbb{B}$, is also regular.

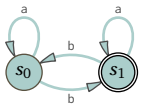
4.16 If two machines have n_0 states and n_1 states, how many states does their product have?

✓ 4.17 For these two,



give the transition table for the product machine. 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.

4.18 Find the cross product of this machine, $\mathcal{M}$ from Example 4.4, with itself.



Set the accepting states so that it accepts the same language as $\mathcal{M}$.

4.19 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.

✓ 4.20 For each, decide whether it is true or false. Briefly justify.
(A) Every language is the subset of a regular language.
(B) Every language has a subset that is not regular.
(C) The union of a regular language and a language that is not regular must be not regular.
(D) The union of two regular languages is regular, without exception.

4.21 Choose the right letter to fill in the blank (with justification): the concatenation of a regular language with a language that is not regular _____ regular.
(A) must be (B) might be, or might be not (C) cannot be

4.22 True or false? Briefly justify.
(A) Regularity is inherited by subsets: if $\mathcal{L}_0$ is a regular language and $\mathcal{L}_1 \subseteq \mathcal{L}_0$ then $\mathcal{L}_1$ is also regular.
(B) Non-regularity is inherited by supersets: if $\mathcal{L}_1$ is not regular and $\mathcal{L}_1 \subseteq \mathcal{L}_0$ then $\mathcal{L}_0$ is also not regular.

4.23 Where $\mathcal{L}$ is a language, define $\mathcal{L}+$ as the language $\mathcal{L}^\frown \mathcal{L}^*$. Show that if $\mathcal{L}$ is regular then so is $\mathcal{L}+$.

4.24 Use closure properties in showing that if $\mathcal{L}$ is regular then the set of even-length strings in $\mathcal{L}$ is also regular.

4.25 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's Example 5.2 shows that $\{ a^n b^n \in \{a,b\}^* \mid n \in \mathbb{N}\}$ is not regular. Show that $\{\sigma \in \{a,b\}^* \mid \sigma \text{ contains the same number of a's as b's}\}$ is not regular using closure properties. *Hint:* one way is to use closure under intersection.

✓ 4.26 Lemma 4.2 gives a counting argument, a pure existence proof, that there are languages that are not regular. But we can also exhibit such a language. Prove that $\mathcal{L} = \{1^k \mid k \in K\}$ is not regular, where K is the Halting problem set, $K = \{e \in \mathbb{N} \mid \phi_e(e)\downarrow\}$.

4.27 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.

4.28 Lemma 4.2 shows that the collection of regular languages over $\mathbb{B}$ is countable. Show that not every individual language in that collection is countable.

4.29 An alternative definition of a regular language is as one generated by a regular grammar, where rewrite rules have three forms: $X \to tY$, or $X \to t$, or $X \to \varepsilon$. That is, the rule head has one nonterminal and the rule body either has a terminal followed by a nonterminal, or a single nonterminal, or is the empty string. This is an example, with the language that it generates.

$$\begin{array}{lcl} S & \to & aS \mid bS \mid aA \\ A & \to & aB \\ B & \to & \varepsilon \mid b \end{array} \qquad \mathcal{L} = \{\sigma \in \{a,b\}^* \mid \sigma = \tau^\frown aa \text{ or } \sigma = \tau^\frown aab\}$$

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. (A) For each nonterminal X make a machine state q_X, where the start state is the one for the start symbol. (B) For each $X \to \varepsilon$ rule make state q_X accepting. (C) For each $X \to tY$ rule put a transition from q_X to q_Y labeled t.

(D) If there are any $X \longrightarrow t$ rules then make an accepting state $\hat{q}$, and for each such rule put a transition from q_X to $\hat{q}$ labeled t.

4.30 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 at least one element $\alpha \in \mathcal{L}$, that is, $\text{pref}(\mathcal{L}) = \{\sigma \in \Sigma^* \mid \text{there is a } \tau \in \Sigma^* \text{ such that } \sigma^\frown\tau = \alpha \in \mathcal{L}\}$

(B) $\text{suff}(\mathcal{L})$ contains the strings that are a suffix of some string in $\mathcal{L}$, that is, $\text{suff}(\mathcal{L}) = \{\sigma \in \Sigma^* \mid \text{there is a } \tau \in \Sigma^* \text{ such that } \tau^\frown\sigma \in \mathcal{L}\}$

(C) $\text{allprefs}(\mathcal{L})$ contains the strings σ such that every prefix of σ is in $\mathcal{L}$, so $\text{allprefs}(\mathcal{L}) = \{\sigma \in \mathcal{L} \mid \text{every } \tau \in \Sigma^* \text{ that is a prefix of } \sigma \text{ has } \tau \in \mathcal{L}\}$

4.31 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 "by construction."

(A) 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.

(B) Use the identity $S - T = S \cap T^c$ to make a similar observation about set difference.

(C) Show that the complement of a regular language is also a regular language.

4.32 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 $n \leq k < 2n$.

4.33 Fix two alphabets Σ_0, Σ_1. A function $h\colon \Sigma_0 \longrightarrow {\Sigma_1}^*$ induces a homomorphism on ${\Sigma_0}^*$ via the operation $h(\sigma^\frown\tau) = h(\sigma)^\frown h(\tau)$ and $h(\varepsilon) = \varepsilon$.

(A) 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 $\hat{h}(101)$.

(B) 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.

(C) Prove that the collection of regular languages is closed under homomorphism, that if $\mathcal{L}$ is regular then so is $h(\mathcal{L})$.

4.34 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.

4.35 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 $\mathcal{L}^R = \{\sigma^R \mid \sigma \in \mathcal{L}\}$.

(A) 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.

(B) We will prove the result by showing that for any regular expression R, the

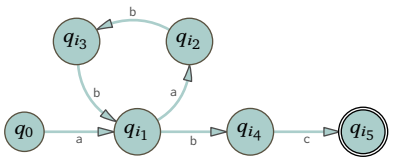
reversal $\mathcal{L}(R)^{\mathsf{R}}$ is described by a regular expression. We will construct this expression by defining a reversal operation on regular expressions. Fix an alphabet Σ and let (1) $\varnothing^{\mathsf{R}} = \varnothing$, (2) $\varepsilon^{\mathsf{R}} = \varepsilon$, (3) $x^{\mathsf{R}} = x$ for any $x \in \Sigma$, (4) $(R_0 \frown R_1)^{\mathsf{R}} = {R_1}^{\mathsf{R}} \frown {R_0}^{\mathsf{R}}$, (5) $(R_0 \,|\, R_1)^{\mathsf{R}} = {R_0}^{\mathsf{R}} \,|\, {R_1}^{\mathsf{R}}$, and (6) $(R*)^{\mathsf{R}} = (R^{\mathsf{R}})*$. (Note the connection between (4) and the prior exercise item.) Now show that R^{R} describes $\mathcal{L}(R)^{\mathsf{R}}$. *Hint:* use induction on the length of the regular expression R.

Section IV.5 Non-regular languages

The prior section show that there are languages that are not regular via a counting argument. We now see a technique to show that specified languages are not regular.[†] This is similar to the second chapter, where we first used a counting argument to prove that there are unsolvable problems and later showed that specific problems such as the Halting problem are unsolvable.

The idea is that although Finite State machines are finite, they can get 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. The key observation is that to process long inputs with only a small number of states, a machine must revisit states, that is, it must cycle.

Cycles inside a machine cause a pattern in what that machine accepts. The diagram below shows a machine that accepts aabbbc (it only shows some of the states, those that the machine traverses in processing this input).



(*)

Because of the cycle, in addition to aabbbc this machine must also accept $\mathsf{a}(\mathsf{abb})^2\mathsf{bc}$ since that string takes the machine through the cycle twice. Likewise, this machine accepts $\mathsf{a}(\mathsf{abb})^3\mathsf{bc}$, and cycling more times pumps out more accepted strings.

5.1 Theorem (Pumping Lemma) Let $\mathcal{L}$ be a regular language. 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 $\sigma = \alpha\frown\beta\frown\gamma$ satisfying: (1) the first two components are short, $|\alpha\beta| \leq p$, (2) β is not empty, and (3) the strings $\alpha\gamma$, $\alpha\beta^2\gamma$, $\alpha\beta^3\gamma$, ... are also members of the language $\mathcal{L}$.

Proof Suppose that $\mathcal{L}$ is recognized by the deterministic Finite State machine $\mathcal{M}$. For p it suffices to use the number of states in $\mathcal{M}$.

[†] **??** contains another way to show that a language is not regular. While somewhat more abstract, it applies to all non-regular languages whereas the result here does not apply to some (see Exercise 5.30).
[‡] If p works then so does any number greater than p.

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, 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 $\langle s_0, \ldots s_i \rangle$ and $\langle s_0, \ldots s_i, \ldots s_j \rangle$ that take the machine to q. That is, i and j are minimal such that $i \neq j$ and the extended transition function gives $\hat{\Delta}(\langle s_0, \ldots s_i \rangle) = \hat{\Delta}(\langle s_0, \ldots s_j \rangle) = q$. Let $\alpha = \langle s_0, \ldots, s_i \rangle$, let $\beta = \langle s_{i+1}, \ldots s_j \rangle$, and let $\gamma = \langle s_{j+1}, \ldots s_k \rangle$.

These strings satisfy conditions (1) and (2). In particular, choosing q, i, and j to be minimal guarantees that that $|\alpha^\frown\beta| \leq p$ 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

$$\alpha^\frown\gamma = \langle s_0, \ldots s_i, s_{j+1}, \ldots s_k \rangle$$

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 $\alpha\gamma$. As to the other strings in (3), for instance with

$$\alpha^\frown\beta^2{}^\frown\gamma = \langle s_0, \ldots s_i, s_{i+1}, \ldots, s_j, s_{i+1}, \ldots s_j, s_{j+1}, \ldots s_k \rangle$$

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. □

We typically use the Pumping Lemma to show that a language is not regular through an argument by contradiction.

5.2 Example The canonical example is to show that this language of matched parentheses is not regular.

$$\mathcal{L} = \{ (^n)^n \in \Sigma^* \;\big|\; n \in \mathbb{N} \} = \{ \varepsilon, \, (), \, (()), \, ((())), \, (^4)^4, \ldots \}$$

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 $\sigma = (^p)^p$.

That string is an element of $\mathcal{L}$ and $|\sigma| \geq p$. Therefore it decomposes into three substrings $\sigma = \alpha^\frown\beta^\frown\gamma$ in a way that satisfies the conditions. Condition (1) is that the length of the prefix $\alpha^\frown\beta$ is less than or equal to p. Because of this, 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 $\alpha\gamma$, $\alpha\beta^2\gamma$, $\alpha\beta^3\gamma$, ... are elements of $\mathcal{L}$. To get the contradiction, consider $\alpha\beta^2\gamma$ (there are other members of the list that we could choose but we only need to choose one). 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 thus the assumption that $\mathcal{L}$ is regular leads to a contradiction.

5.3 Example Recall that a palindrome is a string that reads the same backwards as forwards, such as bab, abbaabba, or $\mathtt{a}^5\mathtt{ba}^5$. We will show that the language over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ of all palindromes, $\mathcal{L} = \{\sigma \in \Sigma^* \bigm| \sigma^{\mathsf{R}} = \sigma\}$ is not regular.

For contradiction assume that $\mathcal{L}$ is regular. The Pumping Lemma says that this language has a pumping length. Call it p and consider $\sigma = \mathtt{a}^p\mathtt{b}\mathtt{a}^p$.

The string σ is an element of $\mathcal{L}$ and $|\sigma| \geq p$. 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 $\alpha\gamma, \alpha\beta^2\gamma, \alpha\beta^3\gamma, \ldots$ are members of $\mathcal{L}$. Consider the first, $\alpha\gamma$ (there are other choices that would work).

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 $\mathtt{a}^p$ 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 is the desired contradiction.

5.4 Remark In that example the string σ has three parts, $\sigma = \mathtt{a}^p \frown \mathtt{b} \frown \mathtt{a}^p$, and it decomposes into three parts, $\sigma = \alpha\frown\beta\frown\gamma$. Don't make the mistake of thinking that the two decompositions line up. The Pumping Lemma does not say that $\alpha = \mathtt{a}^p$, $\beta = \mathtt{b}$, and $\gamma = \mathtt{a}^p$ — indeed, we've shown that β does not contain the b. Instead the lemma's first condition only says that the first two substrings together, $\alpha\beta$, consists exclusively of a's. So perhaps $\alpha\beta = \mathtt{a}^p$, or perhaps γ starts with some a's that are then followed by $\mathtt{ba}^p$. That is, all we know is that $\alpha\beta$ matches the regular expression $\mathtt{a}\star$ while γ matches $\mathtt{a}\star\mathtt{baa}\ldots\mathtt{a}$, with p-many a's at the end.

5.5 Example Consider $\mathcal{L} = \{\mathtt{0}^m\mathtt{1}^n \in \mathbb{B}^* \bigm| m = n+1\} = \{\mathtt{0}, \mathtt{001}, \mathtt{00011}, \ldots\}$, whose members start with a number of 0's that is one more than the number of 1's at the end. 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 $\sigma = \mathtt{0}^{p+1}\mathtt{1}^p \in \mathcal{L}$. 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 (and also, all of σ's 1's are in γ). Condition (2) says that β has at least one character, necessarily 0. Consider Condition (3)'s list: $\alpha\gamma$, $\alpha\beta^2\gamma$, $\alpha\beta^3\gamma$, ... Compare its first entry, $\alpha\gamma$, to σ. The string $\alpha\gamma$ has fewer 0's then does σ but the same number of 1's. So the number of 0's in $\alpha\gamma$ is not one more than the number of 1's. Thus $\alpha\gamma \notin \mathcal{L}$, which contradicts the third condition of the Pumping Lemma.

We can interpret that example to say that Finite State machines cannot 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 $\mathcal{L} = \{\, \mathtt{a}^n \;\big|\; n \text{ is a perfect square} \,\} = \{\varepsilon, \mathtt{a}, \mathtt{a}^4, \mathtt{a}^9, \mathtt{a}^{16}, \ldots\}$ is not regular. For, suppose otherwise. Denote the pumping length by p and consider $\sigma = \mathtt{a}^{(p^2)}$, 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 $\alpha\gamma$, $\alpha\beta^2\gamma$, ...

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 $\mathcal{L} = \{\, \sigma \in \{\mathtt{a}, \mathtt{b}\}^* \;\big|\; \sigma \text{ has as many a's as b's} \,\}$ is not regular. To prove that, observe that the language $\hat{\mathcal{L}} = \{\, \mathtt{a}^m\mathtt{b}^n \in \{\mathtt{a}, \mathtt{b}\}^* \;\big|\; m, n \in \mathbb{N} \,\}$ is regular, described by the regular expression $\mathtt{a}\star\mathtt{b}\star$. Recall that the intersection of two regular languages is regular. But $\mathcal{L} \cap \hat{\mathcal{L}}$ is the set $\{\, \mathtt{a}^n\mathtt{b}^n \;\big|\; n \in \mathbb{N} \,\}$ 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 $\mathcal{L} = \{\, \mathtt{0}^m\mathtt{1}^n \in \mathbb{B}^* \;\big|\; m = n + 1 \,\}$ is not regular but your friend doesn't get it and asks you, "What's wrong with the regular expression $\mathtt{0}^{n+1}\mathtt{1}^n$?" 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. But the Pumping Lemmas third condition requires that infinitely many strings be in the language, 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 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 by using the Pumping Lemma.
(A) $\mathcal{L}_0 = \{ \mathtt{a}^n\mathtt{b}^m \mid n+2=m \}$
(B) $\mathcal{L}_1 = \{ \mathtt{a}^n\mathtt{b}^m\mathtt{c}^n \mid n,m \in \mathbb{N} \}$
(C) $\mathcal{L}_2 = \{ \mathtt{a}^n\mathtt{b}^m \mid n < m \}$

✓ 5.14 For each language over $\Sigma = \{\mathtt{a},\mathtt{b}\}$ produce five strings that are members. Then decide whether that language is regular. Prove your assertion either by producing a regular expression or using the Pumping Lemma.
(A) $\{ \mathtt{a}^n\mathtt{b}^m \in \Sigma^* \mid n = 3 \}$
(B) $\{ \mathtt{a}^n\mathtt{b}^m \in \Sigma^* \mid n+3=m \}$
(C) $\{ \mathtt{a}^n\mathtt{b}^m \in \Sigma^* \mid n,m \in \mathbb{N} \}$
(D) $\{ \mathtt{a}^n\mathtt{b}^m \in \Sigma^* \mid m-n > 12 \}$

✓ 5.15 Each language is non-regular and σ is a good choice as part of a proof using the Pumping Lemma, where p is the pumping length. For each, give the most specific regular expression describing $\alpha\beta$ and γ. Take $\Sigma = \{\mathtt{a},\mathtt{b}\}$.
(A) $\mathcal{L} = \{ \mathtt{a}^n\mathtt{b}^{2n} \mid n \in \mathbb{N} \}$, $\sigma = \mathtt{a}^p\mathtt{b}^{2p}$
(B) $\mathcal{L} = \{ \mathtt{a}^n\mathtt{b}^{n+5} \mid n \in \mathbb{N} \}$, $\sigma = \mathtt{a}^p\mathtt{b}^{p+5}$
(C) $\mathcal{L} = \{ \mathtt{a}^i\mathtt{b}^j\mathtt{a}^{i+j} \mid i \in \mathbb{N} \text{ and } j \in \mathbb{N}^+ \}$, $\sigma = \mathtt{a}^p\mathtt{b}\mathtt{a}^{p+1}$
(D) $\mathcal{L} = \{ \mathtt{a}^i \frown \tau \mid k \in \mathbb{N} \text{ and } |\tau| = k \}$, $\sigma = \mathtt{a}^p\mathtt{b}^p$
(E) $\mathcal{L} = \{ \tau \mid \tau \text{ is a palindrome and } |\tau| \text{ is even} \}$, $\sigma = \mathtt{a}^p\mathtt{b}\mathtt{b}\mathtt{a}^p$

5.16 With a friend you try to apply the Pumping Lemma to $\{ \tau \frown \tau \mid \tau \in \Sigma^* \}$.
(A) List five elements of the language.
(B) You pick $\sigma = \mathtt{a}^p\mathtt{b}\mathtt{a}^p\mathtt{b}$; go through the argument.
(C) Your friend tries $\sigma = \mathtt{a}^p\mathtt{a}^p$ and can't get their argument to go. Suggestions?

✓ 5.17 Use the Pumping Lemma to prove that $\mathcal{L} = \{ \mathtt{a}^{m-1}\mathtt{c}\mathtt{b}^m \mid m \in \mathbb{N}^+ \}$ is not regular. It may help to first produce five strings from the language.

5.18 Show that the language over $\{\mathtt{a},\mathtt{b}\}$ of strings having more a's than b's is not regular.

5.19 One of these is regular, one is not. Which is which? (Prove your assertions.)
(A) $\{ \mathtt{a}^n\mathtt{b}^m \in \{\mathtt{a},\mathtt{b}\}^* \mid n = m^2 \}$
(B) $\{ \mathtt{a}^n\mathtt{b}^m \in \{\mathtt{a},\mathtt{b}\}^* \mid 3 < m,n \}$

5.20 Is $\{ \sigma \in \mathbb{B}^* \mid \sigma = \alpha\beta\alpha^{\mathsf{R}} \text{ for } \alpha,\beta \in \mathbb{B}^* \}$ regular? Either way, prove it.

5.21 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.

5.22 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? (Prove the assertions.)

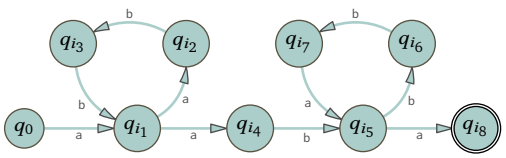
✓ 5.23 Show that there is a Finite State machine that recognizes this language of all sums totaling less than four, $\mathcal{L}_4 = \{a^i b^j c^k \mid i, j, k \in \mathbb{N} \text{ and } i + j = k \text{ and } k < 4\}$. Use the Pumping Lemma to show that no Finite State machine recognizes the language of all sums, $\mathcal{L} = \{a^i b^j c^k \mid i, j, k \in \mathbb{N} \text{ and } i + j = k\}$.

5.24 Decide if each is a regular language of bitstrings:
(A) the number of 0's plus the number of 1's equals five,
(B) the number of 0's minus the number of 1's equals five.

✓ 5.25 Show that $\{0^m 1^n \in \mathbb{B}^* \mid m \neq n\}$ is not regular. *Hint:* use the closure properties of regular languages and consider the set $\{0^m 1^m \in \mathbb{B}^* \mid m \in \mathbb{N}\}$.

5.26 Example 5.7 shows that $\{\sigma \in \{a, b\}^* \mid \sigma \text{ has as many a's as b's}\}$ is not regular. In contrast, show that $\mathcal{L} = \{\sigma \in \{a, b\}^* \mid \sigma \text{ has as many ab's as ba's}\}$ is regular. *Hint:* think of ab and ba as marking a transition from a block of one character to a block of another. For instance, this string has three such transitions aaa b aaa bbb (and is not in a member of $\mathcal{L}$).

✓ 5.27 Rebut someone who says to you, "Sure, for the machine before Theorem 5.1 on page 220, 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."



5.28 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 difference in string lengths.

5.29 Consider $\{a^i b^j c^{i \cdot j} \mid i, j \in \mathbb{N}\}$.
(A) Give five strings from this language.
(B) Show that it is not regular.

5.30 There are non-regular languages that the Pumping Lemma will not prove are non-regular. Take $\Sigma = \{a, b, c\}$ and let $\mathcal{L}_0 = \{ab^n c^n \mid n \in \mathbb{N}\}$ and $\mathcal{L}_1 = \{a^k{}^\frown \tau \mid k \neq 1 \text{ and } \tau \in \{b, c\}^*\}$. We will show that $\mathcal{L} = \mathcal{L}_0 \cup \mathcal{L}_1$ is not regular, although it satisfies the Pumping Lemma in that it has a pumping length $p \in \mathbb{N}$ such that every string $\sigma \in \mathcal{L}$ with $|\sigma| \geq p$ decomposes as $\sigma = \alpha\beta\gamma$, subject to the familiar three conditions.
(A) Show that $\mathcal{L}_1$ is regular.
(B) Show that $\mathcal{L}_0$ is not regular.
(C) Conclude from the prior two that $\mathcal{L}$ is not regular.

(D) Fix $p = 3$. For a string of the form $\sigma = \mathtt{a}\mathtt{b}^n\mathtt{c}^n$ find a decomposition $\sigma = \alpha\beta\gamma$ such that $|\alpha\beta| \leq p$ and $|\beta| > 0$ and every string in $\alpha\gamma, \alpha\beta^2\gamma, \ldots$ is also a member of $\mathcal{L}$.

(E) Again taking $p = 3$, verify the same for a string starting with zero-many a's that is, of the form $\sigma = \tau \in \{\mathtt{b}, \mathtt{c}\}^*$.

(F) Verify it also for a string of the form $\sigma = \mathtt{a}^k \frown \tau$ for $k \geq 2$.

5.31 The proof of the Pumping Lemma shows that where a Finite State machine recognizes a language, the number of states in the machine suffices as a pumping length for that language. But p can be smaller than that.

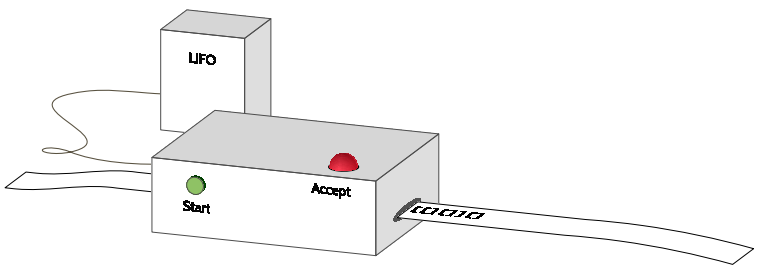
(A) Consider the language $\mathcal{L}$ described by `(01)*`. Construct a deterministic Finite State machine with three states that recognizes this language and argue that this is the minimal number of states for such a machine.

(B) Show that the minimal pumping length for $\mathcal{L}$ is 1.

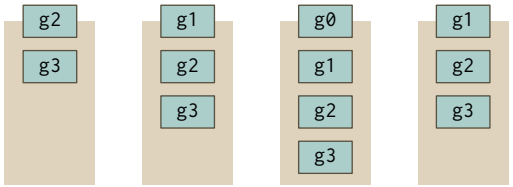
Section IV.6 Pushdown machines

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, `g3` and `g2`. We push `g1` on the stack, and then `g0`. Now, although `g1` is on the stack, we don't have immediate access to it. To get at `g1` we must first pop off `g0`, 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 `g0` but as there are finitely many states and unboundedly many stack arrangements, that strategy has limits.

Definition We will extend the definition of Finite State machines by adding a stack. This stack holds tokens from an alphabet, $\Gamma = \{\texttt{g0}, \texttt{g1}, \ldots\}$ We reserve 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.

Before the definition we will give an example.

6.1 Example We will give a Pushdown machine that recognizes the language of balanced parentheses, $\mathcal{L}_{\text{BAL}}$, containing strings such as `[]` and `[[]]`, as well as `[[][]]` and `[][]`. Precisely stated, $\sigma \in \mathcal{L}_{\text{BAL}}$ if it contains the same number of `[`'s as `]`'s and no prefix of σ contains more `]`'s than `[`'s.

The Pumping Lemma shows that no Finite State machine recognizes $\mathcal{L}_{\text{BAL}}$. But it is recognized by this Pushdown machine. It has two states $Q = \{q_0, q_1\}$, one of which is an accepting state, $F = \{q_1\}$. Its tape alphabet is $\Sigma = \{\texttt{[}, \texttt{]}\}$ and its stack alphabet is $\Gamma = \{\texttt{g0}\}$. The table below gives Δ. Instruction numbers are for ease of reference.

Instruction number	*Input*	*Output*
0	q_0, `[`, $\perp$	q_0, 'g0 $\perp$'
1	q_0, `[`, `g0`	q_0, 'g0 g0'
2	q_0, `]`, `g0`	q_0, ''
3	q_0, `]`, $\perp$	q_0, ''
4	q_0, `B`, $\perp$	q_1, '$\perp$'
5	q_0, `B`, `g0`	q_0, 'g0'

In a Pushdown machine every computation step begins with the machine popping the top character off the stack. There that character is $\perp$. The machine is then in state q_0, is reading `[` on the tape, and the popped character is $\perp$, so instruction 0 applies. The machine goes into state q_0 (which is not a change) and pushes the

[†] 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.

two-token string g0⊥ onto the stack. The ⊥ only replaces what was there already, but the g0 makes a new stack top character.

Here is an example computation accepting the string [[]][].

Step	*Configuration*		
0	[[]][]B	⊥	
	q_0		
1	[]][]B	g0 ⊥	
	q_0		
2	]][]B	g0 g0 ⊥	
	q_0		
3	][]B	g0 ⊥	
	q_0		
4	[]B	⊥	
	q_0		
5	]B	g0 ⊥	
	q_0		
6	B	⊥	
	q_0		
7		⊥	
	q_1		

After step 1 there are two g0'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	[]]B	⊥	
	q_0		
1	]]B	g0 ⊥	
	q_0		
2	]B	⊥	
	q_0		
3	B		
	q_0		

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.

6.2 DEFINITION A nondeterministic Pushdown machine $\langle Q, q_0, F, \Sigma, \Gamma, \Delta\rangle$ consists of a finite set of states $Q = \{q_0, \ldots q_{n-1}\}$, 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 $\Delta\colon Q \times (\Sigma \cup \{\mathtt{B}, \varepsilon\}) \times (\Gamma \cup \{\bot\}) \to \mathscr{P}\big(Q \times (\Gamma \cup \{\bot\})^*\big)$.

A configuration $\mathcal{C}$ 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 $t \in \Sigma \cup \{\mathtt{B}, \varepsilon\}$, $g \in \Gamma \cup \{\bot\}$, and $\gamma \in \Gamma^*$.

The starting configuration $\mathcal{C}_0$ has the machine in state q_0, with a stack containing only the $\bot$ character, and with the read head at the first character of $\tau_0 \mathbin{\frown} \mathtt{B}$ where $\tau_0 \in \Sigma^*$ is the input string.

As for actions, suppose that the machine's configuration $\mathcal{C}_s$ has it in state q_i with the tape head reading t. If there is nothing on the stack, including not even $\bot$, then the computation dead-ends — there is no configuration $\mathcal{C}$ such that $\mathcal{C}_s \vdash \mathcal{C}$ and the 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 $\mathcal{C}_{s+1}$ so that $\mathcal{C}_s \vdash \mathcal{C}_{s+1}$, 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 $\gamma = \langle g_0, \ldots g_m\rangle$ onto the stack in the order that leaves g_0 now at the stack top. (2) If $t = \varepsilon$ (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 $\mathtt{B}$. The machine accepts its initial string τ_0 if at that point it is in a final state. Otherwise it rejects τ_0.

6.3 EXAMPLE Recall that a palindrome is a string that reads the same forwards and backwards, $\sigma = \sigma^{\mathsf{R}}$. There are two cases. An even-length palindrome such as abba breaks into halves, $\mathtt{abba} = \mathtt{ab} \mathbin{\frown} \mathtt{ba}$, where the suffix is the reverse of the prefix. But odd-length palindromes such as abcba have a character in the middle that acts as a pivot. That is, they have the form $\sigma \mathbin{\frown} s \mathbin{\frown} \sigma^{\mathsf{R}}$ for $s \in \Sigma$. The language $\mathcal{L}_{\text{MM}}$ uses $\sigma \in \{\mathtt{a}, \mathtt{b}\}^*$ along with the character $s = \mathtt{c}$ as a middle marker so that $\mathcal{L}_{\text{MM}} = \{\sigma \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* \mid \sigma = \tau \mathbin{\frown} \mathtt{c} \mathbin{\frown} \tau^{\mathsf{R}} \text{ for some } \tau \in \{\mathtt{a}, \mathtt{b}\}^*\}$.

The machine below accepts this language. It has $Q = \{q_0, q_1, q_2\}$, $F = \{q_2\}$, $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, and $\Gamma = \{\mathtt{g0}, \mathtt{g1}\}$.

Inst	*Input*	*Output*
0	q_0, a, $\perp$	q_0, ‘g0 $\perp$’
1	q_0, b, $\perp$	q_0, ‘g1 $\perp$’
2	q_0, c, $\perp$	q_1, ‘$\perp$’
3	q_0, B, $\perp$	q_3, ‘$\perp$’
4	q_0, a, g0	q_0, ‘g0 g0’
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’

Inst	*Input*	*Output*
8	q_0, c, g0	q_1, ‘g0’
9	q_0, c, g1	q_1, ‘g1’
10	q_1, a, g0	q_1, ‘’
11	q_1, b, g1	q_1, ‘’
12	q_1, B, $\perp$	q_2, ‘$\perp$’

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.

Step	*Configuration*	
0	b a c a b B	$\perp$
	q_0	
1	a c a b B	g1 $\perp$
	q_0	
2	c a b B	g0 g1 $\perp$
	q_0	
3	a b B	g0 g1 $\perp$
	q_1	
4	b B	g1 $\perp$
	q_1	
5	B	$\perp$
	q_1	
6		$\perp$
	q_3	

Our final example makes essential use of guessing, by relying on ε transitions.

6.4 Example Consider the language of all even-length palindromes over $\mathbb{B}^*$, $\mathcal{L}_{\text{ELP}} = \{\sigma\sigma^{\mathsf{R}} \mid \sigma \in \mathbb{B}^*\} = \{\varepsilon, 00, 11, 0000, 0110, 1001, 1111, \ldots\}$. 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 $\Gamma = \{\mathtt{g0}, \mathtt{g1}\}$.

Inst	*Input*	*Output*
0	$q_0, \mathtt{0}, \bot$	q_0, ‘g0⊥’
1	$q_0, \mathtt{1}, \bot$	q_0, ‘g1⊥’
2	$q_0, \varepsilon, \bot$	$q_1, \bot$
3	$q_0, \mathtt{0}, \mathtt{g0}$	q_0, ‘g0g0’
4	$q_0, \mathtt{0}, \mathtt{g1}$	q_0, ‘g0g1’
5	$q_0, \mathtt{1}, \mathtt{g0}$	q_0, ‘g1g0’
6	$q_0, \mathtt{1}, \mathtt{g1}$	q_0, ‘g1g1’
7	$q_0, \varepsilon, \mathtt{g0}$	q_1, ‘g0’
8	$q_0, \varepsilon, \mathtt{g1}$	q_1, ‘g1’
9	$q_1, \mathtt{0}, \mathtt{g0}$	q_1, ‘’
10	$q_1, \mathtt{1}, \mathtt{g1}$	q_1, ‘’
11	$q_1, \varepsilon, \bot$	$q_2, \bot$

The machine runs in two phases. Where the input is $\sigma\sigma^{\mathsf{R}}$, the first phase works with σ. If the tape character is `0` 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 `0` 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 path 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.

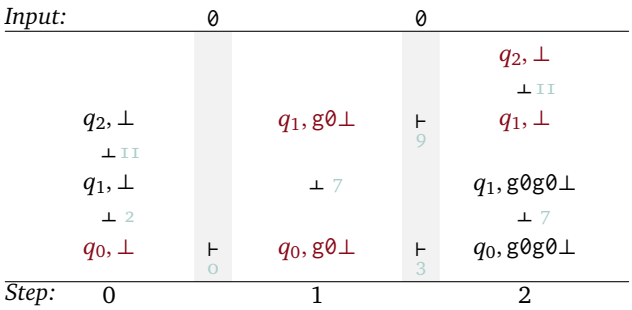
We will show three example computations. For the first, we exhibit a successful maximal path of the computation tree.

Step	*Configuration*	
0	`0 1 1 0`	⊥
	q_0	
1	`1 1 0`	`g0` ⊥
	q_0	
2	`1 0`	`g1 g0` ⊥
	q_1	
3	`0`	`g0` ⊥
	q_1	
4		⊥
	q_2	

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 $\bot$ 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 path succeeds.) So we can omit the terminating `B` that we used earlier.

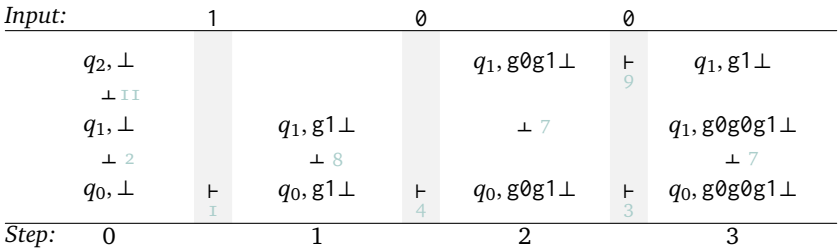
Next is the computation for input `00`. The picture below shows the entire computation tree. The ε transitions are drawn vertically (note the difference between the vertical ‘⊢’ and the bottom symbol). The machine accepts the input

because the highlighted maximal path ends with an empty tape and in the accepting state q_2.



6.5 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 maximal paths end both with an empty string and in an accepting state.



6.6 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.

6.7 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.

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.

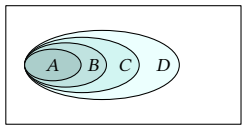
The last result needs deterministic Pushdown machines so we first outline how to define them. In contradistinction to a nondeterministic machine, in a deterministic machine at any step there is exactly one legal move. So to adjust the definition we have for nondeterministic Pushdown machines to one for 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 $\Delta(q_i, \varepsilon, g)$ is a nonempty set and also that $\Delta(q_i, t, g)$ is nonempty for $t \neq \varepsilon$ (see for instance the the prior example's machine in lines 0–2). So we outlaw the possibility that both are nonempty. Example 6.1 and Example 6.3 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

[†] 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.

rely on. In short, no deterministic Pushdown machine accepts $\mathcal{L}_{\text{ELP}}$. So Pushdown machines are different than Finite State machines and Turing machines — for Pushdown machines, nondeterminism changes what can be done.

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.



Class	*Machine type*
A	Finite State, including nondeterministic
B	Deterministic Pushdown
C	Nondeterministic Pushdown
D	Turing

IV.6 Exercises

✓ 6.8 Produce a Pushdown machine that does not halt.

6.9 Consider the Pushdown machine in Example 6.1.
(A) With the input `[][]`, step through the computation as a sequence of $\vdash$ relations.
(B) Do the same but with the input `][][`.

✓ 6.10 Produce a Pushdown machine to accept each language over $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$.
(A) $\{\mathsf{a}^n\mathsf{c}\mathsf{b}^{2n} \mid n \in \mathbb{N}\}$ (B) $\{\mathsf{a}^n\mathsf{c}\mathsf{b}^{n-1} \mid n > 0\}$

✓ 6.11 Give a Pushdown machine that accepts $\{\mathtt{0}^\frown \tau^\frown \mathtt{1} \mid \tau \in \mathbb{B}^*\}$.

✓ 6.12 Write a Pushdown machine that accepts $\{\mathsf{a}^{2n} \mid n \in \mathbb{N}\}$.

6.13 Give a Pushdown machine that accepts $\{\mathsf{a}^{2n}\mathsf{b}^n \mid n \in \mathbb{N}\}$.

✓ 6.14 Example 6.4 discusses the view of a nondeterministic computation as a tree. Draw the tree for that machine these inputs. (A) `0110` (B) `010`

✓ 6.15 Give a grammar for the language in Example 6.4, the even-length palindromes over $\mathbb{B}$.

6.16 Use the Pumping Lemma to show that the language of even-length palindromes from Example 6.4 is not recognized by any Finite State machine.

6.17 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.

6.18 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, a system administrator searching a web server log for the PDF's downloaded from a directory. They might give this command.

```
$ grep "/linearalgebra/.*\textbackslash .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 then `grep` prints that line.

We will illustrate with Racket regexes. As a prototype,

```
> (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. These 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.

Courtesy `xkcd.com`

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.

[†] 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.[†]

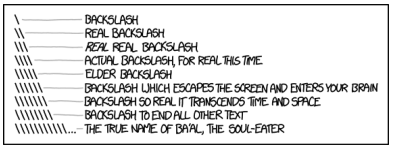
1.1 Example Canadian postal codes have seven characters: the fourth is a space, the first, third, and sixth are letters, and the others are digits. The regular expression `[a-zA-Z]\d[a-zA-Z] \d[a-zA-Z]\d` describes them.

1.2 Example Dates are often given in the 'dd/mm/yy' format. This matches: `\d\d/\d\d/\d\d`.

1.3 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 ε|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.

1.4 Example US postal codes, called ZIP codes, are five digits. Match them with `\d{5}`.

1.5 Example North American phone numbers match `\d{3} \d{3}-\d{4}`.

1.6 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.

1.7 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]+`.

1.8 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*`.

1.9 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,}`.

1.10 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.

1.11 Example Match the text inside a single set of parentheses with `\([^()]*\)`.

1.12 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 '`$`'.

1.13 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.

1.14 Example The web document language HTML document tag for an image, such as `<img src="logo.jpg">`, 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.

1.15 Example The language HTML uses tags such as `<b>boldface text</b>` and `<i>italicized text</i>`. Matching any one tag is straightforward, for instance `<b>[^<]*</b>`. 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 n alternation operator. 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.

1.16 Example This is the language of square strings over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

$$\mathcal{L} = \{\sigma \in \Sigma^* \;\big|\; \sigma = \tau^\frown \tau \text{ for some } \tau \in \Sigma^*\}$$

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.

1.17 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, $\mathcal{L} = \{1^n \;\big|\; n \text{ is not prime}\}$ 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 1.3 `(ε|0|1)\d:\d\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 are given without them. We could change the ending to `(`ε`|\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 1.1. 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.

Courtesy xkcd.com

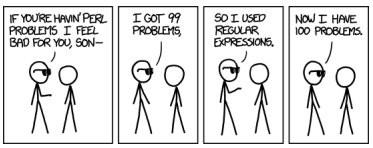
The canonical example of this is the regex describing the official standard for valid email addresses. We show here just five lines out of its 81 but that's enough to make the point about its complexity.

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(
?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\0
```

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 1.5 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 `(`ε`|\s am|\s pm)`. Why not `\s(`ε`|am|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 it 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 MDCCLXXVI, for 1776. Further, we replace IIII with IV, and replace VIIII with IX. Give a regular expression for valid Roman numerals less than 5000.

A.36 Example 1.16 says that $\mathcal{L} = \{\sigma \in \Sigma^* \mid \sigma = \tau^\frown\tau \text{ for some } \tau \in \Sigma^*\}$, the language of square strings over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, is not regular. Verify that.

A.37 Consider $\mathcal{L} = \{\mathtt{0}^n\mathtt{1}\mathtt{0}^n \mid n > 0\}$. (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.

(A)

	[^SPEAK]+	EP|IP|EF
HE|LL|O+		
[PLEASE]+		

(B)

	(A|B|C)\1	(AB|OE|SK)
.*M?O.*		
(AN|FE|BE)		

EXTRA

IV.B The Myhill-Nerode theorem

We have defined regular languages in terms of Finite State machines. Here we will give a characterization that instead goes directly to the properties of the language.

Recall that in this chapter's first section, Remark 1.7 said that the key to designing Finite State machines is to think of each state as about its future, as about the input strings to come.

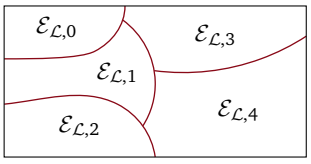
2.1 DEFINITION Fix a language $\mathcal{L}$ over an alphabet Σ, along with two strings $\sigma_0, \sigma_1 \in \Sigma^*$. Then $\tau \in \Sigma^*$ is a distinguishing extension when of $\sigma_0 \frown \tau$ and $\sigma_1 \frown \tau$, one is an element of $\mathcal{L}$ and the other is not. If such an extension exists then the strings are $\mathcal{L}$-distinguishable, otherwise they are $\mathcal{L}$-indistinguishable or $\mathcal{L}$-related, denoted $\sigma_0 \sim_{\mathcal{L}} \sigma_1$.

2.2 LEMMA For any language $\mathcal{L}$, the binary relation $\sim_{\mathcal{L}}$ is an equivalence and hence partitions the universe of all strings into equivalence classes, denoted $\mathcal{E}_{\mathcal{L},j}$.

Proof This is Exercise B.31's item (A). □

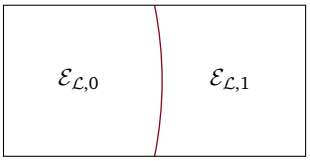
2.3 EXAMPLE Let $\mathcal{L} = \{\sigma \in \{\mathtt{a},\mathtt{b}\}^* \mid |\sigma| = 3\}$, with $\sigma_0 = \mathtt{aa}$ and $\sigma_1 = \mathtt{a}$.[†] Then $\tau = \mathtt{bb}$ is a distinguishing extension because $\sigma_0 \frown \tau = \mathtt{aabb} \notin \mathcal{L}$ while $\sigma_1 \frown \tau = \mathtt{abb} \in \mathcal{L}$.

The prior paragraph brings out that for this language two strings are $\mathcal{L}$-distinguishable if and only if they they have different lengths. So the equivalence classes, the collections of indistinguishable strings, are the length zero strings, $\mathcal{E}_{\mathcal{L},0} = \{\varepsilon\}$, and the length one strings, $\mathcal{E}_{\mathcal{L},1} = \{\mathtt{a},\mathtt{b}\}$, and those of length two, $\mathcal{E}_{\mathcal{L},2} = \{\mathtt{aa},\mathtt{ab},\mathtt{ba},\mathtt{bb}\}$, and length three, $\mathcal{E}_{\mathcal{L},3} = \{\mathtt{aaa},\mathtt{aab},\ \ldots\ \mathtt{bbb}\}$, along with the longer strings, $\mathcal{E}_{\mathcal{L},4} = \{\sigma \mid |\sigma| \geq 4\}$. In the picture below the box is the universe of all strings Σ^*. It is partitioned into the equivalence classes, with every string a member of one and only one class.



[†] Here, $\sigma_1 = \mathtt{a}$ is not a character, it is a length one string containing the character a. We won't worry too much about the distinction.

2.4 EXAMPLE Consider $\mathcal{L} = \{\,\sigma \in \{\mathtt{a},\mathtt{b}\}^* \;\big|\; \sigma \text{ has even length}\,\}$. Notice that if a string σ has even length then $\sigma^\frown\tau \in \mathcal{L}$ for an extension τ if and only if τ has even length. So there is no extension that distinguishes between even length strings. Similar reasoning holds for two odd-length strings. Thus the $\sim_\mathcal{L}$ relation breaks $\{\mathtt{a},\mathtt{b}\}^*$ into two parts, $\mathcal{E}_{\mathcal{L},0} = \{\,\sigma \;\big|\; |\sigma| \text{ is even}\,\}$ and $\mathcal{E}_{\mathcal{L},1} = \{\,\sigma \;\big|\; |\sigma| \text{ is odd}\,\}$. This shows the partition of the universe $\{\mathtt{a},\mathtt{b}\}^*$.



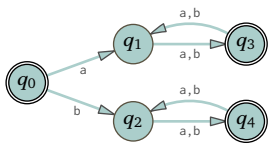
2.5 EXAMPLE The above examples have finitely many $\sim_\mathcal{L}$ classes. Some languages have infinitely many. One is $\mathcal{L} = \{\,\sigma \in \{\mathtt{a},\mathtt{b}\}^* \;\big|\; \sigma = \mathtt{a}^n\mathtt{b}^n \text{ for some } n \in \mathbb{N}\,\}$, from Example 5.2. To show that the number of classes is infinite we don't need to produce them all; it is enough to produce infinitely many unequal classes (or for that matter, infinitely many mutually distinguishable strings).

$$\mathcal{E}_{\mathcal{L},i_0} = \{\varepsilon\} \quad \mathcal{E}_{\mathcal{L},i_1} = \{\mathtt{a}\} \quad \mathcal{E}_{\mathcal{L},i_2} = \{\mathtt{aa}\} \quad \mathcal{E}_{\mathcal{L},i_3} = \{\mathtt{aaa}\} \quad \ldots$$

To verify that these singleton sets are equivalence classes, consider $\sigma = \mathtt{a}^n$. Since $\sigma^\frown\mathtt{b}^n \in \mathcal{L}$, the only candidates for strings that are $\mathcal{L}$-indistinguishable from σ but unequal to it have the form $\alpha = \mathtt{a}^{n+j}\mathtt{b}^j$ with $j > 0$. If $n = 0$ then $\sigma = \varepsilon$ and $\alpha = \mathtt{b}^j$, and an extension distinguishing σ from α is $\mathtt{ab}$. If $n > 0$ then an extension distinguishing $\sigma = \mathtt{a}^n$ from $\alpha = \mathtt{a}^{n+j}\mathtt{b}^j$ is $\mathtt{ab}^{n+1}$.

We next make a connection between the Finite State machines that recognize a language $\mathcal{L}$ and the relationship of $\mathcal{L}$-indistinguishability.

2.6 EXAMPLE This machine recognizes $\mathcal{L} = \{\,\sigma \in \{\mathtt{a},\mathtt{b}\}^* \;\big|\; \sigma \text{ has even length}\,\}$, the language of Example 2.4.



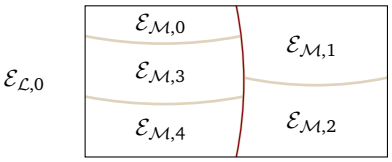
Consider other string inputs, not just the accepted ones, and see where they bring the machine.

Input string σ	ε	a	b	aa	ab	ba	bb	aaa	aab	aba	abb	...
Ending state $\hat{\Delta}(\sigma)$	q_0	q_1	q_2	q_3	q_3	q_4	q_4	q_1	q_1	q_1	q_1	...

The collection of input strings breaks into five sets, those that bring the machine to q_0, those that bring it to q_1, etc. This is another kind of partition, which will prove to be related to but different than the partition above. Denote the classes of this $\mathcal{M}$-related partition with $\mathcal{E}_{\mathcal{M},i} = \{\,\sigma \in \Sigma^* \;\big|\; \hat{\Delta}(\sigma) = q_i\,\}$.

$$\begin{aligned}\mathcal{E}_{\mathcal{M},0} &= \{\varepsilon\}\\ \mathcal{E}_{\mathcal{M},1} &= \{\mathtt{a}, \mathtt{aaa}, \ldots\}\\ \mathcal{E}_{\mathcal{M},2} &= \{\mathtt{b}, \mathtt{baa}, \ldots\}\\ \mathcal{E}_{\mathcal{M},3} &= \{\mathtt{aa}, \mathtt{ab}, \ldots\}\\ \mathcal{E}_{\mathcal{M},4} &= \{\mathtt{ba}, \mathtt{bb}, \ldots\}\end{aligned}$$

Below we lay the language-related partition, with two parts, on top of the machine-related one with five.



(∗)

The $\mathcal{M}$-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.[†]

2.7 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 $\hat{\Delta}(\sigma_0) = \hat{\Delta}(\sigma_1)$, that is, if starting the machine with input σ_0 ends with it in the same state as does starting the machine with input σ_1.

2.8 LEMMA The binary relation of $\mathcal{M}$-related is an equivalence and so partitions the collection of all strings Σ^* into equivalence classes.

Proof See Exercise B.31's item (B). □

2.9 LEMMA Let $\mathcal{M}$ be a deterministic 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. It follows that for any suffix τ, starting $\mathcal{M}$ with the input $\sigma_0 \frown \tau$ causes it to end in the same state as does starting it with the input $\sigma_1 \frown \tau$ (because the machine is deterministic). In particular, $\sigma_0 \frown \tau$ takes $\mathcal{M}$ to an accepting state if and only if $\sigma_1 \frown \tau$ does. So the two strings are $\mathcal{L}$-related. □

The $\mathcal{E}_{\mathcal{M},i}$ classes reflect $\mathcal{M}$'s states, in the following sense. Consider q_1's class $\mathcal{E}_{\mathcal{M},1} = \{\mathtt{a}, \mathtt{aaa}, \ldots\}$ and q_3's class $\mathcal{E}_{\mathcal{M},3} = \{\mathtt{aa}, \mathtt{ab}, \ldots\}$. Just as when the machine is in q_1 and reads an a then it transitions to q_3, so also if we choose any string $\sigma \in \mathcal{E}_{\mathcal{M},1}$ and append an a to it then $\sigma \frown \mathtt{a}$ is an element of $\mathcal{E}_{\mathcal{M},3}$. An example is that choosing $\sigma = \mathtt{a} \in \mathcal{E}_{\mathcal{M},1}$ and appending a to it gives $\mathtt{aa} \in \mathcal{E}_{\mathcal{M},3}$.

This way of thinking of the $\mathcal{M}$ classes has them acting as a machine, in that there are transitions among them just as a machine has. It suggests extending to also think of the $\mathcal{L}$ classes as constituting their own machine. In particular, the prior paragraph's workings of the transitions on the $\mathcal{E}_{\mathcal{M},i}$ classes suggest how to define the transitions in this new machine.

[†] 'Finer' in the sense that sand is finer than gravel.

2.10 DEFINITION Let $\mathcal{L}$ be a language over Σ and let the collection of $\sim_{\mathcal{L}}$ equivalence classes $\mathcal{E}_{\mathcal{L},i}$ be E. The $\mathcal{L}$-machine has states that are the classes, where the start state is the one containing ε. Its accepting states are the ones containing strings from $\mathcal{L}$. The transition operation, $\Delta\colon E\times\Sigma\to E$, is: given input $\mathcal{E}_{\mathcal{L},i}$ and $x\in\Sigma$, choose a σ in the class and then $\sigma^\frown x$ is an element of some $\mathcal{E}_{\mathcal{L},j}$. Set $\Delta(\mathcal{E}_{\mathcal{L},i},x)=\mathcal{E}_{\mathcal{L},j}$.

For instance, the machine for Example 2.6's two-class language is the two-state machine in (**) below.

As stated, the definition allows us to choose any string σ at all from $\mathcal{E}_{\mathcal{L},i}$. We must establish that choosing two different string representatives of the input class does not give two different outputs.

2.11 LEMMA Fix a language $\mathcal{L}$. (1) The transition operation is well-defined: if two strings σ_0,σ_1 are $\mathcal{L}$-related, $\sigma_0\sim_{\mathcal{L}}\sigma_1$, then adjoining a common character $x\in\Sigma$ gives strings that are also $\mathcal{L}$-related, $(\sigma_0{}^\frown x)\sim_{\mathcal{L}}(\sigma_1{}^\frown x)$. (2) If one member of a class is an element of $\mathcal{L}$ then every other member of that class is also an element of $\mathcal{L}$. (3) There is one and only one class containing the empty string.

Proof For the first item, if $\sigma_0{}^\frown x$ were not $\mathcal{L}$-related to $\sigma_1{}^\frown x$ then the single-character string x would be a distinguishing extension. But $\sigma_0\sim_{\mathcal{L}}\sigma_1$ so they have no distinguishing extension.

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.

The third item is trivial since for any string, empty or not, there is one and only one equivalence class containing that string. □

2.12 COROLLARY The $\mathcal{L}$-machine, if it has finitely many states, is a Finite State machine that recognizes $\mathcal{L}$.

Proof By well-definedness of transitions, starting the $\mathcal{L}$-machine with any $\sigma\in\Sigma^*$ as input will cause the machine to end in the class containing σ. By the lemma's item (2), that is an accepting class of the machine if and only if $\sigma\in\mathcal{L}$. □

2.13 EXAMPLE Let $\mathcal{L}=\{\mathtt{ab}^n\mid n\in\mathbb{N}\}=\{\mathtt{a},\mathtt{ab},\mathtt{abb},\ldots\}$. We will first find the equivalence classes and then determine the $\mathcal{L}$-machine's transitions.

There are three classes. First, $\mathcal{E}_{\mathcal{L},0}=\{\varepsilon\}$ because for any string $\sigma\in\{\mathtt{a},\mathtt{b}\}^*$, a distinguishing extension between ε and σ is the single-character string $\mathtt{a}$.

The second class is $\mathcal{E}_{\mathcal{L},1}=\mathcal{L}$. To see that any two elements of this set, $\mathtt{ab}^i,\mathtt{ab}^j\in\mathcal{L}$, are $\mathcal{L}$-related, suppose that $\tau\in\Sigma^*$. If τ has the form $\mathtt{b}^k$ then both of $\mathtt{ab}^i{}^\frown\tau$ and $\mathtt{ab}^j{}^\frown\tau$ are members of $\mathcal{L}$, while if τ has at least one $\mathtt{a}$ then both are not members because they have at least two $\mathtt{a}$'s. It remains to show that if $\sigma_0\in\mathcal{L}$ and $\sigma_1\notin\mathcal{L}$ then they are not $\mathcal{L}$-related. That's because, as in Lemma 2.11's item (2), they have a distinguishing extension of ε.

The final class contains all remaining strings, $\mathcal{E}_{\mathcal{L},2}=\Sigma^*-(\mathcal{E}_{\mathcal{L},0}\cup\mathcal{E}_{\mathcal{L},1})$. We will show that any two elements of this set are $\mathcal{L}$-related (there is no need to argue that elements are not $\mathcal{L}$-related to strings outside the set because we have already shown that in the prior paragraphs). An element $\sigma\in\mathcal{E}_{\mathcal{L},2}$ must have at least one

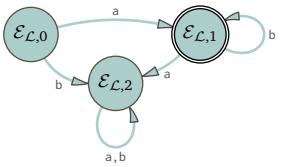
character and must fall into at least one of two cases: either its first item is not a, or the rest of the string contains at least one a. In both cases for any extension τ the string $\sigma^\frown\tau$ that is not an element of $\mathcal{L}$, that is, there are no distinguishing extensions.

In summary, the universe of all strings is partitioned by $\sim_\mathcal{L}$ into these classes.

$$\mathcal{E}_{\mathcal{L},0} = \{\,\varepsilon\,\} \quad \mathcal{E}_{\mathcal{L},1} = \{\,\mathsf{a},\mathsf{ab},\mathsf{abb},\ldots\,\} \quad \mathcal{E}_{\mathcal{L},2} = \{\,\mathsf{b},\mathsf{aa},\mathsf{ba},\mathsf{bb},\mathsf{aaa},\mathsf{aba},\ldots\,\}$$

To compute the transitions, for each class choose one representative element, append in turn each of a and b, and find the resulting classes. As representatives, besides $\varepsilon \in \mathcal{E}_{\mathcal{L},0}$ we can choose the one-character strings $\mathsf{a} \in \mathcal{E}_{\mathcal{L},1}$ and $\mathsf{b} \in \mathcal{E}_{\mathcal{L},2}$.

Δ	a	b
$\varepsilon \in \mathcal{E}_{\mathcal{L},0}$	$\mathsf{a} \in \mathcal{E}_{\mathcal{L},1}$	$\mathsf{b} \in \mathcal{E}_{\mathcal{L},2}$
$+\,\mathsf{a} \in \mathcal{E}_{\mathcal{L},1}$	$\mathsf{aa} \in \mathcal{E}_{\mathcal{L},2}$	$\mathsf{ab} \in \mathcal{E}_{\mathcal{L},1}$
$\mathsf{b} \in \mathcal{E}_{\mathcal{L},2}$	$\mathsf{ba} \in \mathcal{E}_{\mathcal{L},2}$	$\mathsf{bb} \in \mathcal{E}_{\mathcal{L},2}$

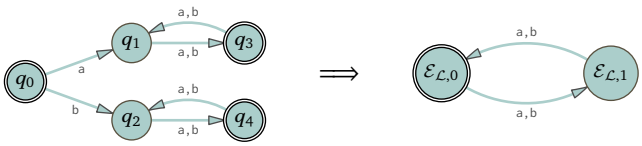


B.14 **Theorem (Myhill-Nerode)** A language $\mathcal{L}$ is regular if and only if the relation $\sim_\mathcal{L}$ gives only finitely many equivalence classes.

Proof For the first direction suppose that $\mathcal{L}$ is a regular language, recognized by a Finite State machine $\mathcal{M}$. The number of elements in the partition associated with $\sim_\mathcal{M}$ is finite, as there is one part for each of $\mathcal{M}$'s reachable states. Consequently the number of element in the partition induced by $\sim_\mathcal{L}$ is finite because by Lemma 2.9, $\sim_\mathcal{L}$'s partition has at most the number of elements that $\sim_\mathcal{M}$'s partition has (as illustrated in Example 2.6's $(*)$ diagram).

For the other direction let the number of elements in the partition induced by $\sim_\mathcal{L}$ be finite. By the corollary above, the $\mathcal{L}$ machine is a Finite State machine that recognizes $\mathcal{L}$. Since there is a Finite Sate machine recognizing it, $\mathcal{L}$ is regular. □

Returning to Example 2.6, the way that the $\mathcal{L}$ classes fit the $\mathcal{M}$ classes in diagram $(*)$ suggests that we are collapsing the $\mathcal{M}$ classes to get the $\mathcal{L}$ machine, and that machine also recognizes the language $\mathcal{L}$, but does so with a minimal number of states.

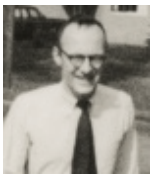
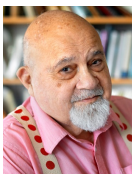


$(**)$

2.15 **Lemma** The $\mathcal{L}$-machine is minimal, meaning that from among all the deterministic Finite State machines that recognize the language $\mathcal{L}$, it has a minimal number of states.

Proof Let a deterministic Finite State machine $\mathcal{M}$ recognize $\mathcal{L}$. Consider two strings $\sigma_0, \sigma_1 \in \Sigma^*$. If those two take $\mathcal{M}$ to the same state, $\hat{\Delta}(\sigma_0) = \hat{\Delta}(\sigma_1)$, then appending any common extension does the same, $\hat{\Delta}(\sigma_0{}^\frown\tau) = \hat{\Delta}(\sigma_1{}^\frown\tau)$.

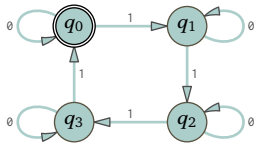
Thus if the two have a distinguishing extension, $\sigma_0 \not\sim_{\mathcal{L}} \sigma_1$, then $\hat{\Delta}(\sigma_0) \neq \hat{\Delta}(\sigma_1)$. Consequently, if there are k-many $\mathcal{E}_{\mathcal{L},i}$ classes then there must be a set of k-many strings that pairwise end in different states. Hence any machine recognizing $\mathcal{L}$ must have at least that many different states. □

John Myhill Sr 1923–1987
and Anil Nerode b 1932

To analyze a language with the Myhill-Nerode theorem, the challenge lies in finding the $\sim_{\mathcal{L}}$ equivalence classes. One helpful point is that Lemma 2.11's item (2) says that if a class contains even one string from $\mathcal{L}$ then it consists entirely of such strings (it may be that more than one class contains strings from $\mathcal{L}$). Another point is that if the given language is recognized by some Finite State machine then with our experience often we see how to build such an $\mathcal{M}$. It can give us insight into the $\sim_{\mathcal{L}}$ classes, since Lemma 2.9 says that the $\sim_{\mathcal{L}}$ classes are groupings of $\sim_{\mathcal{M}}$ classes. Also, if the $\mathcal{M}$ that we build is minimal then by Lemma 2.15 the $\sim_{\mathcal{L}}$ classes equal the $\sim_{\mathcal{M}}$ classes.

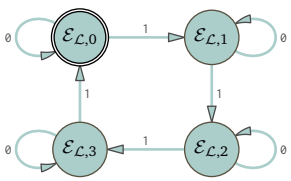
2.16 Example In this chapter's first section, Example 1.4 discusses the language of strings $\sigma \in \mathbb{B}^*$ where the number of 1's is a multiple of four. It recognizes $\mathcal{L}$ using this machine, which is clearly minimal. (We could verify that using Extra C.)



There are four $\mathcal{M}$ equivalence classes, $\mathcal{E}_{\mathcal{M},0} = \{\sigma \mid \hat{\Delta}(\sigma) = q_0\}$, and $\mathcal{E}_{\mathcal{M},1} = \{\sigma \mid \hat{\Delta}(\sigma) = q_1\}$, etc. These are also the $\mathcal{L}$ equivalence classes, $\mathcal{E}_{\mathcal{L},0} = \mathcal{E}_{\mathcal{M},0}$, etc. The class $\mathcal{E}_{\mathcal{L},0}$ contains the strings where the number of 1's is divisible by four without remainder, the class $\mathcal{E}_{\mathcal{L},1}$ contains the strings where the number of 1's leaves a remainder of 1, etc.

For the $\mathcal{L}$-machine, compute the transitions by choosing a representative element from each class, appending in turn each character of the alphabet, and finding the resulting class.

Δ	0	1
$+\,\varepsilon \in \mathcal{E}_{\mathcal{L},0}$	$0 \in \mathcal{E}_{\mathcal{L},0}$	$1 \in \mathcal{E}_{\mathcal{L},1}$
$1 \in \mathcal{E}_{\mathcal{L},1}$	$10 \in \mathcal{E}_{\mathcal{L},1}$	$11 \in \mathcal{E}_{\mathcal{L},2}$
$11 \in \mathcal{E}_{\mathcal{L},2}$	$110 \in \mathcal{E}_{\mathcal{L},2}$	$111 \in \mathcal{E}_{\mathcal{L},3}$
$111 \in \mathcal{E}_{\mathcal{L},3}$	$1110 \in \mathcal{E}_{\mathcal{L},3}$	$1111 \in \mathcal{E}_{\mathcal{L},0}$



Of course, the two machines are essentially the same. If we minimize a minimal machine then we get back pretty much what we started with.

We finish with a reflection. At the chapter's start we constructed machines by hand, perhaps sometimes struggling a bit. In contrast, the Myhill-Nerode result

starts with the desired language $\mathcal{L}$, considers the relation $\sim_{\mathcal{L}}$, and if there are finitely many associated classes then gives a Finite State machine that recognizes $\mathcal{L}$.

That is, by seeing the patterns inside $\mathcal{L}$ given by $\sim_{\mathcal{L}}$, we don't have to make a machine recognizing that language, the mathematics gives it to us. For free, this machine is minimal. This is a deep kind of wizardry — these problems are solved automatically.

IV.B Exercises

✓ B.17 Use the machine from Example 2.13. (A) What class contains the string bba? (B) abba? (C) babab? (D) abbbb?

B.18 Use the $\mathcal{L}$-machine from Example 2.13. For each string, identify the string's class, and then append the character a and identify the ending class for the machine's transition. (A) bba (B) ε (C) abbb (D) a

✓ B.19 This illustrates the point about 'well-defined' in Lemma 2.11's part (1). Consider the $\mathcal{L}$-machine of Example 2.16.

(A) Three representative strings from $\mathcal{E}_{\mathcal{L},0}$ are $\sigma_0 = 00$, $\sigma_1 = 11011$, and $\sigma_2 = 0011111111 = 001^8$. Append 0 to each and name the class of the resulting string. Verify that all three lead to a single class and that in the machine the state $\mathcal{E}_{\mathcal{L},0}$ transitions on input 0 to that class.

(B) Using the same three strings, to each append 1 and name the class of the resulting string. Verify that the three lead to the same class, and that in the machine the state $\mathcal{E}_{\mathcal{L},0}$ transitions on input 1 to it.

(C) Repeat that for three strings from $\mathcal{E}_{\mathcal{L},1}$, with both 0 and 1.

✓ B.20 Example 2.4 gives a language $\mathcal{L} = \{\,\sigma \in \{\,\mathtt{a},\mathtt{b}\,\}^* \;\big|\; \sigma \text{ has even length}\,\}$ with two classes. Produce the transition table and arrow diagram for the associated $\mathcal{L}$-machine.

B.21 Example 2.3 gives a language $\mathcal{L} = \{\,\sigma \in \{\,\mathtt{a},\mathtt{b}\,\}^* \;\big|\; |\sigma| = 3\,\}$ with five classes. Produce the $\mathcal{L}$-machine.

✓ B.22 Let $\mathcal{L}$ be the set of strings from $\{\,\mathtt{a},\mathtt{b}\,\}^*$ ending in a.

(A) Show that $\mathcal{L}$ is an equivalence class, $\mathcal{E}_{\mathcal{L},1}$, for the relation $\sim_{\mathcal{L}}$.

(B) Show that the complement $\mathcal{L}^{\mathsf{c}}$ is also an equivalence class, $\mathcal{E}_{\mathcal{L},0}$, and therefore there are exactly two classes.

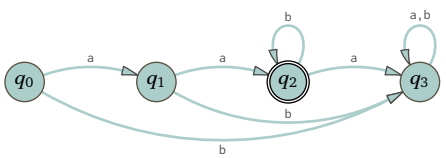
(C) What is the initial state of the $\mathcal{L}$-machine?

(D) What are the accepting states?

(E) Give the transition table and the diagram.

(F) Which of the strings ε, a, b, abba, and bba are accepted by this machine?

✓ B.23 For the language $\mathcal{L} = \{\,\mathtt{a}^2\mathtt{b}^n \;\big|\; n \in \mathbb{N}\,\}$ with alphabet $\Sigma = \{\,\mathtt{a},\mathtt{b}\,\}$ this is a minimal Finite State machine.



List the $\sim_{\mathcal{L}}$ equivalence classes.

✓ B.24 The language $\mathcal{L} = \{ \mathtt{a}^n\mathtt{b} \mid n \in \mathbb{N} \}$ is regular. Find the $\sim_{\mathcal{L}}$ equivalence classes. The alphabet is $\Sigma = \{ \mathtt{a}, \mathtt{b} \}$.

B.25 Find the $\mathcal{L}$ equivalence classes for the language strings that end in `01`. The alphabet is $\mathbb{B}$.

(A) Approach the problem by using Lemma 2.15, building a minimal Finite State machine for the language and using the $\mathcal{M}$ classes as the $\mathcal{L}$ classes.

(B) Instead directly show that the $\mathcal{L}$ class sets are equivalence classes.

B.26 Describe the $\mathcal{L}$ equivalence classes for the set of strings where every `0` is immediately followed by two 1's. The alphabet is $\mathbb{B}$.

B.27 In the $\mathcal{L}$-machines in the examples, the language is one equivalence class. Produce a language $\mathcal{L}$ that is divided by the $\sim_{\mathcal{L}}$ relation into more than one equivalence class.

✓ B.28 The language of palindromes $\mathcal{L} = \{ \sigma \in \mathtt{a}, \mathtt{b}^* \mid \sigma^{\mathsf{R}} = \sigma \}$ is not regular. Find infinitely many $\mathcal{L}$ equivalence classes.

✓ B.29 Show that the language of strings $\sigma \in \mathbb{B}^*$ where the number of `0`'s in σ is the same as the number of 1's is not regular by using the Myhill-Nerode Theorem.

B.30 Show that the language of strings of the form $\mathtt{0}^{2^j}$, that is, $\mathcal{L} = \{ \mathtt{0}, \mathtt{0}^4, \mathtt{0}^8, \ldots \}$, is not regular.

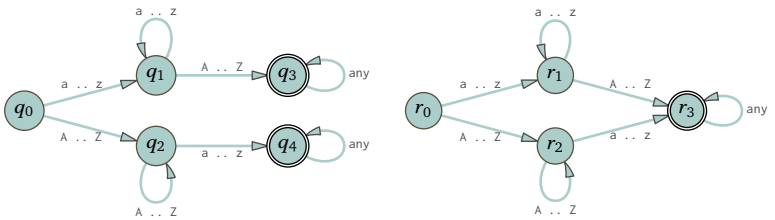
B.31 Recall that a binary relation $\sim$ is an equivalence if it has three properties: (1) reflexivity, that $x \sim x$ for all x, (2) symmetry, that if $x \sim y$ then $y \sim x$, and (3) transitivity, that if $x \sim y$ and $y \sim z$ then also $x \sim z$. (A) Verify Lemma 2.2. (B) Verify Lemma 2.8.

B.32 Generalize the first item in Lemma 2.11 to: if two strings σ_0, σ_1 are $\mathcal{L}$-related, $\sigma_0 \sim_{\mathcal{L}} \sigma_1$, then adjoining any common extension β gives strings that are also $\mathcal{L}$-related, $(\sigma_0 \frown \beta) \sim_{\mathcal{L}} (\sigma_1 \frown \beta)$.

B.33 Show that the equivalence classes for the language $\mathcal{L}$ are the same as for the language that is its complement, $\mathcal{L}^{\mathsf{c}}$.

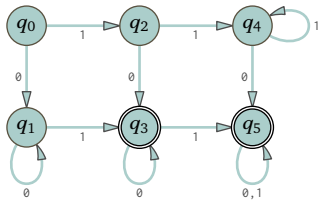
Extra IV.C Machine minimization

Imagine that a person is tasked with ensuring that input for a password form contains both upper and lower case ASCII characters, and produces the machine on the left.



The machine on the right is better in that it has one fewer state. We will give an algorithm, Moore's algorithm (or the table-filling algorithm), that inputs a deterministic Finite State machine and outputs a deterministic machine that is minimal, that from among all of the machines recognizing the same language has the fewest states.[†]

It collapses together redundant states so we begin with an example of those. This recognizes $\mathcal{L} = \{\,\sigma \in \mathbb{B}^* \;\big|\; \sigma \text{ has at least one } \mathtt{0} \text{ and at least one } \mathtt{1}\,\}$.



(∗)

In this chapter's first section, Remark 1.7 recommended designing Finite State machines by thinking about each state's future, anticipating inputs to come. In this machine the future of q_2, "waiting for a 0," is the same as that of q_4. Those states are redundant. Likewise, the future of q_5 matches that of q_3.

To be concrete, this table lists what happens if the machine is started in the given state and the given string is on the tape. Entries contain '+' if the machine then ends in an accepting state, and otherwise are blank. The states q_2 and q_4 have the same rows, at least for the strings listed, as do the states q_3 and q_5.

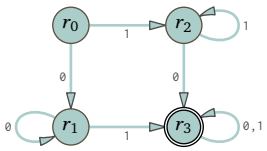
	ε	0	1	00	01	10	11	...
q_0					+	+		...
q_1			+		+	+	+	...
q_2		+		+	+	+		...
q_3	+	+	+	+	+	+	+	...
q_4		+		+	+	+		...
q_5	+	+	+	+	+	+	+	...

(∗∗)

In contrast, q_0 does not have the same row as any other state, nor does q_1.

3.1 Definition Fix a Finite State machine over an alphabet Σ. For two states q and $\hat{q}$, a $\sigma \in \Sigma^*$ is a distinguishing string if starting the machine in q with σ on the tape and starting it in $\hat{q}$ with σ on the tape results in two different outcomes: in one case the machine ends in an accepting state while in the other it rejects. Two states for which there is a distinguishing string are distinguishable, otherwise they are indistinguishable, written $q \sim \hat{q}$.

3.2 Example This is a minimal version of the machine in (∗).



[†] Moore's algorithm is easy to understand and is suitable for small calculations but when writing code be aware that another, Hopcroft's algorithm, is more efficient.

Starting in r_0 and processing the string $\sigma = 1$ ends in a rejecting state, while starting in r_1 and processing σ ends in an accepting state. So the two are distinguished by σ. The states r_0 and r_2, are distinguished by the length one string $\sigma = 0$, and r_0 is distinguished from r_3 by the empty string. Similarly, the pairs r_1, r_2 and r_1, r_3, and r_2, r_3 are all distinguishable. This minimal machine has no indistinguishable states.

States that are indistinguishable are redundant. We will compute whether states are indistinguishable by checking whether they are distinguished by strings of length 0, or by strings of length 1, etc. Two states q and $\hat{q}$ are *n-distinguishable* if there is a distinguishing string of length at most n, otherwise the are *n-indistinguishable*, denoted $q \sim_n \hat{q}$.

Observe that two states q and $\hat{q}$ are 0-indistinguishable if and only if both are accepting states or both are rejecting states.

3.3 LEMMA The relations $\sim_0$, $\sim_1$, ... are equivalences, as is $\sim$, and so partition the states into equivalences classes, the 0-distinguishable classes, the 1-distinguishable classes, ... along with the distinguishable classes.

Proof Exercise C.23. □

3.4 EXAMPLE Here are some n-equivalence classes for the machine (∗), using the information in the table (∗∗).

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	$\mathcal{E}_{2,0} = \{q_0\}$	$\mathcal{E}_{2,1} = \{q_1\}$	$\mathcal{E}_{2,2} = \{q_2, q_4\}$	$\mathcal{E}_{2,3} = \{q_3, q_5\}$

The 0-distinguishable classes divide the rejecting states from the accepting ones.

The 1-distinguishable classes subdivide those, based on the length one strings. Specifically, starting with $\mathcal{E}_{0,0} = \{q_0, q_1, q_2, q_4\}$, we can next distinguish q_0 from q_1 because they differ on the string 1. Similarly, q_0 differs from q_2 and q_4 on 0. And, q_1 differs from q_2 and q_4 on the string 0. So where there was one $\sim_0$ class $\mathcal{E}_{0,0}$ there are now three $\sim_1$ classes, $\mathcal{E}_{1,0} = \{q_0\}$, $\mathcal{E}_{1,1} = \{q_1\}$, and $\mathcal{E}_{1,2} = \{q_2, q_4\}$.

At the next stage, using the length two strings to find the the 2-distinguishability classes does not result in any further subdivisions.

So the algorithm first finds the $\sim_0$ classes, then finds the $\sim_1$ classes, etc., until the classes stop splitting. What remains are the $\sim$ classes, and they serve as states of a minimal machine.

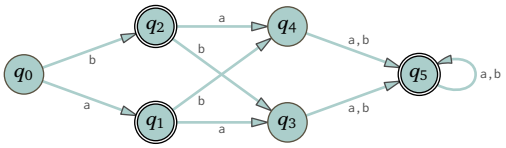
There is one difficulty. In the prior example, to get the $\sim_0$ classes we checked all length 0 string inputs, stage 1 checked all length 1 strings, and stage 2 checked all length 2 strings. If at stage n the algorithm were to check all length n strings then it would take exponentially long, because there are 2^n of them.

To fix that, consider states q and $\hat{q}$ that are not n-distinguishable but are $n+1$-distinguishable. Write a distinguishing string $\tau = \langle s_0, s_1, \ldots s_{n-1}, s_n \rangle$, as $\tau = \alpha^\frown s_n$. Because q and $\hat{q}$ are not n-distinguishable, where α 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 equivalent, $r, \hat{r} \in \mathcal{E}_{n,i}$. Therefore, distinguishing between q and $\hat{q}$ must happen with the final character, s_n. It must take the machine from state r to a state in one class, and also take the machine from $\hat{r}$ to a state in a different class. In short, in looking for a split in passing from the n-distinguishability classes to the $n+1$-distinguishability classes, we need only look at single characters.

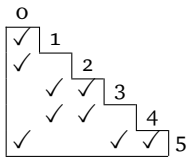
In summary, Moore's algorithm is that nodes q and $\hat{q}$ are $n+1$-indistinguishable if and only if they are n-indistinguishable and also $\Delta(q,x)$ is n-indistinguishable from $\Delta(\hat{q},x)$ for all characters $x \in \Sigma$. The next two examples illustrate, and also show how to use the $\sim$ classes to make minimal machines.

3.5 Example We will find a machine that recognizes the same language as this one but that is minimal.



For bookkeeping we will use triangular tables, with an entry for every pair of different states. We will checkmark pairs that are distinguishable. From stage to stage we fill in more marks, first doing 0-distinguishability, then refining it to 1-distinguishability, etc. When we reach a stage where the table does not change then we are done.

Stage 0 is to checkmark the i, j entries that are 0-distinguishable, where one of q_i and q_j is accepting while the other is not.



Use the blank boxes to read off the $\sim_0$-equivalence classes, because blankness means that the states are mutually 0-indistinguishable. For instance, there are blank boxes in entries $0,3$ and $0,4$ and $3,4$ and this cluster is the first $\sim_0$ equivalence class. Similarly there is a cluster of blank entries in $1,2$ and $1,5$ and $2,5$.

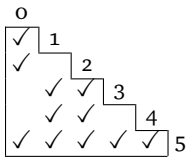
$$\mathcal{E}_{0,0} = \{q_0, q_3, q_4\} \qquad \mathcal{E}_{0,1} = \{q_1, q_2, q_5\}$$

Stage 1 determines whether those $\sim_0$ classes split. For each pair q_i, q_j that are together in a class, and for each input character, compute into which classes that character sends those states.

	a	b
q_0, q_3	$q_1 \in \mathcal{E}_{0,1},\ q_5 \in \mathcal{E}_{0,1}$	$q_2 \in \mathcal{E}_{0,1},\ q_5 \in \mathcal{E}_{0,1}$
q_0, q_4	$q_1 \in \mathcal{E}_{0,1},\ q_5 \in \mathcal{E}_{0,1}$	$q_2 \in \mathcal{E}_{0,1},\ q_5 \in \mathcal{E}_{0,1}$
q_3, q_4	$q_5 \in \mathcal{E}_{0,1},\ q_5 \in \mathcal{E}_{0,1}$	$q_5 \in \mathcal{E}_{0,1},\ q_5 \in \mathcal{E}_{0,1}$
q_1, q_2	$q_3 \in \mathcal{E}_{0,0},\ q_4 \in \mathcal{E}_{0,0}$	$q_4 \in \mathcal{E}_{0,0},\ q_3 \in \mathcal{E}_{0,0}$
q_1, q_5	$q_3 \in \mathcal{E}_{0,0},\ q_5 \in \mathcal{E}_{0,1}$	$q_4 \in \mathcal{E}_{0,0},\ q_5 \in \mathcal{E}_{0,1}$
q_2, q_5	$q_4 \in \mathcal{E}_{0,0},\ q_5 \in \mathcal{E}_{0,1}$	$q_3 \in \mathcal{E}_{0,0},\ q_5 \in \mathcal{E}_{0,1}$

Two states are distinguished when they are sent by a character to members of unequal classes. The first case is in the q_1, q_5 line, where a takes q_1 to $q_3 \in \mathcal{E}_{0,0}$ and takes q_5 to $q_5 \in \mathcal{E}_{0,1}$. So the $\sim_0$ class containing q_1 and q_5, the class $\mathcal{E}_{0,1}$, will split into multiple $\sim_1$ classes. The next line of the computation also shows that q_2 and q_5 are distinguishable.

To record that q_1 and q_5 are distinguishable, add a checkmark in the triangular table's cell for $1, 5$. Also add a $2, 5$ checkmark.



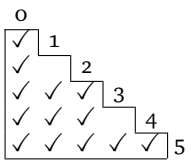
Finish this stage by getting the $\sim_1$ classes as clusters of blank cells. There is a cluster in $0, 3$ and $0, 4$ and $3, 4$. There is also a blank cell in $1, 2$. There is no cluster involving q_5 but since every state must be in some class, it goes in a class by itself.

$$\mathcal{E}_{1,0} = \{q_0, q_3, q_4\} \quad \mathcal{E}_{1,1} = \{q_1, q_2\} \quad \mathcal{E}_{1,2} = \{q_5\}$$

The $\sim_0$ class $\mathcal{E}_{0,1} = \{q_1, q_2, q_5\}$ has split into two $\sim_1$ classes.

Iterate. At stage 2 the single-element set $\mathcal{E}_{1,2}$ can't split so consider only pairs from the other two classes. Calculation shows that q_0 is distinguishable from q_3 and also from q_4. Add to the triangular table checkmarks for $0, 3$ and $0, 4$.

	a	b
q_0, q_3	$q_1 \in \mathcal{E}_{1,1},\ q_5 \in \mathcal{E}_{1,2}$	$q_2 \in \mathcal{E}_{1,1},\ q_5 \in \mathcal{E}_{1,2}$
q_0, q_4	$q_1 \in \mathcal{E}_{1,1},\ q_5 \in \mathcal{E}_{1,2}$	$q_2 \in \mathcal{E}_{1,1},\ q_5 \in \mathcal{E}_{1,2}$
q_3, q_4	$q_5 \in \mathcal{E}_{1,2},\ q_5 \in \mathcal{E}_{1,2}$	$q_5 \in \mathcal{E}_{1,2},\ q_5 \in \mathcal{E}_{1,2}$
q_1, q_2	$q_3 \in \mathcal{E}_{1,0},\ q_4 \in \mathcal{E}_{1,0}$	$q_4 \in \mathcal{E}_{1,0},\ q_3 \in \mathcal{E}_{1,0}$

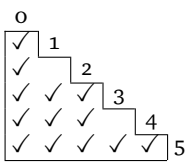


The blank cells give these $\sim_2$ classes.

$$\mathcal{E}_{2,0} = \{q_0\} \quad \mathcal{E}_{2,1} = \{q_1, q_2\} \quad \mathcal{E}_{2,2} = \{q_3, q_4\} \quad \mathcal{E}_{2,3} = \{q_5\}$$

One more iteration.

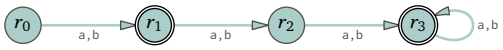
	a	b
q_1, q_2	$q_3 \in \mathcal{E}_{2,2},\ q_4 \in \mathcal{E}_{2,2}$	$q_4 \in \mathcal{E}_{2,2},\ q_3 \in \mathcal{E}_{2,2}$
q_3, q_4	$q_5 \in \mathcal{E}_{2,3},\ q_5 \in \mathcal{E}_{2,3}$	$q_5 \in \mathcal{E}_{2,3},\ q_5 \in \mathcal{E}_{2,3}$



There was no more splitting. The algorithm terminates with these $\sim$ classes.

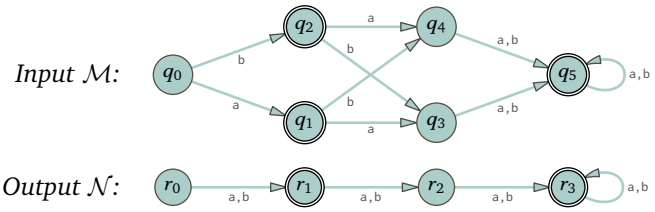
$$\mathcal{E}_0 = \{q_0\} \quad \mathcal{E}_1 = \{q_1, q_2\} \quad \mathcal{E}_2 = \{q_3, q_4\} \quad \mathcal{E}_3 = \{q_5\}$$

To finish, we 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.



To define the transitions between states, consider what happens when we feed the character a to elements of a class, such as the class $r_1 = \mathcal{E}_1 = \{q_1, q_2\}$. For instance, if we choose q_1 and look in the original machine then under input a it goes to q_3. Since q_3 is an element of $\mathcal{E}_2 = r_2$, in the minimal machine the a arrow out of r_1 goes to r_2. Other transitions work the same way.

As to the terminology that this algorithm 'collapses' together the redundant states, consider this picture of the prior example, showing a kind of projection.

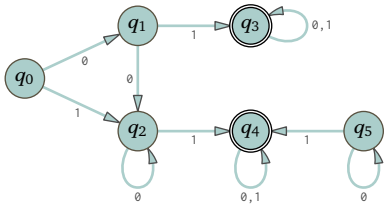


We can express the same using Δ tables.

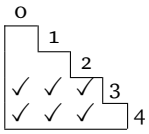
$\Delta_{\mathcal{M}}$	a	b
q_0	q_1	q_2
q_1	q_3	q_4
q_2	q_4	q_3
q_3	q_5	q_5
q_4	q_5	q_5
q_5	q_5	q_5

$\Delta_{\mathcal{N}}$	a	b
r_0	r_1	r_1
r_1	r_2	r_2
r_2	r_3	r_3
r_3	r_3	r_3

3.6 Example We will minimize the machine below. This illustrates one additional point of the algorithm since this machine has an unreachable state, q_5. Start by omitting it.



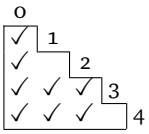
That leaves this stage 0 triangular table and these $\sim_0$ classes.



$$\mathcal{E}_{0,0} = \{q_0, q_1, q_2\} \quad \mathcal{E}_{0,1} = \{q_3, q_4\}$$

For stage 1 check whether those classes split. The calculation shows that q_0 is distinguished from q_1 by 1, and q_0 is distinguished from q_2, also by 1. The triangular table below reflects those updates.

	0	1
q_0, q_1	$q_1 \in \mathcal{E}_{0,0},\ q_2 \in \mathcal{E}_{0,0}$	$q_2 \in \mathcal{E}_{0,0},\ q_3 \in \mathcal{E}_{0,1}$
q_0, q_2	$q_1 \in \mathcal{E}_{0,0},\ q_2 \in \mathcal{E}_{0,0}$	$q_2 \in \mathcal{E}_{0,0},\ q_4 \in \mathcal{E}_{0,1}$
q_1, q_2	$q_2 \in \mathcal{E}_{0,0},\ q_2 \in \mathcal{E}_{0,0}$	$q_3 \in \mathcal{E}_{0,1},\ q_4 \in \mathcal{E}_{0,1}$
q_3, q_4	$q_3 \in \mathcal{E}_{0,1},\ q_4 \in \mathcal{E}_{0,1}$	$q_3 \in \mathcal{E}_{0,1},\ q_4 \in \mathcal{E}_{0,1}$



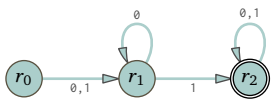
Thus the first $\sim_0$ class splits and these are the $\sim_1$ classes.

$$\mathcal{E}_{1,0} = \{q_0\} \quad \mathcal{E}_{1,1} = \{q_1, q_2\} \quad \mathcal{E}_{1,2} = \{q_3, q_4\}$$

The next iteration, stage 2, shows no more splitting.

	0	1
q_1, q_2	$q_2 \in \mathcal{E}_{1,1},\ q_2 \in \mathcal{E}_{1,1}$	$q_3 \in \mathcal{E}_{1,2},\ q_4 \in \mathcal{E}_{1,2}$
q_3, q_4	$q_3 \in \mathcal{E}_{1,2},\ q_4 \in \mathcal{E}_{1,2}$	$q_3 \in \mathcal{E}_{1,2},\ q_4 \in \mathcal{E}_{1,2}$

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 $\mathcal{E}_2 = \{q_3, q_4\}$.

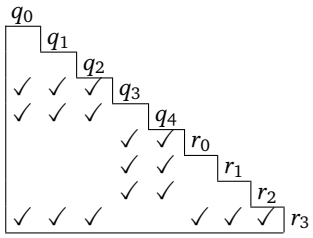


3.7 LEMMA Moore's algorithm outputs a machine that recognizes the same language as the input machine and that is minimal.

Proof See Exercise C.24, which verifies that that Moore's algorithm always halts, that it produces a Finite State machine with a well-defined transition function,[†] that this output machine recognizes the same language as the input machine, and that the output machine is minimal. □

3.8 EXAMPLE As an alternative to the lemma's whole argument, we will illustrate the approach to showing minimality. We use the two machines given at the section's start (we write 'a' for 'a . . z' and 'A' for 'A . . Z'). Call the input machine $\mathcal{M}$ and the output $\mathcal{N}$. Consider the union of the two sets of states. Here is the stage 0 table and classes.

[†] The last paragraph of Example 3.5 describes how to define the transition function of the minimal machine. It appears that if an input class r_i has more than one element q_j then, depending on which one we choose, we could get different output classes. We must show that the output is the same no matter what choice we make.

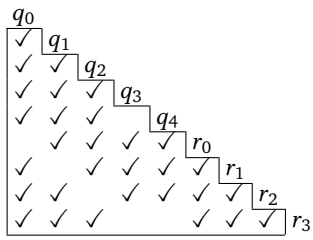


$$\mathcal{E}_{0,0} = \{ q_3, q_4, r_3 \} \quad \mathcal{E}_{0,1} = \{ q_0, q_1, q_2, r_0, r_1, r_2 \}$$

Stage 1 looks at pairs from the two $\sim_0$ classes, calculating whether character transitions split any class.

	a	A
q_3, q_4	$q_3 \in \mathcal{E}_{0,0}$, $q_4 \in \mathcal{E}_{0,0}$	$q_3 \in \mathcal{E}_{0,0}$, $q_4 \in \mathcal{E}_{0,0}$
q_3, r_3	$q_3 \in \mathcal{E}_{0,0}$, $r_3 \in \mathcal{E}_{0,0}$	$q_3 \in \mathcal{E}_{0,0}$, $r_3 \in \mathcal{E}_{0,0}$
q_4, r_3	$q_4 \in \mathcal{E}_{0,0}$, $r_3 \in \mathcal{E}_{0,0}$	$q_4 \in \mathcal{E}_{0,0}$, $r_3 \in \mathcal{E}_{0,0}$
q_0, q_1	$q_1 \in \mathcal{E}_{0,1}$, $q_1 \in \mathcal{E}_{0,1}$	$q_2 \in \mathcal{E}_{0,1}$, $q_3 \in \mathcal{E}_{0,0}$
q_0, q_2	$q_1 \in \mathcal{E}_{0,1}$, $q_4 \in \mathcal{E}_{0,0}$	$q_2 \in \mathcal{E}_{0,1}$, $q_2 \in \mathcal{E}_{0,1}$
q_0, r_0	$q_1 \in \mathcal{E}_{0,1}$, $r_1 \in \mathcal{E}_{0,1}$	$q_2 \in \mathcal{E}_{0,1}$, $r_2 \in \mathcal{E}_{0,1}$
q_0, r_1	$q_1 \in \mathcal{E}_{0,1}$, $r_1 \in \mathcal{E}_{0,1}$	$q_2 \in \mathcal{E}_{0,1}$, $r_3 \in \mathcal{E}_{0,0}$
q_0, r_2	$q_1 \in \mathcal{E}_{0,1}$, $r_3 \in \mathcal{E}_{0,0}$	$q_2 \in \mathcal{E}_{0,1}$, $r_2 \in \mathcal{E}_{0,1}$
q_1, q_2	$q_1 \in \mathcal{E}_{0,1}$, $q_4 \in \mathcal{E}_{0,0}$	$q_3 \in \mathcal{E}_{0,0}$, $q_4 \in \mathcal{E}_{0,0}$
q_1, r_0	$q_1 \in \mathcal{E}_{0,1}$, $r_1 \in \mathcal{E}_{0,1}$	$q_3 \in \mathcal{E}_{0,0}$, $r_2 \in \mathcal{E}_{0,1}$
q_1, r_1	$q_1 \in \mathcal{E}_{0,1}$, $r_1 \in \mathcal{E}_{0,1}$	$q_3 \in \mathcal{E}_{0,0}$, $r_3 \in \mathcal{E}_{0,0}$
q_1, r_2	$q_1 \in \mathcal{E}_{0,1}$, $r_3 \in \mathcal{E}_{0,0}$	$q_3 \in \mathcal{E}_{0,0}$, $r_2 \in \mathcal{E}_{0,1}$
q_2, r_0	$q_4 \in \mathcal{E}_{0,0}$, $r_1 \in \mathcal{E}_{0,1}$	$q_2 \in \mathcal{E}_{0,1}$, $r_2 \in \mathcal{E}_{0,1}$
q_2, r_1	$q_4 \in \mathcal{E}_{0,0}$, $r_1 \in \mathcal{E}_{0,1}$	$q_2 \in \mathcal{E}_{0,1}$, $r_3 \in \mathcal{E}_{0,0}$
q_2, r_2	$q_4 \in \mathcal{E}_{0,0}$, $r_3 \in \mathcal{E}_{0,0}$	$q_2 \in \mathcal{E}_{0,1}$, $r_2 \in \mathcal{E}_{0,1}$
r_0, r_1	$r_1 \in \mathcal{E}_{0,1}$, $r_1 \in \mathcal{E}_{0,1}$	$r_2 \in \mathcal{E}_{0,1}$, $r_3 \in \mathcal{E}_{0,0}$
r_0, r_2	$r_1 \in \mathcal{E}_{0,1}$, $r_3 \in \mathcal{E}_{0,0}$	$r_2 \in \mathcal{E}_{0,1}$, $r_2 \in \mathcal{E}_{0,1}$
r_1, r_2	$r_1 \in \mathcal{E}_{0,1}$, $r_3 \in \mathcal{E}_{0,0}$	$r_3 \in \mathcal{E}_{0,0}$, $r_2 \in \mathcal{E}_{0,1}$

There are many splits, for example q_0 and q_1 are distinguished by A. In the resulting triangular table there are six empty boxes, at q_0, r_0, at q_1, r_1, at q_2, r_2, at q_3, q_4, at q_3, r_3, and at q_4, r_3. We conclude that $\mathcal{E}_{0,0}$ does not split but that $\mathcal{E}_{0,1}$ splits into three.



$$\mathcal{E}_{1,0} = \{ q_3, q_4, r_3 \}$$
$$\mathcal{E}_{1,1} = \{ q_0, r_0 \}$$
$$\mathcal{E}_{1,2} = \{ q_1, r_1 \}$$
$$\mathcal{E}_{1,3} = \{ q_2, r_2 \}$$

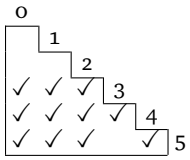
The next stage shows no more splittings. The algorithm has grouped q_0 with r_0, and q_1 with r_1, and q_2 with r_2. It has also grouped q_3 and q_4 together with r_3.

It is not surprising that starting with the minimal machine $\mathcal{N}$ and performing the algorithm results in one state r_i per $\sim$ class. It is also not surprising that the states of $\mathcal{M}$ can end with multiple q_js' in a class, since we have seen that in earlier examples. But the point is that for each r_i there is at least one associated q_j, and therefore $\mathcal{N}$ has a number of states that is less than or equal to the number in $\mathcal{M}$.

We close by describing a common scenario in which minimization plays an important role. 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 an 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 the problem that where the nondeterministic machine has n states, the deterministic machine has 2^n. This section's result alleviates that exponential blow-up. 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.C Exercises

✓ C.9 From the triangular table find the $\sim_i$ classes.



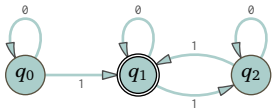
C.10 From the $\sim_i$ classes find the associated triangular table. (A) $\mathcal{E}_{i,0} = \{q_0, q_1\}$, $\mathcal{E}_{i,1} = \{q_2\}$, and $\mathcal{E}_{i,2} = \{q_3, q_4\}$, (B) $\mathcal{E}_{i,0} = \{q_0\}$, $\mathcal{E}_{i,1} = \{q_1, q_2, q_4\}$, and $\mathcal{E}_{i,2} = \{q_3\}$, (C) $\mathcal{E}_{i,0} = \{q_0, q_1, q_5\}$, $\mathcal{E}_{i,1} = \{q_2, q_3\}$, and $\mathcal{E}_{i,2} = \{q_4\}$,

✓ C.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 we compute this table.

	a	b
q_0, q_1	$q_1 \in \mathcal{E}_{0,0}$, $q_1 \in \mathcal{E}_{0,0}$	$q_2 \in \mathcal{E}_{0,0}$, $q_3 \in \mathcal{E}_{0,1}$
q_0, q_2	$q_1 \in \mathcal{E}_{0,0}$, $q_2 \in \mathcal{E}_{0,0}$	$q_2 \in \mathcal{E}_{0,0}$, $q_4 \in \mathcal{E}_{0,1}$
q_0, q_5	$q_1 \in \mathcal{E}_{0,0}$, $q_5 \in \mathcal{E}_{0,0}$	$q_2 \in \mathcal{E}_{0,0}$, $q_5 \in \mathcal{E}_{0,0}$
q_1, q_2	$q_1 \in \mathcal{E}_{0,0}$, $q_2 \in \mathcal{E}_{0,0}$	$q_3 \in \mathcal{E}_{0,1}$, $q_4 \in \mathcal{E}_{0,1}$
q_1, q_5	$q_1 \in \mathcal{E}_{0,0}$, $q_5 \in \mathcal{E}_{0,0}$	$q_3 \in \mathcal{E}_{0,1}$, $q_5 \in \mathcal{E}_{0,0}$
q_2, q_5	$q_2 \in \mathcal{E}_{0,0}$, $q_5 \in \mathcal{E}_{0,0}$	$q_4 \in \mathcal{E}_{0,1}$, $q_5 \in \mathcal{E}_{0,0}$
q_3, q_4	$q_3 \in \mathcal{E}_{0,1}$, $q_4 \in \mathcal{E}_{0,1}$	$q_5 \in \mathcal{E}_{0,0}$, $q_5 \in \mathcal{E}_{0,0}$

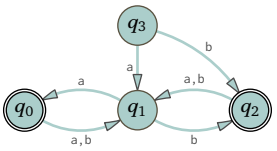
(A) Which states are 1-distinguishable that were not 0-distinguishable? (B) Give the resulting $\sim_1$ classes.

✓ C.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.

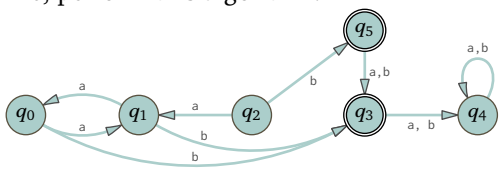


C.13 For many machines we can find the unreachable states by eye, but there is an algorithm. It inputs a machine $\mathcal{M}$ and initializes the set of reachable states to $R_0 = \{q_0\}$. For $n > 0$, step n of the algorithm is: for each $q \in R_n$ find all states $\hat{q}$ reachable from q in one transition and add those to make R_{n+1}. That is, $R_{n+1} = R_n \cup \{\hat{q} = \Delta_{\mathcal{M}}(q, x) \mid q \in R_n \text{ and } x \in \Sigma\}$. 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.

(A)

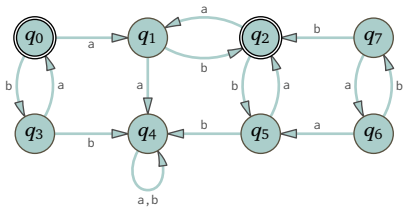


(B)



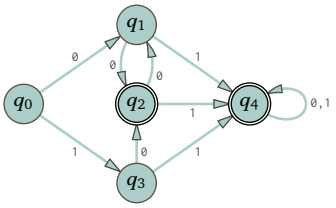
✓ C.14 Perform the minimization algorithm on the machine with redundant states at the start of this section, the one labeled $(*)$.

✓ C.15 This machine accepts strings described by `(ab|ba)*`. Minimize it, using the algorithm of this section.

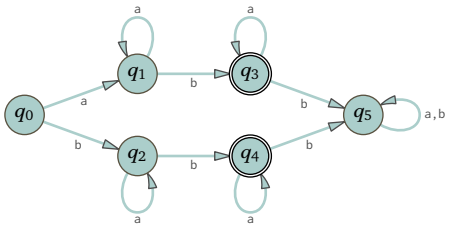


C.16 If a machine's start state is accepting, must the minimized machine's start state be accepting? If so then prove it, and if not then give an example machine where it is false.

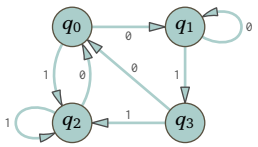
C.17 Minimize.



C.18 Minimize.

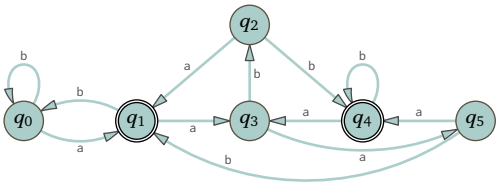


C.19 This machine has no accepting states. Minimize it.



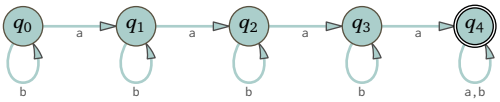
What happens to a machine where all states are accepting?

C.20 Minimize this machine.



C.21 What happens if you perform the minimization procedure on a machine that has an unreachable state, such as Example 3.6, without first omitting the unreachable state?

✓ C.22 Minimize.



Note that the algorithm takes a time that is roughly are equal to the number of states in the machine.

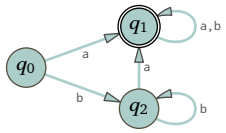
C.23 Verify Lemma 3.3. (A) Verify that each $\sim_n$ is an equivalence relation between states. (B) Verify that $\sim$ is an equivalence.

C.24 We will verify that Moore's algorithm halts on any input machine $\mathcal{M}$ and outputs an $\mathcal{N}$ that recognizes the same language, and that is minimal.

(A) Prove that the algorithm always halts.
(B) Prove that the transition function of $\mathcal{N}$ is well-defined.
(C) Verify that the two machines recognize the same language.
(D) Show that $\mathcal{N}$ is minimal: any $\hat{\mathcal{M}}$ that recognize the same language as $\mathcal{N}$ has at least as many states as $\mathcal{N}$. *Hint:* first follow Example 3.8. Then do an argument

by induction that shows that the start states of the two are indistinguishable, and if two states q and r are indistinguishable then so are the states they transition to on a single-character input. This gives an association of single r's with at least one q, and so there are at least as many q's as r's.

C.25 There are ways to minimize Finite State machines other then 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 not go through why it works, but we will walk through the algorithm using this Finite State machine, $\mathcal{M}$.



(A) Use Moore's algorithm 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 (loops reverse to themselves). 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. (This may give a machine with more than one initial state.)

(C) Convert this to a deterministic machine, using the powerset method. Name the state of this machine u_i. Omit unreachable states.

(D) Repeat the second item by changing the state names from u_i to v_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.

C.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.

Part Three

Computational Complexity

Chapter V Computational Complexity

We started this book by asking what can be done with a mechanism at all. This mirrors the subject's history: when the Theory of Computing began there were no matching mechanisms. 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 computing machines 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 chapter's end we will be able to understand some topics of current research. In particular, we will consider the celebrated question of *P* versus *NP*.

Section V.1 Big $\mathcal{O}$

First we review the definition of the order of growth of functions. It measures how algorithms consume computational resources.

First, an anecdote. Here is a grade school multiplication.

$$\begin{array}{r} 678 \\ \times 42 \\ \hline 1356 \\ 2712 \\ \hline 28476 \end{array}$$

This familiar 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 — and indeed, the only reasonable way — and that in general, to multiply two n digit numbers requires at least 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. Before its second meeting one of the students, A Karatsuba, showed that it is wrong. 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.

Andrey Kolmogorov 1903–1987

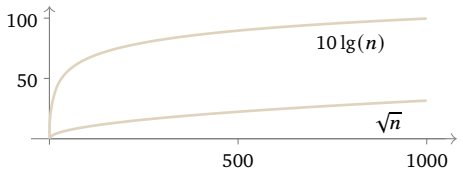
This pattern continues. Every day researchers produce results saying, "for this problem, here is a way to solve it in less time, or less space, etc."[†] People are good at finding algorithms that solve a problem using less of some computational resource. But we are not as good at finding lower bounds, 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 that is like 'less than or equal to'.

Motivation We now develop the criteria for this comparison tool, Big $\mathcal{O}$. This routine has a loop and so takes longer for larger n.

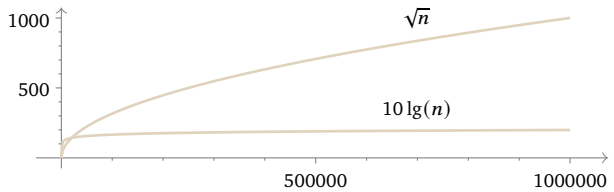
```
(define (f n)
  (for ([i (in-range n)])
    (printf "~a " i)))
```

This is typical. Big $\mathcal{O}$ will describe the relationship between the size of the algorithm's input and the resources that it uses, especially the run time, on inputs of that size.

Next, 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 $10\lg(n)$.[‡] 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 $10\lg(1\,000\,000) \approx 199.32$.

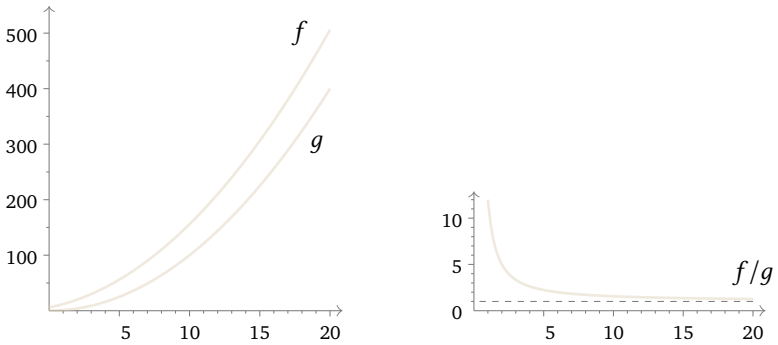


[†] See the Theory of Computing blog feed at `https://theory.report` (Various authors 2017). [‡] We write $\lg(n)$ for $\log_2(n)$. Thus $\lg(n)$ is the power of 2 that produces n, so if $n = 8$ then $\lg(n) = 3$ while if $n = 10$ then $\lg(n) \approx 3.32$.

The first criteria for the definition of big $\mathcal{O}$ is that it should give the relationship between input size and output time in the long run.

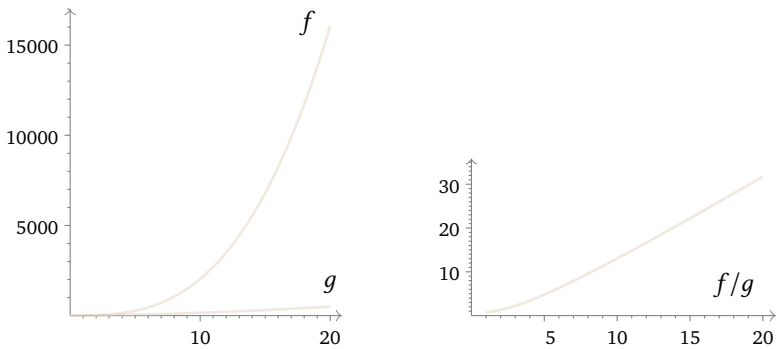
The other criteria for Big $\mathcal{O}$ is more subtle. The next four examples illustrate.

1.1 Example The graph on the left compares $g(x) = x^2 + 5x + 6$ with $f(x) = x^2$. The one on the right shows the ratio f/g.



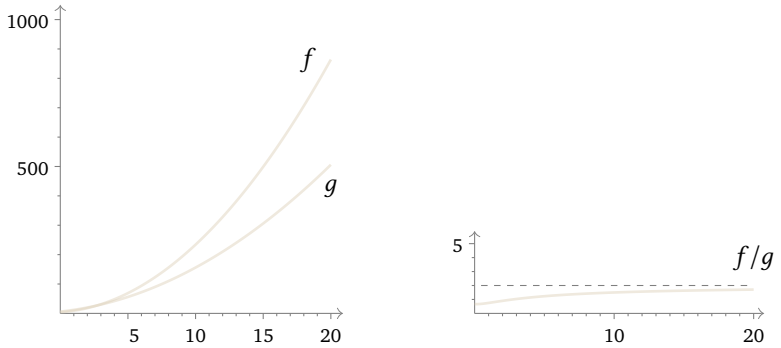
On the left we are struck that g is ahead of f. But the ratios on the right show that this visual is misleading. For large x's, g's terms $5x$ and 6 are swamped by its x^2 term. In the long run these two functions have a ratio of one so they track together — they share the biggest ingredient in their behavior, which is that they are quadratic.

1.2 Example Next compare the same quadratic $g(x) = x^2 + 5x + 6$ with the cubic $f(x) = x^3 + 2x + 3$. Unlike 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 has a strictly higher growth rate than g. They both go to infinity but f goes there faster.

1.3 Example Now compare that quadratic $g(x) = x^2+5x+6$ with $f(x) = 2x^2+3x+4$. Again there is some initial slop, that $f(0) = 4 < g(0) = 6$ and $f(1) = 9 < g(1) = 12$, but we discount that and focus on the long run.

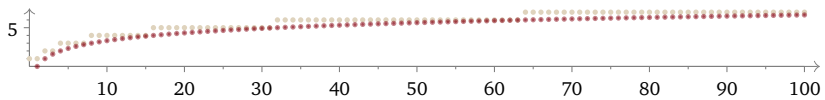


This example differs from Example 1.1 since f stays ahead of g and also gains in an absolute sense: first it is ahead by 10, then by 100, then by 1000, etc. This is because f's dominant term $2x^2$ is twice as large as g's x^2. So it may seem that we should view f's rate as strictly greater than g's. However, unlike in Example 1.2, the ratio between the two is bounded, coming to a limit of 2. The definition of $\mathcal{O}$ will disregard constant factors such as the 2 here, and take these two functions to have growth rates that are equivalent.

1.4 Example We close the motivation with a very important example. Let the function bits: $\mathbb{N} \to \mathbb{N}$ 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.

Input n	0	1	2	3	4	5	6	7	8	9
Binary	0	1	10	11	100	101	110	111	1000	1001
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

The relationship between the third and fourth lines is that $\text{bits}(n) = 1 + \lfloor \lg(n) \rfloor$ (except for the boundary value at input 0, where lg is undefined). The graph below compares bits(n) with $\lg(n)$.



The formula's '1+' and floor operator $\lfloor \cdot \rfloor$ don't matter over the long run. In short, the number of bits required to represent the number n is roughly $\lg n$.

One further point about bits(n). The formula for converting among logarithmic functions with different bases, $\log_c(x) = \log_b(x)/\log_b(c)$, says that they differ only by a constant factor, $1/\log_b(c)$. As Example 1.3 notes, the definition given below we will disregard constant factors. So another reasonable summary is that the number of bits is "a" logarithmic function.

This example illustrates that logarithms are among the functions that we will want to use.

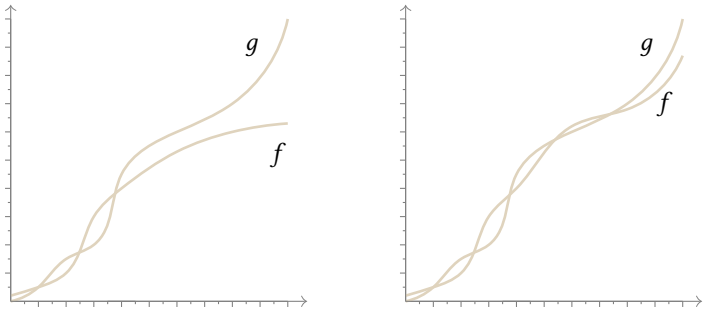
Definition We want to measure consumption of machine resources such as number of ticks spent running or bits of memory used. Those are natural numbers. So we think to focus on functions that input and output natural numbers. However, above we have already found useful a function, lg, that inputs and outputs reals.

1.5 DEFINITION A function f that inputs and outputs real numbers is a complexity function if it has an unbounded domain and is eventually nonnegative. That is, there is an $N \in \mathbb{R}$ such that $x \geq N$ implies that $f(x)$ is defined and that $f(x) \geq 0$.

1.6 DEFINITION Let f and g be complexity functions. Then f is Big $\mathcal{O}(g)$[†] when there are constants $N, C \in \mathbb{R}$ such that if $x \geq N$ then both $g(x)$ and $f(x)$ are defined, and $|f(x)| \leq C \cdot |g(x)|$. Other terminology is that $f \in \mathcal{O}(g)$, or f is of order at most g, or $f = \mathcal{O}(g)$. The set $\mathcal{O}(g)$ is the collection of complexity functions f where f is $\mathcal{O}(g)$.

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 it does not follow the usual rules of equality, such as that $f = \mathcal{O}(g)$ does not allow us to write '$\mathcal{O}(g) = f$' and that $x = \mathcal{O}(x^2)$ and $x^2 = \mathcal{O}(x^2)$ do not imply that $x = x^2$. (4) The absolute values in $|f(x)| \leq C \cdot |g(x)|$ are standard because they fit with the use of the idea in other areas of mathematics. But they can be an annoyance in doing algebra. Complexity functions are eventually nonnegative so if convenient we assume that x is large enough that the functions are positive. (5) Sometimes people say '$f(x)$ is $\mathcal{O}(g(x))$'. This is sloppy because $f(x)$ and $g(x)$ are numbers, not functions. (6) We will focus on using Big $\mathcal{O}$ to describe running time.

Think of $\mathcal{O}$ as about 'less than or equal to', so that 'f is $\mathcal{O}(g)$' means 'f's growth rate is less than or equal to g's'. The sketches below illustrate the two possibilities.



On the left, f's growth rate of growth appears to be strictly less than g's. On the right they appear to have equivalent rates.

To show that f is $\mathcal{O}(g)$ the definition asks us to produce suitable numbers 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

[†] Read aloud as "f is Big Oh of g."

than $2 \cdot x^2$, which in turn is greater than $C \cdot f(x) = 1 \cdot f(x) = x^2$.

1.9 Example If $f(x) = 5x^2$ and $g(x) = x^4$ then to show that f is $\mathcal{O}(g)$ take $N = 2$ and $C = 2$. The verification is that $x > N = 2$ implies that $C \cdot x^4 = 2 \cdot x^2 \cdot x^2 \geq 8x^2 > 5x^2$.

1.10 Example Don't confuse a function having smaller values with it having a smaller growth rate. Let $g(x) = x^2$ and $f(x) = x^2 + 1$. Then g's values are smaller than f's since $g(x) < f(x)$ for all x. But g's growth rate is not smaller; rather, the two rates are equivalent. In particular, to verify that f is $\mathcal{O}(g)$, take $N = 1$ and $C = 2$. Then $x \geq N = 1$ gives $C \cdot g(x) = 2x^2 = x^2 + x^2 \geq x^2 + 1 = f(x)$.

1.11 Example Some pairs of functions aren't comparable, so that neither is $f \in \mathcal{O}(g)$ nor is $g \in \mathcal{O}(f)$. For instance, let $g(x) = x^3$ and consider this function.

$$f(x) = \begin{cases} x^2 & \text{– if } \lfloor x \rfloor \text{ is even} \\ x^4 & \text{– if } \lfloor x \rfloor \text{ is odd} \end{cases}$$

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, and thus f is not $\mathcal{O}(g)$. 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) For any constant $a \in \mathbb{R}^+$, the function $a \cdot f$ is $\mathcal{O}(f)$.

(B) If f_1 is $\mathcal{O}(f_0)$ then the sum $f_0 + f_1$ and difference $f_0 - f_1$ are also $\mathcal{O}(f_0)$.

(C) In contrast with addition, for multiplication we cannot in general drop the lower-order term — even if f_1 is $\mathcal{O}(f_0)$, we cannot always conclude that $f_0 f_1$ is $\mathcal{O}(f_0)$.

Proof See Exercise 1.62. □

That result gives us two principles for simplifying Big $\mathcal{O}$ expressions. First, we can drop positive constant factors. Second, if an expression is a sum of finitely many positive terms then we can drop all but the one with the largest growth rate. Often these two principles lead to a natural growth rate for an expression.

1.13 Example Consider $f(n) = 5x^3 + 3x^2 + 12x$. Applying the lemma's second item with $g_0(x) = 5x^3$ and $g_1(x) = 3x^2$ and $g_2(x) = 12x$ gives that f is $\mathcal{O}(5x^3)$, so we are dropping all of the terms except for the one with the highest growth rate (that this is $5x^3$ is intuitively clear and will follow from Theorem 1.17 below). Then the lemma's first item with $a = 1/5$ lets us drop the constant 5, giving that f is $\mathcal{O}(x^3)$.

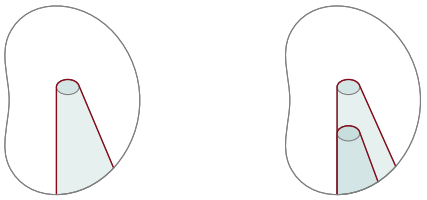
While f is also $\mathcal{O}(x^4)$, and $\mathcal{O}(x^5)$, etc., characterizing it as $\mathcal{O}(x^3)$ is natural.

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 $\Theta(g)$[†] 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.

[†] Read aloud as "f is Big Theta of g."

Proof See Exercise 1.63 □



1.16 FIGURE: Each bean holds the complexity functions. Faster growing functions are drawn higher up. On the left is the cone $\mathcal{O}(g)$ for some g containing all of the functions with growth rate less than or equal to g's. The ellipse at the top is $\Theta(g)$, holding functions with growth rate equivalent to g's. On the right the sketch also has 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.

1.17 THEOREM Let f, g be complexity functions. Suppose that $\lim_{x\to\infty} f(x)/g(x)$ 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 the same order of growth, so that f is $\Theta(g)$ and g is $\Theta(f)$.[#]

Proof See Exercise 1.64. □

It pairs well with this result, familiar from Calculus I.

1.18 THEOREM (L'HÔPITAL'S RULE) Let f and g be complexity functions such that both $f(x)\to\infty$ and $g(x)\to\infty$ as $x\to\infty$, and such that both are differentiable for large enough inputs. If $\lim_{x\to\infty} f'(x)/g'(x)$ exists and equals $L\in\mathbb{R}\cup\{\infty\}$ then $\lim_{x\to\infty} f(x)/g(x)$ also exists and also equals L.

1.19 EXAMPLE Let $f(x)=x^2+5x+6$ and $g(x)=x^3+2x+3$. As often happens, here we get the limit by applying L'Hôpital's Rule more than once.

$$\lim_{x\to\infty}\frac{f(x)}{g(x)}=\lim_{x\to\infty}\frac{x^2+5x+6}{x^3+2x+3}=\lim_{x\to\infty}\frac{2x+5}{3x^2+2}=\lim_{x\to\infty}\frac{2}{6x}=0$$

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 Let $f(x)=3x^2+4x+5$ and $g(x)=x^2$.

$$\lim_{x\to\infty}\frac{3x^2+4x+5}{x^2}=\lim_{x\to\infty}\frac{6x+4}{2x}=\lim_{x\to\infty}\frac{6}{2}=3$$

So their growth rates are roughly the same. That is, f is $\Theta(g)$.

[†] 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.

1.21 Example For $f(x)=5x^4+15$ and $g(x)=x^2$, the limit $\lim_{x\to\infty}(5x^4+15)/x^2=\lim_{x\to\infty}20x^3/2x=\lim_{x\to\infty}10x^2=\infty$ 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 is $\mathcal{O}(x^{0.1})$, and is $\mathcal{O}(x^{0.01})$. In fact, for any $d>0$ no matter how small, $\log_b x$ is $\mathcal{O}(x^d)$ and x^d is not $\mathcal{O}(\log_b x)$.

$$\lim_{x\to\infty}\frac{\log_b x}{x^d}=\lim_{x\to\infty}\frac{1/(x\ln b)}{dx^{d-1}}=\frac{1}{d\ln b}\cdot\lim_{x\to\infty}\frac{1}{x^d}=0$$

The difference in growth rates between logarithmic and polynomial functions is even more marked than that. L'Hôpital's Rule gives that $(\log_b x)^2$ is $\mathcal{O}(x)$.

$$\begin{aligned}\lim_{x\to\infty}\frac{(\log_b x)^2}{x}&=\lim_{x\to\infty}\frac{2(\log_b x)\cdot 1/(x\ln b)}{1}=\lim_{x\to\infty}\frac{2(\ln x/\ln b)/(x\ln b)}{1}\\&=\frac{2}{(\ln b)^2}\cdot\lim_{x\to\infty}\frac{\ln x}{x}=\frac{2}{(\ln b)^2}\cdot\lim_{x\to\infty}\frac{1/x}{1}=0\end{aligned}$$

Exercise 1.54 shows that for every power k the function $(\log_b x)^k$ is $\mathcal{O}(x^d)$ for any $d>0$.

1.23 Example The log-linear function $x\cdot\lg x$ has a similar relationship to the polynomial x^d where $d>1$.

$$\begin{aligned}\lim_{x\to\infty}\frac{x\lg x}{x^d}&=\lim_{x\to\infty}\frac{x\cdot(1/x\ln 2)+\lg x\cdot 1}{dx^{d-1}}=\frac{1}{d}\cdot\lim_{x\to\infty}\frac{(1/\ln 2)+\lg x}{x^{d-1}}\\&=\frac{1}{d}\cdot\lim_{x\to\infty}\frac{1/(x\ln 2)}{(d-1)x^{d-2}}=\frac{1}{d(d-1)\ln 2}\cdot\lim_{x\to\infty}\frac{1}{x^{d-1}}=0\end{aligned}$$

1.24 Example We can compare the polynomial function $f(x)=x^2$ with the exponential function $g(x)=2^x$.

$$\lim_{x\to\infty}\frac{2^x}{x^2}=\lim_{x\to\infty}\frac{2^x\cdot\ln(2)}{2x}=\lim_{x\to\infty}\frac{2^x\cdot(\ln(2))^2}{2}=\infty$$

Thus $f\in\mathcal{O}(g)$ but $g\notin\mathcal{O}(f)$. Induction gives that $\lim_{x\to\infty}2^x/x^k=\infty$ for any k.

1.25 Lemma Logarithmic functions grow more slowly than polynomial functions: if $f(x)=\log_b(x)$ for some base b and $g(x)=a_mx^m+\cdots+a_0$ then f is $\mathcal{O}(g)$ but g is not $\mathcal{O}(f)$. Polynomials grow more slowly than exponentials: if $h(x)=b^x$ for $b>1$ then g is $\mathcal{O}(h)$ but h is not $\mathcal{O}(g)$.

Proof See Exercise 1.65. □

We've defined complexity functions as real number functions rather than natural number ones. The next result ensures that conclusions from the continuous context carry over to the discrete. For instance, if we are considering the two from Example 1.19 as natural number functions then we can shift to the reals, apply L'Hôpital's Rule, and shift back to the naturals.

1.26 LEMMA Let f be a complexity function and consider its restriction $g = f\restriction_{\mathbb{N}}$. Where $L \in \mathbb{R} \cup \{\infty\}$, if $\lim_{x\to\infty} f(x) = L$ then also $\lim_{n\to\infty} g(n) = L$.

Proof Complexity functions are defined for all sufficiently large inputs. In particular, f is defined on all sufficiently large natural number inputs. Therefore since f as a whole approaches the limit L, the restriction g must also approach that limit. □

Although Example 1.11 says that functions are not linearly ordered by Big $\mathcal{O}$, a remarkable fact is that the most important functions do make a line. The table below lists the orders of growth that are most common in practice. Faster-growing functions are nearer the bottom. Many of the rankings come from the L'Hôpital's Rule calculations above.

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	$f_0(n) = \ln(n)$, $f_1(n) = \lg(n^3)$
$\mathcal{O}\big((\lg(n))^2\big)$	Polylogarithmic	$f(n) = (\lg(n))^2 + \lg(n)$
$\mathcal{O}\big((\lg(n))^3\big)$	Also polylogarithmic	$f(n) = (\lg(n))^3 + (\lg(n))^2 + 5$
⋮		
$\mathcal{O}(n)$	Linear	$f(n) = 3n + 4$
$\mathcal{O}(n\lg(n))$	Log-linear	$f_0(n) = 5n\lg(n) + n$, $f_1(n) = \lg(n!)$
$\mathcal{O}(n^2)$	Quadratic	$f(n) = 5n^2 + 2n + 12$
$\mathcal{O}(n^3)$	Cubic	$f(n) = 2n^3 + 12n^2 + 5$
⋮		
$\mathcal{O}(2^{\text{poly}(\lg(n))})$	Quasipolynomial	$f_0(n) = 2^{(\lg(n))^2+3\lg(n)}$, $f_1(n) = n^{\lg(n)}$
⋮		
$\mathcal{O}(2^{\text{poly}(n)})$	Exponential	$f_0(n) = 10\cdot 2^{n^2+4}$, $f_1(n) = 5\cdot 3^n$
⋮		
$\mathcal{O}(n!)$	Factorial	$f(n) = 5\cdot n! + n^{15} - 7$
$\mathcal{O}(n^n)$	–No standard name–	$f(n) = 2\cdot n^n + 3\cdot 2^n$

1.27 TABLE: The order of growth hierarchy (poly() is a polynomial function).

1.28 EXAMPLE We can use this benchmark table to do many of the function comparisons that arise in everyday computing. Suppose that one algorithm runs in time $f(x) = 5x + 3x\lg(x)$ and another runs $g(x) = 3 + 7(\lg x)^3$. Lemma 1.12 and the table give that f is $\Theta(x\lg x)$ while g is $\Theta((\lg x)^3)$. With the table we conclude that g is $\mathcal{O}(f)$ but f is not $\mathcal{O}(g)$. Thus, we needn't do a separate L'Hôpital's Rule calculation for every function comparison.

Tractable vs intractable The motivation behind the development of $\mathcal{O}$ is to understand the resource consumption, and in particular the running time, of algorithms. Our main worry is that an algorithm will just take too long or use too much of some other resource.

That means that for the above table the most important distinction is between

the polynomial and exponential 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.

	$n=1$	$n=10$	$n=100$
$\lg n$	–	1.05×10^{-17}	2.10×10^{-17}
n	3.16×10^{-18}	3.16×10^{-17}	3.16×10^{-16}
$n\lg n$	–	1.05×10^{-16}	2.10×10^{-15}
n^2	3.16×10^{-18}	3.16×10^{-16}	3.16×10^{-14}
n^3	3.16×10^{-18}	3.16×10^{-15}	3.16×10^{-12}
2^n	6.33×10^{-18}	3.24×10^{-15}	4.01×10^{12}

1.29 TABLE: Time taken in years by algorithms with various runtimes. The entries assume a computer running at 10 GHz, 10 000 million ticks per second, which is reasonable for a modern machine. They also use that there are 3.16×10^7 seconds in a year.

In the rightmost column, for the first few rows the relative change is an order of magnitude but the absolute times are small. Then we get to the bottom row. That's not a typo — the final entry really is about 10^{12} years. 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 large.

Another way to understand this is to consider the effect of appending one more character to an input string σ. An algorithm that loops through the characters and so runs in $|\sigma|$ time must do one more loop, about $|\sigma|$ more ticks. But an algorithm that takes $2^{|\sigma|}$ time will take about double the time.

Cobham's thesis is that tractable problems — those that are conceivably solvable in practice — are those for which there is an algorithm whose resource consumption is at most polynomial. If a problem's best available algorithm runs in exponential time then, as we understand things today, the problem is 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. But we will nonetheless pause to elaborate on five points that can cause confusion. For the first, contrast these.

```
(define (g0 n)
  (for ([i (in-range n)])
    (let ([x (* n n)])
      (for ([j (in-range n)])
        (printf "~a " (+ x i j))))))
```

```
(define (g1 n)
  (let ([x (* n n)])
    (for ([i (in-range n)])
      (for ([j (in-range n)])
        (printf "~a " (+ x i j))))))
```

They have the same output but different run times. On the left the code sets the local variable x inside the outer loop, which makes it $n-1$ assignments slower than what's on the right. But our definition gives them both the runtime performance of $\mathcal{O}(n^2)$ so the $n-1$ tick difference is disregarded. Big $\mathcal{O}$ does not capture fine details of implementation.

[†] More discussion on Cobham's Thesis is on page 305.

That leads to the second point. 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, on a RAM model there are registers which each holds an entire integer. And whereas a Turing machine spend lots of steps traversing the tape, the RAM model gets or sets a 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 many 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, the cost of an algorithm is fixed to a machine model. (The most commonly used model is the Turing machine.)

1.30 DEFINITION A machine $\mathcal{M}$ with input alphabet Σ takes time $t_{\mathcal{M}}: \Sigma^* \to \mathbb{N} \cup \{\infty\}$ if that function gives the number of steps that the machine takes to halt on the input $\sigma \in \Sigma^*$. If it does not halt then $t_{\mathcal{M}}(\sigma) = \infty$. The machine runs in input length time $\hat{t}_{\mathcal{M}}: \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ if $\hat{t}_{\mathcal{M}}(n)$ is the maximum of the $t(\sigma)$ over all inputs $\sigma \in \Sigma^*$ of length n. The machine runtime is $\mathcal{O}(f)$ if $\hat{t}_{\mathcal{M}}$ is $\mathcal{O}(f)$.

Our third point about Big $\mathcal{O}$ is that its definition ignores constant factors: does that reduce its value for comparing algorithms? Suppose that an algorithm takes time given by Cn^2 for inputs $n > N$. One of the roles of theory is to be a guide to practice and we can worry that there is a big in-practice difference between an enormous C and a tiny one. Similarly, if N is huge then perhaps our description of the algorithm's runtime behavior is not complete until we understand what happens to inputs less than that number.

Part of the reason that often constants are not specified is that finding them is hard.[‡] For one thing, they are affected by low-level machine details such as the memory addressing and paging, cache hits, and whether the CPU can do some operations in parallel. What's more, experience shows that doing the work to find these numbers often does not change the selection of the best algorithm to use. As Table 1.29 illustrates, knowing at a Big $\mathcal{O}$ level how the algorithm grows is, past some point, more important than knowing the values of the constants.

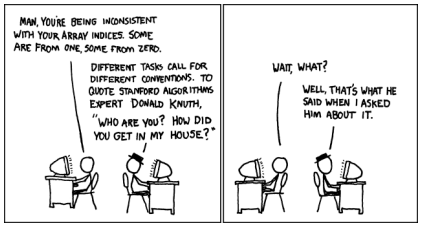
Donald Knuth
b 1938

We could imagine that instead of analyzing particular commercial machines, we all agree on a reference architecture. This is D Knuth's approach in the monumental *Art of Computer Programming* series. But as in the commercial case there is a risk that the standard gets updated. Then published results longer apply, and it is all to do again. Instead, 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. Being reference independent has advantages, especially in a quickly changing area.

This echos the paragraph at the start of this discussion. Big $\mathcal{O}$ makes relative comparisons. Not specifying, say, the precise difference between the cost of a

[†] More on the relationship between algorithms and machine models is in Section 3. [‡] Authors do sometimes state their order of magnitude.

division and the cost of a memory write (within reasonable limits) emphasizes the relative factors. 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 relative statements such as "from here, New York City is twice as far as Boston." Big $\mathcal{O}$ ignores constants because that is inherent in being a unit free, relative measurement.

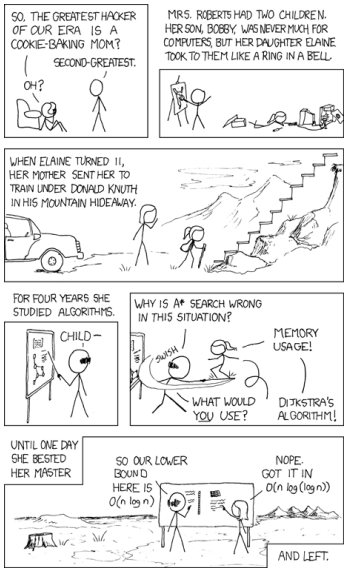


Courtesy xkcd.com

Thus, if we have an algorithm that on input of size n will take $3n$ ticks then we say it is $\mathcal{O}(n)$ simply to describe that doubling the input size will roughly double the number of steps taken. If an algorithm is $\mathcal{O}(n^2)$ then doubling the input size will approximately quadruple the number of steps. We don't want to say more.

This leads to our fourth point. 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 $k \in \{2, \ldots\ n-1\}$ to see if it divides n. The worst case is that it tests all of those numbers, about n of them. Take the size of n to be the number of bits needed to represent n, approximately $\lg n$. Thus 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 exponentiation. 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?



Courtesy xkcd.com

This is about the relationship between the algorithm and the underlying computing model. The programming class may make the engineering judgment that for every use of the program, the input will fit into a 64 bit word. This chooses a computation model like the RAM model, where large numbers take the same time to read as small ones. For this model the relationship between size of the input and the runtime is linear.

This is in part a theory-versus-application thing. In a common programming setting the input is bounded while in a theoretical setting the algorithm accepts arbitrarily large input. An algorithm whose runtime 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 pseudo polynomial.

One last point about Big $\mathcal{O}$. When analyzing an algorithm we usually use the behavior that is the worst case for any input of that size. We could instead consider

the average over all inputs of that size. For instance, the quick sort algorithm takes quadratic time $\mathcal{O}(n^2)$ at worst but on average is $\mathcal{O}(n \lg n)$.

Related to that, consider a machine with this runtime behavior.

$$\hat{t}(n) = \begin{cases} n! & \text{– } n \text{ is a power of ten} \\ n & \text{– otherwise} \end{cases}$$

This is worst-case super exponential time for rare inputs (called "black holes"). However, for most inputs it would be quite fast.[†]

V.1 Exercises

1.31 True or false: if a function is $\mathcal{O}(x^2)$ then it is $\mathcal{O}(x^3)$.

✓ 1.32 Your classmate emails you a draft of an assignment answer that in part says, "The algorithm has a running time that is $\mathcal{O}(n^2)$. So with input $n = 5$ it will take 25 ticks." Make two corrections.

1.33 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.

✓ 1.34 How many bits does it take to express each number in binary? (A) 5 (B) 50 (C) 500 (D) 5 000

✓ 1.35 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 $\mathcal{O}(g)$.

1.36 For each, state whether it is a correct application of the statement "This algorithm runtime is $\mathcal{O}(n^2)$" to $n = 61\,669$.

(A) Roughly, it will take time that is the square of 61 669.

(B) Roughly, it will take time that is the square of the number of digits in 61 669, namely 5.

(C) Roughly, it will take time that is the square of the number of bits in 61 669, namely 16.

(D) None of these.

✓ 1.37 For each find the benchmark function in the order of growth hierarchy, Table 1.27, with same order of growth.

(A) $n^2 + 5n - 2$

(B) $2^n + n^3$

(C) $3n^4 - \lg \lg n$

(D) $\lg n + 5$

1.38 For each give the function on the order of growth hierarchy, Table 1.27, that has the same order of growth. That is, find g in that table where f is $\Theta(g)$.

(A) $f(n) = \begin{cases} n & \text{– if } n < 100 \\ 0 & \text{– else} \end{cases}$

[†] An in-practice example of this is the simplex algorithm, which is very widely used for linear optimization. It is worst-case exponential but typically seems to run in polynomial time.

(B) $f(n) = \begin{cases} 1\,000\,000 \cdot n & \text{– if } n < 10\,000 \\ n^2 & \text{– else} \end{cases}$

(C) $f(n) = \begin{cases} 1\,000\,000 \cdot n^2 & \text{– if } n < 100\,000 \\ \lg n & \text{– else} \end{cases}$

1.39 For each pair of functions, decide whether f is $\mathcal{O}(g)$, or g is $\mathcal{O}(f)$, or both, or neither, by using L'Hôpital's Rule on the ratio f/g. (A) $f(n) = 3n^3 + 2n + 4$, $g(n) = \lg(n)+6$ (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) $f(n) = \lg(n)$, $g(n) = \ln(n)$ (E) $f(n) = n^2 + \lg(n)$, $g(n) = n^4 - n^3$ (F) $f(n) = 55$, $g(n) = n^2 + n$

1.40 (IIS, IIT 2021) Consider the functions $f_1 = 10^n$, $f_2 = n^{\ln n}$, and $f_3 = n^{\sqrt{n}}$. Which of these arranges the functions in increasing order of asymptotic growth rate? (A) f_3, f_2, f_1, (B) f_2, f_1, f_3, (C) f_1, f_2, f_3, (D) f_2, f_3, f_1.

1.41 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) $\lg(5^n)$

✓ 1.42 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 $\mathcal{O}(2^{\text{poly}(n)})$

1.43 Find the smallest power $k \in \mathbb{N}$ 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) $\lfloor \sqrt{5n^7 - 2n^2} \rfloor$

✓ 1.44 For each pair of functions, decide if f is $\mathcal{O}(g)$, or g is $\mathcal{O}(f)$, or both, or neither, by simplifying using Lemma 1.12 and Table 1.27. (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) $f(n) = n^{1.2} + \lg n$, $g(n) = n^{\sqrt{2}} + 2n$

1.45 For each pair, decide whether f is $\mathcal{O}(g)$, or g is $\mathcal{O}(f)$, or both, or neither, by simplifying using Lemma 1.12 and Table 1.27. (A) $f(n) = n^6$, $g(n) = 2^{n/6}$ (B) $f(n) = 3^n$, $g(n) = 3 \cdot 2^n$ (C) $f(n) = \lg(3n)$, $g(n) = \lg(n)$

1.46 Let $Z \colon \mathbb{R} \to \mathbb{R}$ be the zero function, $Z(x) = 0$. Show that Z is $\mathcal{O}(g)$ for every complexity function g.

1.47 To Table 1.29 add a column for $n = 1000$.

✓ 1.48 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

1.49 What is the least input number such that $f(n) = 100\,000 \cdot n^2$ is less than $g(n) = n^3$?

1.50 What is the order of growth of the run time of a deterministic Finite State machine?

✓ 1.51 (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 an only if there exists $L \in \mathbb{R}$ so that $f(x) \leq L$ for all inputs $x \in \mathbb{R}$.

1.52 Where does $g(x) \leq x^{\mathcal{O}(1)}$ place the function g in the order of growth hierarchy? *Hint:* see the last item in the prior question.

1.53 Prove that 2^n is $\mathcal{O}(n!)$. *Hint:* because of the factorial, directly use Definition 1.6 and find suitable $N, C \in \mathbb{N}$.

1.54 Use L'Hôpital's Rule as in Example 1.22 to verify these for any $d \in \mathbb{R}^+$: (A) $(\log_b(x))^2$ is $\mathcal{O}(x^d)$ (B) $(\log_b(x))^3$ is $\mathcal{O}(x^d)$ (C) for any $k \in \mathbb{N}^+$, $(\log_b(x))^k$ is $\mathcal{O}(x^d)$.

✓ 1.55 The Halting problem function K is uncomputable. Place its rate of growth in the order of growth hierarchy, Table 1.27.

✓ 1.56 Show that 2^x is in $\mathcal{O}(3^x)$ but 3^x is not in $\mathcal{O}(2^x)$. Also show that 3^x is $\mathcal{O}(2^{\text{poly}(x)})$.

1.57 Table 1.27 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 $\Theta(n^n)$?

1.58 Two complexity functions f, g are asymptotically equivalent, $f \sim g$, if $\lim_{x\to\infty}(f(x)/g(x)) = 1$. Show that each pair is asymptotically equivalent: (A) $f(x) = x^2 + 5x + 1$ and $g(x) = x^2$, (B) $\lg(x + 1)$ and $\lg(x)$.

1.59 Is there an f so that $\mathcal{O}(f)$ is the set of all polynomials?

1.60 Prove that $x^{\lg x}$ is quasipolynomial.

1.61 The prior exercise shows that $f(x) = x^{\lg x}$ is quasipolynomial. Here we will show that the order of growth of f is strictly between polynomial and exponential. (A) As a preliminary for the next item, show that for any constant $k \in \mathbb{R}$ we have $k \cdot \lg x$ is $\mathcal{O}((\lg x)^2)$ but $(\lg x)^2$ is not $\mathcal{O}(k \cdot \lg x)$. (B) Show that f's growth is strictly greater than polynomial by arguing that for any constant power k, we have $x^k \in \mathcal{O}(x^{\lg x})$ but $x^{\lg x} \notin \mathcal{O}(x^k)$. *Hint:* take the ratio, rewrite using the identity $a = 2^{\lg a}$, and consider the limit of the exponent. (C) As a preliminary for the next item, show that $(\lg x)^2$ is $\mathcal{O}(x)$ but x is not $\mathcal{O}((\lg x)^2)$. (D) Show that f's growth is strictly less than exponential by arguing that $x^{\lg x}$ is $\mathcal{O}(2^x)$ but 2^x is not $\mathcal{O}(x^{\lg x})$. *Hint:* again take the ratio, again rewrite using the identity, and again consider the limit of the exponent.

1.62 Verify the clauses of Lemma 1.12. Take all functions to be complexity functions. (A) If $a \in \mathbb{R}^+$ then af is also $\mathcal{O}(g)$. (B) If f_1 is $\mathcal{O}(f_0)$ then the functions $f_0 + f_1$ and $f_0 - f_1$ are also $\mathcal{O}(f_0)$. (C) If f_1 is $\mathcal{O}(f_0)$, we cannot always conclude that $f_0 f_1$ is $\mathcal{O}(f_0)$

1.63 Verify these clauses of Lemma 1.15. (A) The Big-$\mathcal{O}$ relation is reflexive. (B) It is also transitive.

1.64 We will prove the first of Theorem 1.17's three items; the other two are similar. (This requires the precise definition of a limit, which may be outside of the

prerequisites for this course: $\lim_{x\to\infty} h(x) = 0$ if for all $\varepsilon > 0$ there is an $M \in \mathbb{R}$ such that $x \geq M$ implies that $|h(x)| < \varepsilon$.) Assume that $\lim_{x\to\infty} f(x)/g(x)$ exists and equals 0. (A) Show that f is $\mathcal{O}(g)$. (B) Show that g is not $\mathcal{O}(f)$.

1.65 Prove Lemma 1.25.

Section V.2 A problem miscellany

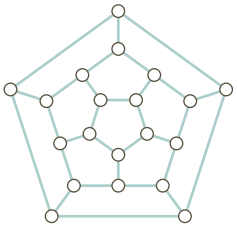
Much of today's work in the Theory of Computation is driven by problems that originate outside of the subject. We will describe some of these problems to get a sense of the ones that people work on and also to use for examples and exercises. All of these problems are well known to anyone in the field.

Problems with stories We start with a few that form part of the folklore. These anecdotes are fun and an important part of the culture, and they also give a sense of how problems arise in general.

William Rowan Hamilton 1805–1865

W R 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.

One of his ventures was a game, *Around the World*. The vertices in the graph below were holes in a wooden board, imagined as 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.

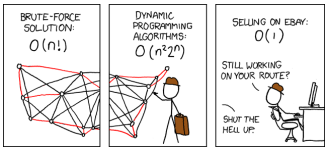
A special case is the Knight's Tour problem, to use a chess knight to make a circuit of the squares on the board. (Recall that a knight moves three squares at a time, with the first two squares in one direction and then the third one perpendicular to that direction.)

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 $S = \{c_0, \ldots c_{k-1}\}$ '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

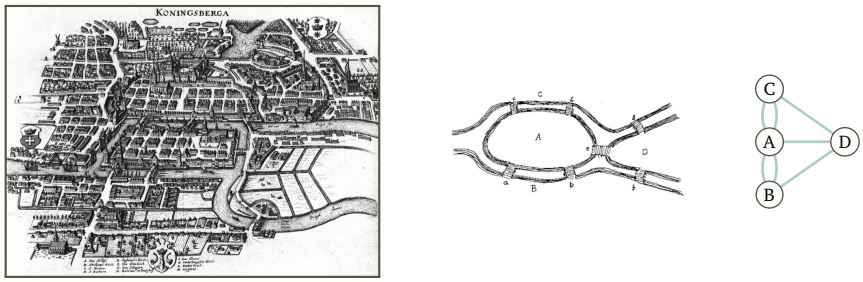
We can start with a map of the state capitals of the forty eight contiguous US states and the distances between them: Montpelier VT to Albany NY is 254 kilometers, etc. From among all trips that visit each city and return back to the start, such as Montpelier → Albany → Harrisburg → ··· → Montpelier, we want the shortest one.

As stated, this is an optimization problem. However we can recast it as a 'yes' or 'no' decision problem. Introduce a parameter bound number B 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 quickly solve the optimization problem, by first asking whether there is a trip bounded by length $B = 1$, then asking if there is a trip of length $B = 2$, etc. When we eventually get a 'yes', we know the shortest length.

The next problem sounds much like Hamiltonian Circuit, in that it involves exhaustively traversing a graph. But it proves to act very differently.

Leonhard Euler
1707–1783

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.



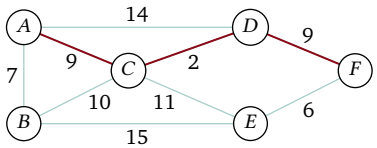
Euler's summary sketch is in the middle and the graph is on the right.

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.



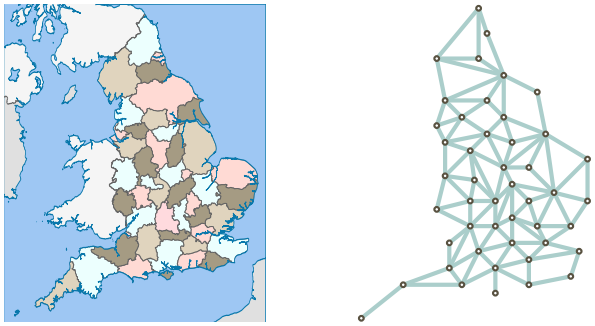
The next problem was discovered in 1852 by a young mathematician, F Guthrie, who was drawing a map of England's counties. He wanted to color them with different colors for counties that share a border. His map required only four colors and he conjectured that for any map, four colors were enough.

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 2.46). 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.

[†] Dijkstra's algorithm is at worst quadratic in the number of vertices.



2.6 ANIMATION: Counties of England and the derived planar graph

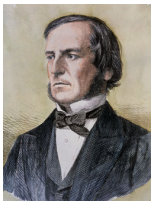
Appel and Haken's post office celebrating

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, $\mathcal{N} = \mathcal{C}_0 \cup \cdots \cup \mathcal{C}_{k-1}$, such that no two same-set vertices are connected.

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, for 'true' or 'false'. 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.



George Boole
1815–1864

$$f(P,Q,R) = (P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (\neg P \vee \neg Q \vee \neg R)$$

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, CNF, if it consists of clauses that are connected with $\wedge$'s, where inside each clause are variables or negations connected by $\vee$'s. The expression above is in that form. The truth table below shows the input-output behavior of the function defined by the expression.

[†] This is in contrast to the *Entscheidungsproblem*.

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.

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 supposition 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.[†]

2.12 EXAMPLE These are two Western-tradition constellations, Ursa Minor and Draco.

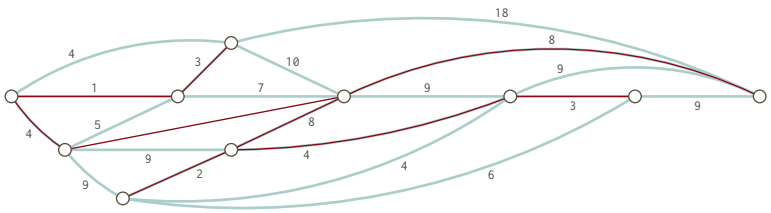
Here we can solve the Vertex-to-Vertex Path problem by eye. For any two vertices in Ursa Minor there is a path and for any two vertices in Draco there is a path. But if the two vertices are in different constellations then there is no path.

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.

[†] The name Vertex-to-Vertex Path is nonstandard. It is usually known as st-Path, st-Connectivity, or STCON (s and t stand for vertices).

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.

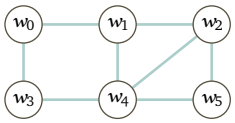


The highlighted subgraph includes all of the vertices, that is, it spans the graph. In addition, its weights total to a minimum from among all of the spanning subgraphs. From that it follows that this subgraph is a tree, meaning that it has no cycles, or else we could eliminate an edge from the cycle 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 such that for any edge, at least one of its ends is a member of that set.

2.15 EXAMPLE A museum posts guards to watch their exhibits. There are eight halls, laid out as edges below. They will put the guards at some of the corners $w_0, \ldots w_5$. What is the smallest number of guards that suffices to watch all of the hallways?

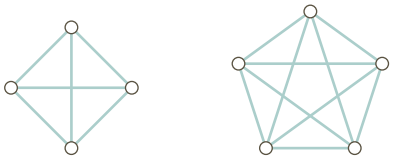


Checking each corner shows that one guard will not suffice. The two-element set $C = \{w_0, w_4\}$ is a vertex cover: every hallway has at least one end in C.

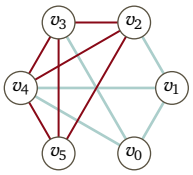
2.16 PROBLEM (Clique) Given a graph and a bound $B \in \mathbb{N}$, decide if the graph has a size B set vertices such that any two are connected.

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 a subgraph like the one below on the left and any graph with a 5 clique has a subgraph like the one the right.



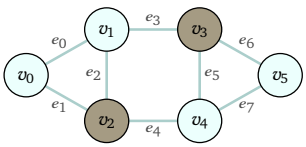
2.17 Example This graph has a 4-clique.



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 This graph cut into differently colored vertices gives the cut set $\{e_1, e_2, e_3, e_4, e_5, e_5\}$ of size six, the largest possible.[†]



2.21 Animation: A partition for the graph with a maximum cut set.

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 $X = \{\mathsf{a}, \mathsf{b}\}$, $Y = \{\mathsf{b}, \mathsf{c}\}$, and $Z = \{\mathsf{a}, \mathsf{d}\}$, so that $n = 2$. Below is a subset of $X \times Y \times Z$ (it actually equals $X \times Y \times Z$).

$$M = \{\langle \mathsf{a}, \mathsf{b}, \mathsf{a}\rangle, \langle \mathsf{a}, \mathsf{c}, \mathsf{a}\rangle, \langle \mathsf{b}, \mathsf{b}, \mathsf{a}\rangle, \langle \mathsf{b}, \mathsf{c}, \mathsf{a}\rangle, \langle \mathsf{a}, \mathsf{b}, \mathsf{d}\rangle, \langle \mathsf{a}, \mathsf{c}, \mathsf{d}\rangle, \langle \mathsf{b}, \mathsf{b}, \mathsf{d}\rangle, \langle \mathsf{b}, \mathsf{c}, \mathsf{d}\rangle\}$$

The set $\hat{M} = \{\langle \mathsf{a}, \mathsf{b}, \mathsf{a}\rangle, \langle \mathsf{b}, \mathsf{c}, \mathsf{d}\rangle\}$ 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 $X \times Y \times Z$.

$$\begin{aligned} M = \{ &\langle 1, 10, 200\rangle, \langle 1, 20, 300\rangle, \langle 2, 30, 400\rangle, \langle 3, 10, 400\rangle, \\ &\langle 3, 40, 100\rangle, \langle 3, 40, 200\rangle, \langle 4, 10, 200\rangle, \langle 4, 20, 300\rangle \} \end{aligned}$$

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 $S = \{n_0, \ldots n_{k-1}\}$ and a target $T \in \mathbb{N}$, decide if a subset of S sums to the target.[‡]

[†] One way to verify this is with a script that checks all two-set partitions of the vertices. [‡] 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.

2.26 EXAMPLE Do some of the numbers $\{911, 22, 821, 563, 405, 986, 165, 732\}$ add to $T = 1173$? The answer is 'yes'; 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 multiset's numbers are multiples of three while the target 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	i_0	i_1	i_2	i_3
Weight	3	4	5	6
Value	50	40	10	30

The answer is 'no'. The highest value that we can get without exceeding the bound is from taking i_0 and i_1, which does not meet the target.

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 $A = \{\texttt{I}, \texttt{a}, \texttt{my}, \texttt{go}, \texttt{rivers}, \texttt{cat}, \texttt{hotel}, \texttt{comb}\}$ has eight words. The size of a word is the number of letters. Then $\hat{A} = \{\texttt{cat}, \texttt{rivers}, \texttt{I}, \texttt{a}, \texttt{go}\}$ 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.

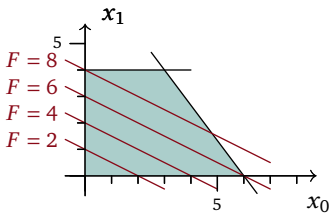
Reps	*Num states*	*States*
55	1	CA
38	1	TX
30	1	FL
28	1	NY
19	2	IL, PA
17	1	OH
16	2	GA, NC
15	1	MI
14	1	NJ
13	1	VA
12	1	WA

Reps	*Num states*	*States*
11	4	AZ, IN, MA, TN
10	5	CO, MD, MN, MO, WI
9	3	AL, SC
8	3	KY, LA, OR
7	3	CT, OK
6	6	AR, IA, KS, MS, NV, UT
5	3	NE, NM
4	7	HI, ID, ME, MT, NH, RI, WV
3	7	AK, DE, DC, ND, SD, VT, WY

The table above gives the numbers for the 2024 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 $F(x_0, \ldots x_n) = c_0x_0 + \cdots + c_nx_n$ subject to linear constraints, ones of the form $a_{i,0}x_0 + \cdots + a_{i,n}x_n \leq b_i$ or $a_{i,0}x_0 + \cdots + a_{i,n}x_n \geq b_i$.

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?

C	A	B
A	G	E
D	O	G

2.36 Animation: Instance of the Crossword problem

2.37 Problem (Fifteen Game) Given an $n \times n$ grid holding tiles numbered $1, \ldots\ n^2 - 1$, 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 find divisors of 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 d that divide n and such that $1 < d < n$.

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.[#]

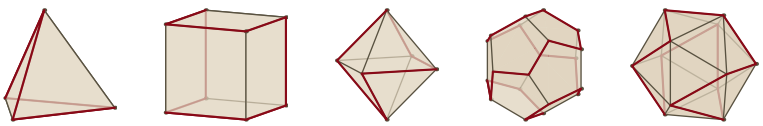
Nitin Saxena (b 1981), Neeraj Kayal (b 1979), Manindra Agrawal (b 1966)

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 many well-respected experts have worked on it without success, nonetheless it could have a solution that is fast. Although opinions of experts have value, 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.

V.2 Exercises

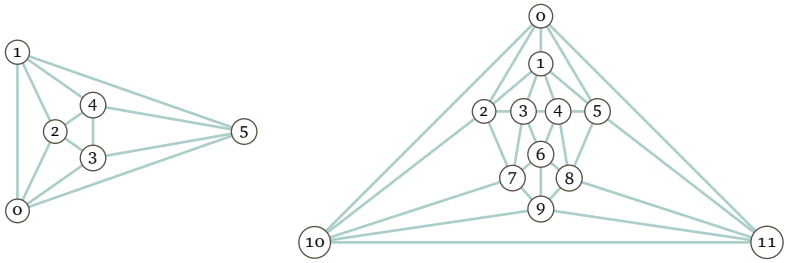
✓ 2.41 Each of the five Platonic solids has a Hamiltonian circuit, as shown.



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 expanded a face in the back of each

[†] No efficient algorithm is known, that is, on a non-quantum computer. [‡] The presumed difficulty of this problem is at the heart of widely used algorithms in cryptography. [#] 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.

solid until we could squash the entire shape down into the plane without any edge crossings.



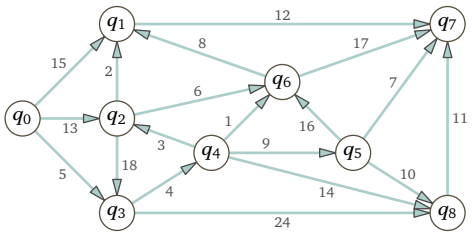
2.42 How many edges are there in a Hamiltonian path?

2.43 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.

2.44 Find the shortest path in this graph



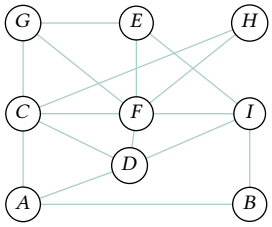
(A) from q_2 to q_7, (B) from q_0 to q_8, (C) from q_8 to q_0.

2.45 Give a planar map that requires four colors.

2.46 (A) Some countries have separated components; for instance, where Königsberg used to be is today a part of Russia called the Kaliningrad Oblast that is unconnected to rest of the country. The Four Color problem disallows such countries. Give a planar map where countries have non-contiguous components that requires five colors. (B) The problem also takes '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.

✓ 2.47 The Course Scheduling problem starts with a list of students and the classes that they wish to take. If a student wishes to take two classes then those two will not be scheduled to meet at the same time. The goal is to find the minimum number of time slots needed to schedule the classes (scheduling every class at

a different time would give no conflicts). The school below offers Astronomy, Biology, Computing, Drama, English, French, Geography, History, and Italian. After students sign up, the graph shows overlaps. 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 that meet 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.

✓ 2.48 Decide if each formula is satisfiable.

(A) $(P \wedge Q) \vee (\neg Q \wedge R)$

(B) $(P \rightarrow Q) \wedge \neg((P \wedge Q) \vee \neg P)$

✓ 2.49 Instead of going from the propositional logic statement to its truth table, we can specify a behavior in a truth table and then produce a statement with that behavior. That statement can be in Conjunctive Normal form. Appendix C says more but here we give an algorithm to find the statement.

P	Q	R	$f(P,Q,R)$
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) We will target the lines that end in F's. The first is the F-T-F line. With it associate the clause $P \vee \neg Q \vee R$, where for F take the variable name and for T take the negation of the name, and join them with $\vee$'s. For each F line, produce such a clause that targets it.

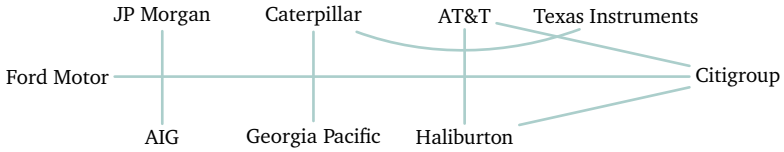
(B) Argue that each clause is F only on its targeted line.

(C) Produce the Boolean expression that consists of the joins those clauses with $\wedge$'s (it is the conjunction of the clauses, which is where this form gets its name). Argue that that it must have the desired truth table.

2.50 If a Boolean expression F is satisfiable, does that imply that its negation $\neg F$ is not satisfiable?

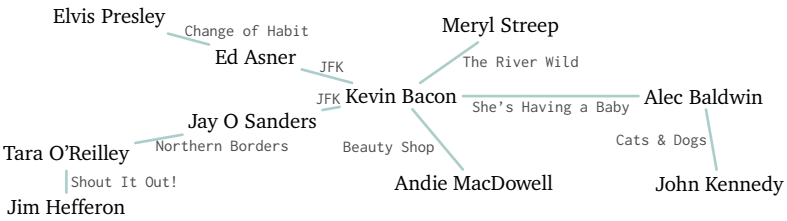
2.51 Some authors define the Satisfiability problem as: given a finite set of propositional logic statements, not just one statement, find if there is a single input tuple $b_0, \ldots b_{j-1}$, where each b_i is either T or F, that satisfies them all. Show that this is equivalent to the definition given in Problem 2.9.

✓ 2.52 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?

✓ 2.53 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?

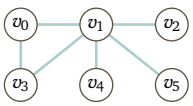
(B) John Kennedy's (no, it is not that John Kennedy)?

(C) Bacon's?

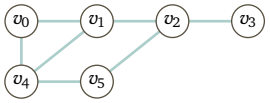
(D) How many movies separate me from Meryl Streep?

2.54 The Vertex Cover problem inputs a graph $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ and a number $B \in \mathbb{N}$, and asks if there is a subset S of at most B vertices such that for each edge $e \in \mathcal{E}$ at least one endpoint is an element of S. The Independent Set problem inputs a graph and a number $\hat{B} \in \mathbb{N}$ and asks if there is a subset $\hat{S}$ with at least $\hat{B}$ vertices such that for each edge at most one endpoint is in $\hat{S}$. The two are related.

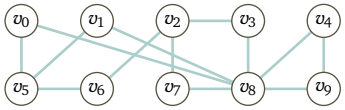
(A) In this six-vertex graph find a vertex cover S with $B = 2$ elements. Find an independent set with $\hat{B} = 4$ elements.



(B) In this six-vertex graph find a vertex cover with $B = 3$ elements, and an independent set with $\hat{B} = 3$ elements.



(C) Find a vertex cover S with $B = 4$ elements in this ten-vertex graph. Find an independent set $\hat{S}$ with $\hat{B} = 6$ elements.

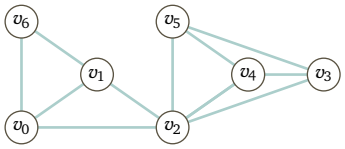


(D) Prove that S is a vertex cover if and only if its complement $\hat{S} = \mathcal{N} - S$ is an independent set.

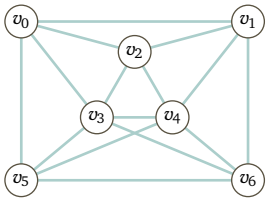
✓ 2.55 What shape is a 3-clique? A 2-clique?

2.56 How many edges does a k-clique have?

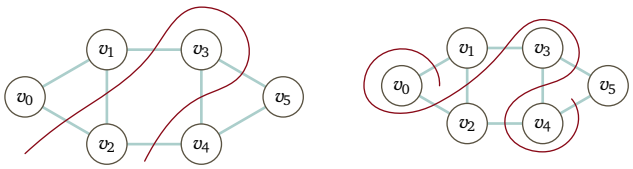
✓ 2.57 List all seven of this graph's 3-cliques.



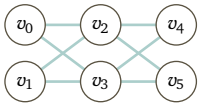
2.58 Does this graph have a 3-clique? A 4-clique? A 5-clique?



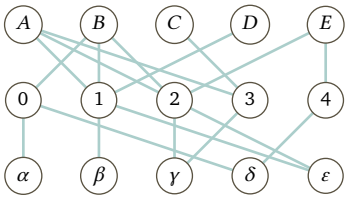
2.59 Example 2.20 exhibits a cut set with six members, as shown on the left. But on the right there are eight cut edges; what's wrong with it?



2.60 Find the maximum cut set for this graph.



✓ 2.61 A college department has instructors A, B, C, D, and E. They need placing into classes 0, 1, 2, 3, and 4. The available time slots are α, β, γ, δ, and ε. This shows which instructors can teach which classes, and which classes can run in which slots.



For example, instructor A can only teach courses 1, 2, and 3. And, course 0 can only run in slots α or δ. Verify that this is an instance of the Three-dimensional Matching problem and find a match.

2.62 Consider Three Dimensional Matching, Problem 2.22. Let $X = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, $Y = \{\mathsf{b}, \mathsf{c}, \mathsf{d}\}$, and $Z = \{\mathsf{a}, \mathsf{d}, \mathsf{e}\}$. (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 entries agree on any coordinate?

2.63 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.

✓ 2.64 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.

2.65 Using the data in Example 2.31, decide if there could be a tie in the 2024 Electoral College.

2.66 Name the prime numbers less than one hundred.

2.67 Decide if each is prime.
(A) 5 477
(B) 6 165
(C) 6 863
(D) 4 207
(E) 7 689

✓ 2.68 Find a divisor of each that is nontrivial (meaning not 1 or the number itself), if one exists. (A) 31 221 (B) 52 424 (C) 9 600 (D) 4 331 (E) 877

2.69 Your friend asks, "Doesn't the polytime solution of Primality automatically give us one for Divisor? Just take the divisor from the first problem's solution and use it to solve the second problem." Help them out.

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. That description should suffice 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 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 $n > 1$, successively try every $k \in (1 \,..\, n)$ to see if it divides n (because it does the steps iteratively, this is not targeting a massively parallel model). We could implement that algorithm with a program written in Racket.

[†] 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. [#] There are now coming up on a million volunteers offering computing time. To join them, visit `https://scienceunited.org/`.

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, there may be many solutions and the algorithm can stop when it has found any 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 k-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 we exhibit 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 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 $\{\sigma \in \mathbb{B})^* \,\big|\, \sigma \text{ is the binary representation of a prime } n \in \mathbb{N}\}$. A problem instance is where we are given a particular bitstring representing a number and we must decide whether it is a set member.

This relates to the discussion from the Languages section, on page 145, about the distinction between deciding a language and recognizing it.

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, 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.

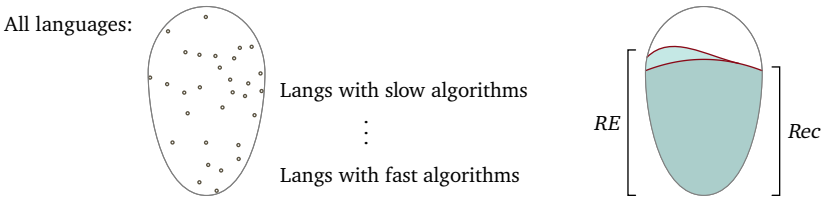
Restated, $\mathcal{P}$ decides the language $\mathcal{L}$ if $\phi_{\mathcal{P}}(\sigma) = \mathbb{1}_{\mathcal{L}}(\sigma)$, that is, if $\phi_{\mathcal{P}}(\sigma) = 1$ when $\sigma \in \mathcal{L}$ and 0 otherwise. In particular, a language decider halts for all inputs. (By

[†] 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 it is true.

contrast, if $\mathcal{P}$ recognizes a language then when $\sigma \notin \mathcal{L}$, possibly the machine halts and returns a value other than 1 but possibly also it does not halt.)

Distinctions between problem types can be fuzzy. In addition, often if we have a task then we could describe it with more than one problem 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 may write, "Let $\mathcal{L}$ be a problem . . ."



3.2 FIGURE: Both of these show the collection of languages, which we often call the 'problems'. On the left, the points 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* is the languages that are computably enumerable. So this diagram makes the point that not all languages have a computable decider or a recognizer — some are unsolvable.

3.3 REMARK One reason that we are interested in language membership decisions comes from practice. For instance, a language compiler must recognize whether a given source file is a member of the language.

Recall also 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 $\mathcal{L} = \{\sigma \in \{a, b\}^* \mid \sigma \text{ contains at least two b's}\}$. Thus, we can compare the power of Finite State machines with that of other machines by comparing the list of languages that they can decide. The same holds for Pushdown machines.

Still another reason to express problems in terms of deciding a language is that in many contexts stating a problem in this way is natural, as we saw with the Halting problem.

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 $\mathsf{SAT} = \{E \mid E \text{ is a satisfiable propositional logic statement}\}$. This restatement is trivial, suggesting that the language recognition problem form is often a natural way to describe the underlying task.

A pattern that we have seen is recasting optimization problems as language decision problems, by using bounds.

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. As stated it is an optimization problem. Recast it by considering the family of languages, $\mathcal{L}_B = \{\mathcal{G} \;\big|\; \mathcal{G} \text{ that has a } B\text{-coloring}\}$ for the parameter $B \in \mathbb{N}^+$. If we could solve the the $\mathcal{L}_B$ problems then given a graph $\mathcal{G}$ we could compute its 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 another optimization problem. Like the prior example we can recast it as a family of language decision problems $\mathcal{TS}_B = \{\mathcal{G} \;\big|\; \text{the graph } \mathcal{G} \text{ has a circuit of length no more than } B\}$ for the parameter $B \in \mathbb{N}$.

What is important about these recastings of optimizations is that they preserve polytime solvability. For instance, for the Traveling Salesman problem, if there is a $k \in \mathbb{N}$ so that for each B we could solve $\mathcal{TS}_B$ in time $\mathcal{O}(n^k)$, then looping through $B = 1$, $B = 2$, etc., will solve the optimization problem in polytime, namely time $\mathcal{O}(n^{k+1})$.

We sometimes recast using a particular bound as a parameter, such as if we want to know whether there is a Traveling Salesman path through the US lower 48 state capitals of total length at most $B = 10\,000$.

$$\mathcal{L}_B = \{\mathcal{G} \;\big|\; \text{the graph has a circuit of length at most } B\}$$

But we sometimes instead consider all of the bounds at once.

$$\mathcal{L} = \{\langle \mathcal{G}, B\rangle \;\big|\; \text{the graph has a circuit of length at most } B\}$$

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 $E(P_0, P_1 \ldots P_k)$ and returns an assignment of truth values for P_0, P_1, ... 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 $E(P_0 = F, P_1, P_2 \ldots P_k)$ 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.

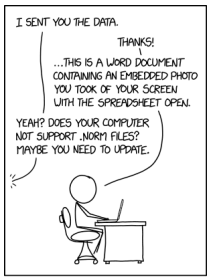
One more point. Some shifts between problem types are not as smooth as the ones mentioned above. Consider the function problem 'given a polynomial p, return its rational roots'. In restating it as a decision problem perhaps its most natural language is this.

$$\mathcal{L} = \{ \langle p, r \rangle \mid p \text{ is a polynomial, } r \text{ is rational, and } p(r) = 0 \}$$

However, this does not capture the essential difficulty in the first statement, which is to produce the root rather than just verify it.

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 average' 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 integers written in unary.

This matters in that the input's form can change the algorithm that we choose. 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`.[†] If the input is in unary then we have a different algorithm entirely.



Courtesy `xkcd.com`

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.[‡]

However, the representations can significantly change the run time. Suppose that a Turing machine inputs and outputs numbers. If the input representation is binary and the output representation is unary then the time that the machine takes is exponential in the input size, just from the time to write output to the tape.

With these points 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

[†] An interesting point about this case is that on a Turing machine, if the machine starts with its I/O head under the final character then the machine only needs to read the final two characters. This is an algorithm for a nontrivial problem that does not read its entire input to come to a decision. It runs in time independent of the input length, time that is $\mathcal{O}(1)$. [‡] That is, the unary case reduces to the binary one. [#] 'Reasonable' means that it is not so inefficient as to greatly change the big-$\mathcal{O}$ behavior.

Master.[†]

3.8 FIGURE: *Basket of Fruit* by Caravaggio (1571–1610)

Basically, if we are circumspect about representations then we usually will not see something like exponential blowup. Most notably, our default is to represent integers in binary and rational numbers as pairs of integers. 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 usually do this. When we do discuss representations, we use $\text{str}(x)$ to denote a convenient, reasonably efficient, bitstring representation of x.[‡]

V.3 Exercises

✓ 3.9 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) pseudo code (C) a Turing machine (D) a flowchart (E) source code (F) an executable (G) a process

3.10 Your friend asks, "So, if a problem is essentially a set of strings, what constitutes a solution?" Answer them.

3.11 What is the difference between a decision problem and a language decision problem?

[†] 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. [‡] Some authors use diamond brackets to stand for a representation, as in $\langle \mathcal{G}, v_0, v_1 \rangle$. In this book we reserve them for sequences.

3.12 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 human eye has a resolution of 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?

3.13 Name something important that cannot be represented in binary.

✓ 3.14 True or false: any two programs that implement the same algorithm must compute the same function. What about the converse?

3.15 Some tasks are hard to express as language decision problems. 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) $\{\langle \sigma, p \rangle \mid p \text{ is a permutation that orders } \sigma\}$

✓ 3.16 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) $\mathcal{L}_1 = \{\sigma \in \{0, \ldots 9\}^* \mid \sigma \text{ represents in decimal a multiple of } 100\}$
(C) $\mathcal{L}_2 = \{\sigma \in \mathbb{B}^* \mid \sigma \text{ has more 1's than 0's}\}$
(D) $\mathcal{L}_3 = \{\sigma \in \mathbb{B}^* \mid \sigma^R = \sigma\}$

3.17 Solve the language decision problem for (A) the empty language, (B) the language $\mathbb{B}$, and (C) the language $\mathbb{B}^*$.

3.18 For each language, sketch an algorithm that solves the language decision problem.
(A) $\{\sigma \in \mathbb{B}^* \mid \sigma \text{ matches the regular expression } \texttt{a*ba*}\}$
(B) The language defined by this grammar

$$\begin{aligned} S &\to AB \\ A &\to aA \mid \varepsilon \\ B &\to bB \mid \varepsilon \end{aligned}$$

3.19 Solve each decision problem about Finite State machines 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^*.

3.20 For each language decision problem, give an algorithm that runs in $\mathcal{O}(1)$. Assume that the input is a bitstring and that the Turing machine starts with its I/O head pointing to the input's initial bit.
(A) The language of minimal-length binary representations of numbers that are nonzero.
(B) The binary representations of numbers that exceed 1000.

3.21 In a graph, a bridge edge is one whose removal disconnects the graph. More precisely, the graph has two vertices that are connected by at least one path before the bridge is removed but are not connected by any path after it is

removed. Consider this problem: 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?

✓ 3.22 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.

✓ 3.23 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

3.24 Each of these inputs a square matrix M with at least 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.

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. (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 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.

✓ 3.30 As stated, the Shortest Path problem, Problem 2.5, is an optimization problem. Convert it into a parametrized family of language decision problems. using the technique outlined following the Traveling Salesman problem, Problem 2.3.

✓ 3.31 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 which other jobs, find the shortest time to finish the entire car.

3.32 As stated, the Hamiltonian Circuit problem, Problem 2.2, is a decision problem. Give a function version of this problem, a search version, and an optimization version.

3.33 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.)

3.34 Give an example of a problem where the decision variant is solvable much more quickly than the search variant.

3.35 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, meaning that for any two vertices there is a path between them.

✓ 3.36 Show how to use an algorithm that solves the Shortest Path problem to solve the Vertex-to-Vertex Path problem.

3.37 Let $\mathcal{L}_F = \{ \langle n, B \rangle \in \mathbb{N}^2 \mid \text{there is an } m \in \{2, \ldots B\} \text{ that divides } n \}$ 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 polytime solution to the language recognition problem to in polytime solve the search problem that is given a number and returns a prime factor of that number.

✓ 3.38 Show that with an algorithm that in polytime solves the Subset Sum problem, Problem 2.25, we could in polytime solve the associated function problem of producing the subset.

Section V.4 *P*

When we discussed Cobham's Thesis, we were talking about the collection of languages that have polytime algorithms.

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, like Cobham's requirement of computability in polytime.[†]

4.2 Example One complexity class is the collection of languages for which there is a Turing machine decider that runs in time $\mathcal{O}(n^4)$.

4.3 Example Another is the collection of languages decided by a 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 cells 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 may study not just Turing machines but other types of machines as well, including nondeterministic Turing machines (see the next section) and Turing machines with access to an oracle for random numbers. As to the resource specification, it often involve bounds on the time or space behavior. But a class could instead be, for instance, the complement of $\mathcal{O}(n^4)$, so the specification isn't always a bound.

Definition The complexity class that we introduce now is the most important one.

4.4 Definition The class *P* consists of those language decision problems $\mathcal{L}$ where there is a deterministic Turing machine to decide membership in $\mathcal{L}$ that runs in polynomial time.

This class is very broad, containing almost every problem we can in practice solve with an algorithm. The following examples illustrate that.

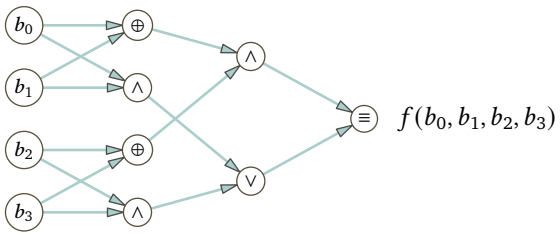
4.5 Example The language $\{\mathcal{G} \mid \text{between any two vertices there is a path}\}$ expresses the Graph Connectedness problem. It is a member of *P*. To verify this we exhibit an algorithm that decides membership in this language and that runs in polynomial time. One is to do a breadth first search of the graph, which has a runtime that is $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$.

[†] Some authors require of classes that 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 must be countable, because there are only countably many Turing machines. Here, we adopt the maximally general definition.

4.6 Example Another member of P is the problem of deciding whether two numbers are relatively prime, $\{\langle n_0, n_1\rangle \in \mathbb{N}^2 \;\big|\; \text{their greatest common divisor is } 1\}$. Euclid's algorithm fits the bill here since it has runtime $\mathcal{O}(\lg(\max(n_0, n_1)))$.

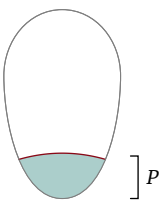
4.7 Example Still another member of P is the String Search problem of deciding substring-ness, $\{\langle \sigma, \tau\rangle \in \Sigma^* \;\big|\; \sigma \text{ is a substring of } \tau\}$. (Often in practice τ is very long and is called the haystack, while σ is short and is the needle.) The algorithm that first tests σ at the initial character of τ, then at the next character, etc., has a runtime of $\mathcal{O}(|\sigma| \cdot |\tau|)$.

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 $\equiv$ returns 1 if and only if its two inputs are the same.



This circuit returns 1 if the sum of the input bits is 0 or 1. 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 More example members of P are: (1) matrix multiplication, taken as $\{\langle M_0, M_1, M_2\rangle \;\big|\; \text{they are matrices with } M_0 \cdot M_1 = M_2\}$, (2) minimal spanning tree, $\{\langle \mathcal{G}, T\rangle \;\big|\; T \text{ is a minimal spanning tree in } \mathcal{G}\}$, and (3) edit distance, the number of character removals, insertions, or substitutions needed to transform between strings, $\{\langle \sigma_0, \sigma_1, n\rangle \;\big|\; \sigma_0 \text{ transforms to } \sigma_1 \text{ in at most } n \text{ edits}\}$.



4.10 Figure: The blob holds 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 so it is wrong to say that an algorithm or Turing machine is in P — instead, it may be that the problem that the algorithm solves 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, depending on the model.[†] So if we have a problem for which there is a polytime algorithm on another natural model then on a Turing machine model it is also in P. So no, changing the model doesn't change whether the problem is in P.

4.11 Remark A variation of Church's thesis, the Extended Church's Thesis, posits that not only are all models of mechanical computation of equal power — Church's Thesis asserts that they compute the same functions — 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 this extended thesis, a problem that falls in the class P using Turing machines also falls in that class using any other reasonable models.

However, this thesis does not enjoy anything like the support of Church's original one. Most importantly, we know of several problems 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. One is the familiar Prime Factorization problem.

As to whether the Quantum Computing model is reasonable, 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-model device in less than centuries. This is subject to scholarly reservations. For one thing, the advantage depends not only on there being a quantum device that accomplishes the task but 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 reservation is that these claim are not about general purpose computing; the problems solved are 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.

However, scientists and engineers are making great progress. In this text we will put this question aside for being as-yet unsettled, but it is worth monitoring developments closely.

[†] We take a model to be 'natural' if it was not invented in order to be a counterexample to this. [‡] One definition of 'reasonable' is "in principle physically realizable" (Bernstein and Vazirani 1997). [#] A Turing machine with access to an oracle of random bits. [§] The first published result was from 2019.

Naturalness We give the class P our attention because there are reasons to suppose that it is the best candidate to take as the collection of tractable problems. We close this section with a discussion of those reasons.

The first appeared in 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. And while their speeds 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 second reason is related: suppose that we try to restrict our interpretation of 'tractable' to, say, $\mathcal{O}(x^4)$. We may well then find ourselves doing a loop whose interior has a $\mathcal{O}(x^4)$ function, and so our new algorithm is $\mathcal{O}(x^5)$. That is, a strength of P is that it has natural closure properties.

4.12 LEMMA The class P is closed under union, intersection, and set complement.

Proof We must show that if two languages are members of P then so is their union and intersection, as well as each's complement. We will do union and leave the other two for Exercise 4.34.

Suppose that $\mathcal{L}_0 \in P$ has the characteristic function ϕ_{e_0} that runs in time $\mathcal{O}(x^{k_0})$, and similarly suppose that $\mathcal{L}_1$'s characteristic function ϕ_{e_1} runs in time $\mathcal{O}(x^{k_1})$. The natural algorithm for the union $\mathcal{L}_0 \cup \mathcal{L}_1$ is to first test the input for membership in $\mathcal{L}_0$ and if that fails then test for membership in $\mathcal{L}_1$. That algorithm is in $\mathcal{O}(x^k)$ where $k = \max(\{k_0, k_1\})$, so it runs in polytime. □

There are a number of other important language operations, including concatenation and Kleene star, under which P is closed. See the exercises.

In addition, a property we'd expect from things that we think of as easy to compute is that if two things are easy then a simple combination should be easy also. Fix two total computable functions $f, g\colon \mathbb{N} \to \mathbb{N}$ and consider this recasting of them as languages.[#]

$$\mathcal{L}_f = \{\text{str}(\,\langle n, f(n)\rangle\,) \;\big|\; n \in \mathbb{N}\} \qquad \mathcal{L}_g = \{\text{str}(\,\langle n, g(n)\rangle\,) \;\big|\; n \in \mathbb{N}\}$$

Under this formulation, P is closed under function addition, scalar multiplication, subtraction, multiplication, and composition. The class P is the smallest nontrivial one with these appealing properties.

But the main reason that P is of such great interest is Cobham's Thesis, the contention that a problem is tractable if and only if it has a solution algorithm that runs in polytime.

Some researchers have objected to this thesis, saying that there are problems whose solution algorithms cannot be improved beyond some huge exponent, such

[†] All of the models, that is, that are natural and non-quantum. [‡] This explains why we may sometime casually say that an algorithm runs in polytime, instead of the strictly speaking correct statement that the Turing machine implementing it runs in polytime. The machine model does not change whether the runtime is polytime, for natural and non-quantum models. [#] Recall that str(...) means that we represent the argument reasonably efficiently.

as $\mathcal{O}(n^{1\,000\,000})$, and these problems are not really tractable. (They point out a similar possibility of large constant multiples.) We know such problems exist because we can artificially produce some using diagonalization. However, our empirical experience over close to a century of computing has been that problems with solution algorithms of very large degree polynomial time complexity do not seem to arise 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 $\mathcal{O}(n^{1\,000\,000})$.

Moreover, in the past when a researcher has produced an algorithm for a problem with a runtime that has even a moderately large degree then, with this foot in the door, the community brings to bear an array of mathematical and algorithmic techniques that lower the runtime degree to reasonable size.

Finally, even if these objections to Cobham's Thesis are 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.[†]

In summary, researchers continue to use P as a baseline for comparisons with other complexity classes.

V.4 Exercises

✓ 4.13 True or False: if the language is finite then the language decision problem is in P.

4.14 Your coworker says something mistaken, "I've got a problem whose algorithm is in P." They are being a little sloppy with terms; how?

✓ 4.15 What is the difference between an order of growth and a complexity class?

✓ 4.16 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 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.

4.17 In class, someone asks the professor, "Why aren't all languages in P? I'll produce a Turing machine so that no matter what the input is, it outputs 1. So it accepts every member of the language, and it runs in polytime for sure." Explain the mistake.

4.18 True or false: if a problem has a logarithmic solution then it is in P.

4.19 True or false: if a language is decided by a machine then its complement is also decided by some machine.

4.20 Prove that the union of two complexity classes is also a complexity class. What about the intersection? Complement?

[†] The contention that if a problem is not in P then it is not solvable in practice has lost some of force in recent years with the rise of SAT solvers. These attack problems that are believed to not be in P and can solve instances of moderately large size, using only moderately large computing resources. See Extra C.

4.21 We have already studied the collection *RE* of languages that are computably enumerable. Recast it as a class of language decision problems. Is it a complexity class?

✓ 4.22 Show that the language of palindromes, $\{\sigma \in \mathbb{B}^* \mid \sigma = \sigma^{\mathsf{R}}\}$, is in *P*.

4.23 Prove that the problem of deciding if two natural numbers are relatively prime is in *P*.

✓ 4.24 Sketch a proof that each problem is in *P*.

(A) Given a bitstring σ, decide if it has the form $\sigma = \tau^\frown\tau^\frown\tau$ for some bitstring τ.

(B) Given a Turing machine, decide whether it halts in less than ten steps.

✓ 4.25 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 283?

(B) Sketch a proof that this problem is in *P*.

4.26 Show that each problem is in *P* by giving an algorithm. Briefly justify that the algorithm runs in polytime.

(A) $\{\sigma \in \{\mathsf{a}, \ldots \mathsf{z}\}^* \mid \sigma \text{ is in alphabetical order}\}$

(B) $\{1^k \mid k \text{ is prime}\}$

(C) Vertex-to-Vertex Path: $\{\langle \mathcal{G}, v_0, v_1\rangle \mid \text{the graph } \mathcal{G} \text{ has a path from } v_0 \text{ to } v_1\}$.

✓ 4.27 Report which of these are currently known to be in *P* and which are not. You may need to look up the fastest known algorithm. (A) Shortest Path (B) Knapsack (C) Euler Path (D) Hamiltonian Circuit

4.28 Is the empty language $\{\}$ a member of *P*?

4.29 Consider the regular languages. (A) Does the set of regular languages constitute a complexity class? (B) Is every regular language in *P*?

4.30 Is *P* countable or uncountable?

4.31 If both $\mathcal{L}_0$ and $\mathcal{L}_1$ are in *P*, and $\mathcal{L}_0 \subseteq \mathcal{L} \subseteq \mathcal{L}_1$, then must $\mathcal{L}$ be in *P*?

✓ 4.32 Is the Halting problem in *P*?

✓ 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 Finish the proof of Lemma 4.12. (A) Prove that *P* is closed under intersection. (B) Prove that *P* is closed under complement.

4.35 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 set of all reversals of strings in the starting language.

4.36 Show that *P* is closed under the concatenation of two languages.

✓ 4.37 Following up on Lemma 4.12, prove that *P* is closed under the union of finitely many languages. That is, prove that if finitely many languages are in *P* then so is their union. Infinitely many?

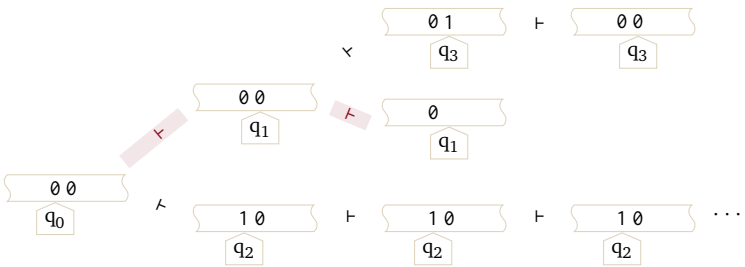
4.38 The complexity class *co-P* contains all the languages whose complement is in *P*. Show that $P = coP$.

Section V.5 *NP*

Recall that a Finite State machine is nondeterministic if from a present configuration and input there may be more than one next state, 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.

$$\mathcal{P} = \{\, q_0 0 1 q_2,\ q_0 0 \mathtt{R} q_1,\ q_1 0 \mathtt{B} q_1,\ q_1 0 1 q_3,\ q_2 1 1 q_2,\ q_3 1 0 q_3 \,\}$$

For such a machine the computational history can be more than a line, it can be a tree. Below is part of that tree for machine $\mathcal{P}$ and input 00. It is directed, meaning that edges point from left to right. The picture shows three maximal paths. The middle one is highlighted. The top and middle one are finite, while the bottom one has infinitely many nodes.



Nondeterministic Turing machines We modify the definition of a Turing machine by omitting the restriction that $\mathcal{P}$ must be deterministic.[†]

5.1 Definition A nondeterministic Turing machine $\mathcal{P}$ is a finite set of instructions $q_p T_p T_n q_n \in Q \times \Sigma \times (\Sigma \cup \{\mathtt{L}, \mathtt{R}\}) \times Q$. The set Q of states is finite. Some of them, $F \subseteq Q$, are accepting states. The tape alphabet set Σ is finite, contains at least two members, including blank, and does not contain the characters L or R.

As with deterministic machines, the association between the present state and input character, and what the machine does next is given by the machine's transition function, $\Delta\colon Q \times \Sigma \to \mathscr{P}((\Sigma \cup \{\mathtt{L}, \mathtt{R}\}) \times Q)$. The difference here is that the transition function outputs a set of pairs ⟨action, next state⟩ instead of just one.

After the deterministic Turing machine definition we gave a full description of how they act as a sequence of '⊢' steps. Exercise 5.40 asks for the same for these machines. As illustrated above, the key difference is that here the computation is not a single line, it is a tree. The nodes are configurations $\mathcal{C}$ and the root node is the initial configuration $\mathcal{C}_0$. This tree is a directed graph, where $\mathcal{C} \vdash \hat{\mathcal{C}}$ means that the second one immediately succeeds the first. A maximal path has the form $\mathcal{C}_0 \vdash \cdots \mathcal{C}_h$, where $\mathcal{C}_h$ has no immediate successors, or is of the form $\mathcal{C}_0 \vdash \mathcal{C}_1 \vdash \cdots$, with infinitely many vertices.

[†] Recall that determinism is the requirement that different four-tuples instructions cannot begin with the same $q_p T_p$.

Having a computation tree instead of a sequence adds a few wrinkles. It might be that some maximal paths terminate while others do not — of course, a deterministic machine either halts or it doesn't — and some might terminate with 1 on the tape while others end with 11. The most natural approach here is not to work with functions computed by these machines but instead to use them as language deciders.

From the Finite State machine discussion we inherit two mental models for how nondeterministic machines act. One is that the machine is unboundedly parallel and simultaneously computes all of the paths. The other is that the machine guesses which maximal path to follow, or is told by some demon, and then deterministically verifies that path. In both models, the machine accepts an input string if there exists at least one maximal path ending in a configuration with an accepting state, and otherwise does not accept. The "otherwise" seemingly could mean that in a non-accepting computation the tree contains some sequences of configurations that don't terminate. However, to use these machines as language deciders we shall want to time them, including how long they take to not accept. So we shall require of language deciders that their trees have no infinite paths.

5.2 LEMMA If a computation tree has no infinite sequence of configurations then it contains only finitely many nodes.

Proof In a computation tree every vertex has only finitely many immediate successors because the number of machine instructions is finite. König's Lemma on page 160 says that for such a graph to have infinitely many vertices, it must have an infinite path. □

5.3 DEFINITION A nondeterministic Turing machine $\mathcal{P}$ accepts an input string if its computation tree has at least one maximal path that ends with a configuration having an accepting state, otherwise it rejects that input. Let $\mathcal{P}$ be such that for any input $\sigma \in \Sigma^*$, the computation tree has no infinite path. Then it decides the language $\mathcal{L}$ when 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 a 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. Pushdown machines are the opposite since there are languages that a nondeterministic Pushdown machine can decide but that no deterministic one can. What about Turing machines?

5.4 THEOREM Deterministic Turing machines and nondeterministic Turing machines decide the same set of languages.

Proof One direction is easy. A deterministic Turing machine is a special case of a nondeterministic one. So if a language is decided by a deterministic machine then it is decided by a nondeterministic one also.

For the converse let the nondeterministic Turing machine $\mathcal{P}$ decide the lan-

[†] Recall how we have been handling having deterministic Turing machines act as language deciders. In line with our approach that these machines compute functions, we have taken it to mean that for any string input, the machine halts and signals 'Accept' or 'Reject' by having 1 or 0 on the tape.

guage $\mathcal{L}$. Fix an input σ. Consider a deterministic machine $\mathcal{Q}$ that does a breadth-first search of $\mathcal{P}$'s computation tree. If $\mathcal{P}$ accepts σ then there is a terminal configuration in the tree where the state is accepting and $\mathcal{Q}$ will find it. If $\mathcal{P}$ does not accept σ then every terminal configuration has a rejecting state. In a nondeterministic decider the computation tree has only finitely many vertices so the search will eventually exhaust all of them, and then $\mathcal{Q}$ rejects σ. □

That proof is, basically, time-slicing. The computers that we have on our desks simulate an unboundedly-parallel machine by having the CPU cycle among processes. For example, if we are editing a program and also showing a video then the computer may update the editor for a time, the update the video, then switch back to the editor, etc. This is a kind of dovetailing, a way of doing a breadth-first traversal of the computation tree.

Speed The excitement about nondeterministic Turing machines is that they might be much faster than deterministic ones.

5.5 Example The Satisfiability problem inputs a Propositional Logic expression and determines whether there are truth values for the variables that make the expression evaluate to *T*.

Is this propositional logic expression satisfiable?

$$E = (P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (\neg P \vee \neg Q \vee \neg R) \wedge (Q \vee R) \qquad (*)$$

The natural approach is to compute a truth table. The one below shows that E is satisfiable because the TTF row ends in *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

The number of table rows grows exponentially: it is 2 raised to the number of input variables. Going through the rows sequentially is very slow.

Each row is easy; it is just that there are lots of rows. This is suited to an unboundedly parallel machine. For each line we could fork a child process that would finish in polytime If at the end any child found a T then the expression is satisfiable. Thus, while a serial machine appears to require exponential time, a nondeterministic machine does the job in polytime.

5.6 Example The Traveling Salesman problem is similar. Trying every graph circuit in turn would be slow. But a nondeterministic machine could do all of the circuits in parallel, or just guess the best circuit — or get it from some oracular demon — and then verify it. That would be fast.

So while adding nondeterminism to Turing machines doesn't compute anything new, we can reasonably conjecture that it computes those things faster.

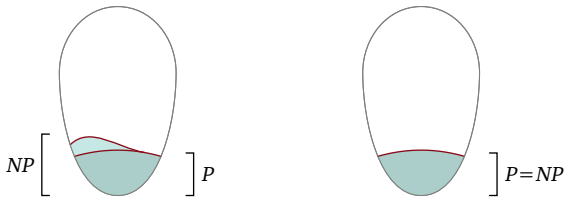
Definition Next we define the most important class of language decision problems associated with nondeterministic Turing machines.

5.7 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 maximal paths of $\mathcal{P}$'s computation terminate in at most $p(|\sigma|)$ steps.

The following is immediate because a deterministic Turing machine is a special case of a nondeterministic one.

5.8 LEMMA $P \subseteq NP$

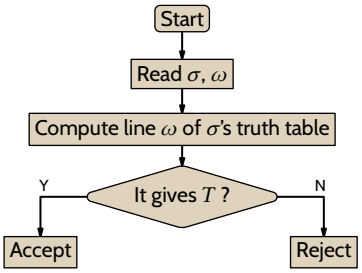
Very important: no one knows whether P is a strict subset. That is, no one knows whether $P \neq NP$ or $P = NP$. We will say much more in the rest of this chapter.



5.9 FIGURE: The two possibilities: $P \subset NP$ and $P = NP$

A pattern in mathematical presentations is to have a definition that is conceptually clear, followed by a result that makes the definition easier to apply. We now give that second one. This is where we use the mental model of the machine guessing, or of being given a hint.

For motivation consider again Satisfiability. Imagine that in Example 5.5 above the demon whispers, "Psst! Try TTF." We can quickly verify that is correct using a deterministic machine.



(**)

It inputs $\sigma = E$ from (*) and also the hint $\omega =$ TTF. It reaches 'Accept' in polytime.

5.10 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 the witness or certificate.

5.11 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 at least one witness ω where $\mathcal{V}$ accepts $\langle \sigma, \omega \rangle$.

Before the proof we will comment on some aspects of both the definition and lemma, and give a few examples.

First we unwind the wording. Our touchstone is the Satisfiability problem. Using the lemma to show that it is in *NP* requires that we produce a deterministic Turing machine verifier. That's given in the flowchart (∗∗) above. Its first input, σ, is the candidate Boolean expression while its second, ω, represents a line of σ's truth table. If σ is satisfiable then there is a suitable witness, a suitable line from the truth table, that $\mathcal{V}$ can check gives a result of *T*. For the expression in Example 5.5's (∗) from above, $\omega = TTF$. (Clearly the verifier can do this checking in polytime.) On the other hand, if a candidate σ is not satisfiable, for example if $\sigma = P \wedge \neg P$, then no ω will cause $\mathcal{V}$ to accept.

Thus, the lemma elucidates something about *NP*: while *P* contains problems where we can deterministically find the answer in polytime, *NP* contains the problems where we can at least deterministically verify answers in polytime.

The second comment is to note the most striking thing about the definition, that it says there exists a witness ω but it does not say where it comes from. A person with a computational mindset may well ask, "but how will we find the ω's?" The point is not how to compute them. The point is whether there exists a deterministic Turing machine $\mathcal{V}$ that can leverage a hint or guess ω, when one is given, to verify in polytime that $\sigma \in \mathcal{L}$. The verifier $\mathcal{V}$ doesn't find the ω's, it just uses them.

Third, 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 fourth comment relates to this asymmetry. What's the point of the 'verify in polytime' requirement: if we are given a correct answer then isn't it just trivial to verify? Imagine for contrast 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. Vrifying seems to require exponential time. That would make the demon's papers, in a sense, useless. So we require that the verification be tractable.

Another aspect of the asymmetry is that it does not seem that $\mathcal{L} \in NP$ implies that its complement $\mathcal{L}^{\mathsf{c}}$ 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 class *coNP* contains the complements of languages from *NP*, we suspect that

[†] With this in mind, perhaps a better term for ω is "potential witness" or "proposed witness." But those are not standard.

NP ≠ *coNP*.

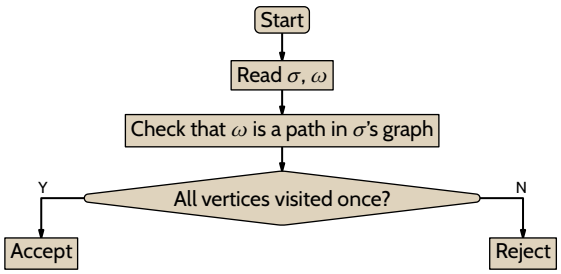
Finally, note that the lemma requires that the verifier runs in time polynomial in $|\sigma|$, not polynomial in the length of its input, $\langle\sigma,\omega\rangle$. If it said the latter then we could check the chess strategy just by using a witness that is exponentially long, which would make $\langle\sigma,\omega\rangle$ exponentially long. Also observe that assuming that $\mathcal{V}$ runs in time polynomial in $|\sigma|$ implies that 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.

5.12 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.

$$\mathcal{L}=\{\,\langle\mathcal{G},v,\hat{v}\rangle \;\big|\; \text{some path in } \mathcal{G} \text{ between } v \text{ and } \hat{v} \text{ visits every vertex exactly once}\,\}$$

To show that this problem is in the class *NP* we produce a verifier $\mathcal{V}$. It is sketched below. In its input $\langle\sigma,\omega\rangle$, the first item is the candidate for membership in the language, $\sigma=\langle\mathcal{G},v,\hat{v}\rangle$. The verifier interprets the second item to be a path, $\omega=\langle v,v_1,\ldots\,\hat{v}\rangle$.

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.



5.13 Example The Composite problem asks whether a given number has nontrivial factors.

$$\mathcal{L}=\{\,n\in\mathbb{N}^+ \;\big|\; n \text{ has a divisor } d \text{ with } 1<d<n\,\}$$

To show that $\mathcal{L}\in NP$ we construct a verifier $\mathcal{V}$. 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 d. Then $\mathcal{V}$ checks that $1<d<n$ and d 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. Clearly we can write this verifier to run in polytime. Therefore $\mathcal{L}\in NP$.

We are ready now for the promised proof of Lemma 5.11.

Proof Suppose first that the language $\mathcal{L}$ is decided by a nondeterministic Turing machine $\mathcal{P}$ in polytime. We will construct a deterministic verifier $\mathcal{V}$ that runs in

polytime. Let $p\colon \mathbb{N} \to \mathbb{N}$ be the polynomial such that for any input $\sigma \in \Sigma^*$, all maximal paths of $\mathcal{P}$'s computation tree terminate in at most $p(|\sigma|)$ steps. If $\sigma \in \mathcal{L}$ then we can use an accepting maximal path to create a witness to σ's acceptance, for instance as in the sequence $\omega = \langle 3, 2 \ldots \rangle$ meaning, "in the computation tree, at the first node take the third child, then take the second child of that, etc." More precisely, $\mathcal{P}$ has a finite number of states k so we can represent the accepting branch 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 and only if some maximal path that accepts the string. 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 $\hat{\mathcal{V}}$ accepting $\langle \tau, \hat{\omega} \rangle$, so there is a way for the prior paragraph to result in acceptance of τ, and so $\hat{\mathcal{P}}$ accepts τ. Conversely, suppose that $\tau \notin \mathcal{L}$. By the definition of a verifier, no witness $\hat{\omega}$ will result in $\hat{\mathcal{V}}$ accepting $\langle \tau, \hat{\omega} \rangle$, and thus $\hat{\mathcal{P}}$ rejects τ. □

5.14 Remark A common reaction to the second half of that proof is, "Wait, $\hat{\mathcal{P}}$ pulls $\hat{\omega}$ out of the air? How is that legal?" This reaction — about whether this is unrealistic — is both common and reasonable so we will respond.

The first response is purely formal. Definition 5.10, 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 '$\hat{\mathcal{P}}$ 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 natively.) When Turing formulated his definition there were no physical computers but they were coming; will we someday have nondeterministic devices? We don't know of any candidates. (There have been proposals involving such things as time travel through wormholes but we

put them aside as speculative, albeit very interesting.)[†] But that doesn't mean that thinking about nondeterministic Turing machines is a purely academic exercise.[‡]

Nondeterministic machines are practical in that the problems that are associated with these machines, the members of *NP*, are eminently practical. Computer scientists have been trying to find fast solutions to many of them since computers have existed. More evidence is that Lemma 5.11 rephrases questions about nondeterministic machines as questions about deterministic ones, the verifiers.

5.15 COROLLARY Every problem in *NP* can be solved in exponential time.

Proof Fix $\mathcal{L} \in NP$. We have seen that 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 $\sigma \in \mathcal{L}$ if and only if $\mathcal{V}$ accepts $\langle \sigma, \omega \rangle$ for at least one witness $\omega \in \Sigma^*$. We have also noted that we can take the length of ω to be bounded by $p(|\sigma|)$.

So, try the computation of $\mathcal{V}$ on input $\langle \sigma, \omega \rangle$ for all ω's of length at most $p(|\sigma|)$. Then $\sigma \in \mathcal{L}$ if and only if at least one of these inputs results in acceptance. There are at most $b^{p(|\sigma|)}$ many such ω's, where b is the number of characters in the alphabet. □

We close by making a connection with the computably enumerable sets. 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 known in theory, polytime verification seems to be the limit of what can be done feasibly.

V.5 Exercises

✓ 5.16 Someone asks, "In Lemma 5.11, since the witness ω is not required to be effectively computable, why can't I just take it to be exactly the problem solution, the bit 1 if $\sigma \in \mathcal{L}$, and 0 if not? Then the verifier can just ignore σ and follow the bit." They are confused. Straighten them out.

5.17 In the context of all branches being finite, 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.

✓ 5.18 Is it satisfiable? (A) $(P \wedge Q) \vee (\neg Q \wedge R)$ (B) $(P \rightarrow Q) \wedge \neg((P \wedge Q) \vee \neg P)$

5.19 True or false? If a language is in *P* then it is in *NP*.

5.20 Uh-oh. You find yourself with a nondeterministic Turing machine where on input σ, one maximal path of the computation tree accepts and one rejects. Some don't halt at all. What is the upshot; what language does it decide?

[†] A caution about quantum computers: well-established physical theory says that subatomic particles can be in a superposition of many states at once. Some popularizations wrongly suggest that quantum computers can therefore try all potential solutions in parallel. But, that we know of, this is false. We believe that we cannot get information on individual parts out of a quantum system and instead must use interference. [‡] Not that there is anything wrong with academic exercises.

✓ 5.21 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.

5.22 Sketch a nondeterministic algorithm to search an array of numbers for the number k. Describe it both in terms of unbounded parallelism and in terms of guessing.

5.23 The Linear Programming problem is described on page 285. The related problem Integer Linear Programming also seeks to maximize a linear objective function $F(x_0, \ldots x_n) = d_0x_0 + \cdots + d_nx_n$, subject to linear constraints $a_{i,0}x_0 + \cdots + a_{i,n}x_n \leq b_i$ (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 parametrized family of language decision problems. Sketch a nondeterministic algorithm, giving both an unbounded parallelism formulation and a guessing formulation.

✓ 5.24 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 $e_0, e_1 > 0$. 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.

5.25 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, $\sum_{a\in\hat{A}} a = \sum_{a\notin\hat{A}} a$.

✓ 5.26 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.

✓ 5.27 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 5.11 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.

✓ 5.28 In the British game show *Countdown*, players are given a multiset of six numbers from $S = \{1, 2, \ldots 10, 25, 50, 75, 100\}$, along with a target integer T from the set $I = \{100. \ldots 999\}$. (Recall that a multiset is like a set but repeats don't collapse.) They must make an arithmetic expression that evaluates to the target. They can use addition, subtraction, multiplication, and division without remainder. Their expression can fail to use some of the given numbers, but cannot use one more than the number of times that it appears in the multiset. Show that the Countdown problem $\mathcal{L}_{\text{CD}} = \{\langle s_0, \ldots s_5, T \rangle \in S^6 \times I \mid \text{a combination of the } s_i \text{ gives } T\}$ is a member of *NP*.

✓ 5.29 Recall that we recast the Traveling Salesman optimization problem as a language decision problem for a parametrized family of languages. Show that each such language is in *NP* by outlining a verifier.

5.30 The Independent Sets problem 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 by an edge. State it as a language decision problem, and use Lemma 5.11 to show that this problem is in *NP*.

✓ 5.31 Use Lemma 5.11 to show that the Knapsack problem is in *NP*.

5.32 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*.

✓ 5.33 The Longest Path problem inputs a graph and a bound, $\langle \mathcal{G}, B \rangle$, and determines 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). Express it as a language decision problem and show that it is in *NP*.

5.34 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 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.

5.35 Is *NP* countable or uncountable?

✓ 5.36 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, which may

be a proper subset of the map's vertices. They must decide if there are two cycles such that every member of V is on at least one of the two cycles, and each cycle has total weight at most $B \in \mathbb{N}$.

5.37 The definition of when a nondeterministic machine decides a language, Definition 5.3, requires that every branch in the computation tree is finite. For recognition of languages we drop that condition. A nondeterministic Turing machine recognizes a language $\mathcal{L}$ when if $\sigma \in \mathcal{L}$ then there is at least one maximal path in the computation tree that accepts σ, while if $\sigma \notin \mathcal{L}$ then no maximal path in the computation tree accepts (some may fail to accept because they are infinite). Show that the nondeterministic Turing machines recognize the same set of languages as deterministic Turing machines.

✓ 5.38 Two graphs $\mathcal{G}_0, \mathcal{G}_1$ are isomorphic if there is a one-to-one and onto function $f : \mathcal{N}_0 \to \mathcal{N}_1$ such that $\{v, \hat{v}\}$ is an edge of $\mathcal{G}_0$ if and only if $\{f(v), f(\hat{v})\}$ is an edge of $\mathcal{G}_1$. 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*.

5.39 (A) Show that the Halting problem in 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.

5.40 Following the definition of Turing machine, on page 8, we gave a formal description of how those machines act. We did the same for Finite State machines on page 184, and for nondeterministic Finite State machines on page 193. Give a formal description of the action of a nondeterministic Turing machine.

Section V.6 Reductions between problems

When we studied uncomputability we considered a sense in which some problems are harder than others. Recall that the Halts on Three problem is to decide membership in the set $S = \{ e \in \mathbb{N} \mid \phi_e(3)\downarrow \}$. We showed that if we could solve it then we could solve the Halting problem. We denoted this relationship between problems 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 $\phi_e^A = \mathbb{1}_B$.[†] In words, we can answer questions about membership in B by being given access to questions about membership in A.

There are reductions other than Turing reductions. Consider these two problems, where the second is a translation or alternate presentation of the other: $S =$

[†] As discussed on page 94, people often get 'reduces to' the wrong way around. Think of 'B reduces to A' informally as meaning that A knows at least as much as B. For instance, where A is the *Entscheidungsproblem* of answering all questions in Mathematics and B is Goldbach's conjecture then B reduces to A because if we could solve A then we would get a solution to B as a side effect.

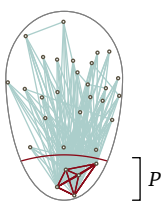
$\{e \mid \phi_e(3)\downarrow\}$ and $T = \{e^2 + 1 \mid \phi_e(3)\downarrow\}$. We can compute answers to questions about T from answers about S since x is in T iff $\sqrt{x-1}$ is a natural number and is in S. We say that B is mapping reducible or many-one reducible to A, denoted $B \leq_m A$, if there is a total computable translation function f such that $x \in B$ if and only if $f(x) \in A$. Thus, $T \leq_m S$.

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 possibly doing more processing with that information. But with $B \leq_m A$ we make only one oracle call, asking whether $f(x) \in A$, and then return that call's result.

This chapter focuses on efficient use of computing resources so one natural thing to do is to define 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 polytime. However, more common is to adapt many-one reduction by restricting the translation function to be computable and 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 $f : \mathbb{B}^* \to \mathbb{B}^*$ 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, $\mathcal{L} \in \mathscr{P}(\mathbb{B}^*)$, 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.

The intuition is that there is a reduction of this kind when one problem is a translation of the other, or a special case.

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. The Vertex-to-Vertex Path problem inputs an unweighted graph and two vertices, and then decides if there is a path between the two. We

[†] If between two problems there is a Karp reduction then there is a Cook reduction also. (The converse does not hold since under Cook reduction a problem and its complement are equivalent but that's not true 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 seem less natural, it is the right one for us to study. In any event, Karp reduction is the standard in computational complexity.

will show that the Vertex-to-Vertex Path problem reduces to Shortest Path, that Vertex-to-Vertex Path $\leq_p$ Shortest Path, by viewing the first as a special case of the second.

First, express them as language decision problems.

$$\mathcal{L}_0 = \{\, \langle \mathcal{G}, v_0, v_1, B\rangle \;\big|\; \text{there is path from } v_0 \text{ to } v_1 \text{ of total weight at most } B \,\}$$
$$\mathcal{L}_1 = \{\, \langle \mathcal{H}, w_0, w_1\rangle \;\big|\; \text{there is path between } w_0 \text{ and } w_1 \,\}$$

We will show that $\mathcal{L}_1 \leq_p \mathcal{L}_0$ by describing a reduction function f. Given $\langle \mathcal{H}, w_0, w_1\rangle$ as input, make a weighted graph $\mathcal{G}$ with the same vertices as $\mathcal{H}$ by giving each edge a weight of 1. Also take the bound to be the number of vertices, $B = |\mathcal{G}|$. 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 runs in polytime.

In the next example, at first glance it doesn't seem that one problem is a translation of the others — for one thing, one is a graph problem and the other is not — so we include some development suggesting how to see a relationship.[†]

6.4 Example The Clique problem is the decision problem for the language $\mathcal{L}_B = \{\, \langle \mathcal{G}, B\rangle \;\big|\; \mathcal{G} \text{ has a clique with } B \text{ vertices} \,\}$. We will sketch that Satisfiability $\leq_p$ Clique, that is, Clique is at least as hard as Satisfiability.

Consider how to satisfy this Boolean expression.

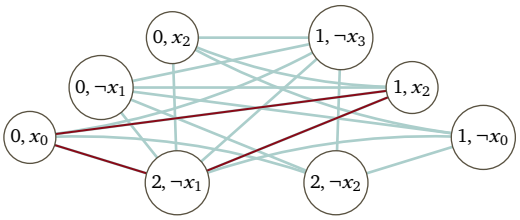
$$E = (x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2)$$

The $\wedge$'s make the statement as a whole T if and only if all of its clauses are T. Inside each clause, the $\vee$'s mean that the clause is T if and only if at least one 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.

Thus, 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 are compatible: if they come from different clauses, $c_0 \neq c_1$, and the literals are not negations of each other, $\ell_0 \neq \neg\ell_1$.

[†] In looking at materials, for example on the web, be aware that it is common to expect readers to work out for themselves the motivation for the reduction function's definition.

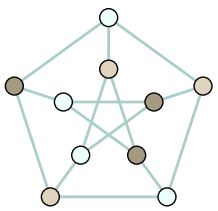


6.5 Animation: Compatibility graph associated with E.

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 as in the prior paragraph and where B is the number of clauses in E. Then $E \in \mathsf{SAT}$ if and only if $f(E) \in \mathcal{L}_B$. Clearly this function can be computed in polytime.

6.6 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.



6.7 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 a outputs a propositional logic expression $E = f(\mathcal{G})$ such that the graph is 3-colorable if and only if the expression is satisfiable.

Suppose that the input graph $\mathcal{G}$ has n-many vertices $\{v_0, \ldots v_{n-1}\}$. Then E has $3n$-many Boolean variables, $a_0, \ldots a_{n-1}$, and $b_0, \ldots b_{n-1}$, and $c_0, \ldots c_{n-1}$. The idea is that if the i-th vertex v_i gets the first color then E will be satisfied when the first variable is true but the others are false, $a_i = T, b_i = F, c_i = F$, 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 E includes two kinds of clauses. For every vertex v_i,

there is a clause saying that the vertex gets at least one color.

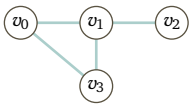
$$(a_i \vee b_i \vee c_i)$$

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.

$$(\neg a_i \vee \neg a_j) \qquad (\neg b_i \vee \neg b_j) \qquad (\neg c_i \vee \neg c_j)$$

Then 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.



$$\begin{aligned}(a_0 \vee b_0 \vee c_0) \wedge (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge (a_3 \vee b_3 \vee c_3)\\ \wedge (\neg a_0 \vee \neg a_1) \wedge (\neg b_0 \vee \neg b_1) \wedge (\neg c_0 \vee \neg c_1)\\ \wedge (\neg a_0 \vee \neg a_3) \wedge (\neg b_0 \vee \neg b_3) \wedge (\neg c_0 \vee \neg c_3)\\ \wedge (\neg a_1 \vee \neg a_2) \wedge (\neg b_1 \vee \neg b_2) \wedge (\neg c_1 \vee \neg c_2)\\ \wedge (\neg a_1 \vee \neg a_3) \wedge (\neg b_1 \vee \neg b_3) \wedge (\neg c_1 \vee \neg c_3)\end{aligned}$$

By construction, we can satisfy the expression with some assignments of truth values for the variables 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.

6.8 Example We will show that Subset Sum $\leq_p$ Knapsack. The Subset Sum problem inputs a multiset $S = \{s_0, \ldots s_{k-1}\} \subset \mathbb{N}$ and a target $T \in \mathbb{N}$. It asks for a sub-multiset whose elements add to the target.

$$\mathcal{L}_1 = \{\langle S, T\rangle \;\big|\; \text{some } R \subseteq S \text{ has } T = \sum_{r\in R} r\}$$

The Knapsack problem starts with a multiset $U = \{u_0, \ldots u_{n-1}\}$ whose elements 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 sub-multiset $A \subseteq U$ whose weight total does not exceed W and whose value total is at least V.

$$\mathcal{L}_0 = \{\langle U, w, v, W, V\rangle \;\big|\; \text{some } A \subseteq U \text{ has } \sum_{a\in A} w(a) \leq W \text{ and } \sum_{a\in A} v(a) \geq V\}$$

The 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 run in 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 $W = V = 126$. Then $\langle S, T\rangle \in \mathcal{L}_1$ iff $\langle S, w, v, W, V\rangle \in \mathcal{L}_0$.

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 $f(\langle S, T\rangle) \in \mathcal{L}_1$. In this way, we think of Subset Sum as the specialization of, or translation of, Knapsack that is obtained by setting $w(s_i) = v(s_i) = s_i$ and $W = V = T$. Clearly f can be done in polytime.

We close with some basic facts about polytime reduction.

6.9 LEMMA (1) Polytime reduction is reflexive: $\mathcal{L} \leq_p \mathcal{L}$ for all languages. (2) It is also transitive: $\mathcal{L}_2 \leq_p \mathcal{L}_1$ and $\mathcal{L}_1 \leq_p \mathcal{L}_0$ imply that $\mathcal{L}_2 \leq_p \mathcal{L}_0$. (3) If $\mathcal{L}_1 \in P$ then every language $\mathcal{L}_0$ that is nontrivial, where $\mathcal{L}_0 \neq \emptyset$ and $\mathcal{L}_0 \neq \Sigma^*$, satisfies that $\mathcal{L}_1 \leq_p \mathcal{L}_0$. (4) The class P is closed downward: if $\mathcal{L}_0 \in P$ and $\mathcal{L}_1 \leq_p \mathcal{L}_0$ then $\mathcal{L}_1 \in P$. (5) So also is the class *NP*.

Proof The first two items and the final one are Exercise 6.35.

For the third, fix a $\mathcal{L}_1 \in P$. Let $\mathcal{L}_0$ be nontrivial so that it has a member $\sigma \in \mathcal{L}_0$ and a nonmember $\tau \notin \mathcal{L}_0$. We will give a polytime reduction function f showing that $\mathcal{L}_1 \leq_p \mathcal{L}_0$. Where $\alpha \in \Sigma^*$, computing whether $\alpha \in \mathcal{L}_1$ can be done in polytime. If it is a member of $\mathcal{L}_1$ then let $f(\alpha) = \sigma$ while if not then let $f(\alpha) = \tau$.

As to the fourth, suppose that there is a polytime algorithm to decide membership in $\mathcal{L}_0$ and that $\mathcal{L}_1 \leq_p \mathcal{L}_0$ via a polytime reduction function g. Then decide 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. □

By the lemma, because Example 6.6 shows Graph Colorability $\leq_p$ Satisfiability and Example 6.4 gives Satisfiability $\leq_p$ Clique, we know that Graph Colorability $\leq_p$ Clique.

We close by reiterating this section's main point: the reducibility $\mathcal{L}_1 \leq_p \mathcal{L}_0$ means that if we could solve the $\mathcal{L}_0$ problem in polynomial time then we could also solve the $\mathcal{L}_1$ problem in polynomial time. (Finally, more examples of problem reductions are in Section 7.)

V.6 Exercises

6.10 Which is the right way to use the phrase 'reduces to' when reading $\mathcal{L}_1 \leq_p \mathcal{L}_0$ aloud, "$\mathcal{L}_1$ reduces to $\mathcal{L}_0$" or "$\mathcal{L}_0$ reduces to $\mathcal{L}_1$?"

✓ 6.11 Your found this online as the intuition behind polytime reduction, $B \leq_p A$: *If you had a black box that can solve instances of problem A, then you could solve any instance of B using polynomial number of steps plus a polynomial number of calls to the black box that solves A.* Is it right?

6.12 The relation $\mathcal{L}_1 \leq_p \mathcal{L}_0$ implies which?

(A) A fast algorithm for $\mathcal{L}_0$ would give a fast algorithm for $\mathcal{L}_1$.

(B) A fast algorithm for $\mathcal{L}_1$ would give a fast one for $\mathcal{L}_0$.

✓ 6.13 Show that if $\mathcal{L}_1 \notin P$ and $\mathcal{L}_1 \leq_p \mathcal{L}_0$ then $\mathcal{L}_0 \notin P$ also. What about *NP*?

6.14 Your friend is confused. "Lemma 6.9 says that every language in P is $\leq_p$ every other nontrivial language. But there are only countably many f's because they each come from some Turing machine, while there are uncountably many languages. So I'm not seeing how there are enough reduction functions for a given language to be related to all others." Un-confuse them.

6.15 Example 6.8 includes as illustration a Subset Sum problem, where $S = \{18, 23, 31, 33, 72, 86, 94\}$ and $T = 126$. Solve it.

6.16 Fix a language decision problem $\mathcal{L}_0$ in $\mathcal{O}(n^3)$, and fix $\mathcal{L}_1 \in \mathcal{O}(n^2)$, and $\mathcal{L}_2 \in \mathcal{O}(2^n)$ but not in P, and $\mathcal{L}_3 \in \mathcal{O}(\lg n)$. In the array entry i, j below, put 'N' if $\mathcal{L}_i \leq_p \mathcal{L}_j$ is not possible and 'Y' if it is.

	$\mathcal{L}_0$	$\mathcal{L}_1$	$\mathcal{L}_2$	$\mathcal{L}_3$
$\mathcal{L}_0$	(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)

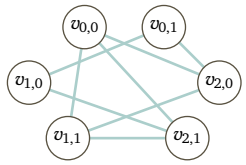
✓ 6.17 Suppose that the language A is polynomial time reducible to the language B, $A \leq_p B$. Which of these are true?

(A) If A is tractably decidable then B is tractably decidable also.

(B) If A is not tractably decidable then B is also not tractably decidable.

6.18 Produce the compatibility graph as in Example 6.4 for $(x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_1) \wedge (x_0 \vee \neg x_1)$.

6.19 Following the method of Example 6.6 give the expression associated with the question of whether this graph is 3-colorable. Is that expression satisfiable?



✓ 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. (If $\alpha = a_0a_1 \ldots a_{n-1}$ and $\beta = b_0b_1 \ldots b_{n-1}$ are length n strings, then β is a cyclic shift of α when there is an index $k \in \{0, \ldots n-1\}$ such that $a_i = b_{(k+i) \bmod n}$ for all $i < n$. Thus, where $\alpha =$ abcde, the cyclic shifts are bcdea, cdeab, ...)

(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) State Cyclic Shift 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 $\alpha^\frown\alpha$.

✓ 6.21 The Shortest Path problem, Problem 2.5, inputs a weighted graph and two vertices and finds a least-weight path between them (or that no such path exists).

(A) The Unweighted Shortest Path problem takes as input an unweighted graph and two of its vertices, and finds a path between the two with the fewest number of edges (or that no such path exists). Express it as language decision problem. Prove that it reduces to the Shortest Path problem. (How does it differ from the Vertex-to-Vertex Path problem?)

(B) The Longest Path problem takes as input a weighted graph and two vertices and finds a greatest-weight path containing no repeated edges (or that no such path exists). Express it as language decision problem. Prove that this problem reduces to the one in the first item.

6.22 Show that Hamiltonian Circuit $\leq_p$ Traveling Salesman. (A) State each as a language decision problem. (B) Produce the reduction function. (C) Argue that it runs in polytime.

6.23 The 3-SAT problem is to decide the satisfiability of Conjunctive Normal Form propositional logic expressions where every clause has at most three literals (see Appendix C). The Strict 3-Satisfiability problem requires that each clause has exactly three unequal literals. We will show that the two problems are inter-reducible.

(A) Express each as a language decision problem.

(B) Show the easy half, that Strict 3-Satisfiability $\leq_p$ 3-SAT.

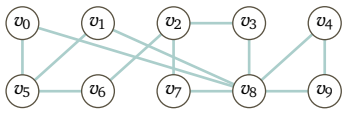
(C) Also show that we can go from clauses with two literals to clauses with three by introducing an irrelevant variable: $P \vee Q$ is equivalent to $(P \vee Q \vee R) \wedge (P \vee Q \vee \neg R)$. Along the same lines, show that P is equivalent to $(P \vee Q \vee R) \wedge (P \vee \neg Q \vee R) \wedge (P \vee Q \vee \neg R) \wedge (P \vee \neg Q \vee \neg R)$.

(D) Show 3-SAT $\leq_p$ Strict 3-Satisfiability.

✓ 6.24 Consider graphs without loops. 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.

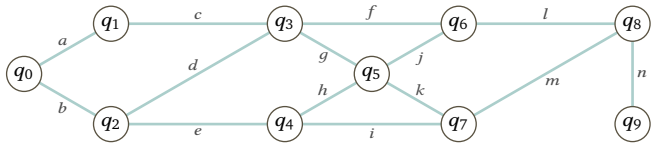
(D) Show that in a graph, S is an independent set if and only if $\mathcal{N} - S$ is a vertex cover, where $\mathcal{N}$ is the set of vertices.

(E) Conclude that Independent Set $\leq_p$ Vertex Cover and that Vertex Cover $\leq_p$ Independent Set.

6.25 The Vertex Cover problem inputs a graph and a bound and decides if there is a vertex set of size at most 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 $S_0 \subseteq S, \ldots S_n \subseteq S$, 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.

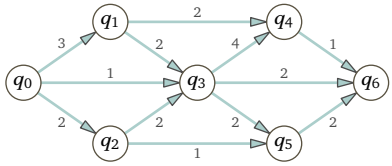
(B) Find a vertex cover of size six for this graph.



(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.

6.26 In this network, each edge is labeled with a capacity.



The Max-Flow problem is to find the maximum total amount that can flow across the network, typically 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, described on page 285, is to optimize a linear function $F(x_0, \ldots x_n) = c_0x_0 + \cdots + c_nx_n$ subject to linear constraints, ones of the form $a_{i,0}x_0 + \cdots + a_{i,n}x_n \leq b_i$ or $a_{i,0}x_0 + \cdots + a_{i,n}x_n \geq b_i$.

(A) Express each as a language decision problem.

(B) For the above network, find the maximum flow by eye.

(C) For each edge v_iv_j in that network, 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.

6.27 The Max-Flow problem inputs a directed graph where each edge is labeled with a capacity, and the task is to find a the maximum amount that can flow from the source node to the sink node (a diagram is in Exercise 6.26). The Matching 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. Drummers will go anywhere that there is employment. The goal is to make the most 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) 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 potential match. Now add a source and a sink node to make a flow diagram.

(C) Express each as a language decision problem.

(D) Show that Matching $\leq_p$ Max-Flow.

✓ 6.28 We will show that the 3-SAT problem is inter-reducible with SAT. These problems assume that the propositional logic expressions are in Conjunctive Normal form, made of clauses joined with $\wedge$'s, so they have the form $C_0 \wedge C_1 \ldots$, where each clause consists of literals joined by $\vee$'s. A propositional logic statement S is satisfiable if there is an assignment of the truth values T or F to the Boolean variables in that statement making it evaluate to true.

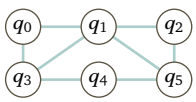
(A) Show the easy half, that 3-SAT $\leq_p$ SAT.

(B) For the other direction start with a clause $C = p_0 \vee p_1 \vee p_2 \vee p_3 \vee p_4$ with more than three literals (each p_i is either a Boolean variable P_i or its negation $\neg P_i$). Introduce a new Boolean variable A and consider the expression $E = (A \vee p_0 \vee p_1) \wedge (\neg A \vee p_2 \vee p_3 \vee p_4)$. Show that an assignment of the truth values T or F to the variables $P_0, \ldots P_4$ makes $C = T$ if and only if $E = T$ under an extension of that assignment that uses the same values for the P_i's and adds a truth value for A.

(C) Show that SAT $\leq_p$ 3-SAT.

✓ 6.29 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) State 3-SAT as a language decision problem.

(D) 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.

(E) With the expression E, make a triangle for each of the two clauses, where the vertices of the first are labeled $c_{0,0}$, $\bar{c}_{0,1}$, and $\bar{c}_{0,2}$, while the vertices of the second are $c_{1,1}$, $c_{1,2}$, and $\bar{c}_{1,3}$. In addition to the edges forming the triangles, also put one connecting $\bar{c}_{0,1}$ with $c_{1,1}$, and one connecting $\bar{c}_{0,2}$ with $c_{1,2}$.

(F) Sketch an argument that 3-SAT $\leq_p$ Independent Set. *Hint:* follow the development in Example 6.4.

6.30 The decision problem Integer Linear Programming starts with a list of linear inequalities with variables $x_0, \ldots x_n$, such as that $a_{i,0}x_0 + \cdots + a_{i,n}x_n \leq b_i$ or that $b_i \leq a_{i,0}x_0 + \cdots + a_{i,n}x_n$, where the numbers $a_{i,j}$ and b_i are integers. It decides if there is a sequence of integers $\langle s_0, \ldots s_n \rangle$ that satisfies all of the constraints.

(A) Consider the propositional logic clause $P_0 \vee \neg P_1 \vee \neg P_2$. Create variables x_0, x_1, and x_2 and list linear constraints such that each must be either 0 or 1. Give a linear inequality that holds if and only if the clause is true.

(B) Show that 3-SAT $\leq_p$ Integer Linear Programming.

6.31 A multivariable polynomial is one with more than one variable, such as $p(x, y) = x^2 - 2y^2 - 1$. Consider the problem D of deciding whether such a polynomial has any integer roots. We will show that 3-SAT $\leq_p$ D.

(A) State D as a language decision problem.

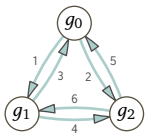
(B) Recall that 3-SAT inputs CNF propositional logic expressions. In CNF a clause is T if and only if any of its literals is T. For instance, $E_0 = P_0 \vee \neg P_1$ is true if and only if $P_0 = T$ or $\neg P_1 = T$. Associate E_0 with the set $S_{E_0} = \{x_0(1-x_0), x_1(1-x_1), x_0(1-x_1)\}$ of three polynomials. Argue that all three equal 0 if and only if both x_0 and x_1 has 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} into a single polynomial so 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 3-SAT $\leq_p$ D.

6.32 Our definition of the Traveling Salesman problem is based on an undirected graph. The Asymmetric Traveling Salesman problem takes place on a directed graph, where the graph's matrix may be asymmetric across the upper left to lower right diagonal, as below. (When there is no path the matrix entry is blank.)



$$\begin{array}{c} \\ g_0 \\ g_1 \\ g_2 \end{array} \begin{array}{c} \begin{array}{ccc} g_0 & g_1 & g_2 \end{array} \\ \begin{pmatrix} & 1 & 2 \\ 3 & & 4 \\ 5 & 6 & \end{pmatrix} \end{array}$$

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) For the other direction, an asymmetric problem instance is a directed graph $\mathcal{G}$ with vertices $g_0, \ldots g_n$. (Take all edge weights to be positive because otherwise we can adjust them all up by $-v + 1$ where v is the most negative edge weight.) Make the associated symmetric graph $\mathcal{H}$, having vertices $h_0, \ldots h_n$ and $\hat{h}_0, \ldots \hat{h}_n$, in two stages. First, if g_i is connected to g_j with weight w then

connect $\hat{h}_i$ to h_j, and also h_i to $\hat{h}_j$, with weight w. Using $\mathcal{G}$ shown above, produce the matrix for this graph, verify that it is symmetric, and draw it.

(C) For the second stage of defining $\mathcal{H}$, let w be the product of $\mathcal{G}$'s maximum edge weight and twice the number of its vertices. For all indices i, connect h_i with $\hat{h}_i$ in both directions with an edge of weight $-w-1$. Verify that the resulting graph matrix is symmetric and draw the graph.

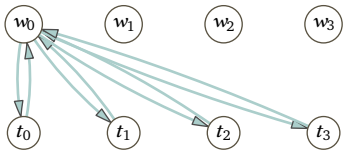
(D) Show that Asymmetric Traveling Salesman $\leq_p$ Traveling Salesman.

6.33 The Assignment problem inputs a set of workers $W = \{w_0, \ldots w_{n-1}\}$ and a same-sized set of tasks $T = \{t_0, \ldots t_{n-1}\}$. For each pair there is a cost $C(w_i, t_j)$, and the problem is to assign a task to each worker for minimal total cost. The Asymmetric Traveling Salesman problem asks for a minimal cost circuit in a directed weighted graph.

(A) By eye, solve this Assignment problem instance.

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) Show that Assignment $\leq_p$ Asymmetric Traveling Salesman. *Hint:* have the reduction function output a graph that is bipartite, i.e, 2-colorable, where each worker is connected to each task by an edge in each direction (this shows only the paired edges for w_0 because showing them all is less clear).



Use the reduction function's input, the assignment cost table, to make edge weights that result in associating optimal assignments with optimal circuits.

6.34 Must a set be polytime reducible to its complement?

(A) Show that $\mathbb{N}$ is not polytime reducible to the empty set.

(B) Show that if $A \leq_p B$ and B is computably enumerable then A is computably enumerable. Conclude that $K^c \not\leq_p K$.

(C) Prove that $\mathcal{L} \leq_p \mathcal{L}^c$ if and only if $\mathcal{L}^c \leq_p \mathcal{L}$.

✓ 6.35 Lemma 6.9 leaves a couple of points undone. (A) Show that $\leq_p$ is reflexive and transitive. (B) It says that if $\mathcal{L}_1 \in P$ then every language $\mathcal{L}_0$ that is nontrivial, where $\mathcal{L}_0 \neq \emptyset$ and $\mathcal{L}_0 \neq \Sigma^*$, satisfies that $\mathcal{L}_1 \leq_p \mathcal{L}_0$. What about trivial languages: which languages reduce to the empty set? to Σ^*? (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 Recall that B is Turing reducible to A, denoted $B \leq_T A$, if there is a Turing machine using oracle A that computes B, so that $\phi_e^A = \mathbb{1}_B$ for some $e \in \mathbb{N}$. And, B is many-one reducible to A, denoted $B \leq_m A$, if there is a total computable function f so that $x \in B$ if and only if $f(x) \in A$. We will show that Turing reducibility is more

general, that a many-one reduction is also a Turing reduction but some Turing reductions are not many-one reductions. (A) Show that if $B \leq_m A$ then $B \leq_T A$. (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.

6.37 Is there a connection between subset and polytime reducibility? Find languages $\mathcal{L}_0, \mathcal{L}_1 \in \mathscr{P}(\mathbb{B}^*)$ 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. But the interest in the class is that it also contains lots of problems that seem to be hard, in that for decades experts have been unable to find fast algorithms for them. Are they are indeed inherently 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 a solution for one problem — specifically, an efficient solution — gives a solution for other problems. 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 and Leonid Levin b 1948

We call these *NP* problems 'maximally hard' because if we could solve one of these then we could solve all problems in *NP*. So each of these is at least as hard. as any other problem in *NP*.

7.1 Theorem (Cook-Levin Theorem) The Satisfiability problem, SAT, is in *NP* and has the property that any problem in *NP* reduces to it: $\mathcal{L} \leq_p$ SAT for any $\mathcal{L} \in NP$.

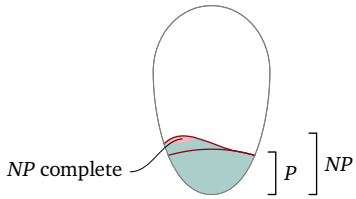
We have already observed that SAT $\in NP$ because, given a Boolean expression, we can use as a witness ω an assignment of truth values that satisfies the expression.

For the proof's other half, given $\mathcal{L} \in NP$, we must show that $\mathcal{L} \leq_p$ SAT. So we must 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.

In outline: we know that $\mathcal{L}$ is decided by a nondeterministic machine $\mathcal{P}$ in time given by a polynomial q. 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 describes the

constraints on 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 $\mathcal{L}$ is *NP* complete if it is a member of *NP* and every problem in *NP* reduces to it, that is, $\hat{\mathcal{L}} \in NP$ implies that $\hat{\mathcal{L}} \leq_p \mathcal{L}$.[†]



7.3 FIGURE: The blob contains all problems, ordered by $\leq_p$. In the bottom are *P* and *NP*. At the top of $NP - P$ the highlighted area contains the *NP* complete problems.

The Cook-Levin Theorem says that there is at least one *NP* complete problem, namely SAT. We shall see that there are many.

7.4 LEMMA Suppose that $\mathcal{L}_0, \mathcal{L}_1 \in NP$. If $\mathcal{L}_0$ is *NP* complete and $\mathcal{L}_0 \leq_p \mathcal{L}_1$ then $\mathcal{L}_1$ is also *NP* complete.

Proof Exercise 7.34. □

The prior section shows that SAT $\leq_p$ Clique so by the lemma that problem is also *NP* complete.


Richard M Karp
b 1935

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.

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 every problem in *NP* reduces to it, usually by finding that an *NP* complete problem that reduces to it. The list below gives the *NP* complete problems most often used. For instance, we might show that 3-SAT $\leq_p \mathcal{L}$.

[†] The relationship that the *NP* complete problems have with the class *NP* is like the relationship that the problems Turing-equivalent to the Halting problem set K have with the computably enumerable sets. In both cases elements of the subset sit at the top of the entire collection: an *NP* complete set is $\geq_p$ every problem in *NP*, and a set Turing-equivalent to K is $\geq_T$ every computably enumerable set.

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, 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 do the second half of showing that it is *NP* complete by proving that the Hamiltonian Circuit problem reduces to it, Hamiltonian Circuit $\leq_p$ 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 note that the reduction function can run in polytime (it examines all pairs of vertices, which takes time that is quadratic in the number of vertices).

For a given problem $\mathcal{L}$, a common way to show that every *NP* problem reduces to it is to show that every *NP* problem reduces to a special case of $\mathcal{L}$.

7.7 Example The Knapsack problem starts with a multiset of objects $S = \{s_0, \ldots s_{k-1}\}$, each with a natural number weight $w(s_i)$ and value $v(s_i)$, along with a weight bound B and value target T. The goal is to find a knapsack $K \subseteq S$ whose elements have total weight no larger than the bound and total value of at least 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 every *NP* problem reduces to Knapsack. It is sufficient to show that every *NP* problem reduces to a special case. Consider the Knapsack instance where $w(s_i) = v(s_i)$ for all $s_i \in S$, and where $B = T = 0.5 \cdot \sum_{0 \le i < k} w(i)$. This shows that any instance of the Partition problem, which is in the above basic list, can be expressed as a Knapsack instance, so Partition $\le_p$ Knapsack.

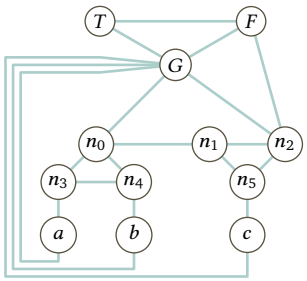
Another common strategy in these proofs is to build the reduction function so that the behavior of the input instance's fundamental units is simulated by subpart of the output instance. Such a construct is a gadget.

7.8 Example Recall that a graph can be 3-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 $\omega = \langle C_0, C_1, C_2 \rangle$ where the C_i partition $\mathcal{G}$'s vertices. Clearly we can produce a polytime verifier that inputs a graph $\mathcal{G}$ along with an ω, and tries to verify that the members of ω are disjoint and together make up all of the vertices, and that never do two same-color vertices lie on the same edge. If the graph is 3-colorable then there is an ω that it can verify, while if the graph is not then there is no such ω.

The other half of showing *NP* completeness is show that every *NP* problem reduces to $\mathcal{L}$. We will sketch the proof that 3-SAT $\le_p$ 3-Coloring. The reduction function inputs a propositional logic expression in Conjunctive Normal form where the clauses never have more than three literals. It outputs a graph that 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 variables represent literals so each could be a Boolean variable P_i or its negation $\neg P_i$). 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 node T or the color of node F. 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 (just as the clause $a \vee b \vee c$ is True if and only if not all of its literals are False).



(*)

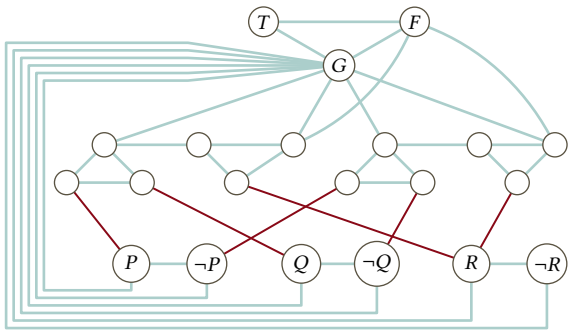
For 'only if', let a, b, and c be the color of F. Then one of n_3 and n_4 is the color of T while the other is the color of G, and hence n_0 is the color of F. Node n_2 is

the color of T because it is connected to both G and F. Those two sentences imply that n_1 is the color of G, and this in turn gives that n_5 is the color of F. But we assumed that c is the color of F so that violates 3-colorability.

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 illustrating what the reduction function outputs when the input is an expression with multiple clauses. Here is the graph for $E = (P \vee Q \vee R) \wedge (\neg P \vee \neg Q \vee R)$.



(**)

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, P and $\neg P$. Similarly there are Q and $\neg Q$ as well as R and $\neg R$. All six share an edge with G so 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 P is connected to $\neg P$, in a 3-colored graph one of these has the color of T and the other has the color of F. The same holds for the other bottom pairs.

The first clause is $P \vee Q \vee R$ so in the left gadget the reduction function outputs the graph with the highlighted edges lead to the nodes marked P, Q, and R. The second clause is $\neg P \vee \neg Q \vee R$ so in the right gadget the edges lead to nodes named accordingly. As the analysis for (∗) shows, this gives a 3-colorable graph if and only if the expression E is satisfiable.

One of Karp's points is 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 non-conflicting times. However, 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 accommodate the requests.

For instance, 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 accommodate 420 classes, this might seem easy. But this college has 1724 students and when each one submits their class requests, $S_i = \{c_{i_0}, c_{i_1}, \ldots c_{i_n}\}$, 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 of 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.

$$\mathcal{L} = \{\langle t, r, \{S_0, S_1, \ldots\}\rangle \;\big|\; \text{a nonconflicting schedule exists}\}$$

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. It is straightforward to write a verifier that inputs $\sigma = \langle t, r, \{S_0, S_1, \ldots\}\rangle$ and ω, and then checks in polytime that ω meets the restrictions.

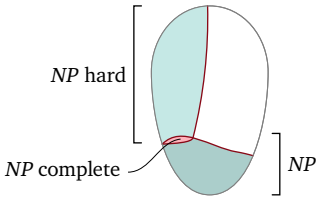
What remains is show that every *NP* problem reduces to this one. This problem is a good fit with the Graph Colorability problem where the nodes of the graph are the classes, where two nodes are connected when some student has requested them both, and where nodes are colored the same if they are in the same time slot. The prior example proves that Graph Colorability is *NP* complete for $k = 3$ colors and an extension of that argument proves the same for any larger k.[‡] So we will show 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. For each student there are two classes that they take, and a graph edge connects those 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.

7.10 DEFINITION A problem $\mathcal{L}$ is *NP* hard if every problem in *NP* reduces to it, that is, if $\hat{\mathcal{L}} \in NP$ implies that $\hat{\mathcal{L}} \leq_p \mathcal{L}$.

[†] We will ignore many issues, such as that lab classes must be in laboratory-equipped rooms. [‡] When $k = 1$ or $k = 2$, the Graph k-Colorability problem is solvable in polytime but no doubt the college has more than one or two rooms.

Thus, a problem is *NP* complete if, in addition to being a member of *NP*, it is also *NP* hard.[†]



7.11 FIGURE: The blob contains all problems, ordered by $\leq_p$. The highlighted intersection is the set of *NP* complete problems.

As the figure illustrates, the collection of *NP* hard problems extends upward.

7.12 LEMMA The equivalent of the Halting problem, the decision problem for $\hat{K} = \{\hat{n} \in \mathbb{B}^* \mid \hat{n} \text{ represents } n \text{ in binary and } \mathcal{P}_n \text{ halts on } n\}$, is *NP* hard but not *NP*. (Here, $\mathcal{P}_n$ is a deterministic Turing machine.)

Proof It is not *NP* because it is not decidable. For *NP* hard fix $\mathcal{L} \in NP$. We will produce a reduction function f that inputs a string $\sigma \in \Sigma^*$ and outputs a representation $\hat{n}$ of an integer, such that $\sigma \in \mathcal{L}$ if and only if $\hat{n} \in \hat{K}$.

The language $\mathcal{L}$ is decided by a nondeterministic Turing machine $\mathcal{P}_{e_0}$ in polytime. This machine halts on every input, either accepting or rejecting. The reduction function outputs the representation $\hat{n}$ of the index n of the following computation. It inputs a string σ and uses e_0 to deterministically simulate running $\mathcal{P}_{e_0}$ on that input. If the simulation accepts σ then this computation halts, while if the simulation rejects then this computation does not halt.

This reduction function runs in polytime.[‡] (A computation of $\phi_n(n)$ may well take more than polytime, because it simulates the nondeterministic machine $\mathcal{P}_{e_0}$, or might not halt at all. But going from the input σ to the output $\hat{n}$ index of that computation takes at most polytime.) Clearly $\sigma \in \mathcal{L}$ if and only if $\hat{n} \in \hat{K}$. □

Before leaving this discussion, we address a natural question: there are lots of problems that are *NP* complete but which problems are not?

Trivially, the definition gives that the empty language and the language of all strings are not *NP* complete, since for each the only set that reduces to it is itself. Also trivially, a problem cannot be *NP* complete if it is not in *NP*. For instance, a problem can be so hard that we cannot even check its solution in polytime (we discussed this earlier with the demon's proposed chess strategy).

The nontrivial aspects of the answer are tied to whether $P = NP$ or $P \neq NP$. We will address this issue in the next subsection but just to not have brushed past it, first note that if $P = NP$ then every nontrivial problem is *NP* hard by Lemma 6.9. So

[†] 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 is a member of that class and is hard for that class.

[‡] As discussed on page 297, the numbers in $\hat{K}$ are represented in binary because If the input was in binary and the output was in unary then f could have exponential runtime.

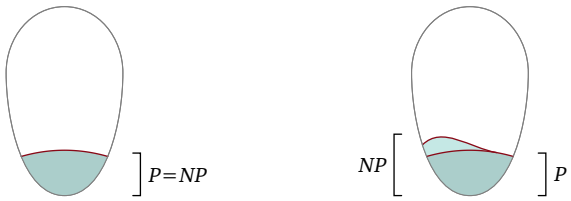
assume that $P \neq NP$. Any algorithms text describes many problems with polytime algorithms and $P \neq NP$ implies that these are not *NP* complete. Thus, if $P \neq NP$ then most problems from everyday programming are not *NP* complete.

In addition, if $P \neq NP$ then there is a problem in $NP - P$ that is not *NP* complete; this result is called Ladner's theorem. Many interesting questions concern these *NP* intermediate problems. The most important one is that 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*?** We have seen that $P \subseteq NP$, and 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?

One of our mental modela of nondeterministic machines is that they are unboundedly parallel. So the *P* versus *NP* question asks: does adding parallelism add speed? Can unbounded parallelism bring problems from super-polynomial to polynomial?

In short, no one knows. We do not know which of these two pictures is right.



7.13 FIGURE: Which is it: $P = NP$ or $P \subsetneq NP$?

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 we will know that $P = NP$. Conversely, if we can show 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 will argue for its importance. As formulated in Karp's original paper, the question of whether *P* equals *NP* might seem of only technical interest.

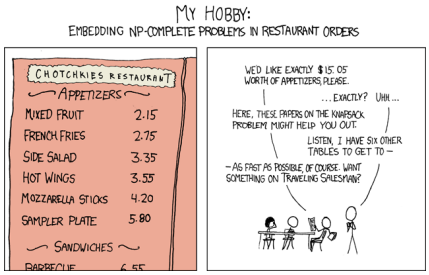
> A large class of computational problems involve the determination of properties of graphs, digraphs, integers, arrays of integers, finite families of finite sets, boolean formulas and elements of other countable domains. Through simple encodings ...

[†] 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. Nonetheless, on classical computers most experts believe that this problem is is in $NP - P$ but not complete.

> 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, which is in the phrase "classic unsolved problems." Karp described that researchers who had been searching for years for an efficient solution to Vertex Cover and those studying Clique are actually working on the same problem, in that the two are inter-translatable, are polytime equivalent. By now the list of *NP* complete problems includes determining the best layout of transistors on a chip, developing accurate financial-forecasting models, 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.



Courtesy xkcd.com

We close this subsection by arguing that among many the important questions that we face, *P* versus *NP* is especially significant, even beyond its practical importance, because of how it fits with the larger ideas of the Theory of Computation.

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've seen many problems that are solvable in principle but where 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. We can view these problems as a transition between the possible and the impossible.[‡]

The sense that the *P* versus *NP* question fits into this larger intellectual setting returns us to the book's opening. Recall the *Entscheidungsproblem* that was a

[†] 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." [‡] This brings to mind the famous lines by the poet R Browning, "A man's reach should exceed his grasp, Or what's a heaven for?"

motivation behind the definition of a Turing machine. It asks for an algorithm that inputs a mathematical statement and outputs whether it is true. Perhaps it is a caricature, but imagine that the job of mathematicians is to prove theorems. Then the *Entscheidungsproblem* asks whether we can replace mathematicians with mechanisms.

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 modify this problem 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, $\sigma_0, \sigma_1, \ldots \sigma_k$, 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 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, is there 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 ω is valid so 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 prove 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,

[†] That is, in a particular subject such as elementary number theory, the set of theorems is computably enumerable. [‡] At the 1930 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 passed away. We don't know if he replied or even read the letter.

> 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 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 up 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, what can be done in a reasonable time (or reasonable space, etc). Taking more time than the lifetime of the universe is not reasonable.

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 Whether *P* equals *NP* is certainly the sexiest question 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 outlines some of the intuition.

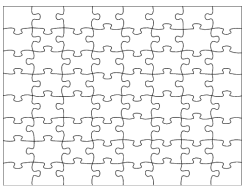
A Selman's plate, courtesy S Selman

One way to think about the question is that a problem is in *P* if finding a solution is fast whereas a problem is in *NP* if verifying the correctness of a given witness is fast. Then $P \subseteq NP$ becomes the observation that if a problem is fast to solve then it must be fast to verify. But the $P \supseteq NP$ inclusion, the contention if a problem has an answer that is fast to verify then that problem must be fast to solve, is not what we would naturally expect. 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 supposition. 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, . . . 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. Similarly, mathematicians find that verifying the correctness of a formally-described proof is routine, while finding that proof in the first place may be the work of a lifetime, or more.

V Strassen has compared our confidence in $P \neq NP$ with our confidence in laws of natural science such as $F = ma$ or $PV = nRT$, "The evidence in favor of $P \neq NP$ . . . is so overwhelming, and the consequences of their failure are so grotesque, that their status may perhaps be compared to that of physical laws rather than that of ordinary mathematical conjectures."

Some commentators have extended this way of thinking beyond the narrow bounds of Theoretical Computer Science. One is A Wigderson, "[M]ost 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 much 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 seems to be hyperbole to say that if $P = NP$ then writing great symphonies would be 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 close this section 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. This chapter began with a story about A Kolmogorov trying to show that multiplication requires n^2 steps only to have a student, A Karatsuba, find a faster way. 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 those 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, it is a genuine possibility 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.

And 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.

D 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.

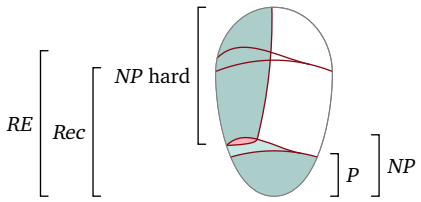
Of course, all this is speculation. Speculating is fun, and in order to make progress in their work, people must have some intuition. But in the end, we look to settle the question with proof.[†]

V.7 Exercises

7.14 This diagram is an extension of one from Theorem 7.11. It orders the universe of problems by $\leq_p$. (Recall that if $A \leq_p B$ then $A \leq_T B$, so it also orders the universe by $\leq_T$. It also assumes that $P \neq NP$ and so the two collections are shown as separate, with P as a strict subset of NP. As a reminder, Rec is

[†] There is also in the air what we could think of as a third side to this debate. There is some recent evidence, as outlined by two Topics at the end of this chapter, that the fact that a problem is NP complete may not mean that in practice it is intractable even for reasonably large-scale instances. Of course this does not change that the P vs NP question is about the limit as the instance size goes to infinity, but in that theory should be at least a rough guide for practice a person can reasonably ask whether the attention given to the question puts the focus of the community in exactly the right place.

the collection of computable languages and *RE* is the collection of computably enumerable languages.



On that, locate these languages.

(A) $K = \{\sigma \mid \sigma \text{ represents in binary } x \in \mathbb{N} \text{ where } \phi_x(x)\downarrow\}$

(B) $E = \{\sigma \mid \sigma \text{ represents in binary an even number}\}$

(C) $\mathcal{L} = \{\langle \mathcal{G}, v_0, v_1, B\rangle \mid \text{there is a path from } v_0 \text{ to } v_1 \text{ of length at most } B\}$

(D) SAT

What about the trivial sets $\varnothing$ and $\mathbb{B}^*$?

✓ 7.15 Your study partner emails you with, "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.

✓ 7.16 Someone in your class says, "I can show that the Hamiltonian Circuit problem is not in *P*, which will demonstrate that $P \neq NP$. The algorithm is: given an instance $\mathcal{G}$, 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 this is not polynomial." Where is their mistake?

✓ 7.17 An online comment 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.

7.18 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.

7.19 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*.

✓ 7.20 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 that we can run today 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.

✓ 7.21 The Traveling Salesman problem is *NP* complete. From $P \neq NP$ which of these statements can we infer? (A) No algorithm solves all instances of

Traveling Salesman. (B) No algorithm solves all instances of Traveling Salesman quickly. (C) Traveling Salesman is in *P*. (D) All algorithms for Traveling Salesman run in polynomial time.

✓ 7.22 For each statement, decide whether it follows from $\mathcal{L}_1 \leq_p \mathcal{L}_0$. (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 $\mathcal{L}_1$.

7.23 Fill in each blank with a set satisfying the condition.

(A) _____ A member of *NP* and *NP* complete.
(B) _____ A member of *NP* but not *NP* complete.
(C) _____ A member of *NP* and *NP* hard.
(D) _____ Not a member of *NP* but *NP* hard.
(E) _____ Not computably enumerable but *NP* hard.
(F) _____ Solvable but not *NP* hard.

7.24 Show that if $P = NP$ then every nontrivial language in *P* is *NP* complete.

7.25 Assume that $P \neq NP$. Show that these are in *NP* but are not *NP* complete.

(A) The even numbers.
(B) The language $\{\mathcal{G} \mid \mathcal{G}$ has a vertex cover of size at most four$\}$. (A vertex cover is a set of vertices where for every edge at least one endpoint is in that 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 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 its input.

Sow how to compute the successor function 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 The section notes that the 3-Satisfiability problem, 3-SAT, is *NP* hard. Prove that 4-Satisfiability is also *NP* hard.

✓ 7.28 The Subset Sum problem inputs a multiset S and a target $T \in \mathbb{N}$, and decides if there is a subset $\hat{S} \subseteq S$ whose elements add to the target. The Partition problem inputs a multiset A and decides whether or not it has a subset $\hat{A} \subset A$ so that the sum of the elements of $\hat{A}$ equals the sum of elements not in $\hat{A}$.

(A) Find a subset of $S = \{3, 4, 6, 7, 12, 13, 19\}$ that adds to $T = 30$.
(B) Partition $A = \{3, 4, 6, 7, 12, 13, 19\}$ into two subsets that add to the same total.
(C) Show that if the sum of the elements in A is odd then it has no partition.

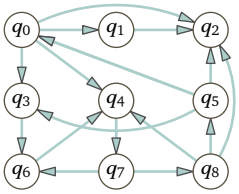
(D) Express each problem as a language decision problem.

(E) Prove that Partition $\leq_p$ Subset Sum. (*Hint:* handle separately the case where the sum of elements in A is odd.)

(F) Conclude that Subset Sum is *NP* complete.

✓ 7.29 The Longest Path problem inputs a graph and finds the longest path that is simple, that does not repeat any vertices. 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.

(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*.

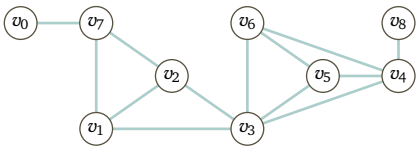
(C) Show that the Hamiltonian Path problem reduces to Longest Path. *Hint:* leverage the bound from the prior item.

(D) Use the prior exercise to conclude that the Longest Path problem is *NP* complete.

✓ 7.30 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.

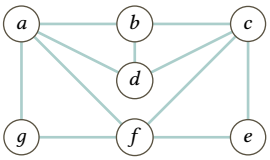
(A) Show that Hamiltonian Path is in *NP*.

(B) This graph has a Hamiltonian path from v_0 to v_8. Find it.



Why must those two be the endpoints?

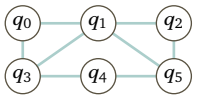
(C) In this graph find a Hamiltonian circuit that begins and ends with node g.



Add a node $\hat{g}$ with the same connections as g, and also add a node $\hat{w}$ that connects only to $\hat{g}$ along with a node w that connects only to g. In that graph find a Hamiltonian path from w to $\hat{w}$.

(D) Show that Hamiltonian Circuit $\leq_p$ Hamiltonian Path.
(E) Conclude that the Hamiltonian Path problem is *NP* complete.

7.31 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) State 3-Satisfiability, 3-SAT, as a language decision problem.
(D) 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.
(E) With the expression E, make a triangle for each of the two clauses, where the vertices of the first are labeled v_0, $\overline{v}_1$, and $\overline{v}_2$, while the vertices of the second are labeled w_1, w_2, and $\overline{w}_3$. In addition to the edges forming the triangles, also put one connecting $\overline{v}_1$ with w_1, and one connecting $\overline{v}_2$ with w_2.
(F) Sketch an argument that 3-Satisfiability $\leq_p$ Independent Set.

7.32 Do we know of any problems in $NP - P$ and that are not *NP* complete?

7.33 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 $\mathcal{L}_1 \in P$.

7.34 Prove Lemma 7.4.

7.35 The class *NP* has some nice closure properties. (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.

7.36 Is the set of *NP* complete sets countable or uncountable?

Section V.8 Look forward: other classes

We close the chapter with an example of a next step that a person could take in exploring the subject. There are many other defined complexity classes.

EXP The Satisfiability problem is a touchstone result among problems in *NP*. On 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.

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.

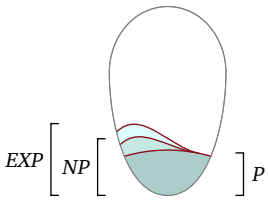
A first, informal, take is that *EXP* contains nearly every problem with which we concern ourselves in practice — it contains most problems that we seriously hope ever to attack. In contrast with polytime, where a rough summary is that

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 The first inclusion is Lemma 5.8. The second is Corollary 5.15. □

We know by a result called the Time Hierarchy Theorem (which lies outside of our scope) that the three classes are not all equal. But where the division is, or divisions are, 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 time that is super-polynomial but sub-exponential. 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.

8.4 DEFINITION Let $f : \mathbb{N} \to \mathbb{N}$. A decision problem $\mathcal{L}$ is an element of $DTIME(f)$ if $\mathcal{L}$ is decided by a deterministic Turing machine that runs in time $\mathcal{O}(f)$. It 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 $c \in \mathbb{N}$.

$$P = \bigcup_{c\in\mathbb{N}} DTIME(n^c) = DTIME(n) \cup DTIME(n^2) \cup DTIME(n^3) \cup \cdots$$

The matching statements hold for NP and EXP.

$$NP = \bigcup_{c\in\mathbb{N}} NTIME(n^c) = NTIME(n) \cup NTIME(n^2) \cup NTIME(n^3) \cup \cdots$$

$$EXP = \bigcup_{c\in\mathbb{N}} DTIME(2^{n^c}) = DTIME(2^n) \cup DTIME(2^{n^2}) \cup DTIME(2^{n^3}) \cup \cdots$$

Proof The only equality that is not immediate is the final one. The issue is the base of 2, since a problem is in EXP if an algorithm for it that runs in time $\mathcal{O}(b^{p(n)})$ for any constant base b and polynomial p. To cover the discrepancy, we will show that $3^n \in \mathcal{O}(2^{(n^2)})$ (this argument works for any base $b > 2$). Consider

$\lim_{x\to\infty} 2^{(x^2)}/3^x$. Rewrite the fraction as $(2^x/3)^x$, which when $x > 2$ is larger than $(4/3)^x$, which goes to infinity. □

8.6 Remark The above description of *NP* reiterates its naturalness. But, as we saw earlier, the characterization that is most useful in practice is that a problem $\mathcal{L}$ is in *NP* if there is a deterministic Turing machine verifier $\mathcal{V}$ that runs in polytime and is such that $\sigma \in \mathcal{L}$ if and only if there is a witness $\omega \in \Sigma^*$ where $\mathcal{V}$ accepts $\langle \sigma, \omega \rangle$.

Space Complexity Researchers have studied many more kinds of complexity classes. For instance, we can also consider how much space a computation uses.[†]

8.7 Definition A deterministic Turing machine runs in space $s : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$[‡] if for all but finitely many inputs σ, the computation on that input uses fewer than $s(|\sigma|)$-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 no more than $s(|\sigma|)$-many cells.

Note that the machine must use at most $s(|\sigma|)$-many cells even on non-accepting computations.

8.8 Definition Let $s : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$. A language decision problem is an element of $DSPACE(s)$, or $SPACE(s)$, if that languages is decided by a deterministic Turing machine that runs in space $\mathcal{O}(s)$. It is an element of $NSPACE(s)$ if it is decided by a nondeterministic Turing machine that runs in space $\mathcal{O}(s)$.

The definitions arise from a sense of a symmetry between time and space. But they are not completely symmetric. 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 $f : \mathbb{N} \to \mathbb{N}$. 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 These are the most essential space complexity classes.

Deterministic logarithmic space $L = DSPACE(\lg n)$

Nondeterministic Logarithmic space $NL = NSPACE(\lg n)$

Polynomial space $PSPACE = DSPACE(n) \cup DSPACE(n^2) \cup \cdots$

Exponential space $EXPSPACE = DSPACE(2^n) \cup DSPACE(2^{n^2}) \cup \cdots$

So for instance, *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 the few results that we have seen suggest, restricting by space instead of time allows for a lot more power.

8.11 Lemma $P \subseteq NP \subseteq PSPACE$

[†] There are still other resources that we could study. One is the number of reads or writes, as distinct from the total number of cells used. [‡] It might seem natural to have the space function output natural numbers, $s : \mathbb{N} \to \mathbb{N}$. But as with the complexity functions in this chapter's first section, we often find real functions convenient, particularly logarithms.

Proof For any problem in *NP*, check all possible witness strings ω. These take at most polynomial space and we can reuse that space when we check the next one. If any such string works then the answer to the problem is 'yes'. Otherwise, the answer is 'no'. □

In the definition of the essential space classes there are two natural classes that are not mentioned. This explains their absence.

8.12 THEOREM (SAVITCH'S THEOREM) For any $f : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$ such that $f(n) \geq \lg(n)$, we have $NSPACE(f) \subseteq DSPACE(f^2)$. Consequently, deterministic polynomial space equals nondeterministic polynomial space, $PSPACE = NPSPACE$. and deterministic exponential space equals nondeterministic exponential space, $EXPSPACE = NEXPSPACE$.

While that result's proof is beyond our scope, nonetheless its statement serves as another 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.

The Zoo The literature contains a great many defined complexity classes. There are so many that they have been gathered into an online Complexity Zoo, at `complexityzoo.net/`.

One way to understand these 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?

For the full story see the Zoo but here is one extra class, to give some flavor of the possibilities.

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's maximal paths accept and if the answer is 'no' then at most one-third of those paths accept. (Here all maximal paths have the same length.) This is often identified as the class of feasible problems for a computer with access to a genuinely-random number source. Investigating whether *BPP* equals *P* is asking whether whether every efficient randomized algorithm can be made deterministic: said with the converse, 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.

EXTRA

V.A RSA Encryption

In this chapter we have built up the sense that there are functions that are intractable to compute. Here we see how to leverage this to engineering advantage. We will describe the celebrated RSA encryption system.

One of the great things about the interwebs — besides the free books — is that we can buy stuff. We input a credit card number and a couple of days later the stuff appears. For this to be practical, the credit card number must be encrypted.

This encryption scheme is subject to a number of constraints. One is that when we visit a web site using a `https` address, that site sends information, called an encryption key, that our browser uses to encrypt the card number. The web site then uses a different key, the decryption key, to get back the original number. It is is critical that the decrypter differs from the encrypter since people on the net can see the encryption information that the site sent, 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.

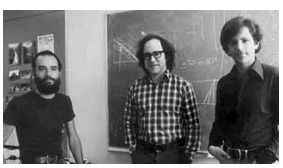
The arithmetic Consider the message 'send money'. As a bitstring[†] it is this (the linebreak is after the space).

```
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 that we do on a computer, including encryption systems, as numerical operations.

To make encryption systems, mathematicians and computer scientists have leveraged that there are there are things we can do easily but that we do not know how to easily undo. That is, there are operations that we can use for encryption that are fast, but such that the operations needed for decryption (without the decrypter) are believed to be so slow that they are completely impractical.

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, contrast the time that it takes to multiply two randomly chosen four-digit numbers by hand with the time that it takes to factor a randomly chosen eight-digit number. For that second job set aside an afternoon; it'll take a while.



Adi Shamir (b 1952), Ron Rivest (b 1947), Leonard Adleman (b 1945)

The algorithm exploits this difference. It was invented in 1976 by 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

[†] In UTF-8.

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 M 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 $m^e \bmod n$. To decrypt it, to recover m, calculate $(m^e)^d \bmod n$. 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 what we believe is an an extremely difficult job.

To choose the three n, e, and d, 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. 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. Cautious people might use 3 072 bits.

1.1 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 $e = 3\,533$. 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 a bitstring that's '`01001000 01101001`'. If he converted that into a decimal number it would be bigger than n so he breaks it into two substrings, getting the numbers 72 and 105. With Alice's public key he computes

$$72^{3533} \bmod 11413 = 10496 \qquad 105^{35333} \bmod 11413 = 4861$$

and sends her the sequence $\langle 10496, 4861\rangle$. Alice recovers his message by using her private key.

$$10496^{6597} \bmod 11413 = 72 \qquad 4861^{6597} \bmod 11413 = 105$$

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

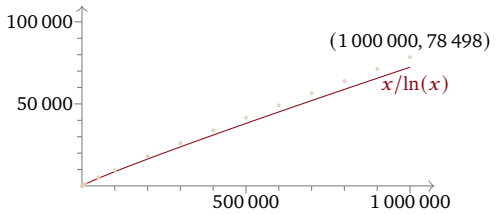
incomplete unless we know what the advantage is of these particular operations. The important point is that they can be done quickly. But undoing them, decrypting, is believed to take a very long time without the key.

To illustrate we show that we can quickly test for primality. We begin 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 $n/\ln(n)$.

$$\lim_{n\to\infty} \frac{\text{number of primes less than } n}{(n/\ln n)} = 1$$

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.

1.3 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. A more efficient way is to 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 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.

1.4 EXAMPLE Consider raising 4 to the 13-th power, modulo $m = 497$. The naive approach of computing 4^{13} and then reducing produces a very large number along the way. Better is to express 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.

1.5 Example Recall Euclid's Algorithm that uses division with remainder to find the greatest common divisor of two numbers. Here we find $\gcd(1071, 462)$. Start with $1071 = 2 \cdot 462 + 147$, then compute $462 = 3 \cdot 147 + 21$, then $147 = 7 \cdot 21 + 0$. This shows that the greatest common divisor is 21.

Reversing the final equation gives $21 = 462 - 3 \cdot (147)$. Substituting for 147 from the equation before the last gives $21 = 462 - 3 \cdot (1071 - 2 \cdot 462) = -3 \cdot 1071 + 3 \cdot 462$. This expresses the greatest common divisor as a linear combination of the two starting numbers.

1.6 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, $sa + tm$ 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 $sa \equiv 1 \pmod{m}$, making s the modulo m inverse of a. □

Euclid's algorithm is efficient, both in time and space, so finding an inverse modulo m is efficient.

With that, we are ready to efficiently test whether a number is prime. The naive way is to try dividing the number by all possible factors. A faster way is based on the next result.

A.7 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 $i \in \{1, \ldots p-1\}$ and reduce modulo p to get numbers $r_i = ia \bmod p$.

We will show that the set $R = \{r_1, \ldots r_{p-1}\}$ equals the set $P = \{1, \ldots p-1\}$. First we argue that $R \subseteq P$. Because p is prime and does not divide i or a, it does not divide their product ia. Thus each r_i is not equivalent modulo p to 0, and so each r_i is a member of $\{1, \ldots p-1\}$.

We finish by claiming that the two sets have the same number of elements, that $r_{i_0} \neq r_{i_1}$ for any unequal two numbers $i_0, i_1 \in P$. Write $r_{i_0} - r_{i_1} = i_0 a - i_1 a = (i_0 - i_1)a$. Because p is prime and does not divide $i_0 - i_1$ or a, as each is smaller in absolute value than p, it therefore does not divide their product. So $r_{i_0} - r_{i_1}$ is not zero.

Finish by taking the product, modulo p, of the numbers $a, 2a, \ldots (p-1)a$.

$$a \cdot 2a \cdots (p-1)a \equiv 1 \cdot 2 \cdots (p-1) \pmod{p}$$
$$(p-1)! \cdot a^{p-1} \equiv (p-1)! \pmod{p}$$

Cancel the $(p-1)!$'s to get the result. □

1.8 Example Let the prime be $p = 7$. Of course, all natural numbers a with $0 < a < p$ are 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	4 096	15 625	46 656
$(a^6 - 1)/7$	0	9	104	585	2 232	6 665

Fermat's Little Theorem gives a test for non-primality: given n, if we find a base a with $0 < a < n$ so that a^{n-1} mod n is not 1 then n is not prime.

1.9 Example Let $n = 415\,692$. 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} mod $n = 1$, then we may start to think that n is prime after all.[†]

This suggests 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 this 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. 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.

1.10 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.

[†] 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, these are rare.

1.11 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 $pq = n$. □

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.12 There are twenty five primes less than or equal to 100. Find them.

✓ A.13 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 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 uses 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.14 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.

EXTRA

V.B Good-enoughness

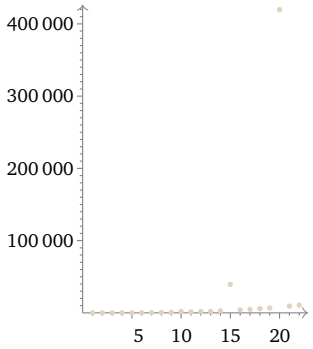
A theory shapes the way that we see the world, how we address what comes before us in practice. For example, Newton's $F = ma$ is a program for analyzing physical situations: if there is an acceleration then we look around for a force. Likewise, Darwin's theory tells us that if we see a change in a species then look for a engineering advantage that can be passed on in reproduction.

Here we will point to a way in which a naive understanding of Complexity Theory can lead to a misunderstanding of of what can be done in practice. Of course, the theorems are right. 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 as those in the class *P*. However, we have already noted that just because a problem is in *P* does not mean that it has an algorithm that we could use in practice. A problem whose fastest algorithm is $\mathcal{O}(n^{1000})$ or one whose algorithm has a huge coefficient, such as $2^{1000} \cdot n^2$, is just not useful. The flip side is that just because a problem is *NP* hard does not mean that it is hard to solve on instances that we see in practice — maybe, for example, it is possible to solve the first 200 000 instances.

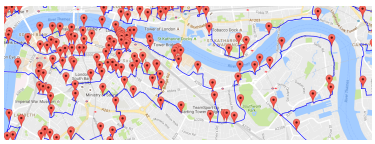
For another way that a problem can fall outside *P* consider this runtime function.



$$f(n) = \begin{cases} n^{\lg n} & \text{– if } n \text{ is a multiple of 5} \\ n^2 & \text{– otherwise} \end{cases}$$

For most inputs f grows slowly but on every fifth one it is super-polynomial. The exceptions could be rarer than that, such as every 10 or every 10^{10} or even every 10^x. The definition of Big $\mathcal{O}$ implies that if there are infinitely many super-polynomial exceptions then the growth of the function as a whole is superpoly. We classify a problem with a best algorithm that is $\mathcal{O}(f)$ as hard. However, if the exception are extremely rare then for any single instance the chance of a fast runtime is awfully good. Problems where there are rare hard instances but most are solvable in practice are said to have black holes.

In short, it is an incomplete understanding to think that *NP* hard problems are sure to be too slow to solve except for extremely small inputs.



London pubs, via Google Earth

An *NP* complete problem for which there are very capable algorithms is the Traveling Salesman problem. We can in a reasonable time find solutions for problem 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 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 by 0.000 023 percent, required eight years of computation time running in parallel on a network of Linux workstations.

Sweden tour

There are a number of systems for solving the Traveling Salesman problem that are widely available. The Free mathematics system *Sage* includes one. Here is a brief example of that. 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.

```
sage: g = graphs.HeawoodGraph()
sage: tsp = g.traveling_salesman_problem()
sage: tsp.adjacency_matrix()
[0 0 0 0 0 1 0 0 0 0 0 0 1]
[0 0 1 0 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 0 1 0 0 0 0 0 0 0 1 0]
[0 0 0 1 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 0 0 0 1 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 0 0 0 1 0 0 0 0 0 1]
[0 0 0 0 1 0 0 0 0 0 1 0 0 0]
[0 1 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 1 0]
```

A wonderful overview talk, *Information, Computation, Optimization* . . . by W Cook is available online; see (YouTube channel Joint Mathematics Meetings 2018).

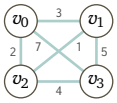
In summary, even though the theory says that a problem is hard does not mean it that it cannot be solved for at least some non-toy instances. The next Extra section makes the same point, for the SATproblem.

V.B Exercises

B.1 Critique this from social media: "The Traveling Salesman problem is *NP* hard. That means that algorithms to solve the problem are very slow. So for an input of 100 vertices 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 Traveling Salesman Problem solution for a circuit involving the graph below.



EXTRA

V.C SAT solvers as oracles

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 reasonably large instances.

Our touchstone hard problem is SAT. This section demonstrates a SAT solver, and uses 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 Soduku.

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

In case it is unfamiliar, the popular Soduku 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 th3 restrictions that every row, column, and subsquare must contain each of the numbers 1–9 once and only once.

As to how hard the problem is, studying the computational complexity of a problem requires that we describe how a solution algorithm's use of some resource grows with the input size. So we reframe the problem to allow instances of different input sizes. The natural way to generalize the puzzle is: instead of eighty one variables $x_{1,1}, \ldots x_{9,9}$ we could use an arbitrary number, $x_1, \ldots x_n$. Instead of those variables taking on 1–9, they could take on a value in 1–k (we can call these 'colors'). And, in place of rows, columns, and subsquares that are driven by the geometry, a problem instance could have arbitrary sets, $S \subseteq \{x_1, \ldots x_n\}$ of size k. With those adjustments, we can show that the Soduku problem is *NP* complete.

However, having noted this generalization, here we limit our attention to the traditional 9×9 board.

To solve puzzle instances using the SAT solver as an oracle we need a reduction Soduku $\leq_p$ SAT. We will produce a function that inputs game boards and outputs Propositional Logic expressions. Those expressions will be in Conjunctive Normal form, CNF. These are the conjunction of clauses where each clause is the disjunction of literals. Two examples are $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$ and $(x_1 \vee x_2 \vee \neg x_3) \wedge x_4 \wedge (\neg x_2 \vee x_3)$. More is in Section C.

The SAT solver that we use needs its input file formatted to a standard called DIMACS. This file format starts with comment lines, with the initial 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*. The variable $x_{1,1,v}$ is T if in our solution the number in row 1 and column 1 is v. Thus, if for instance in the first row and first column the solution 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 $r, c, v \in \{1, \ldots 9\}$ 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 particular starting board instance. 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 $x_{1,1,1}$, $x_{1,1,2}$, $\ldots$ $x_{1,1,9}$ 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 of the puzzle must contain each number 1, ... 9, so that is nine restrictions. For instance, to express that the first row contains an entry with the value 2, in the Propositional Logic statement we include this clause.

$$x_{1,1,2} \vee x_{1,2,2} \vee x_{1,3,2} \vee x_{1,4,2} \vee x_{1,5,2} \vee x_{1,6,2} \vee x_{1,7,2} \vee x_{1,8,2} \vee x_{1,9,2}$$

So the DIMACS format requires 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) )`. The Racket code below produces expresses 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.
```

[†] 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.

```
(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)))
```

The column restrictions are much the same.

```
;; 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 output 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 that code, 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 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.

```
;(define INITIAL-CLAUSES
;  (list (list '(1 3 9)) ; there is a 9 in position (1,3)
;        (list '(1 8 1))
;        (list '(1 9 5))
;        (list '(2 1 5))
;        (list '(2 4 4))
```

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 (saying 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  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 list of one num
    (apply format "~a 0\n" variable-in-clause))
  (define (one-line-of-two variables-in-clause) ; a line from list of two nums
    (apply format "~a ~a 0\n" variables-in-clause))
  (define (one-line-of-nine variables-in-clause) ; a line from list of nine nums
    (apply format "~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.

```
; Exercise
(define INITIAL-CLAUSES
  (list (list '(1 2 3)) ; there is a 3 in position (1,2)
        (list '(1 3 4))
        (list '(1 4 5))
        (list '(1 6 6))
        (list '(1 7 9))
        (list '(2 3 5))
        (list '(2 4 4))
        (list '(3 5 8))
        (list '(3 8 1))
        (list '(4 4 8))
        (list '(4 5 2))
```

```
        (list '(4 6 3))
        (list '(4 9 7))
        (list '(5 1 1))
```

After running this

```
 > (dump-to-file)
```

the first few lines of the file `soduku.cnf` look like this.

```
c soduku.cnf
c DIMACS format file for SAT solver
c 2025-03-15 Jim Hefferon, hefferon.net.  Public Domain.
p cnf 729 3184
164 0
246 0
328 0
411 0
655 0
336 0
256 0
590 0
26 0
598 0
```

Remember that the `0`'s separate clauses.

We can now run the SAT solver, MiniSat (Eén and Sörensson 2005).

```
ftpmaint@millstone:~/computing/src/scheme/complexity$ minisat soduku.cnf soduku.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         |          LEARNT          | 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
```

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 to show the entire board are in `sat-solver.rkt`.

```
> (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.

V.C Exercises

C.1 This board is described online as an especially hard Soduku. 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						

EXTRA

V.D The Bounded Halting problem

This chapter's final section developed the intuition that the class of *NP* complete problems forms a transition between the solvable problems and the unsolvable ones. Here we support that view by providing a connection.

The signature unsolvable problem is the Halting problem.

D.1 PROBLEM (Nondeterministic Bounded Halting PROBLEM) Given an index and a step bound, $\langle e, S\rangle$, where S is specified in unary, decide whether the nondeterministic Turing machine $\mathcal{P}_e$ halts on an empty tape within S steps. That is, decide whether $\mathcal{P}_e$'s computation tree contains a maximal path that reaches a halting state in no more than S many transitions.

This problem is solvable: for instance, we could use a deterministic machine to simulate all of $\mathcal{P}_e$'s at most 2^S possible paths and just check each for a halting state.

Thus, on a serial machine the naive approach to this problem is exponential. Naturally we ask if we can find a faster way.

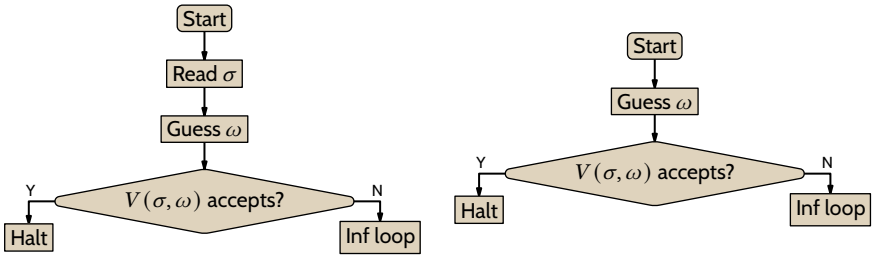
D.2 THEOREM The Nondeterministic Bounded Halting problem is *NP* complete.

Proof Denote the Nondeterministic Bounded Halting problem's language with $\mathcal{B}$.

We first argue that $\mathcal{B}$ is in *NP*. We can write a verifier that inputs a problem instance $\langle e, S\rangle$ and a witness that represents a halting branch (it might be a bitstring, where `0` and `1` describe the two ways that the branch can go). Clearly $\langle e, S\rangle$ is in $\mathcal{B}$ if and only if there exists such a witness. Also, the verifier can run in time polynomial in $|\langle e, S\rangle|$ because S is given in unary (in time S there are at most $|S|$ branchings).

What remains to argue is that $\mathcal{B}$ is *NP* hard. We will show that any $\mathcal{L} \in NP$ has $\mathcal{L} \leq_p \mathcal{B}$. Each such $\mathcal{L}$ has an associated polytime verifier, a deterministic Turing machine $\mathcal{V}$ where $\sigma \in \mathcal{L}$ if and only if there exists an ω so that $\mathcal{V}$ accepts $\langle \sigma, \omega\rangle$ and where $\mathcal{V}$ runs in time bounded by $p(|\sigma|)$ for some polynomial p.

For the reduction we need a computable function f so that $\sigma \in \mathcal{L}$ if and only if $f(\sigma) \in \mathcal{B}$. On the left below is a nondeterministic Turing machine (recall that one of our mental models is that such machines guess values, or receive them from a demon). Let this machine have index e_0. By the *s-m-n* theorem the one on the right is $\mathcal{P}_{s(e_0,\sigma)}$.



Then $\sigma \in \mathcal{L}$ if and only if $f(\sigma) = \langle s(e_0, \sigma), p(|\sigma|)\rangle \in \mathcal{B}$, as required. ∎

Part Four

Appendices

Appendix A Strings

An alphabet is a nonempty and finite set of symbols or 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.

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 `x`, or `y`, or `lastname` as variable names. An example of a string is `(or a ready)`, which is a sequence of five alphabet elements, $\langle$`(`, `or`, `a`, `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 items in the string with the associated lower case roman letter, as in $\sigma = \langle s_0, \ldots\ s_{n-1} \rangle$ and $\tau = \langle t_0, \ldots\ t_{m-1} \rangle$. 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 $|\varepsilon| = 0$.

In place of s_i we sometimes use square brackets, $\sigma[i]$. This form allows us to use $\sigma[-1]$ for the final character, $\sigma[-2]$ for the one before it, etc. We also write $\sigma[i{:}j]$ for the subsequence of symbols between s_i and s_j, including s_i but not s_j, and we write $\sigma[i{:}]$ for the tail that starts with s_i as well as $\sigma[{:}j]$ for $\sigma[0{:}j]$.

We usually work with alphabets having single-character symbols for the convenience of writing strings by omitting the brackets and commas. That is, we write $\sigma =$ `abc` instead of $\langle$`a`, `b`, `c`$\rangle$.[‡] This comes with the disadvantage that without the diamond brackets the empty string is just nothing, which is why we reserve a separate symbol, ε.[#]

The alphabet consisting of the bit characters is $\mathbb{B} = \{$`0`, `1`$\}$ (we sometimes instead use $\mathbb{B}$ for the set $\{0, 1\}$ of the bits themselves). Strings over $\mathbb{B}$ 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."

There are only a few string operations. Let $\sigma = \langle s_0 \ldots\ s_{n-1} \rangle$ and $\tau = \langle t_0, \ldots\ t_{m-1} \rangle$ be strings over an alphabet Σ. The concatenation $\sigma^\frown\tau$ or $\sigma\tau$ appends the second string to the first, $\sigma^\frown\tau = \langle s_0 \ldots\ s_{n-1}, t_0, \ldots\ t_{m-1} \rangle$. Where $\sigma = \tau_0{}^\frown\cdots{}^\frown\tau_{k-1}$, we say that σ decomposes into the τ's, and that each τ_i is a substring of σ. The first

[†] 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.

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 $\sigma^2 = \sigma^\frown\sigma$ and $\sigma^3 = \sigma^\frown\sigma^\frown\sigma$, etc. We write $\sigma^1 = \sigma$ and $\sigma^0 = \varepsilon$. The reversal σ^R of a string takes the symbols in reverse order: $\sigma^R = \langle s_{n-1}, \ldots s_0\rangle$. The empty string's reversal is $\varepsilon^R = \varepsilon$.

For example, let $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and let $\sigma = \mathsf{abc}$ and $\tau = \mathsf{bbaac}$. 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 $\alpha = \mathsf{abba}$, $\beta = \mathsf{cdc}$, and ε.

Exercises

A.1 Let $\sigma = \mathtt{10110}$ and $\tau = \mathtt{110111}$ be bit strings. Find each. (A) $\sigma^\frown\tau$ (B) $\sigma^\frown\tau^\frown\sigma$ (C) σ^R (D) σ^3 (E) $\mathtt{0}^3{}^\frown\sigma$

A.2 Let the alphabet be $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. Suppose that $\sigma = \mathsf{ab}$ and $\tau = \mathsf{bca}$. Find each. (A) $\sigma^\frown\tau$ (B) $\sigma^2{}^\frown\tau^2$ (C) $\sigma^R{}^\frown\tau^R$ (D) σ^3

A.3 Let $\mathcal{L} = \{\sigma \in \mathbb{B}^* \mid |\sigma| = 4 \text{ and } \sigma \text{ starts with } \mathtt{0}\}$. How many elements are in that language?

A.4 Suppose that $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and that $\sigma = \mathsf{abcbccbba}$. (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 $|\sigma^\frown\tau|$? 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 then prove it and if not then give a counterexample. (A) $\alpha^\frown\varepsilon = \alpha$ and $\varepsilon^\frown\alpha = \alpha$ (B) $\alpha^\frown\beta = \beta^\frown\alpha$ (C) $\alpha^\frown\beta^R = \beta^R{}^\frown\alpha^R$ (D) $\alpha^{RR} = \alpha$ (E) $\alpha^{iR} = \alpha^i$

Appendix B Functions

A function is an input-output relationship: each input is associated with a unique output. An example is the association of an input natural number with the output number that is its square. Another is the association of a string of characters with the natural number length of that string. A third is the association of a polynomial $a_n x^n + \cdots + a_1 x + a_0$ with the Boolean value T or F, depending on whether 1 is a root of that polynomial.

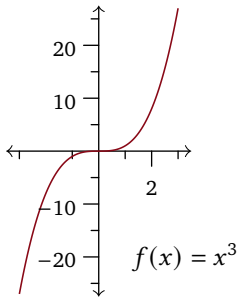
Contrary to what is stated in many introductions, a function isn't a 'rule'. The function that associates a year number with that year's winners of the US baseball World Series isn't given by any rule simpler or more sensible than an exhaustive listing of the cases. Nor is there a rule in the kind of functional association that a database might have, such as linking the government ID of citizens to their birth date. True, many functions 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

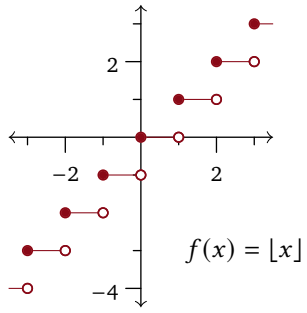
function is that for each input there is exactly one associated output. If we can go from an input to the associated output with a calculation then that's great but even if we cannot, it is still a function.

For a formal definition fix two sets, a domain D and a codomain C. A function or map, $f\colon D \to C$, 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 this 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 $f\colon D \to C$ 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.

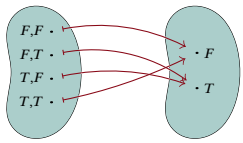
Some functions take more than one input, such as $\text{dist}(x, y) = \sqrt{x^2 + y^2}$. We say that this function is 2-ary, while some other functions are 3-ary, etc. The number of inputs is the function's arity. If the function takes only one input but that input is a tuple then we often drop the parentheses, so we write $f(3, 5)$ instead of $f((3, 5))$.

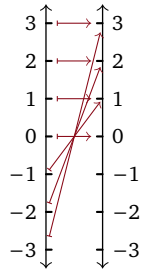
Pictures We often illustrate a function by using the familiar xy axes.





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 of the bean diagram, showing the absolute value function mapping integers to integers.





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 $g\colon \mathbb{R} \to \mathbb{R}$ 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 $f : \mathbb{Z} \to \mathbb{Z}$ given by $x \mapsto 2x$ is the set of even integers.

The difference between the range and codomain is that the codomain is a convenient superset of the range. An example is that for the function $f(x) = \sqrt{2x^4 + 2x^2 + 15}$, we may be content to note that the polynomial is always non-negative and so the output is real, writing $f : \mathbb{R} \to \mathbb{R}$, rather than troubling to find its 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 $g(r) = 1 + r + r^2 + \cdots$ 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 $f : D \to C$ and want to shrink the domain to some subset $S \subseteq D$. The restriction $f \restriction_S$ is the function with domain S and codomain C defined by $f \restriction_S(x) = f(x)$.

Where the domain of a function is finite, the number of elements in the domain is less than or equal to the number in the range.

In the Theory of Computation we sometimes uses these terms in a way that is at odds with the definitions above. For our functions, 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 domain, although the 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' is used to connote that f may be not defined on some $d \in D$.

Well-defined The definition of a function includes the condition that for each domain element there cannot be two associated codomain elements. We refer to this property by saying that functions are well-defined.

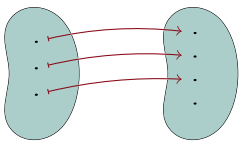
When we are considering a relationship between x's and y's and asking if it is a function, typically it is well-definedness that is at issue.[†] For instance, consider the set of ordered pairs (x, y) where $y^2 = x$. If $x = 9$ then it is related to both $y = 3$ and $y = -3$ so this is not a functional relationship — it is not well-defined. 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 between the input of email name and the output of person to deliver it to be not well-defined.

[†] Sometimes people say that they are, "checking that the function is well-defined." Strictly speaking this is confused because if it is a function then it is by definition well-defined. However, natural language is funny this way — while all tigers have stripes, we may say "striped tiger."

For a function that is suitable for graphing on xy axes, $f\colon \mathbb{R} \to \mathbb{R}$, 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.

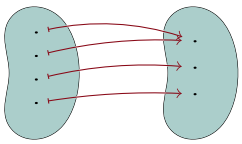
One-to-one and onto The definition of function has an asymmetry: among the ordered pairs (x, y), it requires that each domain element x be in one pair and only one pair, but it does not require the same of the codomain elements.

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 on the right, there is at most one arrow ending at y.



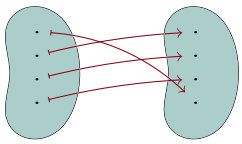
An example is that when the domain is a subset of the codomain, $D \subseteq C$, then the inclusion function $\iota\colon D \to C$ is defined by $\iota(x) = x$. The most common way to prove that a function f is one-to-one is to assume that $f(x_0) = f(x_1)$ and then argue that therefore $x_0 = x_1$. 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 intercepts the function's graph in at most one point.

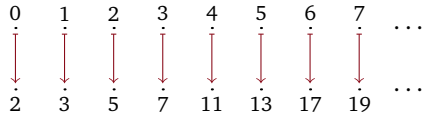
A function is onto (or a surjection) if each codomain element y is in at least one pair. Thus, a function is onto if its codomain equals its range. The function below is onto because every element in the codomain bean has at least one arrow ending at it.



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 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.

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 $f : \mathbb{R} \to \mathbb{R}$, 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 is finite, if a function is a correspondence then its domain has the same number of elements as its range.

Composition and inverse If $f : D \to C$ and $g : C \to B$ then their composition $g \circ f : D \to B$ 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 one thing, they are unequal when $x = \pi$. Composition can fail to commute more dramatically: if $f : \mathbb{R}^2 \to \mathbb{R}$ is given by $f(x_0, x_1) = x_0$, and $g : \mathbb{R} \to \mathbb{R}$ 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. It follows that the composition of correspondences is a correspondence.

An identity function $\mathrm{id} : D \to D$ is given by $\mathrm{id}(d) = d$ for all $d \in D$. It acts as the identity element in function composition, so that if $f : D \to C$ then $f \circ \mathrm{id} = f$ and if $g : C \to D$ then $\mathrm{id} \circ g = g$. As well, if $h : D \to D$ then $h \circ \mathrm{id} = \mathrm{id} \circ h = h$.

Given $f : D \to C$, if $g \circ f = \mathrm{id}$ 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 $f, g : \mathbb{R} \to \mathbb{R}$ 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 Prove each.

(A) Let $g : \mathbb{R}^3 \to \mathbb{R}^2$ be the projection map $(x, y, z) \mapsto (x, y)$ and let $f : \mathbb{R}^2 \to \mathbb{R}^3$ be the injection map $(x, y) \mapsto (x, y, 0)$. Then g is a left inverse of f but not a right inverse.

(B) The function $f : \mathbb{Z} \to \mathbb{Z}$ given by $f(n) = n^2$ has no left inverse.

(C) Where $D = \{0, 1, 2, 3\}$ and $C = \{10, 11\}$, the function $f : D \to C$ given by $0 \mapsto 10$, $1 \mapsto 11$, $2 \mapsto 10$, $3 \mapsto 11$ has more than one right inverse.

B.3 These are about two-sided inverses.

(A) Where $f : \mathbb{Z} \to \mathbb{Z}$ is $f(a) = a + 3$ and $g : \mathbb{Z} \to \mathbb{Z}$ is $g(a) = a - 3$, show that g is inverse to f.

(B) Where $h\colon \mathbb{Z} \to \mathbb{Z}$ 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) Find the inverse of $s\colon \mathbb{R}^+ \to \mathbb{R}^+$ given by $s(x) = x^2$.

B.4 Fix $D = \{0, 1, 2\}$ and $C = \{10, 11, 12\}$. Let $f, g\colon D \to C$ 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, and
(D) show that g has no inverse.

B.5 These are about the interaction of composition with the function properties of being one-to-one and onto.
(A) Prove that the composition of one-to-one functions is one-to-one.
(B) Prove that the 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 each.
(A) A function has an inverse if and only if it 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 are $g \circ f$ and $(g \circ f)^{-1} = f^{-1} \circ g^{-1}$.

B.7 Prove these for a function $f\colon D \to C$ with a finite domain. Together they imply that finite sets have the same number of elements if and only if there is a correspondence between them. *Hint:* for each, you can do induction on either $|\operatorname{ran}(f)|$ or $|D|$.
(A) $|\operatorname{ran}(f)| \leq |D|$
(B) If f is one-to-one then $|\operatorname{ran}(f)| = |D|$.

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 as 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 F.

We can operate on propositions, as with 'it is not the case that 8 is prime' or '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).

	not P
P	$\neg P$
F	T
T	F

		P and Q	*P or Q*
P	Q	$P \wedge Q$	$P \vee Q$
F	F	F	F
F	T	F	T
T	F	F	T
T	T	T	T

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. Its third row, the T Frow, in the $P \wedge Q$ column shows a value of F. Observe that the 'and' operator $\wedge$ accumulates F, in that if any of its inputs are F then $P \wedge Q$ is F. Similarly, the 'or' operator $\vee$ accumulates T.

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 sentences in English is that we can express more things, and with complete precision. 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 a natural 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 such as this E by building them up from their components. The table below shows its input/output behavior.

P	Q	R	$P \vee Q$	$R \wedge Q$	$P \vee (R \wedge Q)$	$\neg(P \vee (R \wedge Q))$	E
F	F	F	F	F	F	T	F
F	F	T	F	F	F	T	F
F	T	F	F	F	F	T	F
F	T	T	F	F	F	T	F
T	F	F	F	F	F	T	F
T	F	T	F	F	F	T	F
T	T	F	F	F	F	T	F
T	T	T	F	F	F	T	F

The three '$\neg$', '$\wedge$', and '$\vee$' are operators (or connectives). There are other operators; here are two common ones.

		P implies Q	*P if and only if Q*
P	Q	$P \to Q$	$P \leftrightarrow Q$
F	F	T	T
F	T	T	F
T	F	F	F
T	T	T	T

Two statements are equivalent (or logically equivalent) if they have equal output values whenever we assign the same values to the input variables. For instance, $P \to Q$ 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 $P \to Q \equiv \neg P \vee Q$

The set of formulas describing when statements are equivalent is Boolean algebra. For instance, these are the distributive laws

$$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R) \qquad P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$$

and these are DeMorgan's laws.

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q \qquad \neg(P \vee Q) \equiv \neg P \wedge \neg Q$$

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
F	F	T
F	T	F
T	F	F
T	T	F

P	Q	R	E_1
F	F	F	F
F	F	T	T
F	T	F	T
F	T	T	F
T	F	F	T
T	F	T	F
T	T	F	F
T	T	T	F

To produce the expression focus on the rows with final column T. The E_0 table has one such row, where $P = F$ and $Q = F$. Consequently, we can use $E_0 \equiv \neg P \wedge \neg Q$.

For the E_1 table on the right, again focus on the rows with T in the final column. Target the second row with $\neg P \wedge \neg Q \wedge R$, target the third row with $\neg P \wedge Q \wedge \neg R$, and target the fifth row with $P \wedge \neg Q \wedge \neg R$. Then E_1 joins these with $\vee$'s.

$$E_1 \equiv (\neg P \wedge \neg Q \wedge R) \vee (\neg P \wedge Q \wedge \neg R) \vee (P \wedge \neg Q \wedge \neg R) \qquad (*)$$

Those propositional logic expressions use Boolean variables such as P, which may take on values of either T or F. In the context of the syntax of how these expressions are constructed, we refer to each variable as 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 common connective (we stick to either $\wedge$ or $\vee$), so that $\neg P \wedge \neg Q \wedge R$ is a clause.

The form of the expression in $(*)$ above is important. A propositional logic expression is in Disjunctive Normal form or DNF if 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 following the DNF approach gives $E_0 \equiv \neg\big((\neg P \wedge Q) \vee (P \wedge \neg Q) \vee (P \wedge Q)\big)$. DeMorgan's second law lets us distribute the outside negation, $\neg(\neg P \wedge \neg Q) \wedge \neg(P \wedge \neg Q) \wedge \neg(P \wedge Q)$. Then distribute the three leading negations with DeMorgan's first law, $E_0 \equiv (P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (\neg P \vee \neg Q)$.

Conjunctive Normal form or CNF is a conjunction of clauses, where each clause is a disjunction of literals. We can construct it without the Boolean algebra. Given a truth table such as E_1's, make a clause for each F line and join them with $\wedge$'s.

$$E_1 \equiv (P \vee Q \vee R) \wedge (P \vee \neg Q \vee \neg R) \wedge (\neg P \vee Q \vee \neg R) \\ \wedge (\neg P \vee \neg Q \vee R) \wedge (\neg P \vee \neg Q \vee \neg R)$$

Fill in the clauses by: in the truth table's associated F line, where the variable X is assigned T we write $\neg X$ and where X is assigned F we write the literal as X.

We use CNF more than DNF. This form has the advantage that an expression evaluates to T if and only if all of its clauses are T. And, a clause is T if and only if at least one of its literals is 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 these. (A) $\neg(P \vee Q)$ (B) $\neg P \wedge \neg Q$ (C) $\neg(P \wedge Q)$ (D) $\neg P \vee \neg Q$

C.2 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.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	
F	F	F	F
F	F	T	T
F	T	F	T
F	T	T	F
T	F	F	F
T	F	T	T
T	T	F	F
T	T	T	F

P	Q	R	
F	F	F	T
F	F	T	F
F	T	F	T
F	T	T	F
T	F	F	F
T	F	T	F
T	T	F	T
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 '$P \to Q$' or '$Q \to P$'.

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/npc2007012636/`

Preface

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

intuitively mechanically computable The word 'mechanically' is ambiguous. It can mean performed according to fixed principles or rules or it can mean performed by a machine. In this chapter we take it to mean both, following Turing's approach that any rule-following procedure can be accomplished by a physical mechanism.

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 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 $x \in \mathbb{N}$, Prime(x) implies 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, 1918–2020

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

then 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 an instrument dial or a slide rule 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 all know computations with no 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 $q_4 1 1 q_5$ won't ever be reached, but it does no harm. It is there for the definition of a Turing machine, so that the Δ function is defined on all $q_p T_p$. See also the note to that definition.

transition function The definition describes Δ as a function $\Delta\colon Q \times \Sigma \to (\Sigma \cup \{\mathtt{L}, \mathtt{R}\}) \times Q$. That is a fudge. In $\mathcal{P}_{\text{pred}}$, the state q_3 is used only for the purpose of halting the machine and so there is no defined next state. In $\mathcal{P}_{\text{add}}$, 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 371. (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 $\Delta\colon Q \times \Sigma \to (\Sigma \cup \{\mathtt{L}, \mathtt{R}\}) \times (Q \cup \hat{Q})$.) 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 we can completely describe the actions of the Turing machine model in only a few paragraphs. A real machine would take a full book by itself. In addition, Turing machines are historically important and the work in Chapter Five needs them.

q is a state We are vague about what a 'states' is. The dictionary definition is a 'mode' of the machine. We may picture that the machine is constructed as a number of internal gears or registers. Then the states are in some sense a listing of all possible gear combinations, or all possible combinations of registers and contents.

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.

Note that this evolution is local, in that all of the action takes place within one cell of the head Adapted from (Wigderson 2017).

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."

$h(t) = -4.9t^2 + 58.4$ Some sources have the height in meters as 56 but the height from base to top is 58.45 (part of the base has sunk below ground).

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 by most researchers 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 is 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.

the mainstream community of researchers takes Church's Thesis as the basis for its work For some idea of additional views see (Zenil 2012).

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 we 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 Perlis'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)

$\mathcal{H}_4(4,3) = 4^{4^4}$ *is much greater than the number of elementary particles in the universe* The radius of the universe if 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 much less than $\mathcal{H}_3(4,4) \approx 10^{154}$ (solve $4^{256} = 10^x$ by taking the logarithm base 10 of both sides to get $x = 256 \cdot \log(4) \approx 154.13$). (Levin 2016)

Ackermann function There are many different Ackermann functions in the literature. A common one is the function of one variable $\mathcal{A}(k,k)$. See (Wikipedia contributors 2024).

a programming language having only bounded loops computes 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 $p(\hat{n} + k \cdot \hat{m}) \equiv p(\hat{n}) \bmod \hat{m}$. 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. □

this relates unbounded search to the Entscheidungsproblem It is possible that neither search will halt. It is possible that the conjecture is true but not provable from the axiom system that we are using.

Collatz conjecture See (Wikipedia contributors 2019a).

$\sin(x)$ *may be calculated via its Taylor polynomial* The Taylor series is $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \cdots$. 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/claude-shannon-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.

the von Neumann architecture Although, that architecture was based on the work of JP Eckert and J Mauchly.

problem of humans living on Mars To get there the idea was to use a rocket ship impelled by dropping 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.

Game of Life Conway explains it here: https://www.youtube.com/watch?v=E8kUJLo4ELA .

J Conway Conway was a magnetic person and extraordinarily creative. Sadly, he died in the Covid-19 pandemic. 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)

zero-player game See `https://www.youtube.com/watch?v=R9Plq-D1gEk`.

B Gosper With R Greenblatt, he started the hacker community, and is particularly well-known among Lisp-ers.

a rabbit Discovered by A Trevorrow in 1986.

anything that can be mechanically computed (Rendell 2011)

Here we will produce a simplified variant There are a number of variants in the the literature. For instance, the hyperoperation used here is not the function actually introduced by Ackermann, which has three inputs. And another student of Hilbert's, G. Sudan, produced a similar function at roughly the same time and for the same purpose, or being computable but not primitive recursive. The hyperoperation itself was defined in 1948 by R Goodstein.

it is not primitive recursive This presentation is based on that of (Hennie 1977), (Smoryński 1991), and (Robinson 1948).

This variant In addition to Péter, development of this variant also came from R Robinson.

a function is primitive recursive See the history at (Brock 2020).

LOOP (Meyer and Ritchie 1966)

the interpreter for LOOP Adapted from (Schnieder 2001)

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, $\mathbb{N} = \{0, 1, 2, \ldots\}$ Its existence is guaranteed by the Axiom of Infinity, one of the standard axioms of Mathematics, the Zermelo-Frankel axioms.

lexicographic order Sometimes called lexical order or dictionary order.

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).

Start by numbering the diagonals 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 trianglar 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). More precisely, the only real-coefficient quadratic polynomials in two variables giving a correspondence from $\mathbb{N}^2$ to $\mathbb{N}$ are $p(x,y)$ and $p(y,x)$, where $p(a,b) = a + [(a+b)(a+b+1)/2]$. 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 $g_j : N \to \hat{S}_j$ To pass from the original collection of infinitely many onto functions $g_i : \mathbb{N} \to \hat{S}_i$ 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. In this book we assume a suitable Choice axiom, and we omit further discussion of that because it would take us far afield.

doesn't matter much For more on "much" see (Rogers 1958).

adding the instruction $q_{j+k}\mathsf{BB}q_{j+k}$

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 subject The classic text (Rogers 1987) says, "It is not inaccurate to say that our theory is, in large part, a 'theory of diagonalization'."

the set of reals is not countable Here is an alternate proof of this fact, which is somewhat easier but not useful for our later development in this text. We shall show that there are more numbers in the real interval $[0\,..\,1]$ than there are natural numbers. Suppose that $f : \mathbb{N} \to [0\,..\,1]$. Around the

point $f(0)$ take an interval of width $1/10$, that is, color red all of the real numbers in the intersection $[0\,..\,1] \cap [f(0) - 1/20\,..\,f(0) + 1/20]$. Next, around the point $f(1)$ take an interval of width $(1/10)^2$, so color red the points in the intersection $[0\,..\,1] \cap [f(1) - 1/200\,..\,f(0) + 1/200]$. Continue this way: at step n take an interval of width $(1/10)^{n+1}$ around the point $f(n)$. Restated, color red all points in the intersection of $[0\,..\,1]$ with $[f(n) - (1/2)\cdot 10^{-(n+1)}\,..\,f(n) + (1/2)\cdot 10^{-(n+1)}]$. At the end, the total length of intervals colored red is at most $1/10 + (1/10)^2 + (1/10)^3 + \cdots$, which is a geometric series that adds to $1/(1-(1/10)) - 1 = 1/9$. This is less than the length of $[0\,..\,1]$ so there are points not covered by any interval, that is, points not colored red. Every $f(n)$ is colored, so there are some points in the interval that are not equal to $f(n)$ for any n.

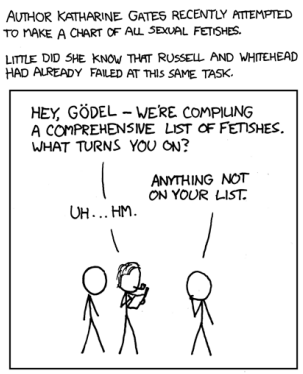
This technique is diagonalization The argument just sketched is often called Cantor's diagonal proof, although it was not Cantor's original argument for the result, and although the argument style is not due to Cantor but instead to Paul du Bois-Reymond. The fact that scientific results are often attributed 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. When suddenly the music stops 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.

That is true but the proof is beyond our scope Also beyond our scope is the argument that for any two sets, one of them has cardinality less than or equal to the other. This is equivalent to the Axiom of Choice.

consider this element of $\mathscr{P}(S)$ This is sometimes called the Russell set because of its relation to Russell's paradox. See also this XKCD.



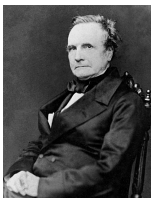
Courtesy XKCD

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 Lichterman 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 protegee 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

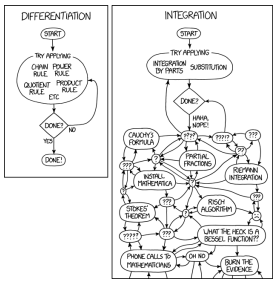
Ada Lovelace (nee Byron), 1815–1852

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/Univac.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.

the n-th decimal place of r If the number r has more than one representation then we can, say, produce the one that ends in 0's.

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's_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.

that is is parametrized by x If we start with a two-input function $\phi_{e_0}(x, y) \colon \mathbb{N}^2 \to \mathbb{N}$ then for each x the *s-m-n* effectively produces a one-input function $\phi_{s(e_0,x)} \colon \mathbb{N} \to \mathbb{N}$. That is, writing A, B, and C in place of $\mathbb{N}$, $\mathbb{N}$, and $\mathbb{N}$, it transforms the set of two-ary maps from $A \times B$ to C into a member of the set of maps from A to D, where D is the set of maps from B to C. In some contexts, such as functional programming, this is called currying.

The Halting *problem is unsolvable by any Turing machine.* The formulation of the Halting problem and the result that it is unsolvable is often distributed to Turing. It actually appeared twenty years later, in works by S Kleene and by M Davis.

Below, the first case's particular output value 42 *doesn't matter* In this book when we need a generic output value, we often use 42 because of its connection with *The Hitchhiker's Guide to the Galaxy*, (Adams 1979). It is also the uniform number of Jackie Robinson. For more on generic values see (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 also breaks into fire. 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 bursts into flame. 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)`.

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 $\mathcal{P}_0 = \{q_0\mathtt{BB}q_0, q_0\mathtt{1}1q_0\}$ 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 $\mathcal{P}_1 = \{q_0\mathtt{B}\mathtt{1}q_1, q_1\mathtt{1}\mathtt{R}q_0\}$ 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.

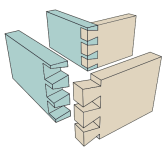
any $n > 4$ such that $2^{(2^n)} + 1$ is prime A number of this form that is prime is a Fermat prime. For $n = 0$ through 4 they are 3, 5, 17, 257, and 65537, all of which are prime. Computer searches up to 30 have not found any more.

a quadrillion, 1×10^{15}

See `https://github.com/jhg023/brocard`.

'extensional' This wording is derived from A Bauer, in `https://cs.stackexchange.com/q/2811/67754`.

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.



RE The acronym is this rather than '*CE*' because historically 'recursively enumerable' was the standard term.

"We shall not go any further into the nature of this oracle apart from saying that it cannot be a machine." (Turing 1938b)

we can't open it to see how it works Opening it would let out the magic smoke (see (Wikipedia contributors 2017f)), the stuff inside of an electronic component that makes it work. Once the smoke get out, the component no longer works.

we will instead describe it conceptually For a full treatment see (Rogers 1987).

jump further up the order This is analogous to the situation with cardinalities, where taking a power set jumps to a larger cardinality.

the notion of partial computable function seems to have an built-in defense against diagonalization (Odifreddi 1992), p 152.

the machine's name is how it behaves A person would perhaps not be astonished to find that their friend 'Boston Jones' was born in Boston, but if a friend named Lottery-winner-on-January-19-2038 Smith' did indeed win then it would certainly raise eyebrows. 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/Nominative_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 of course 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 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 then 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 with light waves travel.

working out nature's rules See `https://www.youtube.com/watch?v=o1dgrvlWML4`.

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 Michelson's statement. Often the significance of things takes time to be apparent

he 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 personification 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 2011a). 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.

development by Darwin and Wallace of the theory of differential reproduction 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 rug is less complex than the machine This is an information theoretic analog of the Second Law of Thermodynamics. E Musk has tweeted 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 traces back to 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.

We can express that in code The development of this part of the subsection comes from (Boro Sitnikovski 2024) — also where the name Boro comes from — and (Avigad 2007).

which n-state Turing Machine does the most computational work before halting R H Bruck 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 of that size.

Think of this as a competition 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.

In the 1962 paper Radó This paper (Radó 1962) is exceptionally clear and interesting.

In 2024, a team of researchers See (Brubaker 2024)

odd perfect number A number is perfect if it is the sum of its divisors. For instance, 6 is $1 + 2 + 3$. Even perfect numbers exist but we do not know if odd ones do.

machines with three or more symbols The case of machines with three states and three symbols is not known. Solving it requires solving a Collatz-like problem that currently no one can do. See `https://www.sligocki.com/2023/10/16/bb-3-3-is-hard.html`.

BB(n) *is unknowable* See (Aaronson 2012a) and the excellent summaries (Aaronson 2020) and (Aaronson 2025). 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 $t(n) = 0 + 1 + \cdots + n = n(n+1)/2$. 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)

we can 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 $\{\texttt{1000001}, \texttt{1100001}\}$ 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.

palindromes in English 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. See also `http://norvig.com/palindrome.html`.

defining Σ^* to be the set of strings of characters from that alphabet That is, we won't be careful to distinguish between the symbols of the alphabet and the single-character strings consisting of just those characters.

In practice usually a language is governed 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 the correct order as (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 The fact that in this book we stick to grammars where each rule head is a single nonterminal 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)

not completely standardized A good standard glossary is `https://web.archive.org/web/20230326040654/https://faculty.math.illinois.edu/~west/openp/gloss.html`.

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 $\Delta\colon Q \times \Sigma \to Q$ 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 $\sigma \in \{0, \ldots 9\}^*$ 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.

simple 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)

allowed users to directly dial long distance in North America See the description of the North America Numbering Plan (Wikipedia contributors 2017g).

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. That was abandoned by the early 1950's. (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 What is no-state, 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 the computer goes somewhere, the instruction pointer points to some address, but what happens is not sensible in terms of the model that you've stated.

As an alternate, 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 (...)` This operator takes a list of possibilities and evaluates to an option, if one is available, that allows the program as a whole to succeed. Here is a small example (from `https://rosettacode.org/wiki/Amb`): first `let` the values `(x (amb 1 2 3))` and `(y (amb 5 4 3))`. Then call `(amb (= (* x y) 8))`. The result is that `x` has the value 2, while y is 4. That is, `amb(1,2,3)` chooses the future in which `x` has value 2, and `amb(7,6,4,5)` chooses 4, in order to ensure that `amb(x*y = 8)` produces a success.

These operators 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($n - 1$))."

demon 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).

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 diagrams This 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.

reserve 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.

$\Delta\colon Q \times (\Sigma \cup \{\mathtt{B}, \varepsilon\}) \times (\Gamma \cup \{\perp\}) \to \mathscr{P}\big(Q \times (\Gamma \cup \{\perp\})^*\big)$ Tape outputs consist of sequences of elements of Γ that optionally end in a $\perp$. So a more precise codomain is $\mathscr{P}(Q \times S)$ for $S = \Gamma^* \cup (\Gamma^* \frown \{\perp\})$.

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 bobnice 2009).

regex golf See https://alf.nu/RegexGolf, and https://nbviewer.jupyter.org/url/norvig.com/ipython/xkcd1313.ipynb.

John Myhill Sr 1923–1987 and Anil Nerode b 1932 Photo credits Paul Halmos, Jason Koski/Cornell University

the two machines are essentially the same The two machines are said to be 'isomorphic'.

Hopcroft's algorithm See (Knuutila 2001)

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 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 2011b)

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, $28\,476 = 21 \cdot 2^{10} + 216 \cdot 2^5 + 60$). 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.

closed the seminar See also https://www.wired.com/story/mathematicians-discover-the-perfect-way-to-multiply/.

The '$f = \mathcal{O}(g)$' notation is very common See also https://whystartat.xyz/wiki/Big_O_notation.

The table below For some idea of where these functions arise in algorithm analysis see https://en.wikipedia.org/wiki/Big_O_notation#Orders_of_common_functions.

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."

input size of only 100 Basically, one hundred bits is what it takes to encode "Jim Hefferon" in UTF-8.

very, very large 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 that 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 $1+2+4+\cdots+2^{63}=2^{64}-1=18\,446\,744\,073\,709\,551\,615\approx 1.84\times 10^{19}$ grains of rice. The surface 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 (Edmunds 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.

there is an algorithm whose resource consumption is at most polynomial 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. In this book we accept Cobham's thesis because it is the most common view and because it gives technical meaning to the informal 'tractable'.

$n-1$ *assignments slower than what's on the right* 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, $\mathcal{O}(1)$.

The most commonly used model is the Turing machine This observation is from A Godson's Turing Award lecture, `https://www.youtube.com/watch?v=f2NiGO8zC1c`, where he ascribes its commonality to the fact that Turing machines come with a built-in notion of time as number of transitions and space as number of tape squares.

is that its definition ignores constant factors This discussion originated as (Stack Exchange author babou and various others 2015).

not complete until we understand what happens to inputs less than that number A great write up of the details of an algorithm for small values is the description of the sorting algorithm used by Python, in (Peters 2023).

their order of magnitude 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 predict	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=JEpsKnWZrJ8&ap`

p=desktop.

the standard gets updated Even Knuth had to update standards, from his machine model MIX to MIX.

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.

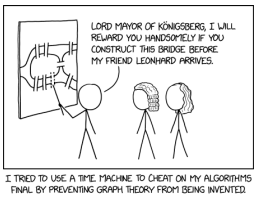
We can start with a map of the state capitals of the forty eight contiguous US states For much more, see https://www.math.uwaterloo.ca/tsp/.

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. It that is not true for this city.

the countries must be contiguous A notable example of a non-contiguous country in the wold today is that Russia is separated from Kaliningrad, the city that used to be known as Kónigsberg.

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/Swart697-707.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 2024 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.

S	A	T	O	R
A	R	E	P	O
T	E	N	E	T
O	P	E	R	A
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 become popular as the "Gem Puzzle." Noyes Chapman had applied for a patent in February, 1880. By that time the game had became 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 $\{\,\sigma \in \Sigma^* \;\big|\; \sigma \text{ is sorted}\,\}$. 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 $0, \ldots |\sigma| - 1$ 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 average 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 — that just as when a person has a hammer then everything looks like a nail, so also with a computer in hand everything looks like bits. But we

mean something less tautological: we mean that in everyday experience 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, even though at first encounter these seem to be the inherently analog.

Beethoven's 9th Symphony The story is that CD's are 72 minutes long so that they can hold this piece. See `https://www.snopes.com/fact-check/roll-over-beethoven/`.

researchers often do not mention representations This is similar to 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." It is just a question of how natural language streamlines communications

the time or space behavior We will concentrate our attention on resource bounds in the range from logarithmic to exponential, because these are the most useful for understanding problems that arise in practice.

This circuit returns 1 *if the sum of the input bits is* 0 *or* 1 This is the circuit's truth table.

$b_3b_2b_1b_0$	$b_0 \oplus b_1 = s$	$b_0 \wedge b_1 = t$	$b_2 \oplus b_3 = u$	$b_2 \wedge b_3 = v$	$s \wedge u = w$	$t \vee v = x$	$w \equiv x$
0000	0	0	0	0	0	0	1
0001	1	0	0	0	0	0	1
0010	1	0	0	0	0	0	1
0011	0	1	0	0	0	1	0
0100	0	0	1	0	0	0	1
0101	1	0	1	0	1	0	0
0110	1	0	1	0	1	0	0
0111	0	1	1	0	0	1	0
1000	0	0	1	0	0	0	1
1001	1	0	1	0	1	0	0
1010	1	0	1	0	1	0	0
1011	0	1	1	0	0	1	0
1100	0	0	0	1	0	1	0
1101	1	0	0	1	0	1	0
1110	1	0	0	1	0	1	0
1111	0	1	0	1	0	1	0

The rightmost column gives the output $f(b_0, b_1, b_2, b_3)$. It contains 1 if and only if the sum of the input bits is 0 or 1.

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`.

This is subject to 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.

adds a few 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.

functions computed by these machines One thing that we could do is to define that the nondeterministic machine computes $f : \mathbb{B}^* \to \mathbb{B}^*$ is that if on an input σ, all maximal tree paths 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 the change of even a single order of magnitude in speed can change who work gets done: we walk at 4 mph, we drive at 40 mph, and we fly at 400 mph. This relates to the picture that opens this chapter.

The verifier $\mathcal{V}$ doesn't find the ω's, it just uses them This is like a 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

require exponential time 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 stated 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 $x^n + y^n \neq z^n$ 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)`. A version that you may find fun, and worth searching for the videos, is `https://en.wikipedia.org/wiki/8_Out_of_10_Cats_Does_Countdown`.

many-one reducible The name reflects 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 to each other. 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 $f(0), f(1), \ldots$ 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

Matching *problem* This is traditionally called the Marriage problem, where the women list men that they would accept and men accept any match. But perhaps a recasting suits the times.

Asymmetric Traveling Salesman (Jonker and Volgenent 1983)

Turing reducibility is more general Wording from `https://cs.stackexchange.com/a/24590/50343`.

Stephen Cook b 1939 and Leonid Levin b 1948 Photo credits University of Toronto, Boston University

at least as hard They are "at least as hard" 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 $\mathcal{O}(n^{1\,000\,000})$ time.

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`.

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 Jeff.

A large class See (Karp 1972).

an ending point That is, as P Pudlàk observes, we treat $P \neq NP$ as an informal axiom. (Pudlàk 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 also 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. See also Aaronson's `https://scottaaronson.blog/?p=1720` and `https://www.scottaaronson.com/papers/pnp.pdf`.

A Wigderson See (Wigderson 2009).

Cook is of much 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 of this happening 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 $2^{10\,000} \approx 10^{10\,000/3.32}$ 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)

D 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)

which lies outside of our scope See (Sipser 2013).

that result's proof is beyond our scope See (Sipser 2013).

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, 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 (that is too long to fit on one line).

$$\begin{aligned}&114,381,625,757,888,867,669,235,779,976,146,612,010,218,296,721,242,\\&362,562,561,842,935,706,935,245,733,897,830,597,123,563,958,705,\\&058,989,075,147,599,290,026,879,543,541\end{aligned}$$

In 1994, a team of about 600 volunteers announced that they had factored n.

$$\begin{aligned}p =&3,490,529,510,847,650,949,147,849,619,903,898,133,417,764,\\&638,493,387,843,990,820,577\end{aligned}$$

and

$$\begin{aligned}q =&\,32,769,132,993,266,709,549,961,988,190,834,461,413,177,642,967,\\&992,942,539,798,288,533\end{aligned}$$

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 $n > 2$.

these are 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 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).

The Free mathematics system Sage includes one See also `https://www.youtube.com/watch?v=q8nQTNvCrjE` about the Concorde TSP solver.

the Soduku *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 looking for it, see the Soduku Wikipedia page.

Here we support that view This material developed from a post by Ege Erdil, `https://www.lesswrong.com/posts/8Af3X8b7f5piqtqZx/computability-and-complexity`.

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, "eenie, meenie, meinie, mo" (Stack Exchange author Jouni Sirén 2016); see also `https://www.youtube.com/watch?v=juXwu0Nqc3I`

reversal σ^R *of a string* The most practical current notion of a character 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`.)

a function isn't a 'rule' For many instances of sources stating the opposite, including many that are reputable, enter a phrase such as 'a function is a rule' into a search engine.

Credits

Prologue

I.1.12 SE user Shuzheng, https://cs.stackexchange.com/q/45589/50343

I.1.13 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

I.2.15 Reddit user Valuable-Glass1106, https://old.reddit.com/r/computerscience/comments/1j3z4jy/how_could_a_multi_tape_turing_machine_be/

I.4.30 SE user Ted, https://math.stackexchange.com/a/75300/12012

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.3.18 User scherk at pbworks.com.

II.3.20 Math StackExchange user Robert Z https://math.stackexchange.com/a/1896328/12012

II.3.28 Michael J Neely

II.3.31 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.5.34 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 used in section. The answer here is not from Andrej Bauer.

II.6.31 SE user Rajesh R

II.8.17 https://mathoverflow.net/questions/33046/arent-oracle-machines-unsound-concepts, (The question there as elaborated is different than this adaptation's.)

II.8.19 SE user Karolis Juodelė

II.8.22 SE user Noah Schweber

II.8.25 http://people.cs.aau.dk/~srba/courses/tutorials-CC-10/t5-sol.pdf

II.9.10 (Rogers 1987), p 214.

II.9.12 (Rogers 1987), p 214.

II.9.17 (Rogers 1987), p 214.

II.A.1 https://www.ias.edu/ideas/2016/pires-hilbert-hotel

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_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/~zarembe/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.3.44 https://cs.stackexchange.com/a/30726
IV.4.7 https://cs.stackexchange.com/q/155353/50343
IV.4.27 SE user jmite, https://cs.stackexchange.com/a/67317/50343.
IV.4.29 (Rich 2008)
IV.4.30 (Rich 2008), https://math.stackexchange.com/a/1102627
IV.5.20 SE user David Richerby, https://cs.stackexchange.com/a/97885/67754
IV.5.24 (Rich 2008)
IV.5.30 SE author Yuval Filmus, https://cs.stackexchange.com/a/41445/50343
IV.5.31 SE user Brian M Scott, https://math.stackexchange.com/a/1508488
IV.C.15 https://www.eecs.wsu.edu/~cook/tcs/l10.html

Complexity

V. Some of the discussion is from https://softwareengineering.stackexchange.com/a/20833.
V. This discussion originated as (Stack Exchange author templatetypedef 2013).
V.1.57 Stack Exchange user Daniel Fischer, https://math.stackexchange.com/a/674039, and Stack Exchange user anon, https://math.stackexchange.com/a/61741
V.1.64 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.2.43 Private communication from P Rombach.
V.2.59 https://stackoverflow.com/q/55051253
V.2.61 Jan Verschelde, http://homepages.math.uic.edu/~jan/mcs401/partitioning.pdf
V.3.10 A.A. at https://rjlipton.wordpress.com/2010/11/07/what-is-a-complexity-class/#comment-8872
V.3.15 SE user Yuval Filmus https://cs.stackexchange.com/a/79729
V.4.17 https://cs.stackexchange.com/q/57518
V.5.16 SE user Simone, https://math.stackexchange.com/q/49509
V.5.22 Paul Black, https://xlinux.nist.gov/dads/HTML/nondetermAlgo.html
V.6.28 SE user JesusIsLord at https://cstheory.stackexchange.com/a/47031/4731
V.6.30 SE user user326210, https://math.stackexchange.com/a/2564255

V.6.32 Thanks to P Rombach of the University of Vermont for help with this.

V. By Psyon (Own work) CC BY-SA 3.0 https://commons.wikimedia.org/wiki/File:Jigsaw_Puzzle.svg

V.7.16 William Gasarch, https://www.cs.umd.edu/~gasarch/COURSES/452/F14/poly.pdf

V.7.20 http://www.cs.princeton.edu/courses/archive/fall02/cos126/exercises/np.html

V.7.21 http://www.cs.princeton.edu/courses/archive/fall02/cos126/exercises/np.html

V.7.22 Kevin Wayne. http://www.cs.princeton.edu/courses/archive/fall02/cos126/exercises/np-sol.html

V.7.29 Y Lyuu, https://www.csie.ntu.edu.tw/~lyuu/complexity/2016/20161129s.pdf

V.7.33 SE user Yuval Filmus https://cs.stackexchange.com/a/132902/50343

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, 2011a). *Rosser's Theorem via Turing machines*. URL: http://www.scottaaronson.com/blog/?p=710 (visited on 12/31/2023).

— (Aug. 14, 2011b). *Why Philosophers Should Care About Computational Complexity*. URL: https://arxiv.org/abs/1108.1791.

— (May 3, 2012a). *The 8oooth 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).

— (July 27, 2020). *The Busy Beaver Frontier*. URL: https://www.scottaaronson.com/papers/bb.pdf (visited on 07/02/2024).

— (Apr. 23, 2025). *How Much Math Is Knowable?* URL: https://www.youtube.com/watch?v=VplMHWSZf5c&t=501s (visited on 04/26/2025).

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.

Avigad, J. (Jan. 9, 2007). "Computability and Incompleteness Lecture Notes". In: URL: https://www.andrew.cmu.edu/user/avigad/Teaching/candi_notes.pdf (visited on 09/26/2024).

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 Compututing* 26.5, pp. 1411–1473.

Bigham, D. S. (Aug. 19, 2014). *How Many Vowels Are There in English? (Hint: It's More Than AEIOUY.)*. 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/.

Boro Sitnikovski (2024). *Deriving a Quine in a Lisp*. URL: https://bor0.wordpress.com/2020/04/24/deriving-a-quine-in-a-lisp/ (visited on 09/26/2024).

Brady, A. H. (Apr. 1983). "The Determination of the Value of Rado's Noncomputable Function $\Sigma(k)$ for Four-State Turing Machines". In: *Mathematics of Computation* 40.162, pp. 647–665.

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-ritchies-lost-dissertation/.

Brower, K. (1983). *The Starship and the Canoe*. Harper Perennial; Reprint edition. ISBN: 978-0060910303.

Brubaker, B. (Apr. 2, 2024). "With Fifth Busy Beaver, Researchers Approach Computation's Limits". In: *Quanta*. URL: https://www.quantamagazine.org/amateur-mathematicians-find-fifth-busy-

beaver-turing-machine-20240702/ (visited on 04/02/2024).
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/pvsnp.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 paroit 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.
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.
Indian Institute of Science and Indian Institutes of Technologies (2021). *Graduate Aptitude Test in Engineering*.
— (2022). *Graduate Aptitude Test in Engineering*.
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.
Knuth, D. E. (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 arithmetico-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).
Pudlàk, 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). *A Turing Machine in Coway's Game of Life, extendable to a Universal Turing Machine*. 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.
Schnieder, H.-J. (2001). "Computability in an Introductory Course on Programming". In: *Bulletin of the European Association for Theoretical Computer Science, EATCS* 73, pp. 153–164.
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).
Sipser, M. (2013). *Introduction to the Theory of Compuation*. 3rd ed. Cengage. ISBN: 978-1-133-18779-0.
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.
— (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%E2%80%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%E2%80%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: %5Curl%7Bhttps://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%E2%80%93Maturin_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%E2%80%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.
— (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: %5Curl%7Bhttps://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 2025-Apr-18]. URL: https://en.wikipedia.org/wiki/Foobar.
— (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.
— (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.
— (2024). *Hyperoperation — Wikipedia, The Free Encyclopedia*. [Online; accessed 9-August-2024]. URL: https://en.wikipedia.org/w/index.php?title=Hyperoperation&oldid=1226527372.
YouTube channel Joint Mathematics Meetings (May 5, 2018). *William Cook: "Information, Computation, Optimization . . ."*. URL: https://www.youtube.com/watch?v=q8nQTNvCrjE (visited on 07/02/2024).
YouTube user navyreviewer (2010). *Mechanical computer part 1*. URL: https://www.youtube.com/watch?v=mpkTHyfr0pM (visited on 08/09/2015).

Zenil, H., ed. (2012). *A Computable Universe, Understanding and Exploring Nature as Computation*. World Scientific. ISBN: 978-9814374293.

Index