

# Theory of Computation

Making Connections



Jim Hefferon

<https://hefferon.net/computation>

Notation	Description
$\mathcal{P}(S)$	power set, collection of all subsets of $S$
$S^c$	complement of the set $S$
$\mathbb{1}_S$	characteristic function of the set $S$
$\langle a_0, a_1, \dots \rangle$	sequence
$\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$	natural numbers $\{0, 1, \dots\}$ , integers, rationals, reals
a, b, ...	characters
$\Sigma$	alphabet, set of characters
$\mathbb{B}$	alphabet of bits, $\mathbb{B} = \{0, 1\}$
$\sigma, \tau$	strings (any lower-case greek letter)
$\varepsilon$	empty string
$\Sigma^*$	set of all strings over the alphabet
$\mathcal{L}$	language, subset of $\Sigma^*$
$\mathcal{P}$	Turing machine, either deterministic or nondeterministic
$\phi$	effective function, function computed by a Turing machine
$\phi(x)\downarrow, \phi(x)\uparrow$	function converges on that input, or diverges
$U\mathcal{P}$	universal Turing machine
$\mathcal{G}$	graph
$\mathcal{M}$	Finite State machine, either deterministic or nondeterministic
$\mathbf{P}$	complexity class of deterministic polynomial time problems
$\mathbf{NP}$	complexity class of nondeterministic polynomial time problems
$\mathcal{V}$	verifier for $\mathbf{NP}$
$SAT$	language for the Satisfiability problem

### Greek letters with pronunciation

Character	Name	Character	Name
$\alpha$	alpha AL-fuh	$\nu$	nu NEW
$\beta$	beta BAY-tuh	$\xi, \Xi$	xi KSIGH
$\gamma, \Gamma$	gamma GAM-muh	$\omicron$	omicron OM-uh-CRON
$\delta, \Delta$	delta DEL-tuh	$\pi, \Pi$	pi PIE
$\varepsilon$	epsilon EP-suh-lon	$\rho$	rho ROW
$\zeta$	zeta ZAY-tuh	$\sigma, \Sigma$	sigma SIG-muh
$\eta$	eta AY-tuh	$\tau$	tau TOW, as in cow
$\theta, \Theta$	theta THAY-tuh	$\upsilon, \Upsilon$	upsilon OOP-suh-LON
$\iota$	iota eye-OH-tuh	$\phi, \Phi$	phi FEE, or FI as in high
$\kappa$	kappa KAP-uh	$\chi$	chi KI, as in high
$\lambda, \Lambda$	lambda LAM-duh	$\psi, \Psi$	psi SIGH, or PSIGH
$\mu$	mu MEW	$\omega, \Omega$	omega oh-MAY-guh

Capitals letters shown are the ones that differ from Roman capitals.

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

This PDF was compiled 2022-Jul-28.

## Preface

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

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

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

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

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

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, automata, and complexity. The audience is undergraduate majors in Computer Science, Mathematics, and nearby areas.

The prerequisite, besides an introductory course in programming, is Discrete

Mathematics. We rely on propositional logic, proof methods including induction, graphs, basic number theory, sets, functions, and relations (appendices establish notation and terminology for strings, functions, and propositional logic). There are reviews for graphs and Big- $\mathcal{O}$  (this material may be less familiar to non-Computer Science students). The Big- $\mathcal{O}$  section requires derivatives.

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

The first example comes right at the start, where we begin with Turing machines. Initiating the course by covering Finite State machines may be mathematically slicker but to a fresh learner, instead starting by asking what can be computed at all is more natural. We use careful motivation with an extensive discussion of Church's Thesis, relying on the intuition that students have gained in their programming experience. Besides providing the best start, this allows us to give algorithms in outline rather than as programs for our computation model, which communicates the ideas better.

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

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

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

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

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

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

**Schedule** Chapter I defines models of computation, Chapter II covers unsolvability, Chapter III does languages and graphs, Chapter IV is automata, and Chapter V is computational complexity. I assign the readings as homework and quiz on them.

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

**License** This book is Free. You can use it without cost and you can redistribute it, for example by posting it on a Learning Management System’s directory for your course. You can also get the  $\text{\LaTeX}$  source from a repository and modify it to suit your class; see <https://hefferon.net/computation>.

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

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

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

My students have made the book better in many ways. I greatly appreciate all of their contributions.

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

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

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

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

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

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

*Teach concepts, not tricks.*

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

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

– E Bander, PERSUASIONS, 2017

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

– A Downey, PROGRAMMING AS A WAY OF THINKING

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

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

Jim Hefferon  
Saint Michael's College  
Colchester, VT USA  
[hefferon.net/computation](http://hefferon.net/computation)  
Version 1.00, 2022-Jul-19.

# Contents

<b>I    Mechanical Computation</b>	<b>3</b>
1 Turing machines . . . . .	3
1 Definition . . . . .	4
2 Effective functions . . . . .	9
2 Church's Thesis . . . . .	14
1 History . . . . .	14
2 Evidence . . . . .	15
3 What it does not say . . . . .	17
4 An empirical question? . . . . .	18
5 Using Church's Thesis . . . . .	18
3 Recursion . . . . .	21
1 Primitive recursion . . . . .	22
4 General recursion . . . . .	31
1 Ackermann functions . . . . .	31
2 $\mu$ recursion . . . . .	33
A Turing machine simulator . . . . .	37
B Hardware . . . . .	42
C Game of Life . . . . .	46
D Ackermann's function is not primitive recursive . . . . .	49
E LOOP programs . . . . .	53
<b>II    Background</b>	<b>61</b>
1 Infinity . . . . .	61
1 Cardinality . . . . .	61
2 Cantor's correspondence . . . . .	68
3 Diagonalization . . . . .	76
1 Diagonalization . . . . .	76
4 Universality . . . . .	83
1 Universal Turing machine . . . . .	84
2 Uniformity . . . . .	86
3 Parametrization . . . . .	87
5 The Halting problem . . . . .	92
1 Definition . . . . .	92
2 Discussion . . . . .	94
3 Significance . . . . .	95
4 General unsolvability . . . . .	96
6 Rice's Theorem . . . . .	103
7 Computably enumerable sets . . . . .	108
8 Oracles . . . . .	113

9	Fixed point theorem . . . . .	121
1	When diagonalization fails . . . . .	122
2	Discussion . . . . .	125
A	Hilbert's Hotel . . . . .	127
B	The Halting problem in intellectual culture . . . . .	128
C	Self Reproduction . . . . .	131
D	Busy Beaver . . . . .	136
E	Cantor in Code . . . . .	139
<b>III</b>	<b>Languages</b>	<b>147</b>
1	Languages . . . . .	147
2	Grammars . . . . .	152
1	Definition . . . . .	152
3	Graphs . . . . .	163
1	Definition . . . . .	163
2	Traversal . . . . .	164
3	Graph representation . . . . .	165
4	Colors . . . . .	166
5	Graph isomorphism . . . . .	167
A	BNF . . . . .	173
<b>IV</b>	<b>Automata</b>	<b>180</b>
1	Finite State machines . . . . .	180
1	Definition . . . . .	180
2	Nondeterminism . . . . .	191
1	Motivation . . . . .	191
2	Definition . . . . .	193
3	$\epsilon$ transitions . . . . .	196
4	Equivalence of the machine types . . . . .	199
3	Regular expressions . . . . .	205
1	Definition . . . . .	205
2	Kleene's Theorem . . . . .	207
4	Regular languages . . . . .	214
1	Definition . . . . .	215
2	Closure properties . . . . .	215
5	Languages that are not regular . . . . .	220
6	Minimization . . . . .	226
7	Pushdown machines . . . . .	237
1	Definition . . . . .	237
2	Nondeterministic Pushdown machines . . . . .	241
3	Context free languages . . . . .	244
A	Regular expressions in the wild . . . . .	245
B	The Myhill-Nerode Theorem . . . . .	253

<b>V Computational Complexity</b>	<b>261</b>
1 Big $\mathcal{O}$ . . . . .	261
1 Motivation . . . . .	262
2 Definition . . . . .	265
3 Tractable and intractable . . . . .	269
4 Discussion . . . . .	271
2 A problem miscellany . . . . .	277
1 Problems that come with stories . . . . .	277
2 More problems, omitting the stories . . . . .	281
3 Problems, algorithms, and programs . . . . .	292
1 Problem types . . . . .	293
2 Statements and representations . . . . .	295
4 P . . . . .	301
1 Definition . . . . .	301
2 Effect of the model of computation . . . . .	303
3 Naturalness . . . . .	303
5 NP . . . . .	307
1 Nondeterministic Turing machines . . . . .	308
2 Speed . . . . .	310
3 Definition . . . . .	310
6 Reductions between problems . . . . .	318
7 NP completeness . . . . .	330
1 P = NP? . . . . .	336
2 Discussion . . . . .	339
8 Other classes . . . . .	345
1 EXP . . . . .	345
2 Time Complexity . . . . .	346
3 Space Complexity . . . . .	347
4 The Zoo . . . . .	348
A RSA Encryption . . . . .	350
B Good-enoughness . . . . .	356
C SAT solvers . . . . .	358
<b>Appendix</b>	<b>367</b>
A Strings . . . . .	368
B Functions . . . . .	370
C Propositional logic . . . . .	374
<b>Notes</b>	<b>379</b>
<b>Bibliography</b>	<b>415</b>



Part One

## Classical Computability



# CHAPTER

## I Mechanical Computation

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

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

### SECTION

#### I.1 Turing machines

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

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

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

First, it (or he or she) has a memory facility, such as the clerk's paper, to store and retrieve information.

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 does not use continuous methods or analog devices. Thus there is no question about the



Alan Turing 1912–  
1954

IMAGE: copyright Kevin Twomey, [kevintwomey.com/lowtech.html](http://kevintwomey.com/lowtech.html) <sup>†</sup>German for “decision problem.”

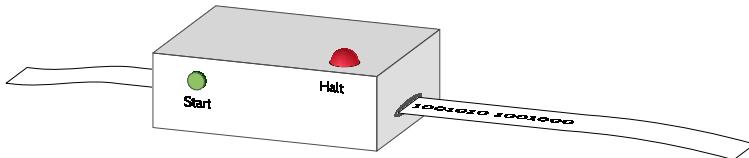
<sup>‡</sup>When it finished computing it might turn on a light for ‘true’, or print the symbol 1. <sup>§</sup>He was a serious candidate for the 1948 British Olympic marathon team.

precision of operations as there might be when, say, reading results off of a slide rule or an instrument dial. Instead, the agent works in a discrete fashion, step-by-step. For instance, if needed they could pause between steps, note where they are (“about to carry a 1”), and later pick up again. We say that at each moment the clerk is in one of a finite set of possible **states**, which we denote  $q_0, q_1, \dots$ .

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

The final question Turing faced is: how smart is the computing agent? For instance, can it multiply? We don’t need to include a special facility for multiplication because we can in principle multiply via repeated addition. We don’t even need addition because we can repeat the successor operation, the add-one operation. In this way Turing pared the computing agent down until it is quite basic, quite easy to understand, until the operations are so elementary that we cannot easily imagine them further divided, while still keeping the 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. For instance, to multiply, the computing agent can get the two input multiplicands from the tape (the drawing shows 74 and 72, represented in binary and separated by a blank), can use the tape for scratch work,

---

<sup>†</sup>It is true that a physical computer such as a cell phone has memory space that is bounded, putting aside storing things in the Cloud. However, that space is extremely large. In this Part, when working with the model devices we find that imposing a bound on memory is irrelevant, or even a hindrance. <sup>‡</sup>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 physically-realizable computations, and so the development here instead says that the resources have no finite upper bound. But really, it doesn’t matter. If we show that something cannot be computed when there are no bounds then we have shown that it cannot be computed on any real-world device.

and can write the output to the tape.

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

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

Consequently, machine steps involve four pieces of information. Call the present state  $q_p$  and the next state  $q_n$ . The other two,  $T_p$  and  $T_n$ , denote the tape symbol that the read/write head is presently pointing to and what happens next with the tape. The things that can happen next with the tape are: moving the tape head left or right without writing, which we denote with  $T_n = L$  or  $T_n = R$ ,<sup>†</sup> or writing a symbol to the tape without moving the head, which we denote with that symbol, so that  $T_n = 1$  means to write a 1 to the tape. As to the set of characters that go on the tape, we will choose whatever is convenient. However, except for finitely many places, every tape is filled with blanks and so that must be one of the symbols. (We denote blank with B when an empty space could cause confusion.)

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

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

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

Below we’ve represented this machine’s initial configuration. This shows a stretch of the tape along with the machine’s state and the position of its read-write head.

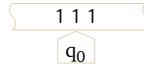
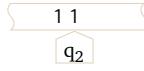
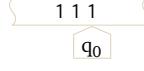
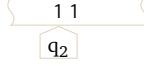
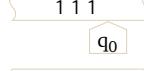
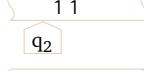
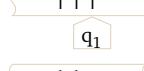
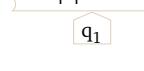


We take the convention that when we press Start the machine is in state  $q_0$ . The

---

<sup>†</sup>Whether we move the tape or the head doesn’t matter, what matters is their relative motion. Thus  $T_n = L$  means that one or the other moves such that the head now points to the location one place to the left. In our drawings, we hold the tape steady and move the head because then comparing graphics step by step is easier.

picture above shows the machine reading 1, so instruction  $q_01Rq_0$  applies. Thus the first step is that the machine moves its tape head right and stays in state  $q_0$ . Below, the first line shows this and later lines show the machine's configuration after later steps. In summary, the computation slides to the right, blanks out the final 1, and slides back to the start.

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

Finally, because there is no state  $q_3$ , no instruction applies and the machine halts.

We can think of this machine as computing the predecessor function

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

because if we initialize the tape so that it contains only a string of  $n$ -many consecutive 1's and the machine's head points to the first, then at the end the tape will have  $n - 1$ -many 1's (except for  $n = 0$ , where the tape will end with no 1's).

## 1.2 EXAMPLE This machine adds two natural numbers.

$$\mathcal{P}_{\text{add}} = \{ q_0Bq_1, q_01Rq_0, q_1B1q_1, q_111q_2, q_2BBq_3, q_21Lq_2, q_3BRq_3, q_31Bq_4, q_4BRq_5, q_411q_5 \}$$

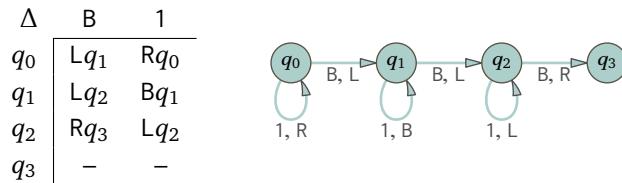
The input numbers are represented by strings of 1's that are separated with a blank. The read/write head starts under the first symbol in the first number. This shows the machine ready to compute  $2 + 3$ .



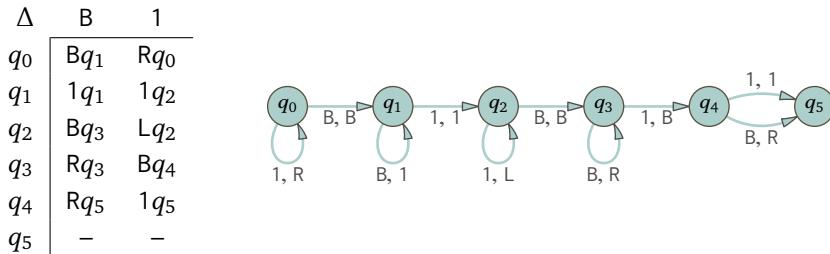
The machine scans right, looking for the blank separator. It changes that to a 1, then scans left until it finds the start. Finally, it trims off a 1 and halts with the read/write head pointing to the start of the string. Here are the steps.

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

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**.



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



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

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

- 1.3 EXAMPLE The machine  $\mathcal{P}_{\text{inf loop}} = \{q_0 B B q_0, q_0 1 1 q_0\}$  never halts, regardless of the input.



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

It is high time for definitions. We take a **symbol** to be something that the device can write and read, for storage and retrieval.<sup>†</sup>

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

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

We denote Turing machines with  $\mathcal{P}$  because, although they are hardware, the things from our everyday experience that they are most like are programs.

Of course, the point of these machines is what they do. A Turing machine is a blueprint for a computation—it is like a program—and so to finish the formalization started by the definition we give a complete description of a machine's action.

We saw in tracing through Example 1.1 and Example 1.2 that a machine acts by directing the transition from one configuration to the next. A **configuration** of a Turing machine is a four-tuple  $\mathcal{C} = \langle q, s, \tau_L, \tau_R \rangle$ , where  $q$  is a state, a member of  $Q$ ,  $s$  is a character from the tape alphabet  $\Sigma$ , and  $\tau_L$  and  $\tau_R$  are strings of elements from the tape alphabet, including possibly the empty string  $\varepsilon$ . 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, line 2 of the trace table of Example 1.2, where the state is  $q = q_0$ , the character under the head  $s$  is the blank, and to the left of the head is  $\tau_L = 11$  while to the right is  $\tau_R = 111$ , graphically represents the configuration  $\langle q, s, \tau_L, \tau_R \rangle$ . That is, a configuration is a snapshot, an instant in a computation.

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

Suppose that at step  $t$  a machine  $\mathcal{P}$  is in configuration  $\mathcal{C}(t) = \langle q, s, \tau_L, \tau_R \rangle$ . To make the next transition, find an instruction  $q_p T_p T_n q_n \in \mathcal{P}$  with  $q_p = q$  and  $T_p = s$ . If there is no such instruction then at step  $t + 1$  the machine  $\mathcal{P}$  **halts**.

Otherwise there will be only one such instruction, by determinism. There

---

<sup>†</sup> How the device does this depends on its construction details. For instance, to have a machine with two symbols, blank and 1, we can either read and write marks on a paper tape, or align magnetic particles on a plastic tape, or bits on a chip, or we can push LEGO bricks to the left or right side of a slot. Discreteness ensures that the machine can cleanly distinguish between the symbols, in contrast with the trouble that can happen, for instance, in reading a continuous instrument dial near a boundary.

are three possibilities. (1) If  $T_n$  is a symbol in the tape alphabet set  $\Sigma$  then the machine writes that symbol to the tape, so that the next configuration is  $C(t+1) = \langle q_n, T_n, \tau_L, \tau_R \rangle$ . (2) If  $T_n = L$  then the machine moves the tape head to the left. That is, the next configuration is  $C(t+1) = \langle q_n, \hat{s}, \hat{\tau}_L, \hat{\tau}_R \rangle$ , where  $\hat{s}$  is the rightmost character of the string  $\tau_L$  (if  $\tau_L = \epsilon$  then  $\hat{s}$  is the blank character), where  $\hat{\tau}_L$  is  $\tau_L$  with its rightmost character omitted (if  $\tau_L = \epsilon$  then  $\hat{\tau}_L = \epsilon$  also), and where  $\hat{\tau}_R$  is the concatenation of  $\langle s \rangle$  and  $\tau_R$ . (3) If  $T_n = R$  then the machine moves the tape head to the right. This is like (2) so we omit the details.

If two configurations are related by being a step apart then we write  $C(i) \vdash C(i+1)$ .<sup>†</sup> A **computation** is a sequence  $C(0) \vdash C(1) \vdash C(2) \vdash \dots$ . We abbreviate such a sequence with  $\vdash^*$ .<sup>‡</sup> If the computation halts then the sequence has a final configuration  $C(h)$ , so we may write a halting computation as  $C(0) \vdash^* C(h)$ .

- 1.5 EXAMPLE In Example 1.1, the pictures that trace the machine's execution show the successive configurations. So the computation is this.

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

That description of the action of a Turing machine emphasizes that it is a **state machine**—Turing machine computation is about the transitions, the discrete steps taking one configuration to another.

**Effective functions** In the chapter's opening we expressed interest not so much in the machines as in the things that they compute. We finish this section by defining the set of functions that are mechanically computable.

A function is an association of inputs with outputs. For Turing machines, the natural computed function is the association of the string on the tape when the machine starts with the one on the tape when it halts, if it ever does halt.

- 1.6 DEFINITION Let  $\mathcal{P}$  be a Turing machine with tape alphabet  $\Sigma$ . The function  $\phi_{\mathcal{P}}: \Sigma^* \rightarrow \Sigma^*$  computed by  $\mathcal{P}$  is: for input  $\sigma \in \Sigma^*$ , the output  $\phi_{\mathcal{P}}(\sigma)$  is the string that results from placing  $\sigma$  on an otherwise blank tape, pointing the read/write head of  $\mathcal{P}$  to  $\sigma$ 's left-most symbol, and running the machine until it halts.<sup>§</sup>

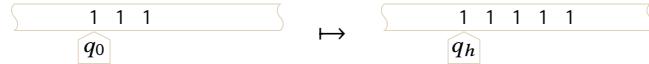
Just specifying the input string is not enough, since the initial position of the head can change the computation. So we specify that when the machine starts, its head is under the string's first character. In this book we like to write machines so that

---

<sup>†</sup>The ‘ $\vdash$ ’ is a turnstile symbol. In this context, read it aloud as “yields.”   <sup>‡</sup>Read this aloud as “yields eventually.”   <sup>§</sup>The input is unambiguous because it is what's on the tape when we start the machine. But the output is more of a problem; in the addition machine input, the separator blank is significant and if we allow significant blanks in the output, then which of the infinitely many blanks matter? There are a number of things that we can do. One is that instead of using blanks as separators, etc., we can enlarge the alphabet and use the new characters for that purpose. Another is that for any string  $\sigma \in \Sigma^*$ , where its length is  $n$  we say that its self-delimiting code string is  $\bar{\sigma} = 1^n \wedge B \wedge \sigma$  (or use any nonblank character in place of 1). Outputting self-delimiting code strings eliminates any ambiguity without requiring a change in alphabet.

they also finish with the head under the first character of the output, which isn't strictly necessary, although it makes composition easier.

This illustrates the computation of a value of a function,  $\phi(111) = 11111$  (when there is only one machine under discussion then we write  $\phi$  instead of  $\phi_P$ ).



The definition says “until it halts.” What if the machine doesn’t halt?

- 1.7 **DEFINITION** If for a Turing machine the value of a computation is not defined on some input  $\sigma \in \Sigma^*$  then we say that the function computed by the machine **diverges** on that input, written  $\phi(\sigma)\uparrow$  (or  $\phi(\sigma) = \perp$ ). Otherwise we say that it **converges**,  $\phi(\sigma)\downarrow$ . If  $\phi$  converges for each input in  $\Sigma^*$  then it is a **total function**.<sup>†</sup>

Very important: note the difference between the machine  $P$  and the function computed by that machine,  $\phi_P$ . For example, the machine  $P_{\text{pred}}$  is a set of four-tuples but the predecessor function is a set of input-output pairs, which we might describe with  $x \mapsto \text{pred}(x)$ . Another example of the difference is that machines halt or fail to halt, while functions converge or diverge.

To claim, for instance, that a Turing machine computes the successor function, or addition of two numbers, we must interpret the input and output strings—we must impose some coding on the strings. Above, for predecessor, we let  $\Sigma = \{1, B\}$  and took the strings to describe natural numbers in unary. (Another encoding common in the literature represents 0 with one stroke, represents 1 with two strokes, etc.) For addition we did the same, using a blank to separate the two. As to non-numbers, in this book we sometimes manipulate strings directly but for more complex things, such as directed graphs, we will also use an encoding. (We could worry that our interpretation of an encoding might be so involved that, as with a horoscope, the computation is happening in the interpretation, not in the Turing machine. But we will stick to cases such as the unary representation of numbers, where it clearly is not an issue.)

When we describe the function computed by a machine, we typically omit the part about interpreting the strings. We say, “this shows  $\phi(3) = 5$ ” rather than, “this shows  $\phi$  taking a string representing 3 in unary to a string representing 5.” The details of the representation are usually not of interest (unless we are counting the time or space that they consume; more on that in the final chapter).

- 1.8 **REMARK** Early researchers, working before actual machines were widely available, needed airtight arguments that there is a mechanical computation of, say, the function that takes in a number  $n$  and returns the  $n$ -th prime. So they did the

<sup>†</sup>Traditionally, a function comes with a domain, the set of inputs on which it is defined. For computable functions we instead take that the set of inputs is  $\Sigma^*$ , understanding that the underlying machine may halt on only some of them. So we introduce the notion of a **partial function**, where some  $W \subseteq \Sigma^*$  is the set of input strings  $\sigma$  such that  $\phi(\sigma)\downarrow$ . If  $W = \Sigma^*$  then  $\phi$  is total. Strictly speaking, every computable  $\phi$  is partial but saying that it is partial usually connotes that it is not total, that  $W \subsetneq \Sigma^*$ .

details, demonstrating that their definitions and arguments accorded with their intuition by building up a large body of highly technical evidence. For us, everyday experience says that machines can use their alphabet, usually binary, to reasonably represent anything that our intuition says is computable. So the development here leaves aside some of the depth of detail in other presentations, simply to get sooner to more material. The next section says more.

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

We close with a summary. We have characterized mechanical computation. We view it as a process whereby a physical system evolves through a sequence of discrete steps that are local, meaning that all the action takes place within one cell of the head. This gives us a precise definition of which of the functions can be mechanically computed. In the next subsection we will discuss this characterization, including the evidence that leads to its widespread acceptance.

## I.1 Exercises

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

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

---

<sup>†</sup>The term ‘recursive’ used to be universal but is now old-fashioned.

(c) For a machine to reach state  $q_{50}$  it must run for at least fifty one steps.

- 1.15 We often have some states that are **halting states**, where we send the machine solely to make it halt. In this case the others are **working states**. For instance, Example 1.1 uses  $q_3$  as a halting state and its working states are  $q_0, q_1$ , and  $q_2$ . Name Example 1.2's halting and working states.
- ✓ 1.16 Trace the execution of  $\mathcal{P}_{\text{inf loop}}$  for ten steps, from a blank tape. Show the sequence of tapes.
- 1.17 Trace the execution on each input of this Turing machine with alphabet  $\Sigma = \{\text{B}, 0, 1\}$  for ten steps, or fewer if it halts.
- $$\{q_0\text{BB}q_4, q_00\text{R}q_0, q_01\text{R}q_1, q_1\text{BB}q_4, q_10\text{R}q_2, q_11\text{R}q_0, q_2\text{BB}q_4, q_20\text{R}q_0, q_21\text{R}q_3\}$$
- (A) 11 (B) 1011 (C) 110 (D) 1101 (E)  $\epsilon$
- ✓ 1.18 Give the transition table for the machine in the prior exercise.
- ✓ 1.19 Write a Turing machine that, if it is started with the tape blank except for a sequence of 1's, will replace those with a blank and then halt.
- ✓ 1.20 Produce Turing machines to perform these Boolean operations, using  $\Sigma = \{\text{B}, 0, 1\}$ . (A) Take the 'not' of a bit  $b \in \Sigma_0 = \Sigma - \{\text{B}\}$ . That is, convert the input  $b = 0$  into the output 1, and convert 1 into 0. (B) Take as input two characters drawn from  $\Sigma_0$  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'.
- 1.21 Give a Turing machine that takes as input a bit string, using the alphabet  $\{\text{B}, 0, 1\}$ , and adds 01 at the back.
- 1.22 Produce a Turing machine that computes the constant function  $\phi(x) = 3$ . It inputs a number written in unary, so that  $n$  is represented as  $n$ -many 1's, and outputs the number 3 in unary.
- ✓ 1.23 Produce a Turing machine that computes the successor function, that takes as input a number  $n$  and gives as output the number  $n + 1$  (in unary).
- ✓ 1.24 Produce a doubler, a Turing machine that computes  $f(x) = 2x$ .
- (A) Assume that the input and output is in unary. Hint: you can erase the first 1, move to the end of the 1's, past a blank, and put down two 1's. Then move left until you are at the start of the first sequence of 1's. Repeat.
- (B) Instead assume that the alphabet is  $\Sigma = \{\text{B}, 0, 1\}$  and the input is represented in binary.
- ✓ 1.25 Produce a Turing machine that takes as input a number  $n$  written in unary, represented as  $n$ -many 1's, and if  $n$  is odd then it gives as output the number 1 in unary, with the head under that 1, while if  $n$  is even it gives the number 0 (which in a unary representation means the tape is blank).
- 1.26 Write a machine  $\mathcal{P}$  with tape alphabet  $\Sigma$  that, in addition to blank B and stroke 1, also contains the comma ',' character. Where  $\Sigma_0 = \Sigma - \{\text{B}\}$ , if we interpret

the input  $\sigma \in \Sigma_0$  as a comma-separated list of natural numbers represented in unary, then this machine should return the sum, also in unary. For instance,  $\phi_P(1111,,111,1) = 11111111$ .

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

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

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

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

*Definition 1.9 says that a set is computable if there is a Turing machine that acts as its characteristic function. That is, the machine is started with the tape blank except for the input string  $\sigma$ , 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  $\sigma$  is in the set) or 0 (meaning it is not). For the next three exercises, produce a Turing machine that acts as the characteristic function of the set.*

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

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

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

1.34 Definition 1.9 talks about a relation being computable. Consider the 'less

than or equal' relation between two natural numbers, i.e., 3 is less than or equal to 5, but 2 is not less than or equal to 1. Produce a Turing machine with tape alphabet  $\Sigma = \{0, 1, B\}$  that takes in two numbers represented in unary and outputs  $\tau = 1$  if the first number is less than the second, and  $\tau = 0$  if not.

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

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

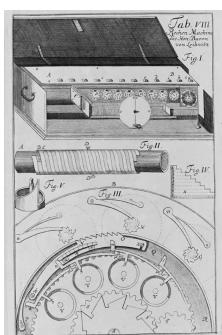
1.37 Show that if a Turing machine has the same configuration at two different steps then it will never halt. Is that sufficient condition also necessary?

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

## SECTION

### I.2 Church's Thesis

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



Leibniz's Stepped Reckoner

In the 1670's, G von Leibniz, the co-creator of Calculus, constructed the first machine that could do addition, subtraction, multiplication, division, and square roots as well. This led him to speculate on the possibility of a machine that manipulates not just numbers but also symbols, and could thereby determine the truth of scientific statements. To settle any dispute, Leibniz wrote, scholars could just 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 define what is 'algorithmic' or 'intuitively mechanically computable' or 'effective'.

A number of mathematicians proposed formalizations. One was A Church,<sup>†</sup> who proposed the  $\lambda$ -calculus. Church and his students used this system to derive many functions that are intuitively mechanically computable, including the polynomial functions as well as number-theoretic functions such as finding the remainder on division. Church suggested to Gödel, the most prominent expert in the area, that we could precisely define the set of effective functions as the set of functions that are  $\lambda$ -computable. But Gödel, who was very careful, was unconvinced.

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

- 2.1 **CHURCH'S THESIS** The set of things that can be computed by a discrete and deterministic mechanism is the same as the set of things that can be computed by a Turing machine.<sup>‡</sup>



Alonzo Church  
1903–1995

Church's Thesis is central to the Theory of Computation. It says that our technical results have a larger importance—they describe the devices that are on our desks and in our pockets. So in this section we pause to expand on some points, particularly ones that experience has shown can lead to misunderstandings.

**Evidence** We cannot prove Church's Thesis. That is, we cannot give a mathematical proof. The definition of a Turing machine, or of lambda calculus or other equivalent schemes, formalizes the notion of 'effective' or 'intuitively mechanically computable'. When a researcher agrees that it correctly explicates 'computable on a discrete and deterministic mechanism' and consents to work within that formalization, they are then free to proceed with reasoning mathematically about these systems. So in a sense, Church's Thesis comes before the mathematics, or at any rate sits outside the usual derivation and verification work of mathematics. 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

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

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

<sup>†</sup> After producing his machine model in 1935, Turing got a PhD in 1938 under Church at Princeton.

<sup>‡</sup> Some authors call this the Church-Turing Thesis. Here we figure that because Turing has the machine, we can give Church possession of the thesis.

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 discreet 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 on general recursive functions will give us a taste of another influential model. However, despite this variation, our experience is that every model yields 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  $\lambda$ -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 that Euclid's Parallel Postulate does not hold, and perhaps a similar cultural blindness is happening here. Nonetheless, if a number of very smart people go off and work independently on a question, and when they come back you find that while they have taken a wide variety of approaches, they all got the same answer, then you may be persuaded 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: the details of the definition of a Turing machine are not essential to what can be computed. For example, we can show that a one-tape machine can compute all of the functions that can be computed by a machine with two or more tapes. Thus, the fact that Definition 1.4's machines have only one tape is not an essential point.

Similarly, machines whose tape is unbounded in only one direction can compute all the functions 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 can compute any intuitively computable function by taking just a single symbol beyond the blank, that is, with  $\Sigma = \{1, B\}$  (we need the blank for all but finitely-many cells on the starting tape, and we need one more to make any distinguishable marks). Likewise, restricting to write-only machines that cannot change non-blank symbols once they are on the tape suffices to compute this set of functions. Also, although restricting to machines having only one state does not suffice, two-state machines are equipowerful with Definition 1.4's machines having unboundedly-many states.

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

on this later 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 still yields 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 those, then identifying the set of computable functions with those computable by single-tape machines would be foolish. But as with the prior argument, 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 convinces scholars still today — is clarity: Turing's analysis is compelling. Gödel noted this in the quote given above 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 in-practice 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 more felicitous.<sup>†</sup>

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 the Turing machine model can account for the logical and arithmetic computations but not the entire system, with 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 TM can easily handle the toaster's internal logic.”

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

---

<sup>†</sup> Scientists who study the brain also find Turing machines to be not the most suitable model. Note though that saying another model is a better fit is different than claiming that there are brain operations that could not, in principle, be done using a Turing machine as a substrate.

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

For a sense of the conversation, we know that the wave equation<sup>†</sup> 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, while our computers cannot?

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 reliably computing a function, we can still wonder whether there are other apparatuses that would.

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

We will not pursue this further, instead only observing that the community of researchers has weighed in by taking Church’s Thesis as the basis for its work. For us, ‘computation’ will refer to the kind of work that Turing analyzed. That’s because we want to think about symbol-pushing, not numerical analysis and not toast.

**Using Church’s Thesis** Church’s Thesis asserts that each of the models of computation—for instance, Turing machines,  $\lambda$  calculus, and the general recursive functions that we will see in the next section—are maximally capable. Here we emphasize it because it imbues our results with a larger importance. When, for instance, we will later describe a function that no Turing machine can compute then, with the thesis in mind, we will interpret the technical statement to mean that this function cannot be computed by *any* discrete and deterministic device.

Another aspect of Church’s Thesis is that because they are each maximally capable, these models and others that we won’t describe therefore all compute the same things. So we can fix one of them as our preferred formalization and get on with the mathematical analysis. For this, we choose Turing machines.

Finally, we will also leverage Church’s Thesis to make life easier. As the exercises in the prior section illustrate, while writing a few Turing machines gives some

---

<sup>†</sup>A partial differential equation that describes the propagation of waves.

insight, after a short while you find that doing more machines does not give any more illumination. Worse, focusing too much on low level machine details (or on the details of any computing model) 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. Often when we want to show that something is computable by a Turing machine, 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, our argument can proceed, “Let  $\mathcal{P}$  be that machine . . .” without us ever having to exhibit a set of four-tuple instructions. Of course, there is some danger that we will get ‘intuitively computable’ wrong. But we all have so much more experience with this than people in the 1930’s that the danger is comparatively minimal. The upside is that we can make rapid progress through the material—we can get things done.

In many cases, to claim that something is intuitively computable we will produce some code doing that thing. For this we like to use a modern programming language and our choice 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: (A) between a Turing Machine and an algorithm? (B) between a Turing Machine and a computer? (C) between a program and a computer? (D) between 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 Reconcer is like an old-timey calculator where you have to pull a lever: they can do some very limited computations, with numbers only. But I'm interested in a modern computer that is vastly more flexible in that it can also work with strings, for instance. I mean, a slide rule is not programmable, is it?” Help them understand.
- ✓ 2.5 Each of these is often given as a counterargument to Church's Thesis. Explain why each is mistaken. (A) Turing machines have an infinite tape so it is not a realistic model. (B) The total size of the universe is finite, so there are in fact only finitely many configurations possible for any computing device, whereas a Turing machine can use more than that many configurations, so it is not a realistic model.
- ✓ 2.6 One of these is a correct statement of Church's Thesis, and the others are not. Which one is right? (A) Anything that can be computed by any mechanism can be computed by a Turing machine. (B) No human computer, or machine that mimics a human computer, can out-compute a Turing machine. (C) The set of things that are computable by a discrete and deterministic mechanism

is the same as the set of things that are computable by a Turing machine.

(D) Every product of a person's mind, or product of a mechanism that mimics the activity of a person's mind, can be produced by some Turing machine.

- 2.7 List two benefits from adopting Church's Thesis.
- ✓ 2.8 Refute this objection to Church's Thesis: "Some computations have unbounded extent. That is, sometimes we look for our programs to halt but some computations, such as an operating system, are designed to never halt. The Turing machine is an inadequate model for these."
- 2.9 The idea of 'intuitively computable' certainly has subtleties. Let  $f, g: \mathbb{N} \rightarrow \mathbb{N}$ .
- (A) If both are intuitively computable then is  $f \circ g$  also intuitively computable?
  - (B) What if  $g$  is computable but  $f$  is not?
- 2.10 The computers that we use are binary. Use Church's Thesis to argue that if they were ternary, where instead of bits with two values they used trits with three, then they would compute exactly the same set of functions.
- 2.11 Use Church's thesis to argue that the indicated function exists and is computable.
- (A) Suppose that  $f_0, f_1: \mathbb{N} \rightarrow \mathbb{N}$  are computable partial functions. Show that  $h: \mathbb{N} \rightarrow \mathbb{N}$  is a computable partial function where  $h(x) = 1$  if  $x$  is in the intersection of the domain of  $f_0$  and the domain of  $f_1$ , and  $h(x)\uparrow$  otherwise.
  - (B) Do the same as in the prior item, but take the union of the two domains.
  - (C) Suppose that  $f: \mathbb{N} \rightarrow \mathbb{N}$  is a computable function that is total. Show that  $h: \mathbb{N} \rightarrow \mathbb{N}$  is a computable partial function, where  $h(x) = 1$  if  $x$  is in the range of  $f$  and  $h(x)\uparrow$  otherwise.
  - (D) Suppose  $f_0, f_1: \mathbb{N} \rightarrow \mathbb{N}$  are computable total functions. Show that their composition  $h = f_1 \circ f_0$  is a computable function  $h: \mathbb{N} \rightarrow \mathbb{N}$ .
  - (E) Suppose  $f_0, f_1: \mathbb{N} \rightarrow \mathbb{N}$  are computable partial functions. Show that their composition is a computable partial function  $f_1 \circ f_0: \mathbb{N} \rightarrow \mathbb{N}$ .
- ✓ 2.12 Suppose that  $f: \mathbb{N} \rightarrow \mathbb{N}$  is a total computable function. Use Church's Thesis to argue that this function is computable:  $h(n) = 0$  if  $n$  is in the range of  $f$ , and  $h(n)\uparrow$  otherwise.
- 2.13 Let  $f, g: \mathbb{N} \rightarrow \mathbb{N}$  be computable functions that may be either total or partial functions. Use Church's Thesis to argue that this function is computable:  $h(n) = 1$  if both  $f(n)\downarrow$  and  $g(n)\downarrow$ , and  $h(n)\uparrow$  otherwise.
- 2.14 Arguing by Church's Thesis relies on our having a solid intuition about what is implementable on a device. The following is not implementable; what goes wrong? "Given a polynomial  $p(x_0, \dots, x_n)$ , we can determine whether or not it has natural number roots by trying all possible settings of the input  $(x_0, \dots, x_n)$  to  $n+1$ -tuples of integers."
- ✓ 2.15 If you allow processes to take infinitely many steps then you can have all kinds of fun. Suppose that you have infinitely many dollars. You run into the Devil. He proposes an infinite sequence of transactions, in each of which he will

hand you two dollars and take from you one dollar. (The first will take 1/2 hour, the second 1/4 hour, etc.) You figure you can't lose. But he proves to be particular about the order in which you exchange bills. First he numbers your bills as 1, 3, 5, ... At each step he buys your lowest-numbered bill and pays you with two higher-numbered bills. Thus, he first accepts from you bill number 1 and pays you with his own bills, numbered 2 and 4. Next he buys from you bill number 2 and pays you with his bills numbered 6 and 8. How much do you end with?

*The next two exercises involve multitape machines. Definition 1.4's transition function for a single tape is  $\Delta: Q \times \Sigma \rightarrow (\Sigma \cup \{L, R\}) \times Q$ . Define a machine with  $k$ -many tapes by extending to  $\Delta: Q \times \Sigma^k \rightarrow (\Sigma \cup \{L, R\})^k \times Q$ . Thus, a typical four-tuple for a  $k = 2$  tape machine with alphabet  $\Sigma = \{0, 1, B\}$  is  $q_4(1, B) \langle 0, L \rangle q_3$ . It means that if the machine is in state  $q_4$  and the head on tape 0 is reading 1 while that on tape 1 is reading a blank, then the machine writes 0 to tape 0, moves left on tape 1, and goes into state  $q_3$ .*

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

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

## SECTION

### I.3 Recursion

In the 1930's a number of researchers besides Turing also saw the need to make precise the notion of mechanically computable function. Here we will outline an approach that is different than Turing's, both to give a sense of another approach and because we will find it useful.<sup>†</sup>

This approach has a classical mathematics flavor. It lists initial functions that are intuitively mechanically computable and it also describes ways to combine existing functions to make new ones, where if the existing ones are intuitively computable then so is the new one. An example of an intuitively computable initial function is successor  $S: \mathbb{N} \rightarrow \mathbb{N}$ , described by  $S(x) = x + 1$ , and a combiner that preserves effectiveness is function composition. Then the composition  $S \circ S$ , the plus-two operation, is also intuitively mechanically computable.

We now introduce another effectiveness-preserving combiner.

---

<sup>†</sup>It also has the advantage of not needing the codings discussed for Turing machines since it works directly with the effective functions.

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



Hermann  
Grassmann  
1809-1877

3.1 EXAMPLE This finds the sum of 3 and 2.

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

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

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

One pretty aspect of Grassmann’s approach is that it extends naturally to other operations. Multiplication has the same form.

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

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

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

And, exponentiation works the same way.

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

---

<sup>†</sup>That is, this is a discrete form of feedback. <sup>‡</sup>So the idea behind this recursion is that addition of larger numbers reduces to addition of smaller ones.

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 exactly fits the definition given above of plus.<sup>†</sup>

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

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

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

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

- 3.3 **DEFINITION** A function  $f$  can be defined by the schema<sup>‡</sup> of primitive recursion from the functions  $g$  and  $h$  when this holds.

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

The bookkeeping is that the arity of  $f$ , the number of inputs, is one more than the arity of  $g$  and one less than the arity of  $h$ . We sometimes abbreviate  $x_0, \dots, x_{k-1}$  as  $\vec{x}$ .

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

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

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

---

<sup>†</sup> Obviously Racket, like every general purpose programming language, comes with a built in addition operator, as in `(+ 3 2)`, and with a multiplication operator, as in `(* 3 2)`, and with other common arithmetic operators as well. <sup>‡</sup>A schema is an underlying organizational pattern or structure.

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

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

The arity bookkeeping is that  $\text{pred}$  has no  $x_i$ 's so  $g$  is a function of zero-many inputs and is therefore constant,  $g() = 0$ , while  $h$  has two inputs,  $h(a, b) = b$  (the first input gets ignored, so it is just a dummy variable).

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

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

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

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

Function composition covers not just the simple case of two functions  $f$  and  $g$  whose composition is defined by  $f \circ g(\vec{x}) = f(g(\vec{x}))$ . It also covers the case of simultaneous substitution, where from  $f(x_0, \dots, x_n)$  and  $h_0(y_1, \dots, y_{m_0}), \dots, h_n(y_1, \dots, y_{m_n})$ , we get  $f(h_0(y_{0,0}, \dots, y_{0,m_0}), \dots, h_n(y_{n,0}, \dots, y_{n,m_n}))$ , which is a function with  $(m_0 + 1) + \dots + (m_n + 1)$ -many inputs.

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

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

---

<sup>†</sup>There are infinitely many projections, one for each pair of natural numbers  $n, i$ . Projection is a generalization of the identity function, which is why we use the letter  $\mathcal{I}$ .

### I.3 Exercises

- ✓ 3.8 What is the difference between primitive recursion and primitive recursive?
- 3.9 What is the difference between total recursive and primitive recursive?
- 3.10 In defining  $0^0$  there is a conflict between the desire to have that every power of 0 is 0 and the desire to have that every number to the 0 power is 1. What does the definition of power given above do?
- ✓ 3.11 As the section body describes, recursion doesn't have to be logically problematic. But some recursions are; consider this one.

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

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

3.12 Consider this function.

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

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

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

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

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

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

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

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

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

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

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

- (A) Find  $s(0), \dots, s(10)$ .  
 (B) Verify that  $t$  is primitive recursive by putting it in the form given in Definition 3.3, giving suitable functions  $g$  and  $h$  (*Hint*:  $g$  is a function of no arguments, a constant). You can use functions already defined in this section.

- 3.17 Consider this recurrence.

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

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

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

- (A) Compute the values for  $n = 1, \dots, 10$ .  
 (B) Verify that  $H$  is primitive recursive by putting it in the form given in Definition 3.3, giving suitable functions  $g$  and  $h$  (*Hint*:  $g$  is a function of no arguments, a constant). You can use functions already defined in this section.

- 3.19 Define the factorial function  $\text{fact}(y) = y \cdot (y - 1) \cdots 1$  by primitive recursion, using product and a constant function.
- ✓ 3.20 Recall that the greatest common divisor of two positive integers is the largest integer that divides both. We can compute the greatest common divisor using Euclid's recursion

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

where  $\text{rem}(a, b)$  is the remainder when  $a$  is divided by  $b$ . Note that this fits the schema of primitive recursion. Use Euclid's method to compute these.

(A)  $\text{gcd}(28, 12)$  (B)  $\text{gcd}(104, 20)$  (C)  $\text{gcd}(309, 25)$

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

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

- ✓ 3.22 See the note above.
- (A) Constant function: for  $k \in \mathbb{N}$ ,  $C_k(\vec{x}) = C_k(x_0, \dots, x_{n-1}) = k$ . Hint: instead of doing the general case with  $k$ , try  $C_4(x_0, x_1)$ , the function that returns 4 for all input pairs. Also, for this you need only use the zero function, successor, and function composition.
- (B) Maximum and minimum of two numbers:  $\max(x, y)$  and  $\min(x, y)$ . Hint: use addition and proper subtraction.
- (C) Absolute difference function:  $\text{absdiff}(x, y) = |x - y|$ .
- 3.23 See the note before Exercise 3.22.
- (A) Sign predicate:  $\text{sign}(y)$ , which gives 1 if  $y$  is greater than zero and 0 otherwise.
- (B) Negation of the sign predicate:  $\text{negsign}(y)$ , which gives 0 if  $y$  is greater than zero and 1 otherwise.
- (C) Less-than predicate:  $\text{lessthan}(x, y) = 1$  if  $x$  is less than  $y$ , and 0 otherwise. (The greater-than predicate is similar.)
- ✓ 3.24 See the note before Exercise 3.22.
- (A) Boolean functions: where  $x, y$  are inputs with values 0 or 1 there is the standard one-input function

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

and two-input functions.

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

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

✓ 3.25 See the note before Exercise 3.22.

(A) Inequality predicate:  $\text{notequal}(x, y) = 0$  if  $x = y$  and 1 otherwise.

(B) Functions defined by a finite and fixed number of cases, as with these.

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

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

(A) Bounded sum function: the partial sums of a series whose terms  $g(i)$  are given by a primitive recursive function,  $S_g(y) = \sum_{0 \leq i < y} g(i) = g(0) + g(1) + \dots + g(y - 1)$  (the sum of zero-many terms is  $S_g(0) = 0$ ). Contrast this with the final item of the prior question; here the number of summands is finite but not fixed.

(B) Bounded product function: the partial products of a series whose terms  $g(i)$  are given by a primitive recursive function,  $P_g(y) = \prod_{0 \leq i < y} g(i) = g(0) \cdot g(1) \cdots g(y - 1)$  (the product of zero-many terms is  $P_g(0) = 1$ ).

(C) Bounded minimization: let  $m \in \mathbb{N}$  and let  $p(\vec{x}, i)$  be a predicate. Then the minimization operator  $M(\vec{x}, i)$ , typically written  $\mu^m i[p(\vec{x}, i)]$ , returns the smallest  $i \leq m$  such that  $p(\vec{x}, i) = 0$ , or else returns  $m$ . Hint: Consider the bounded sum of the bounded products of the predicates.

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

(A) Bounded universal quantification: suppose that  $m \in \mathbb{N}$  and that  $p(\vec{x}, i)$  is a predicate. Then  $U(\vec{x}, m)$ , typically written  $\forall_{i \leq m} p(\vec{x}, i)$ , has value 1 if  $p(\vec{x}, 0) = 1, \dots, p(\vec{x}, m) = 1$  and value 0 otherwise. (The point of writing the functional expression  $U(\vec{x}, m)$  is to emphasize the required uniformity. Stating one formula for the  $m = 1$  case,  $p(\vec{x}, 0) \cdot p(\vec{x}, 1)$ , and another for the  $m = 2$  case,  $p(\vec{x}, 0) \cdot p(\vec{x}, 1) \cdot p(\vec{x}, 2)$ , etc., is the best we can do. We can get a single derivation, that follows the rules in Definition 3.7, and that works for all  $m$ .)

(B) Bounded existential quantification: let  $m \in \mathbb{N}$  and let  $p(\vec{x}, i)$  be a predicate. Then  $A(\vec{x}, m)$ , typically written  $\exists_{i \leq m} p(\vec{x}, i)$ , has value 1 if  $p(\vec{x}, 0) = 0, \dots, p(\vec{x}, m) = 0$  is not true, and has value 0 otherwise.

- (c) Divides predicate: where  $x, y \in \mathbb{N}$  we have  $\text{divides}(x, y)$  if there is some  $k \in \mathbb{N}$  with  $y = x \cdot k$ .
- (d) Primality predicate:  $\text{prime}(y)$  if  $y$  has no nontrivial divisor.
- ✓ 3.28 We will show that the function  $\text{rem}(a, b)$  giving the remainder when  $a$  is divided by  $b$  is primitive recursive.
- (A) Fill in this table.

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

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

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

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

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

- (A) Fill in this table.

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

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

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

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

which from the second month on becomes productive? This leads to a recurrence.

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

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

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

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

3.33 Pascal's Triangle gives the coefficients of the powers of  $x$  in the expansion of  $(x + 1)^n$ . For example,  $(x + 1)^2 = x^2 + 2x + 1$  and row two of the triangle is  $\langle 1, 2, 1 \rangle$ . This recurrence gives the value at row  $n$ , entry  $m$ , where  $m, n \in \mathbb{N}$ .

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

- (A) Compute  $P(3, 2)$ .
  - (B) Compute the other entries from row three:  $P(3, 0)$ ,  $P(3, 1)$ , and  $P(3, 3)$ .
  - (C) Compute the entries in row four.
  - (D) Show that this is primitive recursive. You may use the results from Exercise 3.22 and Exercise 3.26.
- ✓ 3.34 This is McCarthy's 91 function.

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

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

3.35 Show that every primitive recursive function is total.

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

## SECTION

## I.4 General recursion

Every primitive recursive function is intuitively mechanically computable. What about the converse: is every intuitively mechanically computable function primitive recursive? In this section we will answer ‘no’.<sup>†</sup>

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

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

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

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

This is a compelling pattern.

The pattern is especially compelling when we express these functions in the form of the schema of primitive recursion. Start by letting  $\mathcal{H}_0$  be the successor function,  $\mathcal{H}_0 = \mathcal{S}$ .

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

---

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

The pattern shows in the ‘otherwise’ lines. Each one satisfies that  $\mathcal{H}_n(x, y) = \mathcal{H}_{n-1}(x, \mathcal{H}_n(x, y - 1))$ . Because of this pattern we call each  $\mathcal{H}_n$  the **level  $n$**  function, so that addition is the level 1 operation, multiplication is the level 2 operation, and exponentiation is level 3. These ‘otherwise’ lines step the function up from level to level. The definition below takes  $n$  as a parameter, writing  $\mathcal{H}(n, x, y)$  in place of  $\mathcal{H}_n(x, y)$ , to get all the levels into one formula.

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

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

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

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

We prove the level 1 statement  $\mathcal{H}_1(x, y) = x + y$  by induction on  $y$ . For the  $y = 0$  base step, the definition is that  $\mathcal{H}_1(1, x, 0) = x$ , which equals  $x + 0 = x + y$ . For the inductive step, assume that the statement holds for  $y = 0, \dots, y = k$  and consider the  $y = k + 1$  case. The definition is  $\mathcal{H}_1(x, k + 1) = \mathcal{H}_0(x, \mathcal{H}_1(x, k))$ . Apply the inductive hypothesis to get  $\mathcal{H}_0(x, x + k)$ . By the prior paragraph this equals  $x + k + 1 = x + y$ .

The other two,  $\mathcal{H}_2$  and  $\mathcal{H}_3$ , are Exercise 4.13. □

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

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

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

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

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

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

However, hyperoperation's recursion line

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

does not fit the form of primitive recursion.

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

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

Of course, just because one definition has the wrong form doesn't mean that there is no definition with the right form. However, Ackermann<sup>†</sup> proved that there isn't, that  $\mathcal{H}$  is not primitive recursive. The proof is a detour for us so it is in an Extra Section but in summary:  $\mathcal{H}$  grows faster than any primitive recursive function. That is, for any primitive recursive function  $f$  of three inputs, there is a sufficiently large  $N \in \mathbb{N}$  such that for all  $n, x, y \in \mathbb{N}$ , if  $n, x, y > N$  then  $\mathcal{H}(n, x, y) > f(n, x, y)$ . This proof is about uniformity: at every level the function  $\mathcal{H}_n$  is primitive recursive, but no primitive recursive function encompasses all levels at once — there is no one primitive recursive way to compute them all.



Wilhelm  
Ackermann  
1896–1962

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

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

**$\mu$  recursion** The right direction is hinted at in the prior section's Exercise 3.26 and Exercise 3.27. Primitive recursion does bounded operations. We can show that a programming language having only bounded loops computes all of the primitive recursive functions; see the Extra section. To include every function that is intuitively mechanically computable we must add an unbounded operation.

---

<sup>†</sup>We have seen Ackermann already, as one of the people who stated the *Entscheidungsproblem*. Functions having the same recursion as  $\mathcal{H}$  are **Ackermann functions**.

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

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

- 4.6 **EXAMPLE** The polynomial  $p(y) = y^2 + y + 41$  looks interesting because it seems, at least at the start, to output only primes.

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

We could think to test this with a program that searches for non-primes by trying  $p(0)$ , then  $p(1)$ , etc. Start with a function that computes quadratic polynomials,  $p(\vec{x}, y) = p(x_0, x_1, x_2, y) = x_2 y^2 + x_1 y + x_0$  and consider a test for the primality of the output.

$$g(\vec{x}, y) = \begin{cases} 0 & \text{if } p(\vec{x}, y) \text{ is prime} \\ 1 & \text{otherwise} \end{cases}$$

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

Some code illustrates an important point. Start with a test for primality,

```
(define (prime? n)
  (define (prime-helper n c)
    (cond [(< n (* c c)) 0]
          [(zero? (modulo n c)) 1]
          [else (prime-helper n (+ c 1))]))
  (prime-helper n 2))
```

and a way to compute the output of  $y \mapsto x_2 y^2 + x_1 y + x_0$ .

```
(define (p x0 x1 x2 y)
  (+ (* x2 y y) (* x1 y) x0))
```

Now, this is  $g$ .

```
(define (g-sub-p x0 x1 x2 y)
  (prime? (p x0 x1 x2 y)))
```

It is called g-sub-p because p is hard-coded into the source. Likewise the search routine has g-sub-p baked in. That is the point the definition makes with “ $f$  is defined from  $g$ .”

```
(define (f-sub-g x0 x1 x2)
```

---

<sup>†</sup>The vector  $\vec{x}$  abbreviates  $x_0, \dots, x_{n-1}$ .

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

(let ([y 0])
  (f-sub-g-helper y)))
```

With that, the search function finds that the polynomial above returns some non-primes.

```
> (f-sub-g 1 1 41)
40
```

Unbounded search is a theme in the Theory of Computation. For instance, we will later consider the question of which programs halt and a natural way for a program to not halt is because it is looking for something that is not there.

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

- 4.7 EXAMPLE If  $g(x, y) = 1$  for all  $x, y \in \mathbb{N}$  then  $f(x) = \mu y[g(x, y) = 0]$  is undefined for all  $x$ .
- 4.8 DEFINITION A function is **general recursive** or **partial recursive**, or  **$\mu$ -recursive**, or just **recursive**, if it can be derived from the initial operations of the **zero** function  $Z(\vec{x}) = 0$ , the **successor** function  $S(x) = x + 1$ , and the **projection** functions  $I_i(x_0, \dots, x_i \dots x_{k-1}) = x_i$  by a finite number of applications of function composition, the schema of primitive recursion, and minimization.

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

#### I.4 Exercises

*Some of these have answers that are tedious to compute. Perhaps use a computer, for instance by writing a script or using Sage.*

- ✓ 4.9 Find the value of  $\mathcal{H}_4(2, 0)$ ,  $\mathcal{H}_4(2, 1)$ ,  $\mathcal{H}_4(2, 2)$ ,  $\mathcal{H}_4(2, 3)$ , and  $\mathcal{H}_4(2, 4)$ .
- 4.10 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.
- ✓ 4.11 How many years is  $\mathcal{H}_4(3, 3)$  seconds?
- 4.12 What is the ratio  $\mathcal{H}_3(3, 3)/\mathcal{H}_2(2, 2)$ ?
- ✓ 4.13 Finish the proof of Lemma 4.2 by verifying that  $\mathcal{H}_2(x, y) = x \cdot y$  and  $\mathcal{H}_3(x, y) = x^y$ .
- 4.14 This variant of  $\mathcal{H}$  is often labeled “the” Ackermann function.

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

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

- 4.15 Prove that the computation of  $\mathcal{H}(n, x, y)$  always terminates.
- 4.16 In defining general recursive functions, Definition 4.8, 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?
- ✓ 4.17 Let  $g(x, y) = 0$  if  $x + y = 100$  and let  $g(x, y) = 1$  otherwise. Now let  $f(x) = \mu y [g(x, y) = 100]$ . For each, find the value or say that it is not defined. (A)  $f(0)$  (B)  $f(1)$  (C)  $f(50)$  (D)  $f(100)$  (E)  $f(101)$ . Give an expression for  $f$  that does not include  $\mu$ -recursion.
- 4.18 Let  $g(x, y) = 0$  if  $x \cdot y = 100$  and  $g(x, y) = 1$  otherwise. Also let  $f(x) = \mu y [g(x, y) = 100]$ . For each, find the value or say that it is not defined. (A)  $f(0)$  (B)  $f(1)$  (C)  $f(50)$  (D)  $f(100)$  (E)  $f(101)$
- ✓ 4.19 A **Fermat number** has the form  $F_n = 2^{2^n} + 1$  for  $n \in \mathbb{N}$ . The first few,  $F_0 = 3$ ,  $F_1 = 5$ ,  $F_2 = 17$ ,  $F_3 = 257$ , and  $F_4 = 65537$ , are prime. But  $F_5$  is not prime, nor is  $F_6, \dots, F_{32}$ . (We don't know of any higher primes.) Let  $g(x, y) = 0$  if  $y$  is a Fermat prime and larger than  $F_x$ , and let  $g(x, y) = 1$  otherwise. Also let  $f(x) = \mu y [g(x, y) = 0]$ . For each, what can you say? (A)  $f(0)$  (B)  $f(1)$  (C)  $f(50)$  (D)  $f(100)$  (E)  $f(F_4)$
- 4.20 Let  $g(x, y) = 0$  if  $y^2$  is greater than or equal to  $x$ , and let  $g(x, y) = 1$  otherwise. Also let  $f(x) = \mu y [g(x, y) = 0]$ . Find each, or state 'undefined'. (A)  $f(0)$  (B)  $f(1)$  (C)  $f(50)$  (D)  $f(100)$  (E)  $f(x)$
- 4.21 Let  $\text{notrelprime}(x, y) = 0$  if  $x > 1$  and  $y > 1$  and the two are not relatively prime, and let  $\text{notrelprime}(x, y) = 1$  otherwise. Find each  $f(x) = \mu y [\text{notrelprime}(x, y) = 0]$ . (A)  $f(0)$  (B)  $f(1)$  (C)  $f(2)$  (D)  $f(3)$  (E)  $f(4)$  (F)  $f(42)$  (G)  $f(x)$
- 4.22 Let  $g(x, y) = \lceil (x+1)/(y+1) - 1 \rceil$  and let  $f(x) = \mu y [g(x, y) = 0]$ .  
 (A) Find  $f(x)$  for  $0 \leq x < 6$ .  
 (B) Give a description of  $f$  that does not use  $\mu$ -recursion.
- 4.23 (A) Prove that the function  $\text{remtwo}: \mathbb{N} \rightarrow \{0, 1\}$  giving the remainder on division by two is primitive recursive.  
 (B) Use that to prove that this function is  $\mu$ -recursive:  $f(n) = 1$  if  $n$  is even, and  $f(n) \uparrow$  if  $n$  is odd.
- ✓ 4.24 Consider the Turing machine  $\mathcal{P} = \{q_0B1q_1, q_01Rq_0, q_1BRq_2, q_11Lq_1\}$ . Define  $g(x, y) = 0$  if the machine  $\mathcal{P}$ , when started on a tape that is blank except for  $x$ -many consecutive 1's and with the head under the leftmost 1, has halted after step  $y$ . Otherwise,  $g(x, y) = 1$ . Find  $f(x) = \mu y [g(x, y) = 0]$  for  $x < 6$ .
- ✓ 4.25 Define  $g(x, y)$  by: start  $\mathcal{P} = \{q_0B1q_2, q_01Lq_1, q_1B1q_2, q_111q_2\}$  on a tape that is blank except for  $x$ -many consecutive 1's and with the head under the

leftmost 1. If  $\mathcal{P}$  has halted after step  $y$  then  $g(x, y) = 0$  and otherwise  $g(x, y) = 1$ . Let  $f(x) = \mu y [g(x, y) = 0]$ . Find  $f(x)$  for  $x < 6$ . (This machine does the same task as the one in the prior exercise, but faster.)

4.26 Consider this Turing machine.

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

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

- ✓ 4.27 Define  $h: \mathbb{N}^+ \rightarrow \mathbb{N}$  by:  $h(n) = n/2$  if  $n$  is even, and otherwise  $h(n) = 3n + 1$ . Let  $H(n, k)$  be the  $k$ -fold composition of  $h$  with itself, so  $H(n, 1) = h(n)$ ,  $H(n, 2) = h \circ h(n)$ ,  $H(n, 3) = h \circ h \circ h(n)$ , etc. (We can take  $H(n, 0) = 0$ , although its value isn't interesting.) Let  $C(n) = \mu k [H(n, k) = 1]$ .
- (A) Compute  $H(4, 1)$ ,  $H(4, 2)$ , and  $H(4, 3)$ .
  - (B) Find  $C(4)$ , if it is defined.
  - (C) Find  $C(5)$ , it is defined.
  - (D) Find  $C(11)$ , it is defined.
  - (E) Find  $C(n)$  for all  $n \in [1 .. 20]$ , where defined.

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

## EXTRA

### I.A Turing machine simulator

Writing code to simulate a Turing Machine is a reasonable programming project. Here we exhibit an implementation. One of its design goals is to track closely the description of the action of a Turing machine given on page 8.

We earlier saw this Turing machine that computes the predecessor function.

$$\mathcal{P}_{\text{pred}} = \{ q_0BLq_1, q_01Rq_0, q_1BLq_2, q_11Bq_1, q_2BRq_3, q_21Lq_2 \}$$

To simulate it, the program will use this file.

```
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 code shown below along with the machine's file description, as above.

The data structure for a Turing machine is the simplest one, a list of instructions. For the instructions, the program converts each line like the six ones above into a list with four members: a number, two characters, and a number. That is, a Turing machine is stored as a list of lists. The above machine is this.

```
'((0 #\B #\L 1) (0 #\1 #\R 0) (1 #\B #\L 2) (1 #\1 #\B 1) (2 #\B #\R 3) (2 #\1 #\L 2))
```

Now for the code. After some convenience constants,

```
(define BLANK #\B) ;; Easier to read than space
(define STROKE #\1) ;;
(define LEFT #\L) ;; Move tape pointer left
(define RIGHT #\R) ;; Move tape pointer right
```

we define a configuration.

```
;; A configuration is a list of four things:
;; the current state, as a natural number
;; the symbol being read, a character
;; the contents of the tape to the left of the head, as a list of characters
;; the contents of the tape to the right of the head, as a list of characters
(define (make-config state char left-tape-list right-tape-list)
  (list state char left-tape-list right-tape-list))

(define (get-current-state config) (first config))
(define (get-current-symbol config)
  (let ([cs (second config)]) ; make horizontal whitespace like a B
    (if (char-blank? cs)
        #\B
        cs)))
(define (get-left-tape-list config) (third config))
(define (get-right-tape-list config) (fourth config))
```

Note that `get-current-symbol` translates any blank character to a B.

The heart of a Turing machine is its  $\Delta$  function, which inputs the current state and current tape symbol and returns the action to be taken—either L, or R, or a character from the tape alphabet—and the next state.

```
;; delta Find the applicable instruction
(define (delta tm current-state tape-symbol)
  (define (delta-test inst)
    (and (= current-state (first inst))
         (equal? tape-symbol (second inst)))

  (let ([inst (findf delta-test tm)])
    (if (not inst)
        '()
        (list (third inst) (fourth inst))))))
```

(The function `findf` searches through `tm` for a member on which `delta-test` returns true.)

Turing machines work discretely, step by step. If there is no relevant instruction then the machine halts, and otherwise it moves one cell left, one cell right, or writes one character.

```
;; step Do one step; from a config and the tm, yield the next config
;; Returns null if machine halts.
(define (step config tm)
  (let* ([current-state (get-current-state config)]
        [left-tape-list (get-left-tape-list config)]
        [current-symbol (get-current-symbol config)]
        [right-tape-list (get-right-tape-list config)]
        [action-next-state (delta tm current-state current-symbol)])
    (if (null? action-next-state)
        '() ; machine is halted
        (let ([action (first action-next-state)]
```

```
[next-state (second action-next-state)])
(cond
  [(char=? LEFT action) (move-left config next-state)]
  [(char=? RIGHT action) (move-right config next-state)]
  [else (make-config next-state
    action ;; not L or R so it is in tape alphabet
    left-tape-list
    right-tape-list))))))
```

Because moving left and right are more complicated, they are in separate routines.

```
;; tape-right-char Return the element nearest the head on the right side
(define (tape-right-char right-tape-list)
  (if (empty? right-tape-list)
    BLANK
    (car right-tape-list)))

;; tape-left-char Return the element nearest the head on the left
(define (tape-left-char left-tape-list)
  (tape-right-char (reverse left-tape-list)))

;; tape-right-pop Return the right tape list without char nearest the head
(define (tape-right-pop right-tape-list)
  (if (empty? right-tape-list)
    '()
    (cdr right-tape-list)))

;; tape-left-pop Return the left tape list without char nearest the head
(define (tape-left-pop left-tape-list)
  (reverse (tape-right-pop (reverse left-tape-list)))))

;; move-left Respond to Left action
(define (move-left config next-state)
  (let ([left-tape-list (get-left-tape-list config)]
        [prior-current-symbol (get-current-symbol config)]
        [right-tape-list (get-right-tape-list config)])
    ;; push old tape head symbol onto the right tape list
    (make-config next-state
      (tape-left-char left-tape-list) ;; new current symbol
      (tape-left-pop left-tape-list) ;; strip symbol off left
      (cons prior-current-symbol right-tape-list)))))

;; move-right Respond to Right action
(define (move-right config next-state)
  (let ([left-tape-list (get-left-tape-list config)]
        [prior-current-symbol (get-current-symbol config)]
        [right-tape-list (get-right-tape-list config)])
    ;; push old head symbol onto the left tape list
    (make-config next-state
      (tape-right-char right-tape-list) ;; new current symbol
      (reverse (cons prior-current-symbol (reverse left-tape-list)))
      (tape-right-pop right-tape-list))) ;; strip symbol off right
```

Finally, the implementation executes the machine by iterating the operation of a single step (`verbose` is a optional argument, with default `#t`).

```
;; execute Run a Turing machine step-by-step until it halts
(define (execute tm initial-config [verbose #t])
  ;; execute-helper
  ;; config 4-tuple configuration or '()
  ;; stp integer, step number
  ;; history list of 4-tuple configurations
  (define (execute-helper config stp history)
    (if (or (null? config)
```

```
(= (get-current-state config) HALT-STATE))
(begin
  (when verbose
    (fprintf (current-output-port)
            "step ~s: HALT\n"
            stp))
  (reverse history))
(begin
  (when verbose
    (fprintf (current-output-port)
            "step ~s: ~a\n"
            stp
            (configuration->string config)))
  (let ([next-config (step config tm)])
    (execute-helper next-config (add1 stp) (cons next-config history))))))
```

The execute routine calls the following one to give a simple picture of the machine, showing the state number and the tape contents, with the current symbol displayed between asterisks.

```
;; configuration-> string Return a string showing the tape
(define (configuration->string config)
  (let* ([state-number (get-current-state config)]
         [state-string (string-append "q" (number->string state-number))]
         [left-tape (list->string (get-left-tape-list config))]
         [current (string #\* (get-current-symbol config) #\*)] ;; wrap *'s
         [right-tape (list->string (get-right-tape-list config))])
    (string-append state-string ": " left-tape current right-tape)))
```

Besides the prior routine, the implementation has other code to do dull things such as reading the lines from the file and converting them to instruction lists.

```
;; Read a string, interpret the characters

;; comment character is a percent sign
(define COMMENT-START #\%)

;; Can write "(1 a L 3)"
(define s "1 a L 3")
(define s0 "(3 b b 4)")

(define (current-state-string->number s)
  (if (eq? #\<\ ( (string-ref s 0)) ;; allow instr to start with (
      (string->number (substring s 1))
      (string->number s)))
(define (current-symbol-string->char s)
  (string-ref s 0))
(define (action-symbol-string->char s)
  (string-ref s 0))
(define (next-state-string->number s)
  (if (eq? #\) (string-ref s (- (string-length s) 1))) ;; ends with )?
      (string->number (substring s 0 (- (string-length s) 1)))
      (string->number s)))
(define (string->instruction s)
```

There is a bit more code for getting the file name from the command line, etc., that does not bear on simulating a Turing machine, so we will leave it aside.

Below is a run of the simulator, with its command line invocation. It takes input from the file pred.tm shown earlier. When the machine starts the input is 111, with a current symbol of 1 and the tape to the right of 11 (the tape to the left is empty).

```
$ ./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: 111*B*
step 5: q1: 11*B*B
step 6: q2: 1*1*B*B
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 lack of complications also makes it easy to parse, and then to translate into inputs for sophisticated graphics creation programs.

### 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  $\mathcal{P}_{\text{add}}$  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  $\Sigma = \{0, 1, B\}$ . The machine starts with the tape blank except for a contiguous string of 0's and 1's, and with the head under the first non-blank symbol. When it finishes, the tape will have either just a 1 if the input contained the desired substring, or otherwise just a 0. We will do this in stages, building a few of what amounts to subroutines.

(A) Write instructions, starting in state  $q_{10}$ , so that if initially the machine's head is under the first of a sequence of non-blank entries then at the end the head will be to the right of the final such entry.

(B) Write a sequence of instructions, starting in state  $q_{20}$ , so that if initially the head is just to the right of a sequence of non-blank entries, then at the end all entries are blank.

(C) Write the full machine, including linking in the prior items.

A.5 Modify the simulator so that it can run for a limited number of steps.

(A) Modify it so that, given  $k \in \mathbb{N}$ , if the Turing machine hasn't halted after  $k$  steps then the simulator stops.

(B) Do the same, but replace  $k$  with a function  $(k\ n)$  where  $n$  is the input number (assume the machine's input is a string of 1's).

## EXTRA

## I.B Hardware

Following Turing, we've gone through a development based on transition tables. But given a table, is there sure to be a physical implementation with that behavior?

Put another way, in programming languages there are mathematical operators that are constructed from other, simpler, mathematical operators. For instance,  $\sin(x)$  may be calculated via its Taylor polynomial from addition and multiplication. But how do the simplest operators work?

We will show how to get any desired behavior. For this, we will work with machines that take finite binary sequences, bitstrings, as inputs and outputs. The easiest approach is via propositional logic.

Appendix C has a review. Here, the tables write 0 in place of  $F$  and 1 in place of  $T$ , as is the convention in electronics. In an electronic device, these would typically stand for different voltage levels.

		not $P$		$P$ and $Q$		$P$ or $Q$	
		$P$	$\neg P$	$P$	$Q$	$P \wedge Q$	$P \vee Q$
$P$	$\neg P$	0	1	0	0	0	0
		1	0	0	1	0	1
$P$	$\neg P$	1	0	1	0	0	1
		1	1	1	1	1	1

Recall that with the three operators ' $\wedge$ ', ' $\vee$ ' and ' $\neg$ ', we can go from a truth table to a propositional logic expression with that table. That is, we can specify an input-output behavior and then produce an expression having that behavior. Below are two examples.

For the table on the left, focus on the row with output 1. Obviously the expression  $\neg P \wedge \neg Q$  makes this row take on value 1 and every other row take on value 0.

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

For the table on the right again focus on the rows with 1's. Target the second row with the clause  $\neg P \wedge \neg Q \wedge R$ . The clause for the third row is  $\neg P \wedge Q \wedge \neg R$  and the fifth row's is  $P \wedge \neg Q \wedge \neg R$ . Now put these parts together with  $\vee$ 's.

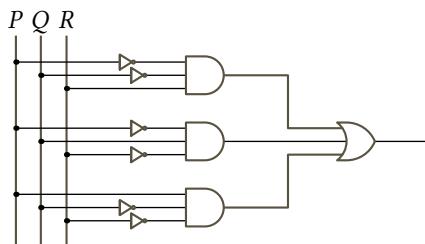
$$(\neg P \wedge \neg Q \wedge R) \vee (\neg P \wedge Q \wedge \neg R) \vee (P \wedge \neg Q \wedge \neg R) \quad (*)$$

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



Claude Shannon  
1916–2001

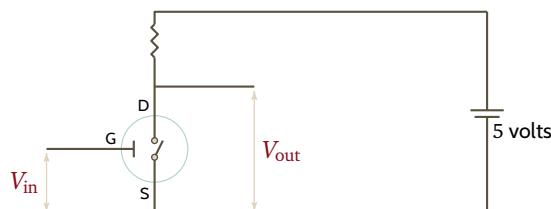
A schematic of a circuit that implements expression (\*), given below, shows three input signals on the three wires at left. For instance, to implement the first clause, the top AND gate is fed the not  $P$ , the not  $Q$ , and the  $R$  signals. The second and third clauses are implemented in the other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



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

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

The answer is that the descriptions above abstract out that issue. Here is the internal construction of a kind of NOT gate.



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

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

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

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

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

## I.B Exercises

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

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

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

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

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

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

B.6 One propositional logic operator that was not covered in the description is **Exclusive Or, XOR**. It is defined by:  $P \text{ XOR } Q$  is 1 if  $P \neq Q$ , and is 0 otherwise.

(A) Specify a truth table and from it construct a disjunctive normal form propositional logic expression.

(B) Use that to make a circuit.

B.7 For the tables below, construct a disjunctive normal form propositional logic expression and use that to make a circuit

(A) for the table on the left,

(B) for the one on the right.

$P$	$Q$	$R$	
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

B.8 A propositional logic operator that was not covered above is **implication**,  $\rightarrow$ .

It is defined by:  $P \rightarrow Q$  is 1 unless  $P$  is 1 and  $Q$  is 0.

(A) Make a truth table and from it construct a disjunctive normal form propositional logic expression.

(B) Use that to make a circuit.

B.9 Construct a circuit with the behavior specified in the tables below: (A) the table on the left, (B) the one on the right.

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

B.10 The most natural way to add two binary numbers works like the grade school addition algorithm. Start at the right with the one's column. Add those two and possibly carry a 1 to the next column. Then add down the next column, including any carry. Repeat this from right to left.

(A) Use this method to add the two binary numbers 1011 and 1101.

(B) Make a truth table giving the desired behavior in adding the numbers in a column. It must have three inputs because of the possibility of a carry. It must also have two output columns, one for the total and one for the carry.

(C) Draw the circuits.

## EXTRA

## I.C Game of Life

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



John von Neumann  
1903-1957

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



John Conway 1937–  
2020

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

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

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

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

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

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

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

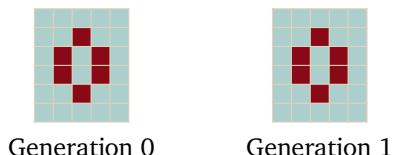


The pictures show the part of the game board containing the cells that are alive. Two generations suffice to show everything that happens, which isn’t much.

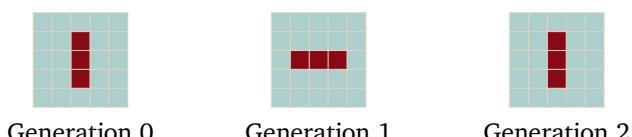
Some other patterns don’t die, but don’t do much of anything, either. This is a **block**. It is stable from generation to generation.



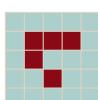
Because it doesn’t change, Conway calls this a “still life.” Another still life is the **beehive**.



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



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

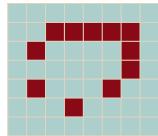


It moves one cell vertically and one horizontally every four generations, crawling across the screen.

### C.1 ANIMATION: Gliding, left and right.

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

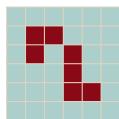
The glider pattern an example of a **spaceship**, a pattern that reappears, displaced, after a number of generations. Here is another, the **medium weight spaceship**.



It also crawls across the screen.

### C.2 ANIMATION: Moving across space.

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



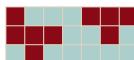
Here it eats a medium weight spaceship.

### C.3 ANIMATION: Eating a spaceship.

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

### I.C Exercises

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



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

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

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

### EXTRA

#### I.D Ackermann's function is not primitive recursive

We have seen that the hyperoperation  $\mathcal{H}$  is the natural generalization of successor, addition, multiplication, etc.

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

We have cited that  $\mathcal{H}$ , while intuitively mechanically computable, is not primitive recursive. We will show here that a closely related variant is not primitive recursive.

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

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



Rózsa Péter  
1905–1977

Any function based on the recursion in the bottom line is called an **Ackermann function**.<sup>†</sup> We will prove that  $\mathcal{A}$  is not primitive recursive.

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

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

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

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

Those are big numbers.

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

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

This is the first item. We will prove it by induction on  $k$ .

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

The  $k = 0$  base step is  $\mathcal{A}(0, y) = y + 1 > y$ . For the inductive step, assume that statement (\*) holds for  $k = 0, \dots, k = n$  and consider the  $k = n + 1$  case.

We must verify this statement,

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

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

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

---

<sup>†</sup>There are many different Ackermann functions in the literature. A common one is the function of one variable  $\mathcal{A}(k, k)$ .

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

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

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

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

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

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

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

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

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

and the second holds because the function  $h$  is level  $k$ . □

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

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

then  $f$  is level  $k + 3$ .

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

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

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

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

and is larger than  $f(\vec{x}, z)$  by the inductive hypothesis.

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

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

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

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

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

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

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

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

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

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

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

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

## I.D Exercises

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

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

D.9 Prove these parts of Lemma D.1. (A) Item B (B) Item C (C) Item D  
(D) Item E (E) Item F

D.10 Verify each identity. (A)  $\mathcal{A}(0, y) = y + 1$  (B)  $\mathcal{A}(1, y) = 2 + (y + 3) - 3$  (C)  $\mathcal{A}(2, y) = 2 \cdot (y + 3) - 3$  (D)  $\mathcal{A}(3, y) = 2y + 3 - 3$  (E)  $\mathcal{A}(4, y) = 2 \uparrow\uparrow (n+3) - 3$   
 In the last one, the up-arrow notation (due to D Knuth) means that there is a power tower containing  $n + 3$  many 2's. Recall that powers do not associate, so  $2^{(2^2)} \neq (2^2)^2$ ; the notation means the first type of association, from the top down.

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

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

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

## EXTRA

### I.E LOOP programs

Compared to general recursive functions, primitive recursive functions have the advantage that their computational behavior is easy to analyze. We will support this contention by giving a programming language that computes primitive recursive functions.

The most familiar looping constructs are `for` and `while`. The difference is that a `while` loop can go an unbounded number of times, but in a `for` loop you know in advance the number of times that the code will pass through the loop.

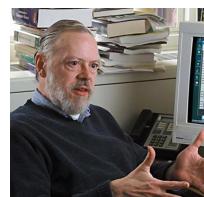
E.1 **THEOREM (MEYER AND RITCHIE, 1967)** A function is primitive recursive if and only if it can be computed without using unbounded loops. More precisely, we can compute in advance, using only primitive recursive functions, how many iterations will occur.

We will show this by computing primitive recursive functions in a language that lacks unbounded loops. Programs in this language execute on a machine model that has registers  $r_0, r_1, \dots$ , which hold natural numbers.

A **LOOP program** is a sequence of instructions, which are of four kinds: (i)  $x = 0$  sets the contents of the register named  $x$  to zero, (ii)  $x = x + 1$  increments the contents of register  $x$ , (iii)  $x = y$  copies the contents of register  $y$  into register  $x$ , leaving  $y$  unchanged, and (iv) `loop x ... end`.

For the last, the dots the middle are replaced by a sequence of any of the four kinds of statements. In particular, it might contain a nested loop. The semantics are that the instructions of the inside program are executed repeatedly, with the number of repetitions given by the natural number in register  $x$ .

Running the program below results in the register  $r_0$  getting the value of 6 (the indenting is only for visual clarity).



Dennis Ritchie  
1941–2011, inventor of C

```
r1 = 0
r1 = r1 + 1
r1 = r1 + 1
r2 = r1
r2 = r2 + 1
r0 = 0
loop r1
  loop r2
    r0 = r0 + 1
  end
end
```

Very important: in `loop x ... end`, changes in the contents of register  $x$  while the inside code is run do not alter the number of times that the machine steps through that loop. Thus, when this loop ends the value in  $r0$  will be twice what it was at the start.

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

We want to interpret LOOP programs as computing functions so we need a convention for input and output. Where the function takes  $n$  inputs, we will start the program after loading the inputs into the registers numbered 0 through  $n - 1$ . And where the function has  $m$  outputs, we take the values to be the integers that remain in the registers numbered 0 through  $m - 1$  when the program has finished. For example, this LOOP program computes the two-input one-output function proper subtraction  $f(x, y) = x \div y$ .

```
loop r1
  r0 = 0
  loop r0
    r1 = r0
    r0 = r0 + 1
  end
end
```

That is, if we load  $x$  into  $r0$  and  $y$  into  $r1$ , and run the above routine, then the output  $x \div y$  will be in  $r0$ .

To show that for each primitive recursive function there is a LOOP program, we can show how to compute each initial function, and how to do the combining operations of function composition and primitive recursion.

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

The composition of two functions is easy. Suppose that  $g(x_0, \dots, x_n)$  and  $f(y_0, \dots, y_m)$  are computed by LOOP programs  $P_g$  and  $P_f$ , and that  $g$  is an  $m$ -output function so that the composition  $f \circ g$  is defined. Then concatenating, so that the instructions of  $P_g$  are followed by the instructions of  $P_f$ , gives the LOOP program for  $f \circ g$ , since it uses the output of  $g$  as input to compute the action of  $f$ .

General composition starts with

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

and produces  $f(h_0(y_{0,0}, \dots y_{0,m_0}), \dots h_n(y_{n,0}, \dots y_{n,m_n}))$ . The issue is that were we to load the sequence of inputs  $y_{0,0}, \dots$  into  $r0, \dots$  and start computing  $h_0$  then, for one thing, there is a danger that it could overwrite the inputs for  $h_1$ . So we must do some machine language-like register manipulations to shuttle data in and out as needed.

Specifically, let  $P_f, P_{h_0}, \dots P_{h_n}$  compute the functions. Each uses a limited number of registers so there is an index  $j$  large enough that no program uses register  $j$ . By definition, the LOOP program  $P$  to compute the composition will be given the sequence of inputs starting in register 0. The first step is to copy these inputs to start in register  $j$ . Next, zero out the registers below register  $j$ , copy  $h_0$ 's arguments down to begin at  $r0$  and run  $P_{h_0}$ . When it finishes, copy its output above the final register holding the inputs (that is, to the register numbered  $(m_0 + 1) + \dots + (m_n + 1)$ ). Repeat for the rest of the  $h_i$ 's. Finish by copying the outputs down to the initial registers, zeroing out the remaining registers, and running  $P_f$ .

The other combiner operation is primitive recursion.

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

Suppose that we have LOOP programs  $P_g$  and  $P_h$ . The register swapping needed is similar to what happens for composition so we won't discuss it. The program  $P_f$  starts by running  $P_g$ . Then it sets a fresh register to 0; call that register  $t$ . Now it enters a loop based on the register  $y$  (that is, successive times through the loop count down as  $y, y-1$ , etc.). The body of the loop computes  $f(x_0, \dots x_{k-1}, t+1) = h(f(x_0, \dots x_{k-1}, t), x_0, \dots x_{k-1}, t)$  by running  $P_h$  and then it increments  $t$ .

Thus if a function is primitive recursive then it is computed by a LOOP program. The converse holds also, but proving it is beyond our scope.

We have an interpreter for the LOOP language with two interesting aspects. The first is that we change the syntax, replacing the C-looking syntax above with a LISP-ish one. For instance, we swap the syntax on the left for that on the right.

<code>r1 = r1 + 1 loop r1   r0 = r0 + 1 end</code>	<code>((incr r1) (loop r1 (incr r0)))</code>
--	--

The advantage of this switch is that the parentheses automatically match the beginning of a loop with the matching end and so the interpreter that we write will not need a stack to keep track of loop nesting.

This interpreter has registers  $r0, r1, \dots$ , that hold natural numbers. We keep track of them in a list of pairs.

<code>; A register is a pair (name:symbol contents:natural number)</code>
---

```
(define REGLIST '())
(define (show-reg) ; debugging
  (write REGLIST) (newline))

(define (clear-reg!)
  (set! REGLIST '()))

;; make-reg-name return the symbol giving the standard name of a register
(define (make-reg-name i)
  (string->symbol (string-append "r" (number->string i)))))

;; Getters and setters for the list of registers
;; set-reg-value! Set the value of an existing register or initialize a new one
(define (set-reg-value! r v)
  (set! REGLIST (alist-update! r v REGLIST equal?)))
;; get-reg Return pair whose car is given r; if no such reg, return (r . 0)
(define (get-reg r)
  (let ((val (assoc r REGLIST)))
    (if val
        val
        (begin
          (set-reg-value! r 0)
          (cons r 0)))))

(define (get-reg-value r)
```

There are an unlimited number of registers; when `set-reg-value!` is asked to act on a register that is not on the list, it puts it on the list.

Besides the initialization done by `set-reg-value!`, two of the remaining three LOOP operations are straightforward.

```
; increment-reg! Increment the register
(define (increment-reg! r)
  (set-reg-value! r (+ 1 (get-reg-value r))))
;; copy-reg! Copy value from r0 to r1, leave r0 unchanged
(define (copy-reg! r0 r1)
  (set-reg-value! r1 (get-reg-value r0)))

;; Implement each operation
(define (intr-zero pars)
  (set-reg-value! (car pars) 0))
(define (intr-incr pars)
  (increment-reg! (car pars)))
(define (intr-copy pars)
  (set-reg-value! (car pars) (get-reg-value (cadr pars))))
```

The last LOOP operation is `loop` itself. Such an instruction can have inside it the body of an entire LOOP program.

```
(define (intr-loop pars)
  (letrec ((reps (get-reg-value (car pars)))
          (body (cdr pars)))
    (iter (lambda (rep)
            (cond
              ((equal? rep 0) '())
              (else (intr-body body)
                    (iter (- rep 1)))))))
  (iter reps)))

;; intr-body Interpret the body of loop programs
(define (intr-body body)
  (cond
    ((null? body) '()))
```

```
(else (let ((next-inst (car body))
           (tail (cdr body)))
      (let ((key (car next-inst))
            (pars (cdr next-inst)))
        (cond
          ((eq? key 'zero) (intr-zero pars))
          ((eq? key 'incr) (intr-incr pars))
          ((eq? key 'copy) (intr-copy pars))
          ((eq? key 'loop) (intr-loop pars))))
      (intr-body tail)))))
```

Finally, there is the code to interpret a program, including initializing the registers so we can view the input-output behavior as computing a function.

```
;; The data is a list of the values to put in registers r0 r1 r2 ..
;; Value of a program is the value remaining in r0 at end.
(define (interpret progr data)
  (init-reg data)
  (intr-body progr)
  (get-reg-value (make-reg-name 0)))

;; init-reg Initialize the registers r0, r1, r2, .. to the values in data
(define (init-reg data)
  (define (init-reg-helper i data)
    (if (null? data)
        '()
        (begin
          (set-reg-value! (make-reg-name i) (car data))
          (init-reg-helper (+ i 1) (cdr data)))))
  (clear-reg!)
  (set-reg-value! (make-reg-name 0) 0)
  (init-reg-helper 0 data))
```

As given, this prints only the value of  $r_0$ , which is all we shall need here.

Here is a sample usage. The LOOP program, in LISP syntax, is pe.

```
#;1> (load "loop.scm")
#;2> (define pe '((incr r0) (incr r0) (loop r0 (incr r0))))
#;3> (interpret pe '(5))
14
```

With an initial value of 5, after being incremented twice then  $r_0$  will have a value of 7. So the loop runs seven times, each time incrementing  $r_0$ , resulting in an output value of 14.

We can now make an interpreter for the C-like syntax shown earlier. We first do some bookkeeping such as splitting the program into lines and dropping comments. Then we convert the instructions as a purely string operation. Thus  $r_0 = 0$  becomes (zero  $r_0$ ). Similarly,  $r_0 = r_0 + 1$  becomes (incr  $r_0$ ) and  $r_0 = r_1$  becomes (copy  $r_0 r_1$ ). Finally, loop  $r_0$  becomes (loop  $r_0$  (note the missing closing paren), and end becomes ).

Here is the second interesting thing about the interpreter. Now that the C-like syntax has been converted to a string in LISP-like syntax, we just need to interpret the string as a list. We write it to a file and then load that file. That is, unlike in many programming languages, in Racket we can create code on the fly.

Here is an example of running the interpreter. The program in C-like syntax is this.

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

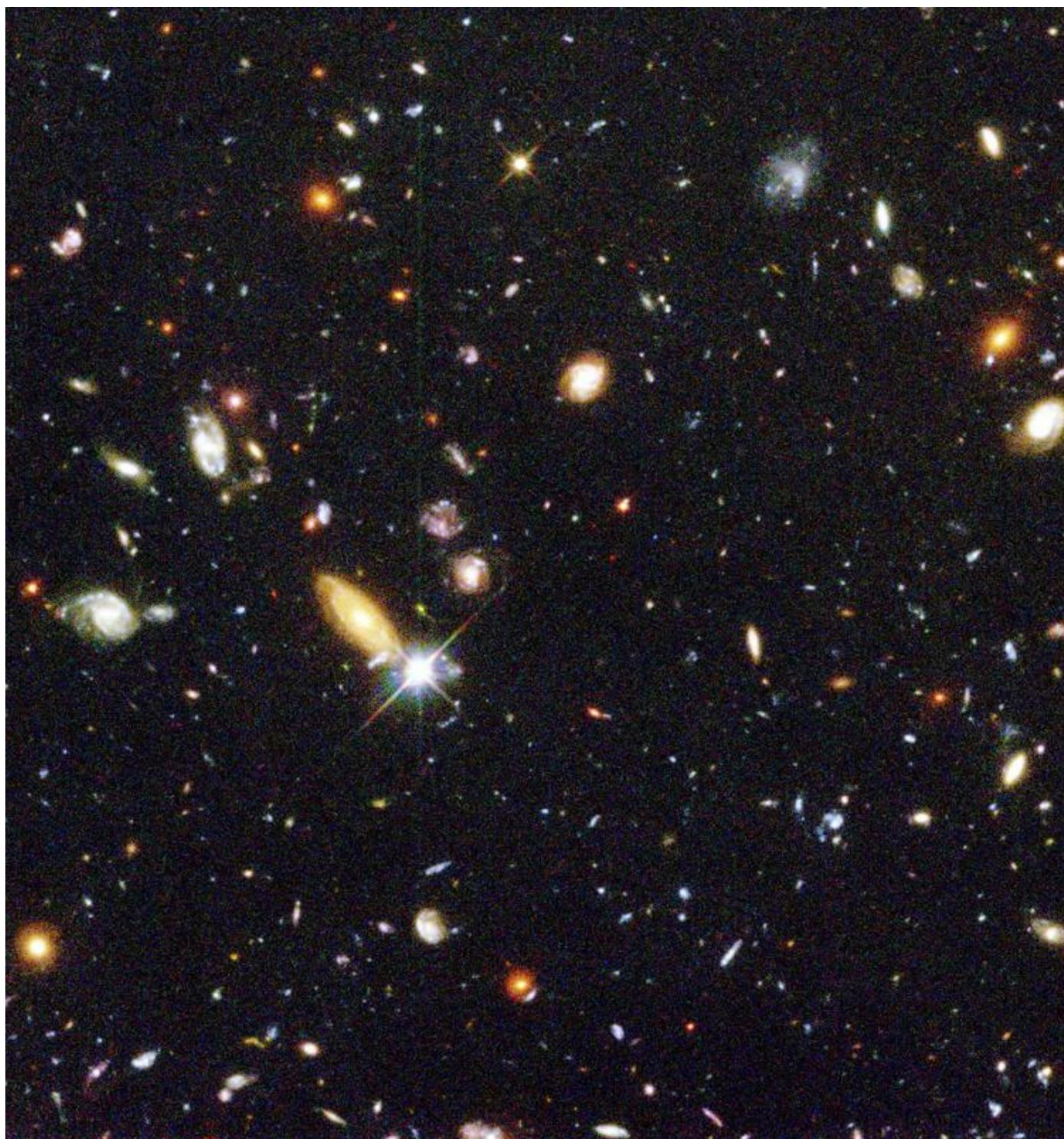
And here we run that in the Racket interpreter.

```
#;4> (define p "r1 = r1 + 1\nr1 = r1 + 1\nloop r1\nr0 = r0 + 1\nend")
#;5> (loop-without-parens p '())
; loading fn.scm ...
2
```

## I.E Exercises

- E.2 Write a LOOP program that triples its input.
- E.3 Write a LOOP program that adds two inputs.
- E.4 Modify the interpreter to allow statements like  $r0 = r0 + 2$ .
- E.5 Modify the interpreter to allow statements like  $r0 = 1$ .
- E.6 Modify the definition of `interpret` so that it takes one more argument, a natural number  $m$ , and returns the contents of the first  $m$  registers.





## CHAPTER

# II Background

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

## SECTION

### II.1 Infinity

We will show that there are more functions than Turing machines, and that therefore there are some functions with no associated machine.

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

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



Galileo Galilei  
1564–1642

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

The second paradox of infinity is **Aristotle's Paradox**. On the left below are two circles, one with a smaller radius. If we roll them through one revolution then the trail left by the smaller one is shorter. However, if we put the smaller inside the larger and roll them, as in a train wheel, then they leave equal-length trails.

---

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

1.2 ANIMATION: Circles of different radii  
have different circumferences.

1.3 ANIMATION: Embedded circles  
rolling together.

As with Galileo's Paradox, we might think of the set of points on the circumference of larger circle as being a bigger set. But the right point of view is that the two sets have the same size, the same number of elements, in that they correspond—point-for-point, the circumference of the smaller matches the circumference of the larger.

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

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

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



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

*Proof* Exercise 1.48. □

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

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

We now give that relation a name. This definition extends from the finite to the infinite Lemma 1.5’s observation about same-sized sets.

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

- 1.8 EXAMPLE Stated in terms of the definition, Galileo’s Paradox is that the set of perfect squares  $S = \{n^2 \mid n \in \mathbb{N}\}$  has the same cardinality as  $\mathbb{N}$  because the function  $f: \mathbb{N} \rightarrow S$  given by  $f(n) = n^2$  is a correspondence. It is one-to-one because if  $f(x_0) = f(x_1)$  then  $x_0^2 = x_1^2$  and thus, since these are natural numbers,  $x_0 = x_1$ . It is onto because any element of the codomain  $y \in S$  is the square of some  $n$  from the domain  $\mathbb{N}$ , that is,  $y = f(n)$  for some  $n$  in the domain, 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  $g(x) = x \cdot (2\pi r_1 / 2\pi r_0)$  is a correspondence; verification is Exercise 1.42.
- 1.10 EXAMPLE The set of natural numbers greater than zero,  $\mathbb{N}^+ = \{1, 2, \dots\}$ , has the same cardinality as  $\mathbb{N}$ . A correspondence is  $f: \mathbb{N} \rightarrow \mathbb{N}^+$  given by  $n \mapsto n + 1$ .



Georg Cantor  
1845–1918

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

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

For us, by far the most important infinite set is  $\mathbb{N}$ .

- 1.12 **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**. A function whose domain is the natural numbers **enumerates**, or **is an enumeration of**, its range.

The idea behind ‘enumeration’ is that  $f : \mathbb{N} \rightarrow S$  lists the range set: first  $f(0)$ , then  $f(1)$ , etc. The listing may have repeats, so that perhaps for some  $n_0 \neq n_1$  we have  $f(n_0) = f(n_1)$ . As always, we are particularly interested in functions that are computable. The phrase ‘a function whose domain is the natural numbers’ implies that the function is total but in section 7 we will show how to use partial functions in the place of total functions.

- 1.13 **EXAMPLE** The set of multiples of three,  $3\mathbb{N} = \{3k \mid k \in \mathbb{N}\}$ , is countable. The natural map  $g : \mathbb{N} \rightarrow 3\mathbb{N}$  is  $g(n) = 3n$ . Of course,  $g$  is effective.
- 1.14 **EXAMPLE** The set  $\mathbb{N} - \{2, 5\} = \{0, 1, 3, 4, 6, 7, \dots\}$  is countable. The function below, both defined and illustrated with a table, closes up the gaps.

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

$f(n) \mid \begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \dots \end{array}$	$\begin{array}{ccccccc} 0 & 1 & 3 & 4 & 6 & 7 & 8 \dots \end{array}$
--	--

This function is clearly one-to-one and onto. It is also computable; we could write a program whose input/output behavior is  $f$ .

- 1.15 **EXAMPLE** The set of prime numbers  $P$  is countable. There is a function  $p : \mathbb{N} \rightarrow P$  where  $p(n)$  is the  $n$ -th prime, so that  $p(0) = 2, p(1) = 3$ , etc. We won’t produce a formula for this function but obviously we can write a program whose input/output behavior is  $p$ , so it is a correspondence that is effective.
- 1.16 **EXAMPLE** Fix the set of symbols  $\Sigma = \{a, \dots, z\}$ . Consider the set of strings made of those symbols, such as  $az$ ,  $xyz$ , and  $abba$ . The set of all such strings, denoted  $\Sigma^*$ , is countable. This table illustrates the correspondence that we get by taking the strings in ascending order of length.

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

(The first entry is the empty string,  $\epsilon = ''$ .) This correspondence is also effective.

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

also effective.

$$\begin{array}{c|ccccccccc} n \in \mathbb{N} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & \dots \\ f(n) \in \mathbb{Z} & 0 & +1 & -1 & +2 & -2 & +3 & -3 & \dots \end{array}$$

We have not given any non-computable functions because a goal of this chapter is to show that such functions exist, and we are not there yet.

We close this section by circling back to the paradoxes of infinity that we began with. In the prior example, the naive expectation is that the positives and the negatives combined make  $\mathbb{Z}$  twice as big as  $\mathbb{N}$ . But this is the point of Galileo's Paradox; the right way to measure how many elements a set has is not through superset and subset, the right way is through cardinality.

Finally, we will mention one more paradox, due to Zeno (circa 450 BC). He imagines 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  $x_1$ . On reaching  $x_1$ , Achilles finds that the tortoise has moved ahead to  $x_2$ . At any  $x_i$ , Achilles will always be behind and so, the tortoise reasons, Achilles can never win. 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  $(0 .. \infty)$ .



1.18 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).)

In this book we shall often leverage open-endedness, usually the open-endedness of  $\mathbb{N}$  at infinity. We have already seen it in Galileo's Paradox.

## II.1 Exercises

- ✓ 1.19 Verify Example 1.13, that the function  $g: \mathbb{N} \rightarrow \{3k \mid k \in \mathbb{N}\}$  given by  $n \mapsto 3n$  is both one-to-one and onto.
- 1.20 A friend tells you, “The perfect squares and the perfect cubes have the same number of elements because these sets are both one-to-one and onto.” Straighten them out.
- 1.21 Let  $f, g: \mathbb{Z} \rightarrow \mathbb{Z}$  be  $f(x) = 2x$  and  $g(x) = 2x - 1$ . Give a proof or a counterexample for each. (A) If  $f$  one-to-one? Is it onto? (B) If  $g$  one-to-one? Onto? (C) Are  $f$  and  $g$  inverse to each other?

- ✓ 1.22 Decide if each function is one-to-one, onto, both, or neither. You cannot just answer ‘yes’ or ‘no’, you must justify the answer.
- (A)  $f: \mathbb{N} \rightarrow \mathbb{N}$  given by  $f(n) = n + 1$
  - (B)  $f: \mathbb{Z} \rightarrow \mathbb{Z}$  given by  $f(n) = n + 1$
  - (C)  $f: \mathbb{N} \rightarrow \mathbb{N}$  given by  $f(n) = 2n$
  - (D)  $f: \mathbb{Z} \rightarrow \mathbb{Z}$  given by  $f(n) = 2n$
  - (E)  $f: \mathbb{Z} \rightarrow \mathbb{N}$  given by  $f(n) = |n|$ .
- 1.23 Decide if each is a correspondence (you must also verify): (A)  $f: \mathbb{Q} \rightarrow \mathbb{Q}$  given by  $f(n) = n + 3$  (B)  $f: \mathbb{Z} \rightarrow \mathbb{Q}$  given by  $f(n) = n + 3$  (C)  $f: \mathbb{Q} \rightarrow \mathbb{N}$  given by  $f(a/b) = |a \cdot b|$ . Hint: this could be thought of as a trick question.
- 1.24 Decide if each set finite or infinite and justify your answer. (A)  $\{1, 2, 3\}$  (B)  $\{0, 1, 4, 9, 16, \dots\}$  (C) the set of prime numbers (D) the set of real roots of  $x^5 - 5x^4 + 3x^2 + 7$
- 1.25 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)  $\mathbb{Z}, \{i^3 \mid i \in \mathbb{Z}\}$
- ✓ 1.26 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  $(-1..1)$ .
- ✓ 1.27 Verify that the function  $f(x) = 1/x$  is a correspondence between the subsets  $(0..1)$  and  $(1..\infty)$  of  $\mathbb{R}$ .
- 1.28 Give a formula for a correspondence between the sets  $\{1, 2, 3, 4, \dots\}$  and  $\{7, 10, 13, 16, \dots\}$ .
- ✓ 1.29 Consider the set of characters  $C = \{0, 1, \dots, 9\}$  and the set of integers  $A = \{48, 49, \dots, 57\}$ .
  - (A) Produce a correspondence  $f: C \rightarrow A$ .
  - (B) Verify that the inverse  $f^{-1}: A \rightarrow C$  is also a correspondence.
- ✓ 1.30 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
- ✓ 1.31 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.
- 1.32 Example 1.8 gives one correspondence between the natural numbers and the perfect squares. Give another.
- 1.33 Fix  $c \in \mathbb{R}$  such that  $c > 1$ . Show that  $f: \mathbb{R} \rightarrow (0..\infty)$  given by  $x \mapsto c^x$  is a correspondence.

1.34 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.

1.35 For each, give functions from  $\mathbb{N}$  to itself. You must justify your claims.

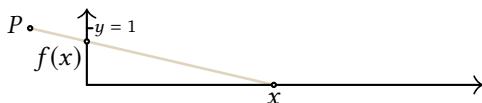
- (A) Give two examples of functions that are one-to-one but not onto.
- (B) Give two examples of functions that are onto but not one-to-one.
- (C) Give two that are neither.
- (D) Give two that are both.

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

1.37 Show that the sets have the same cardinality. (A)  $\{4k \mid k \in \mathbb{N}\}$ ,  $\{5k \mid k \in \mathbb{N}\}$   
 (B)  $\{0, 1, \dots, 99\}$ ,  $\{m \in \mathbb{N} \mid m^2 < 10000\}$  (c)  $\{0, 1, 3, 6, 10, 15, \dots\}$ ,  $\mathbb{N}$

✓ 1.38 Produce a correspondence between each pair of open intervals of reals.

- (A)  $(0 \dots 1), (0 \dots 2)$
- (B)  $(0 \dots 1), (a \dots b)$  for real numbers  $a < b$
- (C)  $(0 \dots \infty), (a \dots \infty)$  for the real number  $a$
- (D) This shows a correspondence  $x \mapsto f(x)$  between a finite interval of reals and an infinite one,  $f: (0 \dots 1) \rightarrow (0 \dots \infty)$ .



The point  $P$  is at  $(-1, 1)$ . Give a formula for  $f$ .

✓ 1.39 Not every set involving irrational numbers is uncountable. The set  $S = \{\sqrt[n]{2} \mid n \in \mathbb{N} \text{ and } n \geq 2\}$  contains only irrational numbers. Show that it is countable.

1.40 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.

1.41 Use the arctangent function to prove that the sets  $(0 \dots 1)$  and  $\mathbb{R}$  have the same cardinality.

1.42 Example 1.9 restates Aristotle's Paradox as: the intervals  $I_0 = [0 \dots 2\pi r_0]$  and  $I_1 = [0 \dots 2\pi r_1]$  have the same cardinality, for  $r_0, r_1 \in \mathbb{R}^+$ .

- (A) Verify it by checking that  $g: I_0 \rightarrow I_1$  given by  $g(x) = x \cdot (r_1/r_0)$  is a correspondence.
- (B) Show that where  $a < b$ , the cardinality of  $[0 \dots 1]$  equals that of  $[a \dots b]$ .
- (C) Generalize by showing that where  $a < b$  and  $c < d$ , the real intervals  $[a \dots b]$  and  $[c \dots d]$  have the same cardinality.

1.43 Suppose that  $D \subseteq \mathbb{R}$ . A function  $f: D \rightarrow \mathbb{R}$  is **strictly increasing** if  $x < \hat{x}$  implies that  $f(x) < f(\hat{x})$  for all  $x, \hat{x} \in D$ . Prove that any strictly increasing

function is one-to-one; it is therefore a correspondence between  $D$  and its range. (The same applies if the function is strictly decreasing.) Does this hold for  $D \subseteq \mathbb{N}$ ?

- ✓ 1.44 A paradoxical aspect of both Aristotle's and Galileo's examples is that they gainsay Euclid's "the whole is greater than the part," because they name sets where that set equinumerous with a proper subset. Here, show that each pair of a set and a proper subset has the same cardinality. (A)  $\mathbb{N}$ ,  $\{2n \mid n \in \mathbb{N}\}$   
 (B)  $\mathbb{N}$ ,  $\{n \in \mathbb{N} \mid n > 4\}$
- 1.45 Example 1.14 illustrates that we can take away a finite number of elements from the set  $\mathbb{N}$  without changing the cardinality. Prove that—prove that if  $S$  is a finite subset of  $\mathbb{N}$  then  $\mathbb{N} - S$  is countable.
- 1.46 (A) Let  $D = \{0, 1, 2, 3\}$  and  $C = \{\text{Spades, Hearts, Clubs, Diamonds}\}$ , and let  $f: D \rightarrow C$  be given by  $f(0) = \text{Spades}$ ,  $f(1) = \text{Hearts}$ ,  $f(2) = \text{Clubs}$ ,  $f(3) = \text{Diamonds}$ . Find the inverse function  $f^{-1}: C \rightarrow D$  and verify that it is a correspondence.  
 (B) Let  $f: D \rightarrow C$  be a correspondence. Show that the inverse function exists. That is, show that associating each  $y \in C$  with the  $x \in D$  such that  $f(x) = y$  gives a well-defined function  $f^{-1}: C \rightarrow D$ .  
 (C) Show that the inverse of a correspondence is also a correspondence, that the function defined in the prior item is a correspondence.
- 1.47 Prove that a set  $S$  is infinite if and only if it has the same cardinality as a proper subset of itself.
- 1.48 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 naturally ask how it interacts with set operations. Here we are interested in the cross product operation—after all, Turing machines 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 in Galelio's Paradox this might lead to a mistaken guess that it has more members than  $\mathbb{N}$ . But  $S$  is countable.

To count it, the mistake to avoid is to go vertically up a column, which will never get to the other column. Instead, alternate between the columns.

### 2.2 ANIMATION: Enumerating $\{0, 1\} \times \mathbb{N}$ .

This illustrates the correspondence as a table.

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

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

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

*Proof* Exercise 2.35; use the above example as a model. □

- 2.4 **EXAMPLE** The natural next set has infinitely many copies:  $\mathbb{N} \times \mathbb{N}$ .

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

Counting up the first column or out the first row won't work; here also we need to alternate. So instead do a breadth-first traversal: start in the lower left with  $\langle 0, 0 \rangle$ , then 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  $\mathbb{N} \times \mathbb{N}$ .

This presents the same correspondence as a table.

Number	0	1	2	3	4	5	6	...
Pair	$\langle 0, 0 \rangle$	$\langle 0, 1 \rangle$	$\langle 1, 0 \rangle$	$\langle 0, 2 \rangle$	$\langle 1, 1 \rangle$	$\langle 2, 0 \rangle$	$\langle 0, 3 \rangle$	...

That this procedure gives a correspondence is perfectly evident. Clearly it is effective, that is, we can write a program to give the association in the above table. But there is a formula for going from the bottom line to the top that is amusing, so we will produce it. For that, Animation 2.5 numbers the diagonals.

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

Consider for example the pair  $\langle 1, 2 \rangle$ . It is on diagonal number 3 and, just as  $3 = 1 + 2$ , in general the diagonal number of a pair is the sum of its entries. Diagonal 0 has one entry, diagonal 1 has two entries, and diagonal 2 has three entries, so before diagonal 3 come six pairs. Thus, on diagonal 3 the initial pair  $\langle 0, 3 \rangle$  gets enumerated as number 6. With that, the pair  $\langle 1, 2 \rangle$  is number 7.

So to find the number corresponding to  $\langle x, y \rangle$ , note first that it lies on diagonal  $d = x + y$ . The number of entries prior to diagonal  $d$  is  $1 + 2 + \dots + d$ . This is an arithmetic series with total  $d(d + 1)/2$ . Thus on diagonal  $d$  the first pair,  $\langle 0, x + y \rangle$ , has number  $(x + y)(x + y + 1)/2$ . The next pair on that diagonal,  $\langle 1, x + y - 1 \rangle$ , gets the number  $1 + [(x + y)(x + y + 1)/2]$ , etc.

- 2.6 **DEFINITION** Cantor's correspondence  $\text{cantor}: \mathbb{N}^2 \rightarrow \mathbb{N}$  or unpairing function, or diagonal enumeration<sup>†</sup> is  $\text{cantor}(x, y) = x + [(x + y)(x + y + 1)/2]$ . Its inverse is the pairing function, pair:  $\mathbb{N} \rightarrow \mathbb{N}^2$ .

<sup>†</sup>Some authors write  $\langle x, y \rangle$ , using diamond brackets for the function that we denote  $\text{cantor}(x, y)$ . Here we only use diamond brackets to denote a sequence.

2.7 EXAMPLE Two early examples are  $\text{cantor}(1, 2) = 7$  and  $\text{cantor}(2, 0) = 5$ . A later one is  $\text{cantor}(0, 36) = 666$ .

2.8 LEMMA Cantor's correspondence is a correspondence, so the cross product  $\mathbb{N} \times \mathbb{N}$  is countable. Further, the sets  $\mathbb{N}^3 = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ , and  $\mathbb{N}^4, \dots$  are all countable.

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

The prior paragraph forms the base step of an induction argument. For example, to do  $\mathbb{N}^3$  the idea is to consider a triple such as  $\langle 1, 2, 3 \rangle$  to be a pair whose first entry is a pair,  $\langle \langle 1, 2 \rangle, 3 \rangle$ . That is, define  $\text{cantor}_3: \mathbb{N}^3 \rightarrow \mathbb{N}$  by  $\text{cantor}_3(x, y, z) = \text{cantor}(\text{cantor}(x, y), z)$ . Exercise 2.29 shows that this function is a correspondence. The full induction step details are routine.  $\square$

2.9 COROLLARY The cross product of finitely many countable sets is countable.

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

2.10 EXAMPLE The set of rational numbers  $\mathbb{Q}$  is countable. We know how to alternate between positives and negatives so we will be done showing this if we count the nonnegative rationals,  $f: \mathbb{N} \rightarrow \mathbb{Q}^+ \cup \{0\}$ . A nonnegative rational number is a numerator-denominator pair  $\langle n, d \rangle \in \mathbb{N} \times \mathbb{N}^+$ , except for the complication that pairs collapse, meaning for instance that when the numerator is 4 and the denominator is 2 then we get the same rational as when  $n = 2$  and  $d = 1$ .

We will count with a program instead of a formula. Given an input  $i$ , the program finds  $f(i)$  by using prior values,  $f(0), f(1), \dots f(i-1)$ . It loops, using the pairing function  $\text{cantor}^{-1}$  to generate pairs:  $\text{cantor}^{-1}(0), \text{cantor}^{-1}(1), \text{cantor}^{-1}(2), \dots$ . For each generated pair  $\langle a, b \rangle$ , if the second entry is 0 or if the rational number  $a/b$  is in the list of prior values then the program rejects the pair, going on to try the next one. The first pair that it does not reject is  $f(i)$ .

The technique of that example is **memoization** or **caching** and it is widely used. For example, when you visit a web site your browser saves any image to your disk. If you visit the site again then your browser checks if the image has changed. If not then it will use the prior copy, reducing download time.

The next result establishes that we can use memoization in general.

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

*Proof* Assume first that  $S$  is countable. If it is empty then we are done. If it is finite

but nonempty,  $S = \{s_0, \dots s_{n-1}\}$ , then this  $f: \mathbb{N} \rightarrow S$  map is onto.

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

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

For the converse assume that either  $S$  is empty or there is an onto map from  $\mathbb{N}$  to  $S$ . If  $S = \emptyset$  then it is countable by Definition 1.12 so suppose that there is an onto map  $f$ . If  $S$  is finite then it is countable so suppose that  $S$  is infinite. Define  $\hat{f}: \mathbb{N} \rightarrow S$  by  $\hat{f}(n) = f(k)$  where  $k$  is the least natural number such that  $f(k) \notin \{\hat{f}(0), \dots \hat{f}(n-1)\}$ . Such a  $k$  exists because  $S$  is infinite and  $f$  is onto. Observe that  $\hat{f}$  is both one-to-one and onto, by construction.  $\square$

This section starts off by noting that it is natural to see how countability interacts with set operations.

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

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

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

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

For item (3) in the two-set case, suppose that  $S_0$  and  $S_1$  are countable. If either set is empty, or both sets are empty, then the result is trivial because for instance  $S_0 \cup \emptyset = S_0$ . So instead suppose that  $f_0: \mathbb{N} \rightarrow S_0$  and  $f_1: \mathbb{N} \rightarrow S_1$  are onto. Lemma 2.3 gives a correspondence taking  $\mathbb{N}$  to  $\{0, 1\} \times \mathbb{N}$ , inputting natural numbers and outputting pairs  $\langle i, j \rangle$  where  $i$  is either 0 or 1. Call that function  $g: \mathbb{N} \rightarrow \{0, 1\} \times \mathbb{N}$ . Then this is the desired function onto the set  $S_0 \cup S_1$ .

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

This approach extends to any finite number of countable sets.

Finally, we start with countably many countable sets,  $S_i$  for  $i \in \mathbb{N}$ , and show that their union  $S_0 \cup S_1 \cup \dots$  is countable. If all but finitely many are empty then we can fall back to the finite case so assume that infinitely many of the sets are

nonempty. Throw out the empty ones because they don't affect the union, write  $S_j$  for the remaining sets, and assume that we have a family of correspondences  $f_j: N \rightarrow S_j$ . Then use Cantor's pairing function: the desired map from  $\mathbb{N}$  onto  $S_0 \cup S_1 \cup \dots$  is  $\hat{f}(n) = f_j(k)$  where  $\text{pair}(n) = \langle j, k \rangle$ .  $\square$

Very important: Lemma 2.3 and Lemma 2.8 on the cross product of countable sets are effectivizable. That is, if sets correspond to  $\mathbb{N}$  via some effective function then their cross product corresponds to  $\mathbb{N}$  via an effective function. We finish this section by applying that to Turing machines—we will give a way to effectively number the Turing machines.

Each Turing machines instruction is a four-tuple, a member of  $Q \times \Sigma \times (\Sigma \cup \{\text{L}, \text{R}\}) \times Q$ , where  $Q$  is the set of states and  $\Sigma$  is the tape alphabet. So by the above results, we can enumerate the set of instructions: there is an instruction that corresponds to 0, one corresponding to 1, etc. This is effective, meaning that there is a program that takes in a natural number and outputs the corresponding instruction, as well as a program that takes in an instruction and outputs the corresponding number.

With that, we can effectively number the Turing machines. One way is: starting with a Turing machine  $\mathcal{P}$ , use the prior paragraph to convert each of its instructions to a number, giving a set  $\{i_0, i_1, \dots, i_n\}$ , and then output the number associated with that machine as  $e = g(\mathcal{P}) = 2^{i_0} + 2^{i_1} + \dots + 2^{i_n}$ .

The association in the other direction is much the same. Given a natural number  $e$ , represent it in binary  $e = 2^{j_0} + \dots + 2^{j_k}$ , form the set of instructions corresponding to the numbers  $j_0, \dots, j_k$ , and that is the output Turing machine. (Except that we must check that this set is deterministic, that no two of the instructions begin with the same  $q_p T_p$ , which we can do effectively, and if it is not deterministic then let the output be the empty machine  $\mathcal{P} = \{\}$ .)

The exact numbering that we use doesn't matter much as long as it is has certain properties, the ones in the following definition, for the rest of the book we will just fix a numbering and cite its properties rather than mess with its details.

2.13 **DEFINITION** A **numbering** is a function that assigns to each Turing machine a natural number. For any Turing machine, the corresponding number is its **index number**, or **Gödel number**, or **description 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  $\phi_e$ .

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

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

instructions, or from the source to the index.

- 2.14 REMARK Here is an alternative scheme that is simple and is useful for thinking about numbering, but that we won't make precise. On a computer, the text of a program is saved as a bit string, which we can interpret as a binary number,  $e$ . In the other direction, given a binary  $e$  on the disk, we can disassemble it into assembly language source code. So there is an association between binary numbers and source code.

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

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

- 2.16 REMARK Stated in terms of everyday programming, we can get infinitely different many source codes that have the same compiled behavior. One way is by starting with one source code and adding to the bottom a comment line containing the number  $k$ .

Now that we have counted the Turing machines we are close to this book's most important result. The next section shows that there are so many natural number functions that they cannot be counted, they cannot be put in correspondence with  $\mathbb{N}$ . This will prove that there are functions not computed by any Turing machine.

## II.2 Exercises

- ✓ 2.17 Extend the table of Example 2.1 through  $n = 12$ . Where  $f(n) = \langle x, y \rangle$ , give formulas for  $x$  and  $y$ .
- ✓ 2.18 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)  $\langle 200, 0 \rangle$
- ✓ 2.19 Corollary 2.12 says that the union of two countable sets is countable.
  - (A) For each of the two sets  $T = \{2k \mid k \in \mathbb{N}\}$  and  $F = \{5m \mid m \in \mathbb{N}\}$  produce a correspondence  $f_T: \mathbb{N} \rightarrow T$  and  $f_F: \mathbb{N} \rightarrow F$ . Give a table listing the values of  $f_T(0), \dots, f_T(9)$  and give another table listing  $f_F(0), \dots, f_F(9)$ .
  - (B) Give a table listing the first ten values for a correspondence  $f: \mathbb{N} \rightarrow T \cup F$ .
- 2.20 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  $\langle 200, 1 \rangle$ .
- ✓ 2.21 Example 2.1 says that the method for two columns extends to three. Give an enumeration of  $\{0, 1, 2\} \times \mathbb{N}$ . That is, where  $g(n) = \langle x, y \rangle$  give a formula for  $x$

and  $y$ . Find the pair corresponding to 0, 10, 100, and 1 000. Find the number corresponding to  $\langle 1, 2 \rangle$ ,  $\langle 1, 20 \rangle$ , and  $\langle 1, 200 \rangle$ .

2.22 Give an enumeration  $f$  of  $\{0, 1, 2, 3\} \times \mathbb{N}$ . That is, where  $f(n) = \langle x, y \rangle$ , give a formula for  $x$  and  $y$ . Also give an enumeration  $f$  of  $\{0, 1, 2, \dots k\} \times \mathbb{N}$ .

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

✓ 2.24 Definition 2.6's function  $\text{cantor}(x, y) = x + [(x + y)(x + y + 1)/2]$  is clearly effective since it is given as a formula. Show that its inverse, pair:  $\mathbb{N} \rightarrow \mathbb{N}^2$ , is also effective by sketching a way to compute it with a program.

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

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

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

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

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

(B) Finish the argument.

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

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

2.31 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.

2.32 Corollary 2.12 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.

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

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

2.35 Give a proof of Lemma 2.3.

✓ 2.36 Consider a programming language using the alphabet  $\Sigma$  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  $\Sigma^5$  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.

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

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

(B) Show that  $g$  is a correspondence.

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

2.38 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  $\text{cantor}(1, y) = \text{cantor}(2, 1)$ .

2.39 It is fun to prove directly, rather than via the cross product, that the countable union of countably many countable sets is countable.

(A) For the union of two countable sets,  $S_0$  and  $S_1$ , partition the natural numbers into the odds and evens,  $C_0$  and  $C_1$ , that is, into the set of numbers whose binary representation does not end in 0 and the set whose representation does end in 0. Each is countably infinite so there are correspondences  $g_0: \mathbb{N} \rightarrow C_0$  and  $g_1: \mathbb{N} \rightarrow C_1$ . Use the  $g$ 's to produce an onto function  $\hat{f}: \mathbb{N} \rightarrow S_0 \cup S_1$ .

(B) For the union of three countable sets  $S_0$ ,  $S_1$ , and  $S_2$ , instead split the natural numbers into three parts: the set  $C_0$  of numbers whose binary expansion does not end in 0, the set  $C_1$  of numbers whose expansion ends in one but not two 0's, and  $C_2$ , those numbers ending in two 0's. (Take 0 to be an element of the second set.) There are correspondences  $g_0: \mathbb{N} \rightarrow C_0$ ,  $g_1: \mathbb{N} \rightarrow C_1$  and  $g_2: \mathbb{N} \rightarrow C_2$ . Produce an onto  $\hat{f}: \mathbb{N} \rightarrow S_0 \cup S_1 \cup S_2$ .

(C) To show that the countable union of countable sets is countable start with countably many countable sets,  $S_i$  for  $i \in \mathbb{N}$ . Assume that there are infinitely many nonempty sets, throw out the empty ones because they don't affect the union, call the rest  $S_j$ , and extend the prior item.

## SECTION

### II.3 Diagonalization

Cantor's definition of cardinality led us to produce correspondences. But it can also happen that no correspondence exists. We now introduce a powerful technique to show that. It is central to the entire Theory of Computation.

**Diagonalization** There is a set so large that it is not countable, that is, a set for which no correspondence exists with  $\mathbb{N}$  or any subset of it. It is the set of reals,  $\mathbb{R}$ .

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

This result is important but so is the technique of proof that we will use. We will pause to develop the intuition behind it. The table below illustrates a function  $f: \mathbb{N} \rightarrow \mathbb{R}$ , listing some inputs and outputs, with the outputs aligned on the decimal point.

$n$	<i>Decimal expansion of <math>f(n)</math></i>
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 ...
4	0 . 1 0 1 0 0 1 0 ...
5	-0 . 6 2 5 5 4 1 8 ...
:	:

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

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

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

(In this argument we have skirted a technicality, that some real numbers have two different decimal representations. For instance,  $1.000\dots = 0.999\dots$  because the two differ by less than 0.1, less than 0.01, etc. This is a potential snag because it means that even though we have constructed a representation that is different than all the representations on the list, it still might not be that the number is different than all the numbers on the list. However, dual representation only happens for decimals when one of the representations ends in 0's while the other ends in 9's. That's why we build  $z$  using 1's and 2's.)

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

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

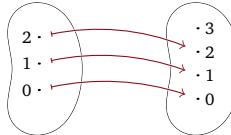
Now let  $z$  be the real number that has 0 to the left of its decimal point, and whose  $i$ -th decimal digit is  $g(f(i)[i])$ . Then for all  $i$ ,  $z \neq f(i)$  because  $z[i] \neq f(i)[i]$ . So  $f$  is not onto.  $\square$

### 3.2 DEFINITION

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

We next define when one set has fewer, or more, elements than another. Out

intuition comes from trying to make a correspondence between the two finite sets  $\{0, 1, 2\}$  and  $\{0, 1, 2, 3\}$ . There are just too many elements in the codomain for any map to cover them all. The best we can do is something like this, which is one-to-one but not onto.



3.3 **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  $T$ .

3.4 **EXAMPLE** There is a one-to-one function from  $\mathbb{N}$  to  $\mathbb{R}$ , namely the inclusion map that sends  $n \in \mathbb{N}$  to itself,  $n \in \mathbb{R}$ . So  $|\mathbb{N}| \leq |\mathbb{R}|$ . (By Theorem 3.1 above the cardinality is actually strictly less.)

3.5 **REMARK** We cannot emphasize too strongly that the work in this chapter, including the prior example, is startling and profound. Some infinite sets have more elements than others. And, in particular, the reals have more elements than the naturals. As dramatized by Galileo's Paradox, this is not just that the naturals are a subset of the reals. Instead it means that the set of naturals cannot be made to correspond with the set of reals. This is like the children's game Musical Chairs. We have countably many chairs  $P_0, P_1, \dots$ , chairs indexed by the natural numbers, but there are so many children, so many real numbers, that some child is left without a chair.

The wording of that definition implies that if both  $|S| \leq |T|$  and  $|T| \leq |S|$  then  $|S| = |T|$ . That is true but the proof is beyond our scope; see Exercise 3.31.

For the next result, recall that a set's **characteristic function**  $\mathbb{1}_S$  is the Boolean function determining membership:  $\mathbb{1}_S(s) = 1$  if  $s \in S$  and  $\mathbb{1}_S(s) = 0$  if  $s \notin S$ . Thus for the set of two letters  $S = \{a, c\}$ , the characteristic function with domain  $\Sigma = \{a, \dots, z\}$  is  $\mathbb{1}_S(a) = 1$ ,  $\mathbb{1}_S(b) = 0$ ,  $\mathbb{1}_S(c) = 1$ ,  $\mathbb{1}_S(d) = 0$ , ...  $\mathbb{1}_S(z) = 0$ . Recall also that the **power set**  $\mathcal{P}(S)$  is the collection of subsets of  $S$ . For instance, if  $S = \{a, c\}$  then  $\mathcal{P}(S) = \{\emptyset, \{a\}, \{c\}, \{a, c\}\}$ .

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

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

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

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

In the table below, the first row lists the values of the characteristic function

$\mathbb{1}_{f(a)}: S \rightarrow \{0, 1\}$  on the inputs a, b, and c. The second row lists the input/output values for  $\mathbb{1}_{f(b)}$ . And, the third row lists  $\mathbb{1}_{f(c)}$ .

$s \in S$	$f(s)$	$\mathbb{1}_{f(s)}(a)$	$\mathbb{1}_{f(s)}(b)$	$\mathbb{1}_{f(s)}(c)$
a	{b, c}	0	1	1
b	{b}	0	1	0
c	{a, b, c}	1	1	1

We show that  $f$  is not onto by producing a member of  $\mathcal{P}(S)$  that is not any of the three sets in (\*). For that, take the table's diagonal, 011, and flip the bits from 0 to 1 or from 1 to 0, to get 100. That's the characteristic function of  $R = \{a\}$ . This set is not equal to  $f(a)$  because their characteristic functions differ on a, it is not  $f(b)$  because they differ on b, and it is not  $f(c)$  because they differ on c.

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

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

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

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

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

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

*Proof* Lemma 2.8 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. Consequently, some natural number function is without an associated Turing machine.  $\square$

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

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

### II.3 Exercises

- 3.9 Your study partner is confused about the diagonal argument. “If you had an infinite list of numbers, it would clearly contain every number, right? I mean, if you had a list that was truly INFINITE, then you simply couldn’t find a number that is not on the list!” Help them out.
- 3.10 Your classmate says, “Professor, I’m confused. The set of numbers with one decimal place, such as 25.4 and 0.1, is clearly countable—just take the integers and shift all the decimal places by one. The set with two decimal places, such as 2.54 and 6.02 is likewise countable, etc. This is countably many sets, each of which is countable, and so the union is countable. The union is the whole reals, so I think that the reals are countable.” Where is their mistake?
- 3.11 Verify Cantor’s Theorem, Theorem 3.6, for these finite sets. (A)  $\{0, 1, 2\}$   
 (B)  $\{0, 1\}$  (C)  $\{0\}$  (D)  $\{\}$
- ✓ 3.12 Use Definition 3.3 to prove that the first set has cardinality less than or equal to the second set.  
 (A)  $S = \{1, 2, 3\}$ ,  $\hat{S} = \{11, 12, 13\}$   
 (B)  $T = \{0, 1, 2\}$ ,  $\hat{T} = \{11, 12, 13, 14\}$   
 (C)  $U = \{0, 1, 2\}$ , the set of odd numbers  
 (D) the set of even numbers, the set of odds
- 3.13 One set is countable and the other is uncountable. Which is which?  
 (A)  $\{n \in \mathbb{N} \mid n + 3 < 5\}$   
 (B)  $\{x \in \mathbb{R} \mid x + 3 < 5\}$
- ✓ 3.14 Characterize each set as countable or uncountable. You need only give a one-word answer. (A)  $[1..4] \subset \mathbb{R}$  (B)  $[1..4] \subset \mathbb{N}$  (C)  $[5..\infty) \subset \mathbb{R}$   
 (D)  $[5..\infty) \subset \mathbb{N}$
- 3.15 List all of the functions with domain  $A_2 = \{0, 1\}$  and codomain  $\mathcal{P}(A_2)$ . How many functions are there for a set  $A_3$  with three elements?  $n$  elements?
- 3.16 List all of the functions from  $S$  to  $T$ . How many are one-to-one?  
 (A)  $S = \{0, 1\}$ ,  $T = \{10, 11\}$   
 (B)  $S = \{0, 1\}$ ,  $T = \{10, 11, 12\}$
- ✓ 3.17 Short answer: fill each blank by choosing from (i) uncountable, (ii) countable or uncountable, (iii) finite, (iv) countable, (v) finite, countably infinite, or uncountable (you might use an answer more than once, or not at all). Give the sharpest conclusion possible. You needn’t give a proof.  
 (A) If  $A$  and  $B$  are finite then  $A \cup B$  is   
 (B) If  $A$  is countable and  $B$  is finite then  $A \cup B$  is   
 (C) If  $A$  is countable and  $B$  is uncountable then  $A \cup B$  is   
 (D) if  $A$  is countable and  $B$  is uncountable then  $A \cap B$  is
- 3.18 Short answer: suppose that  $S$  is countable and consider  $f: S \rightarrow T$ . For both of the items below, list all of these that are possible: (i)  $S$  is finite, (ii)  $T$  is finite,

(iii)  $S$  is countably infinite, (iv)  $T$  is countably infinite, (v)  $T$  is uncountable, where  
 (A) the map is onto, (B) the map is one-to-one.

- ✓ 3.19 Give a set with a larger cardinality than  $\mathbb{R}$ .
- ✓ 3.20 Recall that  $\mathbb{B} = \{0, 1\}$ .
  - (A) Show that the set of finite bit strings,  $\langle b_0 b_1 \dots b_{k-1} \rangle$  where  $b_i \in \mathbb{B}$  and  $k \in \mathbb{N}$ , is countable.
  - (B) An infinite bit string  $f = \langle b_0, b_1, \dots \rangle$  is a function  $f: \mathbb{N} \rightarrow \mathbb{B}$ . Show that the set of infinite bit strings is uncountable, using diagonalization.
- 3.21 Prove that for two sets,  $S \subseteq T$  implies  $|S| \leq |T|$ .
- 3.22 Use diagonalization to show that this is false: all functions  $f: \mathbb{N} \rightarrow \mathbb{N}$  with a finite range are computable.
- 3.23 You study with someone who says, “Yes, obviously there are different sizes of infinity. The plane  $\mathbb{R}^2$  obviously has infinitely many more points than the line  $\mathbb{R}$ , so it is a larger infinity.” Convince them that, although there are indeed different sizes of infinity, their argument is wrong because the cardinality of the plane is the same as the cardinality of the line. *Hint:* consider the function  $f: \mathbb{R}^2 \rightarrow \mathbb{R}$  such that  $f(x, y)$  interleaves the digits of the two input numbers.
- 3.24 In mathematics classes we mostly work with rational numbers, perhaps leaving the impression that irrational numbers are rare. Actually, there are more irrational numbers than rationals. Prove that while the set of rational numbers is countable, the set of irrational numbers is uncountable.
- ✓ 3.25 Example 2.10 shows that the rational numbers are countable. What happens when the diagonal argument given in Theorem 3.1 is applied to a listing of the rationals? Consider a sequence  $q_0, q_1, \dots$  that contains all of the rationals. For each of those numbers use a decimal expansion  $q_i = d_i.d_{i,0}d_{i,1}d_{i,2} \dots$  (with  $d_i \in \mathbb{Z}$  and  $d_{i,j} \in \{0, \dots, 9\}$ ) that does not end in all 9's, so that the decimal expansion is determined.
  - (A) Let  $g$  be the map on the decimal digits 0, 1, … 9 given by  $g(1) = 2$ , and  $g(0) = g(2) = g(3) = \dots = 1$ . Define  $z = \sum_{n \in \mathbb{N}} g(d_{n,n}) \cdot 10^{-(n+1)}$ . Show that  $z$  is irrational.
  - (B) Use the prior item to conclude that the diagonal number  $d = \sum_{n \in \mathbb{N}} d_{n,n} \cdot 10^{-(n+1)}$  is irrational. *Hint:* show that, unlike a rational number, it has no repeating pattern in its decimal expansion.
  - (C) Why is the fact that the diagonal is not rational not a contradiction to the fact that we can enumerate all of the rationals?
- 3.26 Verify Cantor’s Theorem in the finite case by showing that if  $S$  is finite then the cardinality of its power set is  $|\mathcal{P}(S)| = 2^{|S|}$ .
- 3.27 The definition  $R = \{s \mid s \notin f(s)\}$  is the key to the proof of Cantor’s Theorem, Theorem 3.1. This story illustrates the idea: a high school yearbook asks each graduating student  $s_i$  make a list  $f(s_i)$  of class members that they predict will

someday be famous. Define the set of humble students  $H$  to be those who are not on their own list. Show that no student's list equals  $H$ .

3.28 The proof of Theorem 3.1 works 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  $0.01000\dots$  and  $0.00111\dots$ . How could you make the argument work in base two?

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

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

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

3.31 Definition 3.3 extends the definition of equal cardinality to say that  $|A| \leq |B|$  if there is a one-to-one function from  $A$  to  $B$ . The **Schröder–Bernstein theorem** is that if both  $|S| \leq |T|$  and  $|T| \leq |S|$  then  $|S| = |T|$ . We will walk through the proof. It depends on finding chains of images: for any  $s \in S$  we form the associated chain by iterating application of the two functions, both to the right and the left, as here.

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

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

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

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

```

graph LR
    S((S)) -- "f: " --> T((T))
    T -- "g: " --> S
    S -- "0-->" --> b
    S -- "1-->" --> c
    S -- "2-->" --> a
    T -- "a-->" --> 0
    T -- "b-->" --> 1
    T -- "c-->" --> 2
    S --- S_group
    T --- T_group
  
```

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

(A) Consider  $S = \{0, 1, 2, 3\}$  and  $T = \{a, b, c, d\}$ . Let  $f$  associate  $0 \mapsto a, 1 \mapsto b, 2 \mapsto d$  and  $3 \mapsto c$ . Let  $g$  associate  $a \mapsto 0, b \mapsto 1, c \mapsto 2$  and  $d \mapsto 3$ . Check that these maps are one-to-one. List the chain associated with each element of  $S$  and the chain associated with each element of  $T$ .

(B) For infinite sets a chain can have a first element, an element without any preimage. Let  $S$  be the even numbers and let  $T$  be the odds. Let  $f: S \rightarrow T$

be  $f(x) = x + 1$  and let  $g: T \rightarrow S$  be  $g(x) = x + 1$ . Show each map is one-to-one. Show there is a single chain and that it has a first element.

- (c) Argue that we can assume without loss of generality that  $S$  and  $T$  are disjoint sets.
- (d) Assume that  $S$  and  $T$  are disjoint and that  $f: S \rightarrow T$  and  $g: T \rightarrow S$  are one-to-one. Show that every element of either set is in a unique chain, and that each chain is of one of four kinds: (i) those that repeat after some number of terms (ii) those that continue infinitely in both directions without repeating (iii) those that continue infinitely to the right but stop on the left at some element of  $S$ , and (iv) those that continue infinitely to the right but stop on the left at some element of  $T$ .
- (e) Show that for any chain the function below is a correspondence between the chain's elements from  $S$  and its elements from  $T$ .

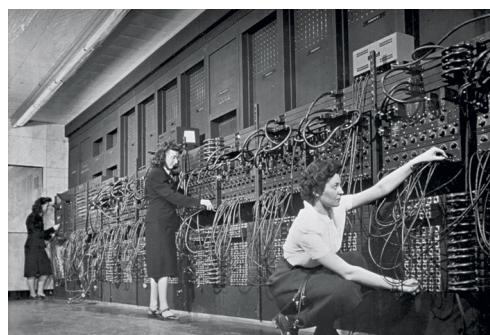
$$h(s) = \begin{cases} f(s) & - \text{if } s \text{ is in a sequence of type (i), (ii), or (iii)} \\ g^{-1}(s) & - \text{if } s \text{ is in a sequence of type (iv)} \end{cases}$$

## SECTION

### II.4 Universality

We have seen a number of Turing machines, such as one whose output is the successor of its input, one that adds two input numbers, and others. These are single-purpose devices, where to get different input-output behavior we need a new machine, that is, new hardware. This was what we meant by saying that a good first take on Turing machines is that they are more like a modern computer program than a modern computer—on a modern computer, to change behavior you don't change the hardware.

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



4.1 FIGURE: ENIAC, reconfigure by rewiring.

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

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



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

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

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



An ouroboros,  
a snake swal-  
lowing its own  
tail

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

We run Turing machines by loading symbols on the tape and pressing Start. So we don't feed machines with machines; we feed them symbols, representations of things. True, we can feed  $\mathcal{U}\mathcal{P}$  a specification of itself, such as a pair  $e, x$  where  $e$  is the index number of  $\mathcal{U}\mathcal{P}$  (and  $x$  is the input), and is thus computationally equivalent to that machine's source. Even so, the universe won't collapse—we can

absolutely use a text editor to edit its own source, or ask a compiler to compile its own executable. Similarly, we can feed  $\mathcal{U}\mathcal{P}$  its own index number. Lots of interesting things happen as a result, but there is no inherent impossibility.

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

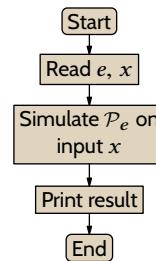
This is a **Universal Turing Machine**.<sup>†</sup> Universal machines are familiar from everyday computing. This figure<sup>‡</sup> outlines the action of a computer operating system that is given a program to run and some data for that program. Think of the program as like some Turing machine,  $\mathcal{P}_e$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

Another computing experience that is like a universal machine is an interpreter. Below is the Racket interpreter. At the first prompt the system gets the source of a routine that takes in  $i$  and totals the first  $i$  numbers. At the second prompt the interpreter runs that source with the input  $i = 4$ .

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

The most direct example of computing systems that act as universal machines is a language's eval statement. At the first prompt below we define a routine that has the interpreter evaluate the expression that is input.<sup>§</sup> In the next prompt we define a list, quoted so that it is not interpreted. This list is the source of a function; lambda (i) ... defines a function of one input,  $i$  (in contrast to the definitions of the functions sum and utm, this function does not have a name and so the definition syntax is different). In the third and fourth prompts, the interpreter evaluates that list and applies it to the numbers 5 and 0. That is, like the loom's punched cards, different inputs cause utm to have different behaviors.

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



<sup>†</sup>We could also define a Universal Turing machine to take the single-number input  $\text{cantor}(e, x)$ . <sup>‡</sup>This is a flow chart, which gives a high level sketch of a routine. We use three types of boxes. Round corner boxes are for Start and End. Rectangles are for the ordinary flow of control. Diamond boxes, which appear in later flow charts, are for decisions, if statements. <sup>§</sup>Writing a program that allows general users to evaluate arbitrary code is powerful but not safe, especially if these users just surf 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.

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

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

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

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

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

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

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

But that's silly. Programs have finite length and so can't have infinitely many cases. Restated, 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 a finitely many such differently-behaving branches. The fact that each Turing machine has only finitely many instructions imposes a condition of uniformity.

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

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

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

```
(define (sequence-of-nines-1 n)
  (if (< n n0)
    1
    0))
```

One aspect of this argument that is surprising is that neither of these two has much to do with  $\pi$ . Also surprising, and perhaps unsettling, is that we have shown that the problem is solvable without showing how to solve it. That is, there is a difference between showing that this function is computable

$$f(n) = \begin{cases} 1 & - \text{if } \pi \text{ has } n \text{ consecutive nines} \\ 0 & - \text{otherwise} \end{cases}$$

and possessing an algorithm to compute it. This observation shows that the assertion “something is computable if you can write a program for it” is naive or at least doesn’t go into enough detail to make the subtleties clear.

In contrast, consider a routine that inputs  $i \in \mathbb{N}$  and outputs the  $i$ -th decimal place of  $\pi$ . Using it, we can write a program that takes in  $n$  and steps through the digits of  $\pi$ , looking for  $n$  consecutive nines. With this approach we are constructing the answer, not just saying that it exists. This approach is also uniform in the sense that we could modify it to use other routines and so look for strings of nines in other numbers. However this approach has the disadvantage that if there is an  $n_0$  where  $\pi$  does not have  $n$  consecutive nines for  $n \geq n_0$ , then this program will search forever without discovering that.

**Parametrization** Universality says that there is a Turing machine that takes in inputs  $e$  and  $x$  and returns the same value as we would get by running  $\mathcal{P}_e$  on input  $x$ , including not halting if that machine does not halt on that input. That is, there is a computable function  $\phi: \mathbb{N}^2 \rightarrow \mathbb{N}$  such that  $\phi(e, x) = \phi_e(x)$  if  $\phi_e(x) \downarrow$  and  $\phi(e, x) \uparrow$  if  $\phi_e(x) \uparrow$ .

There, the  $e$  travels from the function’s argument to the index. We now generalize. Start with a program that takes two inputs such as this one.

```
(define (P x y)
  (+ x y))
```

Freeze the first argument. The result is a one-input program. Here we freeze  $x$  at 7 and at 8.

```
(define (P_7 y)
  (P 7 y))
```

```
(define (P_8 y)
  (P 8 y))
```

This is **partial application** because we are not freezing all of the input variables. Instead, we are **parametrizing** the variable  $x$ , resulting in a family of programs  $P_0$ ,  $P_1$ , etc.

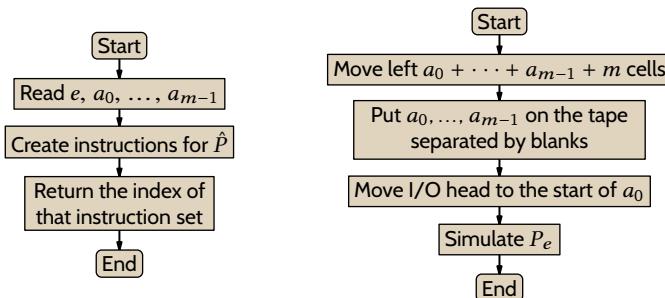
The programs in the family are related to the starting one, obviously. Denoting the function computed by the above starting program  $P$  as  $\psi(x, y) = x + y$ , partial application gives a family of functions:  $\psi_0(y) = y$ ,  $\psi_1(y) = 1 + y$ ,  $\psi_2(y) = 2 + y$ , ... The next result says that from the index of the starting program or function and from the values that are frozen, we can compute the family members.

- 4.4 THEOREM (*s*-*m*-*n* THEOREM, OR PARAMETER THEOREM) For every  $m, n \in \mathbb{N}$  there is a computable total function  $s_{m,n} : \mathbb{N}^{1+m} \rightarrow \mathbb{N}$  such that for an  $m+n$ -ary function  $\phi_e(x_0, \dots x_{m-1}, x_m, \dots x_{m+n-1})$ , freezing the initial  $m$  variables at  $a_0, \dots a_{m-1} \in \mathbb{N}$  gives the  $n$ -ary computable function  $\phi_{s(e, a_0, \dots a_{m-1})}(x_m, \dots x_{m+n-1})$ .<sup>†</sup>

*Proof* We will produce the function  $s$  to satisfy three requirements: it must be effective, it must input an index  $e$  and an  $m$ -tuple  $a_0, \dots a_{m-1}$ , and it must output the index of a machine  $\hat{P}$  that, when given the input  $x_m, \dots x_{m+n-1}$ , will return the value  $\phi_e(a_0, \dots a_{m-1}, x_m, \dots x_{m+n-1})$ , or diverge if that function diverges.

The idea is that the machine that computes  $s$  will construct the instructions for  $\hat{P}$ . We can get from the instruction set to the index using Cantor's encoding, so with that we will be done.

Below on the left is the flowchart for the machine that computes  $s$  and on the right is the flowchart for  $\hat{P}$ .



We are being flexible about the convention for input and output representations for Turing machines but to be clear, in this proof we assume that input is encoded in unary, that multiple 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.

Those flowcharts outline that the machine  $\hat{P}$  does not first read its inputs  $x_m, \dots x_{m+n-1}$ . Instead, first  $\hat{P}$  moves left and puts  $a_0, \dots a_{m-1}$  on the tape, in unary and separated by blanks, and with a blank between  $a_{m-1}$  and  $x_m$ . (Recall that the  $a_i$  are parameters, not variables. They are thus, so to speak, hard coded into  $\hat{P}$ .) Then using universality,  $\hat{P}$  simulates Turing machine  $P_e$  and lets it run on that input list.  $\square$

<sup>†</sup>This result is stated in terms of functions, not machines, because the machines need not be the same. That is, they are not equal sets of four-tuple instructions. But they will have the same input-output behavior, which means the same function.

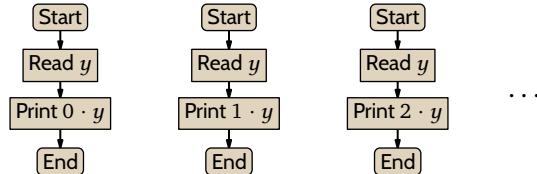
In the notation  $s_{m,n}$ , the subscript  $m$  is the number of inputs being frozen while  $n$  is the number of inputs left free. As the prior example suggests, they can sometimes be a bother and we usually omit them.

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



By Church's Thesis there is a Turing machine whose input-output behavior follows the sketch, so it computes the function  $\psi(x, y) = x \cdot y$ . Let that machine have index  $e$ .

On the left below is the flowchart sketching the machine  $\mathcal{P}_{s_{1,1}(e,0)}$ , which freezes the value of  $x$  to 0 and so computes the function  $\phi_{s_{1,1}(e,0)}(y) = 0$ . For example,  $\phi_{s_{1,1}(e,0)}(5) = 0$ . Similarly, the other two are flowcharts summarizing  $\mathcal{P}_{s_{1,1}(e,1)}$  and  $\mathcal{P}_{s_{1,1}(e,2)}$ , freezing the value of  $x$  at 1 and 2 and computing  $\phi_{s_{1,1}(e,1)}(y) = y$  and  $\phi_{s_{1,1}(e,2)}(y) = 2y$ .



In general, this is the flowchart for  $\mathcal{P}_{s_{1,1}(e,x)}$ .



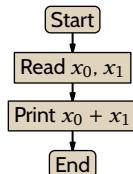
Compare this to (\*) above. This machine does not read  $x$ . Rather, thinking of these charts as programs, not Turing machines,  $x$  is hard-coded into the program body. In summary,  $\mathcal{P}_{s_{1,1}(e,x)}$  is a family of Turing machines, the first three of which are in the prior paragraph, and this family is parametrized by  $x$ . The indices are uniformly computable from  $e$  and  $x$ , using the function  $s_{1,1}$ .

The  $s\text{-}m\text{-}n$  Theorem says that we can hard code the values of parameters into the source. But it says more, namely that the resulting family of functions is uniformly computable—there is a single computable function  $s$  going from the index  $e$  and the parameter value  $x$  to the index of the result in (\*\*). So the  $s\text{-}m\text{-}n$

Theorem is about uniformity.

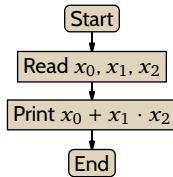
#### II.4 Exercises

- 4.6 Someone in your study group asks, “What can a Universal Turing machine do that a regular Turing machine cannot?” Help them out.
- ✓ 4.7 Has anyone ever built a Universal Turing machine, or a machine equivalent to one?
- 4.8 Can a Universal Turing machine simulate another Universal Turing machine, or for that matter can it simulate itself?
- ✓ 4.9 Your class has someone who says, “Universal Turing machines make no sense to me. How could a machine simulate another machine that has more states?” Correct their misimpression.
- 4.10 Is there more than one Universal Turing machine?
- 4.11 What happens if we feed a Universal Turing machine to itself? For instance, where the index  $e_0$  is such that  $\phi_{e_0}(e, x) = \phi_e(x)$  for all  $x$ , what is the value of  $\phi_{e_0}(e_0, 5)$ ?
- 4.12 Consider the function  $f(x_0, x_1) = 3x_0 + x_0 \cdot x_1$ .
- Freeze  $x_0$  to have the value 4. What is the resulting one-variable function?
  - Freeze  $x_0$  at 5. What is the resulting one-variable function?
  - Freeze  $x_1$  to be 0. What is the resulting function?
- 4.13 Consider  $f(x_0, x_1, x_2) = x_0 + 2x_1 + 3x_2$ .
- Freeze  $x_0$  to have the value 1. What is the resulting two-variable function?
  - What two-variable function results from fixing  $x_0$  to be 2?
  - Let  $a$  be a natural number. What two-variable function results from fixing  $x_0$  to be  $a$ ?
  - Freeze  $x_0$  at 5 and  $x_1$  at 3. What is the resulting one-variable function?
  - What one-variable function results from fixing  $x_0$  to be  $a$  and  $x_1$  to be  $b$ , for  $a, b \in \mathbb{N}$ ?
- ✓ 4.14 Suppose that the Turing machine sketched by this flowchart has index  $e$ .



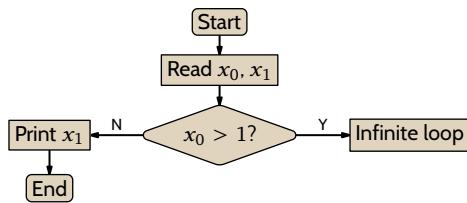
- Describe the function  $\phi_{s_{1,1}(e, 1)}$ .
- What are the values of  $\phi_{s_{1,1}(e, 1)}(0)$ ,  $\phi_{s_{1,1}(e, 1)}(1)$ , and  $\phi_{s_{1,1}(e, 1)}(2)$ ?
- Describe the function  $\phi_{s_{1,1}(e, 0)}$ .
- What are the values of  $\phi_{s_{1,1}(e, 0)}(0)$ ,  $\phi_{s_{1,1}(e, 0)}(1)$ , and  $\phi_{s_{1,1}(e, 0)}(2)$ ?

4.15 Let the Turing machine sketched by this flowchart have index  $e$ .

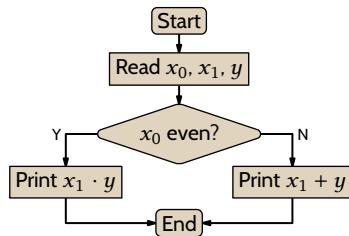


- (A) Describe the function  $\phi_{s_{1,2}(e,1)}$ .
- (B) Find  $\phi_{s_{1,2}(e,1)}(0,1)$ ,  $\phi_{s_{1,2}(e,1)}(1,0)$ , and  $\phi_{s_{1,2}(e,1)}(2,3)$
- (C) Describe the function  $\phi_{s_{2,1}(e,1,2)}$ .
- (D) Find  $\phi_{s_{2,1}(e,1,2)}(0)$ ,  $\phi_{s_{2,1}(e,1,2)}(1)$ , and  $\phi_{s_{2,1}(e,1,2)}(2)$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

✓ 4.16 Suppose that the Turing machine sketched by this flowchart has index  $e$ .



- (A) Describe  $\phi_{s_{1,1}(e,0)}$ . (B) What is  $\phi_{s_{1,1}(e,0)}(5)$ ? (C) Describe  $\phi_{s_{1,1}(e,1)}$ . (D) What is  $\phi_{s_{1,1}(e,1)}(5)$ ? (E) Describe  $\phi_{s_{1,1}(e,2)}$ . (F) What is  $\phi_{s_{1,1}(e,2)}(5)$ ?
- ✓ 4.17 Suppose that the Turing machine sketched by this flowchart has index  $e$ .



We will describe the family of functions parametrized by the arguments  $x_0$  and  $x_1$ .

- (A) Use Theorem 4.4, the  $s\text{-}m\text{-}n$  theorem, to fix  $x_0 = 0$  and  $x_1 = 3$ . Describe  $\phi_{s(e,0,3)}$ . What is  $\phi_{s(e,0,3)}(5)$ ?
  - (B) Use the  $s\text{-}m\text{-}n$  theorem to fix  $x_0 = 1$ . Describe  $\phi_{s(e,1,3)}$ . What is  $\phi_{s(e,1,3)}(5)$ ?
  - (C) Describe  $\phi_{s(e,a,b)}$ .
- ✓ 4.18 (A) Argue that the function  $\psi: \mathbb{N}^2 \rightarrow \mathbb{N}$  given by  $\psi(x,y) = 3x + y$  is computable. (B) Show that there is a family of functions  $\psi_n$  parameterized by  $n$  such that  $\psi_n(y) = 3n + y$ . Hint: take  $e \in \mathbb{N}$  such that  $\psi(x,y) = \phi_e(x,y)$ , and apply the  $s\text{-}m\text{-}n$  theorem.
- ✓ 4.19 Show that there is a total computable function  $g: \mathbb{N} \rightarrow \mathbb{N}$  such that Turing machine  $\mathcal{P}_{g(n)}$  computes the function  $y \mapsto y + n^2$ .

- ✓ 4.20 Show that there is a total computable function  $g: \mathbb{N}^2 \rightarrow \mathbb{N}$  such that Turing machine  $\mathcal{P}_{g(m,b)}$  computes  $x \mapsto mx + b$ .
- ✓ 4.21 Suppose that  $e_0$  is such that  $\phi_{e_0}$  is a Universal Turing machine, in that if given the input  $\text{cantor}(e, x)$  then it returns the same value as  $\phi_e(x)$ . Suppose also that  $e_1$  is such that  $\phi_{e_1}(x) = 4x$  for all  $x \in \mathbb{N}$ . Determine, if possible, the value of these. If it is not possible, briefly describe why not. (A)  $\phi_{e_0}(\text{cantor}(e_1, 5))$   
 (B)  $\phi_{e_1}(\text{cantor}(e_0, 5))$  (C)  $\phi_{e_0}(\text{cantor}(e_0, \text{cantor}(e_1, 5)))$   
 (D)  $\phi_{e_0}(\text{cantor } e_0, \text{cantor}(e_0, 4))$
- 4.22 Suppose that  $e_0$  is such that if  $\phi_{e_0}(\text{cantor}(e, x))$  returns the same value as  $\phi_e(x)$  (or does not converge if that function does not converge). Suppose also that  $\phi_{e_1}(x) = x + 2$  and that  $\phi_{e_2}(x) = x^2$ , for all  $x \in \mathbb{N}$ . If possible determine the value of these (if it is not possible, say why not).  
 (A)  $\phi_{e_0}(\text{cantor}(e_1, 4))$  (B)  $\phi_{e_0}(\text{cantor}(4, e_1))$  (C)  $\phi_{e_1}(\text{cantor}(e_0, \text{cantor}(e_2, 3)))$   
 (D)  $\phi_{e_0}(\text{cantor } e_0, \text{cantor}(e_0, 4))$

## SECTION

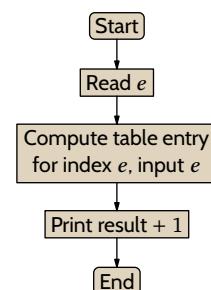
### II.5 The Halting problem

We've showed that there are functions that are not mechanically computable. We gave a counting argument, that there are countably many Turing machines but uncountably many functions and so there are functions with no associated machine. While knowing what's true is great, even better is to exhibit a specific function that is unsolvable. We will now do that.

**Definition** The natural approach to producing such a function is to go through Cantor's Theorem and effectivize it, to turn the proof into a construction.

Here is an illustrative table adapted from the discussion of Cantor's Theorem on page 77. 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 $x$							
	0	1	2	3	4	5	6	...
$\phi_0$	3	1	2	7	7	0	4	...
$\phi_1$	0	5	0	0	0	0	0	...
$\phi_2$	1	4	1	5	9	2	6	...
$\phi_3$	9	1	9	1	9	1	9	...
$\phi_4$	1	0	1	0	0	1	0	...
$\phi_5$	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., so that when the input is 0 then the output is 4, when the input is 1 then the output is 6, etc. It appears that

in the usual diagonalization way, this machine's output does not equal any of the table's rows.

However, that's a puzzle, an apparent contradiction. The flowchart outlines an effective procedure—we can implement this by using a Universal Turing machine, so its output should be one of the rows.

What's the puzzle's resolution? The program's first, second, fourth, and fifth boxes are trivial, so the issue must involve getting through the box in the middle. The answer is that there must be an  $e \in \mathbb{N}$  so that  $\phi_e(e) \uparrow$ , and for that index the Turing machine sketched in the flowchart never gets through the middle box and never prints the apparently contradictory output. That is, to avoid a contradiction the above table must contain  $\uparrow$ 's.

So we have an important insight: the fact that some computations fail to halt on some inputs is central to the nature of computation.

5.1 **DEFINITION**  $K = \{e \in \mathbb{N} \mid \phi_e(e) \downarrow\}$ , that is, Turing machine  $\mathcal{P}_e$  halts on input  $e\}$

5.2 **PROBLEM (HALTING PROBLEM)** Given  $e \in \mathbb{N}$ , determine whether  $\phi_e(e) \downarrow$ , whether Turing machine  $\mathcal{P}_e$  halts on input  $e$ .

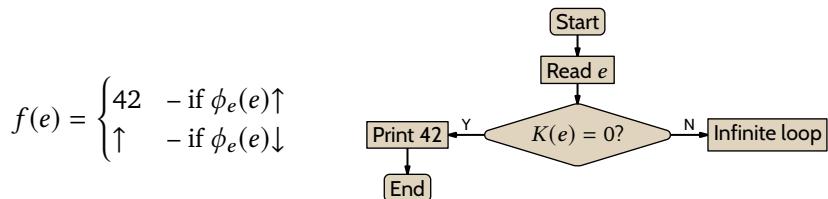
For any  $e \in \mathbb{N}$ , obviously either  $\phi_e(e) \downarrow$  or  $\phi_e(e) \uparrow$ . The Halting problem is whether we can mechanically settle which numbers are members of the set  $K$ .

5.3 **THEOREM (UNSOLVABILITY OF THE HALTING PROBLEM)** The Halting problem is mechanically unsolvable.

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

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

Then the function below is also mechanically computable. The flowchart illustrates how  $f$  is constructed; it uses the above function in its decision box.



(In  $f$ 's top case the output value 42 doesn't matter, all that matters is that  $f$  converges.) Since this function is mechanically computable, it has an index. Let that index be  $\hat{e}$ , so that  $f(x) = \phi_{\hat{e}}(x)$  for all inputs  $x$ .

Now consider  $f(\hat{e}) = \phi_{\hat{e}}(\hat{e})$  (that is, feed the machine  $\mathcal{P}_{\hat{e}}$  its own index). If it diverges then the first clause in the definition of  $f$  means that  $f(\hat{e}) \downarrow$ , which contradicts divergence. If it converges then  $f$ 's second clause means that  $f(\hat{e}) \uparrow$ ,

also impossible. Since assuming that `halt_decider` is mechanically computable leads to a contradiction, that function is not mechanically computable.  $\square$

We say that a problem is **unsolvable** if no Turing machine computes that task. If the problem is to compute the answers to ‘yes’ or ‘no’ questions, so that it is the problem of deciding membership in a set, then we will say that the set is **undecidable**. With Church’s Thesis in mind, we interpret these to mean that the problem is unsolvable by any discrete and deterministic mechanism.

**Discussion** The fact that the Halting Problem is unsolvable does not mean that for no program can we tell whether it halts. Obviously for every input this program adds 1 and then halts.

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

Nor does the unsolvability of the Halting problem mean that we cannot tell if a program does not halt. Consider this one.

```
> (define (f x)
  (display x)(newline)
  (f (+ 1 x)))
```

Once started, this routine just keeps going (here, it was interrupted with control-C).

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

Instead, the unsolvability of the Halting Problem says that there is no single program that, for all  $e$ , correctly decides in a finite time whether  $\mathcal{P}_e$  halts on input  $e$ .

That sentence contains the qualifier ‘single program’ because for any index  $e$ , either  $\mathcal{P}_e$  halts on  $e$  or else it does not. That is, given  $e$ , one of these two programs produces the right answer.

```
(define (no e)
  (display 0))
```

```
(define (yes e)
  (display 1))
```

Of course, guessing which one applies is not what we have in mind in this subject. We want uniformity. We want a program, a single effective procedure, that inputs  $e$  and outputs the right answer.

That sentence also includes the ‘finite time’ qualifier. We could perfectly well 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 for any  $e$ . If  $\mathcal{P}_e$  on input  $e$  halts then this code would discover that. But if  $\mathcal{P}_e$  on input  $e$  fails to halt then this code would not get that result in a finite time.

Thus, the unsolvability of the Halting Problem is about the non-existence of a single program that works across all indices. It speaks to uniformity, or rather, the impossibility of uniformity.

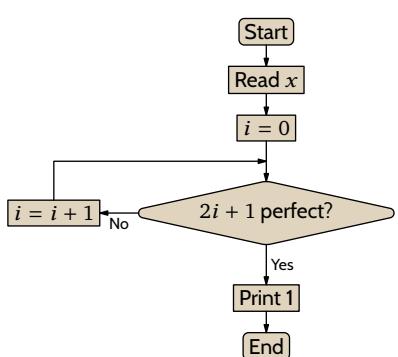
**Significance** A beginning programming class could leave the impression that if a program doesn't halt then it just has a bug, something fixable. So the Halting problem could seem to be not very interesting.

At the start of this section we argued that this impression is wrong. Imagine if we could write a utility for programmers to patch non-halting programs. After writing a program  $\mathcal{P}$  we run it through the utility `always_halt`, which modifies the source so that for any input where  $\mathcal{P}$  fails to halt, the modified program will halt (with some nominal output) but the utility does not change any outputs where  $\mathcal{P}$  does halt. That would give rise to a list of total functions like the one on page 92, and diagonalization would give a contradiction.

Thus, halting or failure to halt is inherent in the nature of computation. 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 some inputs but not on others.

That alone is enough to justify study of the Halting problem but we will give a second reason. If we had a computable function `halt_decider` then we could solve many problems that we currently don't know how to solve.

For instance, a natural number is *perfect* if it is the sum of its proper positive divisors. Thus 6 is perfect because  $6 = 1 + 2 + 3$ . Similarly,  $28 = 1 + 2 + 4 + 7 + 14$  is perfect. The next two perfect numbers are 496 and 8128. Perfect 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 this question. The program shown here searches for an odd perfect number.<sup>†</sup> 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 involving an unbounded search 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 the question by writing  $\mathcal{P}$  to search for such an  $n$ , and give the index

of  $\mathcal{P}$  to `halt_decider`.)

Before moving on, note that unbounded search is a theme in our studies. We

<sup>†</sup>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 give an output.

have seen it earlier, in defining general recursion using  $\mu$  recursion. And, it is at the heart of the Halting problem since the obvious way to test whether  $\phi_e(e) \downarrow$  is to run a brute force computation, an unbounded search, looking for a stage at which the computation halts.

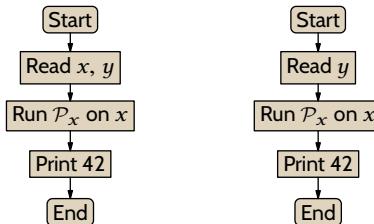
**General unsolvability** We have named one task, the Halting problem, that no mechanical computer can do. With that one in hand we are able to show that a wide class of jobs cannot be done. That is, the Halting problem is part of a larger unsolvability phenomenon.

- 5.4 EXAMPLE Consider the following problem: we want to know if a given Turing machine halts on the input 3. That is, given  $x$ , does  $\phi_x(3) \downarrow$ ?

$$\text{halts\_on\_three\_decider}(x) = \begin{cases} 1 & - \text{if } \phi_x(3) \downarrow \\ 0 & - \text{otherwise} \end{cases}$$

We will show that if `halts_on_three_decider` 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 effectively computable.

The strategy is to create a scheme where being able to determine whether an arbitrary machine halts on 3 allows us to settle questions about the Halting problem. Imagine that we have a particular  $x$  and want to know whether  $\phi_x(x) \downarrow$ . Consider the machine sketched on the right below. It reads the input  $y$  and ignores it, and also gives a nominal output. The action is in the middle box, where it simulates running  $P_x$  on input  $x$ . If that simulation halts then the machine on the right as a whole halts. If that simulation does not halt then this machine as a whole does not halt. That is, the machine on the right halts on input  $y = 3$  if and only if  $P_x$  halts on  $x$  (this is true for all other inputs  $y$  also, but we don't care). So, using this flowchart, we can leverage being able to answer questions about halting on 3 to answer questions about whether  $P_x$  halts on  $x$ .

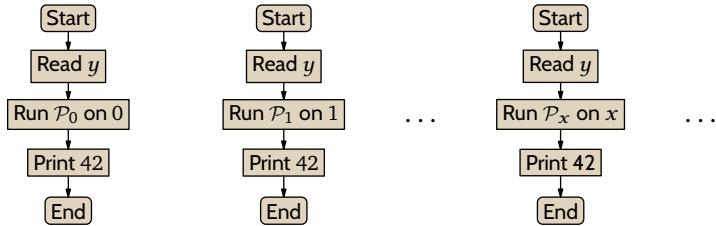


With that motivation we are ready for the argument. For contradiction, assume that `halts_on_three_decider` is mechanically computable. Consider this function.

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

Observe that  $\psi$  is mechanically computable, because it is computed by the flowchart above on the left. So by Church's Thesis there is a Turing machine whose input-output behavior is  $\psi$ . That Turing machine has some index,  $e$ , meaning that  $\psi = \phi_e$ .

Use the  $s\text{-}m\text{-}n$  theorem to parametrize  $x$ , giving  $\phi_{s(e,x)}$ . This is a family of functions, one for  $x = 0$ , one for  $x = 1$ , etc. Below is the family of associated machines. In particular, the one on the right is a repeat of the one on the right above. Notice that it has a 'Read  $y$ ' but no 'Read  $x$ ': for each of these machines, the value used in the middle box is hard-coded into its source.



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

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

The function  $s$  is computable so the assumption that `halts_on_three_decider` is also computable gives that the right side is effectively computable, which in turn implies that the Halting problem is effectively solvable, which it isn't. This contradiction means that `halts_on_three_decider` is not mechanically computable.

- 5.5 **REMARK** We emphasize that  $s(e,x)$  gives a family of infinitely many machines and computable functions in order to make the point that while  $e$  is constant (it is the index of the machine that computes  $\psi$ ), the parameter  $x$  varies. We need that for (\*).
- 5.6 **EXAMPLE** We will show that this function is not mechanically computable: given  $x$ , determine whether  $P_x$  outputs 7 for any input.

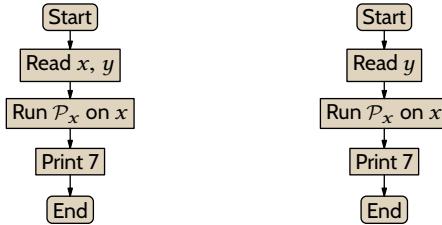
$$\text{outputs\_seven\_decider}(x) = \begin{cases} 1 & - \text{if } \phi_x(y) = 7 \text{ for some } y \\ 0 & - \text{otherwise} \end{cases}$$

Assume otherwise, that `outputs_seven_decider` is computable. 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 outlines how to compute  $\psi$ . Because it is intuitively mechanically computable, Church's Thesis says that there is a Turing machine

whose input-output behavior is  $\psi$ . That Turing machine has an index,  $e$ , so that  $\psi = \phi_e$ .



The *s-m-n* theorem gives a family of functions  $\phi_{s(e,x)}$  parametrized by  $x$ . As in the prior example, this family is associated with infinitely many different machines, one with  $x = 0$ , one with  $x = 1$ , etc. Each such machine has its  $x$  hard-coded into its source. On the right above is the flowchart for a machine whose associated function is in this family.

Then,  $\phi_x(x) \downarrow$  if and only if  $\text{outputs\_seven\_decider}(s(e,x)) = 1$ . If, as we supposed,  $\text{outputs\_seven\_decider}$  is computable, then the composition of two computable functions  $\text{outputs\_seven\_decider} \circ s$  is computable, so the Halting problem is computably solvable, which is not right. Therefore  $\text{outputs\_seven\_decider}$  is not computable.

- 5.7 EXAMPLE We next show that this problem is unsolvable: given  $x$ , determine whether  $\phi_x$  doubles its input, that is, whether  $\phi_x(y) = 2y$  for all  $y$ .

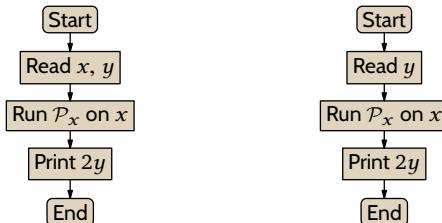
We want to show that this function is not mechanically computable.

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

Assume that it is computable. This function

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

is intuitively mechanically computable by the flowchart on the left below. So by Church's Thesis there is a Turing machine that computes it. It has some index,  $e$ .



Apply the *s-m-n* theorem to get a family of functions  $\phi_{s(e,x)}$  parametrized by  $x$ . The machine  $\mathcal{P}_{s(e,x)}$  is sketched by the flowchart on the right. Then  $\phi_x(x) \downarrow$  if and only if  $\text{outputs\_seven\_decider}(s(e,x)) = 1$ . So the supposition that `doubler_decider` is computable gives that the Halting problem is computable, which is wrong.

These examples show that the Halting problem serves as a touchstone for unsolvability: often we show that something is unsolvable by showing that if we could solve it then we could solve the Halting problem. We say that the Halting problem **reduces to** the given problem.<sup>†</sup>

Before the next subsection, three points. First and most importantly, to reiterate, saying that a problem is unsolvable means that it is unsolvable by a mechanism, that there is no Turing machine that can compute the solution to the problem. There can be functions that solve it but no computable function does.

Second, Turing and Church, independently, used the approach of this section to settle the *Entscheidungsproblem*. It is unsolvable; if there were a way to answer all mathematical questions then we could express questions about Turing machines halting in mathematics, and thereby solve the Halting problem.

The final point is a contrast: not every problem involving Turing machines is unsolvable. One solvable problem inputs an index  $e$  and decides whether  $q_0BLq_1$  is an instruction in  $\mathcal{P}_e$ . Note the contrast between this and the unsolvable problems in this section. The solvable one is searching a list for a match but the unsolvable ones are about the behavior of the computed function — about  $\phi_e$  rather than  $\mathcal{P}_e$ . This contrast is reminiscent of the difference between syntax and semantics in languages. It also recalls the opening of the first chapter where we said that we are most interested in the input-output behavior of the machines, in what they do, and less interested in things such as internal program structure.

## II.5 Exercises

5.8 Someone in your class asks the professor, “I don’t get the point of the Halting problem. If you want programs to halt then just watch them and when they exceed a set number of cycles, send a kill signal.” How to respond?

5.9 True or false: there is no function that solves the Halting Problem, that is, there is no  $f$  such that  $f(e) = 1$  if  $\phi_e(e) \downarrow$  and  $f(e) = 0$  if  $\phi_e(e) \uparrow$ .

✓ 5.10 Your study partner asks you, “The Turing machine  $\mathcal{P} = \{ q_0BBq_0, q_011q_0 \}$  fails to halt for all inputs, that’s obvious. But these unsolvability results say that I cannot know that. Why not?” Explain what they missed.

5.11 A person in your class asks, “What is wrong with this approach to solving the Halting problem? For any given Turing machine there are a finite number of states, isn’t that right? And the tape alphabet is finite, right? So there are

---

<sup>†</sup>Thus the Halting problem reduces to the problem of determining whether a given Turing machine halts on input 3. This uses ‘reduces to’ in the same sense that we would in saying, “finding the roots of a polynomial reduces to factoring that polynomial,” meaning that if we could factor then we could use that to find the roots.

only finitely many state and character pairs that can happen. As the machine runs, just monitor it for a repeat of some pair. A repeat means that the machine is looping, and so it won't halt. No repeat, no loop." What are they missing?

5.12 (This is related to the prior exercise.) Would it be possible for a computer to detect infinite loops and subsequently stop the associated process, or would implementing such logic be solving the Halting problem? Specifically, could the runtime environment do this: after each instruction is executed, it makes a snapshot of all of the relevant memory, the stack and heap data, the registers, the instruction pointer, etc., and before executing a instruction it checks its snapshot against all prior ones, and if there is a repeat then it declares that the program is in an infinite loop?

5.13 This is the [Hailstone function](#).

$$h(x) = \begin{cases} 42 & - \text{if } n = 0 \text{ or } n = 1 \\ h(n/2) & - \text{if } n \text{ is even} \\ h(3n + 1) & - \text{else} \end{cases}$$

The [Collatz conjecture](#) is that  $f$  halts on all  $x \in \mathbb{N}$ . No one knows if it is true. Is it an unsolvable problem to determine whether  $f$  halts on all input?

✓ 5.14 True or false?

- (A) The problem of determining, given  $e$ , whether  $\phi_e(3) \downarrow$  is unsolvable because no function `halts_on_three_decider` exists.
- (B) The existence of unsolvable problems indicates weaknesses in the models of computation, and we need stronger models.

5.15 A set is computable if its characteristic function is a computable function. Consider the set consisting of 1 if Mallory reached the summit of Everest, 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  $x$ , determine whether Turing machine  $\mathcal{P}_x$  runs for at least 42 steps on input 3.
- (B) Given an index  $x$ , determine whether Turing machine  $\mathcal{P}_x$  runs for at least 42 steps on input  $x$ .
- (C) Given an index  $x$ , determine whether Turing machine  $\mathcal{P}_x$  runs for at least  $x$  steps on input  $x$ .

*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  $x$ , determine if  $\phi_x$  is total, that is, if it converges on every input.
- ✓ 5.19 See the instructions above. Given an index  $x$ , decide if the Turing machine  $\mathcal{P}_x$  squares its input. That is, decide if  $\phi_x$  maps  $y \mapsto y^2$ .
- 5.20 See the instructions above. Given  $x$ , determine if the function  $\phi_x$  returns the same value on two consecutive inputs, so that  $\phi_x(y) = \phi_x(y + 1)$  for some  $y \in \mathbb{N}$ .
- ✓ 5.21 See the instructions above. Given an index  $x$ , determine whether  $\phi_x$  fails to converge on input 5.
- 5.22 See the instructions above. Given an index, determine if the computable function with that index fails to converge on all odd numbers.
- 5.23 See the instructions above. Given an index  $e$ , decide if the function  $\phi_e$  computed by machine  $\mathcal{P}_e$  is the function  $x \mapsto x + 1$ .
- 5.24 See the instructions above. Given an index  $e$ , decide if the function  $\phi_e$  fails to converge on both inputs  $x$  and  $2x$ , for some  $x$ .
- ✓ 5.25 For each problem, fill in the blanks to show that it is unsolvable.

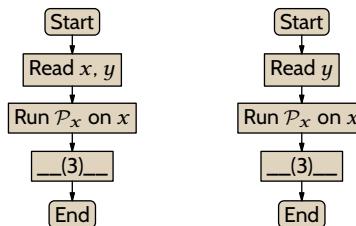
We will show that this is not mechanically computable.

$$\underline{(1)}\text{-decider}(x, y) = \begin{cases} 1 & - \text{if } \underline{(2)} \\ 0 & - \text{otherwise} \end{cases}$$

For that, consider this function.

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

The flowchart on the left shows that  $\psi$  is intuitively mechanically computable.



So by Church's Thesis there is a Turing machine with that behavior. Let that machine have index  $e$ , so that  $\psi(x, y) = \phi_e(x, y)$ . Apply the s-m-n Theorem to parametrize  $x$ . A member of the resulting family of Turing machines is sketched above, on the right. Observe that  $\phi_x(x) \downarrow \iff \underline{(1)}\text{-decider}(s(e, x)) = 1$ . Because the function  $s$  is mechanically computable, if  $\underline{(1)}\text{-decider}$  were to be mechanically computable then the right side would be mechanically computable. But the left side is not mechanically computable, by the unsolvability of the Halting problem. Therefore  $\underline{(1)}\text{-decider}$  is not mechanically computable.

- (A) Given machine index  $e$ , decide if there is an input  $y$  so that it outputs  $y$ .

- (B) Given  $e$ , decide if there is an input  $y$  so that  $\mathcal{P}_e$  outputs 42.  
 (C) Given  $e$ , decide if there is an input  $y$  so that  $\mathcal{P}_e$  outputs  $y + 2$ .

5.26 Fix integers  $a, b, c \in \mathbb{Z}$ . Consider the problem of determining, given  $\text{cantor}(x, y)$ , whether  $ax + by = c$ . Is that problem solvable or unsolvable?

5.27 For each problem, state whether it is solvable, unsolvable, or you cannot tell. You needn't give a proof, just decide. (A) Given  $x$ , decide if  $\mathcal{P}_x$  halts on all even numbers  $y$ . (B) Given  $x$ , decide if  $\mathcal{P}_x$  halts on three or fewer input  $y$ .  
 (C) Given  $x$ , decide if  $\mathcal{P}_4$  halts on input  $x$ . (D) Given  $x$ , decide if  $\mathcal{P}_x$  contains  $q_x$ .

5.28 In some ways a more natural set than  $K = \{x \in \mathbb{N} \mid \phi_x(x) \downarrow\}$  is  $K_0 = \{\langle e, x \rangle \in \mathbb{N}^2 \mid \phi_e(x) \downarrow\}$ . Use the fact that  $K$  is not computable to prove that  $K_0$  is not computable.

5.29 As stated, the Halting problem of determining membership in the set  $K = \{x \mid \phi_x(x) \downarrow\}$  cuts across all Turing machines.

- (A) Produce a single Turing machine,  $\mathcal{P}$ , such that the question of determining membership in  $\{y \mid \phi_{\mathcal{P}}(y) \downarrow\}$  is undecidable.  
 (B) Fix a number  $y$ . Show that the question of whether  $\mathcal{P}$  halts on  $y$  is decidable.

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

- (A) Given  $e$ , determine the number of states in  $\mathcal{P}_e$ .  
 (B) Given  $e$ , determine whether  $\mathcal{P}_e$  halts when the input is the empty string.  
 (C) Given  $e$ , determine if  $\mathcal{P}_e$  halts on input  $e$  within one hundred steps.

5.31 Is  $K$  infinite?

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

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

5.34 **Goldbach's conjecture** is that every even natural number greater than two is the sum of two prime numbers. 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 settle Goldbach's conjecture.

5.35 **Brocard's problem** asks whether there are any numbers for which  $n! + 1$  is a perfect square, besides 4, 5, and 7 (no other solutions exist up to a quadrillion). Show that if we could solve the Halting problem then we could settle this problem.

5.36 If we could solve the Halting problem, then could we solve all problems?

5.37 Show that most problems are unsolvable by showing that there are uncountably many functions  $f : \mathbb{N} \rightarrow \mathbb{N}$  that are not computed by any Turing machine, while the number of function that are computable is countable.

5.38 Give an example of a computable function that is total, meaning that it converges on all inputs, but whose range is not computable.

5.39 A set of bit strings is a **decidable language** if its characteristic function is computable. Prove each.

- (A) The union of two decidable languages is a decidable language.
- (B) The intersection of two decidable languages is a decidable language
- (C) The complement of a decidable language is a decidable language.

## SECTION

### II.6 Rice's Theorem

In the prior section our final point was that the results and examples there give the intuition that it is impossible to mechanically analyze the behavior of Turing machines. In this section we will make this intuition precise.

We can absolutely mechanically analyze some things about Turing machines. For instance, we can write a routine that, given  $e$ , determines whether or not  $\mathcal{P}_e$  has a state  $q_5$ . Similarly, in ordinary programming, we can write a program that parses source code for a variable named  $x1$ . But that is not what we mean by “behavior.” Instead, the variable name is a detail of the the source code’s syntax.

We are thinking about properties of machines that are independent of the internal structure of those machines. Consider a Turing machine  $\mathcal{P}$  that acts as the characteristic function of the set of primes, so it inputs numbers and outputs 1 if the number is prime, and 0 otherwise. Imagine that it has a state  $q_5$  but no state  $q_6$ . If we change the  $q_5$ ’s in its transition table to  $q_6$ ’s then we get a new machine  $\hat{\mathcal{P}}$  with a different internal structure but with the same behavior.

We say that a property is **semantic** if it has to do only with what the machine does—that is, if it has to do with  $\phi$  rather than  $\mathcal{P}$ . Other properties, such as ‘has a state  $q_5$ ’, are **syntactic**. We are interested in semantic properties.

The following definitions give one way to make these ideas precise.

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.<sup>†</sup>

6.2 **DEFINITION** A set  $\mathcal{I}$  of natural numbers is an **index set**<sup>‡</sup> when for all indices  $e, \hat{e} \in \mathbb{N}$ , if  $e \in \mathcal{I}$  and  $\phi_e \simeq \phi_{\hat{e}}$  then also  $\hat{e} \in \mathcal{I}$ .

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. For, suppose that  $e \in \mathcal{I}$  and that

<sup>†</sup> Strictly speaking we don’t need the symbol  $\simeq$ . The definition is that a function is a set of ordered pairs. If  $\phi_e(0) \downarrow$  while  $\phi_e(1) \uparrow$  then the set  $\phi_e$  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,  $\phi = \hat{\phi}$ , as they are equal sets. But we use  $\simeq$  as a reminder that the functions may be partial. <sup>‡</sup> It is called an index set because it is a set of indices.

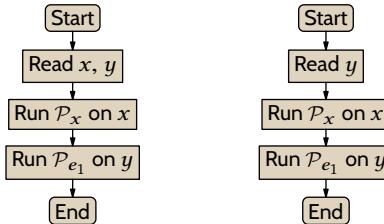
$\hat{e} \in \mathbb{N}$  is such that  $\phi_e \simeq \phi_{\hat{e}}$ . Then the behavior of  $\phi_{\hat{e}}$  is also to double its input:  $\phi_{\hat{e}}(x) = 2x$  for all  $x$ . Thus  $\hat{e} \in \mathcal{I}$  also.

- 6.4 EXAMPLE We can also get an index set by lumping 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  $x$ . Because  $\phi_e \simeq \phi_{\hat{e}}$  we know that either  $\phi_e(x) = 3x$  for all  $x$  or  $\phi_e(x) = x^3$  for all  $x$ , and thus  $\hat{e} \in \mathcal{J}$ .
- 6.5 EXAMPLE The set  $\{e \in \mathbb{N} \mid \mathcal{P}_e \text{ contains an instruction starting with } q_{10}\}$  is not an index set. We can easily produce two Turing machines having the same behavior where one machine contains such an instruction while the other does not.
- 6.6 THEOREM (RICE'S THEOREM) Every index set that is not trivial, that is not empty and not all of  $\mathbb{N}$ , is not computable.

*Proof* Let  $\mathcal{I}$  be an index set. Choose an  $e_0 \in \mathbb{N}$  so that  $\phi_{e_0}(y) \uparrow$  for all  $y$ . Then either  $e_0 \in \mathcal{I}$  or  $e_0 \notin \mathcal{I}$ . We shall show that in the second case  $\mathcal{I}$  is not computable. The first case is similar, and is Exercise 6.33.

So assume  $e_0 \notin \mathcal{I}$ . Since  $\mathcal{I}$  is not empty there is an index  $e_1 \in \mathcal{I}$ . Because  $\mathcal{I}$  is an index set,  $\phi_{e_0} \not\simeq \phi_{e_1}$ . Thus there is an input  $y$  such that  $\phi_{e_1}(y) \downarrow$ .

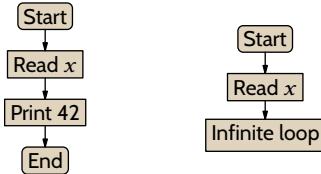
Consider the flowchart on the left below. Note that  $e_1$  is not an input, it is hard-coded into the source. By Church's Thesis there is a Turing machine with that behavior, let it be  $\mathcal{P}_{e_1}$ . Apply the  $s\text{-}m\text{-}n$  theorem to parametrize  $x$ , resulting in the uniformly computable family of functions  $\phi_{s(e,x)}$ , whose computation is outlined on the right.



We've constructed the machine sketched on the right so that if  $\phi_x(x) \uparrow$  then  $\phi_{s(e,x)} \simeq \phi_{e_0}$  and thus  $s(e,x) \notin \mathcal{I}$ . Further, if  $\phi_x(x) \downarrow$  then  $\phi_{s(e,x)} \simeq \phi_{e_1}$  and thus  $s(e,x) \in \mathcal{I}$ . Therefore if  $\mathcal{I}$  were mechanically computable, so that we could effectively check whether  $s(e,x) \in \mathcal{I}$ , then we could solve the Halting problem.  $\square$

- 6.7 EXAMPLE We 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  $I$ , and then verify that it is not empty, that it is not all of  $\mathbb{N}$ , and that it is an index set.

Consider  $\mathcal{I} = \{e \in \mathbb{N} \mid \phi_e(3) \downarrow\}$ . 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. If that machine's index is  $e_0$  then  $e_0 \in \mathcal{I}$ .



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 and where its index is  $e_1$ , we have  $e_1 \notin \mathcal{I}$ .

To finish we must verify that  $\mathcal{I}$  is an index set. Assume that  $e \in \mathcal{I}$  and let  $\hat{e} \in \mathbb{N}$  be such that  $\phi_e \simeq \phi_{\hat{e}}$ . Because  $e \in \mathcal{I}$ , we have that  $\phi_e(3) \downarrow$ . Because  $\phi_e \simeq \phi_{\hat{e}}$ , we have that  $\phi_{\hat{e}}(3) \downarrow$  also, and thus  $\hat{e} \in \mathcal{I}$ . Hence,  $\mathcal{I}$  is an index set.

The above example is the same problem as in the first example of the prior subsection. Note that Rice's Theorem makes the answer considerably simpler. (Of course, the theorem is only possible because of the prior section's work.)

- 6.8 EXAMPLE We can use Rice's Theorem to show that this problem is unsolvable: given  $e$ , decide if  $\phi_e(x) = 7$  for some  $x$ . We must produce an appropriate  $I$ , and then verify that it is nontrivial, that it is not empty and is not all of  $\mathbb{N}$ , and that it is an index set.

Let  $\mathcal{I} = \{e \in \mathbb{N} \mid \phi_e(x) = 7 \text{ for some } x\}$ . This set is not empty because, where  $e_0$  is the index of a Turing Machine that acts as the identity function  $\phi_{e_0}(x) = x$ , we have that  $e_0 \in \mathcal{I}$ . It is not all of  $\mathbb{N}$  because, where  $e_1$  is the index of a Turing Machine that never halts,  $e_1 \notin \mathcal{I}$ . So  $\mathcal{I}$  is nontrivial.

To show that  $\mathcal{I}$  is an index set, start by assuming that  $e \in \mathcal{I}$  and let  $\hat{e} \in \mathbb{N}$  be such that  $\phi_e \simeq \phi_{\hat{e}}$ . By the first assumption,  $\phi_e(x_0) = 7$  for some input  $x_0$ . By the second, the same input gives  $\phi_{\hat{e}}(x_0) = 7$ . Consequently,  $\hat{e} \in \mathcal{I}$ .

- 6.9 EXAMPLE This problem is unsolvable: determine, given an index  $e$ , whether  $\phi_e$  is this.

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

Let  $\mathcal{I} = \{j \in \mathbb{N} \mid \phi_j = f\}$ . The set  $\mathcal{I}$  is not empty because we can write a program with this behavior, and so by Church's Thesis there is a Turing machine with this behavior, and its index is a member of  $\mathcal{I}$ . Also,  $\mathcal{I} \neq \mathbb{N}$  because there is a Turing machine that fails to halt on any input, and its index is not a member of  $\mathcal{I}$ .

To finish we argue that  $\mathcal{I}$  is an index set. So suppose that  $e \in \mathcal{I}$  and that  $\phi_e \simeq \phi_{\hat{e}}$ . Because  $e \in \mathcal{I}$  we have that  $\phi_e(x) = f(x)$  for all inputs  $x$ . Because  $\phi_e \simeq \phi_{\hat{e}}$  we have that  $\phi_e(x) = \phi_{\hat{e}}(x)$  for all  $x$ , and so  $\hat{e}$  is also a member of  $\mathcal{I}$ . Hence,  $\mathcal{I}$  is an index set.

We close by reflecting on the significance of Rice's Theorem.

This result addresses the properties of computable functions. It does not speak to properties of machines unless those properties are about input-output behaviors.

For example, the set of functions computed by C programs whose first character is ‘k’ is not an index set. This brings us back to the declaration in the first paragraph of the first chapter that we are more interested in what the machines do than in the details of their internal construction.

At this chapter’s start we saw that unsolvable problems exist. However, the proof used a counting argument that did not give us natural examples. With the Halting problem we saw that there are unsolvable problems that are interesting, ones that we would naturally ask to solve.

In this section we saw that there are many unsolvable problems. The definition of index set gave us a natural way to encapsulate a behavior, and Rice’s Theorem says that *every* nontrivial index set is unsolvable. So we’ve gone from taking unsolvable problems as exotic, to taking them as things that genuinely do come up, to a point where it seems that every interesting set is unsolvable.

Of course, that’s an overstatement, an overly narrow use of the word ‘interesting’; we’ve all seen and written real-world programs with interesting behaviors. Nonetheless, Rice’s Theorem is especially significant for understanding what can be done with a computer.

## II.6 Exercises

6.10 Your friend is confused, “According to Rice’s Theorem, everything is impossible. Every property of a computer program is non-computable. But I do this supposedly impossible stuff all the time!” Help them out.

6.11 Is  $\mathcal{I} = \{ e \mid \mathcal{P}_e \text{ runs for at least 100 steps on input 5} \}$  an index set?

6.12 Why does Rice’s theorem not show that this problem is unsolvable: given  $e$ , decide whether  $\emptyset \subseteq \{ x \mid \phi_e(x) \downarrow \} \neq \emptyset$ ?

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

6.14 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  $\mathbb{N}$ .

6.15 For each problem, produce the index file needed to apply Rice’s Theorem. (You needn’t apply the theorem, just produce the file.)

(A) Given  $e$ , determine if  $\phi_e(7) = 7$  (and, of course, it converges).

(B) Given  $e$ , determine if  $\phi_e(e) = e$ .

(C) Given  $e$ , determine if  $\phi_{2e}(y) = 7$  for any  $y \in \mathbb{N}$ .

(D) Given  $e$ , determine if  $\phi_e(7)$  converges and is a prime number.

*For each of the problems from Exercise 6.16 to Exercise 6.22, show that it is unsolvable by applying Rice’s theorem. (These repeat the problems from Exercise 5.18 to Exercise 5.24.)*

- ✓ 6.16 Given an index  $x$ , determine if  $\phi_x$  is total, that is, if it converges on every input.

- ✓ 6.17 Given an index  $x$ , decide if the Turing machine  $\mathcal{P}_x$  squares its input. That is, decide if  $\phi_x$  maps  $y \mapsto y^2$ .
- 6.18 Given  $x$ , determine if the function  $\phi_x$  returns the same value on two consecutive inputs, so that  $\phi_x(y) = \phi_x(y + 1)$  for some  $y \in \mathbb{N}$ .
- 6.19 Given an index  $x$ , determine whether  $\phi_x$  fails to converge on input 5.
- 6.20 Given an index, determine if the computable function with that index fails to converge on all odd numbers.
- 6.21 Given an index  $e$ , decide if the function  $\phi_e$  computed by machine  $\mathcal{P}_e$  is  $x \mapsto x + 1$ .
- 6.22 Given an index  $e$ , decide if the function  $\phi_e$  fails to converge on both inputs  $x$  and  $2x$ , for some  $x$ .
- ✓ 6.23 Show that each of these is an unsolvable problem by applying Rice's Theorem.
  - (A) The problem of determining if a function is total, that is, converges on every input.
  - (B) The problem of determining if a function is partial, that is, fails to converge on some input.
- ✓ 6.24 For each problem, fill in the blanks to prove that it is unsolvable.

We will show that  $\mathcal{I} = \{ e \in \mathbb{N} \mid \underline{(1)} \}$  is a nontrivial index set. Then Rice's theorem will give that the problem of determining membership in  $\mathcal{I}$  is algorithmically unsolvable.

First we argue that  $\mathcal{I} \neq \emptyset$ . The routine sketched here:  $\underline{(2)}$  is intuitively computable so by Church's Thesis there is such a Turing machine. That machine's index is an element of  $\mathcal{I}$ .

Next we argue that  $\mathcal{I} \neq \mathbb{N}$ . The sketch:  $\underline{(3)}$  is intuitively computable so by Church's Thesis there is such a Turing machine. Its index is not an element of  $\mathcal{I}$ .

To finish, we show that  $\mathcal{I}$  is an index set. Suppose that  $e \in \mathcal{I}$  and that  $\hat{e}$  is such that  $\phi_e \simeq \phi_{\hat{e}}$ . Because  $e \in \mathcal{I}$ ,  $\underline{(4)}$ . Because  $\phi_e \simeq \phi_{\hat{e}}$ ,  $\underline{(5)}$ . Thus,  $\hat{e} \in \mathcal{I}$ . Consequently,  $\mathcal{I}$  is an index set.

- (A) Given  $e$ , determine if Turing machine  $e$  halts on all inputs  $x$  that are multiples of five.
- (B) Given  $e$ , decide if Turing machine  $e$  ever outputs a seven.
- 6.25 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  $x \in \mathbb{N}$ , whether  $\mathcal{P}_x$  accepts an infinite language.
  - (B) The problem of deciding, given  $e \in \mathbb{N}$ , whether  $\mathcal{P}_e$  accepts the string 101.
- 6.26 Show that this problem is mechanically unsolvable: give  $e$ , determine if there is an input  $x$  so that  $\phi_e(x) \downarrow$ .
- 6.27 We say that a Turing machine has an **unreachable state** if for all inputs, during the course of the computation the machine never enters that state. Show that  $\mathcal{J} = \{ e \mid \mathcal{P}_e \text{ has an unreachable state} \}$  is not an index set.
- 6.28 Your classmate says, "Here is a problem that is about the behavior of machines

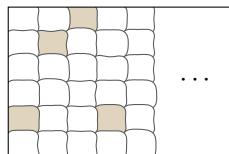
but is also solvable: given  $x$ , determine whether  $\mathcal{P}_x$  only halts on an empty input tape. To solve this problem, give machine  $\mathcal{P}_e$  an empty input and see whether it goes on.” Where are they mistaken?

6.29 Show that no set that is nonempty and finite is an index set.

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

6.31 Index sets can seem abstract. Here is an alternate characterization. The Padding Lemma on page 74 says that every computable function has infinitely many indices. Thus, there are infinitely many indices for the doubling function  $f(x) = 2x$ , infinitely many for the function that diverges on all inputs, etc. In the rectangle below imagine the set of all integers and group them together when they are indices of equal computable functions. The picture below shows such a partition. Select a few parts, such as the ones shown shaded. Take their union. That’s an index set.



More formally stated, consider the relation  $\simeq$  between natural numbers given by  $e \simeq \hat{e}$  if  $\phi_e \simeq \phi_{\hat{e}}$ . (A) Show that this is an equivalence relation. (B) Describe the parts, the equivalence classes. (C) Show that each index set is the union of some of the equivalence classes. *Hint:* show that if an index set contains one element of a class then it contains them all.

6.32 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.

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

## SECTION

### II.7 Computably enumerable sets

To attack the Halting problem the natural thing is to start by simulating  $\mathcal{P}_0$  on input 0 for a single step. Then simulate  $\mathcal{P}_0$  on input 0 for a second step and also simulate  $\mathcal{P}_1$  on input 1 for one step. After that, run  $\mathcal{P}_0$  on 0 for a third step, followed by  $\mathcal{P}_1$  on 1 for a second step, and then  $\mathcal{P}_2$  on 2 for one step. This process cycles among the  $\mathcal{P}_e$  on  $e$  simulations, running each for a step. Eventually you will

see some of these halt and the elements of  $K$  will fill in. On computer systems this interleaving is called time-slicing but in theory discussions it is called **dovetailing**.

We are listing the elements of  $K$ : first  $f(0)$ , then  $f(1), \dots$  (the computable function  $f$  is such that, for instance,  $f(0) = e$  where it happens that  $\mathcal{P}_e$  on input  $e$  is the first of these to halt). Definition 1.12 gives the terminology that a function  $f$  with domain  $\mathbb{N}$  **enumerates** its range.

Why won't this process of gradual enumeration solve the Halting problem? If  $e \in K$  then it will tell us that eventually, but if  $e \notin K$  then it will not.

- 7.1 **DEFINITION** A set of natural numbers is **computable** or **decidable** if its characteristic function is computable, so that we can effectively determine both membership and non-membership.
- 7.2 **DEFINITION** A set of natural numbers is **computably enumerable** (or **recursively enumerable**, or **c.e.**, or **r.e.**) if it is effectively listable, that is, if it is the range of a total computable function, or is the empty set.

So a set  $S$  is computable if there is a Turing machine that decides membership; this machine inputs a number  $x$  and decides either 'yes' or 'no' whether  $x \in S$ . With computably enumerable sets there is a machine that decides 'yes' but that machine need not address 'no'. Computably enumerable sets are also called **semicomputable** or **semidecidable**.

This is the natural way to computably produce sets—picture a stream of numbers  $\phi_e(0), \phi_e(1), \phi_e(2), \dots$  gradually filling out the set. (This list may contain repeats, and the numbers could appear in jumbled up order, that is, not necessarily in ascending order.)

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

*Proof* We will show that the first two are equivalent. That the second and third are equivalent is Exercise 7.34.

Assume first that  $S$  is computably enumerable. If  $S$  is empty then it is the domain of the partial computable function that diverges on all inputs. So instead assume that  $S$  is the range of a total computable  $f$ , and we will describe a computable  $g$  with domain  $S$ . Given the input  $x \in \mathbb{N}$ , to compute  $g(x)$  enumerate  $f(0), f(1), \dots$  and wait for  $x$  to appear as one of the values. If  $x$  does appear then halt the computation (and return some nominal value). If  $x$  never appears then the computation never halts.

For the other direction, assume that  $S$  is the domain of a partial computable function  $g$ , to show that it is computably enumerable. If  $S$  is empty then it is computably enumerable by definition. Otherwise we must produce a total

computable  $f$  whose range is  $S$ . If  $S$  is finite but not empty,  $S = \{s_0, \dots, s_m\}$ , then such a function is given by  $0 \mapsto s_0, \dots, m \mapsto s_m$ , and  $n \mapsto s_0$  for  $n > m$ .

Finally assume that  $S$  is infinite. Fix some  $s_0 \in S$ . Given  $n \in \mathbb{N}$ , run the computations of each of  $g(0), g(1), \dots, g(n)$  for  $n$ -many steps. Possibly some of these computations halt. Define  $f(n)$  to be the least  $k$  where  $g(k)$  halts within  $n$  steps, and so that  $k \notin \{f(0), f(1), \dots, f(n-1)\}$ . If no such  $k$  exists then define  $f(n) = \hat{s}$ ; this makes  $f$  a total function.

If  $t \notin S$  then  $g(t)$  never converges and so  $t$  is never enumerated by  $f$ . If  $s \in S$  then eventually  $g(s)$  must converge, in some number of steps,  $n_s$ . The number  $s$  is then queued for output by  $f$  in the sense that it will be enumerated by  $f$  as, at most,  $f(n_s + s)$ .  $\square$

Many authors define computably enumerable sets using the second or third items. Definition 7.2 is more natural but also more technically awkward.

7.4 **DEFINITION**  $W_e = \{y \mid \phi_e(y) \downarrow\}$

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* First, let  $S \subseteq \mathbb{N}$  be computable. We will produce an effective enumeration  $f$ . If  $S$  is finite, take  $f(0) = s_0, f(1) = s_1, \dots, f(n-1) = s_{n-1}$ , and set  $f(m) \uparrow$  for  $m \geq n$ . The other case is that  $S$  is infinite. For  $f(0)$ , find the smallest element of  $S$  by testing whether  $0 \in S$ , then whether  $1 \in S$ ,  $\dots$ . This search is effective because  $S$  is computable, and it must halt because  $S$  is infinite. Similarly,  $f(k)$  will be the  $k$ -th smallest element in  $S$ .

As to the second item, first suppose that  $S$  is computable. The prior item shows that it is computably enumerable. The complement  $S^c$  is also computable because its characteristic function is  $\mathbb{1}_{S^c} = 1 - \mathbb{1}_S$ . So the prior item shows that  $S^c$  is also computably enumerable.

Finally, suppose that both  $S$  and  $S^c$  are computably enumerable. Let  $S$  be enumerated by  $f$  and let  $S^c$  be enumerated by  $\bar{f}$ . We must give an effective procedure to determine whether a given  $x \in \mathbb{N}$  is an element of  $S$ . We will dovetail the two enumerations: first run the computation of  $f(0)$  for a step and the computation of  $\bar{f}(0)$  for a step, then run the computations of  $f(0)$  and  $\bar{f}(0)$  for a second step, etc. Eventually  $x$  will be enumerated into one or the other.  $\square$

7.6 **COROLLARY** The Halting problem set  $K$  is computably enumerable. Its complement  $K^c$  is not.

*Proof* The set  $K$  is the domain of the function  $f(x) = \phi_x(x)$ , which is mechanically computable by Church's Thesis. If the complement  $K^c$  were computably enumerable then Lemma 7.5 would imply that  $K$  is computable, but it isn't.  $\square$

That result gives one reason to be interested in computably enumerable sets,

namely that the Halting problem set  $K$  falls into the class of computably enumerable sets, as do sets such as  $\{e \mid \phi_e(3)\downarrow\}$  and  $\{e \mid \text{there is an } x \text{ so that } \phi_e(x) = 7\}$ . So this collection of sets contains lots of interesting members.

Another reason that these sets are interesting is philosophical: with Church's Thesis we can think that, in a sense, computable sets are the only sets that we will ever know, and semidecidable sets are ones that we at least half know.

## II.7 Exercises

- ✓ 7.7 You got a quiz question to define computably enumerable. A friend of yours says they answered, "A set that can be enumerated by a Turing machine but that is not computable." Is that right?
- ✓ 7.8 Produce a function that enumerates each set, whose range is the given set.  
 (A)  $\mathbb{N}$  (B) the even numbers (C) the perfect squares (D) the set  $\{5, 7, 11\}$ .
- 7.9 Produce a function that enumerates each set (A) the prime numbers (B) the natural numbers whose digits are in non-increasing order (e.g., 531 or 5331 but not 513).
- 7.10 Are there any computably enumerable sets that are infinite? Finite? Empty?  
 All of the natural numbers?
- 7.11 One of these two is computable and the other is computably enumerable but not computable. Which is which?  
 (A)  $\{e \mid \mathcal{P}_e \text{ halts on input 4 in less than twenty steps}\}$   
 (B)  $\{e \mid \mathcal{P}_e \text{ halts on input 4 in more than twenty steps}\}$
- 7.12 Which of these sets are decidable, which are semidecidable but not decidable, and which are neither? Justify in one sentence.  
 (A) The set of indices  $e$  such that  $\mathcal{P}_e$  takes more than 100 steps on input 7.  
 (B) The set of indices  $e$  such that  $\mathcal{P}_e$  takes less than 100 steps on input 7.
- 7.13 Short answer: for each set state whether it is computable, computably enumerable but not computable, or neither. (A) The set of indices  $e$  of Turing machines that contain an instruction using state  $q_4$ . (B) The set of indices of Turing machines that halt on input 3. (C) The set of indices of Turing machines that halt on input 3 in fewer than 100 steps.
- ✓ 7.14 You read someone online who says, "every countable set  $S$  is computably enumerable because if  $f: \mathbb{N} \rightarrow \mathbb{N}$  has range  $S$  then you have the enumeration  $S$  as  $f(0), f(1), \dots$ " Explain why this is wrong.
- ✓ 7.15 The set  $A_5 = \{e \mid \phi_e(5)\downarrow\}$  is clearly not computable. Show that it is computably enumerable.
- 7.16 Show that the set  $\{e \mid \phi_e(2) = 4\}$  is computably enumerable.
- 7.17 Name a set that has an enumeration but not a computable enumeration.
- 7.18 Name three sets that are computably enumerable but not computable.

- ✓ 7.19 Let  $K_0 = \{\langle e, x \rangle \mid \mathcal{P}_e \text{ halts on input } x\}$ .
  - (A) Show that it is computably enumerable.
  - (B) Show that the columns of  $K_0$ , the sets  $C_e = \{\langle e, x \rangle \mid \mathcal{P}_e \text{ halts on input } x\}$  make up all the computable enumerable sets.
- 7.20 We know that there are subsets of  $\mathbb{N}$  that are not computable. Are the computably enumerable sets the rest of the subsets?
- ✓ 7.21 Show that the set  $\text{Tot} = \{e \mid \phi_e(y) \downarrow \text{ for all } y\}$  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.1 on Unsolvability.
- 7.22 Prove that the set  $\{e \mid \phi_e(3) \uparrow\}$  is not computably enumerable.
- 7.23 Can there be a set such that the problem of determining membership in that set is unsolvable, and also the set is computably enumerable?
- 7.24 Show that the collection of computably enumerable sets is countable.
- 7.25 (A) Prove that every finite set is computably enumerable. (B) Sketch a program that takes as input a finite set and returns a function that enumerates the set.
- 7.26 Prove that every infinite computably enumerable set has an infinite computable subset.
- ✓ 7.27 Consider the function  $\text{steps}$  defined by:  $\text{steps}(e)$  is the minimal number of steps so that Turing machine  $\mathcal{P}_e$  halts if started with  $e$  on its input tape, or is undefined if the machine never halts. (A) Argue that this function is partial computable. (B) Argue that it is not total. (C) Prove that it has no total extension, no total computable  $f : \mathbb{N} \rightarrow \mathbb{N}$  so that if  $\text{steps}(e) \downarrow$  then  $\text{steps}(e) = f(e)$
- 7.28 Let  $f$  be a partial computable function that enumerates an infinite set  $R \subseteq \mathbb{N}$ . Produce a total computable function that enumerates  $R$ .
- 7.29 A set is *enumerable in increasing order* if there is a computable function  $f$  that is increasing:  $n < m$  implies  $f(n) < f(m)$ , and whose range is the set. Prove that an infinite set  $S$  is computable if and only if it is computably enumerable in increasing order.
- 7.30 A set is *computably enumerable without repetition* if it is the range of a computable function that is one-to-one. Prove that a set is computably enumerable and infinite if and only if it is computably enumerable without repetition.
- 7.31 A set is *co-computably enumerable* if its complement is computably enumerable. Produce a set that is neither computably enumerable nor co-computably enumerable.
- 7.32 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) The intersection? (D) The complement?

7.33 Computable enumerability is a property of sets so we can consider its interaction with set operations. (A) Must the union of two computably enumerable sets be computably enumerable? (B) The intersection? (C) The complement?

7.34 Finish the proof of Lemma 7.3 by showing that the second and third items are equivalent.

## SECTION

### II.8 Oracles

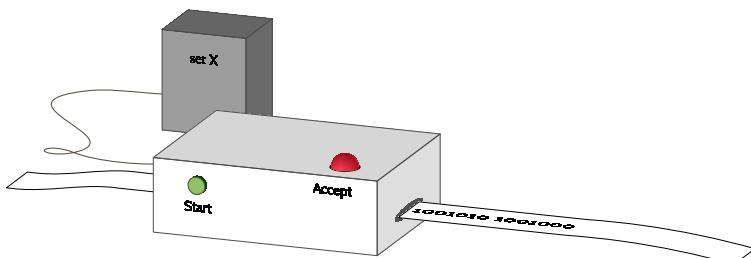
The problem of deciding whether a machine halts is so hard that it is unsolvable. Is this the absolutely hardest problem or are there ones that are even harder?

What does it mean to say that one problem is harder than another? We have compared problem hardness already, for instance when we considered the problem of whether a Turing machine halts on input 3. There we proved that if we could solve the halts-on-3 problem then we could solve the Halting problem. That is, we proved that halts-on-3 is at least as hard as the Halting problem. So, the idea is that one problem is harder than a second if solving the first would also give us a solution to the second.<sup>†</sup>

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 things that are harder than  $K$  then we need the answers to be supplied in some way that won't be a physically-realizable discrete and deterministic mechanism. Consequently, we posit a device, an **oracle**, that we attach to the Turing machine box and that acts as the characteristic function of a set. For example, to see what could be computed if we could solve the Halting problem we can attach a  $K$ -oracle that answers, "Is  $x \in K$ ?" This oracle is a black box, meaning that we can't open it to see how it works.<sup>‡</sup>



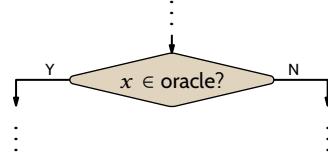
Priestess of Delphi (Collier 1891)



<sup>†</sup>We can instead think that the first problem is more general than the second. For instance, the problem of inputting a natural number and outputting its prime factors is harder than the problem of inputting a natural and determining if it is divisible by seven. Clearly if we could solve the first then we could solve the second. <sup>‡</sup>Opening it would let out the magic smoke.

We could formally define computation with an oracle  $X \subseteq \mathbb{N}$  by extending the definition of Turing machines. But we will instead describe it conceptually. Imagine adding to a programming language a Boolean function oracle, or allowing questions to an oracle in a flowchart.<sup>†</sup>

```
(if (oracle x)
  (begin
    (display "That is in the set")(newline))
  (begin
    (display "It is not in the set"))))
```



Above, we can change the oracle without changing the program code—in the diagram if we unplug the black box  $X$  oracle and replace it with a  $Y$  oracle then the white box is unchanged. Of course, the values returned by the oracle may change, which may change the outcome of running the machine, the two-box system. But the enhanced Turing machine stays the same.

Hence, besides the enhancement of the oracle, the rest of what we have previously developed about machines carries over. For instance, each white box, each oracle Turing machine, has an index. That index is source-equivalent, meaning that from an index we can compute the machine source and from the source we can find the index. Thus, to specify a relative computation, as earlier we specify which machine we are using and which inputs, and in addition we also specify the oracle set. This explains the notations for the **oracle Turing machine**,  $\mathcal{P}_e^X$ , and for the outcome of the **function computed relative to an oracle**,  $\phi_e^X(x)$ .

- 8.1 **DEFINITION** If a function computed from  $X$  is the characteristic function of the set  $S$  then we say that  $S$  is  **$X$ -computable**, or that  $S$  is **Turing reducible** to  $X$  or that  $S$  **reduces to  $X$** , denoted  $S \leq_T X$ .

That is,  $S \leq_T X$  if and only if  $\phi_e^X = \mathbb{1}_S$  for some  $e \in \mathbb{N}$ . Observe that the reduction function is total, since the characteristic function is total.

The terminology ‘ $S$  reduces to  $X$ ’ can seem confused. Think of  $S$  as a problem. Then, as the notation suggests,  $S \leq_T X$  means that problem  $S$  is no harder than problem  $X$ , so that we can solve problem  $S$  by using a solution to  $X$ .<sup>‡</sup>

- 8.2 **THEOREM** (A) A set is computable if and only if it is computable relative to the empty set, or relative to any computable set.  
 (B) (REFLEXIVITY) Every set is computable from itself,  $A \leq_T A$ .  
 (C) (TRANSITIVITY) If  $A \leq_T B$  and  $B \leq_T C$  then  $A \leq_T C$ .

*Proof* For the first item, recall that a set is computable if its characteristic function is computable. If the characteristic function is computable then it can be computed

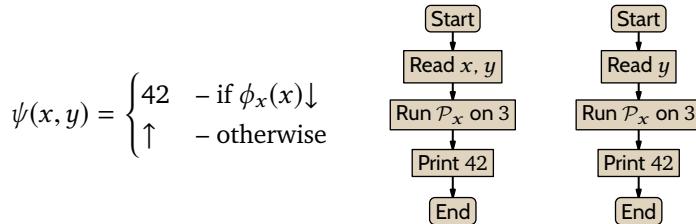
<sup>†</sup>We allow a program to use one such query, or more than one, or none at all. <sup>‡</sup>The phrase ‘reduces to’ also appears in other areas of Mathematics. For instance, in Calculus we may say that finding the area under a polynomial curve reduces to the problem of antiderivatation, because if we can antiderivate then we can compute the area.

by an oracle machine, simply by ignoring the oracle. For the other direction, suppose that the characteristic function  $\mathbb{1}_A$  can be computed by reference to the empty set or any other oracle that is computable, so  $\mathbb{1}_A$  is computed by some  $\mathcal{P}^B$  where  $\mathbb{1}_B$  is a computable function. Then compute  $\mathbb{1}_A$  without reference to an oracle by replacing any oracle calls in the  $\mathcal{P}^B$  process with computations of  $\mathbb{1}_B$ .

The second item is easy: to decide whether an  $x$  is in  $A$ , ask the  $A$  oracle and just output the answer. For the third, 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 in the computation of  $A$  from  $B$  we can replace the  $B$ -oracle calls with calls to  $\mathcal{P}_{\hat{e}}^C$ . That computes the characteristic function of  $A$  directly from  $C$ .  $\square$

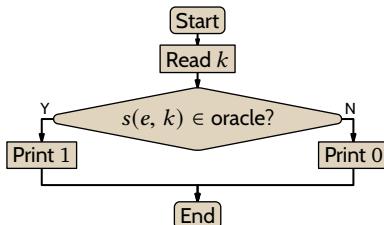
- 8.3 EXAMPLE Recall the problem of determining, given  $e$ , whether  $\mathcal{P}_e$  halts on input 3. It asks for a machine that acts as the characteristic function  $\mathbb{1}_A$  of the set  $A = \{ e \mid \mathcal{P}_e \text{ halts on 3}\}$ . We will show that  $K \leq_T A$ .

We will do it in two steps. The first is a reprise of Example 5.4. There, we considered the function  $\psi: \mathbb{N}^2 \rightarrow \mathbb{N}$  below.



It is computed by the machine sketched in the middle. Thus by Church's Thesis there is a Turing machine whose input-output behavior is  $\psi$ ; let that machine have index  $e$ . Apply the  $s$ - $m$ - $n$  theorem to get a family of machines  $\mathcal{P}_{s(e, x)}$  parametrized by  $x$ . One is sketched on the right. From the right hand sketch it is clear that the computable function  $s$  is such that for any  $k \in \mathbb{N}$ , we have  $k \in K$  if and only if  $s(e, k) \in A$ .

The second step is to build the oracle machine, the white box pictured earlier. The machine below uses the the number  $e$  fixed in the prior paragraph to compute  $K$  from an  $A$  oracle, by asking whether  $s(e, k) \in A$  (recall that the  $s$ - $m$ - $n$  function  $s$  is computable).



A comment about that flowchart, and the ones below: they are simplifications.

They all have a branch for ‘Print 1’ and another for ‘Print 0’. But because oracle machines are general Turing machines, they can have much more complex behavior. For instance, a machine could at some point write four 1’s and later blank out three of them to leave the last as the output. But this simple behavior will do for our purposes.

- 8.4 **LEMMA** Any set that is computable, including  $\emptyset$  or  $\mathbb{N}$ , is Turing reducible to any other set.

*Proof* Let  $C \subseteq \mathbb{N}$  be computable. Then there is a Turing machine that computes the characteristic function of  $C$ . Think of this as an oracle machine that never queries its oracle.  $\square$

The Halting problem is to decide whether  $\mathcal{P}_e$  halts on input  $e$ . A person may perceive that a more natural problem is to decide whether  $\mathcal{P}_e$  halts on input  $x$ .

- 8.5 **DEFINITION**  $K_0 = \{\langle e, x \rangle \mid \mathcal{P}_e \text{ halts on input } x\}$

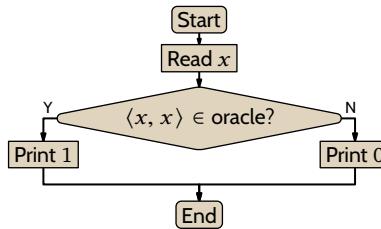
However, we will argue that the two are equivalent, that they are inter-solvable, meaning that if you can solve the one then you can solve the other. Thus your choice is a matter of convenience and convention.<sup>†</sup>

- 8.6 **DEFINITION** Two sets  $A, B$  are **Turing equivalent** or  **$T$ -equivalent**, denoted  $A \equiv_T B$ , if  $A \leq_T B$  and  $B \leq_T A$ .

Showing that two sets are  $T$ -equivalent shows that two seemingly-different problems are actually versions of the same problem. We will greatly expand on this approach in the chapter on Complexity.

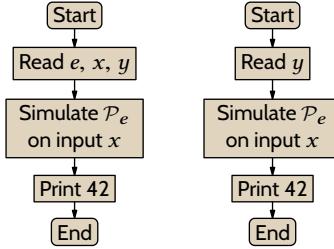
- 8.7 **THEOREM**  $K \equiv_T K_0$ .

*Proof* For  $K \leq_T K_0$ , suppose that we have access to a  $K_0$ -oracle. This will determine  $K$  from that oracle.



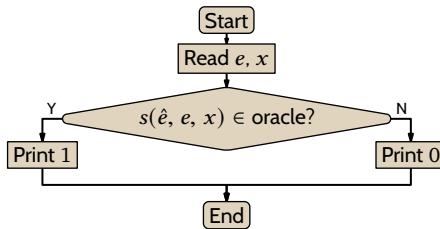
For the  $K_0 \leq_T K$  half, consider the flowchart on the left below; clearly this machine halts for all input triples exactly if  $\langle e, x \rangle \in K_0$ . By Church’s Thesis there is a Turing machine implementing it; let it be machine  $\mathcal{P}_e$ .

<sup>†</sup>For the Halting problem definition we use  $K$  because it is the standard and because it has some technical advantages, including that it falls out of the diagonalization development done at the start of this subsection.



Get the one on the right by applying the  $s\text{-}m\text{-}n$  theorem to parametrize  $e$  and  $x$ . That is, on the right is a sketch of  $\mathcal{P}_{s(\hat{e}, e, x)}$ .

Now to make the oracle machine. Given a pair  $\langle e, x \rangle$ , right-side machine  $\mathcal{P}_{s(\hat{e}, e, x)}$  either halts on all inputs  $y$  or fails to halt on all inputs, depending on whether  $\phi_e(x) \downarrow$ . In particular,  $\mathcal{P}_{s(\hat{e}, e, x)}$  halts on input  $s(\hat{e}, e, x)$  if and only if  $\phi_e(x) \downarrow$ .



(Again, recall that  $s$  is a computable function.)  $\square$

- 8.8 **COROLLARY** The **Halting** problem is at least as hard as any computably enumerable problem:  $W_e \leq_T K$  for all  $e \in \mathbb{N}$ .

*Proof* By Lemma 7.3 the computably enumerable sets are the columns of  $K_0$ .

$$W_e = \{y \mid \phi_e(y) \downarrow\} = \{y \mid \langle e, y \rangle \in K_0\}$$

So  $W_e \leq_T K_0 \equiv_T K$ .  $\square$

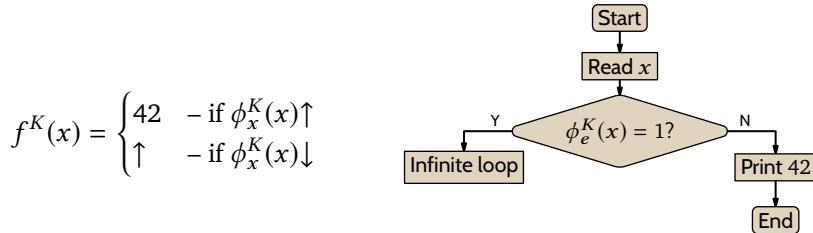
Because the **Halting** problem is in this sense the hardest of the computably enumerable problems, we say that it is **complete** among the c.e. sets.

- 8.9 **THEOREM** There is no index  $e \in \mathbb{N}$  such that  $\phi_e^K$  is the characteristic function of  $K^K = \{x \mid \phi_x^K(x) \downarrow\}$ . That is, where the **relativized Halting problem** is the problem of determining membership in  $K^K$ , its solution is not computable from a  $K$  oracle.

*Proof* Assume otherwise, that there is a mechanical computation relative to a  $K$  oracle that acts as the characteristic function of  $K^K$ .

$$\phi_e^K(x) = \begin{cases} 1 & \text{if } \phi_x^K(x) \downarrow \\ 0 & \text{otherwise} \end{cases} \quad (*)$$

We will adapt the proof that the Halting problem is unsolvable. Following that argument, we consider the function below and note that it is computable relative to a  $K$  oracle. To make that clear, the flowchart illustrates the function's construction; for the branch it uses the assumption that  $\phi_e^K$  from above is computable from  $K$ .



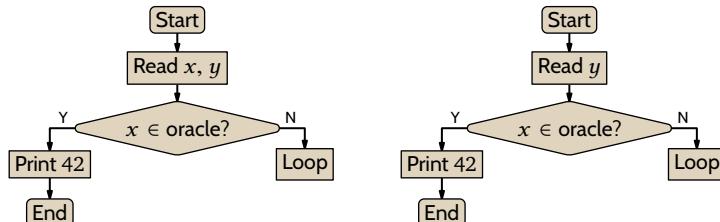
Since  $f$  is computable, it has an index. Let that index be  $\hat{e}$ , so that  $f^K = \phi_{\hat{e}}^K$ .

Now feed  $f$  its own index—that is, consider the value of  $f^K(\hat{e}) = \phi_{\hat{e}}^K(\hat{e})$ . If that diverges then the first clause in the definition of  $f$  gives that  $f^K(\hat{e}) \downarrow$ , which is a contradiction. If instead  $f^K(\hat{e})$  converges then the second clause in the definition of  $f$  gives  $f^K(\hat{e}) \uparrow$ , also impossible. Either way, assuming that  $(*)$  can be computed relative to a  $K$  oracle gives a contradiction.  $\square$

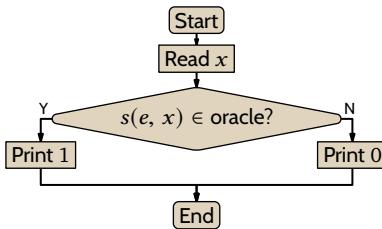
### 8.10 THEOREM Any set $S$ is reducible to its relativized Halting problem, $S \leq_T K^S$ .

This proof is an adaptation of the reducibility ones that we saw earlier in this chapter.

*Proof* On the left is an oracle machine that is intuitively mechanically computable. So Church's Thesis gives that it has an index; it is  $\mathcal{P}_e^X$  for some  $e$ . Use the  $s\text{-}m\text{-}n$  theorem to parametrize  $x$ , giving the uniformly computable family of machines  $\mathcal{P}_{s(e,x)}^X$  charted on the right.



On the right, the machine  $\mathcal{P}_{s(e,x)}^X$  halts for all inputs  $y$  if and only if  $x$  is a member of the oracle. Take the input to be  $y = s(e, x)$  and the oracle to be  $S$  to conclude that  $x \in S$  if and only if  $\phi_{s(e,x)}^S(s(e, x)) \downarrow$ , which holds if and only if  $s(e, x) \in K^S$ . So this oracle machine, which uses the same number  $e$ ,



shows that  $S \leq_T K^S$ . □

### 8.11 COROLLARY $K \leq_T K^K$ , but $K^K \not\leq_T K$

*Proof* This follows from the prior two results. □

At the start of this section we asked whether there are any problems harder than the Halting problem. We've now gotten the answer: for instance, one problem strictly harder than computing the characteristic function of  $K$  is to compute the characteristic function of  $K^K$ .

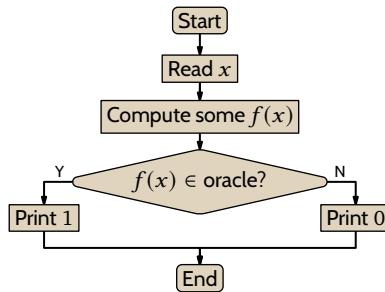
## II.8 Exercises

Recall from page 11 that a Turing machine is a decider for a set if it computes the characteristic function of that set.

- ✓ 8.12 Suppose that the set  $A$  is Turing-reducible to the set  $B$ . Which of these are true?
  - (A) A decider for  $A$  can be used to decide  $B$ .
  - (B) If  $A$  is computable then  $B$  is computable also.
  - (C) If  $A$  is uncomputable then  $B$  is uncomputable too.
- ✓ 8.13 Both oracles and deciders take in a number and return, 0 or 1, whether that number is in the set. What's the difference?
- ✓ 8.14 Your friend says, “Oracle machines are not real, so why talk about them?” What do you say?
- 8.15 Your classmate says they answered a quiz question to define an oracle with, “A set to solve unsolvable problems.” Give them a gentle critique.
- 8.16 Is there an oracle for every problem? For every problem is there an oracle?
- 8.17 A person in your class asks, “Oracles can solve unsolvable problems, right? And  $K^K$  is unsolvable. So an oracle like the  $K$  oracle should solve it.” Help your prof out here; suggest a response.
- 8.18 Your study partner confesses, “I don’t understand relative computation. Any computation using an oracle must make only finitely many oracle calls if it halts. But a finite oracle is computable, and so by Lemma 8.4 it is reducible to any set.” Give them a prompt.
- 8.19 Where  $B \subseteq \mathbb{N}$  is a set, let  $2B = \{2b \mid b \in B\}$ . We will show that  $B \equiv_T 2B$ .

- (A) Give a flowchart sketching a machine that, given access to oracle  $2B$ , will act as the characteristic function of  $B$ . That is, this machine witnesses that  $B \leq_T 2B$ .
- (B) Sketch a machine that, given access to oracle  $B$ , will act as the characteristic function of  $2B$ . This machine witnesses that  $2B \leq_T B$ .
- 8.20 We can use oracles for things other than determining the characteristic functions of sets. Sketch a machine that, with access to the oracle  $P = \{p \in \mathbb{N} \mid p \text{ is prime}\}$ , will input a number and print out a list of the primes dividing that number.
- ✓ 8.21 The set  $S = \{x \mid \phi_e(3)\downarrow \text{ and } \phi_e(4)\downarrow\}$  is not computable. Sketch how to compute it using a  $K$  oracle. That is, sketch an oracle machine that shows  $S \leq_T K$ . *Hint:* as in Example 8.3, you can use the  $s\text{-}m\text{-}n$  theorem to produce a family of machines where the  $x$ -th member halts on all inputs if and only if  $x \in S$ .
- ✓ 8.22 For the set  $S = \{e \mid \phi_e(3)\downarrow\}$ , show that  $S \leq_T K_0$ .
- ✓ 8.23 Show that  $K \leq_T \{x \mid \phi_x(y) = 2y \text{ for all input } y\}$ . *Hint:* one way is to use the  $s\text{-}m\text{-}n$  theorem to produce a family of machines where the  $x$ -th member halts on all inputs if and only if it is a doubler.
- 8.24 Consider the set  $\{x \mid \phi_x(j) = 7 \text{ for some } j\}$ .
- (A) Show that it is not computable, using Rice's theorem.
- (B) Sketch how to compute it using a  $K$  oracle. *Hint:* one way is to use the  $s\text{-}m\text{-}n$  theorem to produce a family of machines where the  $x$ -th member halts on all inputs if and only if machine  $\mathcal{P}_x$  outputs 7 on some input.
- 8.25 Let  $S = \{x \in \mathbb{N} \mid \phi_x(3)\downarrow \text{ and } \phi_{2x}(3)\downarrow \text{ and } \phi_x(3) = \phi_{2x}(3)\}$ . Show  $S \leq_T K$  by producing a way to answer questions about membership in  $S$  from a  $K$  oracle. *Hint:* one way is to apply the  $s\text{-}m\text{-}n$  theorem to produce a family of machines whose  $x$ -th member halts on all inputs if and only if  $x \in S$ .
- 8.26 Recall that a computable function  $\phi$  is total if  $\phi(y)\downarrow$  for all  $y \in \mathbb{N}$ . The set of total functions is  $\text{Tot}$ . Show that  $K \leq_T \text{Tot}$ .
- 8.27 A computable partial function  $\phi_x$  is **extensible** if there is a computable total function  $\phi$  where whenever  $\phi_x(y)\downarrow$  then the two agree,  $\phi_x(y) = \phi(y)$ . The set of extensible functions is  $\text{Ext}$ .
- (A) Show that this function is not a member of  $\text{Ext}$ : if  $x \in K$  then  $\text{steps}(x)$  is the smallest step number  $s$  where  $\mathcal{P}_x$  halts on input  $x$  by step  $s$ , and  $\text{steps}(x)\uparrow$  otherwise.
- (B) Prove that  $K \leq_T \text{Ext}$ .
- 8.28 Let  $A$  and  $B$  be sets. Show that if  $A(q) = B(q)$  for all  $q \in \mathbb{N}$  used in the oracle computation  $\phi^A(x)$  then  $\phi^A(x) = \phi^B(x)$ .
- ✓ 8.29 Prove that  $A \leq_T A^c$  for all  $A \subseteq \mathbb{N}$ .
- ✓ 8.30 Show that  $K \not\leq_T \emptyset$ .
- 8.31 Assume  $A \leq_T B$  and suppose that an outline of the oracle computation looks

like this. (This is not the general case. The actual machine might more than one test. Or, it might write a number of 1's and them conditionally erase all of them, or all but one of them.)



Decide whether each is True or False, and briefly explain.

- (A)  $A^c \leq_T B$
- (B)  $A \leq_T B^c$
- (C)  $A^c \leq_T B^c$

8.32 Is the number of oracles countable or uncountable?

8.33 Let  $A$  and  $B$  be sets. Produce a set  $C$  so that  $A \leq_T C$  and  $B \leq_T C$ .

8.34 Fix an oracle. Prove that the collection of sets computable from that oracle is countable.

8.35 The relation  $\leq_T$  involves sets, so we naturally ask how it interacts with set operations.

- (A) Does  $A \subseteq B$  imply  $A \leq_T B$ ?
- (B) Is  $A \leq_T A \cup B$ ?
- (C) Is  $A \leq_T A \cap B$ ?
- (D) Is  $A \leq_T A^c$ ?

8.36 Let  $A \subseteq \mathbb{N}$ .

- (A) Define when a set is **computably enumerable in an oracle**.
- (B) Show that  $\mathbb{N}$  is computably enumerable in  $A$  for all sets  $A$ .
- (C) Show that  $K^A$  is computably enumerable in  $A$ .

## SECTION

### II.9 Fixed point theorem

Recall our first example of diagonalization, the proof that the set of real numbers is not countable, on page 76. We assume that there is an  $f : \mathbb{N} \rightarrow \mathbb{R}$  and consider its inputs and outputs, as illustrated in this table.

$n$	$f(n)$ 's decimal expansion
0	4 2 . 3 1 2 7 7 0 4 ...
1	2 . 0 1 0 0 0 0 0 ...
2	1 . 4 1 4 1 5 9 2 ...
3	-2 0 . 9 1 9 5 9 1 9 ...
⋮	⋮

Let a decimal representation of the number on row  $n$  be  $d_n = \hat{d}.d_{n,0}d_{n,1}d_{n,2} \dots$ . Go down the diagonal to the right of the decimal point to get the sequence of digits

$\langle d_{0,0}, d_{1,1}, d_{2,2}, \dots \rangle$ . With that sequence, construct a number  $z = 0.z_0 z_1 z_2 \dots$  by making its  $n$ -th decimal place be something other than  $d_{n,n}$ . In our example we took a transformation  $t$  of digits given by  $t(d_{n,n}) = 2$  if  $d_{n,n} = 1$ , and  $t(d_{n,n}) = 1$  otherwise, so that the table above gives  $z = 0.1211 \dots$ . Then the diagonalization argument culminates in verifying that  $z$  is not any of the rows.

**When diagonalization fails** But what if the transformed diagonal is a row,  $z = f(n_0)$ ? Then the member of the array where the diagonal crosses that row is unchanged by the transformation,  $d_{n_0, n_0} = t(d_{n_0, n_0})$ . Conclusion: if diagonalization fails then the transformation has a fixed point.

We will apply this to sequences of computable functions,  $\phi_{i_0}, \phi_{i_1}, \phi_{i_2} \dots$ . We are interested in effectiveness so we don't consider arbitrary sequences of indices but instead take the indices to be computable:  $i_0, i_1, i_2 \dots = \phi_e(0), \phi_e(1), \phi_e(2) \dots$  for some  $e$ . So a sequence of computable functions has this form.

$$\phi_{\phi_e(0)}, \phi_{\phi_e(1)}, \phi_{\phi_e(2)} \dots$$

Below is a table with all such sequences, that is, all effective sequences of effective functions,  $\phi_{\phi_e(n)}$ .

		Sequence term				
		$n = 0$	$n = 1$	$n = 2$	$n = 3$	$\dots$
$e = 0$		$\phi_{\phi_0(0)}$	$\phi_{\phi_0(1)}$	$\phi_{\phi_0(2)}$	$\phi_{\phi_0(3)}$	$\dots$
$e = 1$		$\phi_{\phi_1(0)}$	$\phi_{\phi_1(1)}$	$\phi_{\phi_1(2)}$	$\phi_{\phi_1(3)}$	$\dots$
$e = 2$		$\phi_{\phi_2(0)}$	$\phi_{\phi_2(1)}$	$\phi_{\phi_2(2)}$	$\phi_{\phi_2(3)}$	$\dots$
$e = 3$		$\phi_{\phi_3(0)}$	$\phi_{\phi_3(1)}$	$\phi_{\phi_3(2)}$	$\phi_{\phi_3(3)}$	
$\vdots$		$\vdots$	$\vdots$	$\vdots$		

Each entry  $\phi_{\phi_e(n)}$  is a computable function. If the index  $\phi_e(n)$  diverges then the function as whole diverges.

The natural transformation is this, where  $f$  is a computable function.

$$\phi_x \xrightarrow{tf} \phi_{f(x)}$$

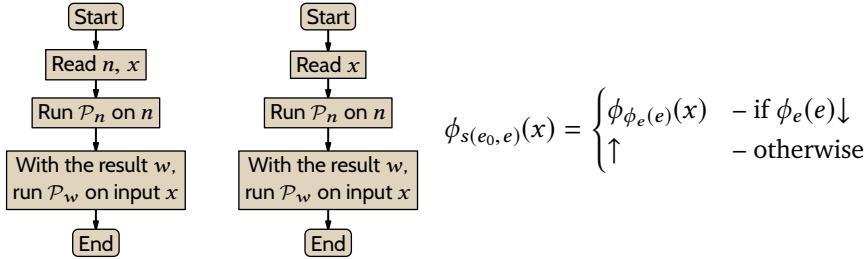
We next argue that under this transformation, diagonalization fails. Thus, the transformation  $tf$  has a fixed point.

- 9.1 THEOREM (FIXED POINT THEOREM, KLEENE 1938)<sup>†</sup> For any total computable function  $f$  there is a number  $k$  such that  $\phi_k = \phi_{f(k)}$ .

*Proof* The array diagonal is  $\phi_{\phi_0(0)}, \phi_{\phi_1(1)}, \phi_{\phi_2(2)} \dots$ . The flowchart on the left below is a sketch of a function  $f(n, x) = \phi_{\phi_n(n)}(x)$ . Church's Thesis says that some Turing

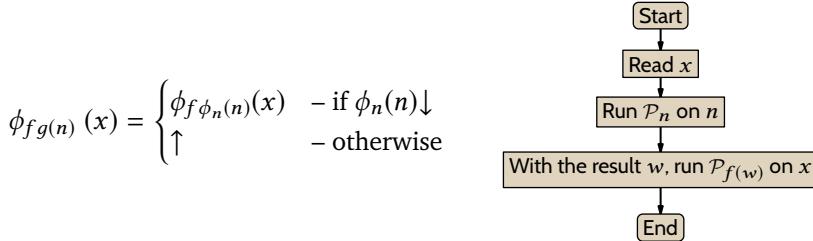
<sup>†</sup>This is also known as the Recursion Theorem but there is another widely used result of that name. This name is more descriptive so we'll go with it.

machine computes this function; let that machine have index  $e_0$ . Apply the  $s\text{-}m\text{-}n$  theorem to parametrize  $n$ , giving the right chart, which describes the family of machines. The  $n$ -th member of that family,  $\phi_{s(e_0, n)}$ , computes the  $n$ -th function on the diagonal.



The index  $e_0$  is fixed, so  $s(e_0, n)$  is a function of one variable. Let  $g(n) = s(e_0, n)$ , so that the diagonal functions are  $\phi_{g(n)}$ . This function  $g$  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 total.



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 the table's rows. Let it be row  $v$ , so that  $\phi_{fg(m)} = \phi_{\phi_v(m)}$  for all  $m$ . Consider where the diagonal sequence  $\phi_{g(n)}$  intersects that row:  $\phi_{g(v)} = \phi_{\phi_v(v)} = \phi_{fg(v)}$ . The fixed point for  $f$  is  $k = g(v)$ .  $\square$

So when we try to diagonalize out of the partial computable functions, we fail. That is, the notion of partial computable function seems to have an in-built defense against diagonalization.

The Fixed Point Theorem applies to any total computable function. Consequently, it leads to many surprising results.

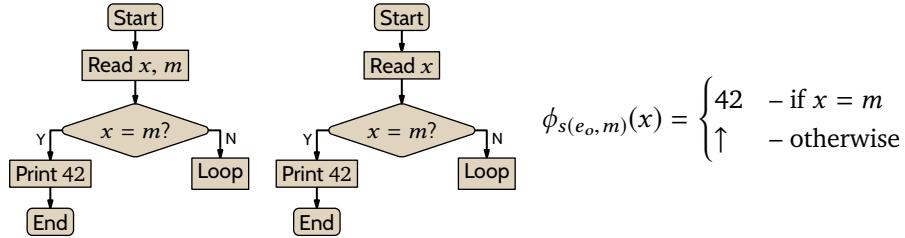
9.2 **COROLLARY** There is an index  $e$  so that  $\phi_e = \phi_{e+1}$ .

*Proof* The function  $f(x) = x + 1$  is computable and total. So there is an  $e \in \mathbb{N}$  such that  $\phi_e = \phi_{f(e)}$ .  $\square$

9.3 **COROLLARY** There is an index  $e$  such that  $P_e$  halts only on  $e$ .

*Proof* Consider the program described by the flowchart on the left. By Church's Thesis it can be done with a Turing machine,  $P_{e_0}$ . Parametrize to get the program

on the right,  $\mathcal{P}_{s(e_0, m)}$ .

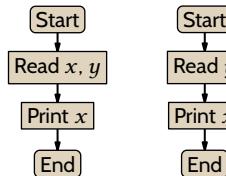


Since  $e_0$  is fixed (it is the index of the machine sketched on the left),  $s(e_0, x)$  is a total computable function of one variable,  $f(m) = s(e_0, m)$ , where the associated Turing machine halts only on input  $m$ . The Fixed Point Theorem gives a fixed point,  $\phi_{f(e)} = \phi_e$ , and the associated Turing machine  $\mathcal{P}_e$  halts only on  $e$ .  $\square$

This says that there is a Turing machine that halts only on one input, its index. Rephrased for rhetorical effect, this machine's name is its behavior.<sup>†</sup>

#### 9.4 COROLLARY There is an $m \in \mathbb{N}$ such that $\phi_m(x) = m$ for all inputs $x$ .

*Proof* Consider the function  $\psi(x, y) = x$ . As the flowchart on the left illustrates, it is computable.<sup>‡</sup> So by Church's Thesis there is a Turing machine that computes it. Let that machine have index  $e$ , so that  $\psi(x, y) = \phi_e(x, y) = x$ .



Apply the  $s\text{-}m\text{-}n$  theorem to get a family of uniformly computable functions parametrized by  $x$ , given by  $\phi_{s(e, x)}(y) = x$ . Because  $e$  is fixed, as it is the number of the Turing machine that computes  $\psi$ , define  $g: \mathbb{N} \rightarrow \mathbb{N}$  by  $g(x) = s(e, x)$ . This function is total. The Fixed Point Theorem says that there is a  $m \in \mathbb{N}$  with  $\phi_m(y) = \phi_{g(m)}(y) = \phi_{s(e, m)}(y) = m$  for all  $y$ .  $\square$

#### 9.5 REMARK Every Turing machine has some index number but here the index is related to its machine's behavior. Imagine finding that in our numbering scheme, machine $\mathcal{P}_7$ outputs 7 on all inputs. This may seem to be an accident of the choice of scheme. But it isn't an accident; the corollary says that something like this must happen for any acceptable numbering.

The Fixed Point Theorem is deep, showing surprising and interesting behaviors that occur in any sufficiently powerful computation system. For instance, since a Turing machine's index is source-equivalent, the prior result raises the question

<sup>†</sup>Here, 'name' is used as an equivalent of 'index' that is meant to be evocative. <sup>‡</sup>The flowchart is overkill here since the function is obviously computable. But when it is not obvious, as in some exercises, we need an outline of how to compute the function.

of whether there is a program that prints its own source, that self-reproduces. In addition to the discussion below, Extra C has more.

**Discussion** The Fixed Point Theorem and its proof are often considered mysterious, or at any rate obscure. Here we will expand on a few points.

One aspect that bears explication is the use-mention distinction. Compare the sentence *Atlantis is a mythical city* to *There are two a's in "Atlantis"*. In the first, we say that 'Atlantis' is **used** because it has a value, it points to something. In the second, 'Atlantis' is not referring to something — its value is itself — so we say that it is **mentioned**.<sup>†</sup>

A version of the use-mention distinction happens in computer programming, with pointers. The C language program below illustrates. The second line's asterisk means that *x* and *y* are pointers. While the compiler associates *x* and *y* with memory cells, we are not so much interested in the contents of these cells as in the contents of the memory cells that they name. The first diagram imagines that the compiler happens to associate *x* with memory address 123 and *y* with 124. It further imagines that the contents of cell 123 is the number 901 and the contents of cell 124 is 902. We say that *x* points to 901 and *y* points to 902.

The second diagram in the sequence shows the code running. Because of the  $*x = 42$ , the system puts 42 where *x* points: it does not put 42 in location 123, rather it puts 42 in the location referred to by the contents of 123, namely, cell 901.<sup>‡</sup> Then the code sets *y* to point to the same address as *x*, address 901. Finally, it puts 13 where *y* points, which is at this moment the same cell to which *x* points.



Courtesy xkcd.com

#### 9.6 ANIMATION: Pointers in a C program.

The *x* and *y* variables are being considered at different levels of meaning than ordinary variables. On one level, *x* refers to the contents of 123, while on another level it is about the contents of those contents, what's in address 901.

As to the role played by the use-mention distinction in the Fixed Point Theorem,

---

<sup>†</sup>We see this in programming books. In the sentence, “The number of players is *players*” the first ‘*players*’ refers to people while the second is a variable from the program. The typewriter font helps with the distinction. <sup>‡</sup>Using the \* operator to access the value stored at a pointer is called dereferencing that pointer. There is a matching referencing operator, &, that gives the address of an existing variable.

the proof starts by taking  $g(e)$  to be the name of this procedure.

$$\phi_{g(e)}(x) = \phi_{s(e_0, e)}(x) = \begin{cases} \phi_{\phi_e(e)}(x) & - \text{if } \phi_e(e) \downarrow \\ \uparrow & - \text{otherwise} \end{cases}$$

Don't be fooled by the notation; it is not the case that  $g(e)$  equals  $\phi_e(e)$  but instead  $g(e)$  is an index of the flowchart on the right in the proof, describing the procedure that computes the function above. Regardless of whether  $\phi_e(e) \downarrow$ , we can nonetheless compute the index  $g(n)$  and from it the instructions for the function. There is an analogy here with Atlantis—despite that the referred-to city doesn't exist we can still sensibly assert things about its name.

Informally, what  $g(e)$  names is, “Given input  $x$ , run  $\mathcal{P}_e$  on input  $e$  and if it halts with output  $w$  then run  $\mathcal{P}_w$  on input  $x$ .” Shorter: “Produce  $\phi_e(e)$  and then do  $\phi_e(e)$ .”

Next, from  $f$  we consider the composition and give it a name  $f \circ g = \phi_v$ . Substituting  $v$  into the prior paragraph gives that  $g(v)$  names, “Compute  $\phi_v(v)$  and then do  $\phi_v(v)$ .” That's the same as “Compute  $f \circ g(v)$  and then do  $f \circ g(v)$ .” Note the self-reference; it may naively appear that to compute  $g(v)$  we need to compute  $g(v)$ , that the instructions for  $g(v)$  paradoxically contains itself as a subpart.

Then  $g(v)$  first computes the name of  $f \circ g(v)$  and after that runs the machine numbered  $f \circ g(v)$ . So  $g(v)$  and  $f \circ g(v)$  are names for machines that compute the same function. Thus  $g(v)$  does not contain itself; more precisely, the set of instructions for computing  $g(v)$  does not contain itself. Instead, it contains a name for the instructions for computing itself.

## II.9 Exercises

✓ 9.7 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?

✓ 9.8 Show each. (A) There is an index  $e$  such that  $\phi_e = \phi_{e+7}$ . (B) There is an  $e$  such that  $\phi_e = \phi_{2e}$ .

9.9 Show that there must be a Turing machine  $\mathcal{P}_e$  whose input/output behavior is the same as  $\mathcal{P}_{\hat{e}}$ , where all of the digits in  $\hat{e}$  are one larger than the digits in  $e$  (except that a 9 in  $e$  changes to a 0 in  $\hat{e}$ ).

9.10 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  $x \mapsto 42$

9.11 We will prove that there is an  $m$  such that  $W_m = \{x \mid \phi_m(x) \downarrow\} = \{m^2\}$ .  
 (A) You want to show that there is a uniformly computable family of functions

like this.

$$\phi_{s(e,x)}(y) = \begin{cases} 42 & - \text{if } y = x^2 \\ \uparrow & - \text{otherwise} \end{cases}$$

Define a suitable  $\psi: \mathbb{N}^2 \rightarrow \mathbb{N}$ , argue that it is intuitively mechanically computable, and apply the *s-m-n* Theorem to get the family of  $\phi_{s(e,x)}$ .

(b) Observe that  $e$  is fixed so that  $s(e,x)$  is a function of one variable only, and name that function  $g: \mathbb{N} \rightarrow \mathbb{N}$ .

(c) Apply the Fixed Point Theorem to get the desired  $m$ .

✓ 9.12 We will show there is an index  $m$  so that  $W_m = \{y \mid \phi_m(y)\downarrow\}$  is the set consisting of one element, the  $m$ -th prime number.

(A) Argue that  $p: \mathbb{N} \rightarrow \mathbb{N}$  such that  $p(x)$  is the  $x$ -th prime is computable.

(B) Use it and the *s-m-n* Theorem to get that this family of functions is uniformly computable:  $\phi_{s(e,x)}(y) = 42$  if  $y = p(x)$  and diverges otherwise.

(C) Draw the desired conclusion.

✓ 9.13 Prove that there exists  $m \in \mathbb{N}$  such that  $W_m = \{y \mid \phi_m(y)\downarrow\} = 10^m$ .

9.14 Show there is an index  $e$  so that  $W_e = \{\phi_e(x)\downarrow\} = \{0, 1, \dots, e\}$ .

9.15 The Fixed Point Theorem says that for all  $f$  (which are computable and total) there is an  $n$  so that  $\phi_n = \phi_{f(n)}$ . What about the statement in which we flip the quantifiers: for all  $n \in \mathbb{N}$ , does there exist a total and computable function  $f: \mathbb{N} \rightarrow \mathbb{N}$  so that  $\phi_n = \phi_{f(n)}$ ?

9.16 Prove or disprove the existence of the set. (A)  $W_m = \{\phi_m(y)\downarrow\} = \mathbb{N} - \{m\}$   
 (B)  $W_m = \{x \mid \phi_m(x) \text{ diverges}\}$

9.17 Corollary 9.3 shows that there is a computable function  $\phi_n$  with domain  $\{n\}$ .

(A) Show that there is a computable function  $\phi_m$  with range  $\{m\}$ .

(B) Is there a computable function  $\phi_m$  with range  $\{2m\}$ ?

9.18 Prove that  $K$  is not an index set. Hint: use Corollary 9.3 and the Padding Lemma, Lemma 2.15.

## EXTRA

### II.A Hilbert's Hotel

A famous mathematical fable dramatizes the question of countable and uncountable sets.

Once upon a time there was an infinite hotel. The rooms were numbered 0, 1, ..., naturally. One day, when every room was occupied, someone new came to the front desk; could the hotel accommodate? The clerk hit on the right idea. They moved each guest up a room, that is, the guest in room  $n$  moved to room  $n + 1$ , leaving room 0 empty. So this hotel always has space for a new guest, or a finite number of new guests.

Next a bus rolls in with infinitely many people  $p_0, p_1, \dots$ . The clerk has the idea to move each guest to a room with twice the number, putting the guest from room  $n$  into room  $2n$ . Now the odd-numbered rooms are empty, so  $p_i$  can go in room  $2i + 1$ , and everyone has a room.

Then in rolls a convoy of buses, infinitely many of them, each with infinitely many people:  $B_0 = \{p_{0,0}, p_{0,1}, \dots\}$ , and  $B_1 = \{p_{1,0}, p_{1,1}, \dots\}$ , etc. By now the spirit is clear: move each current guest to a new room with twice the number and the new people go into the odd-numbered rooms, in the breadth-first order that we use to count  $\mathbb{N} \times \mathbb{N}$ .

After this experience the clerk may well suppose that there is always room in the infinite hotel, that it can fit any set of guests at all, with a sufficiently clever method. 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 in the hotel. One such set is  $\mathbb{R}$ .<sup>†</sup>



Plenty of empty rooms in this hotel.

## II.A Exercises

- A.1 Imagine the hotel is empty. A hundred buses arrive, where bus  $B_i$  contains passengers  $b_{i,0}, b_{i,1}, \dots$ , etc. Give a scheme for putting them in rooms.
- A.2 Give a formula assigning a room to each person from the infinite bus convoy.
- A.3 The hotel builds a parking lot. Each floor  $F_i$  has infinitely many spaces  $f_{i,0}, f_{i,1}, \dots$ . And, no surprise, there are infinitely many floors  $F_0, F_1, \dots$ . One day the hotel is empty and buses arrive, one per parking space, each with infinitely many people. Give a way to accommodate all these people.
- A.4 The management is irked that this hotel cannot fit all of the real numbers. So they announce plans for a new hotel, with a room for each  $r \in \mathbb{R}$ . Can they now cover every possible set of guests?

### EXTRA

## II.B The Halting problem in intellectual culture

The Halting problems and the related results are about limits. Interpreted in the light of Church's Thesis, they say that there are things that we cannot do. These results had an impact on the culture of mathematics, but they also had a significant impact on the wider intellectual world. We will briefly outline that impact.

Note that the statements here are in the context of the history of European intellectual culture, the context in which early Theory of Computation results appeared. A broader view is beyond our scope.

---

<sup>†</sup> 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.

With Napoleon's downfall in the early 1800's, many people in Europe felt a swing back to a sense of order and optimism, fueled by progress. For example, in the history of Turing's native England, Queen Victoria's reign from 1837 to 1901 seemed to many English commentators to be an extended period of prosperity and peace. Across wider Europe, people 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.<sup>†</sup>

In science this optimism was expressed by A A Michelson, who wrote in 1899, "The more important fundamental laws and facts of physical science have all been discovered, and these are now so firmly established that the possibility of their ever being supplanted in consequence of new discoveries is exceedingly remote."

The twentieth century physicist R Feynman has likened science to working out the rules of a game by watching it being played, "to try to understand nature is to imagine that the gods are playing some great game like chess. . . . And you don't know the rules of the game, but you're allowed to look at the board from time to time, in a little corner, perhaps. And from these observations, you try to figure out what the rules are of the game." Around the year 1900 many observers thought that we basically had got the rules and that although there might remain a couple of obscure things like castling, those would be worked out soon enough.



David Hilbert  
1862–1943

In Mathematics, this view was most famously voiced in an address given by Hilbert in 1930, "We must not believe those, who today, with philosophical bearing and deliberative tone, prophesy the fall of culture and accept the *ignorabimus*. For us there is no *ignorabimus*, and in my opinion none whatever in natural science. In opposition to the foolish *ignorabimus* our slogan shall be: We must know — we will know." ('*Ignorabimus*' means 'that which we must be forever ignorant of' or 'that thing we will never fully penetrate').<sup>‡</sup> 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.

<sup>†</sup>This is not to say that this perception is justified. Disease and poverty were rampant, colonialism and imperialism ruined the lives of millions, 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 zeitgeist included a sense of progress, of winning. <sup>‡</sup>Below we will cite some things as turning points that occur before 1930; how can that be? For one thing, cultural shifts always involve muddled timelines. For another, 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. That announcement came at the same meeting as Hilbert's speech, on the day before. Gödel was in the audience for Hilbert's address and whispered to O Taussky-Todd, "He doesn't get it."



Queen Victoria opens the Great Exhibition of the Works of Industry of All Nations, 1851

But starting in the early 1900's, that changed. Exhibit A is the picture to the right. That the modern mastery of mechanisms can have terrible effect became apparent to everyone during World War I, 1914–1918. Ten million military men died. Overall, seventeen million people died. With universal conscription, probably the men in this picture did not want to be here. They were killed by someone who probably also did not want to be here, who never knew that he killed them, and who simply entered coordinates into a firing mechanism. If you were at those coordinates, it didn't matter how brave you were, or how strong, or how right was your cause—you died. The zeitgeist shifted: Pandora's box had opened and the world is 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, which has at its heart randomness, that included the uncertainty principle, and that led to the atom bomb.

After experiments during a solar eclipse in 1919 provided strong support for his theories, Einstein became an overnight celebrity. He was seen as having changed our view of the universe from Newtonian clockwork to one where "everything is relative." His work showed that the universe has limits and that old certainties break down: nothing can travel faster than light and even the commonsense idea of two things happening at the same instant falls apart.

There were many reflections of this loss of certainty. For example, the generation of writers and artists who came of age in World War I—including Fitzgerald, Hemingway, and Stein—became known as the Lost Generation. They expressed their experience through themes of alienation, isolation, and dismay. In music, composers such as Debussy and Mahler, broke with the traditional forms in ways that were often hard for listeners—Stravinsky's *Rite of Spring* caused a near riot at its premiere in 1913. As for visual arts, the painting here shows the same themes.



S Dali's 1931 *Persistence of Memory*.



Gödel and his friend

In mathematics, much the same inversion of the standing order happened in 1930 with K Gödel's announcement of the Incompleteness Theorem. This says that if we fix a sufficiently strong formal system such as the elementary number theory of  $\mathbb{N}$  with addition and multiplication then there are statements that, while true in the system, cannot be proved in that system.<sup>†</sup> The theorem is clearly about what cannot be done—there are things that are true that can never be proved.

This statement of hard limits seemed to many observers to be especially striking

---

<sup>†</sup>Gödel produces a statement that asserts, in a coded way, "This statement cannot be proved." If false then it could be proved but false statements cannot be proved in the natural numbers. So it must be true. But then it, indeed, is true but cannot be proved to be so.

in mathematics, which had held a traditional place as the most solid of knowledge. For example, I Kant said, “I assert that in any particular natural science, one encounters genuine scientific substance only to the extent that mathematics is present.”

Gödel’s Theorem is closely related to the Halting problem. In a mathematical proof, each step must be verifiable as either an axiom or as a deduction that is valid from the prior steps. So proving a mathematical theorem is a kind of computation.<sup>†</sup> 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.

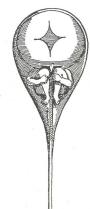
## EXTRA

### II.C Self Reproduction

Where do babies come from?

Some early investigators, working without a microscope, thought that the development of a fetus is that it basically just expands, while retaining its essential features of one head, two arms, etc. Projecting backwards, they posited a *homunculus*, a small human-like figure that when given life swells to become a person.

One issue with this hypothesis is that the person may become a parent. So inside each homunculus are its children? And the grandchildren? That is, one problem is infinite regress. Of course today we know that sperm and egg don’t contain bodies, they contain DNA, code to create bodies.



Sperm drawing, 1695

**Paley’s watch** In 1802, W Paley famously argued for the existence of a god from a perception of unexplained order in the natural world.

In crossing a heath, . . . suppose I had found a watch upon the ground . . . [W]hen we come to inspect the watch we perceive . . . that its several parts are framed and put together for a purpose, e.g., that they are so formed and adjusted as to produce motion, and that motion so regulated as to point out the hour of the day . . . the inference we think is inevitable, that the watch must have a maker—that there must have existed, at some time and at some place or other, an artificer or artificers who formed it for the purpose which we find it actually to answer, who comprehended its construction and designed its use.

The marks of design are too strong to be got over. Design must have had a designer. That designer must have been a person. That person is GOD.

---

<sup>†</sup>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.



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 argument was a very influential before the discovery by Darwin and Wallace of descent with modification through natural selection. It shows that from among all the things in the natural world to marvel at—the graceful shell of a nautilus, the precision of an eagle's eye, or consciousness—the greatest wonder for many observers was self-reproduction.

Many thinkers contended that self-reproduction had a special position, that mechanisms cannot self-reproduce. Picture a robot that assembles cars; it seemed plausible that this is possible only because the car is in some way less complex than the robot. In this line of reasoning, machines are only able to produce things that are less complex than themselves. But, that contention is wrong. The Fixed Point Theorem gives self-reproducing mechanisms.

**Quines** The Fixed Point Theorem shows that there is a number  $m$  such that  $\phi_m(x) = m$  for all inputs. Think of  $m$  as the function's name, so that this machine names itself; this is self-reference. Said another way,  $\mathcal{P}_m$ 's name is its behavior.

Since we can go effectively from the index  $m$  to the machine source, in a sense this machine knows its source. A **quine** is a program that outputs its own source code. We will next step through the nitty-gritty of making a quine.<sup>†</sup> We will use the C language since it is low-level and so the details are not hidden.

The first thing a person might think is to include the source as a string within the source code itself. Below is a start at that, which we can call `try0.c`.<sup>‡</sup> But this is obviously naive. The string would have to contain another string, etc. Like the homunculus theory, this leads to an infinite regress. Instead, we need a program that somehow contains instructions for computing a part of itself.

I MET A TRAVELER FROM AN ANTIQUE LAND WHO SAID: "I MET A TRAVELER FROM AN ANTIQUE LAND, WHO SAID: "I MET A TRAVELER FROM AN ANTIQUE LAND WHO SAID: "I MET..."



Courtesy [xkcd.com](http://xkcd.com)

```
main() {
    printf("main()\n ... ");
}
```

<sup>†</sup>The easiest such program finds its source file on the disk and prints it. That is cheating. <sup>‡</sup>The backslash-n gives a newline character.

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`.<sup>†</sup>

```
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. Just escape some newlines and quotation marks.<sup>‡</sup> This program, `try3.c`, works.

```
char *e="char*e=\"% %c; %c main() {printf(e,34,e,34,10,10);}%c";  
main(){printf(e,34,e,34,10,10);}
```

Quines are possible in any complete model of computation; the exercises ask for them in a few languages.

**Know thyself** A program that prints itself can seem to be a parlor trick. But for routines to have access to their code is useful. For example, to write a `toString(obj)` method you probably want your method to ask `obj` for its source. Another example, more nefarious, is a computer virus that transmits copies of its code to other machines.

We will show how a routine can know its source. We will start with an alternate presentation of a machine that prints itself.

First, two technical points. One is that given two programs we can combine them into one, so that we run the first and then run the second. (Similarly, given we can combine two Turing machines by renumbering the states so they do not conflict and then changing the final states in the first machine to have the number of the starting state in the second.)

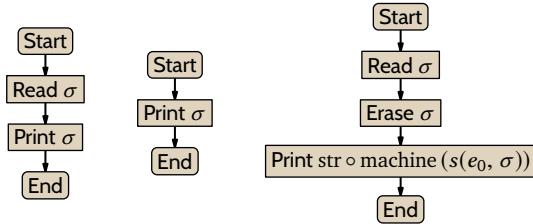
The other point is that we have fixed a numbering of Turing machines that is ‘acceptable’, meaning that there is a computable function from indices to machines and another computable function back. Write  $\mathcal{T}$  for the set of Turing machines and let the function  $\text{str}: \mathcal{T} \rightarrow \mathbb{B}^*$  input a Turing machine and return a standard bitstring representation of that machine (i.e., its source), let  $\text{machine}: \mathbb{N} \rightarrow \mathcal{T}$  input an index  $e$  and return the machine  $\mathcal{P}_e$ , and let  $\text{idx}: \mathbb{B}^* \rightarrow \mathbb{N}$  input the string representation of  $\mathcal{P}_e$  and returns the index of that machine,  $e$  (if the input string doesn’t represent a Turing machine then it doesn’t matter what this function does). Do this in such a way that  $\text{idx}$  is the inverse of the function  $\text{str} \circ \text{machine}$ .

Consider the machines sketched below. The first computes the function  $\text{echo}(\sigma) = \sigma$ . Let it have index  $e_0$  and apply the *s-m-n* Theorem to get the family

---

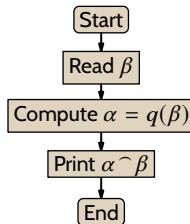
<sup>†</sup>The `char *e="..."` construct gives a string. In the C language `printf(...)` command the first argument is a string. In that string double quotes expand to single quotes, `%c` takes a character substitution from any following arguments, and `%s` takes a string substitution. <sup>‡</sup>The 10 is the ASCII encoding for newline and 34 is ASCII for a double quotation mark.

of machines sketched in the middle,  $\mathcal{P}_{s(e_0, \sigma)}$ , each of which ignores its input and just prints  $\sigma$ .



On the right,  $s(e_0, \sigma)$  is the index of the middle machine so  $\text{str} \circ \text{machine}(s(e_0, \sigma))$  is the standard string representation of the middle machine for  $\sigma$ . Thus, if  $\sigma$  is the standard representation of a Turing machine  $\mathcal{P}$  then when the machine on the right is done, the tape will contain only the standard representation of the middle machine for  $\sigma$ , the machine that ignores its input and prints out  $\sigma$ . Call the machine on the right  $\mathcal{Q}$  and call the function that it computes  $q: \Sigma^* \rightarrow \Sigma^*$ .

The machine that prints itself is a combination of two machines,  $\mathcal{A}$  and  $\mathcal{B}$ . Here's  $\mathcal{B}$ .



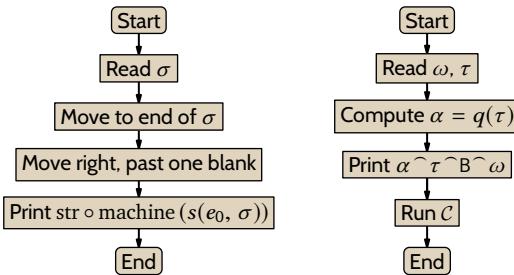
The other machine is  $\mathcal{A} = \mathcal{P}_{s(e_0, \text{str}(\mathcal{B}))}$ , which ignores anything on the tape and prints out the string representation of  $\mathcal{B}$ .

The action of the combination on an empty tape is that first the  $\mathcal{A}$  part prints out the standard string representation  $\text{str}(\mathcal{B})$ . Then  $\mathcal{B}$  reads it in as  $\beta$ , computes  $\alpha = \text{str}(\mathcal{A})$ , concatenates the two string representations  $\alpha \circ \beta$ , and prints it. This is the string representation of itself, of the combination of  $\mathcal{A}$  with  $\mathcal{B}$ .

To get a machine that computes with its own source we will extend this approach. The idea is to start with a machine  $\mathcal{C}$  that takes two inputs, a string representation of a machine and a string, and then get the desired machine  $\mathcal{D}$  that uses its own representation.

**C.1 THEOREM** For any Turing machine  $\mathcal{C}$  that computes a two-input function  $c: \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$  there is a machine  $\mathcal{D}$  that computes a one-input function  $d: \Sigma^* \rightarrow \Sigma^*$  where  $d(\omega) = c(\text{str}(\mathcal{D}), \omega)$ .

The machine  $\mathcal{D}$  is the combination of three machines,  $\mathcal{A}$ ,  $\mathcal{B}$ , and  $\mathcal{C}$ . First, as shown on the left, modify  $\mathcal{Q}$  to write its output after a string already on the tape, because we need to leave the input  $\omega$  on the tape.



Second, modify  $\mathcal{A}$  to be  $\mathcal{P}_{s(e_0, \text{str}(\mathcal{B}) \wedge \text{str}(\mathcal{C}))}$ , which ignores anything on the tape and prints out the string representation of the combination of machines  $\mathcal{B}$  and  $\mathcal{C}$ .

Next, as shown on the right, modify  $\mathcal{B}$  to input two strings,  $\omega$  and  $\tau$ . Apply  $q$  to the second to compute  $\mathcal{A}$ 's standard representation  $\alpha$ . Print out the concatenation of  $\alpha$  and  $\tau$ , then a blank, and then the input  $\omega$ . That has the form of two inputs; finish by running the machine  $\mathcal{C}$  on them.

**Verbing** English can accomplish a self-reference with, “This sentence has 32 characters.” But formal languages such as programming languages usually don’t have a self-reference operator like ‘this sentence’. The above discussion shows that no such operator is necessary. We can also use those techniques in English, as here.

Print out two copies of the following, the second in quotes: “Print out two copies of the following, the second in quotes:”

The verb ‘to quine’ means “to write a sentence fragment a first time, and then to write it a second time, but with quotation marks around it” For example, from ‘say’ we get “say ‘say’”. And, quining ‘quine’ gives “quine ‘quine’.”

In this linguistic analogy of the self-reproducing programs, the word plays the role of the data, the part played by the machine  $\mathcal{A}$  or the part played by `try3.c`'s string `char *e`. In the slogan “Produce the machine, and then do the machine,” they are the ‘produce’ part. The machine  $\mathcal{B}$  plays the role of the verb ‘quine’, and is the ‘do’ part.

**Reflections on Trusting Trust** K Thompson is one of the two main creators of the UNIX operating system. For this and other accomplishments he won the Turing Award, the highest honor in computer science. He began his acceptance address with this.

In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. . . .

More precisely stated, the problem is to write a source program that, when compiled and executed, will produce as output an exact copy of its source. If you have never done this, I urge you to try it on your own. The discovery of how to do it is a revelation that far surpasses any benefit obtained by being told how to do it. The part about “shortest” was just an incentive to demonstrate skill and determine a winner.



Ken Thompson  
b 1943

This celebrated essay develops a quine and goes on to show how the existence of such code poses a security threat that is very subtle and just about undetectable. The entire address (Thompson 1984) is widely available; everyone should read it.

## II.C Exercises

C.2 Produce a Scheme quine.

C.3 Produce a Python quine.

C.4 Consider a Scheme function `diag` that is given a string  $\sigma$  and returns a string with each instance of  $x$  in  $\sigma$  replaced with a quoted version of  $\sigma$ . Thus `diag("hello x world")` returns `hello 'hello x world' world`. Show that `print(diag('print(diag(x))'))` is a quine.

C.5 Write a program that defines a function  $f$  taking a string as input, and produces its output by applying  $f$  to its source code. For example, if  $f$  reverses the given string, then the program should outputs its source code backwards.

C.6 Write a two-level polyglot quine, a program in one language that outputs a program in a second language, which outputs the original program.

### EXTRA

## II.D Busy Beaver

Here is a try at solving the Halting problem. For any  $n \in \mathbb{N}$ , the set of Turing machines having  $n$  many tuples or fewer is finite. For some members of this set  $\mathcal{P}_e(e)$  halts and for some members it does not, but because the set is finite the list of which Turing machines halt must also be finite. Finite sets are computable. So to solve the Halting problem, given a Turing Machine  $\mathcal{P}$ , find how many instructions it has and just compute the associated finite halting information set.

The problem with this plan is uniformity, or rather lack of it — there is no single computable function that accepts inputs of the form  $\langle n, e \rangle$  and that outputs 1 if the  $n$ -instruction machine  $\mathcal{P}_e(e)$  halts, or 0 otherwise.

The natural adjustment, the uniform attack, is to start all of the machines having  $n$  or fewer instructions and dovetail their computations until no more of them will ever converge. That is, consider  $D: \mathbb{N} \rightarrow \mathbb{N}$ , where  $D(n)$  is the minimal number of steps after which all of the  $n$ -instruction machines that will ever converge have done so. We can prove that  $D$  is not computable. For, assume otherwise. Then to compute whether  $\mathcal{P}_e$  halts on input  $e$ , find how many instructions  $n$  are in the machine  $\mathcal{P}_e$ , compute  $D(n)$ , and run  $\mathcal{P}_e(e)$  for  $D(n)$ -many steps. If  $\mathcal{P}_e(e)$  has not halted by then, it never will. Of course, this contradicts the unsolvability of the Halting problem.

The function  $D$  may seem like just another uncomputable function; why is it especially enlightening? Observe that if a function  $\hat{D}$  has values larger than  $D$ , if  $\hat{D}(n) \geq D(n)$  for all sufficiently large  $n$ , then  $\hat{D}$  is also not computable. This gives

us an insight into one way that functions can fail to be computable: they can grow too fast.<sup>†</sup>



Rare moment of rest

So, which  $n$ -line program is the most productive? The **Busy Beaver problem** is: which  $n$ -state Turing Machine leaves the most 1's after halting, when started on an empty tape?

Think of this as a competition—who can write the busiest machine? To have a competition we need precise rules, which differ in unimportant ways from the conventions we have adopted in this book. So we fix a definition of Turing Machines where there is a single tape that is unbounded at one end, there are two tape symbols 1 and B, and where transitions are of the form  $\Delta(\text{state}, \text{tape symbol}) = \langle \text{state}, \text{tape symbol}, \text{head shift} \rangle$ .

**Busy Beaver is unsolvable** Write  $\Sigma(n)$  for the largest number of 1's that any  $n$  state machine, when started on a blank tape, leaves on the tape after halting. Write  $S(n)$  for the most moves, that is, transitions.

Why isn't  $\Sigma$  computable? The obvious thing is to do a breadth-first search: there are finitely many  $n$ -state machines, start them all on a blank tape, and await developments.

That won't work because some of the machines won't halt. At any moment you have some machines that have halted and you can see how many 1's are on each such tape, so you know the longest so far. But as to the not-yet-halted ones, who knows? You can by-hand see that this one or that one will never halt and so you can figure out the answer for  $n = 1$  or  $n = 2$ . But there is no algorithm to decide the question for an arbitrary number of states.

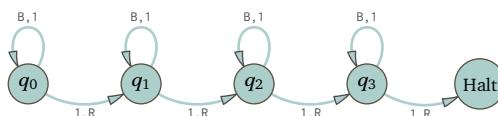


Tibor Radó 1895–  
1965

#### D.1 THEOREM (RADÓ, 1962) The function $\Sigma$ is not computable.

*Proof* Let  $f: \mathbb{N} \rightarrow \mathbb{N}$  be computable. We will show that  $\Sigma \neq f$  by showing that  $\Sigma(n) > f(n)$  for infinitely many  $n$ .

First note that there is a Turing Machine  $\mathcal{M}_j$  having  $j$  many states that writes  $j$ -many 1's to a blank tape. For instance, here is  $\mathcal{M}_4$ .



Also note that we can compose two Turing machines. The illustration below shows two machines on the left. On the right, we have combined the final states of the first machine with the start state of the second.

<sup>†</sup>Note the connection with the Ackermann function: we showed that it is not primitive recursive because it grows faster than any primitive recursive function.



Let  $F: \mathbb{N} \rightarrow \mathbb{N}$  be this function.

$$F(m) = (f(0) + 0^2) + (f(1) + 1^2) + (f(2) + 2^2) + \cdots + (f(m) + m^2)$$

It has the properties: if  $0 < m$  then  $f(m) < F(m)$ , and  $m^2 \leq F(m)$ , and  $F(m) < F(m+1)$ . It is intuitively computable so Church's Thesis says there is a Turing machine  $\mathcal{M}_F$  that computes it. Let that machine have  $n_F$  many states.

Now consider the Turing machine  $\mathcal{P}$  that performs  $\mathcal{M}_j$  and follows that with the machine  $\mathcal{M}_F$ , and then follows that with another copy of the machine  $\mathcal{M}_F$ . If started on a blank tape this machine will first produce  $j$ -many 1's, then produce  $F(j)$ -many 1's, and finally will leave the tape with  $F(F(j))$ -many 1's. Thus its productivity is  $F(F(j))$ . It has  $j + 2n_F$  many states.

Compare that with the  $j + 2n_F$ -state Busy Beaver machine. By definition  $F(F(j)) \leq \Sigma(j + 2n_F)$ . Because  $n_F$  is constant (it is the number of states in the machine  $\mathcal{M}_F$ ), the relation  $j + 2n_F \leq j^2 < F(j)$  holds for sufficiently large  $j$ . Since  $F$  is strictly increasing,  $F(j + 2n_F) < F(F(j))$ . Combining gives  $f(j + 2n_F) < F(j + 2n_F) < F(F(j)) \leq \Sigma(j + 2n_F)$ , as required.  $\square$

**What is known** That  $\Sigma(0) = 0$  and  $\Sigma(1) = 1$  follow straight from the definition. (The convention is to not count the halt state, so  $\Sigma(0)$  refers to a machine consisting only of a halting state.) Radó noted in his 1962 paper that  $\Sigma(2) = 4$ . In 1964 Radó and Lin showed that  $\Sigma(3) = 6$ .

D.2 EXAMPLE This is the three state Busy Beaver machine.

Δ	B	1
$q_0$	$q_1, 1, R$	$q_4, 1, R$
$q_1$	$q_2, 0, R$	$q_1, 1, R$
$q_2$	$q_3, 1, L$	$q_0, 1, L$

In 1983 A Brady showed that  $\Sigma(4) = 107$ . As to  $\Sigma(5)$ , even today no one knows.

Here are the current world records.

$n$	1	2	3	4	5	6
$\Sigma(n)$	1	4	6	13	$\geq 4098$	$\geq 3.5 \times 10^{18267}$
$S(n)$	1	6	21	107	$\geq 47\,176\,870$	$\geq 7.4 \times 10^{36534}$

Not only are Busy Beaver numbers very hard to compute, at some point they become impossible. In 2016, A Yedida and S Aaronson obtained an  $n$  for which  $\Sigma(n)$  is unknowable. To do that, they created a programming language where programs compile down to Turing machines. With this, they constructed a 7918-state Turing machine that halts if there is a contradiction within the standard axioms

for Mathematics, and never halts if those axioms are consistent. We believe that these axioms are consistent, so we believe that this machine doesn't halt. However, Gödel's Second Incompleteness Theorems shows that there is no way to prove the axioms are consistent using the axioms themselves, so  $\Sigma(n)$  is unknowable in that even if we were given the number  $n$ , we could not use our axioms to prove that it is right, to prove that this machine halts.

So one way for a function to fail to be computable is if it grows faster than any computable function. Note, however, that this is not the only way. There are functions that grow slower than some computable function but are nonetheless not computable.

## II.D Exercises

- ✓ D.3 Give the computation history, the sequence of configurations, that come from running the three-state Busy Beaver machine. *Hint:* you can run it on the Turing machine simulator.
- ✓ D.4 (A) How many Turing machines with tape alphabet {B, 1} are there having one state? (B) Two? (c) How many with  $n$  states?
- D.5 How many Turing machines are there, with a tape alphabet  $\Sigma$  of  $n$  characters and having  $n$  states?
- D.6 Show that there are uncomputable functions that grow slower than some computable function. *Hint:* There are uncountably many functions with output in the set  $\mathbb{B}$ .
- D.7 Give a diagonal construction of a function that is greater than any computable function.

## EXTRA

### II.E Cantor in Code

In this section we show that Cantor's correspondence between  $\mathbb{N}$  and  $\mathbb{N} \times \mathbb{N}$  is effective. The most straightforward way to show that these functions can be computed is to exhibit code.

Recall that in this table

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

the map from top row to bottom is Cantor's pairing function because it outputs pairs, while its inverse, from bottom to top, is the unpairing function.

First, unpairing. Given  $\langle x, y \rangle$ , we determine the diagonal that it lies on,

```
;; triangle-num return 1+2+3+..+n
;; natural number -> natural number
(define (triangle-num n)
  (/ (* (+ n 1)
        n)
    2))
```

and then use that to find the value.

```
;; cantor-unpairing Cantor number of the pair (x,y)
;; natural number, natural number -> natural number
(define (cantor-unpairing x y)
  (let ([d (+ x y)])
    (+ (triangle-num d)
       x)))
```

Using the function is easy.

```
$ racket
Welcome to Racket v8.3 [cs].
> (require "counting.rkt")
> (cantor-unpairing 1 1)
4
> (cantor-unpairing 34 10)
1024
>
```

A person may wonder about the choice of the elements of the pair as the arguments to `cantor-unpairing`. Perhaps it should instead input the pair itself? But the `apply` operator makes the switch easy.

```
> (cantor-unpairing 10 12)
263
> (apply cantor-unpairing '(10 12))
263
```

Next, pairing. Given a natural number  $c$ , to find the associated  $\langle x, y \rangle$ , we first find the diagonal on which it will fall. Where the diagonal is  $d(x, y) = x + y$ , let the associated triangle number be  $t(x, y) = d(d + 1)/2 = (d^2 + d)/2$ . Then  $0 = d^2 + d - 2t$ . Applying the familiar formula  $(-b \pm \sqrt{b^2 - 4ac})/(2a)$  gives this.

$$d = \frac{-1 + \sqrt{1 - 4 \cdot 1 \cdot (-2t)}}{2 \cdot 1} = \frac{-1 + \sqrt{1 + 8t}}{2}$$

(Of the ‘ $\pm$ ’, we kept only the ‘ $+$ ’ because the other root is negative.) To find the number of the diagonal containing the pair  $\langle x, y \rangle$  with  $\text{pair}(x, y) = c$ , take the floor,  $d = \lfloor (-1 + \sqrt{1 + 8c})/2 \rfloor$ .

```
;; diag-num Give number of diagonal containing Cantor pair numbered c
;; natural number -> natural number
(define (diag-num c)
  (let ([s (integer-sqrt (+ 1 (* 8 c)))]
        (floor (quotient (- s 1)
                        2))))
```

and then

```
;; cantor-pairing Given the cantor number, return the pair with that number
;; natural number -> (natural number natural number)
(define (cantor-pairing c)
  (let* ([d (diag-num c)]
        [t (triangle-num d)])
    (list (- c t)
          (- d (- c t)))))
```

Use this in the natural way.

```
> (cantor-pairing 15)
'(0 5)
> (cantor-pairing (cantor-unpairing 10 12))
'(10 12)
```

We can reproduce the table from the section's start.

```
> (for ([i '(0 1 2 3 4 5)])
  (display (cantor-pairing i))(newline))
(0 0)
(0 1)
(1 0)
(0 2)
(1 1)
(2 0)
```

Extending to triples is straightforward. These routines are perhaps misnamed—they might be better named `cantor-tupling-3` and `cantor-untupling-3`—but we will stick with what we have.

```
;; cantor-unpairing-3 Cantor number of a triple
;; natural number, natural number, natural number -> natural number
(define (cantor-unpairing-3 x0 x1 x2)
  (cantor-unpairing x0 (cantor-unpairing x1 x2)))
```

```
;; cantor-pairing-3 Return the triple for (cantor-unpairing-3 x0 x1 x2) => c
;; natural number -> (natural natural natural)
(define (cantor-pairing-3 c)
  (cons (car (cantor-pairing c))
    (cantor-pairing (cadr (cantor-pairing c)))))
```

Using these routines is also straightforward.

```
> (cantor-pairing-3 172)
'(1 2 3)
> (for ([i '(0 1 2 3 4 5 6 7 8 9)])
  (display (cantor-pairing-3 i))(newline))
(0 0 0)
(0 0 1)
(1 0 0)
(0 1 0)
(1 0 1)
(2 0 0)
(0 0 2)
(1 1 0)
(2 0 1)
(3 0 0)
```

Similar routines do four-tuples.

```
;; cantor-unpairing-4 Number quads
;; natural natural natural natural -> natural
(define (cantor-unpairing-4 x0 x1 x2 x3)
  (cantor-unpairing x0 (cantor-unpairing-3 x1 x2 x3)))
```

```
;; cantor-pairing-4 Find the quad that corresponds to the given natural
;; natural -> natural natural natural natural
(define (cantor-pairing-4 c)
  (let ((pr (cantor-pairing c)))
    (cons (car pr)
      (cantor-pairing-3 (cadr pr))))))
```

Here are the first few four-tuples.

```
> (for ([i '(0 1 2 3 4 5 6)])
  (display (cantor-pairing-4 i))(newline))
(0 0 0 0)
(0 0 0 1)
(1 0 0 0)
(0 1 0 0)
(1 0 0 1)
(2 0 0 0)
(0 0 1 0)
```

The routines for triples and four-tuples show that there is a general pattern so what the heck, let's extend to tuples of any size. We don't need these but they are fun.

For the function unpair:  $\mathbb{N}^k \rightarrow \mathbb{N}$ , which we also call cantor, we can determine  $k$  by peeking at the number of inputs. Thus cantor-unpairing-n generalizes cantor-unpairing, cantor-unpairing-3, etc., by taking a tuple of any length.

```
;; cantor-unpairing-n any-sized tuple Be cantor-unpairing-n where n is the tuple length
;; (natural ..) of n elets -> natural
(define (cantor-unpairing-n . args)
  (cond
    [(null? args) 0]
    [(= 1 (length args)) (car args)]
    [(= 2 (length args)) (cantor-unpairing (car args) (cadr args))]
    [else
      (cantor-unpairing (car args) (apply cantor-unpairing-n (cdr args))))]))
```

```
> (cantor-unpairing-n 0 0 1 0)
6
> (cantor-unpairing-n 1 2 3 4)
159331
```

To generalize to the function pair:  $\mathbb{N} \rightarrow \mathbb{N}^k$ , the difficulty is that we don't know the arity  $k$  and we must specify it separately.

```
;; cantor-pairing-arity return the list of the given arity making the cantor number c
;; If arity=0 then only c=0 is valid (others return #f)
;; natural natural -> (natural .. natural) with arity-many elements
(define (cantor-pairing-arity arity c)
  (cond
    [(= 0 arity)
     (if (= 0 c)
         '()
         '(begin
            (display "ERROR: cantor-pairing-arity with arity=0 requires c=0") (newline)
            #f))]
    [(= 1 arity) (list c)]
    [else (cons (car (cantor-pairing c))
                (cantor-pairing-arity (- arity 1) (cadr (cantor-pairing c))))]))
```

This shows the routine acting like cantor-pairing-4.

```
> (for ([i '(0 1 2 3 4 5 6)])
  (display (cantor-pairing-arity 4 i))(newline))
(0 0 0 0)
(0 0 0 1)
(1 0 0 0)
(0 1 0 0)
(1 0 0 1)
(2 0 0 0)
(0 0 1 0)
```

The cantor-pairing-arity routine is not uniform in that it covers only one arity at a time. Said another way, cantor-unpairing-arity is not the inverse of cantor-pairing-n in that we have to tell it the tuple's arity.

```
> (cantor-unpairing-n 3 4 5)
1381
> (cantor-pairing-arity 3 1381)
'(3 4 5)
```

To cover tuples of all lengths, to give a correspondence between the natural numbers and the set of sequences of natural numbers, we define two matched routines, cantor-pairing-omega and cantor-unpairing-omega.

```
> (for ([i '(\0 1 2 3 4 5 6 7 8)])
  (display (cantor-pairing-omega i))(newline))
()
()
()
()
()
()
()
()
()
()
()
()
()
()
()
()
()
()
()
```

The idea of cantor-pairing-omega is to interpret its input c as a pair  $\langle x, y \rangle$ , that is,  $c = \text{pair}(x, y)$ . It returns a tuple of length  $x + 1$ , where  $y$  is the tuple's cantor number. (The reason for the +1 in  $x + 1$  is that the empty tuple is associated with  $c = 0$ . Then rather than have all later pairs  $\langle 0, y \rangle$  not be associated with any number, we next use the one-tuple  $\langle 0 \rangle$ , and after that we use  $\langle 1 \rangle$ , etc.)

```
; ; cantor-pairing-omega Inverse of cantor-unpairing-omega (but with arguments inserted)
; ; natural -> (natural .. )
(define (cantor-pairing-omega c)
  (let* ([pr (cantor-pairing c)]
         [a (car pr)]
         [cantor-number (cadr pr)])
    (cond
      [(and (= a 0)
            (= cantor-number 0)) '()]
      [ (= a 0) (list (- cantor-number 1))]
      [else (cantor-pairing-arity (+ 1 a) cantor-number)])))
```

```
; ; cantor-unpairing-omega encode the arity in the first component
; ; natural natural .. -> natural
(define (cantor-unpairing-omega . tuple)
  (let ([arity (length tuple)])
    (cond
      [= arity 0] (cantor-unpairing 0 0))
      [= arity 1] (cantor-unpairing 0 (+ 1 (car tuple))))
      [else
        (let ([newtuple (list (- arity 1)
                             (apply cantor-unpairing-n tuple))])
          (apply cantor-unpairing newtuple))))
```

This shows their use.

```
> (cantor-unpairing-omega 1 2 3 4)
12693741448
> (cantor-pairing-omega 12693741448)
'(1 2 3 4)
```

## II.E Exercises

- E.1 What is the pair with Cantor number 42?
- E.2 What is the pair with the number 666?
- E.3 What is the formula for the gaps between one-tuples?
- E.4 What is the first number matched by cantor-pairing-omega with a four-tuple?

Part Two  
**Automata**



# CHAPTER

## III Languages

Turing machines input strings and output strings, sequences of tape symbols. So a natural way to work is to represent a problem as a string, put it on the tape, run a computation, and end with the solution as a string.

Everyday computers work the same way. Consider a program that finds the shortest driving distance between cities. Probably we work by inputting the map distances as a strings of symbols and inputting the desired two cities as two strings, and after running the program we have the output directions as a string. So strings, and collections of strings, are essential.

### 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.<sup>†</sup> 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  $\Sigma$ , although an exception is the alphabet of bits,  $\mathbb{B} = \{0, 1\}$ . A **string** over an alphabet is a sequence of symbols from that alphabet. We use lower case Greek letters such as  $\sigma$  and  $\tau$  to denote strings. We use  $\varepsilon$  to denote the empty string, the length zero sequence of symbols. The set of all strings over  $\Sigma$  is  $\Sigma^*$ .<sup>‡</sup>

- 1.1 **DEFINITION** A **language**  $\mathcal{L}$  over an alphabet  $\Sigma$  is a set of strings drawn from that alphabet. That is,  $\mathcal{L} \subseteq \Sigma^*$ .
- 1.2 **EXAMPLE** The set of bitstrings that begin with 1 is  $\mathcal{L} = \{1, 10, 11, 100, \dots\}$ .
- 1.3 **EXAMPLE** Another language over  $\mathbb{B}$  is the finite set  $\{1000001, 1100001\}$ .
- 1.4 **EXAMPLE** Let  $\Sigma = \{a, b\}$ . The language consisting of strings where the number of a's is twice the number of b's is  $\mathcal{L} = \{\varepsilon, aab, aba, baa, aaaabb, \dots\}$ .
- 1.5 **EXAMPLE** Let  $\Sigma = \{a, b, c\}$ . The language of length-two strings over that alphabet is  $\mathcal{L} = \Sigma^2 = \{aa, ab, ba, \dots, cc\}$ . Over the same alphabet, this language consists

---

IMAGE: *The Tower of Babel*, by Pieter Bruegel the Elder (1563) <sup>†</sup>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. <sup>‡</sup>For more on strings see the Appendix on page 368.

of string of length three that are in ascending order.

$$\{ \text{aaa}, \text{bbb}, \text{ccc}, \text{aab}, \text{aac}, \text{abb}, \text{abc}, \text{acc}, \text{bbc}, \text{bcc} \}$$

- 1.6 **DEFINITION** A **palindrome** is a string that reads the same forwards as backwards.

Some words from English that are palindromes are ‘kayak’, ‘noon’, and ‘racecar’.

- 1.7 **EXAMPLE** The language of palindromes over  $\Sigma = \{ \text{a}, \text{b} \}$  is  $\mathcal{L} = \{ \sigma \in \Sigma^* \mid \sigma = \sigma^R \}$ . A few members are abba, aaabaaa, and a.

- 1.8 **EXAMPLE** Let  $\Sigma = \{ \text{a}, \text{b}, \text{c} \}$ . Pythagorean triples  $\langle i, j, k \rangle \in \mathbb{N}^3$  are those where  $i^2 + j^2 = k^2$ . A few such triples are  $\langle 3, 4, 5 \rangle$ ,  $\langle 5, 12, 13 \rangle$ , and  $\langle 8, 15, 17 \rangle$ . One way to describe Pythagorean triples is with this language.

$$\begin{aligned} \mathcal{L} &= \{ \text{a}^i \text{b}^j \text{c}^k \in \Sigma^* \mid i, j, k \in \mathbb{N} \text{ and } i^2 + j^2 = k^2 \} \\ &= \{ \text{aaabbbbccccc} = \text{a}^3 \text{b}^4 \text{c}^5, \text{a}^5 \text{b}^{12} \text{c}^{13}, \text{a}^8 \text{b}^{15} \text{c}^{17}, \dots \} \end{aligned}$$

- 1.9 **EXAMPLE** The empty set is a language  $\mathcal{L} = \{ \}$  over any alphabet. So is the set whose single element is the empty string  $\hat{\mathcal{L}} = \{ \epsilon \}$ . These two languages are different, because the first has no members.

We can think that a natural language such as English consists of sentences, which are strings of words from a dictionary. Here  $\Sigma$  is the set of dictionary words and  $\sigma$  is a sentence. This explains the definition of “language” as a set of strings. Of course, our definition allows a language to be any set of strings at all, while in English you can’t form a sentence by just taking any crazy sequence of words; a sentence must be constructed according to rules. We will study sets of rules, grammars, later in this chapter.

- 1.10 **DEFINITION** A collection of languages is a **class**.

- 1.11 **EXAMPLE** Fix an alphabet  $\Sigma$ . The collection of all finite languages over that alphabet is a class.

- 1.12 **EXAMPLE** Let  $\mathcal{P}_e$  be a Turing machines, using the input alphabet  $\Sigma = \{ \text{B}, \text{1} \}$ . The set of strings  $\mathcal{L}_e = \{ \sigma \in \Sigma^* \mid \mathcal{P}_e \text{ halts on input } \sigma \}$  is a language. The collection of all such languages, of the  $\mathcal{L}_e$  for all  $e \in \mathbb{N}$ , is the class of computably enumerable languages over  $\Sigma$ .

We next consider operations on languages. They are sets so the operations of union, intersection, etc., apply. However, for instance the union of a language over  $\{ \text{a} \}^*$  with a language over  $\{ \text{b} \}^*$  is an awkward marriage, a combination of strings of a’s with strings of b’s. That is, the union of a language over  $\Sigma_0$  with a language over  $\Sigma_1$  is a language over  $\Sigma_0 \cup \Sigma_1$ . The same thing happens for intersection.

- 1.13 **DEFINITION (OPERATIONS ON LANGUAGES)** The **concatenation of languages**,  $\mathcal{L}_0 \mathcal{L}_1$  or  $\mathcal{L}_0 \mathcal{L}_1$ , is the language of concatenations,  $\{ \sigma_0 \sigma_1 \mid \sigma_0 \in \mathcal{L}_0 \text{ and } \sigma_1 \in \mathcal{L}_1 \}$ .

For any language, the **power**,  $\mathcal{L}^k$ , is the language consisting of the concatenation of  $k$ -many members,  $\mathcal{L}^k = \{\sigma_0 \hat{\cdot} \cdots \hat{\cdot} \sigma_{k-1} \mid \sigma_i \in \mathcal{L}\}$  when  $k > 0$ . We take  $\mathcal{L}^1 = \mathcal{L}$  and  $\mathcal{L}^0 = \{\varepsilon\}$ .<sup>†</sup> The **Kleene star of a language**,  $\mathcal{L}^*$ , is the language consisting of the concatenation of any number of strings.

$$\mathcal{L}^* = \{\sigma_0 \hat{\cdot} \cdots \hat{\cdot} \sigma_{k-1} \mid k \in \mathbb{N} \text{ and } \sigma_0, \dots, \sigma_{k-1} \in \mathcal{L}\}$$

This includes the concatenation of 0-many strings, so that  $\varepsilon \in \mathcal{L}^*$  if  $\mathcal{L} \neq \emptyset$ .

The **reversal**,  $\mathcal{L}^R$ , of a language  $\mathcal{L}$  is the language of reversals,  $\mathcal{L}^R = \{\sigma^R \mid \sigma \in \mathcal{L}\}$ .

- 1.14 EXAMPLE Where the language is the set of bitstrings  $\mathcal{L} = \{1000001, 1100001\}$  then the reversal is  $\mathcal{L}^R = \{1000001, 10000011\}$ .
- 1.15 EXAMPLE If the language  $\mathcal{L}$  consists of two strings  $\{a, bc\}$  then the second power of that language is  $\mathcal{L}^2 = \{aa, abc, bca, bcba\}$ . Its Kleene star is this.

$$\mathcal{L}^* = \{\varepsilon, a, bc, aa, abc, bca, bcba, aaa, \dots\}$$

- 1.16 REMARK Here are two points about Kleene star. Earlier, for an alphabet  $\Sigma$  we defined  $\Sigma^*$  to be the set of strings over that alphabet, of any length. The two definitions agree if we take each character in the alphabet to be a length-one string.

Also, we have two choices to define the operation of repeatedly choosing strings. We could choose a string  $\sigma$  and then replicate, getting the  $\sigma^k$ 's. Or, we could repeat choosing strings, getting  $\sigma_0 \hat{\cdot} \sigma_1 \hat{\cdot} \cdots \hat{\cdot} \sigma_{k-1}$ 's. The second is more useful and that's the definition of  $\mathcal{L}^*$ .

We close with a comment that bears on how we will use languages in later chapters. We have defined that a machine ‘decides’ a language if it computes whether or not its input is a language member. However, we have seen the distinction between computable and computably enumerable, that for some sets, for all inputs there is a machine that determines in a finite time whether the input is a member of that set, but the machine cannot determine in a finite time that the input is not in the set. We will say that a machine **recognizes** (or **accepts**, or **semidecides**) a language when, given an input, if the input is in the language then that machine computes that it is, while if the input is not in the language then the machine will never incorrectly report that it is. (It may explicitly determine that it is not, or simply fail to report a conclusion, for example by failing to halt.) In short, deciding means that on any input the machine correctly computes all ‘yes’ and all ‘no’ answers, while recognizing requires only that it correctly computes all ‘yes’ answers and never incorrectly compute a ‘no’.

### III.1 Exercises

- 1.17 List five of the shortest strings in each language, if there are five.

---

<sup>†</sup>For technical convenience we take  $\mathcal{L}^0 = \{\varepsilon\}$  even when  $\mathcal{L} = \emptyset$ ; see Exercise 1.36.

- (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  $\beta$  is a string then  $\beta^\sim \beta^R$  is a palindrome. Do all palindromes have that form?
- ✓ 1.21 Let  $\mathcal{L}_0 = \{\epsilon, a, aa, aaa\}$  and  $\mathcal{L}_1 = \{\epsilon, b, bb, bbb\}$ . (A) List all the members of  $\mathcal{L}_0^\sim \mathcal{L}_1$ . (B) List all the members of  $\mathcal{L}_1^\sim \mathcal{L}_0$ . (C) List all the members of  $\mathcal{L}_0^2$ . (D) List ten members, if there are ten, of  $\mathcal{L}_0^*$ .
- ✓ 1.22 List five members of each language, if there are five, and if not list them all.  
 (A)  $\{\sigma \in \{a, b\}^* \mid \sigma = a^n b \text{ for } n \in \mathbb{N}\}$  (B)  $\{\sigma \in \{a, b\}^* \mid \sigma = a^n b^n \text{ for } n \in \mathbb{N}\}$   
 (C)  $\{1^n 0^{n+1} \in \mathbb{B}^* \mid n \in \mathbb{N}\}$  (D)  $\{1^n 0^{2n} 1 \in \mathbb{B}^* \mid n \in \mathbb{N}\}$
- ✓ 1.23 Where  $\mathcal{L} = \{a, ab\}$ , list each. (A)  $\mathcal{L}^2$  (B)  $\mathcal{L}^3$  (C)  $\mathcal{L}^1$  (D)  $\mathcal{L}^0$
- 1.24 Where  $\mathcal{L}_0 = \{a, ab\}$  and  $\mathcal{L}_1 = \{b, bb\}$  find each. (A)  $\mathcal{L}_0^\sim \mathcal{L}_1$  (B)  $\mathcal{L}_1^\sim \mathcal{L}_0$   
 (C)  $\mathcal{L}_0^2$  (D)  $\mathcal{L}_1^2$  (E)  $\mathcal{L}_0^2 \cap \mathcal{L}_1^2$
- 1.25 Suppose that the language  $\mathcal{L}_0$  has three elements and  $\mathcal{L}_1$  has two. Knowing only that information, for each of these, what is the least number of elements possible and what is the greatest number possible? (A)  $\mathcal{L}_0 \cup \mathcal{L}_1$  (B)  $\mathcal{L}_0 \cap \mathcal{L}_1$   
 (C)  $\mathcal{L}_0^\sim \mathcal{L}_1$  (D)  $\mathcal{L}_1^2$  (E)  $\mathcal{L}_1^R$  (F)  $\mathcal{L}_0^* \cap \mathcal{L}_1^*$
- 1.26 Let  $\mathcal{L} = \{a, b\}$ . Why is  $\mathcal{L}^0$  defined to be  $\{\epsilon\}$ ? Why not  $\emptyset$ ?
- 1.27 What is the language that is the Kleene star of the empty set,  $\emptyset^*$ ?
- ✓ 1.28 Is the  $k$ -th power of a language the same as the language of  $k$ -th powers?
- 1.29 Does  $\mathcal{L}^*$  differ from  $(\mathcal{L} \cup \{\epsilon\})^*$ ?
- 1.30 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.
- 1.31 Prove that  $\mathcal{L}^* = \mathcal{L}^0 \cup \mathcal{L}^1 \cup \mathcal{L}^2 \cup \dots$ .
- 1.32 Consider the empty language  $\mathcal{L}_0 = \emptyset$ . For any language  $\mathcal{L}_1$ , describe  $\mathcal{L}_1^\sim \mathcal{L}_0$ .
- 1.33 A language  $\mathcal{L}$  over some  $\Sigma$  is **finite** if  $|\mathcal{L}| < \infty$ .  
 (A) If the language is finite must the alphabet be finite?  
 (B) Show that there is some bound  $B \in \mathbb{N}$  where  $|\sigma| \leq B$  for all  $\sigma \in \mathcal{L}$ .  
 (C) Show that the class of finite languages is closed under finite union. That is, show that if  $\mathcal{L}_0, \dots, \mathcal{L}_k$  are finite languages over a shared alphabet for some  $k \in \mathbb{N}$  then their union is also finite.

- (D) Show also that the class of finite languages is closed under finite intersection and finite concatenation.
- (E) Show that the class of finite languages is not closed under complementation or Kleene star.

1.34 What is the difference between the languages  $\mathcal{L} = \{\sigma \in \Sigma^* \mid \sigma = \sigma^R\}$  and  $\hat{\mathcal{L}} = \{\sigma^R \mid \sigma \in \Sigma^*\}$ ?

1.35 For any language  $\mathcal{L} \subseteq \Sigma^*$  we can form the set of prefixes.

$$\text{Pref}(\mathcal{L}) = \{\tau \in \Sigma^* \mid \sigma \in \mathcal{L} \text{ and } \tau \text{ is a prefix of } \sigma\}$$

Where  $\Sigma = \{a, b\}$  and  $\mathcal{L} = \{abaaba, bba\}$ , find  $\text{Pref}(\mathcal{L})$ .

- 1.36 This explains why we define  $\mathcal{L}^0 = \{\epsilon\}$  even when  $\mathcal{L} = \emptyset$ .
- (A) Show that  $\mathcal{L}^m \cap \mathcal{L}^n = \mathcal{L}^{m+n}$  for any  $m, n \in \mathbb{N}^+$ .
  - (B) Show that if  $\mathcal{L}_0 = \emptyset$  then  $\mathcal{L}_0 \cap \mathcal{L}_1 = \mathcal{L}_1 \cap \mathcal{L}_0 = \emptyset$ .
  - (C) Argue that if  $\mathcal{L} \neq \emptyset$  then the only sensible definition for  $\mathcal{L}^0$  is  $\{\epsilon\}$ .
  - (D) Why would  $\mathcal{L} = \emptyset$  throw a monkey wrench if the works unless we define  $\mathcal{L}^0 = \{\epsilon\}$ ?

1.37 Prove these for any alphabet  $\Sigma$ .

- (A) For any natural number  $n$  the language  $\Sigma^n$  is countable.
- (B) The language  $\Sigma^*$  is countable.

1.38 Another way of defining the powers of a language is:  $\mathcal{L}^0 = \{\epsilon\}$ , and  $\mathcal{L}^{k+1} = \mathcal{L}^k \cap \mathcal{L}$ . Show this is equivalent to the one given in Definition 1.13.

1.39 True or false: if  $\mathcal{L} \cap \mathcal{L} = \mathcal{L}$  then either  $\mathcal{L} = \emptyset$  or  $\epsilon \in \mathcal{L}$ ? If it is true then prove it and if it is false give a counterexample.

1.40 Prove that no language contains a representation for each real number.

1.41 The operations of languages form an algebraic system. Assume these languages are over the same alphabet. Show each.

- (A) Language union and intersection are commutative,  $\mathcal{L}_0 \cup \mathcal{L}_1 = \mathcal{L}_1 \cup \mathcal{L}_0$  and  $\mathcal{L}_0 \cap \mathcal{L}_1 = \mathcal{L}_1 \cap \mathcal{L}_0$ .
- (B) The language consisting of the empty string is the identity element with respect to language concatenation, so  $\mathcal{L} \cap \{\epsilon\} = \mathcal{L}$  and  $\{\epsilon\} \cap \mathcal{L} = \mathcal{L}$ .
- (C) Language concatenation need not be commutative; there are languages such that  $\mathcal{L}_0 \cap \mathcal{L}_1 \neq \mathcal{L}_1 \cap \mathcal{L}_0$ .
- (D) Language concatenation is associative,  $(\mathcal{L}_0 \cap \mathcal{L}_1) \cap \mathcal{L}_2 = \mathcal{L}_0 \cap (\mathcal{L}_1 \cap \mathcal{L}_2)$ .
- (E)  $(\mathcal{L}_0 \cap \mathcal{L}_1)^R = \mathcal{L}_1^R \cap \mathcal{L}_0^R$ .
- (F) Concatenation is left distributive over union,  $(\mathcal{L}_0 \cup \mathcal{L}_1) \cap \mathcal{L}_2 = (\mathcal{L}_0 \cap \mathcal{L}_2) \cup (\mathcal{L}_1 \cap \mathcal{L}_2)$ , and also right distributive.
- (G) The empty language is an annihilator for concatenation,  $\emptyset \cap \mathcal{L} = \mathcal{L} \cap \emptyset = \emptyset$ .
- (H) The Kleene star operation is idempotent,  $(\mathcal{L}^*)^* = \mathcal{L}^*$ .

## SECTION

### III.2 Grammars

We have defined that a language is a set of strings. But this allows for any willy-nilly set. In practice a language is usually given by rules.

Here is an example. Native English speakers will say that the noun phrase “the big red barn” sounds fine but that “the red big barn” sounds wrong. That is, sentences in natural languages are constructed in patterns and the second of those does not follow the English pattern. Artificial languages such as programming languages also have syntax rules, usually very strict rules.

A grammar a set of rules for the formation of strings in a language, that is, it is an analysis of the structure of 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 This is a subset of the rules for English: (1) a sentence can be made from a noun phrase followed by a verb phrase, (2) a noun phrase can be made from an article followed by a noun, (3) a noun phrase can also be made from an article then an adjective then a noun, (4) a verb phrase can be made with a verb followed by a noun phrase, (5) one article is ‘the’, (6) one adjective is ‘young’, (7) one verb is ‘caught’, (8) two nouns are ‘man’ and ‘ball’.

This is a convenient notation for the rules just listed.

```

⟨sentence⟩ → ⟨noun phrase⟩ ⟨verb phrase⟩
⟨noun phrase⟩ → ⟨article⟩ ⟨noun⟩
⟨noun phrase⟩ → ⟨article⟩ ⟨adjective⟩ ⟨noun⟩
⟨verb phrase⟩ → ⟨verb⟩ ⟨noun phrase⟩
⟨article⟩ → the
⟨adjective⟩ → young
⟨verb⟩ → caught
⟨noun⟩ → man | ball
  
```

Each line is a **production** or **rewrite rule**. Each has one arrow,  $\rightarrow$ .<sup>†</sup> To the left of each arrow is a **head** and to the right is a **body** or **expansion**. Sometimes two rules have the same head, as with  $\langle \text{noun phrase} \rangle$ . There are also two rules for  $\langle \text{noun} \rangle$  but we have abbreviated by combining the bodies using the ‘|’ pipe symbol.<sup>‡</sup>

The rules use two different components. The ones written in typewriter type, such as *young*, are from the alphabet  $\Sigma$  of the language. These are **terminals**. The ones written with angle brackets and in italics, such as  $\langle \text{article} \rangle$ , are **nonterminals**. These are like variables and are used for intermediate steps.

---

<sup>†</sup>Read the arrow aloud as “may produce,” or “may expand to,” or “may be constructed as.” <sup>‡</sup> Read aloud as “or.”

The two symbols ‘ $\rightarrow$ ’ and ‘|’ are neither terminals nor nonterminals. They are **metacharacters**, part of the syntax of the rules themselves.

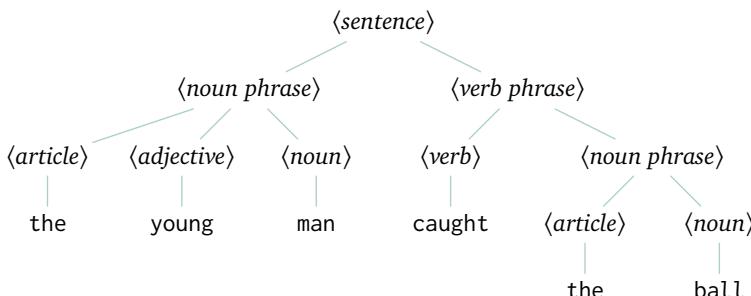
The rewrite rules govern the **derivation** of strings in the language. Under the grammar for English every derivation starts with  $\langle \text{sentence} \rangle$ . During a derivation, intermediate strings contain a mix of nonterminals and terminals. In our grammars every rule has a head with a single nonterminal. So to get the next string pick a nonterminal in the present string and substitute a matching body.

$$\begin{aligned}
 \langle \text{sentence} \rangle &\Rightarrow \langle \text{noun phrase} \rangle \langle \text{verb phrase} \rangle \\
 &\Rightarrow \langle \text{article} \rangle \langle \text{adjective} \rangle \langle \text{noun} \rangle \langle \text{verb phrase} \rangle \\
 &\Rightarrow \text{the } \langle \text{adjective} \rangle \langle \text{noun} \rangle \langle \text{verb phrase} \rangle \\
 &\Rightarrow \text{the young } \langle \text{noun} \rangle \langle \text{verb phrase} \rangle \\
 &\Rightarrow \text{the young man } \langle \text{verb phrase} \rangle \\
 &\Rightarrow \text{the young man } \langle \text{verb} \rangle \langle \text{noun phrase} \rangle \\
 &\Rightarrow \text{the young man caught } \langle \text{noun phrase} \rangle \\
 &\Rightarrow \text{the young man caught } \langle \text{article} \rangle \langle \text{noun} \rangle \\
 &\Rightarrow \text{the young man caught the } \langle \text{noun} \rangle \\
 &\Rightarrow \text{the young man caught the ball}
 \end{aligned}$$

Note that the single line arrow  $\rightarrow$  is for rules while the double line arrow  $\Rightarrow$  is for derivations.<sup>†</sup>

The derivation above always substitutes for the leftmost nonterminal, so it is a **leftmost derivation**. However, in general we could substitute for any nonterminal.

The **derivation tree** or **parse tree** is an alternative representation.<sup>‡</sup>



- 2.2 **DEFINITION** A **context-free grammar** is a four-tuple  $\mathcal{G} = \langle \Sigma, N, S, P \rangle$ . First,  $\Sigma$  is an alphabet, whose elements are the **terminal symbols**. Second,  $N$  is a set of **nonterminals** or **syntactic categories**. (We assume that  $\Sigma$  and  $N$  are disjoint and that neither contains metacharacters.) Third,  $S \in N$  is the **start symbol**. Fourth,  $P$  is a set of **productions** or **rewrite rules**.

We will take the start symbol to be the head of the first rule.

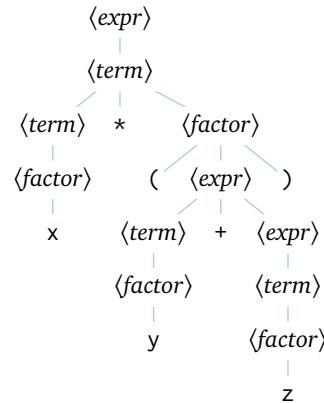
<sup>†</sup> Read ‘ $\Rightarrow$ ’ aloud as “derives” or “expands to.” <sup>‡</sup> The terms ‘terminal’ and ‘nonterminal’ come from where the components lie in this tree.

- 2.3 EXAMPLE This context free grammar describes algebraic expressions that involve only addition, multiplication, and parentheses.

$$\begin{aligned}\langle \text{expr} \rangle &\rightarrow \langle \text{term} \rangle + \langle \text{expr} \rangle \mid \langle \text{term} \rangle \\ \langle \text{term} \rangle &\rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{factor} \rangle \\ \langle \text{factor} \rangle &\rightarrow ( \langle \text{expr} \rangle ) \mid a \mid b \mid \dots \mid z\end{aligned}$$

Here is a derivation of the string  $x*(y+z)$ .

$$\begin{aligned}\langle \text{expr} \rangle &\Rightarrow \langle \text{term} \rangle \\ &\Rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \\ &\Rightarrow \langle \text{factor} \rangle * \langle \text{factor} \rangle \\ &\Rightarrow x * \langle \text{factor} \rangle \\ &\Rightarrow x * ( \langle \text{expr} \rangle ) \\ &\Rightarrow x * ( \langle \text{term} \rangle + \langle \text{expr} \rangle ) \\ &\Rightarrow x * ( \langle \text{term} \rangle + \langle \text{term} \rangle ) \\ &\Rightarrow x * ( \langle \text{factor} \rangle + \langle \text{term} \rangle ) \\ &\Rightarrow x * ( \langle \text{factor} \rangle + \langle \text{factor} \rangle ) \\ &\Rightarrow x * ( y + \langle \text{factor} \rangle ) \\ &\Rightarrow x * ( y + z )\end{aligned}$$



In that example the rules for  $\langle \text{expr} \rangle$  and  $\langle \text{term} \rangle$  are recursive. But we don't get stuck in an infinite regress because the question is not whether you could perversely keep expanding  $\langle \text{expr} \rangle$  forever; the question is whether, given a string such as  $x*(y+z)$ , you can find a terminating derivation.

In the prior example the nonterminals such as  $\langle \text{expr} \rangle$  or  $\langle \text{term} \rangle$  describe the role of those components in the language, as did the English grammar fragment's  $\langle \text{noun phrase} \rangle$  and  $\langle \text{article} \rangle$ . But in the examples and exercises below we often use small grammars whose terminals and nonterminals do not have any particular meaning. For these cases, we often move from the verbose notation like ' $\langle \text{sentence} \rangle \rightarrow \langle \text{noun phrase} \rangle \langle \text{verb phrase} \rangle$ ' to writing single letters, with nonterminals in upper case and terminals in lower case.

- 2.4 EXAMPLE This two-rule grammar has one nonterminal, S.

$$S \rightarrow aSb \mid \epsilon$$

Here is a derivation of the string  $a^2b^2$ .

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\epsilon bb \Rightarrow aabb$$

Similarly,  $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaa\epsilon bbb \Rightarrow aaabb$  is a derivation of  $a^3b^3$ . For this grammar, derivable strings have the form  $a^n b^n$  for  $n \in \mathbb{N}$ .

We next give a complete description of how the production rules govern the derivations. Each rule in a context free grammar has the form 'head  $\rightarrow$  body'

where the head consists of a single nonterminal. The body is a sequence of terminals and nonterminals. Each step of a derivation has the form below, where  $\tau_0$  and  $\tau_1$  are sequences of terminals and non-terminals.

$$\tau_0 \text{ } \widehat{\text{head}} \text{ } \tau_1 \quad \Rightarrow \quad \tau_0 \text{ } \widehat{\text{body}} \text{ } \tau_1$$

That is, if there is a match for the rule's head then we can replace it with the body.

Where  $\sigma_0, \sigma_1$  are sequences of terminals and nonterminals, if they are related by a sequence of derivation steps then we may write  $\sigma_0 \Rightarrow^* \sigma_1$ . Where  $\sigma_0 = S$  is the start symbol, if there is a derivation  $\sigma_0 \Rightarrow^* \sigma_1$  that finishes with a string of terminals  $\sigma_1 \in \Sigma^*$  then we say that  $\sigma_1$  **has a derivation** from the grammar.<sup>†</sup>

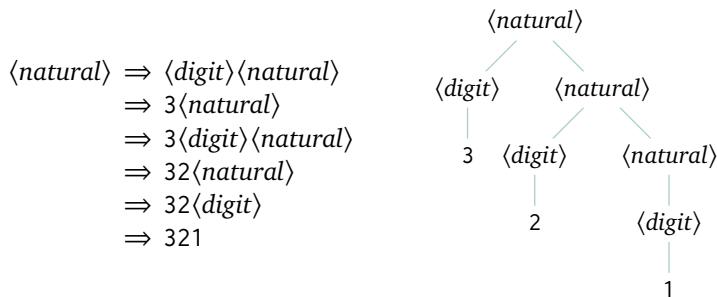
This description is like the one on page 8 detailing how a Turing machine's instructions determine the evolution of the sequence of configurations that is a computation. That is, production rules are like a program, directing a derivation. However, one difference from that page's description is that there Turing machines are deterministic, so that from a given input string there is a determined sequence of configurations. Here, from a given start symbol a derivation can branch out to go to many different ending strings.

**2.5 DEFINITION** The **language derived from a grammar** is the set of strings of terminals having derivations that begin with the start symbol.

**2.6 EXAMPLE** This grammar's language is the set of representations of natural numbers.

$$\begin{aligned} \langle \text{natural} \rangle &\rightarrow \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{natural} \rangle \\ \langle \text{digit} \rangle &\rightarrow 0 \mid \dots \mid 9 \end{aligned}$$

This is a derivation for the string 321, along with its parse tree.



<sup>†</sup>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  $\sigma_0$  and  $\sigma_1$  are strings of terminals. (The  $\sigma_i$ 's can be empty.) For example, consider this grammar: (i)  $S \rightarrow aBSc \mid abc$ , (ii)  $Ba \rightarrow aB$ , (iii)  $Bb \rightarrow bb$ . Rule (ii) says that if you see a string followed by a then you can replace that string with a followed by that thing. Grammars with heads of the form  $\sigma_0 X \sigma_1$  are **context sensitive** because we can only substitute for X in the context of  $\sigma_0$  and  $\sigma_1$ . These grammars describe more languages than the context free ones that we are using. But our definition satisfies our needs and is the class of grammars that you will see in practice.

2.7 EXAMPLE The language of this grammar

$$\langle \text{natural} \rangle \rightarrow \varepsilon \mid 1 \langle \text{natural} \rangle$$

is the set of strings representing natural numbers in unary.

2.8 EXAMPLE Any finite language is derived from a grammar. This one gives the language of all length 2 bitstrings, using the brute force approach of just listing all the member strings.

$$S \rightarrow 00 \mid 01 \mid 10 \mid 11$$

This gives the length 3 bitstrings by using the nonterminals to keep count.

$$A \rightarrow 0B \mid 1B$$

$$B \rightarrow 0C \mid 1C$$

$$C \rightarrow 0 \mid 1$$

2.9 EXAMPLE For this grammar

$$S \rightarrow aSb \mid T \mid U$$

$$T \rightarrow aS \mid a$$

$$U \rightarrow Sb \mid b$$

an alternative is to replace T and U by their expansions to get this.

$$S \rightarrow aSb \mid aS \mid a \mid Sb \mid b$$

It generates the language  $\mathcal{L} = \{a^i b^j \in \{a, b\}^* \mid i \neq 0 \text{ or } j \neq 0\}$ .

This prior example is the first one that we have seen where the generated language is not clear, so we will do a formal verification. We will show mutual containment, that the generated language is a subset of  $\mathcal{L}$  and that it is also a superset.

The rule that eliminates T and U shows that any derivation step  $\tau_0 \xrightarrow{\cdot} \text{head} \xrightarrow{\cdot} \tau_1 \Rightarrow \tau_0 \xrightarrow{\cdot} \text{body} \xrightarrow{\cdot} \tau_1$  only adds a's on the left and b's on the right, so every string in the language has the form  $a^i b^j$ . That same rule shows that in any terminating derivation S must eventually be replaced by either a or b. Together these two give that the generated language is a subset of  $\mathcal{L}$ .

For containment the other way, we will prove that every  $\sigma \in \mathcal{L}$  has a derivation. We will use induction on the length  $|\sigma|$ . By the definition of  $\mathcal{L}$  the base case is  $|\sigma| = 1$ . In this case either  $\sigma = a$  or  $\sigma = b$ , each of which obviously has a derivation.

For the inductive step, suppose that every string from  $\mathcal{L}$  of length  $k = 1, \dots, k = n$  has a derivation for  $n \geq 1$  and let  $\sigma$  have length  $n + 1$ . Write  $\sigma = a^i b^j$ . There are three cases: either  $i > 1$ , or  $j > 1$ , or  $i = j = 1$ . If  $i > 1$  then  $\hat{\sigma} = a^{i-1} b^j$  is a string of length  $n$ , so by the inductive hypothesis it has a derivation  $S \Rightarrow \dots \Rightarrow \hat{\sigma}$ . Prefixing that derivation with a  $S \Rightarrow aS$  step will put an additional a on the left. The  $j > 1$  case works the same way, and  $\sigma = a^1 b^1$  is easy.

- 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

$$\begin{aligned}\langle \text{stmt} \rangle &\rightarrow \text{if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \\ \langle \text{stmt} \rangle &\rightarrow \text{if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle\end{aligned}$$

and this code string.

```
if enrolled(s) if studied(s) grade='P' else grade='F'
```

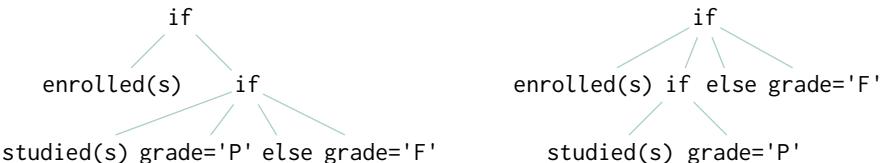
Here are the first two lines of one derivation

$$\begin{aligned}\langle \text{stmt} \rangle &\Rightarrow \text{if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \\ &\Rightarrow \text{if } \langle \text{bool} \rangle \text{ if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle\end{aligned}$$

and here are the first two of another.

$$\begin{aligned}\langle \text{stmt} \rangle &\Rightarrow \text{if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle \\ &\Rightarrow \text{if } \langle \text{bool} \rangle \text{ if } \langle \text{bool} \rangle \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle\end{aligned}$$

That is, we cannot tell whether the else in the code line is associated with the first if or the second. The resulting parse trees for the full code line dramatize the difference



as do these copies of the code string indented to show 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**.

- 2.11 EXAMPLE This grammar for elementary algebra expressions

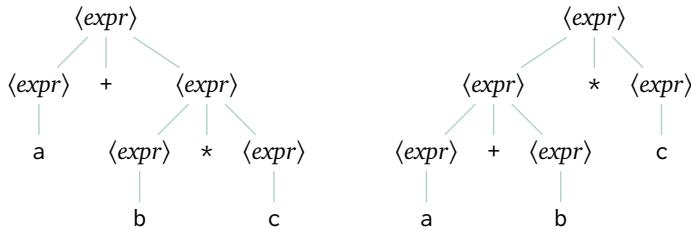
$$\begin{aligned}\langle \text{expr} \rangle &\rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle \\ &\quad | \langle \text{expr} \rangle * \langle \text{expr} \rangle \\ &\quad | ( \langle \text{expr} \rangle ) \quad | \text{a} \mid \text{b} \mid \dots \text{z}\end{aligned}$$

is ambiguous because  $a+b*c$  has two leftmost derivations.

$$\begin{aligned}\langle \text{expr} \rangle &\Rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle \Rightarrow a + \langle \text{expr} \rangle \\ &\Rightarrow a + \langle \text{expr} \rangle * \langle \text{expr} \rangle \Rightarrow a + b * \langle \text{expr} \rangle \Rightarrow a + b * c\end{aligned}$$

$$\begin{aligned}\langle \text{expr} \rangle &\Rightarrow \langle \text{expr} \rangle * \langle \text{expr} \rangle \Rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle * \langle \text{expr} \rangle \\ &\Rightarrow a + \langle \text{expr} \rangle * \langle \text{expr} \rangle \Rightarrow a + b * \langle \text{expr} \rangle \Rightarrow a + b * c\end{aligned}$$

The two give different parse trees.



Again, the issue is that we get two different behaviors. For instance, substitute 1 for a, and 2 for b, and 3 for c. The left tree gives  $1 + (2 \cdot 3) = 7$  while the right tree gives  $(1 + 2) \cdot 3 = 9$ .

In contrast, this grammar for elementary algebra expressions is unambiguous.

$$\begin{aligned}\langle \text{expr} \rangle &\rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \\ &\quad | \langle \text{term} \rangle \\ \langle \text{term} \rangle &\rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \\ &\quad | \langle \text{factor} \rangle \\ \langle \text{factor} \rangle &\rightarrow (\langle \text{expr} \rangle) \\ &\quad | a \quad | b \quad | \dots \quad | z\end{aligned}$$

Choosing grammars that are not ambiguous is important in practice.

### III.2 Exercises

- ✓ 2.12 Use the grammar of Example 2.3. (A) What is the start symbol? (B) What are the terminals? (C) What are the nonterminals? (D) How many rewrite rules does it have? (E) Give three strings derived from the grammar, besides the string in the example. (F) Give three strings in the language  $\{+, *\}, (\, , a \dots, z\}^*$  that cannot be derived.
- 2.13 Use the grammar of Exercise 2.15. (A) What is the start symbol? (B) What are the terminals? (C) What are the nonterminals? (D) How many rewrite rules does it have? (E) Give three strings derived from the grammar besides the ones in the exercise, or show that there are not three such strings. (F) Give three strings in the language  $\mathcal{L} = \{\sigma \in \Sigma \cup \{\text{space}\}^* \mid \Sigma \text{ is the set of terminals}\}$  that cannot be derived from this grammar, or show there are not three such strings.
- 2.14 Use this grammar.

$$\begin{aligned}\langle \text{natural} \rangle &\rightarrow \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{natural} \rangle \\ \langle \text{digit} \rangle &\rightarrow 0 \mid 1 \mid \dots \mid 9\end{aligned}$$

- (A) What is the alphabet? What are the terminals? The nonterminals? What

is the start symbol? (B) For each production, name the head and the body.  
 (C) Which are the metacharacters that are used? (D) Derive 42. Also give its parse tree.  
 (E) Derive 993 and give the associated parse tree. (F) How can  $\langle \text{natural} \rangle$  be defined in terms of  $\langle \text{natural} \rangle$ ? Doesn't that lead to infinite regress?  
 (G) Extend this grammar to cover the integers. (H) With this grammar, can you derive +0? -0?

- ✓ 2.15 From this grammar

$$\begin{aligned}\langle \text{sentence} \rangle &\rightarrow \langle \text{subject} \rangle \langle \text{predicate} \rangle \\ \langle \text{subject} \rangle &\rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle \\ \langle \text{predicate} \rangle &\rightarrow \langle \text{verb} \rangle \langle \text{direct object} \rangle \\ \langle \text{direct object} \rangle &\rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle \\ \langle \text{article} \rangle &\rightarrow \text{the} \mid \text{a} \\ \langle \text{noun} \rangle &\rightarrow \text{car} \mid \text{wall} \\ \langle \text{verb} \rangle &\rightarrow \text{hit}\end{aligned}$$

derive each of these: (A) the car hit a wall (B) the car hit the wall  
 (C) the wall hit a car.

- 2.16 In the language generated by this grammar.

$$\begin{aligned}\langle \text{sentence} \rangle &\rightarrow \langle \text{subject} \rangle \langle \text{predicate} \rangle \\ \langle \text{subject} \rangle &\rightarrow \langle \text{article} \rangle \langle \text{noun}_1 \rangle \\ \langle \text{predicate} \rangle &\rightarrow \langle \text{verb} \rangle \langle \text{direct-object} \rangle \\ \langle \text{direct-object} \rangle &\rightarrow \langle \text{article} \rangle \langle \text{noun}_2 \rangle \\ \langle \text{article} \rangle &\rightarrow \text{the} \mid \text{a} \mid \epsilon \\ \langle \text{noun}_1 \rangle &\rightarrow \text{dog} \mid \text{flea} \\ \langle \text{noun}_2 \rangle &\rightarrow \text{man} \mid \text{dog} \\ \langle \text{verb} \rangle &\rightarrow \text{bites} \mid \text{licks}\end{aligned}$$

- (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.

$$\begin{aligned}\langle \text{sentence} \rangle &\Rightarrow \langle \text{subject} \rangle \langle \text{predicate} \rangle \\ &\Rightarrow \langle \text{article} \rangle \langle \text{noun}_1 \rangle \langle \text{predicate} \rangle \\ &\Rightarrow \langle \text{article} \rangle \langle \text{noun}_1 \rangle \langle \text{verb} \rangle \langle \text{direct object} \rangle \\ &\Rightarrow \langle \text{article} \rangle \langle \text{dog} \mid \text{flea} \rangle \langle \text{verb} \rangle \langle \text{article} \rangle \langle \text{noun}_2 \rangle \\ &\Rightarrow \langle \text{article} \rangle \langle \text{dog} \mid \text{flea} \rangle \langle \text{verb} \rangle \langle \text{article} \rangle \langle \text{man} \mid \text{dog} \rangle\end{aligned}$$

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

$$\begin{aligned} S &\rightarrow TbU \\ T &\rightarrow aT \mid \varepsilon \\ U &\rightarrow aU \mid bU \mid \varepsilon \end{aligned}$$

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.

$$\begin{aligned} S &\rightarrow aABb \\ A &\rightarrow aA \mid a \\ B &\rightarrow Bb \mid b \end{aligned}$$

(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?

$$\begin{aligned} A &\rightarrow aA \mid B \\ B &\rightarrow bB \mid cA \end{aligned}$$

✓ 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 Here is a grammar for propositional logic expressions in Conjunctive Normal form.

$$\begin{aligned} \langle CNF \rangle &\rightarrow (\langle Disjunction \rangle) \wedge \langle DNF \rangle \mid (\langle Disjunction \rangle) \\ \langle Disjunction \rangle &\rightarrow \langle Literal \rangle \vee \langle Disjunction \rangle \mid \langle Literal \rangle \\ \langle Literal \rangle &\rightarrow \neg \langle Variable \rangle \mid \langle Variable \rangle \\ \langle Variable \rangle &\rightarrow x_0 \mid x_1 \mid \dots \end{aligned}$$

For more, see Appendix C.

(A) Derive  $(x_0 \vee \neg x_1) \wedge (x_1 \vee x_2)$ .

(B) Show that you cannot derive  $(\neg x_0 \wedge \neg x_1) \vee (\neg x_0 \wedge \neg x_2)$ .

2.27 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.

2.28 What is the language generated by a grammar with a set of production rules that is empty?

- 2.29 Create a grammar for each of these languages.
- the language of all character strings  $\mathcal{L} = \{a, \dots, z\}^*$
  - the language of strings of at least one digit  $\{\sigma \in \{0, \dots, 9\}^* \mid |\sigma| \geq 1\}$
- ✓ 2.30 This is a grammar for postal addresses. Note the use of the empty string  $\epsilon$  to make some components optional, such as  $\langle \text{opt suffix} \rangle$  and  $\langle \text{apt num} \rangle$ .

```

⟨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⟩
⟨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  $\langle EOL \rangle$  expands to an end of line such as ASCII 10, while (space) signifies a whitespace character such as ASCII 0 or ASCII 32, or even more exotic characters such as en-space or em-space.

- (A) Give a derivation for this address.

President  
1600 Pennsylvania Avenue  
Washington, DC

- (B) Why is there no derivation for this address?

Sherlock Holmes  
221B Baker Street  
London, UK

Suggest a modification of the grammar so that this address is in the language.

- (C) Give three reasons why this grammar is inadequate. (Perhaps no grammar would suffice other than just listing every address in the world.)

2.31 Recall Turing's prototype computer, a clerk doing the symbolic manipulations to multiply two large numbers. Deriving a string from a grammar has a similar feel and we can write grammars to do computations. Fix the alphabet  $\Sigma = \{1\}$ , so that we can interpret derived strings as numbers represented in unary.

(A) Produce a grammar whose language is the even numbers,  $\{1^{2n} \mid n \in \mathbb{N}\}$ .

(B) Do the same for the multiples of three,  $\{1^{3n} \mid n \in \mathbb{N}\}$ .

- ✓ 2.32 Here is a grammar notable for being small.

$$\langle \text{sentence} \rangle \rightarrow \text{buffalo } \langle \text{sentence} \rangle \mid \epsilon$$

(A) Derive a sentence of length one, one of length two, and one of length three.

(B) Give those sentences semantics, that is, make sense of them as English sentences.

2.33 Here is a grammar for LISP.

$$\begin{aligned} \langle s \text{ expression} \rangle &\rightarrow \langle \text{atomic symbol} \rangle \\ &\mid (\langle s \text{ expression} \rangle . \langle s \text{ expression} \rangle) \\ &\mid \langle \text{list} \rangle \\ \langle \text{list} \rangle &\rightarrow ( \langle \text{list-entries} \rangle ) \\ \langle \text{list-entries} \rangle &\rightarrow \langle s \text{ expression} \rangle \\ &\mid \langle s \text{ expression} \rangle \langle \text{list-entries} \rangle \\ \langle \text{atomic symbol} \rangle &\rightarrow \langle \text{letter} \rangle \langle \text{atom part} \rangle \\ \langle \text{atom part} \rangle &\rightarrow \epsilon \\ &\mid \langle \text{letter} \rangle \langle \text{atom part} \rangle \\ &\mid \langle \text{number} \rangle \langle \text{atom part} \rangle \\ \langle \text{letter} \rangle &\rightarrow a \mid \dots z \\ \langle \text{number} \rangle &\rightarrow 0 \mid \dots 9 \end{aligned}$$

Give a derivation for each string. (A) (a . b) (B) (a . (b . c))  
(C) ((a . b) . c)

2.34 Using the Example 2.11's unambiguous grammar, produce a derivation for  $a + (b * c)$ .

2.35 The simplest example of an ambiguous grammar is

$$S \rightarrow S \mid \epsilon$$

(A) What is the language generated by this grammar?

(B) Produce two different derivations of the empty string.

2.36 This is a grammar for the language of bitstrings  $\mathcal{L} = \mathbb{B}^*$ .

$$\langle \text{bit-string} \rangle \rightarrow 0 \mid 1 \mid \langle \text{bit-string} \rangle \langle \text{bit-string} \rangle$$

Show that it is ambiguous.

2.37 (A) Show that this grammar is ambiguous by producing two different leftmost derivations for a-b-a.

$$E \rightarrow E - E \mid a \mid b$$

(b) Derive a-b-a from this grammar, which is unambiguous.

$$E \rightarrow E - T \mid T$$

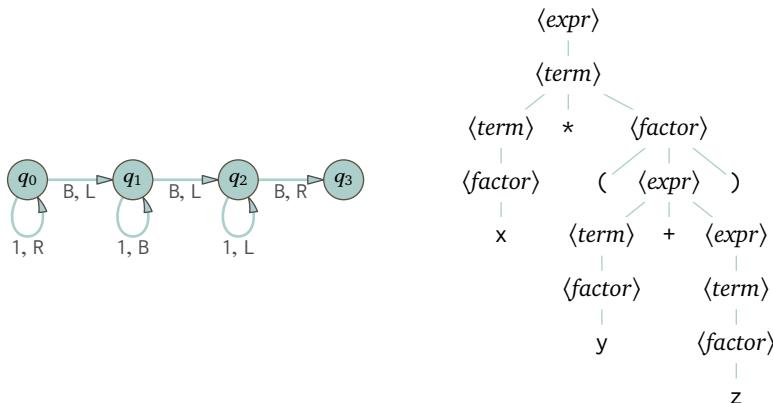
$$T \rightarrow a \mid b$$

2.38 Use the grammar from the footnote on 155 to derive aaabbbcc.

## SECTION

### III.3 Graphs

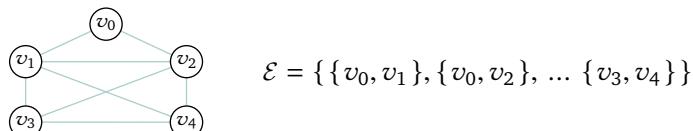
Researchers in the Theory of Computation often state their problems, and the solution of those problems, in the language of Graph Theory. Here are two examples we have already seen. Both have vertices connected by edges that represent a relationship between the vertices.



**Definition** We start with the basics.

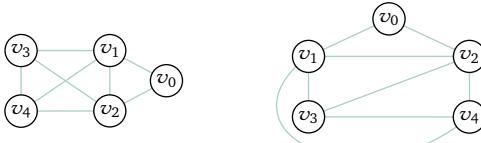
3.1 **DEFINITION** A **simple graph** is an ordered pair  $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$  where  $\mathcal{N}$  is a finite set of **vertices**<sup>†</sup> or **nodes** and  $\mathcal{E}$  is a set of **edges**. Each edge is a set of two distinct vertices.

3.2 **EXAMPLE** This simple graph  $\mathcal{G}$  has five vertices  $\mathcal{N} = \{v_0, \dots, v_4\}$  and eight edges.



Important: a graph is not its picture. Both of these pictures show the same graph as above because they show the same vertices and the same connections.

<sup>†</sup>Graphs can have infinitely many vertices but we won't ever need that. For convenience we will stick to finite ones.



Instead of writing  $e = \{v, \hat{v}\}$  we often write  $e = v\hat{v}$ . Since sets are unordered we could write the same edge as  $e = \hat{v}v$ .

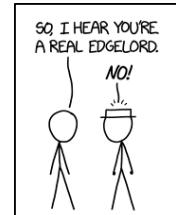
There are many variations of that definition, used for modeling various circumstances. One variant of a simple graph allows that some vertices connect to themselves, forming a **loop**. Another is a **multigraph**, which allows two vertices to have more than one edge between them.

Still another is a **weighted graph**, which gives each edge a real number **weight**, perhaps signifying the distance or the cost in money or in time to traverse that edge.

A very common variation is a **directed graph** or **digraph**, where edges have a direction, as in a road map that includes one-way streets. In a digraph, 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 at the start of this section is a digraph and also has loops.

Some important graph variations involve the nature of the connections. A **tree** is an undirected connected graph with no cycles (the definition of cycle is just below but it is what it sounds like it is, a closed path). A **directed acyclic graph** or **DAG** is a directed graph with no directed cycles.

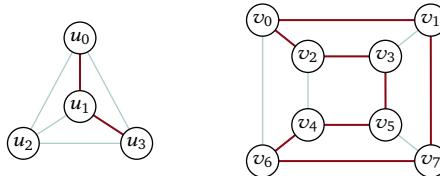
**Traversal** Many problems involve moving through a graph.



Courtesy xkcd.com

3.3 **DEFINITION** Two graph edges are **adjacent** if they share a vertex, so that they have the form  $e_0 = uv$  and  $e_1 = vw$ . A **walk** is a sequence of adjacent edges  $\langle v_0v_1, v_1v_2, \dots, v_{n-1}v_n \rangle$ . Its **length** is the number of edges,  $n$ . If the initial vertex  $v_0$  equals the final vertex  $v_n$  then is **closed** walk, otherwise it is **open**. If no edge occurs twice then it is a **trail**. If a trail's vertices are distinct, except possibly that the initial vertex equals the final vertex, then it is a **path**. A closed path with at least one edge is a **cycle**.

3.4 **EXAMPLE** On the left is a path from  $u_0$  to  $u_3$ ; it is also a trail and a walk. On the right is a cycle.



- 3.5 **DEFINITION** The vertex  $v_1$  is **reachable** from the vertex  $v_0$  if there is a path from  $v_0$  to  $v_1$ . A graph is **connected** if between any two vertices there is a path.
- 3.6 **DEFINITION** In a graph, a **circuit** is a closed walk that either contains all of the edges, making it an **Euler circuit**, or all of the vertices, making it a **Hamiltonian circuit**.
- 3.7 **EXAMPLE** The graph on the right of Example 3.4 is a Hamiltonian circuit but not an Euler circuit.
- 3.8 **DEFINITION** Where  $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$  is a graph, a **subgraph**  $\hat{\mathcal{G}} = \langle \hat{\mathcal{N}}, \hat{\mathcal{E}} \rangle$  satisfies  $\hat{\mathcal{N}} \subseteq \mathcal{N}$  and  $\hat{\mathcal{E}} \subseteq \mathcal{E}$ . A subgraph with every possible edge, a  $\hat{\mathcal{N}}$  such that if  $e = v_i v_j \in \mathcal{E}$  and  $v_i, v_j \in \hat{\mathcal{N}}$  then  $e \in \hat{\mathcal{E}}$  also, is an **induced subgraph**.
- 3.9 **EXAMPLE** In the graph  $\mathcal{G}$  on the left of Example 3.4, consider the highlighted path  $\mathcal{E} = \{u_0 u_1, u_1 u_3\}$ . Taking those edges along with the vertices that they contain,  $\hat{\mathcal{N}} = \{u_0, u_1, u_3\}$ , gives a subgraph  $\hat{\mathcal{G}}$ .  
Also in  $\mathcal{G}$ , the induced subgraph involving the set of vertices  $\{u_0, u_2, u_3\}$  is the outer triangle.

We will need this result in Chapter Five.

- 3.10 **LEMMA (KÖNIG'S LEMMA)** Suppose that in a connected tree each vertex has finite degree, that is, it is adjacent to only finitely many other vertices. If the tree has infinitely many vertices then it has an infinite path.

*Proof* Let the tree have infinitely many vertices and fix one,  $v_0$ . The tree is connected so starting at  $v_0$  we can reach every other vertex via some path. For each of  $v_0$ 's neighbors  $v$ , let the set of vertices that can be reached by a path from  $v$  that does not go through  $v_0$  be  $R_v$ . At least one of these  $R_v$  must contain infinitely many such vertices or else the graph would be a union of the finitely many finite sets  $R_v$  (along with  $\{v_0\}$ ). Pick one such neighbor and call it  $v_1$ . Iterating gives an infinite path  $\langle v_0, v_1, \dots \rangle$ .  $\square$

**Graph representation** A common way to represent a graph in a computer 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 otherwise.

$$\mathcal{M}(\mathcal{G}) = \begin{pmatrix} & v_0 & v_1 & v_2 & v_3 & v_4 \\ v_0 & 0 & 1 & 1 & 0 & 0 \\ v_1 & 1 & 0 & 1 & 1 & 1 \\ v_2 & 1 & 1 & 0 & 1 & 1 \\ v_3 & 0 & 1 & 1 & 0 & 1 \\ v_4 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \quad (*)$$

- 3.11 **DEFINITION** For a graph  $\mathcal{G}$ , the **adjacency matrix**  $\mathcal{M}(\mathcal{G})$  representing the graph has  $i, j$  entries equal to the number of edges from  $v_i$  to  $v_j$ .

This definition covers 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 a graph has a loop then the matrix has a diagonal entry that is a positive integer. If the graph is directed and has a one-way edge from  $v_i$  to  $v_j$  then the  $i, j$  entry records that edge but the  $j, i$  entry does not. And for a multigraph, where there are multiple edges from one vertex to another, the associated entry will be larger than 1.

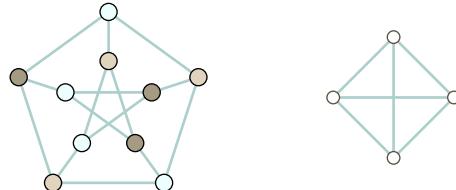
- 3.12 **LEMMA** Let the matrix  $\mathcal{M}(\mathcal{G})$  represent the graph  $\mathcal{G}$ . Then in its matrix multiplicative  $n$ -th power the  $i, j$  entry is the number of paths of length  $n$  from vertex  $v_i$  to vertex  $v_j$ .

*Proof* Exercise 3.38. □

**Colors** We sometimes partition a graph's vertices.

- 3.13 **DEFINITION** A  **$k$ -coloring** of a graph, for  $k \in \mathbb{N}$ , is a partition of vertices into  $k$ -many classes such that no two adjacent vertices are in the same class.

On the left is a graph that is 3-colored.

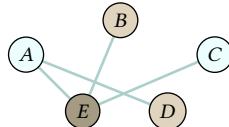


On the right the graph has no 3-coloring. The argument goes: the four vertices are completely connected to each other. If two get the same color then they will be adjacent same-colored vertices. So a coloring requires four colors.

- 3.14 **EXAMPLE** This shows five committees, where some committees share some members. How many time slots do we need in order to schedule all committees so that no member has to be in two meetings at once?

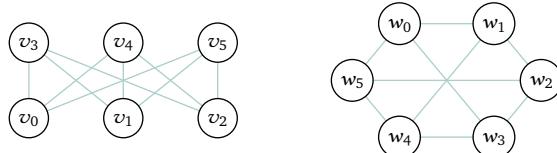
<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>
Armis	Crump	Burke	India	Burke
Jones	Edwards	Frank	Harris	Jones
Smith	Robinson	Ke	Smith	Robinson

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.

**Graph isomorphism** We sometimes want to know when two graphs are essentially identical. Consider these two.



They have the same number of vertices and the same number of edges. Further, on the right as well as on the left there are two classes of vertices where all the vertices in the first class connect to all the vertices in the second class (on the left the two classes are the top and bottom rows while on the right they are  $\{w_0, w_2, w_4\}$  and  $\{w_1, w_3, w_5\}$ ). A person may suspect that as in Example 3.2 these are two ways to draw the same graph, with the vertex names changed for further obfuscation.

That's true: if we make a correspondence between the vertices in this way

Vertex on left	$v_0$	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$
Vertex on right	$w_0$	$w_2$	$w_4$	$w_1$	$w_3$	$w_5$

then as a consequence the edges also correspond.

Edge on left	$\{v_0, v_3\}$	$\{v_0, v_4\}$	$\{v_0, v_5\}$	$\{v_1, v_3\}$	$\{v_1, v_4\}$	$\{v_1, v_5\}$
Edge on right	$\{w_0, w_1\}$	$\{w_0, w_3\}$	$\{w_0, w_5\}$	$\{w_2, w_1\}$	$\{w_2, w_3\}$	$\{w_2, w_5\}$
Edge on left (cont)	$\{v_2, v_3\}$	$\{v_2, v_4\}$	$\{v_2, v_5\}$			
Edge on right	$\{w_2, w_1\}$	$\{w_2, w_3\}$	$\{w_2, w_5\}$			

**3.15 DEFINITION** Two graphs  $\mathcal{G}$  and  $\hat{\mathcal{G}}$  are **isomorphic** if there is a one-to-one and onto map  $f: \mathcal{N} \rightarrow \hat{\mathcal{N}}$  such that  $\mathcal{G}$  has an edge  $\{v_i, v_j\} \in \mathcal{E}$  if and only if  $\hat{\mathcal{G}}$  has the associated edge  $\{f(v_i), f(v_j)\} \in \hat{\mathcal{E}}$ .

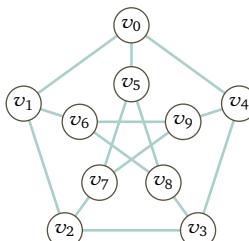
To verify that two graphs are isomorphic the most natural thing is to produce the map  $f$  and then verify that in consequence the edges also correspond. The exercises have examples.

Showing that graphs are not isomorphic usually entails finding some graph-theoretic way in which they differ. A common and useful such property is to consider the **degree of a vertex**, the total number of edges touching that vertex with the proviso that a loop from the vertex to itself counts as two. The **degree sequence** of a graph is the non-increasing sequence of its vertex degrees. Thus, the graph in

Example 3.14 has degree sequence  $\langle 3, 2, 1, 1, 1 \rangle$ . Exercise 3.37 shows that if graphs are isomorphic then associated vertices have the same degree and thus graphs with different degree sequences are not isomorphic. Also, if the degree sequences are equal then they help us construct an isomorphism, if there is one; examples of this are in the exercises. (Note, though, that there are graphs with the same degree sequence that are not isomorphic.)

### 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 to Massachusetts and New Hampshire. Massachusetts is adjacent to every other state. New Hampshire is adjacent to Maine, Massachusetts, and Vermont. Rhode Island is adjacent to Connecticut and Massachusetts. Vermont is adjacent to Massachusetts and New Hampshire. Give the graph describing the adjacency relation.
  - (B) In the game of Rock-Paper-Scissors, Rock beats Scissors, Paper beats Rock, and Scissors beats Paper. Give the graph of the ‘beats’ relation; note that this is a directed relation.
  - (C) The number  $m \in \mathbb{N}$  is related to the number  $n \in \mathbb{N}$  by being its divisor if there is a  $k \in \mathbb{N}$  with  $m \cdot k = n$ . Give the divisor relation graph among positive natural numbers less than or equal to 12.
  - (D) The river Pregel cut the town of Königsberg into four land masses. There were two bridges from mass 0 to mass 1 and one bridge from mass 0 to mass 2. There was one bridge from mass 1 to mass 2, and two bridges from mass 1 to mass 3. Finally, there was one bridge from mass 2 to 3. Consider masses related by bridges. Give the graph (it is a multigraph).
  - (E) In our Mathematics program you must take Calculus II before you take Calculus III, and you must take Calculus I before II. You must take Calculus II before Linear Algebra, and to take Real Analysis you must have both Linear Algebra and Calculus III.
- 3.17 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?
- 3.18 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?

3.19 Let a graph  $\mathcal{G}$  have vertices  $\{v_0, \dots, v_5\}$  and the edges  $v_0v_1, v_0v_3, v_0v_5, v_1v_4, v_3v_4$ , and  $v_4v_5$ .

(A) Draw  $\mathcal{G}$ .

(B) Give its adjacency matrix.

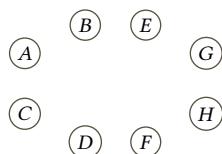
(C) Find all subgraphs with four nodes and four edges.

(D) Find all induced subgraphs with four nodes and four edges.

3.20 A graph is not a drawing, it is a set of vertices and edges. So a single graph may have quite different pictures. Consider a graph  $\mathcal{G}$  with the vertices  $\mathcal{N} = \{A, \dots, H\}$  and these edges.

$$\mathcal{E} = \{AB, AC, AG, AH, BC, BD, BF, CD, CE, DE, DF, EF, EG, FH, GH\}$$

(A) Connect the dots below to get one drawing.



(B) A **planar graph** is one that can be drawn in the plane so that its edges do not cross. Show that  $\mathcal{G}$  is planar.

3.21 Fill in the table's blanks.

	Closed or open?	Vertices can repeat?	Edges can repeat?
Walk	—	—	—
Trail	—	—	—
Circuit	—	—	—
Path	—	—	—
Cycle	—	—	—

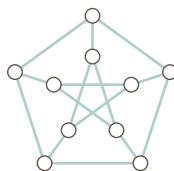
- ✓ 3.22 Morse code represents text with a combination of a short sound, written ‘·’ and pronounced “dit,” and a long sound, written ‘–’ and pronounced “dah.” Here are the representations of the twenty six English letters.

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.

3.23 Show that every tree has a 2-coloring.

3.24 This is the Petersen graph.



(A) Show that it has no 2-coloring. (B) Give a 3-coloring.

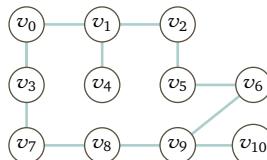
3.25 A person keeps six species of fish as pets. Species *A* cannot be in a tank with species *B* or *C*. Species *B* cannot be with *A*, *C*, or *E*. Species *C* cannot be with *A*, *B*, *D*, or *E*. Species *D* cannot be with *C* or *F*. Species *E* cannot be together with *B*, *C*, or *F*. Finally, species *F* cannot be in with *D* or *E*.

(A) Draw the graph where the nodes are species and the edges represent the relation ‘cannot be together’.

(B) Find the chromatic number.

(C) Interpret it.

- ✓ 3.26 If two cell towers are within line of sight of each other then they must get different frequencies. Here 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.

3.27 For the graph in the prior exercise, give the degree sequence.

- ✓ 3.28 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.

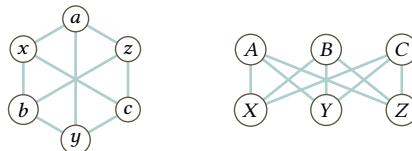
3.29 Find the degree sequence of the graph in Example 3.2 and of the two graphs of Example 3.4.

3.30 Give the array representation, like that in equation (\*), for the graphs of Example 3.4.

3.31 Draw a graph for this adjacency matrix.

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

✓ 3.32 These two graphs are isomorphic.

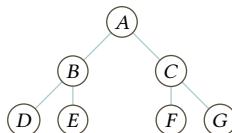


(A) Define the function giving the correspondence.

(B) Verify that under that function the edges then also correspond.

3.33 Give the degree sequences of the two graphs in the prior exercise.

✓ 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 Consider building a simple graph by starting with  $n$  vertices.

(A) How many potential edges are there?

(B) How many such graphs are there?

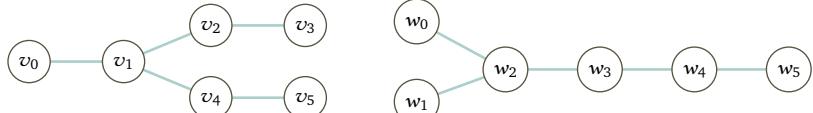
3.37 We can use degrees and degree sequences to help find isomorphisms, or to show that graphs are not isomorphic. (Here we allow graphs to have loops and to have multiple edges between vertices, but we do not make the extension to directed edges or edges with weights.)

(A) Show that if two graphs are isomorphic then they have the same number of vertices.

(B) Show that if two graphs are isomorphic then they have the same number of edges.

(C) Show that if two graphs are isomorphic and one has a vertex of degree  $k$  then so does the other.

- (D) Show that if two graphs are isomorphic then for each degree  $k$ , the number of vertices of the first graph having that degree equals the number of vertices of the second graph having that degree. Thus, isomorphic graphs have degree sequences that are equal.
- (E) Verify that while these two graphs have the same degree sequence, they are not isomorphic. Hint: consider the paths starting at the degree 3 vertex.



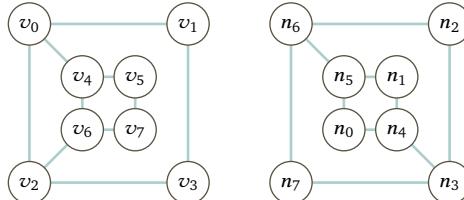
- (F) Use the prior result to show that the two graphs of Example 3.4 are not isomorphic.

As in the final item, in arguments we often use the contrapositive of these statements. For instance, the first item implies that if they do not have the same number of vertices then they are not isomorphic.

### 3.38 Prove Lemma 3.12.

- (A) An edge as a length-1 walk. Show that in the product of the matrix with itself  $(\mathcal{M}(\mathcal{G}))^2$  the entry  $i, j$  is the number of length-two walks.
- (B) Show that for  $n > 2$ , the  $i, j$  entry of the power  $(\mathcal{M}(\mathcal{G}))^n$  equals the number of length  $n$  walks from  $v_i$  to  $v_j$ .

### 3.39 Consider these two graphs, $\mathcal{G}_0$ and $\mathcal{G}_1$ .



- (A) List the vertices and edges of  $\mathcal{G}_0$ .
- (B) Do the same for  $\mathcal{G}_1$ .
- (C) Give the degree sequences of  $\mathcal{G}_0$  and  $\mathcal{G}_1$ .
- (D) 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 $\mathcal{G}_1$	$n_6$	$n_2$	$n_7$	$n_3$	$n_5$	$n_1$	$n_0$	$n_4$

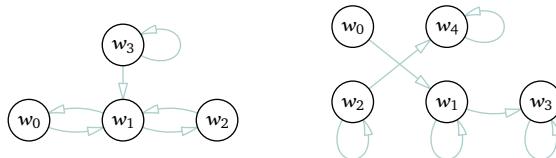
Find the image, under the correspondence, of the edges of  $\mathcal{G}_0$ . Do they match the edges of  $\mathcal{G}_1$ ?

- (E) Of course, failure of any one proposed map does not imply that the two cannot be isomorphic. Nonetheless, argue that they are not isomorphic.

### 3.40 In a graph, for a node $q_0$ there may be some nodes $q_i$ that are unreachable, so there is no path from $q_0$ to $q_i$ .

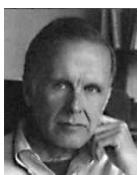
- (A) Devise an algorithm that inputs a directed graph and a start node  $q_0$ , and finds the set of nodes that are unreachable from  $q_0$ .

(b) Apply your algorithm to these two starting with  $w_0$ .



## EXTRA

### III.A BNF



John Backus  
1924–2007

We shall introduce some grammar notation conveniences that are widely used. Together they are called **Backus-Naur form, BNF**.

The study of grammar, the rules for phrase structure and forming sentences, has a long history, dating back as early as the fifth century BC. Mathematicians, including A Thue and E Post, began systematizing it as rewriting rules in the early 1900's. The BNF variant was produced by J Backus in the late 1950's as part of the design of the early computer language ALGOL60. Since then these rules have become a standard way to express grammars.

One difference from the prior subsection is a minor typographical change. Originally the metacharacters were not typeable with a standard keyboard. The advantage of having metacharacters not on a keyboard is that most likely all of the language characters are typeable. So there is no need to distinguish, say, between the pipe character | when used as a part of a language and when used as a metacharacter. But the disadvantage lies in having to type the untypeable. In the end the convenience of having typeable characters won over the technical gain of having to typographically distinguish metacharacters. For instance, for a long time there were not arrows on a standard keyboard so in place of the arrow symbol ' $\rightarrow$ ', BNF uses ' $::=$ '. (These adjustments were made by P Naur, as editor of the ALGOL60 report.)<sup>†</sup>



Peter Naur  
1928–2016

BNF is both clear and concise, it can express the range of languages that we ordinarily want to express, and it smoothly translates to a parser.<sup>‡</sup> That is, BNF is an impedance match—it fits with what we typically want to do. Here we will incorporate some extensions for grouping and replication that are like what you will see in the wild.

A.1 **EXAMPLE** This is a BNF grammar for real numbers with a finite decimal part. Take the rules for  $\langle \text{natural} \rangle$  from Example 2.6.

$\langle \text{start} \rangle ::= -\langle \text{fraction} \rangle \mid +\langle \text{fraction} \rangle \mid \langle \text{fraction} \rangle$

$\langle \text{fraction} \rangle ::= \langle \text{natural} \rangle \mid \langle \text{natural} \rangle . \langle \text{natural} \rangle$

<sup>†</sup>There are other typographical issues that arise with grammars. While many authors write nonterminals with diamond brackets, as we do, others use other conventions such as a separate type style or color.

<sup>‡</sup>BNF is only loosely defined. Several variants do have standards but what you see often does not conform to any published standard.

This derivation for 2.718 is rightmost.

```

⟨start⟩ ⇒ ⟨fraction⟩ ⇒ ⟨natural⟩ . ⟨natural⟩
⇒ ⟨natural⟩ . ⟨digit⟩⟨natural⟩ ⇒ ⟨natural⟩ . ⟨digit⟩⟨digit⟩⟨natural⟩
⇒ ⟨natural⟩ . ⟨digit⟩⟨digit⟩⟨digit⟩ ⇒ ⟨natural⟩ . ⟨digit⟩⟨digit⟩8
⇒ ⟨natural⟩ . ⟨digit⟩18 ⇒ ⟨natural⟩ . 718 ⇒ 2.718

```

Here is a derivation for 0.577 that is neither leftmost nor rightmost.

```

⟨start⟩ ⇒ ⟨fraction⟩ ⇒ ⟨natural⟩ . ⟨natural⟩
⇒ ⟨natural⟩ . ⟨digit⟩⟨natural⟩ ⇒ ⟨natural⟩ . 5⟨natural⟩
⇒ ⟨natural⟩ . 5⟨digit⟩⟨natural⟩ ⇒ ⟨digit⟩ . 5⟨digit⟩⟨natural⟩
⇒ ⟨digit⟩ . 5⟨digit⟩⟨digit⟩ ⇒ ⟨digit⟩ . 5⟨digit⟩7 ⇒ 0.5⟨digit⟩7
⇒ 0.577

```

- A.2 EXAMPLE Time is a difficult engineering problem. One issue is representing times and one solution 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 a BNF grammar. (See Exercise 2.29 for some nonterminals.) This grammar includes some metacharacter extensions discussed below.

```

⟨date-fullyear⟩ ::= <4-digits>
⟨date-month⟩ ::= <2-digits>
⟨date-mday⟩ ::= <2-digits>
⟨time-hour⟩ ::= <2-digits>
⟨time-minute⟩ ::= <2-digits>
⟨time-second⟩ ::= <2-digits>
⟨time-secfrac⟩ ::= . <1-or-more-digits>
⟨time-numoffset⟩ ::= (+ | -) <time-hour⟩ : <time-minute⟩
⟨time-offset⟩ ::= Z | <time-numoffset⟩
⟨partial-time⟩ ::= <time-hour⟩ : <time-minute⟩ : <time-second⟩
    [<time-secfrac⟩]
⟨full-date⟩ ::= <date-fullyear⟩ - <date-month⟩ - <date-mday⟩
⟨full-time⟩ ::= <partial-time⟩ <time-offset⟩
⟨date-time⟩ ::= <full-date⟩ T <full-time⟩

```

That example shows a BNF notation in the ⟨time-numoffset⟩ rule, where the parentheses are used as metacharacters to group a choice between the terminals + and -. It shows another extension in the ⟨partial-time⟩ rule, which includes square brackets as metacharacters. These denote that the ⟨time-secfrac⟩ is optional.

The square brackets is a very common construct: another example is this syntax for if ... then ... with an optional else ...

```
<if-stmt> ::= if <boolean-expr> then <stmt-sequence>
    [ else <stmt-sequence> ] <end if> ;
```

To show repetition, BNF may use a superscript Kleene star  $*$  to mean ‘zero or more’ or a  $^+$  to mean ‘one or more’. This shows parentheses and repetition.

```
<identifier> ::= <letter> ( <letter> | <digit> ) $^*$ 
```

Each of these extension constructs is not necessary since we can express them in plain BNF, without the extensions. For instance, we could replace the prior rule with this.

```
<identifier> ::= <letter> | <letter> <atoms>
<atoms> ::= <letter> <atoms> | <digit> <atoms> |  $\epsilon$ 
```

But these constructs come up often enough that adopting an abbreviation is convenient.

A.3 EXAMPLE This grammar for Python floating point numbers shows all three abbreviations.

```
<floatnumber> ::= <pointfloat> | <exponentfloat>
<pointfloat> ::= [<intpart>] <fraction> | <intpart> .
<exponentfloat> ::= (<intpart> | <pointfloat>) <exponent>
<intpart> ::= <digit> $^+$ 
<fraction> ::= . <digit> $^+$ 
<exponent> ::= (e | E) [+ | -] <digit> $^+$ 
```

As part of the *<pointfloat>* rule, the first *<intpart>* is optional. An *<intpart>* consists of one or more digits. And an expansion of *<exponent>* must start with a choice between e or E.

A.4 REMARK Passing from the grammar to a parser for that grammar is mechanical. We can write a program that does it, that takes as input a grammar (for example in BNF) and gives as output the source code of a program that will parse files following that grammar’s format. This is a **parser-generator**, also called a **compiler-compiler** (while that term is zingy, it is also misleading because a parser is only part of a compiler).

### III.A Exercises

- ✓ A.5 US ZIP codes have five digits, and may have a dash and four more digits at the end. Give a BNF grammar.
- A.6 Write a grammar in BNF for the language of palindromes.
- ✓ A.7 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.8 Example A.3 uses some BNF convenience abbreviations.
  - (A) Give a grammar equivalent to  $\langle \text{pointfloat} \rangle$  that doesn't use square brackets.
  - (B) Do the same for the repetition operator in  $\langle \text{intpart} \rangle$ 's rule, and for the grouping in  $\langle \text{exponent} \rangle$ 's rule (you can use  $\langle \text{intpart} \rangle$  here).
- ✓ A.9 In Roman numerals the letters I, V, X, L, C, D, and M stand for the values 1, 5, 10, 50, 100, 500, and 1 000. We write the letters from left to right in descending order of value, so that XVI represents the number that we would ordinarily write as 16, and MDCCCLVIII represents 1958. We always write the shortest possible string, so we do not write IIIII because we can instead write V. However, as we don't have a symbol whose value is larger than 1 000 we must represent large numbers with lots of M's.
  - (A) Give a grammar for the strings that make sense as Roman numerals.
  - (B) Often Roman numerals are written in subtractive notation: for instance, 4 is represented as IV, because four I's are hard to distinguish from three of them in a setting such as a watch face. In this notation 9 is IX, 40 is XL, 90 is XC, 400 is CD, and 900 is CM. Give a grammar for the strings that can appear in this notation.

A.10 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.11 Here is a grammar for LISP.

```

⟨s-expression⟩ ::= ⟨atomic-symbol⟩
| ( ⟨s-expression⟩ . ⟨s-expression⟩ )
| ⟨list⟩

⟨list⟩ ::= ( ⟨s-expression⟩* )

⟨atomic-symbol⟩ ::= ⟨letter⟩ ⟨atom-part⟩

⟨atom-part⟩ ::= ⟨empty⟩
| ⟨letter⟩ ⟨atom-part⟩
| ⟨number⟩ ⟨atom-part⟩

⟨letter⟩ ::= a | b | ... z

⟨number⟩ ::= 1 | 2 | ... 9

```

Derive the s-expression (cons (car x) y).

A.12 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.



# COMPUT ECCLESIASTIQUE



## CHAPTER

# IV Automata

Our touchstone model of mechanical computation is the Turing machine. A Turing machine has two components, a CPU and a tape. We will now take the tape away and study the CPU alone. So we consider what can be done with states alone.

Alternatively stated, a person could object to our characterization of Turing machines as the right model of what can be done by an idealized discrete and deterministic device, on the following grounds: While each Turing Machine has finitely many states, it can write to the tape. So there is a machine that writes a character, moves right, writes a character, moves right again, etc. This machine passes through infinitely many configurations. But no physical device can do that, even an idealized one. In this chapter we study what can be done by a machine with a bounded number of possible configurations.

## SECTION

### IV.1 Finite State machines

We produce a new model of computation, the Finite State machine, by modifying the Turing machine definition. We will strip out the capability to write, changing from read/write to read-only. It will turn out that these machines can do many things, but not as many as Turing machines.

**Definition** We defined Turing machines as sets of instructions. We have occasionally referred to those instructions, and we will again soon, but what matters most about them is that they describe the machine's next-state function,  $\Delta$ . It is this function that governs how the machine transitions from configuration to configuration during a computation.

For the definition of Finite State machines we will just give the transition function. We will use the same type of transition tables and transition graphs as with Turing machines, as the preliminary examples below illustrate.

- 1.1 EXAMPLE A power switch has two states,  $q_{\text{off}}$  and  $q_{\text{on}}$ , and its input alphabet has one symbol, toggle.

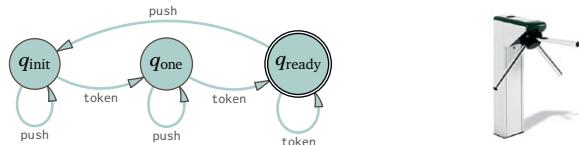


---

IMAGE: The astronomical clock in Notre-Dame-de-Strasbourg Cathedral, for computing the date of Easter. Easter falls on the first Sunday after the full moon on or after the spring equinox. Calculation of this date was a great challenge for mechanisms of that time, 1843.

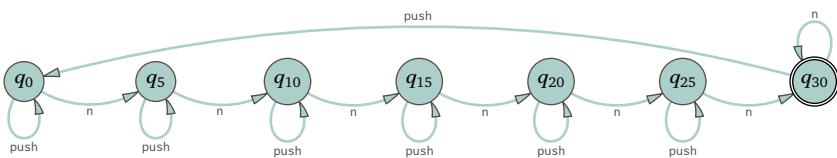
The state  $q_{\text{on}}$  is drawn with a double circle. Without a way to write, Finite State machines need some way to indicate the computational's outcome. We say that this is an **accepting state** or **final state** (in the transition function tables we mark final states with '+').

- 1.2 EXAMPLE Operate this turnstile by putting in two tokens and then pushing through. It has three states and its input alphabet is  $\Sigma = \{\text{token}, \text{push}\}$ .

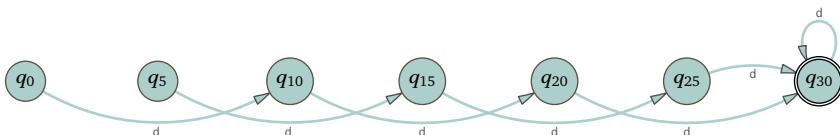


As we saw with Turing machines, the states are a limited form of memory. For instance,  $q_{\text{one}}$  is how the turnstile “remembers” that it has so far received one token.

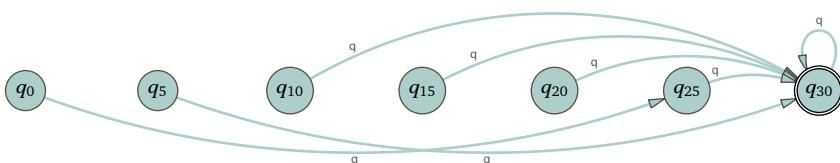
- 1.3 EXAMPLE This vending machine dispenses items that cost 30 cents.<sup>†</sup> The picture is complex so we will show it in three layers. First are the arrows for nickels and pushing the dispense button.



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 we could give change, but here we ignore that. Next comes the arrows for dimes

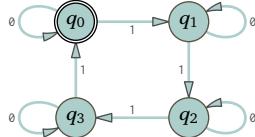


and for quarters.



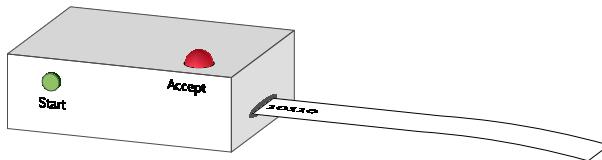
<sup>†</sup> US coins are: 1 cent coins that are not used here, nickles are 5 cents, dimes are 10 cents, and quarters are 25 cents.

- 1.4 EXAMPLE This machine, when started in state  $q_0$  and fed bitstrings, will keep track of the remainder modulo 4 of the number of 1's in the input.



- 1.5 DEFINITION A **Finite State machine** or **Finite State automata**  $\langle Q, q_{\text{start}}, F, \Sigma, \Delta \rangle$  has a finite set of states  $Q$ , one of which is the **start state**  $q_{\text{start}}$ , a subset  $F \subseteq Q$  of **final states** or **accepting states**, a finite **input alphabet** set  $\Sigma$ , and a **next-state function** or **transition function**  $\Delta: Q \times \Sigma \rightarrow Q$ .

A full description of the action of a Finite State machine, like the one that we gave for Turing machines, comes below on page 186, after some more examples. But the basic idea is: to work a Finite State machine, put the finite-length input on the tape and press Start.



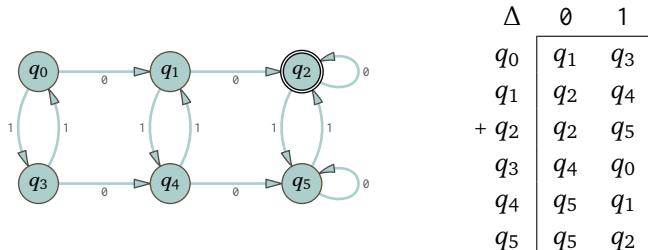
The machine consumes the input, at each step reading, acting on, and then deleting the present tape character. We've pictured that the final states turn on the red light. So once the string is fully consumed, to find the computation's result see whether the light is on. If the light is on then we say that the machine **accepts** the starting input string, otherwise it **rejects** that string.

Here is a trace of the steps when Example 1.4's modulo 4 machine is started on 10110.

<i>Step</i>	<i>Configuration</i>	<i>Step</i>	<i>Configuration</i>
0	1 0 1 1 0 q <sub>0</sub>	3	1 0 q <sub>2</sub>
1	0 1 1 0 q <sub>1</sub>	4	0 q <sub>3</sub>
2	1 1 0 q <sub>1</sub>	5	 q <sub>3</sub>

Because these machines consumes one character per step and stops once they are gone, they are sure to halt—there is no Halting problem for Finite State machines.

- 1.6 EXAMPLE This machine accepts a string if and only if it contains at least two 0's as well as an even number of 1's.

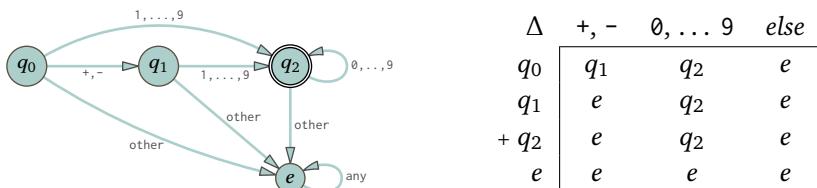


This machine illustrates the key to designing Finite State machines, that each state has an intuitive meaning. The state  $q_4$  means “so far the machine has seen one 0 and an odd number of 1's.” And  $q_5$  means “so far the machine has seen two 0's but an odd number of 1's.”

The drawing brings out this principle. Its first row has states that have so far seen an even number of 1's, while the second row's states have seen an odd number. Its first column holds states have seen no 0's, the second column holds states have seen one, and the third column has states that have seen two 0's.

- 1.7 EXAMPLE This machine accepts strings that are valid as decimal representations of integers. Thus, it accepts ‘21’ and ‘-707’ but does not accept ‘501-’.

Both the transition graph and the table group some inputs together when they result in the same action. For instance, when in state  $q_0$  this machine does the same thing whether the input is + or -, namely it passes into  $q_1$ .

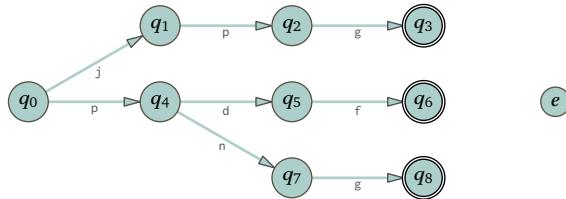


Any bad input character sends the machine to the state  $e$ . Finite State machines in practice often has such an error state, which is a sink state in that once the machine enters that state, it never leaves.

Our Finite State machine descriptions will usually take that the alphabet is clear from the context. For instance, Example 1.6's alphabet is  $\Sigma = \mathbb{B} = \{0, 1\}$ . The prior example just says ‘else’; in machines designed for practice, the alphabet is the set of characters that someone could conceivably enter. Often this includes letters such as a and A or characters such as exclamation point or open parenthesis. Design of a machine up to a modern standard might use all of Unicode. But for the examples and exercises here, we will use small alphabets.<sup>†</sup>

<sup>†</sup>We often use a, b, etc., in the alphabet because saying ‘ $a^2$ ’ is less awkward than ‘ $0^2$ ’ or even ‘two 0's’.

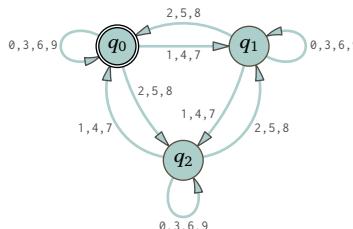
- 1.8 EXAMPLE This machine accepts strings that are members of the set { jpg, pdf, png } of filename extensions. Notice that it has more than one final state.



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. (Putting in all the edges would make a mess. Cases such as this are where the transition table is better than the graph picture. But most of our machines are small so we typically prefer the picture.)

This example illustrates that for any finite language there is a Finite State machine that accepts a string if and only if it is a member of the language. The idea is: for strings that have common prefixes, the machine steps through the shared parts together, as here with pdf and png. Exercise 1.45 asks for a proof.

- 1.9 EXAMPLE Although they have no scratch memory, some Finite State machines accomplish reasonably hard tasks, such as some kinds of arithmetic. This machine accepts strings representing a natural number that is a multiple of three, such as 15 and 8013.



Because  $q_0$  is an accepting state, this machine accepts the empty string. Exercise 1.23 asks for a modification of this machine to accept only non-empty strings.

- 1.10 EXAMPLE Finite State machines are easy to translate to code. Here is a Scheme version of the multiple of three machine.

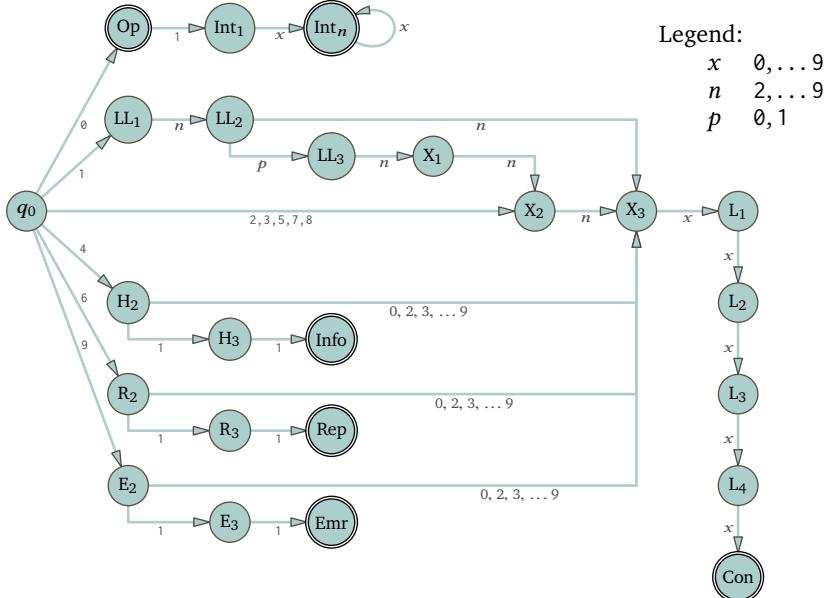
```
;; Decide if the input represents a multiple of three
(define (multiple-of-three-fsm input-string)
  (let ((state 0))
    (if (= 0 (multiple-of-three-fsm-helper state input-string))
        (display "accepted")
        (display "rejected")))
    (display (newline)))))

;; tail-recursive helper fcn
(define (multiple-of-three-fsm-helper state tau)
  (let ((tau-list (string->list tau)))
    (cond
      ((empty? tau-list) state)
      ((= 0 (mod (char->integer (list-ref tau-list 0)) 3)) (multiple-of-three-fsm-helper state (cdr tau-list)))
      (else (multiple-of-three-fsm-helper (+ state 1) (cdr tau-list))))))
```

```
(if (null? tau-list)
    state
    (multiple-of-three-fsm-helper (delta state (car tau-list))
        (list->string (cdr tau-list)))))

(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)))))
```

- 1.11 EXAMPLE This is how phone numbers were originally handled in North America. Consider 1-802-555-0101. The initial 1 signifies that the call should leave the local exchange to go to the long lines. The 802 is an area code; the system can tell this is not a same-area local exchange because its second digit is 0 or 1. Next comes the 555, routing the call to a particular physical local switching office. That office processes the line number of 0101 and makes the connection.



Today the picture is much more complicated. For example, no longer are area

codes required to have a middle digit of 0 or 1. This additional complication is possible because instead of switching with physical devices, today we do it in software.

After the definition of Turing machine we gave a formal description of the action of those machines. We next do the same here.

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  $\tau$  is a (possibly empty) string,  $\tau \in \Sigma^*$ . We start a machine with some **input** string  $\tau_0$  and say that the **initial configuration** is  $\mathcal{C}_0 = \langle q_0, \tau_0 \rangle$ .

A Finite State machine acts by a sequence of **transitions** from one configuration to another. For  $s \in \mathbb{N}^+$ , the machine's configuration after the  $s$ -th transition is its configuration at **step  $s$** ,  $\mathcal{C}_s$ . Here is the rule for making a transition (we sometimes say it is an **allowed** or **legal** transition, for emphasis). Suppose that the machine is in the configuration  $\mathcal{C}_s = \langle q, \tau_s \rangle$ . In the case that  $\tau_s$  is not empty, pop the string's leading symbol  $c$ . That is, where  $c = \tau_s[0]$ , take  $\tau_{s+1} = \langle \tau_s[1], \dots, \tau_s[k] \rangle$  for  $k = |\tau_s| - 1$ . Then the machine's next state is  $\hat{q} = \Delta(q, c)$  and its next configuration is  $\mathcal{C}_{s+1} = \langle \hat{q}, \tau_{s+1} \rangle$ . Denote this before-after relationship between configurations by  $\mathcal{C}_s \vdash \mathcal{C}_{s+1}$ .<sup>†</sup> The other case is that the string  $\tau_s$  is empty. This is the **halting configuration**  $\mathcal{C}_h$ . No transitions follow a halting configuration.

At each transition the length of the tape string drops by one so every computation eventually reaches a halting configuration  $\mathcal{C}_h = \langle q, \varepsilon \rangle$ . A **Finite State machine computation** is a sequence  $\mathcal{C}_0 \vdash \mathcal{C}_1 \vdash \mathcal{C}_2 \vdash \dots \vdash \mathcal{C}_h$ . We can abbreviate such a sequence with  $\vdash^*$ , as in  $\mathcal{C}_0 \vdash^* \mathcal{C}_h$ .<sup>‡</sup>

If the ending state is a final state,  $q \in F$ , then the machine **accepts** the input  $\tau_0$ , otherwise it **rejects**  $\tau_0$ .

Notice that, as with the formalism for Turing machines, the heart of the definitions is the transition function  $\Delta$ . It makes the machine move step-by-step, from configuration to configuration, in response to the input.

- 1.12 **EXAMPLE** The multiple of three machine of the prior example gives the computation.  $\langle q_0, 5013 \rangle \vdash \langle q_2, 013 \rangle \vdash \langle q_2, 13 \rangle \vdash \langle q_0, 3 \rangle \vdash \langle q_0, \varepsilon \rangle$ . Since  $q_0$  is an accepting state, the machine accepts 5013.

- 1.13 **DEFINITION** The set of strings accepted by a Finite State machine  $\mathcal{M}$  is the **language of that machine**,  $\mathcal{L}(\mathcal{M})$ , or the language **recognized** (or **decided**, or **accepted**), by the machine.

(For Finite State machines, deciding a language is equivalent to recognizing it, because the machine must halt. ‘Recognized’ is more the common term.)

- 1.14 **DEFINITION** For any Finite State machine with transition function  $\Delta: Q \times \Sigma \rightarrow Q$ , the **extended transition function**  $\hat{\Delta}: \Sigma^* \rightarrow Q$  gives the state in which the machine ends after starting in the start state and consuming the given string.

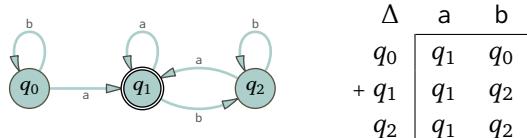
Here is an equivalent constructive definition of  $\hat{\Delta}$ . Fix a Finite State machine  $\mathcal{M}$  with transition function  $\Delta: Q \times \Sigma \rightarrow Q$ . To begin, set  $\hat{\Delta}(\varepsilon) = \{q_0\}$ . Then for

---

<sup>†</sup>As with Turing machines, read the symbol  $\vdash$  aloud as “yields in one step.” <sup>‡</sup>Read the symbol  $\vdash^*$  as “yields eventually” or simply “yields.”

$\tau \in \Sigma^*$ , define  $\hat{\Delta}(\tau \cap t) = \Delta(q_{i_0}, t) \cup \Delta(q_{i_1}, t) \cup \dots \cup \Delta(q_{i_k}, t)$  for any  $t \in \Sigma$ . Finally, observe that a string  $\sigma \in \Sigma^*$  is accepted by the machine if  $\hat{\Delta}(\sigma)$  is a final state.

1.15 EXAMPLE This machine's extended transition function  $\hat{\Delta}$



extends its ordinary transition function  $\Delta$  in that it repeats the first row of  $\Delta$ 's table.

$$\hat{\Delta}(a) = q_1 \quad \hat{\Delta}(b) = q_0$$

(We disregard the difference between  $\Delta$ 's input characters and  $\hat{\Delta}$ 's input length one strings.) Here is  $\hat{\Delta}$ 's effect on the length two strings.

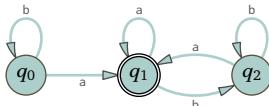
$$\hat{\Delta}(aa) = q_1 \quad \hat{\Delta}(ab) = q_2 \quad \hat{\Delta}(ba) = q_1 \quad \hat{\Delta}(bb) = q_0$$

This brings us back to determinism because  $\hat{\Delta}$  would not be well-defined without it;  $\Delta$  has one next state for all input configurations and so, by induction, for all input strings  $\hat{\Delta}$  has one output ending state.

Finally, note the similarity between  $\hat{\Delta}$  and  $\phi_e$ , the function computed by the Turing machine  $P_e$ . Both take as input the contents of their machine's start tape, and both give as output their machine's result.

#### IV.1 Exercises

- ✓ 1.16 Using this machine, trace through the computation when the input is
  - abba
  - bab
  - bbaabbaa.



1.17 True or false: because a Finite State is finite, its language must be finite.

1.18 Rebut “no Finite State machine can recognize the language  $\{ a^n b \mid n \in \mathbb{N} \}$  because  $n$  is infinite.”

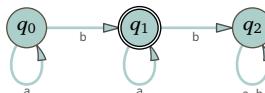
1.19 Your classmate says, “I have a language  $\mathcal{L}$  that recognizes the empty string  $\varepsilon$ .” Explain to them the mistake.

- ✓ 1.20 How many transitions does an input string of length  $n$  cause a Finite State machine to undergo?  $n$ ?  $n + 1$ ?  $n - 1$ ? How many (not necessarily distinct) states will the machine have visited after consuming the string?

- ✓ 1.21 For each of these formal descriptions of a language, give a one or two sentence English-language description. Also list five strings that are elements as well as five that are not, if there are five.
- $\mathcal{L} = \{\alpha \in \{a, b\}^* \mid \alpha = a^nba^n \text{ for } n \in \mathbb{N}\}$
  - $\{\beta \in \{a, b\}^* \mid \beta = a^nba^m \text{ for } n, m \in \mathbb{N}\}$
  - $\{ba^n \in \{a, b\}^* \mid n \in \mathbb{N}\}$
  - $\{a^nba^{n+2} \in \{a, b\}^* \mid n \in \mathbb{N}\}$
  - $\{\gamma \in \{a, b\}^* \mid \gamma \text{ has the form } \gamma = \alpha^\frown \alpha \text{ for } \alpha \in \{a, b\}^*\}$
- ✓ 1.22 For the machines of Example 1.6, Example 1.7, Example 1.8, and Example 1.9, answer these. (A) What are the accepting states? (B) Does it recognize the empty string  $\varepsilon$ ? (C) What is the shortest string that each accepts? (D) Is the language of accepted strings infinite?
- 1.23 Modify the machine of Example 1.9 so that it accepts only non-empty strings.
- ✓ 1.24 Here is a good way to develop Finite State machines. First, for each language, name five strings that are in the language and five that are not (the alphabet is  $\Sigma = \{a, b\}$ ). Second, design the machine that will recognize the language by articulating what each state is doing, what it means. Thus, for each language here, besides the ten strings in the first step, also give a one-sentence English description of each state that you use. Finally, give both a circle diagram and a transition function table.
- $\mathcal{L}_1 = \{\sigma \in \Sigma^* \mid \sigma \text{ has at least one } a \text{ and at least one } b\}$
  - $\mathcal{L}_2 = \{\sigma \in \Sigma^* \mid \sigma \text{ has fewer than three } a's\}$
  - $\mathcal{L}_3 = \{\sigma \in \Sigma^* \mid \sigma \text{ ends in } ab\}$
  - $\mathcal{L}_4 = \{a^n b^m \in \Sigma^* \mid n, m \geq 2\}$
  - $\mathcal{L}_5 = \{a^n b^m a^p \in \Sigma^* \mid m = 2\}$
- 1.25 Produce the transition graph picturing this transition function. What is the machine's language?

$\Delta$	a	b
$q_0$	$q_2$	$q_1$
$+ q_1$	$q_0$	$q_2$
$q_2$	$q_2$	$q_2$

- ✓ 1.26 What language is recognized by this machine?



- ✓ 1.27 Give a Finite State machine over  $\Sigma = \{a, b, c\}$  that accepts any string containing the substring abc. As in Example 1.6, give a brief description of each state's intuitive meaning in the machine.

- 1.28 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. Give a Finite State machine recognizing this language. As in Example 1.6, give a brief description of the intuitive meaning of each state.
- ✓ 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 give a transition graph and table for a Finite State machine recognizing the language. Use  $\Sigma = \{a, b\}$ .
- (A)  $\{\sigma \in \Sigma^* \mid \sigma \text{ has at least two a's}\}$
  - (B)  $\{\sigma \in \Sigma^* \mid \sigma \text{ has exactly two a's}\}$
  - (C)  $\{\sigma \in \Sigma^* \mid \sigma \text{ has less than three a's}\}$
  - (D)  $\{\sigma \in \Sigma^* \mid \sigma \text{ has at least one a followed by at least one b}\}$
- 1.30 Produce a Finite State machine over the alphabet  $\Sigma = \{A, \dots, Z, 0, \dots, 9\}$  that accepts only the string 911, and a machine that accepts any string but that one.
- 1.31 Using Example 1.15, apply the extended transition function to all of the length three and length four string inputs.
- ✓ 1.32 Consider a language of comments, which begin with the two-character string /#, end with #/, and have no #/ substrings in the middle. Give a Finite State machine to recognize that language. (Just producing the transition graph is enough.)
- 1.33 For each language, give five strings from that language and five that are not (if there are not that many then list all of the strings that are possible). Also give a Finite State machine that recognizes the language. Use  $\Sigma = \{a, b\}$ .
- (A)  $\mathcal{L} = \{\sigma \in \{a, b\}^* \mid \sigma \text{ ends in aa}\}$
  - (B)  $\{\sigma \in \{a, b\}^* \mid \sigma = \varepsilon\}$
  - (C)  $\{\sigma \in \{a, b\}^* \mid \sigma = a^3b \text{ or } \sigma = ba^3\}$
  - (D)  $\{\sigma \in \{a, b\}^* \mid \sigma = a^n \text{ or } \sigma = b^n \text{ for } n \in \mathbb{N}\}$
- 1.34 What happens when the input to an extended transition function is the empty string?
- ✓ 1.35 Produce a Finite State machine that recognizes each.
- (A)  $\{\sigma \in \{0, \dots, 9\}^* \mid \sigma \text{ has either no 0's or no 2's}\}$
  - (B)  $\{\sigma \in \{0, \dots, 9\}^* \mid \sigma \text{ is the decimal representation of a multiple of 5}\}$
- ✓ 1.36 Give a Finite State machine over the alphabet  $\Sigma = \{A, \dots, Z\}$  that accepts only strings in which the vowels occur in ascending order. (The traditional vowels, in ascending order, are A, E, I, O, and U.)
- ✓ 1.37 Consider this grammar.
- $$\begin{aligned} \langle \text{real} \rangle &\rightarrow \langle \text{posreal} \rangle \mid + \langle \text{posreal} \rangle \mid - \langle \text{posreal} \rangle \\ \langle \text{posreal} \rangle &\rightarrow \langle \text{natural} \rangle \mid \langle \text{natural} \rangle . \mid \langle \text{natural} \rangle . \langle \text{natural} \rangle \\ \langle \text{natural} \rangle &\rightarrow \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{natural} \rangle \\ \langle \text{digit} \rangle &\rightarrow 0 \mid \dots \mid 9 \end{aligned}$$

(A) Give five strings that are in its language and five that are not. (B) Is the string .12 in the language? (c) Briefly describe the language. (d) Give a Finite State machine that recognizes the language.

1.38 Produce five strings in each language and five that are not. Also produce a Finite State machine to recognize it.

- (A)  $\{\sigma \in \mathbb{B}^* \mid \text{every 1 in } \sigma \text{ has a 0 just before it and just after}\}$
- (B)  $\{\sigma \in \mathbb{B}^* \mid \sigma \text{ represents a number divisible by 4 in binary}\}$
- (C)  $\{\sigma \in \{0, \dots, 9\}^* \mid \sigma \text{ represents an even number in decimal}\}$
- (D)  $\{\sigma \in \{0, \dots, 9\}^* \mid \sigma \text{ represents a multiple of 100 in decimal}\}$

1.39 Consider  $\{\sigma \in \{0, \dots, 9\}^* \mid \sigma \text{ represents a multiple of 4 in base 10}\}$ . Briefly describe a Finite State machine. You need not give the full graph or table.

1.40 As in Example 1.12, find the computation for the multiple of three machine with the initial string 2332.

1.41 We will through the formal definition of the extended transition function that follows Definition 1.14 by applying it to the machine in Example 1.6. (A) Use the definition to find  $\hat{\Delta}(0)$  and  $\hat{\Delta}(1)$ . (B) Use the definition to find  $\hat{\Delta}$ 's output on inputs 00, 01, 10, and 11. (c) Find its action on all length three strings.

✓ 1.42 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.

1.43 Let  $\Sigma = \mathbb{B}$ . (A) List all of the different Finite State machines over  $\Sigma$  with a single state,  $Q = \{q_0\}$ . (Ignore whether a state is final or not; we will do that below.) (B) List all the ones with two states,  $Q = \{q_0, q_1\}$ . (C) How many machines are there with  $n$  states? (D) What if we distinguish between machines with different sets of final states?

✓ 1.44 *Propositiones ad acuendos iuvenes* (problems to sharpen the young) is the oldest collection of mathematical problems in Latin. It is by Alcuin of York (735–804), royal advisor to Charlemagne and head of the Frankish court school. One problem, *Propositio de lupo et capra et fasciculo cauli*, is particularly famous: *A man had to transport to the far side of a river a wolf, a goat, and a bundle of cabbages. The only boat he could find was one that could carry only two of them. For that reason, he sought a plan which would enable them all to get to the far side unhurt. Let him, who is able, say how it could be possible to transport them safely.* A wolf cannot be left alone with a goat nor can a goat be left alone with cabbages. Construct the relevant Finite State machine and use it to solve the problem.

1.45 Show, as suggested by Example 1.8, that for any finite language there is a Finite State machine recognizing that language.

1.46 There are languages not recognized by any Finite State machine. Fix an alphabet  $\Sigma$  with at least two members. (A) Show that the number of Finite

State machines with that alphabet is infinite. (b) Show that it is countable. (c) Show that the number of languages over that alphabet is uncountable.

## SECTION

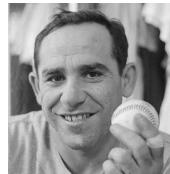
### IV.2 Nondeterminism

Turing machines and Finite State machines both have the property that, given the current state and current character, the next state is completely determined. Once you lay out an initial tape and push Start then the machine just walks through the determined steps. We now consider machines that are nondeterministic, which may have configurations where there is more than one next state that it could move to, or perhaps just one, or perhaps even no state at all.

**Motivation** Imagine a grammar with some rules and a start symbol. You are given a string and asked if has a derivation. The challenge to these problems is that you sometimes have to guess which path the derivation should take. For instance, if you have  $S \rightarrow aS \mid bA$  then from  $S$  you can do two different things; which one will work?

In the grammar section's derivation exercises, we expect that an intelligent person will have the insight to guess the right way. However, if instead you were writing a program then you might have it try every case—you might do a breadth-first traversal of the tree of all derivations—until you found a success.

The American philosopher and Hall of Fame baseball catcher Yogi 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.



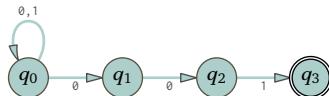
Yogi Berra  
1925–2015

The prototypical example is the celebrated Traveling Salesman problem, that of finding the shortest circuit of 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 and returns back to where it began in less than, say, 16 000 kilometers. We start at Montpelier, the capital of Vermont. From there, we could fork a process for each potential next capital, making forty seven new processes. The process that is assigned Concord, New Hampshire, for instance, 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 trip, but if even one succeeds then we consider the overall search a success.

That computation description is nondeterministic in that while it is happening the machine is simultaneously in many different states. It imagines an unboundedly-parallel machine, where whenever there is a desire for an additional computing agent, a CPU, one is available.<sup>†</sup> Such a machine is angelic in that whenever it wants more computational resources, those resources just appear.

We will have two ways to think about nondeterminism, two mental models.<sup>‡</sup> The first was 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  $\emptyset$ . It also has states with a deficit of edges; e.g., no 1 arrow leaves  $q_1$ , so if it is in that state and reads that input then it passes to no state at all.



The graphic below shows what happens with input  $00001$ . We take the computation history as a tree. For example, on the first  $0$  the computation splits in two, so the machine is now in two states at once.



Persian angel,  
1555

## 2.2 ANIMATION: Steps in the nondeterministic computation.

When we considered the forking approach to string derivations or to the Traveling Salesman, we observed that if a solution exists then some child process would find it. The same happens here; there is a branch of the computation tree that

<sup>†</sup>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.

<sup>‡</sup>While these models are helpful in learning and thinking about nondeterminism, they are not part of the formal definitions and proofs.

accepts the input string. There are also branches that are not successful. The one at the bottom dies after step 2 because when the present state  $q_2$  and the input is 0 this machine passes to no-state.<sup>†</sup> The branch at the top never dies but also does not accept the input. However, we don't care about unsuccessful branches. We only care that there is a successful one. So we will define that a nondeterministic machine accepts an input if the computation tree has at least one branch that accepts the input.

The machine in the above example accepts a string if it ends in two 0's and a 1. Having been fed the input  $\sigma = 00001$ , the problem that the machine faces is: when it should stop going around  $q_0$ 's loop and start to the right? Our definition has the machine accepting this input so the machine has solved this problem—viewed from the outside we could say, perhaps a bit fancifully, that the machine has correctly guessed. This is our second model for nondeterminism. We will imagine programming by calling a function, some  $\text{amb}(\sigma)$ , that somehow guess a successful sequence.

Saying that the machine guesses is jarring. Based on programming classes, a person's intuition may well be that "guessing" is not mechanically accomplishable. Instead, we can imagine that the machine is furnished with the answer ("go around twice, then off to the right") and only has to check it. This way of expressing the second mental model is demonic because the furnisher seems to be a supernatural being who somehow knows answers that cannot otherwise be found, but you are suspicious and must check that the answer is not a trick. Under this model, a nondeterministic computation accepts the input if there exists a branch of the computation tree that a deterministic machine, if told what branch to take, could verify.

Below we shall describe nondeterminism using both paradigms: as a machine being in multiple states at once and also as a machine guessing. As mentioned above, here we will do that for Finite State machines but in the fifth chapter we will return to it in the context of Turing machines.

**Definition** A nondeterministic Finite State machine's next-state function does not output single states, it outputs sets of states.

2.3 **DEFINITION** A **nondeterministic Finite State machine**  $\langle Q, q_{\text{start}}, F, \Sigma, \Delta \rangle$  consists of a finite **set of states**  $Q$ , one of which is the **start state**  $q_{\text{start}}$ , a subset  $F \subseteq Q$  of **accepting states or final states**, a finite **input alphabet** set  $\Sigma$ , and a **next-state function**  $\Delta: Q \times \Sigma \rightarrow \mathcal{P}(Q)$ .

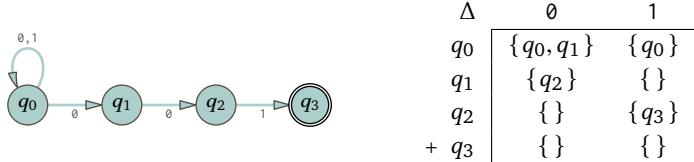
We will use these machines in three ways. First, they are useful in practice; both below and in the exercises are examples of jobs that are more easily solved in this way. Second, we will use them to prove Kleene's Theorem, Theorem 3.10. Finally, they give us an initial encounter with nondeterminism, which is critical for this book's fifth chapter.



Flauros, Duke of Hell

<sup>†</sup>No-state cannot be an accepting state, since it isn't a state at all.

- 2.4 EXAMPLE This is Example 2.1's nondeterministic Finite State machine, along with its transition function.



In this nondeterministic machine the entries of the array are not states, they are sets of states.

Nondeterministic machines may seem conceptually fuzzy so the formalities are a help. Contrast these definitions with the ones for deterministic machines.

A **configuration** is a pair  $C = \langle q, \tau \rangle$ , where  $q \in Q$  and  $\tau \in \Sigma^*$ . A machine starts with an **initial configuration**  $C_0 = \langle q_0, \tau_0 \rangle$ . The string  $\tau_0$  is the **input**.

Following the initial configuration there may be one or more sequences of **transitions**. Suppose that there is a machine configuration  $C_s = \langle q, \tau_s \rangle$ . For  $s \in \mathbb{N}^+$ , in the case where  $\tau_s$  is not the empty string, a transition pops the string's leading symbol  $c$  to get  $\tau_{s+1}$ , takes the machine's next state to be a member  $\hat{q}$  of the set  $\Delta(q, c)$  and then takes a subsequent configuration to be  $C_{s+1} = \langle \hat{q}, \tau_{s+1} \rangle$ . Denote that two configurations are connected by a transition with  $C_s \vdash C_{s+1}$ . The other case is that  $\tau_s$  is the empty string. This is a **halting configuration**,  $C_h$ . After  $C_h$ , no transitions follow.

A **nondeterministic Finite State machine computation** is a sequence of transitions that ends in a halting configuration,  $C_0 = \langle q_0, \tau_0 \rangle \vdash C_1 \vdash C_2 \vdash \dots \vdash C_h = \langle q, \varepsilon \rangle$ . From an initial configuration there may be many such sequences. If at least one ends with a halting state, with  $q \in F$ , then the machine **accepts** the input  $\tau_0$ , otherwise it **rejects**  $\tau_0$ .

- 2.5 EXAMPLE For the nondeterministic machine of Example 2.1, the graphic shows this allowed sequence of transitions.

$$\langle q_0, 00001 \rangle \vdash \langle q_0, 0001 \rangle \vdash \langle q_0, 001 \rangle \vdash \langle q_1, 01 \rangle \vdash \langle q_2, 1 \rangle \vdash \langle q_3, \varepsilon \rangle$$

Because this sequence ends in an accepting state, the machine accepts 00001.

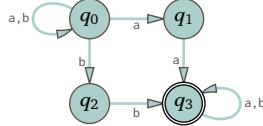
- 2.6 DEFINITION For a nondeterministic Finite State machine  $\mathcal{M}$ , the set of accepted strings is the **language of the machine**  $\mathcal{L}(\mathcal{M})$ , or the language **recognized**, (or **accepted**), by that machine.<sup>†</sup>

We will also adapt the definition of the **extended transition function**  $\hat{\Delta}: \Sigma^* \rightarrow Q$ . Fix a nondeterministic  $\mathcal{M}$  with transition function  $\Delta: Q \times \Sigma \rightarrow Q$ . Start with  $\hat{\Delta}(\varepsilon) = \{q_0\}$ . Where  $\hat{\Delta}(\tau) = \{q_{i_0}, q_{i_1}, \dots, q_{i_k}\}$  for  $\tau \in \Sigma^*$ , define  $\hat{\Delta}(\tau \cdot t) =$

<sup>†</sup>Below we will define something called  $\varepsilon$  transitions that make 'recognized' the right idea here, instead of 'decided'.

$\Delta(q_{i_0}, t) \cup \Delta(q_{i_1}, t) \cup \dots \cup \Delta(q_{i_k}, t)$  for any  $t \in \Sigma$ . Then the machine accepts  $\sigma \in \Sigma^*$  if and only if any element of  $\hat{\Delta}(\sigma)$  is a final state.

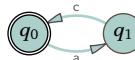
- 2.7 EXAMPLE The language recognized by this nondeterministic machine



is the set of strings containing the substring aa or bb. For instance, the machine accepts abaaba because there is a sequence of allowed transitions ending in an accepting state, namely this one.

$$\langle q_0, \text{abaaba} \rangle \vdash \langle q_0, \text{baaba} \rangle \vdash \langle q_0, \text{aabaa} \rangle \vdash \langle q_1, \text{aba} \rangle \vdash \langle q_2, \text{ba} \rangle \vdash \langle q_2, \text{a} \rangle \vdash \langle q_2, \varepsilon \rangle$$

- 2.8 EXAMPLE With  $\Sigma = \{a, b, c\}$ , this nondeterministic machine



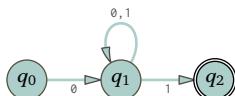
recognizes the language  $\{(ac)^n \mid n \in \mathbb{N}\} = \{\varepsilon, ac, acac, \dots\}$ . The symbol b isn't attached to any arrow so it won't play a part in any accepting string.

Often a nondeterministic Finite State machines is easier to write than a deterministic machine that does the same job.

- 2.9 EXAMPLE This nondeterministic machine that accepts any string whose next to last character is a, on the left, is simpler than the deterministic machine on the right.



- 2.10 EXAMPLE This machine accepts  $\{\sigma \in \mathbb{B}^* \mid \sigma = 0^\wedge \tau^\wedge 1 \text{ where } \tau \in \mathbb{B}^*\}$ .



- 2.11 EXAMPLE This is a garage door opener listener that waits to hear the remote control send the signal 0101110. That is, it recognizes the language  $\{\sigma^\wedge 0101110 \mid \sigma \in \mathbb{B}^*\}$ .



- 2.12 REMARK Having seen a couple of examples we pause to again acknowledge, as we did when we discussed the angel and the demon, that something about nondeterminism is unsettling. If we feed  $\tau = 010101110$  to the prior example's

listener then it accepts.

$$\begin{aligned} \langle q_0, 010101110 \rangle &\vdash \langle q_0, 10101110 \rangle \vdash \langle q_0, 0101110 \rangle \vdash \langle q_1, 101110 \rangle \\ &\vdash \langle q_2, 01110 \rangle \vdash \langle q_3, 1110 \rangle \vdash \langle q_4, 110 \rangle \vdash \langle q_5, 10 \rangle \vdash \langle q_6, 0 \rangle \vdash \langle q_7, \varepsilon \rangle \end{aligned}$$

But the machine's chain of states is set up for the string  $0101110$  that begins with two  $01$ 's while  $\tau$  begins with three. How can the machine guess that it should ignore the first  $01$  but act on the second? Of course, in mathematics we can consider whatever we can define precisely. But we have so far studied devices that are in principle physically realizable and so this may seem to be a shift.

It is not. We will next show how to convert any nondeterministic Finite State machine into a deterministic one that does the same job. So we can take a nondeterministic Finite State machine to be an abbreviation, a shorthand, a convenience. This obviates at least some of the paradox of guessing, at least for Finite State machines.

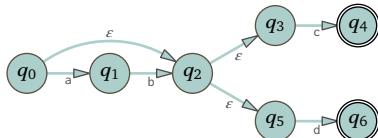
**$\varepsilon$  transitions** Another extension, beyond nondeterminism, is to allow  $\varepsilon$  transitions or  $\varepsilon$  moves. We alter the definition of a nondeterministic Finite State machine, so that instead of  $\Delta: Q \times \Sigma \rightarrow \mathcal{P}(Q)$ , the transition function's signature is  $\Delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$ .<sup>†</sup> The associated behavior is that the machine can transition spontaneously, without consuming any input.<sup>‡</sup>

- 2.13 EXAMPLE This machine recognizes valid integer representations. Note the  $\varepsilon$  between  $q_0$  and  $q_1$ .



Because of the  $\varepsilon$  it can accept strings that do not start with a  $+$  or  $-$  sign. For instance, with input 123 the machine can begin by following the  $\varepsilon$  transition to state  $q_1$ , then read the 1 and transition to  $q_2$ , and stay there while processing the 2 and 3. This is a branch of the computation tree accepting the input, and so the string 123 is in the machine's language.

- 2.14 EXAMPLE A machine may follow two or more  $\varepsilon$  transitions. From  $q_0$  this machine may stay in that state, or transition to  $q_2$ , or  $q_3$ , or  $q_5$ , all without consuming any input.



That is, the language of this machine is the four element set  $\mathcal{L} = \{abc, abd, ac, ad\}$ .

<sup>†</sup>Assume  $\varepsilon \notin \Sigma$    <sup>‡</sup>Or, think of it as transitioning on consuming the empty string  $\varepsilon$ .

We can give a precise definition of the action of a nondeterministic Finite State machine with  $\varepsilon$  transitions.

First we define the collection of states reachable by  $\varepsilon$  moves from a given state. For that we use  $E: Q \times \mathbb{N} \rightarrow \mathcal{P}(Q)$  where  $E(q, i)$  is the set of states reachable from  $q$  within at most  $i$ -many  $\varepsilon$  transitions. That is, set  $E(q, 0) = \{q\}$  and where  $E(q, i) = \{q_{i_0}, \dots, q_{i_k}\}$ , set  $E(q, i+1) = E(q, i) \cup \Delta(q_{i_0}, \varepsilon) \cup \dots \cup \Delta(q_{i_k}, \varepsilon)$ . Observe that these are nested,  $E(q, 0) \subseteq E(q, 1) \subseteq \dots$  and that each is a subset of  $Q$ . But  $Q$  has only finitely many states so there must be an  $\hat{i} \in \mathbb{N}$  where the sequence of sets stops growing,  $E(q, \hat{i}) = E(q, \hat{i}+1) = \dots$ . Define the  $\varepsilon$  closure function  $\hat{E}: Q \rightarrow \mathcal{P}(Q)$  by  $\hat{E}(q) = E(q, \hat{i})$ .

With that, we are ready to describe the machine's action. As before, a **configuration** is a pair  $C = \langle q, \tau \rangle$ , where  $q \in Q$  and  $\tau \in \Sigma^*$ . A machine starts with some **initial configuration**  $C_0 = \langle q_0, \tau_0 \rangle$ , where the string  $\tau_0$  is the **input**.

The key description is that of a **transition**. Consider a configuration  $C_s = \langle q, \tau_s \rangle$  for  $s \in \mathbb{N}$  and suppose that  $\tau_s$  is not the empty string. We will describe a configuration  $C_{s+1} = \langle \hat{q}, \tau_{s+1} \rangle$  that is related to the given one by  $C_s \vdash C_{s+1}$ . (As with the earlier description of nondeterministic machines without  $\varepsilon$  transitions, there may be more than one configuration related in this way to  $C_s$ .)

The string is easy; just pop the leading character to get  $\tau_s = t^\frown \tau_{s+1}$  where  $t \in \Sigma$ . To get a legal state  $\hat{q}$ : (i) find the  $\varepsilon$  closure  $\hat{E}(q) = \{q_{s_0}, \dots, q_{s_k}\}$ , (ii) let  $\bar{q}$  be an element of the set  $\Delta(q_{s_0}, t) \cup \Delta(q_{s_1}, t) \cup \dots \cup \Delta(q_{s_k}, t)$ , and (iii) take  $\hat{q}$  to be an element of the  $\varepsilon$  closure  $\hat{E}(\bar{q})$ .

If  $\tau_s$  is the empty string then this is a **halting configuration**,  $C_h$ . No transitions follow  $C_h$ .

A **nondeterministic Finite State machine computation** is a sequence of transitions ending in a halting configuration,  $C_0 = \langle q_0, \tau_0 \rangle \vdash C_1 \vdash C_2 \vdash \dots \vdash C_h = \langle q, \varepsilon \rangle$ . From a given  $C_0$  there may be many such sequences. If at least one ends with a halting state, having  $q \in F$ , then the machine **accepts** the input  $\tau_0$ , otherwise it **rejects**  $\tau_0$ .

With that, we will modify the definition of the extended transition function  $\hat{\Delta}: \Sigma^* \rightarrow Q$  from section 2. Begin by defining  $\hat{\Delta}(\varepsilon) = \hat{E}(q_0)$ . Then the rule for going from a string to its extension is that for  $\tau \in \Sigma^*$  and where  $\hat{\Delta}(\tau) = \{q_{i_0}, q_{i_1}, \dots, q_{i_k}\}$ .

$$\hat{\Delta}(\tau^\frown t) = \hat{E}(\Delta(q_{i_0}, t)) \cup \dots \cup \hat{E}(\Delta(q_{i_k}, t)) \quad \text{for } t \in \Sigma$$

Observe that this nondeterministic machine with  $\varepsilon$  transitions accepts a string  $\sigma \in \Sigma^*$  if any one of the states in  $\hat{\Delta}(\sigma)$  is a final state.

- 2.15 **REMARK** Certainly these are an intricate set of definitions but they demonstrate something important. In the examples and the homework we use informal terms such as “guess” and “demon.” Nonetheless, we can perfectly well give definitions and results with full precision.

- 2.16 EXAMPLE For the machine of Example 2.14, this sequence shows that it accepts abc

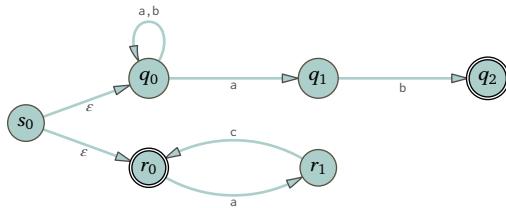
$$\langle q_0, \text{abc} \rangle \vdash \langle q_1, \text{bc} \rangle \vdash \langle q_2, \text{c} \rangle \vdash \langle q_3, \text{c} \rangle \vdash \langle q_4, \varepsilon \rangle$$

(note the  $\varepsilon$  transition between  $q_2$  and  $q_3$ ). This sequence shows that it also accepts the input string d.

$$\langle q_0, \text{d} \rangle \vdash \langle q_5, \text{d} \rangle \vdash \langle q_6, \varepsilon \rangle$$

One reason to consider  $\varepsilon$  transitions is that they can make solving a complex job much easier.

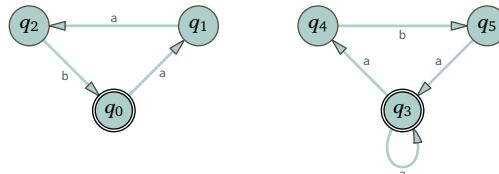
- 2.17 EXAMPLE An  $\varepsilon$  transition can put two machines together with a parallel connection. This shows a machine whose states are named with  $q$ 's combined with one whose states are named with  $r$ 's.



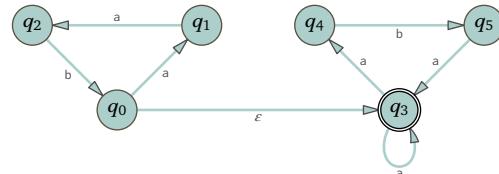
The top nondeterministic machine's language is  $\{ \sigma \in \Sigma^* \mid \sigma \text{ ends in ab} \}$  and the bottom machine's language is  $\{ \sigma \in \Sigma^* \mid \sigma = (\text{ac})^n \text{ for some } n \in \mathbb{N} \}$ , where  $\Sigma = \{ a, b, c \}$ . The language for the entire machine is the union.

$$\{ \sigma \in \Sigma^* \mid \text{either } \sigma \text{ ends in ab or } \sigma = (\text{ac})^n \text{ for } n \in \mathbb{N} \}$$

- 2.18 EXAMPLE An  $\varepsilon$  transition can also make a serial connection between machines. The machine on the left below recognizes the language  $\{ (\text{aab})^m \mid m \in \mathbb{N} \}$  and the machine on the right recognizes  $\{ (\text{a|aba})^n \mid n \in \mathbb{N} \}$ .



If we insert an  $\varepsilon$  bridge

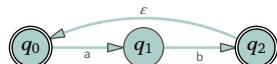


from each of the left side's final states to the right side's initial state (in this example there happens to be only one such state) then the combined machine accepts strings in the concatenation of those languages.

$$\mathcal{L}(\mathcal{M}) = \{\sigma \in \{a, b\}^* \mid \sigma = (aab)^m(a|aba)^n \text{ for } m, n \in \mathbb{N}\}$$

For example, it accepts aabaababa and aabaabaaa.

- 2.19 EXAMPLE An  $\epsilon$  transition edge can also produce the Kleene star of a nondeterministic machine. For instance, without the  $\epsilon$  edge this machine's language is  $\{\epsilon, ab\}$ , while with it the language is  $\{(ab)^n \mid n \in \mathbb{N}\}$ .



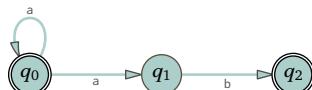
**Equivalence of the machine types** We next prove that nondeterminism does not change what we can do with Finite State machines.

- 2.20 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  $\epsilon$  transitions.

Inclusion in one direction is easy; any deterministic machine is, basically, a nondeterministic machine. That is, 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, by starting with a nondeterministic machine with  $\epsilon$  transitions and building a deterministic machine that recognizes the same language. The two examples below show the **powerset construction**. We will give a complete description of the algorithm just before the second example. (We won't give a proof that it works simply because the examples are entirely convincing.)

- 2.21 EXAMPLE Consider this nondeterministic machine,  $\mathcal{M}_N$ , with no  $\epsilon$  transitions.



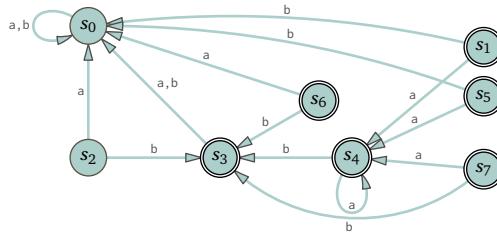
The associated deterministic machine  $\mathcal{M}_D$  is shown below. Each member is a set of  $\mathcal{M}_N$ 's states,  $s_i = \{q_{i_1}, \dots, q_{i_k}\}$ . The start state of  $\mathcal{M}_D$  is  $s_1 = \{q_0\}$ , and a state of  $\mathcal{M}_D$  is accepting if any of its elements are accepting states in  $\mathcal{M}_N$ .

As an illustration of constructing the transition table, suppose that  $\mathcal{M}_N$  is in  $s_5 = \{q_0, q_2\}$  and is reading a. The next state combines the next states due to  $q_0$

with the next states due to  $q_2$ . Thus,  $\Delta_D(s_5, a) = \{ q_0, q_1 \}$ , which is  $s_4$ .

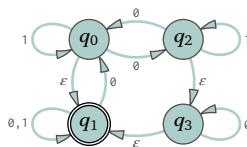
	$\Delta_D$	a	b
$s_0 = \{ \}$	$s_0$	$s_0$	
$+ s_1 = \{ q_0 \}$	$s_4$	$s_0$	
$s_2 = \{ q_1 \}$	$s_0$	$s_3$	
$+ s_3 = \{ q_2 \}$	$s_0$	$s_0$	
$+ s_4 = \{ q_0, q_1 \}$	$s_4$	$s_3$	
$+ s_5 = \{ q_0, q_2 \}$	$s_4$	$s_0$	
$+ s_6 = \{ q_1, q_2 \}$	$s_0$	$s_3$	
$+ s_7 = \{ q_0, q_1, q_2 \}$	$s_4$	$s_3$	

Besides the notational convenience, naming the sets of states as  $s_i$ 's makes clear that  $\mathcal{M}_D$  is deterministic. So does its transition graph.



We next extend the powerset construction to handle  $\epsilon$  transitions. Basically, we will follow the  $\epsilon$ 's. For a state  $q$ , the  $\epsilon$  closure  $\hat{E}(q)$  is the set of the states that are reachable from  $q$  via some number of  $\epsilon$  transitions. This includes  $q$  itself, as it is the result of zero-many transitions. The  $\epsilon$  closure of a set of states is the union of the  $\epsilon$  closures of the members.

## 2.22 EXAMPLE In this nondeterministic machine



these are the  $\epsilon$  closures.

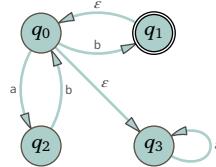
state $q$	$q_0$	$q_1$	$q_2$	$q_3$
$\epsilon$ closure $\hat{E}(q)$	$\{ q_0, q_1 \}$	$\{ q_1 \}$	$\{ q_1, q_2, q_3 \}$	$\{ q_1, q_3 \}$

We can now give the full algorithm for starting with a nondeterministic machine  $\mathcal{M}_N$  and constructing a deterministic machine  $\mathcal{M}_D$  with the same behavior, even if  $\mathcal{M}_N$  has  $\epsilon$  transitions. Elements of  $\mathcal{M}_D$  are sets of states from  $\mathcal{M}_N$ . The start state of  $\mathcal{M}_D$  is the  $\epsilon$  closure of  $\{ q_0 \}$ . A state of  $\mathcal{M}_D$  is accepting if it contains any element of the  $\epsilon$  closure of any of  $\mathcal{M}_N$ 's accepting states.

As to the next-state function, consider a subset of  $\mathcal{M}_N$ 's states  $S$  and a tape

character  $x$ . We compute  $\Delta_D(S, x)$  in three steps. First, gather the  $\varepsilon$  closures of all of the  $q \in S$ , giving a set  $A = \cup_{q \in S} \hat{E}(q)$ . This is a subset of the states of  $\mathcal{M}_N$ . Second, apply  $\mathcal{M}_N$ 's next state function to  $A$ 's elements, giving  $B = \cup_{a \in A} \Delta_N(a, x)$ . This is also a subset of the states of  $\mathcal{M}_N$ . Finally, gathering the  $\varepsilon$  closures of  $B$ 's states gives  $\Delta_D(S, x) = \cup_{b \in B} \hat{E}(b)$ .

2.23 EXAMPLE Consider this nondeterministic machine.



The table below goes through the computation. The start state is  $\hat{E}(q_0) = s_7$ . A state is accepting if it contains  $q_1$ .

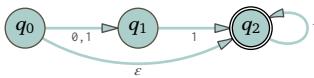
The next-state computation is involved, so the table includes intermediate steps. The column labeled  $A_i$  holds the  $\varepsilon$  closure of  $s_i$ 's states,  $\cup_{q \in s_i} \hat{E}(q)$ . This is used for both inputs a and b, so listing it once is a savings. The column labeled  $B_{i,a}$  holds  $\cup_{q \in A} \Delta_N(q, a)$  (note the a's), while  $B_{i,b}$  holds  $\cup_{q \in A} \Delta_N(q, b)$ . Rows have a dash when their  $A_i$  is equal to an  $A_j$  from an earlier row, so there is no need to repeat the computation.

$\Delta_D$	$A_i$	$B_{i,a}$	$\hat{E}(B_{i,a})$	$B_{i,b}$	$\hat{E}(B_{i,b})$
$s_0 = \{\}$	{}	{}	$s_0$	{}	$s_0$
$+ s_1 = \{q_0\}$	{ $q_0, q_3$ }	{ $q_2, q_3$ }	$s_{10}$	{ $q_1$ }	$s_{12}$
$+ s_2 = \{q_1\}$	{ $q_0, q_1, q_3$ }	{ $q_2, q_3$ }	$s_{10}$	{ $q_1$ }	$s_{12}$
$+ s_3 = \{q_2\}$	{ $q_2$ }	{}	$s_0$	{ $q_0$ }	$s_7$
$+ s_4 = \{q_3\}$	{ $q_3$ }	{ $q_3$ }	$s_4$	{}	$s_0$
$+ s_5 = \{q_0, q_1\}$	{ $q_0, q_1, q_3$ }	—	$s_{10}$	—	$s_{12}$
$+ s_6 = \{q_0, q_2\}$	{ $q_0, q_2, q_3$ }	{ $q_2, q_3$ }	$s_{10}$	{ $q_0, q_1$ }	$s_{12}$
$+ s_7 = \{q_0, q_3\}$	{ $q_0, q_3$ }	—	$s_{10}$	—	$s_{12}$
$+ s_8 = \{q_1, q_2\}$	{ $q_0, q_1, q_2, q_3$ }	{ $q_2, q_3$ }	$s_{10}$	{ $q_0, q_1$ }	$s_{12}$
$+ s_9 = \{q_1, q_3\}$	{ $q_0, q_1, q_3$ }	—	$s_{10}$	—	$s_{12}$
$s_{10} = \{q_2, q_3\}$	{ $q_2, q_3$ }	{ $q_3$ }	$s_4$	{ $q_0$ }	$s_7$
$+ s_{11} = \{q_0, q_1, q_2\}$	{ $q_0, q_1, q_2, q_3$ }	—	$s_{10}$	—	$s_{12}$
$+ s_{12} = \{q_0, q_1, q_3\}$	{ $q_0, q_1, q_3$ }	—	$s_{10}$	—	$s_{12}$
$+ s_{13} = \{q_0, q_2, q_3\}$	{ $q_0, q_2, q_3$ }	—	$s_{10}$	—	$s_{12}$
$+ s_{14} = \{q_1, q_2, q_3\}$	{ $q_0, q_1, q_2, q_3$ }	—	$s_{10}$	—	$s_{12}$
$+ s_{15} = \{q_0, q_1, q_2, q_3\}$	{ $q_0, q_1, q_2, q_3$ }	—	$s_{10}$	—	$s_{12}$

## IV.2 Exercises

2.24 Give the transition function for the machine of Example 2.7, and of Example 2.8.

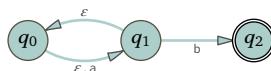
- ✓ 2.25 Consider this machine.



- (A) Does it accept the empty string? (B) The string  $\emptyset$ ? (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 should the prof respond?

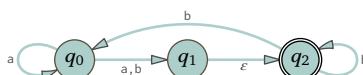
2.27 Your friend objects, “Epsilon transitions don’t make any sense because the machine below will never get its first step done; it just endlessly follows the epsilons.” Correct their misimpression.



- ✓ 2.28 Give the transition graph of a nondeterministic Finite State machine that accepts valid North American local phone numbers, strings of the form  $d^3-d^4$ , with three digits, followed by a hyphen character, and then four digits.

2.29 Draw the transition graph of a nondeterministic machine that recognizes the language  $\{ \sigma = \tau_0\tau_1\tau_2 \in \mathbb{B}^* \mid \tau_0 = 1, \tau_2 = 1, \text{ and } \tau_1 = (00)^k \text{ for some } k \in \mathbb{N} \}$ .

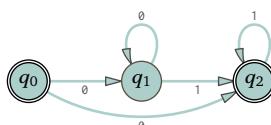
2.30 This machine has  $\Sigma = \{ a, b \}$ .



- (A) What is the  $\epsilon$  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  $\Delta$  for the machine in the prior exercise. It should have three columns, for  $a$ ,  $b$ , and  $\epsilon$ .

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)  $\emptyset$ ?  $1$ ? (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?

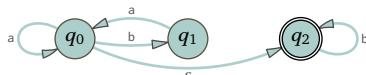
- ✓ 2.33 Give diagrams for nondeterministic Finite State machines that recognize the given language and that have the given number of states. Use  $\Sigma = \mathbb{B}$ .
- $\mathcal{L}_0 = \{\sigma \mid \sigma \text{ ends in } 00\}$ , having three states
  - $\mathcal{L}_1 = \{\sigma \mid \sigma \text{ has the substring } 0110\}$ , with five states
  - $\mathcal{L}_2 = \{\sigma \mid \sigma \text{ contains an even number of } 0\text{'s or exactly two } 1\text{'s}\}$ , with six states
  - $\mathcal{L}_3 = \{0\}^*$ , with one state

2.34 This table

$\Delta$	a	b
$q_0$	$\{q_0\}$	$\{q_1, q_2\}$
$q_1$	$\{q_3\}$	$\{q_3\}$
$q_2$	$\{q_1\}$	$\{q_3\}$
+ $q_3$	$\{q_3\}$	$\{q_3\}$

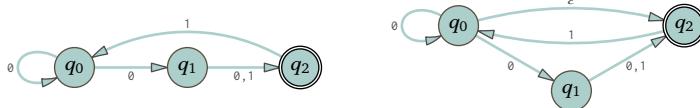
gives the next-state function for a nondeterministic Finite State machine. (A) Draw the transition graph. (B) What is the recognized language? (C) Give the next-state table for a deterministic machine that recognizes the same language.

- ✓ 2.35 Draw the graph of a nondeterministic Finite State machine over  $\mathbb{B}$  that accepts strings with the suffix 111000111.
- ✓ 2.36 For each draw the transition graph for a Finite State machine, which may be nondeterministic, that accepts the given strings from  $\{a, b\}^*$ .
- Accepted strings have a second character of a and next to last character of b.
  - Accepted strings have second character a and the next to last character is also a.
- ✓ 2.37 Make a table giving the  $\epsilon$  closure function  $\hat{E}$  for the machine in Example 2.14.
- ✓ 2.38 Find the nondeterministic Finite State machine that accepts all bitstrings that begin with 10. Use the algorithm given above to produce the transition function table of a deterministic machine that does the same.
- 2.39 Find a nondeterministic Finite State machine that recognizes this language of three words:  $\mathcal{L} = \{\text{cat}, \text{cap}, \text{carumba}\}$ .
- 2.40 Give a nondeterministic Finite State machine over  $\Sigma = \{a, b, c\}$  recognizing the language of strings that omit at least one of the characters in the alphabet.
- ✓ 2.41 What is the language of this nondeterministic machine with  $\epsilon$  transitions?



2.42 Find a deterministic machine and a nondeterministic machine that recognizes the set of bitstrings containing the substring 11. You need not construct the deterministic machine from the other; you can just construct it using any native wit that you may possess.

- ✓ 2.43 For each, follow the construction above to make a deterministic machine with the same language.



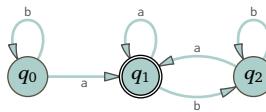
- ✓ 2.44 For each give a nondeterministic Finite State machine over  $\Sigma = \{0, 1, 2\}$ .
- The machine recognizes the language of strings whose final character appears exactly twice in the string.
  - The machine recognizes the language of strings whose final character appears exactly twice in the string, but in between those two occurrences is no higher digit.
- ✓ 2.45 For each give a nondeterministic Finite State machine with  $\epsilon$  transitions over that recognizes each language over  $\mathbb{B}$ . (The deterministic machines for some of these are much harder.)
- In each string, every 0 is followed immediately by a 1.
  - Each string contains 000 followed, possibly with some intermediate characters, by 001.
  - In each string the first two characters equals the final two characters, in order. (*Hint:* what about 000?)
  - There is either an even number of 0's or an odd number of 1's.
- 2.46 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.
- 2.47 A grammar is **right linear** if every production rule has the form  $\langle n_1 \rangle \rightarrow x \langle n_2 \rangle$ , where the right side has a single terminal followed by a single nonterminal. With this right linear grammar we can associate this nondeterministic Finite State machine.
- $$\begin{aligned} S &\rightarrow a A \\ A &\rightarrow a A \mid b B \\ B &\rightarrow b B \mid b \end{aligned}$$
- 
- (A) Give three strings from the language of the grammar and show that they are accepted by the machine.
- (B) Describe the language of the grammar and the machine.
- 2.48 Decide whether each problem is solvable or unsolvable by a Turing machine.
- $\mathcal{L}_{DFA} = \{\langle M, \sigma \rangle \mid \text{the deterministic Finite State machine } M \text{ accepts } \sigma\}$
  - $\mathcal{L}_{NFA} = \{\langle M, \sigma \rangle \mid \text{the nondeterministic machine } M \text{ accepts } \sigma\}$
- 2.49 (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<sup>†</sup> was studying a mathematical model of neurons. These are like Finite State machines in that they do not have scratch memory. He noted patterns to the languages that are recognized by such devices.

For instance, this Finite State machine



Stephen  
Kleene  
1909–1994

accepts strings that have some number of b's (perhaps zero many), followed by at least one a, possibly then followed by at least one b, and then at least one a. Kleene introduced a convenient way, called regular expressions, to denote constructs such as “any number of” and “followed by.” He gave the definition in the first subsection below and supported it with the theorem in the second subsection.

**Definition** A regular expression is a string that describes a language. We will introduce these with a few examples. These use the alphabet  $\Sigma = \{a, \dots, z\}$ .

- 3.1 **EXAMPLE** The string  $h(a|e|i|o|u)t$  is a regular expression describing strings that start with h, have a vowel in the middle, and end with t. That is, this regular expression describes the language consisting of five words of three letters each,  $\mathcal{L} = \{\text{hat}, \text{het}, \text{hit}, \text{hot}, \text{hut}\}$ .

The pipe ‘|’ operator, which is a kind of ‘or’, and the parentheses, which provide grouping, are not part of the strings being described; they are **metacharacters**.

Besides the pipe operator and parentheses, the regular expression also uses concatenation since the initial h is concatenated with  $(a|e|i|o|u)$ , which in turn is concatenated with t.

- 3.2 **EXAMPLE** The next regular expression is  $ab^*c$ . It describes the language whose words begin with an a, followed by any number of b's (including possibly zero-many b's), and end with a c. So ‘\*’ means ‘repeat the prior thing any number of times’. This regular expression describes the language  $\mathcal{L} = \{ac, abc, abbc, \dots\}$ .

- 3.3 **EXAMPLE** There is an interaction between pipe and star. Consider the regular expression  $(b|c)^*$ . It could mean either ‘any number of repetitions of picking a b or c’ or ‘pick a b or c and repeat that character any number of times’.

The definition has it mean the first. Thus the language described by  $a(b|c)^*$  consists of words starting with a and ending with any mixture of b's and c's, so that  $\mathcal{L} = \{a, ab, ac, abb, abc, acb, acc, \dots\}$ .

In contrast, to describe the language whose members begin with a and end

<sup>†</sup>Pronounced KLAY-nee. He was a student of Church, like Turing.

with any number of b's or any number of c's,  $\hat{\mathcal{L}} = \{ a, ab, abb, \dots, ac, acc, \dots \}$ , use the regular expression  $a(b^*|c^*)$ .

That is, the rules for operator precedence are: star binds most tightly, then concatenation, then the pipe alternation operator,  $|$ . To get another order, use parentheses.

- 3.4 DEFINITION Let  $\Sigma$  be an alphabet not containing any of the metacharacters  $,$ ,  $|$ , or  $*$ . A **regular expression** over  $\Sigma$  is a string that can be derived from this grammar

$$\begin{aligned} \langle \text{regex} \rangle &\rightarrow \langle \text{concat} \rangle \\ &| \langle \text{regex} \rangle '| \langle \text{concat} \rangle \\ \langle \text{concat} \rangle &\rightarrow \langle \text{simple} \rangle \\ &| \langle \text{concat} \rangle \langle \text{simple} \rangle \\ \langle \text{simple} \rangle &\rightarrow \langle \text{char} \rangle \\ &| \langle \text{simple} \rangle ^* \\ &| ( \langle \text{regex} \rangle ) \\ \langle \text{char} \rangle &\rightarrow \emptyset | \varepsilon | x_0 | x_1 | \dots \end{aligned}$$

where the  $x_i$  characters are members of  $\Sigma$ .<sup>†</sup>

As to their semantics, what regular expressions mean, we will define that recursively. We start with the bottom line, the single-character regular expressions, and give the language that each describes. We will then do the forms on the other lines, for each interpreting it as the description of a language.

The language described by the single-character regular expression  $\emptyset$  is the empty set,  $\mathcal{L}(\emptyset) = \emptyset$ . The language described by the regular expression consisting of only the character  $\varepsilon$  is the one-element language consisting of only the empty string,  $\mathcal{L}(\varepsilon) = \{ \varepsilon \}$ . If the regular expression consists of just one character from the alphabet  $\Sigma$  then the language that it describes contains only one string and that string has only that single character, as in  $\mathcal{L}(a) = \{ a \}$ .

We finish by defining the semantics of the operations. Start with regular expressions  $R_0$  and  $R_1$  describing languages  $\mathcal{L}(R_0)$  and  $\mathcal{L}(R_1)$ . Then the pipe symbol describes the union of the languages, so that  $\mathcal{L}(R_0|R_1) = \mathcal{L}(R_0) \cup \mathcal{L}(R_1)$ . Concatenation of the regular expressions describes concatenation of the languages,  $\mathcal{L}(R_0 \cdot R_1) = \mathcal{L}(R_0) \cdot \mathcal{L}(R_1)$ . And the Kleene star of the regular expression describes the star of the language,  $\mathcal{L}(R_0^*) = \mathcal{L}(R_0)^*$ .

- 3.5 EXAMPLE Consider the regular expression  $aba^*$  over  $\Sigma = \{ a, b \}$ . It is the concatenation of  $a$ ,  $b$ , and  $a^*$ . The first describes the single-element language  $\mathcal{L}(a) = \{ a \}$ . Likewise, the second describes  $\mathcal{L}(b) = \{ b \}$ . Thus, the string  $ab$

---

<sup>†</sup>As we have done with other grammars, here we use the pipe symbol  $|$  as a metacharacter, to collapse rules with the same left side. But pipe also appears in regular expressions. For that usage we wrap it in single quotes, as ' $|$ '.

describes the concatenation of the two, another one-element language.

$$\begin{aligned}\mathcal{L}(ab) &= \mathcal{L}(a) \cap \mathcal{L}(b) = \{\sigma \in \Sigma^* \mid \sigma = \sigma_0 \cap \sigma_1 \text{ where } \sigma_0 \in \mathcal{L}(a) \text{ and } \sigma_1 \in \mathcal{L}(b)\} \\ &= \{ab\}\end{aligned}$$

The regular expression  $a^*$  describes the star of the language  $\mathcal{L}(a)$ , namely  $\mathcal{L}(a^*) = \{a^n \mid n \in \mathbb{N}\}$ . Concatenating it with  $\mathcal{L}(ab)$  gives this.

$$\begin{aligned}\mathcal{L}(aba^*) &= \{\sigma \in \Sigma^* \mid \sigma = \sigma_0 \cap \sigma_1 \text{ where } \sigma_0 \in \mathcal{L}(ab) \text{ and } \sigma_1 \in \mathcal{L}(a^*)\} \\ &= \{ab, aba, abaa, aba^3, \dots\} \\ &= \{aba^n \mid n \in \mathbb{N}\}\end{aligned}$$

We finish this subsection with some constructs that appear often. These examples use  $\Sigma = \{a, b, c\}$ .

- 3.6 EXAMPLE The language consisting of strings of  $a$ 's whose length is a multiple of three,  $\mathcal{L} = \{a^{3k} \mid k \in \mathbb{N}\} = \{\epsilon, aaa, aaaaa, \dots\}$ , is described by  $(aaa)^*$ .

Note that the empty string is a member of that language. A common gotcha is to forget that star is for any number of repetitions, including zero-many.

- 3.7 EXAMPLE To match any character we can list them all. The language over  $\Sigma = \{a, b, c\}$  of three-letter words ending in  $bc$  is  $\{abc, bbc, cbc\}$ . The regular expression  $(a|b|c)bc$  describes it. (In practice the alphabet can be very large so that listing all of the characters is impractical; see Extra A.)

- 3.8 EXAMPLE The regular expression  $a^*(\epsilon|b)$  describes the language of strings that have any number of  $a$ 's and optionally end in one  $b$ ,  $\mathcal{L} = \{\epsilon, b, a, ab, aa, aab, \dots\}$ .

Similarly, to describe the language consisting of words with between three and five  $a$ 's,  $\mathcal{L} = \{aaa, aaaa, aaaaa\}$  we can use  $aaa(\epsilon|a|aa)$ .

- 3.9 EXAMPLE The language  $\{b, bc, bcc, ab, abc, abcc, aab, \dots\}$  has words starting with any number of  $a$ 's (including zero-many  $a$ 's), followed by a single  $b$ , and then ending in fewer than three  $c$ 's. To describe it we can use  $a^*b(\epsilon|c|cc)$ .

**Kleene's Theorem** The next result justifies our study of regular expressions because it shows that they describe the languages of interest.

- 3.10 **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.

- 3.11 **LEMMA** If a language is described by a regular expression then there is a Finite State machine that recognizes that language.

*Proof* We will show that for any regular expression  $R$  there is a machine that accepts strings matching that expression. We use induction on the structure of

regular expressions.

Start with regular expressions consisting of a single character. If  $R = \emptyset$  then  $\mathcal{L}(R) = \{\}$  and the machine on the left below recognizes  $\mathcal{L}(R)$ . If  $R = \varepsilon$  then  $\mathcal{L}(R) = \{\varepsilon\}$  and the machine in the middle recognizes this language. If the regular expression is a character from the alphabet, such as  $R = a$ , then the machine on the right works.



We finish by handling the three operations. Let  $R_0$  and  $R_1$  be regular expressions. The inductive hypothesis gives a machine  $\mathcal{M}_0$  whose language is described by  $R_0$  and a machine  $\mathcal{M}_1$  whose language is described by  $R_1$ .

First consider alternation,  $R = R_0 \mid R_1$ . Create the machine recognizing the language described by  $R$  by joining those two machines in parallel: introduce a new state  $s$  and use  $\varepsilon$  transitions to connect  $s$  to the start states of  $\mathcal{M}_0$  and  $\mathcal{M}_1$ . See Example 2.17.

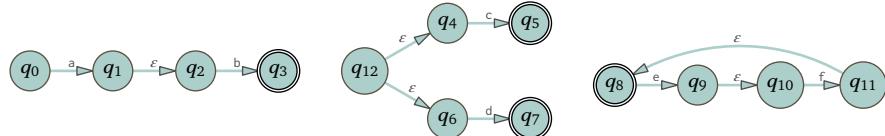
Next consider concatenation,  $R = R_0 \cdot R_1$ . Join the two machines serially: for each accepting state in  $\mathcal{M}_0$ , make an  $\varepsilon$  transition to the start state of  $\mathcal{M}_1$  and then convert all those accepting states of  $\mathcal{M}_0$  to be non-accepting states. See 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  $\varepsilon$  transition to the start state, and then make the start state an accepting state. See Example 2.19.  $\square$

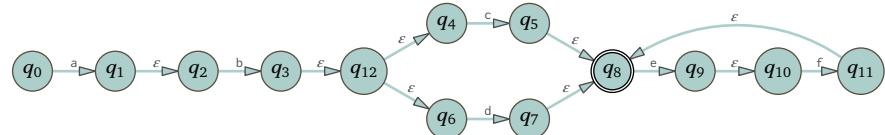
- 3.12 EXAMPLE Building a machine for the regular expression  $ab(c \mid d)(ef)^*$  starts with machines for the single characters.



Put these atomic components together



to get the complete machine.



This machine is nondeterministic. For a deterministic one use the conversion

process that we saw in the prior section.

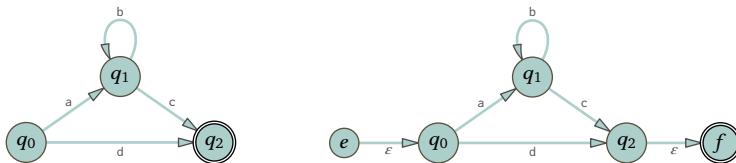
- 3.13 **LEMMA** Any language recognized by a Finite State machine is described by a regular expression.

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 the proof we will generalize transition graphs to allow edge labels that are regular expressions. We will eliminate states , keeping the recognized language the same. We will be done when there remain only two states, with one edge between them. That edge's label is the desired regular expression.

Before the proof, an example. Consider the machine on the left below.



The proof starts as above on the right by introducing a new start state guaranteed to have no incoming edges,  $e$ , and a new final state guaranteed to be unique,  $f$ . Then the proof eliminates  $q_1$  as below.



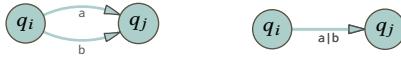
Clearly this machine recognizes the same language as the starting machine.

*Proof* Call the machine  $\mathcal{M}$ . If it has no accepting states then the regular expression is  $\emptyset$  and we are done. Otherwise, we will transform  $\mathcal{M}$  to a new machine,  $\hat{\mathcal{M}}$ , with the same language, on which we can execute 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 a  $\epsilon$  transition to  $f$  (by the prior paragraph there is at least one such accepting state). Change all the accepting states to non-accepting ones and then make  $f$  accepting.

Next introduce a new start state,  $e$ . Make a  $\epsilon$  transition between it and  $q_0$ , (Ensuring that  $\hat{\mathcal{M}}$  has at least two states allows us to handle machines of all sizes uniformly.)

Because the edge labels are regular expressions, we can arrange that from any  $q_i$  to any  $q_j$  is at most one edge, because if  $\mathcal{M}$  has more than one edge then in  $\hat{\mathcal{M}}$  use the pipe,  $|$ , to combine the labels, as here.

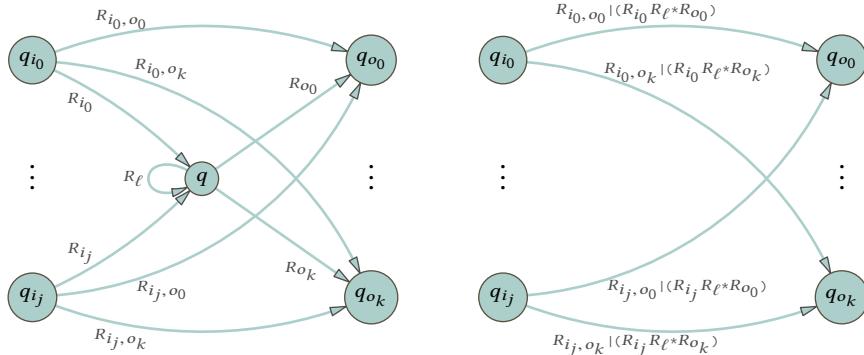


Do the same with loops, that is, cases where  $i = j$ . Like the prior transformations, clearly this does not change the language of accepted strings.

The last part of transforming to  $\hat{\mathcal{M}}$  is to drop any useless states. If a state node other than  $f$  has no outgoing edges then drop it along with the edges into it. The language of the machine will not change because this state cannot lead to an accepting state, since it doesn't lead anywhere, and this state is not itself accepting as only  $f$  is accepting.

Along the same lines, if a state node  $q$  is not reachable from the start  $e$  then can drop that node along with its incoming and outgoing edges. (The idea of unreachable is clear but for a formal definition see Exercise 3.34.)

With that,  $\hat{\mathcal{M}}$  is ready for state elimination. Below are before and after pictures. The before picture shows a state  $q$  to be eliminated. There are states  $q_{i_0}, \dots, q_{i_j}$  with an edge leading into  $q$ , and states  $q_{o_0}, \dots, q_{o_k}$  that receive an edge leading out of  $q$ . (By the setup work above,  $q$  has at least one incoming and at least one outgoing edge.) In addition,  $q$  may have a loop.

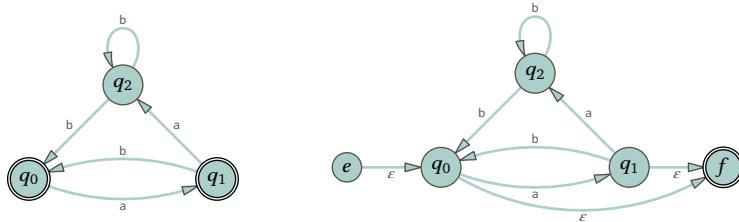


(Here is a subtle point: possibly some of the states shown on the left of each of the two pictures equal some shown on the right. For example, possibly  $q_{i_0}$  equals  $q_{o_0}$ . If so then the shown edge  $R_{i_0, o_0}$  is a loop.)

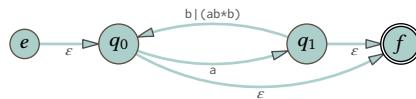
Eliminate  $q$  and the associated edges by making the replacements shown in the after picture. Observe that the set of strings taking the machine from any incoming state  $q_i$  to any outgoing state  $q_o$  is unchanged. So the language recognized by the machine is unchanged.

Repeat this elimination until all that remains are  $e$  and  $f$ , and the edge between them. (The machine has finitely many states so this procedure must eventually stop.) The desired regular expression is edge's label.  $\square$

- 3.14 EXAMPLE Consider  $\mathcal{M}$  on the left. Introduce  $e$  and  $f$  to get  $\hat{\mathcal{M}}$  on the right.



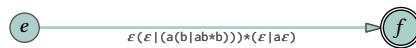
Start by eliminating  $q_2$ . In the terms of the proof's key step,  $q_1 = q_{i_0}$  and  $q_0 = q_{o_0}$ . The regular expressions are  $R_{i_0} = a$ ,  $R_{o_0} = b$ ,  $R_{i_0, o_0} = b$ , and  $R_\ell = b$ . That gives this machine.



Next eliminate  $q_1$ . There is one incoming node  $q_0 = q_{i_0}$  and two outgoing nodes  $q_0 = q_{o_0}$  and  $f = q_{o_1}$ . (Note that  $q_0$  is both an incoming and outgoing node; this is the subtle point mentioned in the proof.) The regular expressions are  $R_{i_0} = a$ ,  $R_{o_0} = b | (ab^*b)$ , and  $R_{o_1} = \epsilon$ .



All that remains is to eliminate  $q_0$ . The sole incoming node is  $e = q_{i_0}$  and the sole outgoing node is  $f = q_{o_0}$ , and so  $R_{i_0} = \epsilon$ ,  $R_{o_0} = \epsilon | a\epsilon$ , and  $R_\ell = \epsilon | a(b | ab^*b)$ .



This regular expression simplifies. For instance,  $a\epsilon = a$ .

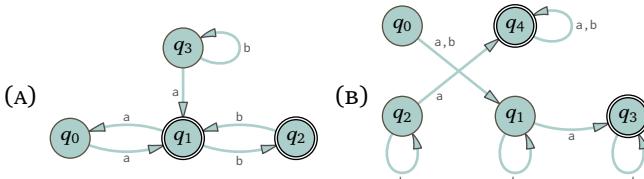
### IV.3 Exercises

- 3.15 Decide if the string  $\sigma$  matches the regular expression  $R$ . (A)  $\sigma = 0010$ ,  $R = 0*10$  (B)  $\sigma = 101$ ,  $R = 1*01$  (C)  $\sigma = 101$ ,  $R = 1*(0|1)$  (D)  $\sigma = 101$ ,  $R = 1*(0|1)*$  (E)  $\sigma = 01$ ,  $R = 1*01*$
- ✓ 3.16 For each regular expression produce five bitstrings that match and five that do not, or as many as there are if there are not five. (A)  $01*$  (B)  $(01)^*$  (C)  $1(0|1)1$  (D)  $(0|1)(\epsilon|1)0*$  (E)  $\emptyset$
- 3.17 Give a brief plain English description of the language for each regular expression. (A)  $a*cb^*$  (B)  $aa^*$  (C)  $a(a|b)*bb$
- ✓ 3.18 For these regular expressions and for each element of  $\{a, b\}^*$  that is of length less than or equal to 3, decide if it is a match. (A)  $a*b$  (B)  $a^*$  (C)  $\emptyset$  (D)  $\epsilon$  (E)  $b(a|b)a$  (F)  $(a|b)(\epsilon|a)a$

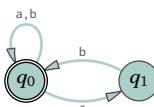
- 3.19 For these regular expressions, decide if each element of  $\mathbb{B}^*$  of length at most 3 is a match. (A)  $0*1$  (B)  $1*0$  (C)  $\emptyset$  (D)  $\varepsilon$  (E)  $0(0|1)^*$  (F)  $(100)(\varepsilon|1)0^*$
- ✓ 3.20 A friend says to you, “The point of parentheses is that you first do inside the parentheses and then do what’s outside. So Kleene star means ‘match the inside and repeat’, and the regular expression  $(0*1)^*$  matches the strings  $001001$  and  $010101$  but not  $01001$  and  $00000101$ , where the substrings are unequal.” Straighten them out.
- 3.21 Produce a regular expression for the language of bitstrings with a substring consisting of at least three consecutive 1’s.
- 3.22 Someone who sits behind you in class says, “I don’t get it. I got a regular expression that I am sure is right. But the book got a different one.” Explain what is up.
- 3.23 For each language, give five strings that are in the language and five that are not. Then give a regular expression describing the language. Finally, give a Finite State machine that accepts the language. (A)  $\mathcal{L}_0 = \{ a^n b^{2m} \mid m, n \geq 1 \}$   
 (B)  $\mathcal{L}_0 = \{ a^n b^{3m} \mid m, n \geq 1 \}$
- 3.24 Give a regular expression for the language over  $\Sigma = \{ a, b, c \}$  whose strings are missing at least one letter, that is, the strings that are either without any a’s, or without any b’s, or without any c’s.
- 3.25 Give a regular expression for each language. Use  $\Sigma = \{ a, b \}^*$ . (A) The set of strings starting with b. (B) The set of strings whose second-to-last character is a. (C) The set of strings containing at least one of each character. (D) The strings where the number of a’s is divisible by three.
- 3.26 Give a regular expression to describe each language over the alphabet  $\Sigma = \{ a, b, c \}$ . (A) The set of strings starting with aba. (B) The set of strings ending with aba. (C) The set of strings containing the substring aba.
- ✓ 3.27 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.
- ✓ 3.28 Give a regular expression to describe each language over the alphabet  $\Sigma = \{ a, b \}$ . (A) Every a is both immediately preceded and immediately followed by a b character. (B) Each string has at least two b’s that are not followed by an a.
- 3.29 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.
- 3.30 Give a regular expression to describe each language.  
 (A)  $\{ \sigma \in \{ a, b \}^* \mid \sigma \text{ ends with the same symbol it began with, and } \sigma \neq \varepsilon \}$   
 (B)  $\{ a^i b a^j \mid i \text{ and } j \text{ leave the same remainder on division by three} \}$

- ✓ 3.31 Give a regular expression for each language over  $\mathbb{B}^*$ .
- The strings representing a binary number that is a multiple of eight.
  - The bitstrings where the first character differs from the final one.
  - The bitstrings where no two adjacent characters are equal.
- ✓ 3.32 Produce a Finite State machine whose language equals the language described by each regular expression. (A)  $a^*ba$  (B)  $ab^*(a|b)^*$
- 3.33 Fix a Finite State machine  $\mathcal{M}$ . Kleene's Theorem shows that the set of strings taking  $\mathcal{M}$  from the start state to the set of final states is regular.
- Show that for any set of states  $S \subseteq Q_{\mathcal{M}}$  the set of strings taking  $\mathcal{M}$  from the start state to one of the states in  $S$  is regular.
  - Show that the set of strings taking  $\mathcal{M}$  from any single state to any other single state is regular.
- 3.34 Part of the proof of Lemma 3.13 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  $\epsilon$  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  $\epsilon$  closure. The union of  $S_i$  with the collection of the states reached in this way is the set  $S_{i+1}$ . Stop when  $S_i = S_{i+1}$ , at which point it is the set of ever-reachable states. The unreachable states are the others.

For each machine use that definition to find the set of unreachable states.

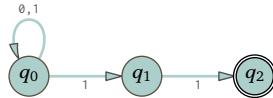


- ✓ 3.35 Show that the set of languages over  $\Sigma$  that are described by a regular expression is countable. Conclude that there are languages not recognized by any Finite State machine.
- 3.36 Construct the parse tree for these regular expressions over  $\Sigma = \{a, b\}$ .
- $a(b|c)$
  - $ab^*(a|c)^*$
- 3.37 Construct the parse tree for Example 3.3's  $a(b|c)^*$  and  $a(b^*|c^*)$ .
- ✓ 3.38 Get a regular expression by applying the method of Lemma 3.13's proof to this machine.



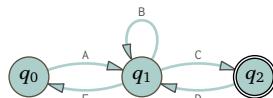
- (A) Get  $\hat{\mathcal{M}}$  by introducing  $e$  and  $f$ . (B) Where  $q = q_0$ , describe which state from the machine is playing the diagram's before picture role of  $q_{i_0}$ , which edge is  $R_{i_0}$ , etc. (C) Eliminate  $q_0$ .

3.39 Apply method of Lemma 3.13'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.

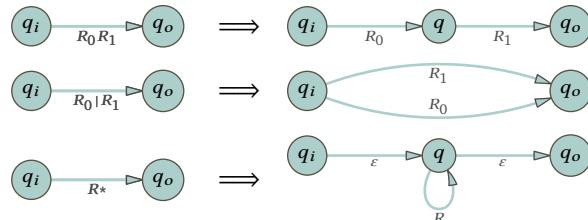
3.40 Apply the state elimination method of Lemma 3.13's proof to eliminate  $q_1$ . Note that each of the states  $q_0$  and  $q_2$  are of the kind described in the proof's comment on the subtle point.



3.41 An alternative proof of Lemma 3.11 reverses the steps of Lemma 3.13. This is the **subset method**. Start by labeling the single edge on a two-state machine with the given regular expression.



Then instead of eliminating nodes, introduce them.



Use this approach to get a machine that recognizes the language described by these regular expressions. (A)  $a|b$  (B)  $ca^*$  (c)  $(a|b)c^*$  (D)  $(a|b)(b^*|a^*)$

## SECTION

### IV.4 Regular languages

We have seen that deterministic Finite State machines, nondeterministic Finite State machines, and regular expressions all describe the same set of languages. The fact that we can describe these languages in so many different ways suggests that there is something natural and important about them.<sup>†</sup>

<sup>†</sup>This is like the fact that the equivalence of Turing machines, general recursive functions, and all kinds of other models suggests that the computable sets form a natural and important collection. Neither collection is only a historical artifact of what happened to be first explored.

**Definition** We will isolate and study these languages.

- 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** The number of regular languages over an alphabet is countably infinite. The collection of languages over that alphabet is uncountable, and consequently there are languages that are not regular.

*Proof* Fix an alphabet  $\Sigma$ . Recall that, as defined in Appendix A, any alphabet is nonempty and finite. Thus there are infinitely many regular languages over that alphabet, because every finite language is regular—just list all the cases as in Example 1.8—and there are infinitely many finite languages.

Next we argue that the number of regular languages is countable. This holds because the number of regular expressions over  $\Sigma$  is countable: clearly there are finitely many regular expressions of length 1, of length 2, etc. The union of those is a countable union of countable sets, and so is countable.

We finish by showing that the set of languages over  $\Sigma$ , the set of all  $\mathcal{L} \subseteq \Sigma^*$ , is uncountable. The set  $\Sigma^*$  is countably infinite by the argument of the prior two paragraphs. The set of all  $\mathcal{L} \subseteq \Sigma^*$  is the power set of  $\Sigma^*$ , and so has cardinality greater than the cardinality of  $\Sigma^*$ , which makes it uncountable.  $\square$

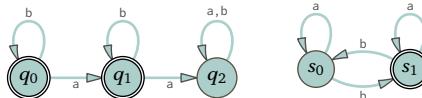
**Closure properties** In proving Lemma 3.11, the first half of Kleene’s Theorem, we showed that if two languages  $\mathcal{L}_0, \mathcal{L}_1$  are regular then their union  $\mathcal{L} \cup \mathcal{L}_1$  is regular, their concatenation  $\mathcal{L}_0 \cap \mathcal{L}_1$  is regular, and the Kleene star  $\mathcal{L}_0^*$  is regular also. Briefly, where  $R_0$  is a regular expression describing the language  $\mathcal{L}_0$  and  $R_1$  describes  $\mathcal{L}_1$  then the regular expression  $R_0 | R_1$  describes  $\mathcal{L}_0 \cup \mathcal{L}_1$ , and  $R_0 R_1$  describes the concatenation  $\mathcal{L}_0 \cap \mathcal{L}_1$ , and  $R_0^*$  describes  $\mathcal{L}_0^*$ .

Recall that a structure is **closed** under an operation if performing that operation on its members always yields another member. The next result restates the above paragraph in this language.

- 4.3 **LEMMA** The collection of regular languages is closed under the union of two sets, the concatenation of two sets, and Kleene star.

We can ask about the closure of regular languages under other operations. 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.

$$\begin{array}{c} \Delta_0 \quad \begin{array}{cc} a & b \end{array} \\ + q_0 \left| \begin{array}{cc} q_1 & q_0 \\ q_2 & q_1 \end{array} \right. \\ + q_1 \left| \begin{array}{cc} q_2 & q_2 \end{array} \right. \\ q_2 \end{array} \qquad \begin{array}{c} \Delta_1 \quad \begin{array}{cc} a & b \end{array} \\ s_0 \left| \begin{array}{cc} s_0 & s_1 \\ s_1 & s_0 \end{array} \right. \\ + s_1 \end{array}$$

Consider a machine  $\mathcal{M}$  whose states are

$$Q_0 \times Q_1 = \{(q_0, s_0), (q_0, s_1), (q_1, s_0), (q_1, s_1), (q_2, s_0), (q_2, s_1)\}$$

and whose transitions are given by  $\Delta((q_i, r_j)) = (\Delta_0(q_i), \Delta_1(r_j))$ , as here.

$$\begin{array}{c} \Delta \quad \begin{array}{cc} a & b \end{array} \\ \begin{array}{c} (q_0, s_0) \\ (q_0, s_1) \\ (q_1, s_0) \\ (q_1, s_1) \\ (q_2, s_0) \\ (q_2, s_1) \end{array} \left| \begin{array}{cc} (q_1, s_0) & (q_0, s_1) \\ (q_1, s_1) & (q_0, s_0) \\ (q_2, s_0) & (q_1, s_1) \\ (q_2, s_1) & (q_1, s_0) \\ (q_2, s_0) & (q_2, s_1) \\ (q_2, s_1) & (q_2, s_0) \end{array} \right. \end{array}$$

This machine runs  $\mathcal{M}_0$  and  $\mathcal{M}_1$  in parallel. For instance, if we feed the string aba to  $\mathcal{M}$ , then the machine's states go from  $(q_0, s_0)$  to  $(q_1, s_0)$ , then to  $(q_1, s_1)$ , and then to  $(q_2, s_1)$ . This is simply because  $\mathcal{M}_0$  passes from  $q_0$  to  $q_1$ , then to  $q_1$ , and then  $q_2$ , while  $\mathcal{M}_1$  does  $s_0$  to  $s_0$ , then to  $s_1$ , and finally to  $s_1$ .

The above table does not specify which states are accepting. Suppose that we say that the accepting states  $(q_i, s_j)$  are the ones where both  $q_i$  and  $s_j$  are accepting. Then by the prior paragraph,  $\mathcal{M}$  accepts a string  $\sigma$  if both  $\mathcal{M}_0$  and  $\mathcal{M}_1$  accept it. That is, this specification of accepting states causes  $\mathcal{M}$  to accept the intersection of the language of  $\mathcal{M}_0$  and the language of  $\mathcal{M}_1$ .

**4.5 THEOREM** The collection of regular languages is closed under the intersection of two sets, the difference of two sets, and set complement.

*Proof* Start with two Finite State machines,  $\mathcal{M}_0$  and  $\mathcal{M}_1$ , which accept languages  $\mathcal{L}_0$  and  $\mathcal{L}_1$  over some  $\Sigma$ . 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  $\mathcal{M}$  accepts the intersection of the languages,  $\mathcal{L}_0 \cap \mathcal{L}_1$ . If the accepting states of  $\mathcal{M}$  are those pairs where the first component state is accepting but the second is not, then  $\mathcal{M}$  accepts the set difference of the languages,  $\mathcal{L}_0 - \mathcal{L}_1$ . A special case of that is when  $\mathcal{L}_0$  is the set of all strings,  $\Sigma^*$ , whereby  $\mathcal{M}$  accepts the complement of  $\mathcal{L}_1$ .  $\square$

These closure properties often make it easier to show that a language is regular.

**4.6 EXAMPLE** To show that the language

$$\mathcal{L} = \{\sigma \in \mathbb{B}^* \mid \sigma \text{ has an even number of } 0\text{'s and more than two } 1\text{'s}\}$$

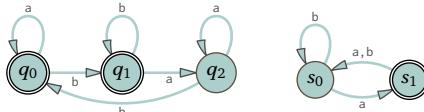
is regular, we could produce a machine that recognizes it, or give a regular expression. Or, we can instead note that  $\mathcal{L}$  is this intersection,

$$\{\sigma \in \mathbb{B}^* \mid \sigma \text{ has an even number of } 0\text{'s}\} \cap \{\sigma \in \mathbb{B}^* \mid \sigma \text{ has more than two } 1\text{'s}\}$$

and producing machines for those two is easy.

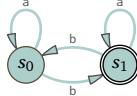
#### IV.4 Exercises

- ✓ 4.7 True or false? Obviously you must justify each answer.
  - (A) Every regular language is finite.
  - (B) Over  $\mathbb{B}$ , the empty language is not regular.
  - (C) The intersection of two languages is regular.
  - (D) Over  $\mathbb{B}$ , the language of all strings,  $\mathbb{B}^*$ , is not regular.
  - (E) Every Finite State machine accepts at least one string.
  - (F) For every Finite State machine there is one that has fewer states but recognizes the same language.
- 4.8 One of these is true and one is false. Which is which?
  - (A) Any finite language is regular.
  - (B) Any regular language is finite.
- 4.9 Is  $\{\sigma \in \mathbb{B}^* \mid \sigma \text{ represents a power of 2 in binary}\}$  a regular language?
- 4.10 Is English a regular language?
- ✓ 4.11 Show that each language over  $\Sigma = \{a, b\}$  is regular.
  - (A)  $\{\sigma \in \Sigma^* \mid \sigma \text{ starts and ends with } a\}$
  - (B)  $\{\sigma \in \Sigma^* \mid \text{the number of } a\text{'s is even}\}$
- ✓ 4.12 True or false? Justify your answer.
  - (A) If  $\mathcal{L}_0$  is a regular languages and  $\mathcal{L}_1 \subseteq \mathcal{L}_0$  then  $\mathcal{L}_1$  is also a regular language.
  - (B) If  $\mathcal{L}_0$  is not regular and  $\mathcal{L}_0 \subseteq \mathcal{L}_1$  then  $\mathcal{L}_1$  is also not regular.
  - (C) If  $\mathcal{L}_0 \cap \mathcal{L}_1$  is regular then each of the two is regular.
- ✓ 4.13 Suppose that the language  $\mathcal{L}$  over  $\mathbb{B}$  is regular. Show that the language  $\hat{\mathcal{L}} = \{1^\sigma \mid \sigma \in \mathcal{L}\}$ , also over  $\mathbb{B}$ , is also regular.
- 4.14 If machines have  $n_0$  states and  $n_1$  states, then how many states does the product have?
- 4.15 For these two machines,



give the transition table for the cross product. Sepcify 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.

- 4.16 Find the machine that is the cross product of the second machine,  $\mathcal{M}_1$ , from Example 4.4, with itself.



with itself. Set the accepting states so that it accepts the same language,  $\mathcal{L}_1$ .

4.17 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 simple machines.

4.18 For each, state True or False and give a justification.

- (A) Every language is the subset of a regular language.
- (B) The union of a regular language and a language that is not regular must be not regular.
- (C) Every language has a subset that is not regular.
- (D) The union of two regular languages is regular, without exception.

4.19 Fill in the blank (with justification): The concatenation of a regular language with a not-regular language \_\_\_\_\_ regular. (A) must be (B) might be, or might be not (C) cannot be

4.20 Where  $\mathcal{L}$  is a language, define  $\mathcal{L}^+$  as the language  $\mathcal{L} \cap \mathcal{L}^*$ . Show that if  $\mathcal{L}$  is regular then so is  $\mathcal{L}^+$ .

4.21 True or false: all finite languages are regular, and there are countably many finite languages, and there are countably many regular sets, so therefore all regular languages are finite.

4.22 Use closure properties to show that if  $\mathcal{L}$  is regular then the set of even-length strings in  $\mathcal{L}$  is also regular.

4.23 Example 4.6 shows that closure properties can make easier some arguments that a language is regular. It can do the same for arguments that a language is not regular. The next section shows that  $\{a^n b^n \in \{a, b\}^* \mid n \in \mathbb{N}\}$  is not regular (this is a restatement of Example 5.2). Use that and closure properties to show that  $\{\sigma \in \{a, b\}^* \mid \sigma \text{ contains the same number of } a\text{'s as } b\text{'s}\}$  is not regular.  
*Hint:* one way is to use closure under intersection.

4.24 Prove that the collection of regular languages over  $\Sigma$  is closed under each of the operations.

- (A)  $\text{pref}(\mathcal{L})$  contains those strings that are a prefix of some string in the language, that is,  $\text{pref}(\mathcal{L}) = \{\sigma \in \Sigma^* \mid \text{there is a } \tau \in \Sigma^* \text{ such that } \sigma \cap \tau \in \mathcal{L}\}$
- (B)  $\text{suff}(\mathcal{L})$  contains the strings that are a suffix of some string in the language, that is,  $\text{suff}(\mathcal{L}) = \{\sigma \in \Sigma^* \mid \text{there is a } \tau \in \Sigma^* \text{ such that } \tau \sigma \in \mathcal{L}\}$
- (C)  $\text{allprefs}(\mathcal{L})$  contains the strings such that all of the prefixes are in the language, that is,  $\text{allprefs}(\mathcal{L}) = \{\sigma \in \mathcal{L} \mid \text{for all } \tau \in \Sigma^* \text{ that is a prefix of } \sigma, \tau \in \mathcal{L}\}$

4.25 Lemma 4.2 gives a counting argument, a pure existence proof, that there are languages that are not regular. But we can also exhibit one. Prove that  $\mathcal{L} = \{1^k \mid k \in K\}$  is not regular, where  $K$  is the Halting problem set,  $K = \{e \in \mathbb{N} \mid \phi_e(e) \downarrow\}$ .

4.26 Lemma 4.2 shows that the collection of regular languages over  $\mathbb{B}$  is countable. Show that not every individual language in it is countable.

- ✓ 4.27 An alternative definition of a regular language is one generated by a **regular grammar**, where rewrite rules have three forms:  $X \rightarrow tY$ , or  $X \rightarrow t$ , or  $X \rightarrow \epsilon$ . That is, the rule head has one nonterminal and rule body has a terminal followed by a nonterminal, or possibly a single nonterminal or the empty string. This is an example, with the language that it generates.

$$S \rightarrow aS \mid bS$$

$$S \rightarrow aA$$

$$A \rightarrow aB$$

$$B \rightarrow \epsilon \mid b$$

$$\mathcal{L} = \{\sigma \in \{a, b\}^* \mid \sigma = \tau^\wedge aa \text{ or } \sigma = \tau^\wedge aab\}$$

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  $X \rightarrow \epsilon$  rule make state  $q_X$  accepting. (C) For each  $X \rightarrow tY$  rule put a transition from  $q_X$  to  $q_Y$  labeled  $t$ . (D) If there are any  $X \rightarrow t$  rules then make an accepting state  $\bar{q}$ , and for each such rule put a transition from  $q_X$  to  $\bar{q}$  labeled  $t$ .

4.28 We can give an alternative proof of Theorem 4.5, that the collection of regular languages is closed under set intersection, set difference, and set complement, that does not rely on a somewhat mysterious “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  $S - T = S \cup T^c$  to make a similar observation about set difference.
- (C) Show that the complement of a regular language is also a regular language.

4.29 Prove that the language recognized by a Finite State machine with  $n$  states is infinite if and only if the machine accepts at least one string of length  $k$ , where  $n \leq k < 2n$ .

4.30 Fix two alphabets  $\Sigma_0, \Sigma_1$ . A function  $h: \Sigma_0 \rightarrow \Sigma_1^*$  induces a **homomorphism** on  $\Sigma_0^*$  via the operation  $h(\sigma^\wedge \tau) = h(\sigma)^\wedge h(\tau)$  and  $h(\epsilon) = \epsilon$ .

- (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  $\hat{\mathcal{L}} = \{\sigma^\wedge 1 \mid \sigma \in \mathbb{B}^*\}$ ; 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  $h(\mathcal{L})$ .

4.31 The proofs here works with deterministic Finite State machines. Find a nondeterministic Finite State machine  $\mathcal{M}$  so that producing another machine  $\hat{\mathcal{M}}$

by taking the complement of the accepting states,  $F_{\mathcal{M}'} = (F_{\mathcal{M}})^c$ , will not result in the language of the second machine being the complement of the language of the first.

4.32 We will show that the class of regular languages is closed under reversal. Recall that the reversal of the language is defined to be the set of reversals of the strings in the language  $\mathcal{L}^R = \{\sigma^R \mid \sigma \in \mathcal{L}\}$ .

- (A) Show that for any two strings the reversal of the concatenation is the concatenation, in the opposite order, of the reversals  $(\sigma_0 \wedge \sigma_1)^R = \sigma_1^R \wedge \sigma_0^R$ .

*Hint:* do induction on the length of  $\sigma_1$ .

- (B) We will prove the result by showing that for any regular expression  $R$ , the reversal  $\mathcal{L}(R)^R$  is described by a regular expression. We will construct this expression by defining a reversal operation on regular expressions. Fix an alphabet  $\Sigma$  and let (i)  $\emptyset^R = \emptyset$ , (ii)  $\epsilon^R = \epsilon$ , (iii)  $x^R = x$  for any  $x \in \Sigma$ , (iv)  $(R_0 \wedge R_1)^R = R_1^R \wedge R_0^R$ , (v)  $(R_0 \mid R_1)^R = R_0^R \mid R_1^R$ , and (vi)  $(R^*)^R = (R^R)^*$ . (Note the relationship between (iv) and the prior exercise item.) Now show that  $R^R$  describes  $\mathcal{L}(R)^R$ . *Hint:* use induction on the length of the regular expression  $R$ .

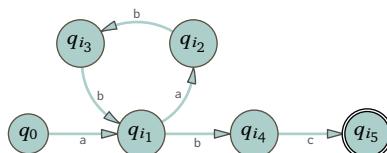
## SECTION

### IV.5 Languages that are not regular

The prior section gave a counting argument to show that there are languages that are not regular. Now we produce a technique to show that specific languages are not regular.

The idea is that, although Finite State machines are finite, they can handle arbitrarily long inputs. This chapter's first example, the power switch from Example 1.1, has only two states but even if we toggle it hundreds of times, it still keeps track of whether the switch is on or off. To handle these long inputs with only a small number of states, a machine must revisit states, that is, it must loop.

Loops inside a machine causes a pattern in what a machine accepts. The diagram below shows a machine that accepts aabbabc (it only shows some of the states, those that the machine traverses in processing this input).



Besides aabbabc, this machine must also accept  $a(abb)^2bc$  because that string takes the machine through the loop twice, and then to the accepting state. Likewise, this machine accepts  $a(abb)^3bc$  and looping more times pumps out more accepted strings.

- 5.1 **THEOREM (PUMPING LEMMA)** Let  $\mathcal{L}$  be a regular language. Then there is a constant  $p \in \mathbb{N}$ , the **pumping length** for the language, such that every string  $\sigma \in \mathcal{L}$  with  $|\sigma| \geq p$  decomposes into three substrings  $\sigma = \alpha \cap \beta \cap \gamma$  satisfying: (1) the first two components are short,  $|\alpha\beta| \leq p$ , (2)  $\beta$  is not empty, and (3) all of the strings  $\alpha\gamma, \alpha\beta^2\gamma, \alpha\beta^3\gamma, \dots$  are also members of the language  $\mathcal{L}$ .

*Proof* Suppose that  $\mathcal{L}$  is recognized by the Finite State machine  $\mathcal{M}$ . Denote the number of states in  $\mathcal{M}$  by  $p$ . 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. 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 the input  $\sigma$ , the machine revisits at least one state.

Of the states that are repeated as the machine processes  $\sigma$ , fix the one  $q$  that it visits first. Also fix  $\sigma$ 's first two substrings  $\langle s_0, \dots, s_i \rangle$  and  $\langle s_0, \dots, s_i, \dots, s_j \rangle$  that take the machine to  $q$ . That is,  $i$  and  $j$  are minimal such that  $i \neq j$  and such that the extended transition function gives  $\hat{\Delta}(\langle s_0, \dots, s_i \rangle) = \hat{\Delta}(\langle s_0, \dots, s_j \rangle) = q$ . Let  $\alpha = \langle s_0, \dots, s_i \rangle$ , let  $\beta = \langle s_{i+1}, \dots, s_j \rangle$ , and let  $\gamma = \langle s_{j+1}, \dots, s_k \rangle$  be the rest of  $\sigma$ . (Think of  $\alpha$  as the initial substring of  $\sigma$  that transitions the machine up to the loop, think of  $\beta$  as the substring that takes the machine around the loop once, and  $\gamma$  is the rest of  $\sigma$ . Possibly  $\alpha$  is empty. Possibly also  $\gamma$  is empty, or perhaps it includes a substring that transitions the machine around the loop a number of times after  $\beta$  has taken it around the first time.) These strings satisfy conditions (1) and (2). Choosing  $q$ ,  $i$ , and  $j$  to be minimal guarantees that, for instance, if the string  $\sigma$  brings machine around a loop a hundred times then we don't fix an  $\alpha$  that includes the first ninety nine loops, and that therefore is longer than  $p$ .

For condition (3), this string

$$\alpha \cap \gamma = \langle s_0, \dots, s_i, s_{j+1}, \dots, s_k \rangle$$

brings the machine from the start state  $q_0$  to  $q$ , and then to the same ending state as did  $\sigma$ . That is,  $\hat{\Delta}(\alpha\gamma) = \hat{\Delta}(\alpha\beta\gamma)$  and so the machine accepts  $\alpha\gamma$ . The other strings in (3) work the same way. For instance, for

$$\alpha \cap \beta^2 \cap \gamma = \langle s_0, \dots, s_i, s_{i+1}, \dots, s_{j-1}, s_{i+1}, \dots, s_{j+1}, \dots, s_k \rangle$$

the substring  $\alpha$  brings the machine from  $q_0$  to  $q$ , the first  $\beta$  brings it around to  $q$  again, the second  $\beta$  makes the machine loop to  $q$  yet again, and finally  $\gamma$  brings it to the same ending state as did  $\sigma$ .  $\square$

Often, how we use the Pumping Lemma is to show that a language is not regular by an argument by contradiction.

- 5.2 **EXAMPLE** The classic example is to show that this language of matched parentheses is not regular. The alphabet is the set of parentheses characters,  $\Sigma = \{(), ()\}$ .

$$\mathcal{L} = \{ ({}^n)^n \in \Sigma^* \mid n \in \mathbb{N} \} = \{ \epsilon, (), ((())), (((())), ({}^4)^4, \dots \}$$

For contradiction, assume that it is regular. Then the Pumping Lemma says that  $\mathcal{L}$  has a pumping length,  $p$ .

Consider the string  $\sigma = ({}^p)^p$ . It is an element of  $\mathcal{L}$  and its length is greater than or equal to  $p$  so the Pumping Lemma applies. Hence  $\sigma$  decomposes into three substrings  $\sigma = \alpha\beta\gamma$  satisfying the conditions. Condition (1) is that the length of the prefix  $\alpha\beta$  is less than or equal to  $p$ . Because of this condition, and because the first  $p$ -many characters are all open parentheses, we know that both  $\alpha$  and  $\beta$  are composed of open parentheses. Condition (2) is that  $\beta$  is not empty so it consists of at least one  $($ .

Condition (3) is that the strings  $\alpha\gamma$ ,  $\alpha\beta^2\gamma$ ,  $\alpha\beta^3\gamma$ , ... are members of  $\mathcal{L}$ . To get the contradiction, consider  $\alpha\beta^2\gamma$  (other choices would also work). Compared with  $\sigma = \alpha\beta\gamma$ , this string has an extra  $\beta$ , 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 it must be a member of  $\mathcal{L}$ , and consequently the assumption that  $\mathcal{L}$  is regular is impossible.

We have seen many examples of things that Finite State machines can do. Here we see something that they cannot. Matching parentheses, and other types of matching, is something that we often want to do, for instance, in a compiler. So the Pumping Lemma helps us show that for some common computing tasks, regular languages are not enough.

- 5.3 EXAMPLE Recall that a palindrome is a string that reads the same backwards as forwards, such as bab, abbaabba, or  $a^5ba^5$ . We will prove that the language  $\mathcal{L} = \{\sigma \in \Sigma^* \mid \sigma^R = \sigma\}$  of all palindromes over  $\Sigma = \{a, b\}$  is not regular.

For contradiction assume that this language is regular. The Pumping Lemma says that  $\mathcal{L}$  has a pumping length. Call it  $p$ . Consider  $\sigma = a^pba^p$ , which is an element of  $\mathcal{L}$  and has more than  $p$  characters. Thus it decomposes as  $\sigma = \alpha\beta\gamma$ , subject to the three conditions. Condition (1) is that  $|\alpha\beta| \leq p$  and so both substrings  $\alpha$  and  $\beta$  are composed entirely of a's. Condition (2) is that  $\beta$  is not the empty string and so  $\beta$  consists of at least one a.

Condition (3) gives that all of the strings  $\alpha\gamma$ ,  $\alpha\beta^2\gamma$ ,  $\alpha\beta^3\gamma$ , ... are also in the language. We will get the desired contradiction from its first element,  $\alpha\gamma$  (the other list members also give contradictions but we only need one).

Compared to  $\sigma = \alpha\beta\gamma$ , in  $\alpha\gamma$  the  $\beta$  is gone. Because  $\alpha$  and  $\beta$  consist entirely of a's, the substring  $\gamma$  has the b character from  $\sigma$  and hence also has the  $a^p$  that follows b. So in passing from  $\sigma = \alpha\beta\gamma$  to  $\alpha\gamma$  we've omitted at least one a before the b but none of the a's after it. Thus  $\alpha\gamma$  is not a palindrome, which contradicts the Pumping Lemma's third condition.

- 5.4 REMARK In that example  $\sigma$  has three parts,  $\sigma = a^p \wedge b \wedge a^p$ , and it decomposes into three parts,  $\sigma = \alpha\beta\gamma$ . Don't make the mistake of thinking that the two decompositions match. The Pumping Lemma does not say that  $\alpha = a^p$ ,  $\beta = b$ , and  $\gamma = a^p$  — indeed, as the example says the Pumping Lemma gives that  $\beta$  does not contain the b. Instead, the Pumping Lemma only says that the first two strings

together,  $\alpha \cap \beta$ , consists exclusively of a's. So it could be that  $\alpha\beta = a^p$ , or it could instead be that the  $\gamma$  starts with some a's that are then followed by  $ba^p$ .

- 5.5 EXAMPLE Consider  $\mathcal{L} = \{0^m 1^n \in \mathbb{B}^* \mid m = n + 1\} = \{0, 001, 00011, \dots\}$ . Its members have a number of 0's that is one more than the number of 1's. We will prove that it is not regular.

For contradiction assume otherwise, that  $\mathcal{L}$  is regular, and set  $p$  as its pumping length. Consider  $\sigma = 0^{p+1} 1^p \in \mathcal{L}$ . Because  $|\sigma| \geq p$ , the Pumping Lemma gives a decomposition  $\sigma = \alpha\beta\gamma$  satisfying the three conditions. Condition (1) says that  $|\alpha\beta| \leq p$ , so that the substrings  $\alpha$  and  $\beta$  have only 0's. Condition (2) says that  $\beta$  has at least one character, necessarily 0. Consider Condition (3)'s list:  $\alpha\gamma, \alpha\beta^2\gamma, \alpha\beta^3\gamma, \dots$ . Compare its first entry,  $\alpha\gamma$ , to  $\sigma$ . The string  $\alpha\gamma$  has fewer 0's than does  $\sigma$  but the same number of 1's. So the number of 0's in  $\alpha\gamma$  is not one more than its number of 1's. Thus  $\alpha\gamma \notin \mathcal{L}$ , which contradicts the Pumping Lemma.

We can interpret that example to say that Finite State machines cannot correctly recognize a predecessor-successor relationship. We can similarly use the Pumping Lemma to show Finite State machines cannot recognize other arithmetic relations.

- 5.6 EXAMPLE The language  $\mathcal{L} = \{a^n \mid n \text{ is a perfect square}\} = \{\varepsilon, a, a^4, a^9, a^{16}, \dots\}$  is not regular. For, suppose otherwise. Fix a pumping length  $p$  and consider  $\sigma = a^{(p^2)}$ , so that  $|\sigma| = p^2$  and thus  $\sigma \in \mathcal{L}$ .

By the Pumping Lemma,  $\sigma$  decomposes into  $\alpha\beta\gamma$ , subject to the three conditions. Condition (1) is that  $|\alpha\beta| \leq p$ , which implies that  $|\beta| \leq p$ . Condition (2) is that  $0 < |\beta|$ . Now consider the strings  $\alpha\gamma, \alpha\beta^2\gamma, \dots$

The definition of the language  $\mathcal{L}$  is that after  $\sigma$  the next longest string has length  $(p + 1)^2 = p^2 + 2p + 1$ , and the difference between  $p^2$  and  $p^2 + 2p + 1$  is strictly greater than  $p$ . But the Pumping Lemma requires that the gap between the length  $|\sigma| = |\alpha\beta\gamma|$  and the length  $|\alpha\beta^2\gamma|$  be at most  $p$ , because  $0 < |\beta| \leq p$ . Thus the length of  $\alpha\beta^2\gamma$  is not a perfect square, which contradicts the Pumping Lemma's assertion that  $\alpha\beta^2\gamma \in \mathcal{L}$ .

Sometimes we can solve problems by using the Pumping Lemma in conjunction with the closure properties of regular languages.

- 5.7 EXAMPLE The language  $\mathcal{L} = \{\sigma \in \{a, b\}^* \mid \sigma \text{ has as many a's as b's}\}$  is not regular. To prove that, observe that the language  $\hat{\mathcal{L}} = \{a^m b^n \in \{a, b\}^* \mid m, n \in \mathbb{N}\}$  is regular, described by the regular expression  $a^* b^*$ . Recall that the intersection of two regular languages is regular. But  $\mathcal{L} \cap \hat{\mathcal{L}}$  is the set  $\{a^n b^n \mid n \in \mathbb{N}\}$  and Example 5.2 shows that this language isn't regular, after we substitute a and b for the parentheses.

In previous sections we saw how to show that a language is regular, either by producing a Finite State machine that recognizes it or by producing a regular expression that describes it. Being able to show that a language is not regular nicely balances that.

But our interest is motivated by more than symmetry. A Turing machine can solve the problem of Example 5.2, of recognizing strings of balanced parentheses, but we now know that a Finite State machine cannot. Therefore we now know that to solve this problem we need scratch memory. So the results in this section speak to the resources needed to solve the problems.

## IV.5 Exercises

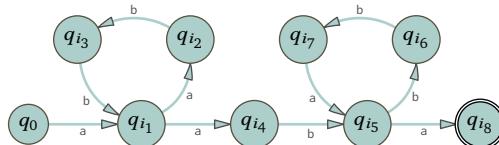
*A useful technique when you are stuck on a language description is to try listing five strings that are in the language and five that are not. Another is to describe the language in prose, as though over a telephone. Both help you think through the formalities.*

- ✓ 5.8 Example 5.5 shows that  $\{0^m 1^n \in \mathbb{B}^* \mid m = n + 1\}$  is not regular but your friend doesn't get it and asks you, "What's wrong with the regular expression  $0^{n+1} 1^n$ ?" Explain it to them.
- 5.9 Example 5.2 uses  $\alpha\beta^2\gamma$  to show that the language of balanced parentheses is not regular. Instead get the contradiction by showing that  $\alpha\gamma$  is not a member of the language.
- 5.10 Your friend has been thinking. They say, "Hey, the diagram just before Theorem 5.1 doesn't apply unless the language is infinite. Sometimes languages are regular because they only have like three or four strings. So the Pumping Lemma is wrong." In what way do they need to further refine their thinking?
- 5.11 Someone in the class emails you, "If a language has string with length greater than the number of states, which is the pumping length, then it cannot be a regular language." Correct?
- ✓ 5.12 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. (A)  $\mathcal{L}_0 = \{a^n b^m \mid n + 2 = m\}$  (B)  $\mathcal{L}_1 = \{a^n b^m c^n \mid n, m \in \mathbb{N}\}$  (C)  $\mathcal{L}_2 = \{a^n b^m \mid n < m\}$
- ✓ 5.13 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  $\gamma$  consists of )'s." Give them a prompt.
- 5.14 In class someone asks, "Isn't it true that languages don't have a unique pumping length? That if a length of  $p = 5$  will do then  $p = 6$  will also do?" Before the prof answers, what do you think?
- 5.15 Show that the language over  $\{a, b\}$  consisting of strings having more a's than b's is not regular.
- ✓ 5.16 For each language over  $\Sigma = \{a, b\}$  produce five strings that are members. Then decide if that language is regular. You must prove each assertion by either producing a regular expression or using the Pumping Lemma.

- (A)  $\{a^n b^m \in \Sigma^* \mid n = 3\}$  (B)  $\{a^n b^m \in \Sigma^* \mid n + 3 = m\}$  (C)  $\{\alpha^\wedge \alpha \mid \alpha \in \Sigma^*\}$   
(D)  $\{a^n b^m \in \Sigma^* \mid n, m \in \mathbb{N}\}$  (E)  $\{a^n b^m \in \Sigma^* \mid m - n > 12\}$

5.17 One of these is regular and one is not. Which is which? You must prove your assertions. (A)  $\{a^n b^m \in \{a, b\}^* \mid n = m^2\}$  (B)  $\{a^n b^m \in \{a, b\}^* \mid 3 < m, n\}$

- ✓ 5.18 Use the Pumping Lemma to prove that  $\mathcal{L} = \{a^{m-1} cb^m \mid m \in \mathbb{N}^+\}$  is not regular. It may help to first produce five strings from the language.
- 5.19 Is  $\{\sigma \in \mathbb{B}^* \mid \sigma = \alpha\beta\alpha^R \text{ for } \alpha, \beta \in \mathbb{B}^*\}$  regular? Either way, prove it.
- 5.20 Prove that the language  $\mathcal{L} = \{\sigma \in \{1\}^* \mid |\sigma| = n! \text{ for some } n \in \mathbb{N}\}$  is not regular. Hint: the differences  $(n+1)! - n!$  grow without bound.
- 5.21 One of these is regular, one is not:  $\{0^m 1 0^n \mid m, n \in \mathbb{N}\}$  and  $\{0^n 1 0^n \mid n \in \mathbb{N}\}$ . Which is which? Of course, you must prove your assertions.
- ✓ 5.22 Show that there is a Finite State machine that recognizes this language of all sums totaling less than four,  $\mathcal{L}_4 = \{a^i b^j c^k \mid i, j, k \in \mathbb{N} \text{ and } i + j = k \text{ and } k < 4\}$ . Use the Pumping Lemma to show that no Finite State machine recognizes the language of all sums,  $\mathcal{L} = \{a^i b^j c^k \mid i, j, k \in \mathbb{N} \text{ and } i + j = k\}$ .
- 5.23 Decide if each is a regular language of bitstrings: (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.
- ✓ 5.24 Show that  $\{0^m 1^n \in \mathbb{B}^* \mid m \neq n\}$  is not regular. Hint: use the closure properties of regular languages.
- 5.25 Example 5.7 shows that  $\{\sigma \in \{a, b\}^* \mid \sigma \text{ has as many } a's \text{ as } b's\}$  is not regular. In contrast, show that  $\mathcal{L} = \{\sigma \in \{a, b\}^* \mid \sigma \text{ has as many } ab's \text{ as } ba's\}$  is regular. Hint: think of ab and ba as marking a transition from a block of one character to a block of another.
- ✓ 5.26 Rebut someone who says to you, “Sure, for the machine before Theorem 5.1, on page 220, a single loop will cause  $\sigma = \alpha^\wedge \beta^\wedge \gamma$ . But if the machine had a double loop like below then you’d need a longer decomposition.”



5.27 Show that  $\{\sigma \in \mathbb{B}^* \mid \sigma = 1^n \text{ where } n \text{ is prime}\}$  is not a regular language. Hint: the third condition's sequence has a constant positive length difference.

5.28 Consider  $\{a^i b^j c^{i-j} \mid i, j \in \mathbb{N}\}$ . (A) Give five strings from this language. (B) Show that it is not regular.

5.29 The language  $\mathcal{L}$  described by the regular expression  $a^* b b b b^*$  is a regular language. We can apply the Pumping Lemma to it. The proof of the Pumping Lemma says that for the pumping length we can use the number of states in

a machine that recognizes the language. Here that gives  $p = 4$ . (A) Consider  $\sigma = \text{abbb}$ . Give a decomposition  $\sigma = \alpha\beta\gamma$  that satisfies the three conditions. (B) Do the same for  $\sigma = \text{b}^{15}$ .

5.30 For a regular language, a pumping length  $p$  is a number with the property that every word of length  $p$  or more can be pumped, that is, can be decomposed so that it satisfies the three properties of Theorem 5.1. The proof of that theorem shows that where a Finite State machine recognizes the language, the number of states in the machine suffices as a pumping length. But  $p$  can be smaller.

(A) Consider the language  $\mathcal{L}$  described by  $(01)^*$ . Construct a deterministic Finite State machine with three states that recognizes this language.

(B) Show that the minimal pumping length for  $\mathcal{L}$  is 1.

5.31 Nondeterministic Finite State machines can always be made to have a single accepting state. For deterministic machines that is not so.

(A) Show that any deterministic Finite State machine that recognizes the finite language  $\mathcal{L}_1 = \{\varepsilon, a\}$  must have at least two accepting states.

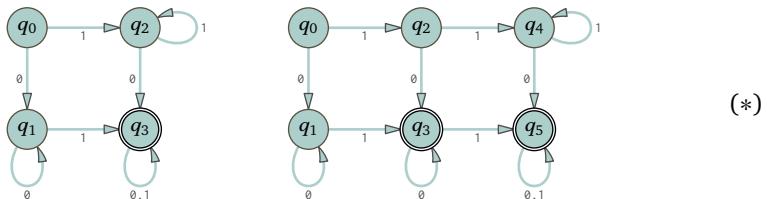
(B) Show that any deterministic Finite State machine that recognizes  $\mathcal{L}_2 = \{\varepsilon, a, aa\}$  must have at least three accepting states.

(C) Show that for any  $n \in \mathbb{N}$  there is a regular language that is not recognized by any deterministic Finite State machine with at most  $n$  accepting states.

## SECTION

### IV.6 Minimization

Contrast these two Finite State machines. For each, the language of accepted strings is  $\{\sigma \in \mathbb{B}^* \mid \sigma \text{ has at least one } 0 \text{ and at least one } 1\}$ .



Informally, our experience is that to make a properly designed machine, each state needs a meaning or job that is well-defined in terms of the machine's language, and that is different from the meaning of all other states. For instance, on the left  $q_2$  means "have seen at least one 1 but still waiting for a 0."

The machine on the right doesn't meet this design principle because the meaning of  $q_4$  is subsumed by that of  $q_2$  since in terms of that machine's language, "have seen two 1's but still waiting for a 0" is a subcase of  $q_2$ 's meaning. Similarly,  $q_5$ 's meaning is subsumed by  $q_3$ 's. Restated in a suggestive way: the pairs of states  $q_2$  and  $q_4$  have the same future, as do  $q_3$  and  $q_5$ . This machine has redundant states.

We will give an algorithm that starts with a Finite State machine and then finds the smallest machine, the machine with the fewest number of states, that recognizes the same language. This algorithm collapses together redundant states.

- 6.1 **DEFINITION** In a Finite State machine over  $\Sigma$ , where  $n \in \mathbb{N}$  we say that two states  $q, \hat{q}$  are  **$n$ -distinguishable** if there is a string  $\sigma \in \Sigma^*$  with  $|\sigma| \leq n$  such that starting the machine in state  $q$  and giving it input  $\sigma$  ends in an accepting state while starting it in  $\hat{q}$  and giving it  $\sigma$  does not, or vice versa. Otherwise the states are  **$n$ -indistinguishable**,  $q \sim_n \hat{q}$ .

Two states  $q, \hat{q}$  are **distinguishable** if there is an  $n$  for which they are  $n$ -distinguishable. Otherwise they are **indistinguishable**,  $q \sim \hat{q}$ .

- 6.2 **EXAMPLE** Starting the machine on the left above in state  $q_0$  and feeding it  $\sigma = 0$  ends in the non-accepting state  $q_1$ . But starting it in  $q_2$  and processing the same input  $0$  ends in the accepting state  $q_3$ . So  $q_0$  and  $q_2$  are 1-distinguishable, and therefore are distinguishable.

Another example is that  $q_2$  and  $q_3$  are 0-distinguishable, via  $\sigma = \epsilon$ . That is, a state that is not accepting is 0-distinguishable from a state that is accepting.

- 6.3 **EXAMPLE** More happens with the machine in the right. This table gives the result of starting in each state and feeding the machine each possible length 0, length 1, and length 2 string. As suggested by the definition, the table doesn't give the resulting state but instead records whether the resulting state is accepting,  $F$ , or nonaccepting,  $Q - F$ .

	$\epsilon$	0	1	00	01	10	11	
$q_0$	$Q - F$	$Q - F$	$Q - F$	$Q - F$	$F$	$F$	$Q - F$	
$q_1$	$Q - F$	$Q - F$	$F$	$Q - F$	$F$	$F$	$F$	
$q_2$	$Q - F$	$F$	$Q - F$	$F$	$F$	$F$	$Q - F$	(**)
$q_3$	$F$	$F$	$F$	$F$	$F$	$F$	$F$	
$q_4$	$Q - F$	$F$	$Q - F$	$F$	$F$	$F$	$Q - F$	
$q_5$	$F$	$F$	$F$	$F$	$F$	$F$	$F$	

The effect of the length 0 input string  $\epsilon$  is that there are two kinds of states: members of  $\{q_0, q_1, q_2, q_4\}$  are the nonaccepting resulting states and members of  $\{q_3, q_5\}$  are the accepting states.

The two columns for the length 1 strings 0 and 1 show all four possible behaviors. The state  $q_0$  is sent to a nonaccepting result for both inputs. The state  $q_1$  goes to a nonaccepting result for 0 but an accepting result for 1. The states  $q_2$  and  $q_4$  result in accepting and nonaccepting for inputs 0 and 1. And  $q_3$  and  $q_5$  go to accepting results for both inputs.

Thus for instance, the definition gives that  $q_0$  is 1-distinguishable from  $q_1$  because the two result columns say  $Q - F, Q - F$  and  $Q - F, F$ . In total we have four classes,  $\{q_0\}$ ,  $\{q_1\}$ ,  $\{q_2, q_4\}$ , and  $\{q_3, q_5\}$ , of sets of states that are 1-distinguishable.

Next consider the length 2 strings. In the columns for these strings, the states  $q_2$  and  $q_4$  both have  $F, F, F$ , and  $Q - F$ . And the states  $q_3$  and  $q_5$  both have  $F, F$ ,  $F$ , and  $F$ . So the length 2 strings do not further divide the states—the relation of 2-distinguishable gives the same four classes of states as does the relations of 1-distinguishable.

- 6.4 LEMMA The relations  $\sim_0, \sim_1, \dots$  and  $\sim$  are equivalence relations.

*Proof* Exercise 6.23. □

Consider again the machine with redundant states that we saw in (\*) above. We denote the  $\sim_0$  equivalence classes with  $\mathcal{E}_{0,0}$  and  $\mathcal{E}_{0,1}$ , we denote the  $\sim_1$  classes with  $\mathcal{E}_{1,0}, \dots, \mathcal{E}_{1,3}$ , etc.

$n$	$\sim_n$ classes			
0	$\mathcal{E}_{0,0} = \{q_0, q_1, q_2, q_4\}$	$\mathcal{E}_{0,1} = \{q_3, q_5\}$		
1	$\mathcal{E}_{1,0} = \{q_0\}$	$\mathcal{E}_{1,1} = \{q_1\}$	$\mathcal{E}_{1,2} = \{q_2, q_4\}$	$\mathcal{E}_{1,3} = \{q_3, q_5\}$
2	$\mathcal{E}_{2,0} = \{q_0\}$	$\mathcal{E}_{2,1} = \{q_1\}$	$\mathcal{E}_{2,2} = \{q_2, q_4\}$	$\mathcal{E}_{2,3} = \{q_3, q_5\}$

When we first considered this machine, we spotted that  $q_2$  and  $q_4$  are redundant. Note that all three lines have them in a class together. The same holds for  $q_3$  and  $q_5$ . We now develop an algorithm<sup>†</sup> that first finds the  $\sim_0$  classes, then the  $\sim_1$  classes, etc. At the end it has the  $\sim$  classes. Those serve as the states of the minimal machine.

For the algorithm, consider what it takes for two states  $q$  and  $\hat{q}$  to be  $n + 1$ -distinguishable but not  $n$ -distinguishable. Let them be distinguished by the length  $n + 1$  string  $\sigma = \langle s_0, s_1, \dots, s_{n-1}, s_n \rangle = \tau \hat{\tau} s_n$ . Because the states are not  $n$ -distinguishable, where the prefix  $\tau$  brings the machine from  $q$  to a state  $r$ , and where  $\tau$  brings the machine from  $\hat{q}$  to some  $\hat{r}$ , the two  $r$  and  $\hat{r}$  must be in the same class  $\mathcal{E}_{n,i}$ . Therefore, distinguishing between these states must involve  $\sigma$ 's final character  $s_n$ . It must take  $r$  to a state in one class  $\mathcal{E}_{n+1,j}$  and take  $\hat{r}$  to a state in another class,  $\mathcal{E}_{n+1,\hat{j}}$ .

That is, when we built the table in (\*\*) we listed all of the length two strings. But we didn't need to do that. To find the  $\sim_{n+1}$  classes we don't need to test whole strings, we need only test single characters to see whether they split the  $\sim_n$  classes, the  $\mathcal{E}_{n,i}$ 's. This is a considerable speedup.

Consider again the machine we have been working with. Its  $\sim_1$  classes are  $\mathcal{E}_{1,0}, \mathcal{E}_{1,1}, \mathcal{E}_{1,2}$ , and  $\mathcal{E}_{1,3}$ . To check for additional splitting in going to the  $\sim_2$  classes, we see if the states in  $\mathcal{E}_{1,2}$  and  $\mathcal{E}_{1,3}$  are sent to states in different  $\sim_1$  classes on being fed single characters.

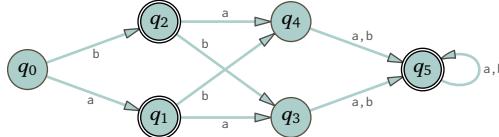
$\mathcal{E}_{1,2}$	0	1	0	1	$\mathcal{E}_{1,3}$	0	1	0	1
$q_2$	$q_3$	$q_4$	$\mathcal{E}_{1,3}$	$\mathcal{E}_{1,2}$	$q_3$	$q_3$	$q_5$	$\mathcal{E}_{1,3}$	$\mathcal{E}_{1,3}$
$q_4$	$q_5$	$q_4$	$\mathcal{E}_{1,3}$	$\mathcal{E}_{1,2}$	$q_5$	$q_5$	$q_5$	$\mathcal{E}_{1,3}$	$\mathcal{E}_{1,3}$

<sup>†</sup>This is **Moore's algorithm**. It is easy and suitable for small calculations but if you are writing code then be aware that another algorithm, **Hopcroft's algorithm**, is more efficient.

(We need only test the classes with more than one member,  $\mathcal{E}_{1,2}$  and  $\mathcal{E}_{1,3}$ , because the singleton classes cannot split.) In both tables there is no split, because the right side of the rows are the same for all the classes members. So we can stop—the  $\sim_1$  classes are the  $\sim$  classes.

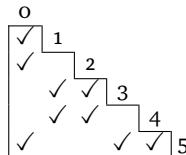
The examples of this algorithm below show how to translate this into a minimal machine, and add a table notation that simplifies the computation.

- 6.5 EXAMPLE We will find a machine that recognizes the same language as this one but that has a minimum number of states.



To do bookkeeping we will use triangular tables that have an entry for every two-element set  $\{i, j\}$  where  $i \neq j$  are indices of states  $q_i$  and  $q_j$ .

Start by checkmarking the  $i, j$  entries where one of  $q_i$  and  $q_j$  is accepting while the other is not.



Checkmarked pairs denote states that are 0-distinguishable. Pairs with a blank are states that are 0-indistinguishable. Here are the  $\sim_0$ -equivalence classes.

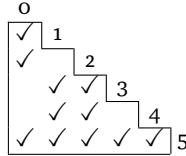
$$\mathcal{E}_{0,0} = \{q_0, q_3, q_4\} \quad \mathcal{E}_{0,1} = \{q_1, q_2, q_5\}$$

Next we investigate those classes to see if they can be split. For each  $\sim_0$  class, look at all pairs of states. The table below lists them on the left. For each single character input, see where that character sends the two states in the pair; that's listed in the middle. On the right the table lists associated  $\sim_0$  classes.

	a	b	a	b
$q_0, q_3$	$q_1, q_5$	$q_2, q_5$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$
$q_0, q_4$	$q_1, q_5$	$q_2, q_5$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$
$q_3, q_4$	$q_5, q_5$	$q_5, q_5$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$
$q_1, q_2$	$q_3, q_4$	$q_4, q_3$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$
$q_1, q_5$	$q_3, q_5$	$q_4, q_5$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$
$q_2, q_5$	$q_4, q_5$	$q_3, q_5$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$

If any entry on the right is a pair consisting of two different classes then the two states are 1-distinguishable. To the triangular table add a checkmark in that entry.

For instance,  $q_1$  and  $q_5$  are together in  $\mathcal{E}_{0,1}$ . From the table under the character a are two different classes,  $\mathcal{E}_{0,0}$  and  $\mathcal{E}_{0,1}$ . In the triangular table, we add a checkmark in entry 1, 5. Similarly,  $q_2$  and  $q_5$  are together in  $\mathcal{E}_{0,1}$  but b takes them to different classes,  $\mathcal{E}_{0,0}$  and  $\mathcal{E}_{0,1}$ . We add a checkmark in entry 2, 5. Here is the table now.

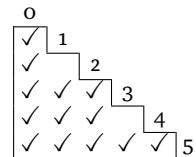


We have found that  $q_5$  can be 1-distinguished from  $q_1$  and  $q_2$ . In short,  $\mathcal{E}_{0,1} = \{q_1, q_2, q_5\}$  splits into two  $\sim_1$  classes.

$$\mathcal{E}_{1,0} = \{q_0, q_3, q_4\} \quad \mathcal{E}_{1,1} = \{q_1, q_2\} \quad \mathcal{E}_{1,2} = \{q_5\}$$

Iterate. We now look to subdivide the  $\sim_1$ -equivalence classes. The table list every pair of indices together in some  $\mathcal{E}_{1,j}$ .

	a	b	a	b	
$\{q_0, q_3\}$	$\{q_1, q_5\}$	$\{q_2, q_5\}$	$\mathcal{E}_{1,1}, \mathcal{E}_{1,2}$	$\mathcal{E}_{1,1}, \mathcal{E}_{1,2}$	
$\{q_0, q_4\}$	$\{q_1, q_5\}$	$\{q_2, q_5\}$	$\mathcal{E}_{1,1}, \mathcal{E}_{1,2}$	$\mathcal{E}_{1,1}, \mathcal{E}_{1,2}$	
$\{q_3, q_4\}$	$\{q_5, q_5\}$	$\{q_5, q_5\}$	$\mathcal{E}_{1,2}, \mathcal{E}_{1,2}$	$\mathcal{E}_{1,2}, \mathcal{E}_{1,2}$	
$\{q_1, q_2\}$	$\{q_3, q_4\}$	$\{q_4, q_3\}$	$\mathcal{E}_{1,0}, \mathcal{E}_{1,0}$	$\mathcal{E}_{1,0}, \mathcal{E}_{1,0}$	
					✓
					✓
					✓
					✓
					✓
					✓

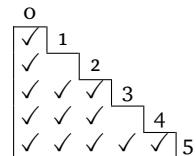


We have found that  $q_0$  is 2-distinguishable from  $q_3$  and  $q_4$  (but they are not distinguishable from each other). The triangular table shown has been updated to contain checkmarks in the 0, 3 and 0, 4 entries. In shory, the  $\sim_1$  class  $\mathcal{E}_{1,0}$  splits into two  $\sim_2$  classes.

$$\mathcal{E}_{2,0} = \{q_0\} \quad \mathcal{E}_{2,1} = \{q_1, q_2\} \quad \mathcal{E}_{2,2} = \{q_3, q_4\} \quad \mathcal{E}_{2,3} = \{q_5\}$$

One more iteration gives this.

	a	b	a	b	
$\{q_1, q_2\}$	$\{q_3, q_4\}$	$\{q_4, q_3\}$	$\mathcal{E}_{2,2}, \mathcal{E}_{2,2}$	$\mathcal{E}_{2,2}, \mathcal{E}_{2,2}$	
$\{q_3, q_4\}$	$\{q_5, q_5\}$	$\{q_5, q_5\}$	$\mathcal{E}_{2,3}, \mathcal{E}_{2,3}$	$\mathcal{E}_{2,3}, \mathcal{E}_{2,3}$	



There is no splitting. The algorithm stops with these  $\sim$  equivalence classes.

$$\mathcal{E}_{2,0} = \{q_0\} \quad \mathcal{E}_{2,1} = \{q_1, q_2\} \quad \mathcal{E}_{2,2} = \{q_3, q_4\} \quad \mathcal{E}_{2,3} = \{q_5\}$$

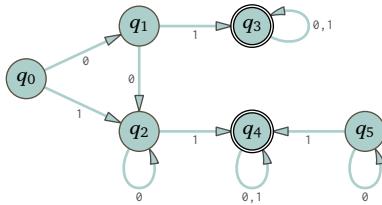
The diagram below shows the minimized machine, with  $r_0$  as a name for  $\mathcal{E}_{2,0}$ , with  $r_1$  for  $\mathcal{E}_{2,1}$ , etc. Its start state  $r_0$  is the one containing  $q_0$ . Its final states are the ones containing final states of the original machine.



As to the connections between states, for instance consider  $r_0 = \{q_0\}$ . In the original machine,  $q_0$  under input  $a$  goes to  $q_1$ . Since  $q_1$  is an element of  $\mathcal{E}_{2,1} = r_1$ , the  $a$  arrow out of  $r_0$  goes to  $r_1$ .

The algorithm has one more step, which did not come up in the prior example. If there are states that are unreachable from  $q_0$  then we start by omitting those.

### 6.6 EXAMPLE Minimize this machine.



To start,  $q_5$  cannot be reached from  $q_0$ . Drop it. That leaves these  $\sim_0$  classes, the non-final states and the final states, and this initial table.

$\mathcal{E}_{0,0} = \{q_0, q_1, q_2\}$	$\mathcal{E}_{0,1} = \{q_3, q_4\}$	<table border="1"> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>✓</td> <td>✓</td> </tr> <tr> <td>✓</td> <td>✓</td> </tr> <tr> <td>✓</td> <td>✓</td> </tr> </table>	0	1	✓	✓	✓	✓	✓	✓
0	1									
✓	✓									
✓	✓									
✓	✓									

Next we see if the  $\sim_0$  classes split.

0	1	0	1	0	1
$q_0, q_1$	$q_1, q_2$	$q_2, q_3$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$	
$q_0, q_2$	$q_1, q_2$	$q_2, q_4$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$	
$q_1, q_2$	$q_2, q_2$	$q_3, q_4$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$	
$q_3, q_4$	$q_3, q_4$	$q_3, q_4$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$	

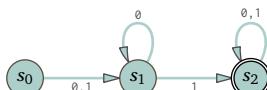
  

0	1	2	3	4
✓	✓	✓	✓	
✓	✓	✓	✓	
✓	✓	✓	✓	
✓	✓	✓	✓	

On being fed a 1 the states  $q_0$  and  $q_1$  go to resulting states,  $q_2$  and  $q_3$ , that are in different  $\sim_0$  classes. The same is true for the second row. The triangular table shown is updated to contains that information. In short,  $\mathcal{E}_{0,0} = \{q_0, q_1, q_2\}$  splits in two.

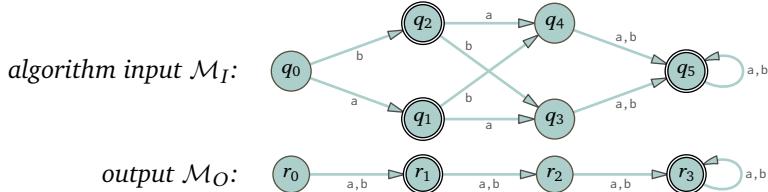
$$\mathcal{E}_{1,0} = \{q_0\} \quad \mathcal{E}_{1,1} = \{q_1, q_2\} \quad \mathcal{E}_{1,2} = \{q_3, q_4\}$$

On the next iteration no more splitting happens (verification is easy).



This minimized machine has three states, one for each  $\sim$  class.

We will close with a proof that this algorithm returns a minimal machine. For that, consider the drawing below. It has the input machine on top, so we can imagine that its states project down onto the output machine's states with  $p(q_0) = r_0$ ,  $p(q_1) = p(q_2) = r_1$ ,  $p(q_3) = p(q_4) = r_2$ , and  $p(q_5) = r_3$ .



The point is that the transition arrows work — the algorithm groups together  $\mathcal{M}_I$ 's states to make  $\mathcal{M}_O$ 's states in a way that respects the starting machine's transitions.

The tables below make the same point. The left table is the transition function of the starting machine,  $\Delta_{\mathcal{M}_I}$ . The right table groups the  $q$ 's into  $r$ 's, so it shows  $\Delta_{\mathcal{M}_O}$ . The states are grouped in a way that allows the transitions in  $\mathcal{M}_O$  to be derived from the transitions in  $\mathcal{M}_I$ . For instance,  $q_1$  and  $q_2$  project to  $r_1$ , and when presented with an input  $a$  they each transition to a state ( $q_3$  and  $q_4$  respectively) that projects to  $r_2$ .

$\Delta_{\mathcal{M}_I}$	a	b	$\Delta_{\mathcal{M}_O}$	a	b
$q_0$	$q_1$	$q_2$	$r_0$	$r_1$	$r_1$
$q_1$	$q_3$	$q_4$	$r_1$	$r_2$	$r_2$
$q_2$	$q_4$	$q_3$	$r_2$	$r_3$	$r_3$
$q_3$	$q_5$	$q_5$	$r_3$	$r_3$	$r_3$
$q_4$	$q_5$	$q_5$	$r_3$	$r_3$	$r_3$
$q_5$	$q_5$	$q_5$			

More precisely, the algorithm is such that  $\Delta_{\mathcal{M}_O}(p(q), x) = p(\Delta_{\mathcal{M}_I}(q, x))$  for all  $q \in \mathcal{M}_I$  and  $x \in \Sigma$ .

- 6.7 **LEMMA** The algorithm above returns a machine that recognizes the same language as the input machine,  $\mathcal{L}(\mathcal{M}_O) = \mathcal{L}(\mathcal{M}_I)$ , and that from among all of the machine recognizing the same language has the minimal number of states.

*Proof* We will argue that the algorithm halts for all input machines, that the returned machine recognizes the same language, and that it has a minimal number of states. The first is easy: the algorithm halts at a step where no class splits and since these machines have only finitely many states, that step must appear.

The returned machine recognizes the same language because the transition function of the output machine respects the transition function of the input. Start both machines on the same string,  $\sigma \in \Sigma^*$ . The input machine  $\mathcal{M}_I$  starts in  $q_0$  while the output machine  $\mathcal{M}_O$  starts in a state  $\mathcal{E}_0$  that contains  $q_0$ . The first character of  $\sigma$  moves  $\mathcal{M}_I$  to a state  $\hat{q}$  and moves  $\mathcal{M}_O$  to a state that contains  $\hat{q}$ . The processing by the two machines proceeds in this way until the string runs out.

Then  $\mathcal{M}_I$  is in a final state if and only if  $\mathcal{M}_O$  is in a state that contains that final state, which is itself a final state of  $\mathcal{M}_O$ . Thus the two machines accept the same set of strings.

For the third, that  $\mathcal{M}_O$  has a minimal number of states, let  $\hat{\mathcal{M}}$  be a machine that recognizes the same language as  $\mathcal{M}_O$ . We will show that it has at least as many states by associating each state in  $\mathcal{M}_O$  with at least one state in  $\hat{\mathcal{M}}$  such that never are different states in  $\mathcal{M}_O$  associated with the same state in  $\hat{\mathcal{M}}$ .

Consider the union of the sets of states of the two machines (assume that they have no states in common). As above, start by saying that two states in the union are 0-indistinguishable if either both are final in their own machine or neither is final. Step  $n + 1$  of this process, also as above, begins with  $\sim_n$  classes  $\mathcal{E}_{n,0}, \dots, \mathcal{E}_{n,k}$  of states from the union that are  $n$ -indistinguishable. For each such class, see if it splits. That is, see if there are two states in that class that are sent by a character  $x \in \Sigma$  to different  $\sim_n$  classes. This gives the  $\sim_{n+1}$  classes. When we reach a step with no splitting then we know which states are indistinguishable; these form the  $\sim$  classes.

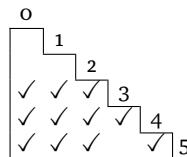
Notice that the start states in the two machines are indistinguishable and are in the same  $\sim$  class, because  $\mathcal{L}(\mathcal{M}_O) = \mathcal{L}(\hat{\mathcal{M}})$ . In addition, if two states are indistinguishable then their successor states on any one input symbol  $x \in \Sigma$  are also indistinguishable from each other, simply because if a string  $\sigma$  distinguishes between the successors then  $x^\sigma$  distinguishes between the original two states. In turn, the successors of these successors are indistinguishable, etc.

Now, say that states in  $\mathcal{M}_O$  and  $\hat{\mathcal{M}}$  are associated if they are indistinguishable, that is, if they are in the same  $\sim$  class. We first show that every state  $q$  of  $\mathcal{M}_O$  is associated with at least one state of  $\hat{\mathcal{M}}$ . Because  $\mathcal{M}_O$  is the output of the minimization process, it has no inaccessible state. So there is a string that takes the start state of  $\mathcal{M}_O$  to  $q$ . This string takes the start state of  $\hat{\mathcal{M}}$  to some  $\hat{q}$ , and the prior paragraph applies to give that  $q \sim \hat{q}$ .

We finish by showing that there cannot be two different states of  $\mathcal{M}_O$  that are both associated with the same state of  $\hat{\mathcal{M}}$ . If there were two such states then by Lemma 6.4 they would be indistinguishable from each other. But that's impossible because  $\mathcal{M}_O$  is the output of the minimization process, which ensures that all of its states are distinguishable.  $\square$

## IV.6 Exercises

6.8 From the triangular table find the  $\sim_i$  classes.



6.9 From the  $\sim_i$  classes find the associated triangular table. (A)  $\mathcal{E}_{i,0} = \{q_0, q_1\}$ ,

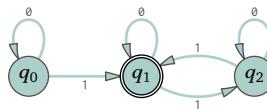
$\mathcal{E}_{i,1} = \{q_2\}$ , and  $\mathcal{E}_{i,2} = \{q_3, q_4\}$ , (B)  $\mathcal{E}_{i,0} = \{q_0\}$ ,  $\mathcal{E}_{i,1} = \{q_1, q_2, q_4\}$ , and  $\mathcal{E}_{i,2} = \{q_3\}$ , (C)  $\mathcal{E}_{i,0} = \{q_0, q_1, q_5\}$ ,  $\mathcal{E}_{i,1} = \{q_2, q_3\}$ , and  $\mathcal{E}_{i,2} = \{q_4\}$ ,

- ✓ 6.10 Suppose that  $\mathcal{E}_{0,0} = \{q_0, q_1, q_2, q_5\}$  and  $\mathcal{E}_{0,1} = \{q_3, q_4\}$ , and from the machine you compute this table.

	a	b	a	b
$q_0, q_1$	$q_1, q_1$	$q_2, q_3$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$
$q_0, q_2$	$q_1, q_2$	$q_2, q_4$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,1}$
$q_0, q_5$	$q_1, q_5$	$q_2, q_5$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$
$q_1, q_2$	$q_1, q_2$	$q_3, q_4$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$
$q_1, q_5$	$q_1, q_5$	$q_3, q_5$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,0}$
$q_2, q_5$	$q_2, q_5$	$q_4, q_5$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,0}$
$q_3, q_4$	$q_3, q_4$	$q_5, q_5$	$\mathcal{E}_{0,1}, \mathcal{E}_{0,1}$	$\mathcal{E}_{0,0}, \mathcal{E}_{0,0}$

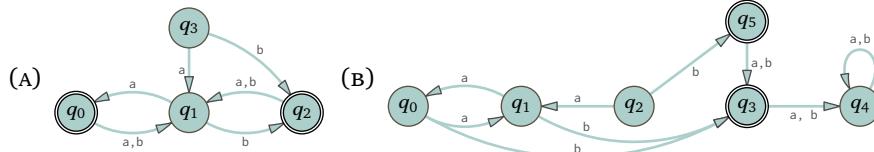
(A) Which lines of the table do you checkmark? (B) Give the resulting  $\sim_1$  classes.

- ✓ 6.11 This machine accepts strings with an odd parity, with an odd number of 1's. Minimize it, using the algorithm described in this section. Show your work.



- ✓ 6.12 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,  $Q - R$ .

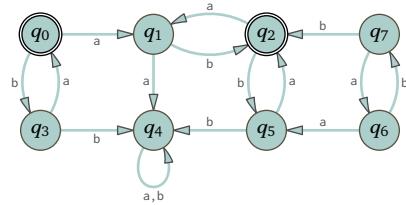
For each machine, perform this algorithm. Show the steps.



- ✓ 6.13 Perform the minimization algorithm on the machine with redundant states at the start of this section, the one on the right in (\*) on page 226.

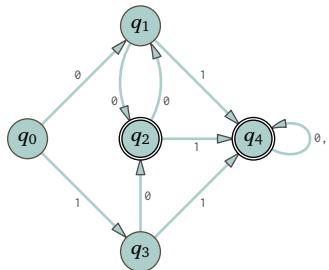
6.14 What happens when you minimize a machine that is already minimal?

- ✓ 6.15 This machine accepts strings described by  $(ab|ba)^*$ . Minimize it, using the algorithm of this section and showing the work.

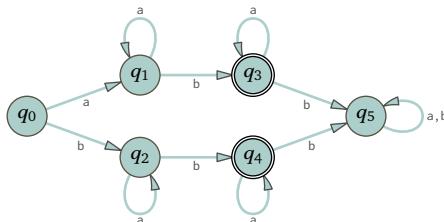


6.16 If a machine's start state is accepting, must the minimized machine's start state be accepting? If you think "yes" then prove it, and if you think "no" then give an example machine where it is false.

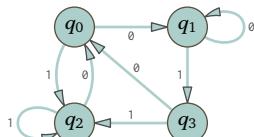
6.17 Minimize this machine.



6.18 Minimize this. Show the work, including producing the diagram of the minimized machine.

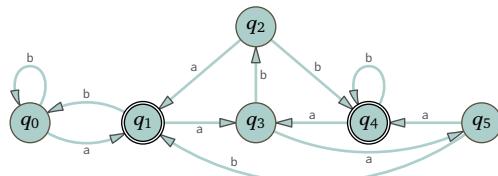


6.19 This machine has no accepting states. Minimize it.



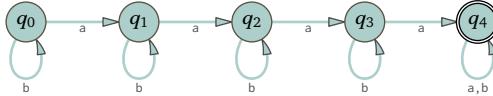
What happens to a machine where all states are accepting?

6.20 Minimize this machine.



6.21 What happens if you perform the minimization procedure in Example 6.6 without first omitting the unreachable state?

✓ 6.22 Minimize.

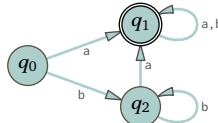


Note that the algorithm takes, roughly, a number of steps that are equal to the number of states in the machine.

6.23 Verify Lemma 6.4.

- (A) Verify that each  $\sim_n$  is an equivalence relation between states.
- (B) Verify that  $\sim$  is an equivalence.

6.24 There are ways to minimize Finite State machines other than the one given in this section. One is Brzozowski's algorithm, which has the advantage of being surprising and fun in that you perform some steps that seem a bit wacky and unrelated to elimination of states and then at the end it has worked. (However, it has the disadvantage of taking worst-case exponential time.) We will walk through the algorithm using this Finite State machine,  $\mathcal{M}$ .



- (A) Use the algorithm in this section to minimize it.
- (B) Instead, get a new machine by taking  $\mathcal{M}$ , changing the state names to be  $t_i$  instead of  $q_i$ , and reversing all the arrows. This gives a nondeterministic machine. Mark what used to be the initial state as an accepting state, and mark what used to be the accepting state as an initial state. (In general, this may result in a machine with more than one initial state.)
- (C) Use the method described in an earlier section to convert this into a deterministic machine, whose states are named  $u_i$ . Omit unreachable states.
- (D) Repeat the second item by changing the state names to  $v_i$  instead of  $u_i$ , and reversing all the arrows. Mark what used to be the initial state as an accepting state and mark what used to be the accepting state as an initial state.
- (E) Convert to a deterministic machine and compare with the one in the first item.

6.25 For each language  $\mathcal{L}$  recognized by some Finite State machine, let  $\text{rank}(\mathcal{L})$  be the smallest number  $n \in \mathbb{N}$  such that  $\mathcal{L}$  is accepted by a Finite State machine with  $n$  states. Prove that for every  $n$  there is a language with that rank.

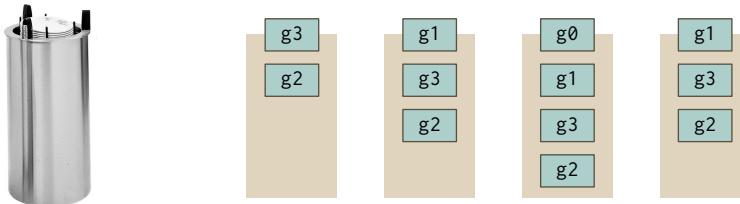
## SECTION

## IV.7 Pushdown machines

No Finite State machine can recognize the language of balanced parentheses. So this machine model is not powerful enough to use, for instance, to decide whether input strings are valid programs in a modern programming language. To handle nested parentheses, the natural data structure is a pushdown stack. We will now supplement a Finite State machine by giving it access to a stack.

Like a Turing machine tape, a stack is unbounded storage. But it has restrictions that the tape does not. It is like the restaurant dish dispenser below. When you push a new dish on, its weight compresses the spring so all the old dishes move down and the latest dish is the only one that you can reach. When you pop a dish off, a spring pushes the remaining dishes up so 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 data structure. First the stack has two characters,  $g_3$  and  $g_2$ . We push  $g_1$  on the stack, and then  $g_0$ . Now, although  $g_1$  is on the stack, we don't have immediate access to it. To get at  $g_1$  we must first pop off  $g_0$ , as in the last stack shown.



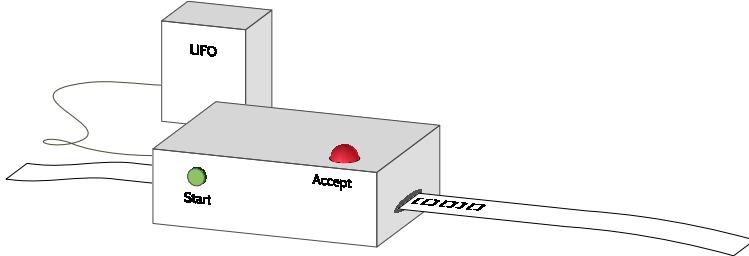
Once something is popped, it is gone. We could include in the machine a state whose intuitive meaning is that we have just popped  $g_0$  but because there are finitely many states that strategy has limits.

**Definition** To define these machines we will extend the definition of Finite State machines.

- 7.1 **DEFINITION** A **Pushdown machine**  $\langle Q, q_0, F, \Sigma, \Gamma, \Delta \rangle$  consists of a finite set of states  $Q = \{q_0, \dots, q_{n-1}\}$ , including a **start state**  $q_0$ , a subset  $F \subseteq Q$  of **accepting states**, a nonempty **input alphabet**  $\Sigma$ , a nonempty **stack alphabet**  $\Gamma$ , and a **transition function**  $\Delta: Q \times (\Sigma \cup \{\text{B}, \varepsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$ .

We assume that the stack alphabet  $\Gamma$  contains the character that we use to mark the stack bottom,  $\perp$ .<sup>†</sup> The rest of  $\Gamma$  is  $g_0, g_1, \dots$ . We also assume that the tape alphabet  $\Sigma$  does not contain the blank or the character  $\varepsilon$ .<sup>‡</sup>

<sup>†</sup> Read that character aloud as “bottom.” <sup>‡</sup>The definition allows  $\varepsilon$  to appear in two separate places, as the second component of  $\Delta$ 's inputs and also as the empty string, from  $\Gamma^*$ . However, one of those is in the inputs and the other is in the outputs so it isn't ambiguous.



The transition function describes how these machines act. For the input  $\langle q_i, s, g_j \rangle \in Q \times (\Sigma \cup \{B, \varepsilon\}) \times \Gamma$  there are two cases. When the character  $s$  is an element of  $\Sigma \cup \{B\}$  then an instruction  $\Delta(q_i, s, g_j) = \langle q_k, \gamma \rangle$  applies when the machine is in state  $q_i$  with the tape head reading  $s$  and with the character  $g_j$  on top of the stack. If there is no such instruction then the computation halts, with the input string not accepted. If there is such an instruction then the machine does this: (i) the read head moves one cell to the right, (ii) the machine pops  $g_j$  off the stack and pushes the characters of the string  $\gamma = \langle g_{i_0}, \dots, g_{i_m} \rangle$  onto the stack in the order from  $g_{i_m}$  first to  $g_{i_0}$  last, and (iii) the machine enters state  $q_k$ . The other case for the input  $\langle q_i, s, g_j \rangle$  is when the character  $s$  is  $\varepsilon$ . Everything is the same except that the tape head does not move. (We use this case to manipulate the stack without consuming any input.)

As with Finite State machines, Pushdown machines don't write to the tape but only consume the tape characters. However, unlike Finite State machines they can fail to halt, see Exercise 7.6.

The starting configuration has the machine in state  $q_0$ , reading the first character of  $\sigma \in \Sigma^*$ , and with the stack containing only  $\perp$ . A machine accepts its input  $\sigma$  if, after starting in its starting configuration and after scanning all of  $\sigma$ , it eventually enters an accepting state  $q \in F$ .

Notice that at each step the machine pops a character off the stack. If we want to leave the stack unchanged then as part of the instruction we must push that character back on. In addition, if the machine reaches a configuration where the stack is empty then it will lock up and be unable to perform any more instructions.<sup>†</sup>

## 7.2 EXAMPLE

This grammar generates the language of balanced parentheses.

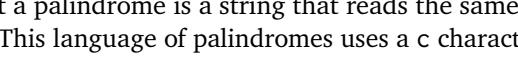
$$S \rightarrow [ S ] \mid SS \mid \varepsilon \quad \mathcal{L}_{BAL} = \{ \varepsilon, [], [[], ], [][], [[][]], [[[[]]]], \dots \}$$

The Pumping Lemma shows that no Finite State machine recognizes this language. But it is recognized by a Pushdown machine. This machine has states  $Q = \{q_0, q_1, q_2\}$  with accepting states  $F = \{q_1\}$ , and alphabets  $\Sigma = \{[, ]\}$  and  $\Gamma = \{g_0, \perp\}$ . The table gives its transition function  $\Delta$ , with the instructions numbered for ease of reference.

<sup>†</sup>An alternative to the final state definition of acceptance we are using is to define that a machine accepts its input if after consuming that input, it empties the stack. The definitions are equivalent in that a string is accepted by either type of machine if it is accepted by the other.

<i>Instr no</i>	<i>Input</i>	<i>Output</i>
0	$q_0, [, \perp$	$q_0, 'g\theta\perp'$
1	$q_0, [, g\theta$	$q_0, 'g\theta g\theta'$
2	$q_0, ], g\theta$	$q_0, \epsilon$
3	$q_0, ], \perp$	$q_2, \epsilon$
4	$q_0, B, \perp$	$q_1, \epsilon$

It keeps a running tally of the number of [’s minus the number of ]’s, as the number of  $g\theta$ ’s on the stack. This computation starts with the input [[[]][] and ends in an accepting state.

<i>Step</i>	<i>Configuration</i>
0	
1	
2	
3	
4	
5	
6	
7	

- 7.3 EXAMPLE Recall that a palindrome is a string that reads the same forwards and backwards,  $\sigma = \sigma^R$ . This language of palindromes uses a c character as a middle marker.

$$\mathcal{L}_{MM} = \{ \sigma \in \{a, b, c\}^* \mid \sigma = \tau^c c \tau^R \text{ for some } \tau \in \{a, b\}^* \}$$

When the Pushdown machine below is reading  $\tau$  it pushes characters onto the stack;  $g0$  when it reads a and  $g1$  when it reads b. That’s state  $q_0$ . When the machine hits the middle c, it reverses. It enters  $q_1$  and starts popping; when reading a it checks that the popped character is  $g0$ , and when reading b it checks that what popped is  $g1$ . If the machine hits the stack bottom at the same moment that the input runs out then it goes into the accepting state  $q_3$ .

<i>Instr no</i>	<i>Input</i>	<i>Output</i>	<i>Instr no</i>	<i>Input</i>	<i>Output</i>
0	$q_0, a, \perp$	$q_0, 'g0\perp'$	9	$q_1, a, g0$	$q_1, \epsilon$
1	$q_0, b, \perp$	$q_0, 'g1\perp'$	10	$q_1, a, g1$	$q_2, \epsilon$
2	$q_0, \epsilon, \perp$	$q_3, \epsilon$	11	$q_1, a, \perp$	$q_2, \epsilon$
3	$q_0, a, g0$	$q_0, 'g0g0'$	12	$q_1, b, g0$	$q_2, \epsilon$
4	$q_0, a, g1$	$q_0, 'g0g1'$	13	$q_1, b, g1$	$q_1, \epsilon$
5	$q_0, b, g0$	$q_0, 'g1g0'$	14	$q_1, b, \perp$	$q_2, \epsilon$
6	$q_0, b, g1$	$q_0, 'g1g1'$	15	$q_1, B, g0$	$q_2, \epsilon$
7	$q_0, c, g0$	$q_1, 'g0'$	16	$q_1, B, g1$	$q_2, \epsilon$
8	$q_0, c, g1$	$q_1, 'g1'$	17	$q_1, B, \perp$	$q_3, \epsilon$

This computation has the machine accept bacab.

<i>Step</i>	<i>Configuration</i>
0	b a c a b q0 $\perp$
1	b a c a b q0 g1 $\perp$
2	b a c a b q0 g0 g1 $\perp$
3	b a c a b q1 g0 g1 $\perp$
4	b a c a b q1 g1 $\perp$
5	b a c a b q1 $\perp$
6	b a c a b q3 $\perp$

- 7.4 REMARK Stack machines are often used in practice, particularly 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.

This language is quite efficient. But it is more suited to situations where the code is written by a program, such as with a word processor or `LATEX`, than to

situations where a person is writing it.

**Nondeterministic Pushdown machines** To get nondeterminism we alter the definition in two ways. The first is minor: we don't need the input character blank,  $B$ , because a nondeterministic machine can guess when the input string ends.

The second alteration changes the transition function  $\Delta$ . We now allow the same input to give different outputs,  $\Delta: Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma^*)$ . (If the set of outputs is empty then we take the machine to freeze, resulting in a computation that does not accept the input.) As always with nondeterminism, we can conceptualize this either as that the computation evolves as a tree or as that the machine guesses one of the branches in that tree.

7.5 EXAMPLE This grammar generates the language of all palindromes over  $\mathbb{B}^*$ .

$$P \rightarrow \epsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1 \quad \mathcal{L}_{PAL} = \{\epsilon, 0, 1, 00, 11, 000, 010, 101, 111, \dots\}$$

This language is not recognized by any Finite State machine, but it is recognized by a Pushdown machine.

This machine has  $Q = \{q_0, q_1, q_2\}$  with accepting states  $F = \{q_2\}$ , and languages  $\Sigma = \mathbb{B}$  and  $\Gamma = \{g0, g1, \perp\}$ .

During its first phase it puts  $g0$  on the stack when it reads the input  $0$  and puts  $g1$  on the stack when it reads  $1$ . During the second phase, if it reads  $0$  then it only proceeds if the popped stack character is  $g0$  and if it reads  $1$  then it only proceeds if it popped  $g1$ .

Instr no	Input	Output	Instr no	Input	Output
0	$q_0, 0, \perp$	$q_0, 'g0\perp'$	9	$q_0, 1, g0$	$q_1, 'g1g0'$
1	$q_0, 1, \perp$	$q_0, 'g1\perp'$	10	$q_0, 1, g1$	$q_1, 'g1g1'$
2	$q_0, \epsilon, \perp$	$q_2, \epsilon$	11	$q_0, 0, g0$	$q_1, 'g0'$
3	$q_0, 0, g0$	$q_0, 'g0g0'$	12	$q_0, 0, g1$	$q_1, 'g1'$
4	$q_0, 1, g0$	$q_0, 'g1g0'$	13	$q_0, 1, g0$	$q_1, 'g0'$
5	$q_0, 0, g1$	$q_0, 'g0g1'$	14	$q_0, 1, g1$	$q_1, 'g1'$
6	$q_0, 1, g1$	$q_0, 'g1g1'$	15	$q_1, 0, g0$	$q_1, \epsilon$
7	$q_0, 0, g0$	$q_1, 'g0g0'$	16	$q_1, 1, g1$	$q_1, \epsilon$
8	$q_0, 0, g1$	$q_1, 'g0g1'$	17	$q_1, 0, g0$	$q_2, \epsilon$
			18	$q_1, 1, g1$	$q_2, \epsilon$

How does the machine know when to change from phase one to two? It is nondeterministic—it guesses. For instance, compare instructions 3 and 7, which show the same input associated with two different outputs.

Here the machine accepts the string  $0110$ . In the calculation it uses instructions 0, 9, 16, and 17.

Step	Configuration
0	
1	
2	
3	
4	

Here is the machine accepting  $01010$  using instructions 0, 4, 12, 16, and 17.

Step	Configuration
0	
1	
2	
3	
4	
5	

The nondeterminism is crucial. In the first example, after step 1 the machine is in state  $q_0$ , is reading a 1, and the character that will be popped off the stack is  $g_0$ . Both instructions 3 and 9 apply to that configuration. But, applying instruction 3 would not lead to the machine accepting the input string. The computation shown instead applies instruction 9, going to state  $q_1$ , whose intuitive meaning is that the machine switches from pushing to popping.

We have given two mental models of nondeterminism. One is that the machine guesses when to switch, and that for this even-length string making that switch halfway through is the right guess. We say the string is accepted because there exists a guess that is correct, that ends in acceptance. (That there exist incorrect guesses is not relevant.)

Taking the other view of nondeterminism omits guessing and instead sees the computation as a tree. In one branch the machine applies instruction 3 and in another it applies instruction 9. By definition, for this machine the string is accepted because there is at least one accepting branch (the above table of the sequence of configurations shows the tree's accepting branch).

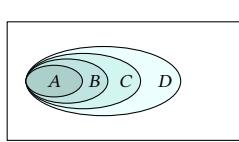
Input strings with odd length are different. In the language of guessing, the machine needs to guess that it must switch from pushing to popping at the middle character, but it must not push anything onto the stack since that thing would never get popped off. Instead, when instruction 12 pops the top character  $g_1$  off the stack, as all instructions do when they are executed, it immediately pushes it back on. The net effect is that in this turn around from pushing to popping the stack is unchanged.

Recall that deterministic Finite State machines can do any jobs that nondeterministic ones can do. The palindrome result shows that for Pushdown machines the situation is different. While nondeterministic Pushdown machines can recognize the language of palindromes, that job cannot be done by deterministic Pushdown machines. So for Pushdown machines, nondeterminism changes what can be done.

Intuitively, Pushdown machines are between Turing machines and Finite State machines in that they have a kind of unbounded read/write memory, but it is limited. We've shown that they are more powerful than Finite State machines because they can recognize the language of balanced parentheses.

There is a relevant result that we will mention but not prove: there are jobs that Turing machines can do but that no Pushdown machine can do. One is the decision problem for the language  $\{\sigma \hat{\wedge} \sigma \mid \sigma \in \mathbb{B}^*\}$ . The intuition is that this language contains strings such as 1010, 10101010, etc. A Pushdown machine can push the characters onto the stack, as it does for the language of balanced parentheses, but then to check that the second half matches the first it would need to pop them off in reverse order.<sup>†</sup>

The diagram below summarizes. The box encloses all languages of bitstrings, all subsets of  $\mathbb{B}^*$ . The nested sets enclose those languages recognized by some Finite State machine, or some Pushdown machine, etc.



<i>Class</i>	<i>Machine type</i>
<i>A</i>	Finite State, including nondeterministic
<i>B</i>	Pushdown
<i>C</i>	nondeterministic Pushdown
<i>D</i>	Turing

---

<sup>†</sup>Another way to tell that the set of languages recognized by a nondeterministic Pushdown machine is a strict subset of the set of languages recognized by a Turing machine is to note that there is no Halting Problem for Pushdown machines. We can write a program that inputs a string and a Pushdown machine, and decides whether it is accepted. But of course we cannot write such a program for Turing machines. Since the languages differ and since anything computed by a Pushdown machine can be computed by a Turing machine, the languages of Pushdown machines must be a strict subset.

**Context free languages** In the section on Grammars we restricted our attention to production rules where the head consists of a single nonterminal, such as  $S \rightarrow cS$ . 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.

If a language has a grammar in which all the rules are of the first type, of the type we described in Chapter III's Section 2, then it is a **context free** language. Most modern programming languages are context free, including C, Java, Python, and Scheme. So grammars that are context sensitive, without the restriction of being required to be context free, are much less common in practice.

We will state, but not prove, the connection with this section: a language is recognized by some nondeterministic Pushdown machine if and only if it has a context free grammar.

#### IV.7 Exercises

- ✓ 7.6 Produce a Pushdown Automata that does not halt.
- ✓ 7.7 Produce a Pushdown machine to recognize each language over  $\Sigma = \{a, b, c\}$ .
  - (A)  $\{a^n cb^{2n} \mid n \in \mathbb{N}\}$
  - (B)  $\{a^n cb^{n-1} \mid n > 0\}$
- 7.8 Give a Pushdown machine that recognizes  $\{\theta^\sim \tau^\sim 1 \mid \tau \in \mathbb{B}^*\}$ .
- 7.9 Consider the Pushdown Automata in Example 7.2.
  - (A) It has an asymmetry in the definition. In line 3 it specifies that if there are too many ]'s then the machine should go to the error state  $q_2$ . But there is no line specifying what to do if there are too many ['. Why is it not needed?
  - (B) Prove that this machine recognizes the language of balanced parentheses defined by the grammar.
- 7.10 Give a Pushdown machine that recognizes  $\{a^{2n} b^n \mid n \in \mathbb{N}\}$ .
- ✓ 7.11 Example 7.5 discusses the view of a nondeterministic computation as a tree. Draw the tree for that machine these inputs. (A) 0110 (B) 01010
- ✓ 7.12 The grammar  $Q \rightarrow oQo \mid 1Q1 \mid \epsilon$  generates a different language of palindromes than the grammar in Example 7.5. What is this language?
- 7.13 Write a context-free grammar for  $\{a^n bc^n \in \{a, b, c\}^* \mid n \in \mathbb{N}\}$ , the language where the number of a's before the b is the same as the number of c's after it.
- 7.14 Find a grammar that generates the language  $\{\sigma^\sim b^\sim \sigma^R \mid \sigma \in \{a, b\}^*\}$ .
- 7.15 The grammar  $Q \rightarrow oQo \mid 1Q1 \mid \epsilon$  generates a different language of palindromes than the grammar in Example 7.5. What is this language?
- 7.16 Find a grammar for the language over  $\sigma = \{a, b, c\}$  consisting of palindromes that contain at most three c's. *Hint:* Use two nonterminals, with one for the case of not adjoining c's.

- 7.17 Show that the language of all palindromes from Example 7.5 is not recognized by any Finite State machine. *Hint:* you can use the Pumping Lemma.
- 7.18 Show that a string  $\sigma \in \mathbb{B}^*$  is a palindrome  $\sigma = \sigma^R$  if and only if it is generated by the grammar given in Example 7.5. *Hint:* Use induction in both directions.
- 7.19 Show that the set of pushdown automata is countable.
- 7.20 Show that any language recognized by a Pushdown machine is recognized by some Turing machine.
- 7.21 There is a Pumping Lemma for Context Free languages: if  $\mathcal{L}$  is Context Free then it has a pumping length  $p \geq 1$  such that any  $\sigma \in \mathcal{L}$  with  $|\sigma| \geq p$  decomposes into five parts  $\sigma = \alpha^\wedge \beta^\wedge \gamma^\wedge \delta^\wedge \zeta$  subject to the conditions (i)  $|\alpha\beta\gamma| \leq p$ , (ii)  $|\beta\delta| \geq 1$ , and (iii)  $\alpha\beta^n\gamma\delta^n\zeta \in \mathcal{L}$  for all  $n \in \mathbb{N}$ .
- (A) Use it to show that  $\{ a^n b^n c^n \mid n \in \mathbb{N} \}$  is not Context Free.
  - (B) Show that  $\{ \sigma^2 \mid \sigma \in \mathbb{B}^* \}$  is not Context Free.
- 7.22 For both Turing machines and Finite State machines, after we gave an informal description of how they act we supplemented that with a formal one. Supply that for Pushdown machines.
- (A) Define a configuration.
  - (B) Define the meaning of the yields symbol  $\vdash$  and a transition step.
  - (C) Define when a machine accepts a string.

## EXTRA

## IV.A Regular expressions in the wild

In practice, working with computers often calls for text operations, that is, for symbol pushing. Regular expressions are an important tool. Modern programming languages such as Racket and Python include capabilities for extended regular expressions, which we will call **regexes**, that go beyond the small-scale theory examples we saw earlier.

As an example, imagine a system administrator searching a web server log for the PDF's downloaded from a directory. They might type this.

```
grep "/linearalgebra/.*\.\pdf" /var/log/apache2/access.log
```

The grep utility program looks through the log file line by line. If a line matches the pattern then grep prints that line. That pattern is a regex.

Different languages have small variations on the syntax of their regexes. Below, we will use Racket regexes. As a prototype to experiment with those regexes, `(regexp-match? #px"^[A-Z][A-Z][0-9][A-Z][A-Z]\$" "KE1AZ")` returns `#t`. (The command `(regexp-match? #px"[0-9]" "KE1AZ")` also returns `#t`, although its expression doesn't account for the letters. Programmers often want to find the expression anywhere in the string, but for us the caret and dollar sign fit better.)

The in-practice extensions to regular expressions 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 not possible with the theoretical regular expressions that we studied earlier.

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,<sup>†</sup> we can shorten it further to  $[0-9]$ . Along the same lines,  $[A-Za-z]$  matches a singleton English letter.

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,  $[^\wedge0-9]$  matches a non-digit and  $[^\wedge A-Za-z]$  matches a character that is not an ASCII letter.

The most common lists have short abbreviations. Another abbreviation for the digits is  $\backslash d$ . Use  $\backslash D$  for the ASCII non-digits,  $\backslash s$  for the whitespace characters (space, tab, newline, formfeed, and line return) and  $\backslash S$  for ASCII characters that are non-whitespace. Cover the alphanumeric characters (upper and lower case ASCII letters, digits, and underscore) with  $\backslash w$  and cover the ASCII non-alphanumeric characters with  $\backslash W$ . And —the big kahuna—the dot ‘.’ is a metacharacter that matches any member of the alphabet at all.<sup>‡</sup>

- A.1 EXAMPLE Canadian postal codes have seven characters: the fourth is a space, the first, third, and sixth are letters, and the others are digits. The regular expression  $[a-zA-Z]\backslash d[a-zA-Z]\backslash d[a-zA-Z]\backslash d$  describes them.
- A.2 EXAMPLE Dates are often given in the ‘dd/mm/yy’ format. This matches:  $\backslash d\backslash d/\backslash d\backslash d/\backslash d\backslash d$ .
- A.3 EXAMPLE In the twelve hour time format some typical times strings are ‘8:05 am’ or ‘10:15 pm’. You could use this (note the empty string at the start).



Courtesy xkcd.com

<sup>†</sup>The digits 0 through 9 are contiguous in both ASCII and Unicode. <sup>‡</sup>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.

$(|0|1)\d:\d\d\s(am|pm)$

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.

Operator precedence is: repetition binds most strongly, then concatenation, and then alternation (force different meanings with parentheses). Thus, ab\* is equivalent to a(b\*), and ab|cd is equivalent to (ab)|(cd).

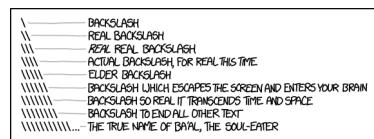
**Quantifiers** In the theoretical cases we saw earlier, to match ‘at most one a’ we used  $\epsilon|a$ . In practice we can write something like  $(|a)$ , as we did above for the twelve hour times. But depicting the empty string by just putting nothing there can be confusing. Modern languages make question mark a metacharacter and allow you to write a? for ‘at most one a’.

For ‘at least one a’ modern languages use a+, so the plus sign is another metacharacter. More generally, we often want to specify quantities. For instance, to match five a’s extended regular expressions 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,}.

As earlier, to match any of these metacharacters you must escape them. For instance, To be or not to be\? matches the famous question.

**Cookbook** All of the extensions to regular expressions that we are seeing are driven by the desires of working programmers. Here is a pile of examples showing them accomplishing practical work, matching things you’d want to match.

- A.4 EXAMPLE US postal codes, called ZIP codes, are five digits. Match them with \d{5}.
- A.5 EXAMPLE North American phone numbers match \d{3} \d{3}-\d{4}.
- A.6 EXAMPLE The regex  $(-|\+)?\d+$  matches an integer, positive or negative. The question mark makes the sign optional. The plus sign makes sure there is at least one digit.
- A.7 EXAMPLE A natural number represented in hexadecimal can contain the usual digits, along with the additional characters ‘a’ through ‘f’ (sometimes capitalized). Programmers often prefix such a representation with 0x, so the expression is  $(0x)?[a-fA-F0-9]^+$ .
- A.8 EXAMPLE A C language identifier begins with an ASCII letter or underscore and then can have arbitrarily many more letters, digits, or underscores: [a-zA-Z\_]\w\*.



Courtesy xkcd.com

- A.9 EXAMPLE Match a user name of between three and twelve letters, digits, underscores, or periods with `[\w\.]{3,12}`. Match a password that is at least eight characters long with `.{8,}`.
- A.10 EXAMPLE Match a valid username on Reddit: `[\w-]{3,20}`. The hyphen, because it comes last in the square brackets, matches itself (as distinguished from the hyphen in `[a-z]`). And no, Reddit does not allow a period in a username.
- A.11 EXAMPLE The International Standards Organization date format calls for dates like ‘yyyy-mm-dd HH:MM:SS’ (along with many other variants). The regex `\d{4}-\d{2}-\d{2} (\d{2}:\d{2}:\d{2})?` will match them.
- A.12 EXAMPLE Match the text inside a single set of parentheses with `\(^()*)\``.
- A.13 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!** You can also match the start of a line and end of line with the metacharacters caret ‘`^`’ and dollar sign ‘`$`’.

- A.14 EXAMPLE Match lines starting with ‘Theorem’ using `^Theorem`. Match lines ending with `end{equation*}` using `end{equation\*}\$`.

Regex engines in modern languages let you specify that the match is case insensitive (although they differ in the syntax that you use to achieve this).

- A.15 EXAMPLE The web document language HTML document tag for an image, such as ``, uses either of the keys `src` or `img` to give the name of the file containing the image. Those strings can be in upper case or lower case, or any mix. Racket uses a ‘`?i:`’ syntax to mark part of the regex as insensitive: `\s+(?i:(img|src))=`. (Note also the double backslash, which is how Racket escapes the backslash.)

**Beyond convenience** The regular expression engines that come with recent programming languages have capabilities beyond matching only those languages that are recognized by Finite State machines.

- A.16 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 a pipe. However, instead we can have the system remember what it finds at the start and look for that again at the end. Thus, the regex `<([^\>]+)>.*</\1>` matches HTML tags like the ones given. Its second character is an open parenthesis, and the `\1` refers to everything between that open parenthesis and the matching close parenthesis (and, that is not a typo; Racket's syntax calls for double backslashes). As is hinted by the `1`, you can also have a second match with `\2`, etc.

That is a **back reference**. It is very convenient. However, it gives extended regular expressions more power than the theoretical regular expressions that we studied earlier.

- A.17 EXAMPLE This is the language of **squares** over  $\Sigma = \{a, b\}$ .

$$\mathcal{L} = \{\sigma \in \Sigma^* \mid \sigma = \tau \hat{\cdot} \tau \text{ for some } \tau \in \Sigma^*\}$$

Some members are `aabaab`, `baaabaaa`, and `aa`. The Pumping Lemma shows that the language of squares is not regular; see Exercise A.37. Describe this language with the regex `(.+)\1`; note the back-reference.

- A.18 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 prime numbers represented in unary,  $\mathcal{L} = \{1^n \mid n \text{ is not prime}\}$ . It is described by the regex `^1?$_|^^(1+?)\\1+$`. A brief explanation: the `^1?$_` matches a string that is either zero-many or one-many 1's. The `^(1+?)\\1+$` matches a group of 1's repeated one or more times. Being able to divide the number of 1's into some number of subgroups is what characterizes a unary number as composite, that is, not prime.

**Tradeoffs** Regexes are powerful tools. But they come with downsides.

For instance, the regular expression for twelve hour time from Example A.3 (`(\s|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 given without them. We could change the ending to `(\s|am|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.<sup>†</sup>



Courtesy [xkcd.com](http://xkcd.com)

<sup>†</sup> Some substrings are elided so it fits in the margins.

(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 can have a tendency to grow, to accrete subcases like this.

Another example is the Canadian postal expression in Example A.1. Not every matching string has a corresponding physical post office—for one thing, no valid codes begin with Z. And US ZIP codes work the same way; there are fewer than 50 000 assigned ZIP codes, so many five digits strings are not in use. Changing the regular expressions to cover only those codes actually in use would make them just lists of strings, which would change frequently.

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.

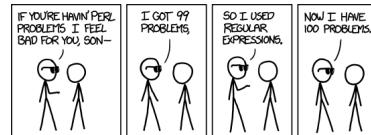
```
(?:(?:\r\n)?[ \t])*?(?:(?:^(?)<@,;:\\".\[\] \000-\031]+?(?:^(?:\r\n)?[ \t]
)+|\Z|(?=[\"]()<@,;:\\".\[\]))|"^(?:[^"\r\\]|\\.|(?:^(?:\r\n)?[ \t]))*(?:^(?:\r\n)?[ \t])*(?:\.(?:^(?:\r\n)?[ \t])*?(?:[^()<@,;:\\".\[\]]))|^"(?:[^"\r\\]|\\.|(?:^(?:\r\n)?[ \t]))*(?:^(?:\r\n)?[ \t])*)*?(?:^(?:\r\n)?[ \t])*)*@(?:^(?:\r\n)?[ \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 regular expressions 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 regular expression 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 regexps 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](http://xkcd.com)

## IV.A Exercises

- ✓ A.19 Which of the strings matches the regex  $ab+c$ ? (A) abc (B) ac (C) abbb (D) bbc
- A.20 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.21 Give an extended regular expression 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.22 Give an extended regular expression 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.23 Give an extended regular expression 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.24 Give an extended regular expression matching strings that contain an open square bracket and an open curly brace.
- ✓ A.25 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.26 Example A.5 gives a regex for North American phone numbers.  
(A) They are sometimes written with parentheses around the area code. Extend the regex to cover this case.  
(B) Sometimes phone numbers do not include the area code. Extend to cover this also.
- A.27 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.28 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.29 Give a regex describing a floating point number.
- ✓ A.30 Give a suitable extended regular expression.

- (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.31 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.32 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.33 In the Tradeoffs discussion, we change the ending to  $(\epsilon \mid \text{am} \mid \text{pm})$ . Why not  $\text{\s}(\epsilon \mid \text{am} \mid \text{pm})$ , which factors out the whitespace?
- A.34 Imagine that you decide to avoid regexes but still want to do the sanity check for email addresses discussed above, of accepting the string if and only if it consists of a nonempty string of characters, followed by @, followed by a nonempty string of characters. Implement that as a routine in your favorite language.
- A.35 Give a regex that matches no string.
- ✓ A.36 The Roman numerals from grade school use the letters I, V, X, L, C, D, and M to represent 1, 5, 10, 50, 100, 500, and 1000. They are written in descending order of magnitude, from M to I, and are written greedily so that we don't write six I's but rather VI. Thus, the date written on the book held by the Statue of Liberty is MDCCCLXXVI, for 1776. Further, we replace IIII with IV, and replace VIII with IX. Give a regular expression for valid Roman numerals less than 5000.
- A.37 Example A.17 says that the language of squares over  $\Sigma = \{a, b\}$

$$\mathcal{L} = \{\sigma \in \Sigma^* \mid \sigma = \tau^\wedge \tau \text{ for some } \tau \in \Sigma^*\}$$

is not regular. Verify that.

A.38 Consider  $\mathcal{L} = \{0^n 1 0^n \mid n > 0\}$ . (A) Show that it is not regular. (B) Find a regex.

A.39 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.40 In a **regex crossword** each row and column has a regular expression. You have to find strings for those rows and columns that meet the constraints.

	$[^SPEAK]^+$	$EP \mid IP \mid EF$	
(A)			
	$HE \mid LL \mid O^+ \quad [PLEASE]^+$	$[ ] \quad [ ]$	

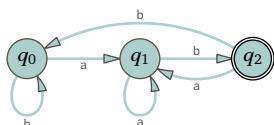
	$(A \mid B \mid C) \setminus 1$	$(AB \mid OE \mid SK)$
(B)		
	$.^*M?O.^* \quad (AN \mid FE \mid BE)$	$[ ] \quad [ ]$

#### EXTRA

#### IV.B The Myhill-Nerode Theorem

We defined regular languages in terms of Finite State machines. Here we will give a characterization that does not depend on that.

This deterministic Finite State machine accepts strings that end in ab.

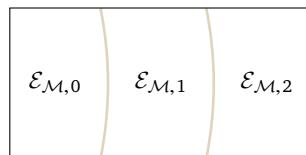


Consider other strings over  $\Sigma = \{a, b\}$ , not just the accepted ones, and see where they bring the machine.

Input string $\sigma$	$\epsilon$	a	b	aa	ab	ba	bb	aaa	aab	aba	abb
Ending state $\hat{\Delta}(\sigma)$	$q_0$	$q_1$	$q_0$	$q_1$	$q_2$	$q_1$	$q_0$	$q_1$	$q_2$	$q_1$	$q_0$

The collection of all strings  $\Sigma^*$ , pictured below, breaks into three sets, those that bring the machine to  $q_0$ , those that bring the machine to  $q_1$ , and those that bring the machine to  $q_2$ .

$$\begin{aligned}\mathcal{E}_{M,0} &= \{\epsilon, b, bb, abb, \dots\} \\ \mathcal{E}_{M,1} &= \{a, aa, ba, aba, \dots\} \\ \mathcal{E}_{M,2} &= \{ab, aab, abb, \dots\}\end{aligned}$$



B.1 **DEFINITION** Let  $\mathcal{M}$  be a Finite State machine with alphabet  $\Sigma$ . Two strings  $\sigma_0, \sigma_1 \in \Sigma^*$  are  **$\mathcal{M}$ -related** if starting the machine with input  $\sigma_0$  ends with it in the same state as does starting the machine with input  $\sigma_1$ .

B.2 **LEMMA** The binary relation of  $\mathcal{M}$ -related is an equivalence, and so partitions the collection of all strings  $\Sigma^*$  into equivalence classes.

*Proof* We must show that the relation is reflexive, symmetric, and transitive. Reflexivity, that any input string  $\sigma$  brings the machine to the same state as itself, is obvious. So is symmetry, that if  $\sigma_0$  brings the machine to the same state as  $\sigma_1$  then  $\sigma_1$  brings it to the same state as  $\sigma_0$ . Transitivity is straightforward: if  $\sigma_0$  brings  $\mathcal{M}$  to the same state as  $\sigma_1$ , and  $\sigma_1$  brings it to the same state as  $\sigma_2$ , then  $\sigma_0$  brings it to the same state as  $\sigma_2$ .  $\square$

So a machine gives rise to a partition. Does it go the other way—if we have a partition of a language, is there an associated machine?

This converse is ruled out by a counting argument. Consider the alphabet  $\mathbb{B} = \{0, 1\}$ . There are uncountably many partitions of the language  $\mathbb{B}^*$  but there are only countably many Finite State machines with that alphabet. Thus it is not the case that for any partition there is an associated machine.

But some further reflection on the above example gives a limit on which partition arise due to the action of Finite State machines. The  $\mathcal{M}$  relation gives rise to a partition where there is one class for each state, and thus the partition has only finitely many classes. We will show that if a partition of a language has finitely many classes then there is an associated machine. Further, the argument below is constructive, meaning that it shows us how to make the machine from the partition.

B.3 **DEFINITION** Suppose that  $\mathcal{L}$  is a language over  $\Sigma$ . Two strings  $\sigma, \hat{\sigma} \in \Sigma^*$  are  **$\mathcal{L}$ -related** (or  **$\mathcal{L}$ -indistinguishable**), denoted  $\sigma \sim_{\mathcal{L}} \hat{\sigma}$ , when for every suffix  $\tau \in \Sigma^*$  we have  $\sigma^\wedge \tau \in \mathcal{L}$  if and only if  $\hat{\sigma}^\wedge \tau \in \mathcal{L}$ . Otherwise, the two strings are  **$\mathcal{L}$ -distinguishable**.

Said another way, the two strings  $\sigma$  and  $\hat{\sigma}$  can be  $\mathcal{L}$ -distinguished when there is a suffix  $\tau$  that separates them: of the two  $\sigma^\wedge \tau$  and  $\hat{\sigma}^\wedge \tau$ , one is an element of  $\mathcal{L}$  while the other is not.

B.4 **LEMMA** For any language  $\mathcal{L}$ , the binary relation  $\sim_{\mathcal{L}}$  is an equivalence, and thus gives rise to a partition of all strings.

*Proof* Reflexivity, that  $\sigma \sim_{\mathcal{L}} \sigma$ , is trivial. So is symmetry, that  $\sigma_0 \sim_{\mathcal{L}} \sigma_1$  implies  $\sigma_1 \sim_{\mathcal{L}} \sigma_0$ . For transitivity suppose  $\sigma_0 \sim_{\mathcal{L}} \sigma_1$  and  $\sigma_1 \sim_{\mathcal{L}} \sigma_2$ . If  $\sigma_0^\wedge \tau \in \mathcal{L}$  then by the first supposition  $\sigma_1^\wedge \tau \in \mathcal{L}$ , and the second supposition in turn gives  $\sigma_2^\wedge \tau \in \mathcal{L}$ . Similarly  $\sigma_0^\wedge \tau \notin \mathcal{L}$  implies that  $\sigma_2^\wedge \tau \notin \mathcal{L}$ . Thus  $\sigma_0 \sim_{\mathcal{L}} \sigma_2$ .  $\square$

B.5 **EXAMPLE** Let  $\mathcal{L}$  be the set  $\{\sigma \in \mathbb{B}^* \mid \sigma \text{ has an even number of } 1\text{'s}\}$ . We can find the parts of the partition. If two strings  $\sigma_0, \sigma_1$  both have an even number of 1's

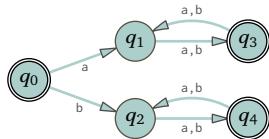
then they are  $\mathcal{L}$ -related. That's because for any  $\tau \in \mathbb{B}^*$ , if  $\tau$  has an even number of 1's then  $\sigma_0 \hat{\cdot} \tau \in \mathcal{L}$  and  $\sigma_0 \hat{\cdot} \tau \in \mathcal{L}$ , while if  $\tau$  has an odd number of 1's then the concatenations will not be members of  $\mathcal{L}$ . Similarly, two strings both have an odd number of 1's then they are  $\mathcal{L}$ -related. So the relationship  $\sim_{\mathcal{L}}$  gives rise to this partition of  $\mathbb{B}^*$ .

$$\mathcal{E}_{\mathcal{L},0} = \{\varepsilon, 0, 00, 11, 000, 011, 101, 110, \dots\} \quad \mathcal{E}_{\mathcal{L},1} = \{1, 01, 10, 001, 010, \dots\}$$

- B.6 EXAMPLE Let  $\mathcal{L}$  be  $\{\sigma \in \{a, b\}^* \mid \sigma \text{ has the same number of } a\text{'s as } b\text{'s}\}$ . Then two members of  $\mathcal{L}$ , two strings  $\sigma_0, \sigma_1 \in \Sigma^*$  with the same number of  $a$ 's as  $b$ 's, are  $\mathcal{L}$ -related. This is because for any suffix  $\tau$ , the string  $\sigma_0 \hat{\cdot} \tau$  is an element of  $\mathcal{L}$  if and only if  $\sigma_1 \hat{\cdot} \tau$  is an element of  $\mathcal{L}$ , which happens if and only if  $\tau$  has the same number of  $a$ 's as  $b$ 's.

Similarly, two strings  $\sigma_0, \sigma_1$  such that the number of  $a$ 's is one more than the number of  $b$ 's are  $\mathcal{L}$ -related because for any suffix  $\tau$ , the string  $\sigma_0 \hat{\cdot} \tau$  is an element of  $\mathcal{L}$  if and only if  $\sigma_1 \hat{\cdot} \tau$  is an element of  $\mathcal{L}$ , namely if and only if  $\tau$  has one fewer  $a$  than  $b$ . Following this reasoning,  $\sim_{\mathcal{L}}$  partitions  $\{a, b\}^*$  into the infinitely many parts  $\mathcal{E}_{\mathcal{L},i} = \{\sigma \in \{a, b\}^* \mid \text{the number of } a\text{'s minus the number of } b\text{'s equals } i\}$ , where  $i \in \mathbb{Z}$ .

- B.7 EXAMPLE This machine  $\mathcal{M}$  recognizes  $\mathcal{L} = \{\sigma \in \{a, b\}^* \mid \sigma \text{ has even length}\}$ .



We will compare the partitions induced by the two relations above. The  $\mathcal{M}$ -related relation breaks  $\{a, b\}^*$  into five parts, one for each state (since each state in  $\mathcal{M}$  is reachable).

$$\mathcal{E}_{\mathcal{M},0} = \{\varepsilon\}$$

$$\mathcal{E}_{\mathcal{M},1} = \{a, aaa, aab, aba, abb, aaaaa, aaaab, \dots\}$$

$$\mathcal{E}_{\mathcal{M},2} = \{b, baa, bab, bba, bbb, baaaa, baaab, \dots\}$$

$$\mathcal{E}_{\mathcal{M},3} = \{aa, ab, aaaa, aaab, aaba, aabb, abaa, abab, abba, abbb, aaaaaa, \dots\}$$

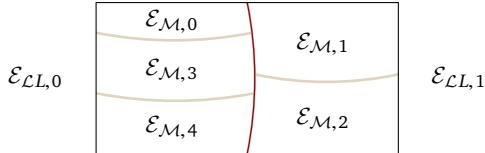
$$\mathcal{E}_{\mathcal{M},4} = \{ba, bb, baaa, baab, baba, babb, bbaa, bbab, bbba, bbbb, baaaaa, \dots\}$$

The  $\mathcal{L}$ -related relation breaks  $\{a, b\}^*$  into two parts.

$$\mathcal{E}_{\mathcal{L},0} = \{\sigma \mid \sigma \text{ has even length}\} \quad \mathcal{E}_{\mathcal{L},1} = \{\sigma \mid \sigma \text{ has odd length}\}$$

Verify this by noting that if two strings are in  $\mathcal{E}_{\mathcal{L},0}$  then adding a suffix  $\tau$  will result in a string that is a member of  $\mathcal{L}$  if and only if the length of  $\tau$  is even, and the same reasoning holds for  $\mathcal{E}_{\mathcal{L},1}$  and odd-length  $\tau$ 's.

The sketch below shows the universe of strings  $\{a, b\}^*$ , partitioned in two ways. There are two  $\mathcal{L}$ -related parts, the left and right halves. The five  $\mathcal{M}$ -related parts are subsets of the  $\mathcal{L}$ -related parts.



That is, the  $\mathcal{M}$ -related partition is finer than the  $\mathcal{L}$ -related partition ('fine' in the sense that sand is finer than gravel).

- B.8 **LEMMA** Let  $\mathcal{M}$  be a Finite State machine that recognizes  $\mathcal{L}$ . If two strings are  $\mathcal{M}$ -related then they are  $\mathcal{L}$ -related.

*Proof* Assume that  $\sigma_0$  and  $\sigma_1$  are  $\mathcal{M}$ -related, so that starting  $\mathcal{M}$  with input  $\sigma_0$  causes it to end in the same state as starting it with input  $\sigma_1$ . Thus for any suffix  $\tau$ , giving  $\mathcal{M}$  the input  $\sigma_0 \hat{\cdot} \tau$  causes it to end in the same state as does the input  $\sigma_1 \hat{\cdot} \tau$ . In particular,  $\sigma_0 \hat{\cdot} \tau$  takes  $\mathcal{M}$  to a final state if and only if  $\sigma_1 \hat{\cdot} \tau$  does. So the two strings are  $\mathcal{L}$ -related.  $\square$

- B.9 **LEMMA** Let  $\mathcal{L}$  be a language. (1) If two strings  $\sigma_0, \sigma_1$  are  $\mathcal{L}$ -related,  $\sigma_0 \sim_{\mathcal{L}} \sigma_1$ , then adjoining a common extension  $\beta$  gives strings that are also  $\mathcal{L}$ -related,  $\sigma_0 \hat{\cdot} \beta \sim_{\mathcal{L}} \sigma_1 \hat{\cdot} \beta$ . (2) If one member of a part  $\sigma_0 \in \mathcal{E}_{\mathcal{L},i}$  is an element of  $\mathcal{L}$  then every member of that part  $\sigma_1 \in \mathcal{E}_{\mathcal{L},i}$  is also an element of  $\mathcal{L}$ .

*Proof* For (1), start with two strings  $\sigma_0, \sigma_1$  that are  $\mathcal{L}$ -related. By definition, no extension  $\tau$  will  $\mathcal{L}$ -distinguish the two—it is not the case that one of  $\sigma_0 \hat{\cdot} \tau, \sigma_1 \hat{\cdot} \tau$  is in  $\mathcal{L}$  while the other is not. Taking  $\beta \hat{\cdot} \hat{\tau}$  for  $\tau$  gives that for the two strings  $\sigma_0 \hat{\cdot} \beta$  and  $\sigma_1 \hat{\cdot} \beta$ , no extension  $\hat{\tau}$  will  $\mathcal{L}$ -distinguish the two. So they are  $\mathcal{L}$ -related.

Item (2) is even easier: if  $\sigma_0 \sim_{\mathcal{L}} \sigma_1$  and  $\sigma_0 \in \mathcal{L}$  but  $\sigma_1 \notin \mathcal{L}$  then they are distinguished by the empty string, which contradicts that they are  $\mathcal{L}$ -related.  $\square$

- B.10 **EXAMPLE** We will milk Example B.7 for another observation. Take a string  $\sigma$  from  $\mathcal{E}_{\mathcal{M},1}$  and append an a. The result  $\sigma \hat{\cdot} a$  is a member of  $\mathcal{E}_{\mathcal{M},3}$ , simply because if the machine is in state  $q_1$  and it receives an a then it moves to state  $q_3$ . Likewise, if  $\sigma \in \mathcal{E}_{\mathcal{M},4}$ , then  $\sigma \hat{\cdot} b$  is a member of  $\mathcal{E}_{\mathcal{M},2}$ . If adding the alphabet character  $x \in \Sigma$  to one string  $\sigma$  from  $\mathcal{E}_{\mathcal{L},i}$  results in a string  $\sigma \hat{\cdot} x$  from  $\mathcal{E}_{\mathcal{L},j}$  then the same will happen for any string from  $\mathcal{E}_{\mathcal{L},i}$ .

In this example we see that's true because the  $\mathcal{E}_{\mathcal{M}}$ 's are contained in the  $\mathcal{E}_{\mathcal{L}}$ 's. The key step of the next result is to find it even in a context where there is no machine.

- B.11 **THEOREM (MYHILL-NERODE)** A language  $\mathcal{L}$  is regular if and only if the relation  $\sim_{\mathcal{L}}$  has only finitely many equivalence classes.

*Proof* One direction is easy. Suppose that  $\mathcal{L}$  is a regular language. Then it is recognized by a Finite State machine  $\mathcal{M}$ . By Lemma B.8 the number of elements in the partition induced by  $\sim_{\mathcal{L}}$  is finite because the number of elements in the partition associated with being  $\mathcal{M}$ -related is finite, as there is one part for each of  $\mathcal{M}$ 's reachable states.

For the other direction suppose that the number of elements in the partition associated with being  $\mathcal{L}$ -related is finite. We will show that  $\mathcal{L}$  is regular by producing a Finite State machine that recognizes  $\mathcal{L}$ .

The machine's states are the partition's elements, the  $\mathcal{E}_{\mathcal{L},i}$ 's. That is,  $s_i$  is  $\mathcal{E}_{\mathcal{L},i}$ . The start state is the part containing the empty string  $\varepsilon$ . A state is final if that part contains strings from the language  $\mathcal{L}$  (Lemma B.9 (2) says that each part contains either no strings from  $\mathcal{L}$  or consists entirely of strings from  $\mathcal{L}$ ).

The transition function is: for any state  $s_i = \mathcal{E}_{\mathcal{L},i}$  and alphabet element  $x$ , compute the next state  $\Delta(s_i, x)$  by starting with any string in that part  $\sigma \in \mathcal{E}_{\mathcal{L},i}$ , appending the character to get a new string  $\hat{\sigma} = \sigma \hat{x}$ , and then finding the part containing that string, the  $\mathcal{E}_{\mathcal{L},j}$  such that  $\hat{\sigma} \in \mathcal{E}_{\mathcal{L},j}$ . Then  $\Delta(s_i, x) = s_j$ .

We must verify that this transition function is well-defined. That is, the definition of  $\Delta(s_i, x)$  as given potentially depends on which string  $\sigma$  you choose from  $s_i = \mathcal{E}_{\mathcal{L},i}$ , and we must check that choosing a different string cannot lead to a different resulting part. This follows from (1) in Lemma B.9: take two starting strings from the same part  $\sigma_0, \sigma_1 \in \mathcal{E}_{\mathcal{L},i}$  and make a common extension by the one-element string  $\beta = \langle x \rangle$  so the results are in the same part  $\sigma_0 \sim_{\mathcal{L}} \sigma_1$ .

Here is an equivalent way to describe the next-state function that is illuminating. Recall that we write the part containing  $\sigma$  as  $[\![\sigma]\!]$ . Then the definition of the transition function for the machine under construction is  $\Delta([\![\sigma]\!], x) = [\![\sigma \hat{x}]\!]$ . With that, a simple induction shows that the extended transition function in the new machine is  $\hat{\Delta}(\alpha) = [\![\alpha]\!]$ .

Finally, we must verify that the language recognized by this machine is  $\mathcal{L}$ . For any string  $\sigma \in \Sigma^*$ , starting this machine with  $\sigma$  as input will cause the machine to end in the partition containing  $\sigma$ ; this is what the prior paragraph says. This string will be accepted by this machine if and only if  $\sigma \in \mathcal{L}$ .  $\square$

## IV.B Exercises

- ✓ B.12 Find the  $\mathcal{L}$  equivalence classes for each regular set. The alphabet is  $\Sigma = \{a, b\}$ .
  - (A)  $\mathcal{L}_0 = \{a^n b \mid n \in \mathbb{N}\}$
  - (B)  $\mathcal{L}_1 = \{a^2 b^n \mid n \in \mathbb{N}\}$
- ✓ B.13 For each language describe the  $\mathcal{L}$  equivalence classes. The alphabet is  $\mathbb{B}$ .
  - (A) The set of strings ending in 01
  - (B) The set of strings where every 0 is immediately followed by two 1's
  - (C) The set of strings with the substring 0110
  - (D) The set of strings without the substring 0110

- ✓ B.14 The language of palindromes  $\mathcal{L} = \{\sigma \in a, b^* \mid \sigma^R = \sigma\}$  is not regular. Find infinitely many  $\mathcal{L}$  equivalence classes.
- ✓ B.15 Show that the language  $\mathcal{L} = \{a^n b^n \mid n \in \mathbb{N}\}$  is not regular by using the Myhill-Nerode Theorem.

Part Three

## Computational Complexity



## CHAPTER

# V Computational Complexity

Earlier, we asked what can be done with a mechanism at all. This mirrors the subject's history: when the Theory of Computing began there were no physical computers. Researchers were driven by considerations such as the *Entscheidungsproblem*. The subject was interesting, the questions compelling, and there were plenty of problems, but the initial phase had a theory-driven feel.

A natural next step is to look to do jobs efficiently. When physical computers became widely available, that's exactly what happened. Today, the Theory of Computing has incorporated many questions that at least originate in applied fields, and that need answers that are feasible.

We will review how we determine the practicality of algorithms, the order of growth of functions. Then we will see a collection of the kinds of problems that drive the field today. By the end of this chapter we will be at the research frontier and we will state some things without proof, as well as discuss some things about which we are not sure. In particular, we will consider the celebrated question of P versus NP.

## SECTION

### V.1 Big $\mathcal{O}$

We begin by reviewing the definition of the order of growth of functions. We will study this because it measures how algorithms consume computational resources.

First, an anecdote. Here is a grade school multiplication.

$$\begin{array}{r} 678 \\ \times 42 \\ \hline 1356 \\ 2712 \\ \hline 28476 \end{array}$$

The algorithm combines each digit of the multiplier 42 with each digit of the multiplicand 678, in a nested loop. A person could sensibly feel that this is the right way to compute multiplication—indeed, the only reasonable way—and that in general, to multiply two  $n$  digit numbers requires about  $n^2$ -many operations.

---

IMAGE: Striders can walk on water because they are five orders of magnitude smaller than us. This change of scale changes the world—bugs see surface tension as more important than gravity. Similarly, finding an algorithm that changes the time that it takes to solve a problem from  $n^2$  to  $n \cdot \lg n$  can make something easy that was previously not practical.

In 1960, A Kolmogorov organized a seminar at Moscow State University aimed at proving this. But, before the seminar's second meeting, one of the students, A Karatsuba, discovered that it is false. He produced a clever algorithm that used only  $n^{\lg(3)} \approx n^{1.585}$  operations. At the next meeting, Kolmogorov explained the result and closed the seminar.

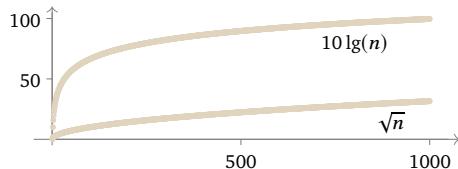


Andrey Kolmogorov 1903–1987

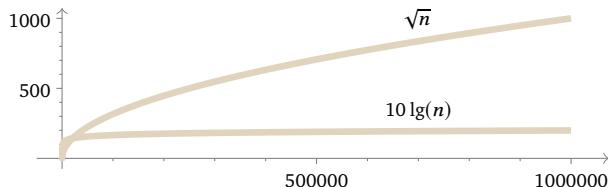
And this continues: every day, researchers produce results saying, “for this job, here is a way to do it in less time, or less space, etc.”<sup>†</sup> People are good at finding clever algorithms that solve a problem using less of some computational resource. But we are not good at finding lower bounds, at proving “no algorithm, no matter how clever, can do the job faster than such and such.” This is one reason that we will compare the growth rates of resource use by algorithms using a measure, Big  $\mathcal{O}$ , that is like ‘less than or equal to’.

**Motivation** To compare the performance of algorithms, we need a way to measure that performance. Typically, an algorithm takes longer on longer input. So we describe the time performance of an algorithm with a function whose argument is the input size and whose value is the maximum time that the algorithm takes on all inputs of that size, or less.

Next we develop the criteria for the definition of Big  $\mathcal{O}$ , the tool that we use to compare those functions. Suppose first that we have two algorithms. When the input is size  $n \in \mathbb{N}$ , one takes  $\sqrt{n}$  many ticks while the other takes  $10 \cdot \lg(n)$ .<sup>‡</sup> Initially,  $\sqrt{n}$  looks better. For instance,  $\sqrt{1000} \approx 31.62$  and  $10 \lg(1000) \approx 99.66$ .



However, for large  $n$  the value  $\sqrt{n}$  is much bigger than  $10 \lg(n)$ . For instance,  $\sqrt{1\,000\,000} = 1\,000$  while  $10 \lg(1\,000\,000) \approx 199.32$ .



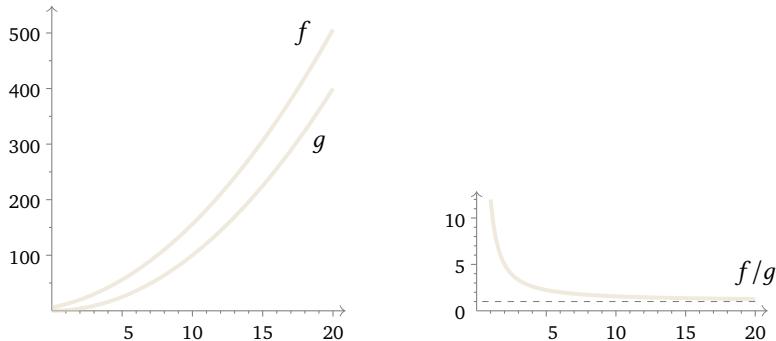
<sup>†</sup>See the Theory of Computing blog feed at <http://cstheory-feed.org/> (Various authors 2017).

<sup>‡</sup>Recall that  $\lg(n) = \log_2(n)$ . That is, compute  $\lg(n)$  by starting with  $n$  and then finding the power of 2 that produces it, so if  $n = 8$  then  $\lg(n) = 3$  and if  $n = 10$  then  $\lg(n) \approx 3.32$ .

So the first criteria is that big O's definition must focus on what happens in the long run.

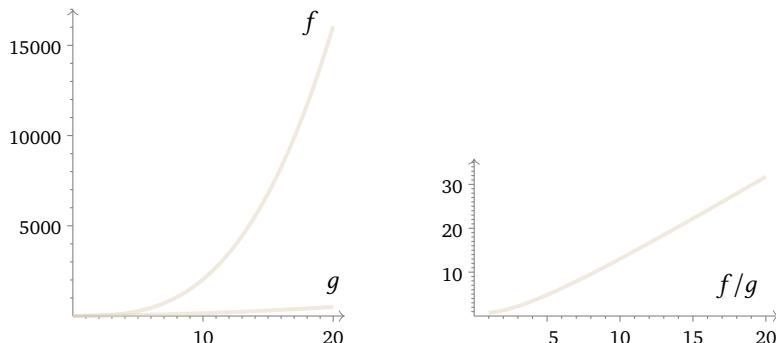
The second criteria is more subtle. Consider the following four examples.

- 1.1 EXAMPLE These graphs compare  $f(n) = n^2 + 5n + 6$  with  $g(n) = n^2$ . The graph on the right compares them in ratio,  $f/g$ .<sup>†</sup>



On the left our eye is struck that  $n^2 + 5n + 6$  is ahead of  $n^2$ . But on the right the ratios show that this is misleading. For large inputs,  $f$ 's  $5n$  and 6 are swamped by the highest order term, the  $n^2$ . Consequently these two functions track together—by far the biggest factor in the behavior of these two is that they are both quadratic—and their long run behavior is basically the same.

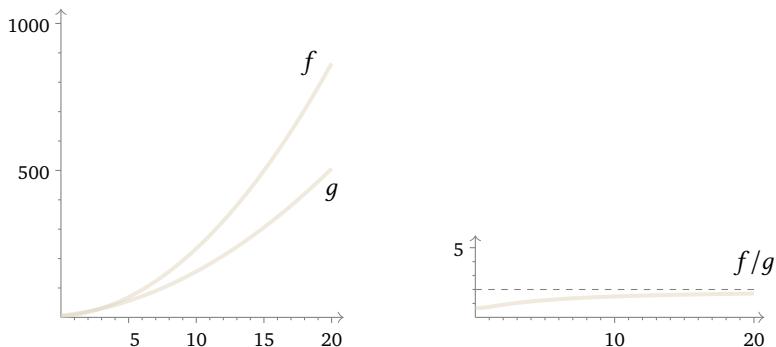
- 1.2 EXAMPLE Next compare the quadratic  $g(n) = n^2 + 5n + 6$  with the cubic  $f(x) = n^3 + 2n + 3$ . In contrast to the prior example, these two don't track together. Initially  $g$  is larger, with  $g(0) = 6 > f(0) = 3$  and  $g(1) = 12 > f(1) = 6$ . But then the cubic accelerates ahead of the quadratic, so much that at the scale of the image, the graph of  $g$  doesn't rise much above the axis.



<sup>†</sup>These graphs show functions whose domain is the real numbers. It may seem more natural, because Turing machines are discrete devices, to have the domain be the natural numbers. But we will see that real functions are the most convenient ones for complexity comparisons.

On the right, the ratio rises to infinity. So  $f$  is a faster-growing function than  $g$ . They both go to infinity but  $f$  goes there faster.

- 1.3 EXAMPLE Now compare the quadratics  $f(x) = 2n^2 + 3n + 4$  and  $g(n) = n^2 + 5n + 6$ . We've already seen, as described above, that the function comparison definition needs to discount the initial behavior that  $f(0) = 4 < g(0) = 6$  and  $f(1) = 9 < g(1) = 12$ , and instead focus on the long run.

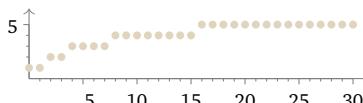


This example differs from Example 1.1 in that in the long run,  $f$  stays ahead of  $g$  and gains in an absolute sense, because of  $f$ 's dominant term  $2n^2$ , compared with  $g$ 's  $n^2$ . So it may appear that we should view  $g$ 's rate as less than  $f$ 's. However unlike in the prior example,  $f$  does not accelerate away. Instead, the ratio between the two is bounded. We will take  $g$  to be equivalent to  $f$ .

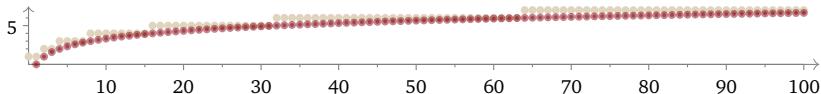
- 1.4 EXAMPLE We close the motivation with a very important example. Let the function bits:  $\mathbb{N} \rightarrow \mathbb{N}$  give the number of bits needed to represent its input in binary. The bottom line of this table gives  $\lg(n)$ , the power of 2 that equals  $n$ .

<i>Input n</i>	0	1	2	3	4	5	6	7	8	9
<i>Binary</i>	0	1	10	11	100	101	110	111	1000	1001
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

This shows  $\text{bits}(n)$ , the table's third line, for  $n \in \{1, \dots, 30\}$ .



The relationship between the third and fourth lines is that  $\text{bits}(n) = 1 + \lfloor \lg(n) \rfloor$ , except that  $\text{bits}(n) = 1$  if  $n = 0$ . The graph below compares  $\text{bits}(n)$  with  $\lg(n)$ . (Note the change in the horizontal and vertical scales, and that the domain of  $\lg(n)$  does not include 0.)



This illustrates that over the long run the ‘+1’ and the floor do not matter much. A reasonable summary is that the base 2 logarithm,  $\lg n$ , describes the number of bits required to represent the number  $n$ .

Further, the formula for converting among logarithmic functions with different bases,  $\log_c(x) = \log_b(x)/\log_b(c)$ , shows that they differ only by the constant factor  $1/\log_b(c)$ . As Example 1.3 notes, with the function comparison definition given below we will disregard constant factors. So even the base does not matter—another reasonable summary is that the number of bits is “a” logarithmic function.

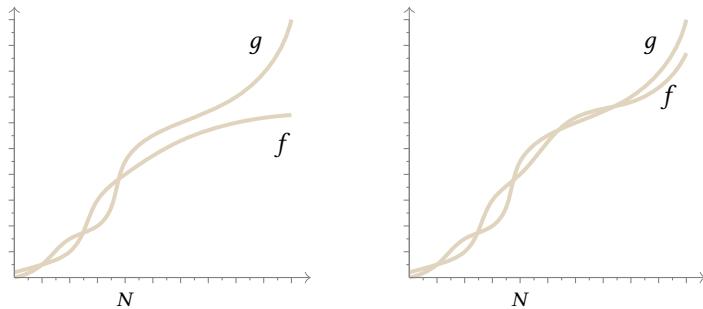
**Definition** Machine resource sizes, such as the number of bits of the input and of memory, are natural numbers. So to describe the performance of algorithms we may think to focus on functions that input and output natural numbers. However, above we have already found useful a function,  $\lg$ , that inputs and outputs reals.<sup>†</sup> So instead we will consider a subset of the real functions.<sup>†</sup>

- 1.5 **DEFINITION** A **complexity function**  $f$  is one that inputs real number arguments and outputs real number values, and (1) has an **unbounded domain** in that there is a number  $N \in \mathbb{R}^+$  such that  $x \geq N$  implies that  $f(x)$  is defined, and (2) is **eventually nonnegative** in that there is a number  $M \in \mathbb{R}^+$  so that  $x \geq M$  implies that  $f(x) \geq 0$ .
- 1.6 **DEFINITION** Let  $g$  be a complexity function. Then **Big O** of  $g$ ,  $\mathcal{O}(g)$ , is the set of complexity functions  $f$  satisfying that there are constants  $N, C \in \mathbb{R}^+$  so that if  $x \geq N$  then both  $g(x)$  and  $f(x)$  are defined and  $C \cdot g(x) \geq f(x)$ . We say that  $f$  is  $\mathcal{O}(g)$ , or that  $f \in \mathcal{O}(g)$ , or that  $f$  is of order at most  $g$ , or that  $f = \mathcal{O}(g)$ .
- 1.7 **REMARKS** (1) We use the letter ‘ $\mathcal{O}$ ’ because this is about the order of growth. (2) The term ‘complexity function’ is not standard but we will find it convenient. (3) We may say ‘ $x^2 + 5x + 6$  is  $\mathcal{O}(x^2)$ ’ instead of ‘ $f$  is  $\mathcal{O}(g)$  where  $f(x) = x^2 + 5x + 6$  and  $g(x) = x^2$ ’. (4) The ‘ $f = \mathcal{O}(g)$ ’ notation is very common, but awkward. It does not follow the usual rules of equality, such as that  $f = \mathcal{O}(g)$  does not allow us to write ‘ $\mathcal{O}(g) = f$ ’. Another is that  $x = \mathcal{O}(x^2)$  and  $x^2 = \mathcal{O}(x^2)$  together do not imply that  $x = x^2$ . (5) Some authors allow negative real outputs and write the inequality with absolute values,  $f(x) \leq C \cdot |g(x)|$ . (6) Sometimes you see ‘ $f$  is  $\mathcal{O}(g)$ ’ stated as ‘ $f(x)$  is  $\mathcal{O}(g(x))$ ’. Speaking strictly, this is wrong because  $f(x)$  and  $g(x)$  are numbers, not functions.

Think of ‘ $f$  is  $\mathcal{O}(g)$ ’ as meaning that  $f$ ’s growth rate is less than or equal to  $g$ ’s rate. The sketches below illustrate. On the left  $g$  appears to accelerate away

<sup>†</sup>Using real functions has the disadvantage that it can seem to leave out natural number functions such as  $n!$ . One way to deal with this is to extend these to take real number arguments, for instance, extending the factorial to  $|x|!$ , whose domain is the set of nonnegative reals (or to the more advanced  $\Gamma$  function).

from  $f$ , so that  $g$ 's growth rate is greater than  $f$ 's. On the right the two seem to track together, so that  $f$  is  $\mathcal{O}(g)$  and also  $g$  is  $\mathcal{O}(f)$ .



To apply the definition we must produce, and verify, suitable  $N$  and  $C$ .

- 1.8 EXAMPLE Let  $f(x) = x^2$  and  $g(x) = x^3$ . Then  $f$  is  $\mathcal{O}(g)$ , as witnessed by  $N = 2$  and  $C = 1$ . The verification is:  $x > N = 2$  implies that  $g(x) = x^3 = x \cdot x^2$  is greater than  $2 \cdot x^2$ , which in turn is greater than  $x^2 = C \cdot f(x) = 1 \cdot f(x)$ .

If  $f(x) = 5x^2$  and  $g(x) = x^4$  then to show  $f$  is  $\mathcal{O}(g)$  take  $N = 2$  and  $C = 2$ . The verification is that  $x > N = 2$  implies that  $C \cdot x^4 = 2 \cdot x^2 \cdot x^2 \geq 8x^2 > 5x^2$ .

- 1.9 EXAMPLE Don't confuse a function having values that are smaller with its growth rate being smaller. Let  $g(x) = x^2$  and  $f(x) = x^2 + 1$ , so that  $g(x) < f(x)$ . But  $g$ 's growth rate is not smaller; rather,  $f$  is  $\mathcal{O}(g)$ . To verify, take  $N = 2$  and  $C = 2$ . Then  $x \geq N = 2$  gives  $C \cdot g(x) = 2x^2 = x^2 + x^2 > x^2 + 1 = f(x)$ .

- 1.10 EXAMPLE Let  $Z: \mathbb{R} \rightarrow \mathbb{R}$  be the zero function,  $Z(n) = 0$ . Then  $Z$  is  $\mathcal{O}(g)$  for every complexity function  $g$ . Verify that with  $N = 1$  and  $C = 1$ .

- 1.11 EXAMPLE Some pairs of functions aren't comparable, so that neither  $f \in \mathcal{O}(g)$  nor  $g \in \mathcal{O}(f)$ . For an instance, let  $g(x) = x^3$  and consider this function.

$$f(x) = \begin{cases} x^2 & \text{if } \lfloor x \rfloor \text{ is even} \\ x^4 & \text{if } \lfloor x \rfloor \text{ is odd} \end{cases}$$

This  $f$  is not  $\mathcal{O}(g)$ , because for inputs where  $\lfloor x \rfloor$  is odd there is no constant  $C$  that gives  $C \cdot x^3 \geq x^4$  for all  $x$ . Likewise,  $g$  is not  $\mathcal{O}(f)$  because of  $f$ 's behavior when  $\lfloor x \rfloor$  is even.

- 1.12 LEMMA (ALGEBRAIC PROPERTIES) Let these be complexity functions.

- (A) If  $f$  is  $\mathcal{O}(g)$  then for any constant  $a \in \mathbb{R}^+$ , the function  $a \cdot f$  is  $\mathcal{O}(g)$ .
- (B) If  $f_0$  is  $\mathcal{O}(g_0)$  and  $f_1$  is  $\mathcal{O}(g_1)$  then the sum  $f_0 + f_1$  is  $\mathcal{O}(g)$  where  $g(x) = \max(g_0(x), g_1(x))$ . So if both  $f_0$  and  $f_1$  are  $\mathcal{O}(g)$  then  $f_0 + f_1$  is also  $\mathcal{O}(g)$ .
- (C) If  $f_0$  is  $\mathcal{O}(g_0)$  and  $f_1$  is  $\mathcal{O}(g_1)$  then the product  $f_0 f_1$  is  $\mathcal{O}(g_0 g_1)$ .

That result gives us two principles for simplifying Big  $\mathcal{O}$  expressions. First: if an expression is a sum of finitely many terms of which one has the largest growth rate, then we can drop the other terms. Second: if an expression is a product of factors then we can drop constants, factors that do not depend on the input.

- 1.13 EXAMPLE Consider  $f(n) = 5x^3 + 3x^2 + 12x$ . Looking to the first principle, the term with the largest growth rate is  $5x^3$  (this is intuitively clear and it will follow from Theorem 1.17 below) and applying the lemma's second item with  $g(x) = 5x^3$  gives that  $f$  is  $\mathcal{O}(5x^3)$ . Next, because one of  $5x^3$ 's factors is constant, the second simplification principle—applying the lemma's first item with  $a = 1/5$ —gives that  $f$  is  $\mathcal{O}(x^3)$ .

- 1.14 DEFINITION Complexity functions  $f$  and  $g$  have **equivalent growth rates**, or the **same order of growth**, if  $f$  is  $\mathcal{O}(g)$  and also  $g$  is  $\mathcal{O}(f)$ . We say that  $f$  is  $\Theta(g)$  (read ‘ $f$  is big-Theta of  $g$ ’), or, what is the same thing, that  $g$  is  $\Theta(f)$ .

- 1.15 LEMMA The Big- $\mathcal{O}$  relation is reflexive, so  $f$  is  $\mathcal{O}(f)$ . It is also transitive, so if  $f$  is  $\mathcal{O}(g)$  and  $g$  is  $\mathcal{O}(h)$  then  $f$  is  $\mathcal{O}(h)$ . Thus, having equivalent growth rates is an equivalence relation between functions.



1.16 FIGURE: Each bean contains the complexity functions. Faster growing functions are higher, so that  $x^5$  would be shown above  $x^4$ . 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. The sketch on the right adds the cone  $\mathcal{O}(f)$  for some  $f$  in  $\mathcal{O}(g)$ .

The next result makes calculations involving Big  $\mathcal{O}$  easier for most of the functions that we work with, such as polynomial and logarithmic functions.

- 1.17 THEOREM Let  $f, g$  be complexity functions. Suppose that  $\lim_{x \rightarrow \infty} f(x)/g(x)$  exists and equals  $L \in \mathbb{R} \cup \{\infty\}$ .
- (A) If  $L = 0$  then  $g$  grows faster than  $f$ , that is,  $f$  is  $\mathcal{O}(g)$  but  $g$  is not  $\mathcal{O}(f)$ .<sup>†</sup>
  - (B) If  $L = \infty$  then  $f$  grows faster than  $g$ , so  $g$  is  $\mathcal{O}(f)$  but  $f$  is not  $\mathcal{O}(g)$ .<sup>‡</sup>
  - (C) If  $L$  is between 0 and  $\infty$  then the two functions have equivalent growth rates, so that  $f$  is  $\Theta(g)$  and  $g$  is  $\Theta(f)$ .<sup>§</sup>

It pairs well with the following, from Calculus.

<sup>†</sup>This case is denoted  $f$  is  $o(g)$ . <sup>‡</sup>The ‘ $g$  is  $\mathcal{O}(f)$ ’ is denoted  $f$  is  $\Omega(g)$ . <sup>§</sup>If  $L = 1$  then  $f$  and  $g$  are asymptotically equivalent, denoted  $f \sim g$ .

- 1.18 **THEOREM (L'HÔPITAL'S RULE)** Let  $f$  and  $g$  be complexity functions such that both  $f(x) \rightarrow \infty$  and  $g(x) \rightarrow \infty$  as  $x \rightarrow \infty$ , and such that both are differentiable for large enough inputs. If  $\lim_{x \rightarrow \infty} f'(x)/g'(x)$  exists and equals  $L \in \mathbb{R} \cup \{\infty\}$  then  $\lim_{x \rightarrow \infty} f(x)/g(x)$  also exists and also equals  $L$ .

- 1.19 **EXAMPLE** Where  $f(x) = x^2 + 5x + 6$  and  $g(x) = x^3 + 2x + 3$ .

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{x \rightarrow \infty} \frac{x^2 + 5x + 6}{x^3 + 2x + 3} = \lim_{x \rightarrow \infty} \frac{2x + 5}{3x^2 + 2} = \lim_{x \rightarrow \infty} \frac{2}{6x} = 0$$

Then Theorem 1.17 says that  $f$  is  $\mathcal{O}(g)$  but  $g$  is not  $\mathcal{O}(f)$ . That is,  $f$ 's growth rate is less than  $g$ 's.

Next consider  $f(x) = 3x^2 + 4x + 5$  and  $g(x) = x^2$ .

$$\lim_{x \rightarrow \infty} \frac{3x^2 + 4x + 5}{x^2} = \lim_{x \rightarrow \infty} \frac{6x + 4}{2x} = \lim_{x \rightarrow \infty} \frac{6}{2} = 3$$

So the growth rates of the two are equivalent. That is,  $f$  is  $\Theta(g)$ .

For  $f(x) = 5x^4 + 15$  and  $g(x) = x^2 - 3x$ , this

$$\lim_{x \rightarrow \infty} \frac{5x^4 + 15}{x^2 - 3x} = \lim_{x \rightarrow \infty} \frac{20x^3}{2x - 3} = \lim_{x \rightarrow \infty} \frac{60x^2}{2} = \infty$$

shows that  $f$ 's growth rate is strictly greater than  $g$ 's rate— $g$  is  $\mathcal{O}(f)$  but  $f$  is not  $\mathcal{O}(g)$ .

- 1.20 **EXAMPLE** The logarithmic function  $f(x) = \log_b(x)$  grows very slowly:  $\log_b(x)$  is  $\mathcal{O}(x)$ , and  $\log_b(x)$  is  $\mathcal{O}(x^{0.1})$ , and is  $\mathcal{O}(x^{0.01})$ , and in fact  $\log_b(x)$  is  $\mathcal{O}(x^d)$  for any  $d > 0$ , no matter how small, by this equation.

$$\lim_{x \rightarrow \infty} \frac{\log_b(x)}{x^d} = \lim_{x \rightarrow \infty} \frac{(1/x \ln(b))}{dx^{d-1}} = \frac{1}{d \ln(b)} \cdot \lim_{x \rightarrow \infty} \frac{1}{x^d} = 0$$

By Theorem 1.17 that calculation also shows that  $x^d$  is not  $\mathcal{O}(\log_b(x))$ .

The difference in growth rates is even stronger than that. L'Hôpital's Rule, along with the Chain Rule, gives that  $(\log_b(x))^2$  is  $\mathcal{O}(x)$  because this is 0.

$$\lim_{x \rightarrow \infty} \frac{(\log_b(x))^2}{x} = \lim_{x \rightarrow \infty} \frac{2 \ln(x)/(x \ln(b))}{1} = \frac{2}{\ln(b)} \cdot \lim_{x \rightarrow \infty} \frac{\ln(x)}{x} = \frac{2}{\ln(b)} \cdot \lim_{x \rightarrow \infty} \frac{1/x}{1}$$

Further, Exercise 1.47 shows that for every power  $k$  the function  $(\log_b(x))^k$  is  $\mathcal{O}(x^d)$  for any positive  $d$ , no matter how small.

The log-linear function  $x \cdot \lg(x)$  has a similar relationship to the polynomials

$x^d$ , where  $d > 1$ .

$$\begin{aligned}\lim_{x \rightarrow \infty} \frac{x \lg(x)}{x^d} &= \lim_{x \rightarrow \infty} \frac{x \cdot (1/x \ln(2)) + 1 \cdot \ln(x)}{dx^{d-1}} = \frac{1}{d} \cdot \lim_{x \rightarrow \infty} \frac{(1/\ln(2)) + \ln(x)}{x^{d-1}} \\ &= \frac{1}{d(d-1)} \cdot \lim_{x \rightarrow \infty} \frac{(1/x)}{x^{d-2}} = \frac{1}{d(d-1)} \cdot \lim_{x \rightarrow \infty} \frac{1}{x^{d-1}} = 0\end{aligned}$$

- 1.21 EXAMPLE We can compare the polynomial  $f(x) = x^2$  to the exponential  $g(x) = 2^x$ .

$$\lim_{x \rightarrow \infty} \frac{2^x}{x^2} = \lim_{x \rightarrow \infty} \frac{2^x \cdot \ln(2)}{2x} = \lim_{x \rightarrow \infty} \frac{2^x \cdot (\ln(2))^2}{2} = \infty$$

Thus,  $f$  is in  $\mathcal{O}(g)$  but  $g$  is not in  $\mathcal{O}(f)$ .

An easy induction argument gives that

$$\lim_{x \rightarrow \infty} \frac{2^x}{x^k} = \infty$$

for any  $k$ , and so  $x^k$  is in  $\mathcal{O}(2^x)$  but  $2^x$  is not in  $\mathcal{O}(x^k)$ .

- 1.22 LEMMA Logarithmic functions grow more slowly than polynomial functions: if  $f(x) = \log_b(x)$  for some base  $b$  and  $g(x) = a_m x^m + \dots + a_0$  then  $f$  is  $\mathcal{O}(g)$  but  $g$  is not  $\mathcal{O}(f)$ . Polynomial functions grow more slowly than exponential functions: where  $h(x) = b^x$  for some base  $b > 1$  then  $g$  is  $\mathcal{O}(h)$  but  $h$  is not  $\mathcal{O}(g)$ .

As we mentioned earlier, for thinking about the resources used by mechanical computations, the most natural functions map  $\mathbb{N}$  to  $\mathbb{N}$ . But we've defined complexity functions as mapping  $\mathbb{R}$  to  $\mathbb{R}$ . One motivation is that some functions that we want to work with, such as logarithms, are real functions. Another is that L'Hôpital's Rule, which uses the derivative and so needs reals, is a big convenience. The next result assures that our conclusions in the continuous context carry over to the discrete. (This lemma does not cover the detail of cases where the functions are only defined for inputs larger than some value  $N$  but this version is easier to state and makes the same point.)

- 1.23 LEMMA Let  $f_0, f_1 : \mathbb{R} \rightarrow \mathbb{R}$ , and consider the restrictions to a discrete domain  $g_0 = f_0|_{\mathbb{N}}$  and  $g_1 = f_1|_{\mathbb{N}}$ . Where  $L \in \mathbb{R} \cup \{\infty\}$ ,
- (A) for  $a \in \mathbb{R}$ , if  $L = \lim_{x \rightarrow \infty} (af_0)(x)$  then  $L = \lim_{n \rightarrow \infty} (ag_0)(n)$ ,
  - (B) if  $L = \lim_{x \rightarrow \infty} (f_0 + f_1)(x)$  then  $L = \lim_{n \rightarrow \infty} (g_0 + g_1)(n)$ ,
  - (C) if  $L = \lim_{x \rightarrow \infty} (f_0 \cdot f_1)(x)$  then  $L = \lim_{n \rightarrow \infty} (g_0 \cdot g_1)(n)$ , and
  - (D) when the expressions are defined, if  $L = \lim_{x \rightarrow \infty} (f_0/f_1)(x)$  then  $L = \lim_{n \rightarrow \infty} (g_0/g_1)(n)$ .

**Tractable and intractable** This table lists orders of growth that appear most often in practice. They are listed with faster-growing functions further down the table.

Order	Name	Examples
$\mathcal{O}(1)$	Bounded	$f_0(n) = 1, f_1(n) = 15$
$\mathcal{O}(\lg(\lg(n)))$	Double logarithmic	$f(n) = \ln(\ln(x))$
$\mathcal{O}(\lg(n))$	Logarithmic	$f_0(n) = \ln(n), f_1(n) = \lg(n^3)$
$\mathcal{O}((\lg(n))^c)$	Polylogarithmic	$f(n) = (\lg(n))^3$
$\mathcal{O}(n)$	Linear	$f_0(n) = n, f_1(n) = 3n + 4$
$\mathcal{O}(n \lg(n)) = \mathcal{O}(\lg(n!))$	Log-linear	$f(n) = 5n \ln(n) + n$
$\mathcal{O}(n^2)$	Polynomial (quadratic)	$f_0(n) = 5n^2 + 2n + 12$
$\mathcal{O}(n^3)$	Polynomial (cubic)	$f(n) = 2n^3 + 12n^2 + 5$
$\vdots$		
$\mathcal{O}(2^n)$	Exponential	$f(n) = 10 \cdot 2^n$
$\mathcal{O}(3^n)$	Exponential	$f(n) = 6 \cdot 3^n + n^2$
$\vdots$		
$\mathcal{O}(n!)$	Factorial	
$\mathcal{O}(n^n)$	-No standard name-	

1.24 TABLE: The Hardy hierarchy.

We often split that hierarchy between the polynomial and exponential functions; the table below shows why. It lists how long a job will take if we use an algorithm that runs in time  $\lg n$ , time  $n$ , etc. (A modern computer runs at 10 GHz, 10 000 million ticks per second, and there are  $3.16 \times 10^7$  seconds in a year.)

	$n = 1$	$n = 10$	$n = 50$	$n = 100$
$\lg n$	–	$1.05 \times 10^{-17}$	$1.79 \times 10^{-17}$	$2.11 \times 10^{-17}$
$n$	$3.17 \times 10^{-18}$	$3.17 \times 10^{-17}$	$1.58 \times 10^{-16}$	$3.17 \times 10^{-16}$
$n \lg n$	–	$1.05 \times 10^{-16}$	$8.94 \times 10^{-16}$	$2.11 \times 10^{-15}$
$n^2$	$3.17 \times 10^{-18}$	$3.17 \times 10^{-16}$	$7.92 \times 10^{-15}$	$3.17 \times 10^{-14}$
$n^3$	$3.17 \times 10^{-18}$	$3.17 \times 10^{-15}$	$3.96 \times 10^{-13}$	$3.17 \times 10^{-12}$
$2^n$	$6.34 \times 10^{-18}$	$3.24 \times 10^{-15}$	$3.57 \times 10^{-3}$	$4.02 \times 10^{12}$

1.25 TABLE: Time taken, in years, by algorithms whose behavior is given by some functions from the order of growth hierarchy, on some input sizes  $n$ .

Consider the final column,  $n = 100$ . Between the initial rows the relative change is an order of magnitude, which is a lot, but the absolute times are small. Then we get to the final row. That's not a typo—the last entry really is  $10^{12}$  years. This is a huge change, both relatively and absolutely. The universe is  $14 \times 10^9$  years old. So this computation, even with input size of only 100, would take longer than the age of the universe. Exponential growth is very, very much larger than polynomial growth.

Another way to understand this point is to consider the effect of adding one more bit to an algorithm's input, such as by passing from the length ten  $\sigma_0 = 1101001010$  to the length eleven  $\sigma_1 = 11010010101$ . An algorithm that

loops through the bits will just do one more loop, so it takes ten percent more time. But an algorithm that takes  $2^{|\sigma|}$  time will take double the time.

**Cobham's thesis** is that the **tractable** problems—the ones that are at least conceivably solvable in practice—are those for which there is an algorithm whose resource consumption is polynomial.<sup>†</sup> For instance, if a problem's best available algorithm runs in exponential time then we may say that the problem is, or at least appears, **intractable**.

**Discussion** Big  $\mathcal{O}$  is about relative scalability: an algorithm whose runtime behavior is  $\mathcal{O}(n^2)$  scales worse than one whose behavior is  $\mathcal{O}(n \lg n)$ , but better than one whose behavior is  $\mathcal{O}(n^3)$ . Is there more to say?

True, that is the essence. Nonetheless, experience shows that there are points about Big  $\mathcal{O}$  that can puzzle learners. Here we will elaborate on those.

The first is that Big  $\mathcal{O}$  is for algorithms, not programs. Contrast these two Racket functions.

```
(define (g0 n)
  (for ([i '(0 1 2 3 4)])
    (let ([x (* n n)])
      (display (+ i x)) (newline))))
```

```
(define (g1 n)
  (let ([x (* n n)])
    (for ([i '(0 1 2 3 4)])
      (display (+ i x)) (newline))))
```

On the left  $g0$  sets the local variable  $x$  inside the loop. That makes it slower than the right by four assignments. But Big  $\mathcal{O}$  disregards this constant time difference. That is, Big  $\mathcal{O}$  is not the right tool for characterizing fine coding details. Big  $\mathcal{O}$  works at a higher level, such as for comparing running times among algorithms.

That fits with our second point about Big  $\mathcal{O}$ . We use it to help pick the best algorithm, to rank them according to how much they use of some computing resources. But algorithms are tied to an underlying computing model.<sup>‡</sup> So for the comparison we need a definition of the time used on a particular machine model.

- 1.26 **DEFINITION** A machine  $\mathcal{M}$  with input alphabet  $\Sigma$  **takes time**  $t_{\mathcal{M}}: \Sigma^* \rightarrow \mathbb{N} \cup \{\infty\}$  if that function gives the number of steps that the machine takes to halt on input  $\sigma \in \Sigma^*$ . If  $\mathcal{M}$  does not halt then  $t_{\mathcal{M}}(\sigma) = \infty$ . The machine **runs in input length time**  $\hat{t}_{\mathcal{M}}: \mathbb{N} \rightarrow \mathbb{N} \cup \{\infty\}$  if  $\hat{t}_{\mathcal{M}}(n)$  is the maximum of the  $t(\sigma)$  over all inputs  $\sigma \in \Sigma^*$  of length  $n$ . The machine **runs in time**  $\mathcal{O}(f)$  if  $\hat{t}_{\mathcal{M}}$  is  $\mathcal{O}(f)$ .

Besides the Turing machine, another model that is widely used in the Theory of Computing, is the **Random Access machine (RAM)**. Whereas a Turing machine cell stores only a single symbol, so that to store an integer you may need multiple cells, a RAM model machine has registers that each store an entire integer. And

<sup>†</sup> Cobham's Thesis is widely accepted, but not universally accepted. Some researchers object that if an algorithm runs in time  $n^{100}$  or if it runs in time  $Cn^2$  but with an enormous  $C$  then the solution is not actually practical. A rejoinder to that objection notes that when someone announces an algorithm with a large exponent or large constant then typically over time the approach gets refined, shrinking those two. In any event, polynomial time is markedly better than exponential time. In this book we accept the thesis because it gives technical meaning to the informal 'acceptably fast'. <sup>‡</sup> More discussion of the relationship between algorithms and machine models is in Section 3.

whereas to get to a cell a Turing machine may spend a lot of steps moving through the tape, the RAM model gets to each register contents in one step.

Close analysis shows that if we start with an algorithm intended for a RAM model machine and execute it on a Turing machine then this may add as much as  $n^3$  extra ticks to the runtime, so that if the algorithm is  $\mathcal{O}(n^2)$  on the RAM then on the Turing machine it can be  $\mathcal{O}(n^5)$ .<sup>†</sup> Thus, to understand the cost of an algorithm, we must first settle on a model, and only then discuss the Big  $\mathcal{O}$ .

We have already brought up our third issue about Big  $\mathcal{O}$ , in relation to Cobham's Thesis. The definition of Big  $\mathcal{O}$  ignores constant factors; does that greatly reduce its value for comparing algorithms? More precisely, if for inputs  $n > N$ , our algorithm takes time given by  $C \cdot n^2$  then don't we need to know  $C$  and  $N$ ? After all, if one algorithm has runtime  $C_0 n$  for an enormous  $C_0$  while another is  $C_1 n^2$  for tiny  $C_1$ , could that not make the second algorithm more useful? Similarly, could a huge  $N$  mean that we need to describe what happens before that point?

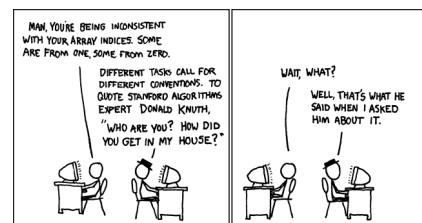


Donald Knuth  
b 1938

Part of the answer is that finding these constants is hard.<sup>‡</sup> Machines vary widely in their low-level details such as the memory addressing and paging, cache hits, and whether the CPU can do some operations in parallel, and these details can make a tremendous difference in constants such as  $C$  and  $N$ . Imagine doing the analysis on a commercially available machine and then the vendor releases a new model, so it is all to do again. That would be discouraging. And what's more, experience shows that doing the work to find the exact numbers usually does not change the algorithm that gets picked. As Table 1.25 illustrates, knowing at a Big  $\mathcal{O}$  level how the algorithm grows is much more influential than knowing the exact constant values.

Instead of analyzing commercial machines, we could agree on specifications for a reference architecture — this is the approach taken by D Knuth in the monumental *Art of Computer Programming* series — but again there is the risk that we might have to update that standard. Then, published results from some time ago may no longer apply, because they refer to an old standard. Again, discouraging. So the analysis that we do to find the Big  $\mathcal{O}$  behavior of an algorithm usually refers to an abstract computer model, such as a Turing machine or a RAM model, and it usually does not go to the extent of finding the constants. That is, being reference independent is, in a quickly changing field, an advantage.

This ties back to the paragraph at the start of this discussion. Not taking into account the precise difference between, say, the cost of a division and the cost of a



Courtesy [xkcd.com](http://xkcd.com)

<sup>†</sup>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 takes constant time,  $\mathcal{O}(1)$ . <sup>‡</sup>People do sometimes note the order of magnitude of these constants.

memory write (as long as those costs lie between reasonable limits) implies that constant factors are meaningless, and we focus on relative comparisons. Here is an analogy: absolute measurement of distance involves units such as miles or kilometers, but being able to make statements irrespective of the unit constants requires making relative statements such as, “from here, New York City is twice as far as Boston.”

That is, the entire answer is that we say that an algorithm that on input of size  $n$  will take  $3n$  ticks is  $\mathcal{O}(n)$  in order to express that, roughly, doubling the input size will no more than double the number of steps taken. Similarly, if an algorithm is  $\mathcal{O}(n^2)$  then doubling the input size will at most quadruple the number of steps. The Big O notation ignores constants because that is inherent in being a unit free, relative measurement.

However, for a person with the sense that leaving out the constants makes this measure approximate, then certainly, Big O is only a rough comparison. It cannot say with precision which of two algorithms will be absolutely better when they are cast into code and run on a particular platform, for input sizes in a given range.<sup>†</sup>

This leads to our fourth point about Big O. Understanding how an algorithm performs as the input size grows requires that we define the input size.

Consider an algorithm for factoring numbers that inputs a natural number  $n$  and tests each  $k \in \{2, \dots, n-1\}$  to see if it divides  $n$ . If  $n$  is prime then it tests all of those  $k$ 's, which is roughly  $n$ -many divisions. We can take the size of  $n$  to be the number of bits needed to represent  $n$  in binary, approximately  $\lg n$ . That is, for this algorithm the input is of size  $\lg n$  and the number of operations is about  $n$ . That's exponential growth—passing from  $\lg n$  to  $n$  requires exponentiating—so this algorithm is  $\mathcal{O}(2^b)$ , where  $b$  is the number of input bits.

However, in a programming class this algorithm would likely be described as linear, as  $\mathcal{O}(n)$ , with the reasoning that for the input  $n$  there are about  $n$ -many divisions. How to explain the difference between these two Big O estimates?

This is another example of the relationship between an algorithm and an underlying computing model. A programmer may make the engineering judgment that for every use of their program the input will fit into a 64 bit word. They are selecting a computation model, like the RAM model, where larger numbers take the same time to read as smaller numbers. With this model, the prior paragraph applies and the algorithm is linear.

So this difference in Big O estimates is in part an application versus theory thing. In the common programming application setting, where the bit size of the



Courtesy [xkcd.com](http://xkcd.com)

<sup>†</sup> For that, use benchmarks.

inputs is bounded, the runtime behavior is  $\mathcal{O}(n)$ . In a theoretical setting, accepting input that is arbitrarily large and so the runtime is a function of the bit size of the inputs, the algorithm is  $\mathcal{O}(2^b)$ . An algorithm whose behavior as a function of the input is polynomial, but whose behavior as a function of the bit size of the input is exponential, is said to be **pseudopolynomial**.

A fifth and final point about Big  $\mathcal{O}$ . When we are analyzing an algorithm we can consider the behavior that is the worst case for any input of that size (as in Definition 1.26), or the behavior that is the average over all inputs of that size. For instance, the quicksort algorithm takes quadratic time  $\mathcal{O}(n^2)$  at worst, but on average is  $\mathcal{O}(n \lg n)$ . Note, though, that worst-case analysis is the most common.

## V.1 Exercises

- 1.27 True or false: if a function is  $\mathcal{O}(n^2)$  then it is  $\mathcal{O}(n^3)$ .
- ✓ 1.28 Your classmate emails you a draft of an assignment answer that says, “I have an algorithm with running time that is  $\mathcal{O}(n^2)$ . So with input  $n = 5$  it will take 25 ticks.” Make two corrections.
- 1.29 Suppose that someone posts to a group that you are in, “I’m working on a problem that is  $\mathcal{O}(n^3)$ .” Explain to them, gently, how their sentence is mistaken.
- ✓ 1.30 How many bits does it take to express each number in binary? (A) 5 (B) 50  
(C) 500 (D) 5 000
- ✓ 1.31 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  $\mathcal{O}(g)$ .
- ✓ 1.32 For each, find the function on the Hardy hierarchy, Table 1.24, that has the same rate of growth. (A)  $n^2 + 5n - 2$  (B)  $2^n + n^3$  (C)  $3n^4 - \lg \lg n$   
(D)  $\lg n + 5$
- 1.33 For each, give the function on the Hardy hierarchy, Table 1.24, that has the same rate of growth. That is, find  $g$  in that table where  $f$  is  $\Theta(g)$ .
- (A)  $f(n) = \begin{cases} n & \text{if } n < 100 \\ 0 & \text{else} \end{cases}$
- (B)  $f(n) = \begin{cases} 1\,000\,000 \cdot n & \text{if } n < 10\,000 \\ n^2 & \text{else} \end{cases}$
- (C)  $f(n) = \begin{cases} 1\,000\,000 \cdot n^2 & \text{if } n < 100\,000 \\ \lg n & \text{else} \end{cases}$
- ✓ 1.34 For each pair, find the limit of the ratio  $f/g$  to decide of  $f$  is  $\mathcal{O}(g)$ , or  $g$  is  $\mathcal{O}(f)$ , or both, or neither. (A)  $f(n) = 3n^3 + 2n + 4$ ,  $g(n) = \ln(n) + 6$   
(B)  $f(n) = 3n^3 + 2n + 4$ ,  $g(n) = n + 5n^3$  (C)  $f(n) = (1/2)n^3 + 12n^2$ ,  $g(n) = n^2 \ln(n)$   
(D)  $f(n) = \lg(n) = \log_2(n)$ ,  $g(n) = \ln(n)$  (E)  $f(n) = n^2 + \lg(n)$ ,  $g(n) = n^4 - n^3$   
(F)  $f(n) = 55$ ,  $g(n) = n^2 + n$

- ✓ 1.35 For each pair of functions simplify using Lemma 1.12 to decide if  $f$  is  $\mathcal{O}(g)$ , or  $g$  is  $\mathcal{O}(f)$ , or both, or neither. (A)  $f(n) = 4n^2 + 3$ ,  $g(n) = (1/2)n^2 - n$  (B)  $f(n) = 53n^3$ ,  $g(n) = \ln n$  (C)  $f(n) = 2n^2$ ,  $g(n) = \sqrt{n}$  (D)  $f(n) = n^{1.2} + \lg n$ ,  $g(n) = n^{\sqrt{2}} + 2n$  (E)  $f(n) = n^6$ ,  $g(n) = 2^{n/6}$  (F)  $f(n) = 3^n$ ,  $g(n) = 3 \cdot 2^n$  (G)  $f(n) = \lg(3n)$ ,  $g(n) = \lg(n)$
- 1.36 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)  $\lg(5^n)$
- ✓ 1.37 For each, state true or false. (A)  $5n^2 + 2n$  is  $\mathcal{O}(n^3)$  (B)  $2 + 4n^3$  is  $\mathcal{O}(\lg n)$  (C)  $\ln n$  is  $\mathcal{O}(\lg n)$  (D)  $n^3 + n^2 + n$  is  $\mathcal{O}(n^3)$  (E)  $n^3 + n^2 + n$  is  $\mathcal{O}(2^n)$
- 1.38 For each find the smallest  $k \in \mathbb{N}$  so that the given function is  $\mathcal{O}(n^k)$ . (A)  $n^3 + (n^4/10\,000\,000)$  (B)  $(n+2)(n+3)(n^2 - \lg n)$  (C)  $5n^3 + 25 + \lceil \cos(n) \rceil$  (D)  $9 \cdot (n^2 + n^3)^4$  (E)  $\lfloor \sqrt[3]{5n^7 + 2n^2} \rfloor$
- 1.39 Consider Table 1.25. (A) Add a column for  $n = 200$ . (B) Add a row for  $3^n$ .
- ✓ 1.40 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)  $2^n$
- 1.41 Sometimes in practice we must choose between two algorithms where the performance of one is better than the performance of the other in a big- $\mathcal{O}$  sense, but where the first has a long initial segment of poorer performance. What is the least input number such that  $f(n) = 100\,000 \cdot n^2$  is less than  $g(n) = n^3$ ?
- 1.42 What is the order of growth of the run time of a deterministic Finite State machine?
- ✓ 1.43 (A) Verify that the function  $f(x) = 7$  is  $\mathcal{O}(1)$ . (B) Verify that  $f(x) = 7 + \sin(x)$  is  $\mathcal{O}(1)$ . So if a function is in  $\mathcal{O}(1)$ , that does not mean that it is a constant function. (C) Verify that  $f(x) = 7 + (1/x)$  is also  $\mathcal{O}(1)$ . (D) Show that a complexity function  $f$  is  $\mathcal{O}(1)$  if and only if it is bounded above by a constant, that is, if and only if there exists  $L \in \mathbb{R}$  so that  $f(x) \leq L$  for all inputs  $x \in \mathbb{R}$ .
- 1.44 Where does  $g(x) \leq x^{\mathcal{O}(1)}$  place the function  $g$  in the Hardy hierarchy?  
*Hint:* see the prior question.
- 1.45 Let  $f(x) = 2x$  and  $g(x) = x^2$ . Prove directly from Definition 1.6 that  $f$  is  $\mathcal{O}(g)$ , but that  $g$  is not  $\mathcal{O}(f)$ .
- 1.46 Prove that  $2^n$  is  $\mathcal{O}(n!)$ . *Hint:* because of the factorial, consider these natural number functions and find suitable  $N, C \in \mathbb{N}$ .
- 1.47 Use L'Hôpital's Rule as in Example 1.20 to verify these for any  $d \in \mathbb{R}^+$ : (A)  $(\log_b(x))^3$  is  $\mathcal{O}(x^d)$  (B) for any  $k \in \mathbb{N}^+$ ,  $(\log_b(x))^k$  is  $\mathcal{O}(x^d)$ .
- 1.48 Assume that  $g: \mathbb{R} \rightarrow \mathbb{R}$  is increasing, so that  $x_1 \geq x_0$  implies that  $g(x_1) \geq g(x_0)$ . Let  $f: \mathbb{R} \rightarrow \mathbb{R}$  be a constant function. Show that  $f$  is  $\mathcal{O}(g)$ .
- 1.49 (A) Show that there is a computable function whose output values grow at a rate that is  $\mathcal{O}(1)$ , one whose values grow at a rate that is  $\mathcal{O}(n)$ , one for  $\mathcal{O}(n^2)$ , etc.

(B) The Halting problem function  $K$  is uncomputable. Place its rate of growth in the Hardy hierarchy, Table 1.24. (c) Produce a function that is not computable because its output values are larger than those of any computable function. (You need not show that the rate of growth is greater, only that the outputs are larger.)

1.50 Show that the naive algorithm to test if the input is prime, which just checks whether it is divisible by any number  $m \geq 2$  such that  $m < n$ , is pseudopolynomial. (*Hint:* we can check whether one number divides another in quadratic time.)

✓ 1.51 Show that  $\mathcal{O}(2^x) \in \mathcal{O}(3^x)$  but  $\mathcal{O}(2^x) \neq \mathcal{O}(3^x)$ .

1.52 Table 1.24 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  $\Theta(n^n)$ ?

✓ 1.53 Two complexity functions  $f, g$  are **asymptotically equivalent**,  $f \sim g$ , if  $\lim_{x \rightarrow \infty} (f(x)/g(x)) = 1$ . Show that each pair is asymptotically equivalent:

(A)  $f(x) = x^2 + 5x + 1$  and  $g(x) = x^2$ ,

(B)  $\lg(x+1)$  and  $\lg(x)$ .

1.54 Is there an  $f$  so that  $\mathcal{O}(f)$  is the set of all polynomials?

1.55 There are orders of growth between polynomial and exponential. Specifically,  $f(x) = x^{\lg x}$  is one. (A) Show that  $\lg(x) \in \mathcal{O}((\lg(x))^2)$  but  $(\lg(x))^2 \notin \mathcal{O}(\lg(x))$ . (B) Argue that for any 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  $a = 2^{\lg(a)}$ , and consider the limit of the exponent. (C) Show that  $x^{\lg x} = 2^{(\lg x)^2}$ . *Hint:* take the logarithm of both halves. (D) Show that  $x^{\lg x}$  is in  $\mathcal{O}(2^x)$ . *Hint:* form the ratio using the prior item.

1.56 Verify the clauses of Lemma 1.12.

(A) If  $a \in \mathbb{R}^+$  then  $af$  is also  $\mathcal{O}(g)$ .

(B) The function  $f_0 + f_1$  is  $\mathcal{O}(g)$ , where  $g$  is defined by  $g(n) = \max(g_0(n), g_1(n))$ .

(C) The product  $f_0 f_1$  is  $\mathcal{O}(g_0 g_1)$ .

1.57 Verify these clauses of Lemma 1.15. (A) The big- $\mathcal{O}$  relation is reflexive.

(B) It is also transitive.

1.58 Theorem 1.17 says that if the limit of the ratio of two functions exists then we can determine the  $\mathcal{O}$  relationship between the two. Assume that  $f$  and  $g$  are complexity functions.

(A) Suppose that  $\lim_{x \rightarrow \infty} f(x)/g(x)$  exists and equals 0. Show that  $f$  is  $\mathcal{O}(g)$ .

*(Hint:* this requires a rigorous definition of the limit.)

(B) We can give an example where  $f$  is  $\mathcal{O}(g)$  even though  $\lim_{x \rightarrow \infty} f(x)/g(x)$  does not exist. Verify that, where  $g(x) = x$  and where  $f(x) = x$  when  $\lfloor x \rfloor$  is odd and  $f(x) = 2x$  when  $\lfloor x \rfloor$  is even.

1.59 Prove Lemma 1.22.

**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. These are all well known. They are part of the culture of the field.

**Problems that come with stories** We start with a few problems that come with stories. Besides being fun, and an important part of the field's culture, these also give a sense of where problems come from.

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 ground work for the development of quantum mechanics. In mathematics, he is best known as the inventor of the quaternion number system.



William Rowan  
Hamilton  
1805–1865

One of his ventures was a game, *Around the World*. The vertices in the graph below were holes labeled with the names of world cities. Players put pegs in the holes, looking for a circuit that visits each city once and only once.

**2.1 ANIMATION: Hamilton's *Around the World* game**

It did not make Hamilton rich. But it did get him associated with a great problem.

**2.2 PROBLEM (Hamiltonian Circuit)** Given a graph, decide if it contains a **Hamiltonian circuit**, a cyclic path that includes each vertex once and only once.

This is stated as a type of problem called a decision problem, because it asks for a 'yes' or 'no' answer. The next section will say more about problem types.

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 the starting direction.)

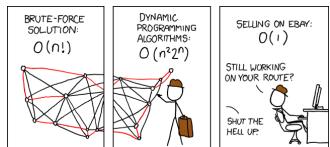


42	59	44	9	40	21	46	7
61	10	41	58	45	8	39	20
12	45	60	55	22	57	6	47
53	62	11	30	25	28	19	38
32	13	54	27	56	23	48	5
63	52	31	24	29	26	37	18
14	33	2	51	16	35	44	9
1	64	15	34	3	50	17	36

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 graph, where we call the vertices  $S = \{c_0, \dots c_{k-1}\}$  ‘cities’ and we call the edge weight  $d(c_i, c_j) \in \mathbb{N}^+$  for all  $c_i \neq c_j$  the ‘distance’ between the cities, find the shortest-distance circuit that visits every city and returns back to the start.



Courtesy xkcd.com

We can start with a map of the state capitals of the forty eight contiguous US states and the distances between them: Montpelier VT to Albany NY is 254 kilometers, etc. From among all trips that visit each city and return back to the start, such as Montpelier → Albany → Harrisburg → ... → Montpelier, we want the shortest one.

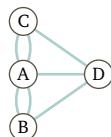
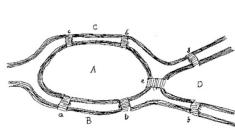
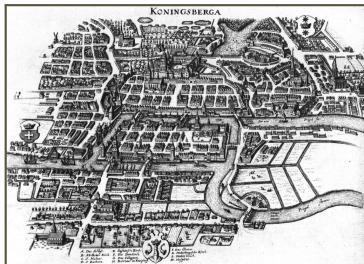
As stated, this is an optimization problem. However, we can recast it as a decision problem. Introduce a bound  $B \in \mathbb{N}$  and change the problem statement to ‘decide if there is a circuit of total distance less than  $B$ ’. If we had an algorithm to quickly solve this decision problem then we could also quickly solve the optimization problem: ask whether there is a trip bounded by length  $B = 1$ , then ask if there is a trip of length  $B = 2$ , etc. When we eventually get a ‘yes’, we know the length of the shortest trip.

The next problem sounds much like Hamiltonian Circuit, in that it involves exhaustively traversing a graph. But it proves to act very differently.

Today the city of Kaliningrad is in 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. Among these citizens the question arose: can a person cross each bridge once and only once, and arrive back at the start? No one could think of a way but no one could think of a reason that there was no way. A local mayor wrote to Euler, who proved that no circuit is possible. This paper founded Graph Theory.



Leonhard Euler  
1707–1783



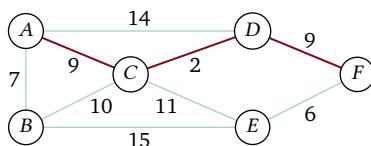
Euler's summary sketch is in the middle and the graph is on the right.

- 2.4 **PROBLEM (Euler Circuit)** Given a graph, find a circuit that traverses each edge once and only once, or declare 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 a phone for the shortest driving directions to someplace.

- 2.5 **PROBLEM (Shortest Path)** Given a weighted graph and two vertices, find the shortest path between them, or report that no path exists.

There is an algorithm that solves this problem quickly.<sup>†</sup> For instance, with this graph we could look for the cheapest path from  $A$  to  $F$ .



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.

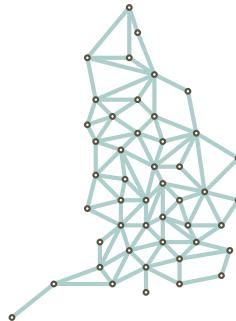
Guthrie imposed the condition that the countries must be contiguous and he defined 'sharing a border' to mean sharing an interval, not just a point (see Exercise 2.48). Below is a map and a graph version of the same problem. In the graph, counties are vertices and edges connect ones that are adjacent. A crucial point is that the graph is **planar**— we can draw it in the plane so that its edges do not cross.



Augustus De  
Morgan  
1806–1871

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. It is a special case of the **Graph Colorability** problem, given next.

<sup>†</sup>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

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. It remained unsolved until 1976, when K Appel and W Haken reduced the proof to 1936 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 the things studied, and not just be satisfied when a computer program that perhaps seems to be bug-free assures us that theorems are verified.<sup>†</sup> In the interim, though, a new generation has appeared that is more comfortable with the process and now computer proofs are routine, or at least not as controversial.

- 2.7 **PROBLEM (Graph Colorability)** Given a graph and a number  $k \in \mathbb{N}$ , decide whether the graph is  **$k$ -colorable**, whether we can partition its vertices into  $k$ -many sets,  $\mathcal{N} = \mathcal{C}_0 \cup \dots \cup \mathcal{C}_{k-1}$ , such that no two same-set vertices are connected.
- 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 in *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*.

A variable is Boolean if it takes only the values  $T$  or  $F$ . We focus on Boolean expressions that connect variables using the and operator,  $\wedge$ , the or operator,  $\vee$ , and the not operator,  $\neg$ . (For more, see Appendix C.) This Boolean function is given by an expression with three variables.

$$f(P, Q, R) = (P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (\neg P \vee \neg Q \vee \neg R)$$

An expression is **satisfiable** if some combination of input  $T$ 's and  $F$ 's makes it evaluate to  $T$ . It is in **Conjunctive Normal form** if it consists of clauses of variables



George  
Boole  
1815–1864

<sup>†</sup>This is in contrast to the *Entscheidungsproblem*.

or negations connected by  $\vee$ 's, where the clauses are connected with  $\wedge$ 's. This **truth table** shows the input-output behavior of the function defined by the expression.

$P$	$Q$	$R$	$P \vee Q$	$P \vee \neg Q$	$\neg P \vee Q$	$\neg P \vee \neg Q \vee \neg R$	$f(P, Q, R)$
$F$	$F$	$F$	$F$	$T$	$T$	$T$	$F$
$F$	$F$	$T$	$F$	$T$	$T$	$T$	$F$
$F$	$T$	$F$	$T$	$F$	$T$	$T$	$F$
$F$	$T$	$T$	$T$	$F$	$T$	$T$	$F$
$T$	$F$	$F$	$T$	$T$	$F$	$T$	$F$
$T$	$F$	$T$	$T$	$T$	$F$	$T$	$F$
$T$	$T$	$F$	$T$	$T$	$T$	$T$	$T$
$T$	$T$	$T$	$T$	$T$	$T$	$F$	$F$

That  $T$  in the final column witnesses that this formula is satisfiable.

2.9 **PROBLEM (Satisfiability, SAT)** Decide if a given Boolean expression is satisfiable.

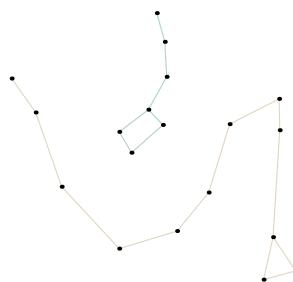
2.10 **PROBLEM (3-Satisfiability, 3-SAT)** Given a propositional logic formula in Conjunctive Normal form in which each clause has at most three variables, decide if it is satisfiable.

Observe that if the number of input variables is  $v$  then the number of rows in the truth table is  $2^v$ . So solving *SAT* appears to require exponential time. Whether that is right is a very important question, as we will see in later sections.

**More problems, omitting the stories** We will list more examples. All of these are widely known, part of the culture of the field. A speaker in this field would assume they the audience knew all of these.

2.11 **PROBLEM (Vertex-to-Vertex Path)** Given a graph and two vertices, find if the second is reachable from the first, that is, if there is a path to the second from the first.<sup>†</sup>

2.12 **EXAMPLE** These are two Western-tradition constellations, Ursa Minor and Draco.



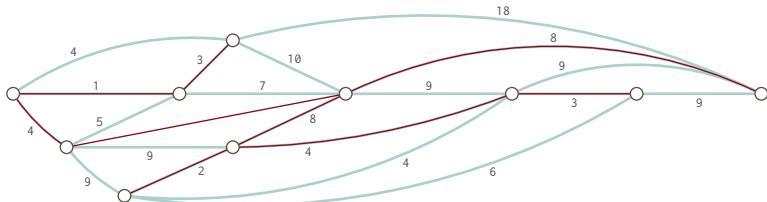
Here we can solve the **Vertex-to-Vertex Path** problem by eye. For any two vertices

<sup>†</sup>There are lots of problems about paths so just calling this the Path problem is confusing. But this name is nonstandard. It is known as *st*-Path, *st*-Connectivity, or STCON.

in Ursa Minor there is a path and for any two vertices in Draco there is a path. But if the two are in different constellations then there is no path.

- 2.13 **PROBLEM (Minimum Spanning Tree)** Given a weighted undirected graph, find a **minimum spanning tree**, 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.

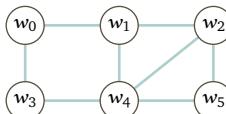


The highlighted subgraph includes all of the vertices, that is, it **spans** the graph. In addition, its weights total to a minimum from among all of the spanning subgraphs. From that it follows that this subgraph is a **tree**, meaning that it has no cycles, or else we could eliminate an edge from the cycle and thereby lower the edge weight total without dropping any vertices.

This problem looks like the **Hamiltonian Circuit** problem in requiring that the subgraph contain all the vertices. One difference is that for the **Minimum Spanning Tree** problem we know algorithms that are quick, that are  $\mathcal{O}(n \lg n)$ .

- 2.14 **PROBLEM (Vertex Cover)** Given a graph and a bound  $B \in \mathbb{N}$ , decide if the graph has a size  $B$  **vertex cover**, a set of vertices,  $C$ , such that for any edge, at least one of its ends is a member of  $C$ .

- 2.15 **EXAMPLE** A museum has valuable exhibits. So they post guards. There are eight halls, laid out as below. To be maximally efficient, they will post guards at the corners  $w_0, \dots, w_5$ . What is the smallest number of guards that will suffice to watch all of the hallways?

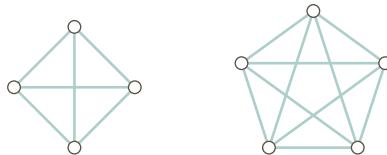


Obviously, one guard will not do. A two-element set such that for every hallway, at least one if its ends in in the set is  $C = \{w_0, w_4\}$ .

- 2.16 **PROBLEM (Clique)** Given a graph and a bound  $B \in \mathbb{N}$ , decide if the graph has a size  $B$  **clique**, a set of  $B$ -many vertices such that any two are connected.

The term ‘clique’ comes from studying social networks. If the graph nodes represent people and the edges connect friends then a clique is a set of  $B$ -many people who are all friends with each other.

A graph with a 4-clique has the subgraph like the one below on the left and any graph with a 5 clique has the subgraph like the one the right.



- 2.17 EXAMPLE Decide if this graph has a 4-clique.

2.18 ANIMATION: Instance of the Clique problem

- 2.19 PROBLEM (Broadcast) Given a graph with initial vertex  $v_0$ , and a bound  $B \in \mathbb{N}$ , decide if a message can spread from  $v_0$  to every other vertex within  $B$  steps. At each step, any node that has heard the message can transmit it to at most one adjacent node.

2.20 ANIMATION: Instance of the Broadcast problem

- 2.21 EXAMPLE In the graph no vertex is more than three edges away from the initial one. The animation shows it taking four steps to broadcast.

- 2.22 PROBLEM (Three-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.

- 2.23 EXAMPLE Let  $X = \{a, b\}$ ,  $Y = \{b, c\}$ , and  $Z = \{a, d\}$ , so that  $n = 2$ . Then  $M = X \times Y \times Z$  contains these eight elements.

$$M = \{\langle a, b, a \rangle, \langle a, c, a \rangle, \langle b, b, a \rangle, \langle b, c, a \rangle, \langle a, b, d \rangle, \langle a, c, d \rangle, \langle b, b, d \rangle, \langle b, c, d \rangle\}$$

The set  $\hat{M} = \{\langle a, b, a \rangle, \langle b, c, d \rangle\}$  has 2 elements and they disagree in the first coordinate, as well as on the second and third.

- 2.24 EXAMPLE Fix  $n = 4$  and consider  $X = \{1, 2, 3, 4\}$ ,  $Y = \{10, 20, 30, 40\}$ , and  $Z = \{100, 200, 300, 400\}$ , all four-element sets. Also fix this subset of  $X \times Y \times Z$ .

$$M = \{\langle 1, 10, 200 \rangle, \langle 1, 20, 300 \rangle, \langle 2, 30, 400 \rangle, \langle 3, 10, 400 \rangle, \\ \langle 3, 40, 100 \rangle, \langle 3, 40, 200 \rangle, \langle 4, 10, 200 \rangle, \langle 4, 20, 300 \rangle\}$$

A matching is  $\hat{M} = \{\langle 1, 20, 300 \rangle, \langle 2, 30, 400 \rangle, \langle 3, 40, 100 \rangle, \langle 4, 10, 200 \rangle\}$ .

- 2.25 PROBLEM (Subset Sum) Given a multiset of natural numbers  $S = \{n_0, \dots, n_{k-1}\}$  and a target  $T \in \mathbb{N}$ , decide if a subset of  $S$  sums to the target.

Recall that a multiset is like a set in that the order of the elements is not significant but is different than a set in that repeats do not collapse: the multiset  $\{1, 2, 2, 3\}$  is different than the multiset  $\{1, 2, 3\}$ .

- 2.26 EXAMPLE Decide if some of the numbers  $\{911, 22, 821, 563, 405, 986, 165, 732\}$  add to  $T = 1173$ . One such collection is  $\{165, 986, 22\}$ .

- 2.27 EXAMPLE No sum of the numbers  $\{831, 357, 63, 987, 117, 81, 6785, 606\}$  adds to  $T = 2105$ . All of the numbers are multiples of three, while the target  $T$  is not.

- 2.28 PROBLEM (Knapsack) Given a finite set  $S$  whose elements  $s$  have a weight  $w(s) \in \mathbb{N}^+$  and a value  $v(s) \in \mathbb{N}^+$ , along with a weight bound  $B \in \mathbb{N}^+$  and a value target  $T \in \mathbb{N}^+$ , 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.

Imagine that we have items to pack in a knapsack and we can carry at most ten pounds. Can we pack a value of  $T = 100$  or more?

Item	a	b	c	d
Weight	3	4	5	6
Value	50	40	10	30

We pack the most value while keeping to the weight limit by taking items (a) and (b). So we cannot meet the value target.

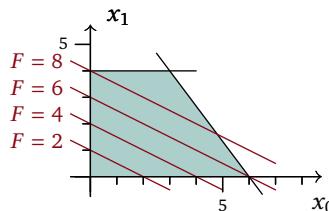
- 2.29 PROBLEM (Partition) Given a finite multiset  $A$  that has for each of its elements an associated positive number size  $s(a) \in \mathbb{N}^+$ , decide if there is a division of the set into two halves,  $\hat{A}$  and  $A - \hat{A}$ , so that the total of the sizes is the same in both halves,  $\sum_{a \in \hat{A}} s(a) = \sum_{a \notin \hat{A}} s(a)$ .

- 2.30 EXAMPLE The set  $A = \{I, a, my, go, rivers, cat, hotel, comb\}$  has eight words. The size of a word,  $s(\sigma)$ , is the number of letters. Then  $\hat{A} = \{cat, river, I, a, go\}$  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 of representatives depends in part on the state's population. Below are the numbers for the 2020 election; all of a state's representatives vote for the same person (we will ignore some fine points). The Partition Problem asks if there could be a tie.

<i>Reps</i>	No. states	States	<i>Reps</i>	No. states	States
55	1	CA	11	4	AZ, IN, MA, TN
38	1	TX	10	4	MD, MN, MO, WI
29	2	FL, NY	9	3	AL, CO, SC
20	2	IL, PA	8	2	KY, LA
18	1	OH	7	3	CT, OK, OR
16	2	GA, MI	6	6	AR, IA, KS, MS, NV, UT
15	1	NC	5	3	NE, NM, WV
14	1	NJ	4	5	HI, ID, ME, NH, RI
13	1	VA	3	8	AK, DE, DC, MT, ND, SD, VT, WY
12	1	WA			

- 2.32 PROBLEM (Linear Programming) Optimize a linear function  $F(x_0, \dots, x_n) = c_0x_0 + \dots + c_nx_n$  (either maximize or minimize it), subject to linear constraints, ones of the form  $a_{i,0}x_0 + \dots + a_{i,n}x_n \leq b_i$  or  $a_{i,0}x_0 + \dots + a_{i,n}x_n \geq b_i$ .
- 2.33 EXAMPLE Maximize  $F(x_0, x_1) = x_0 + 2x_1$  subject to  $4x_0 + 3x_1 \leq 24$ ,  $x_1 \leq 4$ ,  $x_0 \geq 0$  and  $x_1 \geq 0$ . The shaded region has the points that satisfy all the inequalities; these are said to be 'feasible' points.



The level lines of  $F$  show 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  $3 \times 3$  grid?

2.36 ANIMATION: Instance of the Crossword problem

- 2.37 **PROBLEM (15 Game)** Given an  $n \times n$  grid holding tiles numbered  $1, \dots, n - 1$ , 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 was popularized as a toy.



The final three problems, all of which involve divisors and primes, are inextricably linked, and indeed may be hard to tell apart at first glance. They have an impeccable history. In 1801, 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.” But as we understand these problems today, they appear to be quite different 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.<sup>†</sup> However, as is so often the case, at this moment we also have no proof that no such algorithm exists.<sup>‡</sup> Not all numbers of a given length are equally hard to factor. The hardest numbers to factor, using the best currently known techniques, are semiprimes, the product of two prime numbers.

- 2.39 **PROBLEM (Prime Factorization)** Given a number  $n \in \mathbb{N}$ , produce its decomposition into a product of primes.

Factoring seems to be hard. But what if you only want to know whether a number is prime and don’t care about its factors?

- 2.40 **PROBLEM (Primality)** Given a number  $n \in \mathbb{N}$ , determine if it is prime; that is, decide if there are no numbers  $a$  that divide  $n$  and such that  $1 < a < n$ .

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 hard. After all, for centuries, many of the smartest people in the world had worked

---

<sup>†</sup>No efficient algorithm is known on a non-quantum computer. <sup>‡</sup>There is no proof despite centuries of ingenious attacks on the problem by many of the brightest minds of the past, and of today. The presumed difficulty of this problem is at the heart of widely used algorithms in cryptography.

on composites and primes, and none of them had produced a fast test.<sup>†</sup>

However, in 2002 M Agrawal, N Kayal, and N Saxena produced such an algorithm, the AKS algorithm.<sup>‡</sup> Today, refinements of their technique run in  $\mathcal{O}(n^6)$ .

This dramatically illustrates that even though a problem is high profile, and even though many well-respected experts have worked on it, does not mean that the problem will never be solved. Although experts are expert and their opinions have value, nonetheless they can be wrong. People producing a result that gainsays established orthodoxy has happened before and will happen again. In short, one correct proof outweighs any number of expert opinions.



Nitin Saxena (b 1981),  
Neeraj Kayal (b 1979),  
Manindra Agrawal  
(b 1966)

## V.2 Exercises

2.41 Name the prime numbers less than one hundred.

2.42 Decide if each is prime. (A) 5 477 (B) 6 165 (C) 6 863 (D) 4 207  
(E) 7 689

✓ 2.43 Find a proper divisor of each. (A) 31 221 (B) 52 424 (C) 9 600 (D) 4 331  
(E) 877

2.44 We can specify a propositional logic behavior in a truth table and then produce such a statement in conjunctive normal form.

$P$	$Q$	$R$	
$F$	$F$	$F$	$T$
$F$	$F$	$T$	$T$
$F$	$T$	$F$	$F$
$F$	$T$	$T$	$F$
$T$	$F$	$F$	$T$
$T$	$F$	$T$	$F$
$T$	$T$	$F$	$T$
$T$	$T$	$T$	$F$

(A) The two terms  $P$  and  $\neg P$  are **atoms**. So are  $Q$ ,  $\neg Q$ ,  $R$ , and  $\neg R$ . Produce a three-atom clause that evaluates to  $F$  only on the  $F\text{-}T\text{-}F$  line.

(B) Produce three-atom clauses for each of the other truth table lines having the value  $F$  on the right.

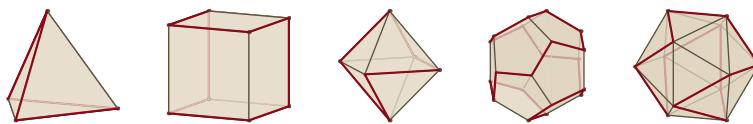
(C) Take the conjunction of those four clauses and verify that it has the given behavior.

✓ 2.45 Decide if each formula is satisfiable.

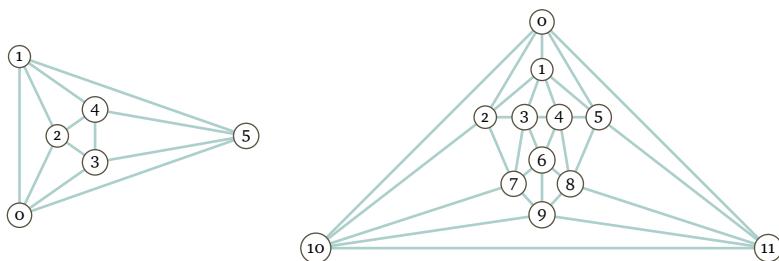
(A)  $(P \wedge Q) \vee (\neg Q \wedge R)$   
(B)  $(P \rightarrow Q) \wedge \neg((P \wedge Q) \vee \neg P)$

<sup>†</sup>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. <sup>‡</sup>At the time that they did most of the work, Kayal and Saxena were undergraduates.

- ✓ 2.46 Each of the five Platonic solids has a Hamiltonian circuit, as shown.



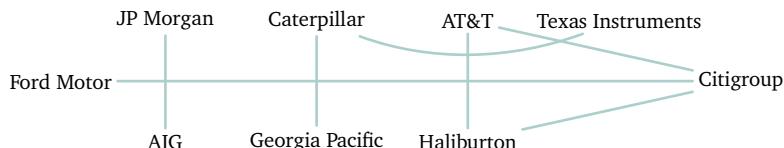
Hamilton used the fourth, the dodecahedron, for his game. Find a Hamiltonian circuit for the third and the fifth, the octahedron and the icosahedron. To make the connections easier to see, below we have grabbed a face in the back of each solid, and expanded it until we could squash the entire shape down into the plane without any edge crossings.



- 2.47 Give a planar map that requires four colors.

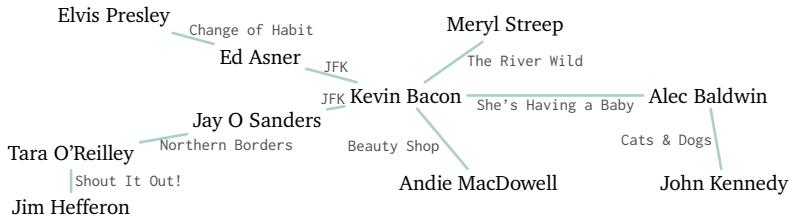
- 2.48 (A) The Four Color problem requires that the countries be contiguous, that they not consist of separated regions (that is, components). Give a planar map that consists of separated regions that requires five colors. (B) We also define adjacent to mean sharing a border that is an interval, not just a point. Give a planar map that, without that restriction, would require five colors.

- ✓ 2.49 Solve Example 2.26.  
 ✓ 2.50 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?

- ✓ 2.51 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?

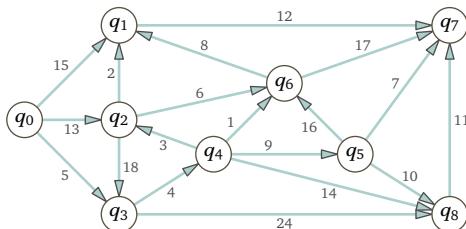
- (A) What is Elvis's Bacon number?
  - (B) John Kennedy's (no, it is not that John Kennedy)?
  - (C) Bacon's?
  - (D) How many movies separate me from Meryl Streep?
- ✓ 2.52 This Knapsack instance has no solution when the weight bound is  $B = 73$  and the value target is  $T = 140$ .

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.

- 2.53 Using the data in Example 2.31, decide if there could be a tie in the 2020 Electoral College.

- 2.54 Find the shortest path in this graph

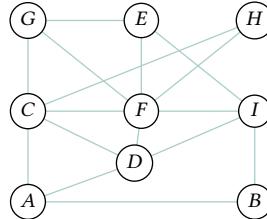


- (A) from  $q_2$  to  $q_7$ , (B) from  $q_0$  to  $q_8$ , (C) from  $q_8$  to  $q_0$ .

- 2.55 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.

- ✓ 2.56 What shape is a 3-clique? A 2-clique?
- 2.57 How many edges does a  $k$ -clique have?
- ✓ 2.58 The **Course Scheduling** problem starts with a list of students and the classes that they wish to take, and then finds how many time slots are needed to schedule the classes. If there is a student taking two classes then those two will not be

scheduled to meet at the same time. Here is an instance: a school has classes in Astronomy, Biology, Computing, Drama, English, French, Geography, History, and Italian. After students sign up, the graph below shows which classes have an overlap. For instance Astronomy and Biology share at least one student while Biology and Drama do not.

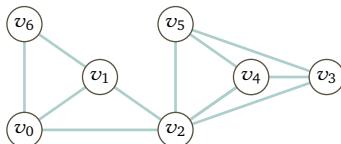


What is the minimum number of class times that we must use? In graph coloring terms, we define that classes meeting at the same time are the same color and we ask for the minimum number of colors needed so that no two same-colored vertices share an edge. (A) Show that no three-coloring suffices. (B) Produce a four-coloring.

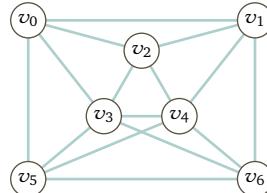
2.59 If a Boolean expression  $F$  is satisfiable, does that imply that its negation  $\neg F$  is not satisfiable?

2.60 Some authors define the **Satisfiability** problem as: given a finite set of propositional logic statements, find if there is a single input tuple  $b_0, \dots b_{j-1}$ , where each  $b_i$  is either  $T$  or  $F$ , that satisfies them all. Show that this is equivalent to the definition given in Problem 2.9.

- ✓ 2.61 Find all 3-cliques in this graph.



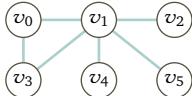
2.62 Is there a 3-clique in this graph? A 4-clique? A 5-clique?



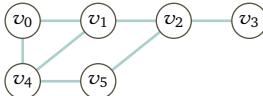
2.63 Recall that **Vertex Cover** inputs a graph  $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$  and a number  $k \in \mathbb{N}$ , and asks if there is a subset  $S$  of at most  $k$  vertices such that for each edge at least one endpoint is an element of  $S$ . The **Independent Set** problem inputs a graph

and a number  $\hat{k} \in \mathbb{N}$  and asks if there is a subset  $\hat{S}$  with at least  $\hat{k}$  vertices such that for each edge at most one endpoint is in  $\hat{S}$ . The two are obviously related.

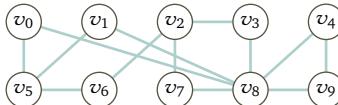
- (A) In this graph find a vertex cover  $S$  with  $k = 2$  elements. Find an independent set with  $\hat{k} = 4$  elements.



- (B) In this graph find a vertex cover with  $k = 3$  elements, and an independent set with  $\hat{k} = 3$  elements.

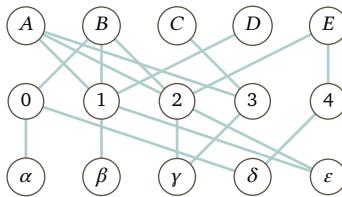


- (C) In this graph find a vertex cover  $S$  with  $k = 4$  elements. Find an independent set  $\hat{S}$  with  $\hat{k} = 6$  elements.



- (D) Prove that  $S$  is a vertex cover if and only if its complement  $\hat{S} = \mathcal{N} - S$  is an independent set.

- ✓ 2.64 A college department has instructors  $A, B, C, D$ , and  $E$ . They need placing into courses 0, 1, 2, 3, and 4. The available time slots are  $\alpha, \beta, \gamma, \delta$ , and  $\varepsilon$ . This shows which instructors can teach which courses, and which courses can run in which slots.



For example, instructor  $A$  can only teach courses 1, 2, and 3. And, course 0 can only run at time  $\alpha$  or time  $\delta$ . Verify that this is an instance of the Three-dimensional Matching problem and find a match.

- 2.65 Consider Three Dimensional Matching, Problem 2.22. Let  $X = \{a, b, c\}$ ,  $Y = \{b, c, d\}$ , and  $Z = \{a, d, e\}$ . (A) List the elements of  $M = X \times Y \times Z$ . (B) Is there a three element subset  $\hat{M}$  whose triples have the property that no two of them agree on any coordinate?

- 2.66 In Example 2.21 the broadcast takes four steps. Can it be done in fewer?

## 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. More precisely, it is a uniform family of tasks, typically 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 conclusions about the first subsumes the second and also because the first feels more natural.<sup>†</sup> 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.<sup>‡</sup> An algorithm is not a program, although it should be described in a way that is detailed enough that implementing it is routine for an experienced professional. The description should be complete enough to analyze the 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.<sup>§</sup>



Science United

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.

To illustrate the differences between the problems, algorithms, and programs: we’ve discussed the problem of **Prime Factorization**. One algorithm is brute force that is given an input  $n > 1$  and tries every  $k \in (1 .. n)$  to see if  $k$  divides  $n$ . We could implement that algorithm with a program written in Racket.

---

<sup>†</sup>There are interesting problems with only one task, such as computing the digits of  $\pi$ . <sup>‡</sup>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). <sup>§</sup>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 type is the **optimization problem**. These ask for a solution that is best according to some metric. The Shortest Path problem is one of these, as is the Minimal Spanning Tree problem.

A perhaps less familiar problem type is the **search problem**. For these, while there may be many solutions in the search space, the algorithm can stop when it has found one. An example inputs a Propositional Logic statement and outputs any truth table line witnessing that the statement is satisfiable. A second is the problem that inputs a weighted graph, two vertices, and a bound  $B \in \mathbb{R}$ , and finds a path between the vertices that costs less than the bound. Another example is that of finding a  $B$ -coloring for a graph. Still another is the Knapsack problem. In all of these, we want to find if there is a way to solve the problem, such as a way to pack the knapsack, and if there is at least one then we are done.

A **decision problem** is one with a ‘Yes’ or ‘No’ answer.<sup>†</sup> The first problem that we saw, the *Entscheidungsproblem*, is one of these.<sup>‡</sup> 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  $\phi_e$  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 some language and asked for an algorithm to decide if the input is a member of that language. We will see many examples in the rest of this chapter but just to give one: we can express the task of deciding the primality of a natural number, the Primality problem, as that of deciding membership in the set of bitstrings  $\{\sigma \in \mathcal{P}(\mathbb{B}) \mid \sigma \text{ is the binary representation of a prime } n \in \mathbb{N}\}$ .

This relates to the discussion from the Languages section, on page 149, about the distinction between deciding a language and recognizing it. We are ready for the following.

- 3.1 **DEFINITION** A language  $\mathcal{L}$  is **decided** by a Turing machine, or is **Turing machine decidable**, if the function computed by that machine is the characteristic function of the language. The language is **recognized**, or **accepted**, by a machine when for each input  $\sigma \in \mathbb{B}^*$ , if  $\sigma \in \mathcal{L}$  then the machine returns 1, while if  $\sigma \notin \mathcal{L}$  then either the machine does not halt or it returns something other than 1.

Restated,  $\mathcal{P}$  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. Thus, if  $\mathcal{P}$  decides the language then it halts for all inputs.

---

<sup>†</sup>Although a decision problem calls for producing a function of a kind, a Boolean function, they are important enough to be a separate category. <sup>‡</sup>Recall that the word is German for “decision problem” and that it asks for an algorithm to decide, given a mathematical statement, whether that statement is true or false.

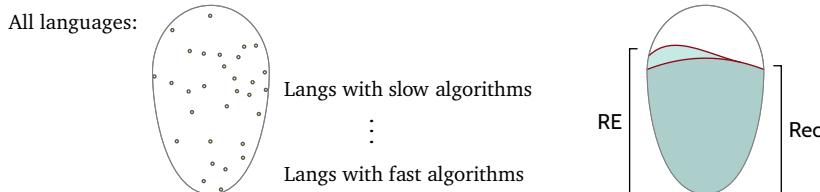
- 3.2 REMARK One reason that we are interested in language membership decisions comes from practice. A language compiler must recognize whether a given source file is a member of the language.

Another reason is that Finite State machines can only do one thing, decide languages, and we did lots of these such as producing a machine that decides if an input string is a member of  $\mathcal{L} = \{\sigma \in \{a, b\}^* \mid \sigma \text{ contains at least two } b's\}$  or proving that no Finite State machine can determine membership in  $\{a^n b^n \mid n \in \mathbb{N}\}$ . Thus, to compare Finite State machines with other machines we must compare which languages they can decide.

Still another reason is that in many contexts stating a problem in this way is natural, as we saw with the Halting problem.

Distinctions between problem types can be fuzzy. In addition, often we can describe a task with more than one type. An instance is the task of determining the evenness of a natural number. We could express it as the function problem ‘given  $n$ , return its remainder on division by 2’, or as the language decision problem of determining membership in  $\mathcal{L} = \{2k \mid k \in \mathbb{N}\}$ .

When we have a choice of problem types, we prefer language decision problems. It is our default interpretation of ‘problem’ and we will focus on them in the rest of the book. In addition, we will be sloppy about the distinction between the decision problem for a language and the language itself; for instance, we will write  $\mathcal{L}$  for a problem.



3.3 FIGURE: Both of these show the collection of languages,  $\mathcal{P}(B^*)$ , which we often call the ‘problems’. On the left, the dots in the blob emphasize that this is a collection of separate sets, not a continuum. It is drawn with quickly-solvable problems, those with a fast decider, at the bottom. But there is a catch. On the right the shaded collection **Rec** consists of the Turing computable languages. Similarly, **RE** consists of the languages that are computably enumerable. So this diagram makes the point that not all languages have a decider (or a recognizer)—some languages are perfectly good problems, but they are unsolvable.

Many problem types can be recast as decision problems.

- 3.4 EXAMPLE The Satisfiability problem, as we have stated it, is a decision problem. We can recast it as the problem of determining membership in the language  $SAT = \{E \mid E \text{ is a satisfiable propositional logic statement}\}$ . This recasting is trivial, suggesting that the language recognition problem form is a natural way to describe the underlying task.

- 3.5 EXAMPLE Consider the **Satisfying assignment** problem that takes in a boolean expression  $E(P_0, P_1 \dots)$  and returns an assignment of truth values for  $P_0, P_1, \dots$  that makes  $E$  evaluate to  $T$ , or a flag that there is no such assignment. This is a search problem. We will show that it is equivalent to the language decision problem  $SAT$  in that if we could solve one in polytime then we could also solve the other in polytime. One way is easy: if we could solve the **Satisfying assignment** problem in polytime then given an expression  $E$  we can apply the search algorithm to it and if there is a successful outcome then we've shown that  $E \in SAT$  in polytime.

For the other direction suppose that we have a polytime algorithm for  $SAT$ . Given an expression  $E$ , apply  $SAT$ 's algorithm to determine whether it is satisfiable at all. If so then we need to return an assignment. First fix  $P_0$  at  $F$  and run  $SAT$ 's algorithm. If  $E(P_0 = F, P_1, P_2 \dots)$  is not satisfiable then our assignment needs  $P_0 = T$ , 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 around the polytime algorithm for  $SAT$ , and therefore the algorithm as a whole is polytime.

Recasting optimization problems as language decision problems often involves a parametrized sequence of languages.

- 3.6 EXAMPLE The **Chromatic Number** problem inputs a graph and returns a minimal  $k \in \mathbb{N}$  such that the graph is  $k$ -colorable. Recast it by considering the family of languages,  $\mathcal{L}_B = \{\mathcal{G} \mid \mathcal{G} \text{ that has a } B\text{-coloring}\}$  for  $B \in \mathbb{N}^+$ . If we could solve these language decision problems then we could compute the minimal chromatic number by testing  $B = 1, B = 2$ , etc., until we find the smallest  $B$  for which  $\mathcal{G} \in \mathcal{L}_B$ .
- 3.7 EXAMPLE The **Traveling Salesman** problem is an optimization problem. Recast it as a sequence of language decision problems as above: consider a parameter  $B \in \mathbb{N}$  and define  $\mathcal{T}\mathcal{S}_B = \{\mathcal{G} \mid \text{the graph } \mathcal{G} \text{ has a circuit of length no more than } B\}$ .

These recastings of optimization problems preserve polytime solvability. For instance, for the **Traveling Salesman** problem, if for each  $B$  we could solve member of the family  $\mathcal{T}\mathcal{S}_B$  in time  $\mathcal{O}(n^k)$  then looping through  $B = 1, B = 2$ , etc., will solve the optimization problem in polytime, namely time  $\mathcal{O}(n^{k+1})$ .

One more remark. Sometimes selecting the problem type requires judgment. Consider the task of finding rational number roots of a polynomial. As a function problem it is ‘given a polynomial  $p$ , return its rational roots’, but a natural language decision problem restatement is ‘decide if  $\langle p, r \rangle$ , belongs to the set of sequences consisting of a polynomial and one of its rational roots’. The second just ask us to plug  $r$  into  $p$  and does not seem to capture the essential difficulty.

**Statements and representations** To be complete, the description of a problem must include the form of the inputs and outputs. For instance, if we state a problem as: ‘input two numbers and output their midpoint’ then we have not fully specified what needs to be done. The input or output might use strings representing decimal numbers, or might be floating point representations, or even might be in unary.

This matters in that the input’s form can change the algorithm that we choose

or its runtime behavior. Suppose that we must decide whether a number is divisible by four. If the input is in binary then the algorithm is immediate: a number is divisible by four if and only if in its final two bits are 00.<sup>†</sup> In contrast, if it is in unary then we may scan the 1's, keeping track of the current remainder modulo 4.

On the other hand, the representation doesn't matter in 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 by converting to binary and then applying the binary algorithm.<sup>‡</sup> Typically the costs of different representations don't change the Big  $\mathcal{O}$  runtime behavior. For example we might have a graph algorithm whose run time is  $\mathcal{O}(n \lg n)$ . Even for this minimal time, we can find a representation for the input graphs, such as where inputting takes  $\mathcal{O}(n)$  time, that leaves the algorithm analysis conclusion unchanged at  $\mathcal{O}(n \lg n)$ .

With this in mind, we will take the view, which we call **Lipton's Thesis**, that everything of interest can be represented with reasonable efficiency by bitstrings.<sup>§</sup> This applies to all of the mathematical problems stated earlier. But it also applies to cases that may seem less natural, such as that we can use bitstrings to represent with sufficient fidelity Beethoven's 9th Symphony or an exquisite Old Master.<sup>||</sup>



3.8 FIGURE: *Basket of Fruit* by Caravaggio (1571–1610)

---

<sup>†</sup>Thus, on a Turing machine, if when the machine starts the head is under the final character, then the machine does not even need to read the entire input to decide the question. The algorithm runs in time independent of the input length. <sup>‡</sup>That is, the unary case reduces to the binary one. <sup>§</sup>'Reasonable' means that it is not so inefficient as to greatly change the big- $\mathcal{O}$  behavior. <sup>||</sup>This is in a way like Church's Thesis. We cannot prove it, but our experience with digital reproduction of music, movies, etc., finds that it is so.

Consequently, in practice researchers often do not mention representations. We may describe the Shortest Path problem as, “Given a weighted graph and two vertices . . .” in place of the more complete, “Given the following reasonably efficient bitstring representation of a weighted graph  $\mathcal{G}$  and vertices  $v_0$  and  $v_1$ , . . .” Outside of this section we also do this, leaving implementation details to a programmer. (When we do discuss representations, we use  $\text{str}(x)$  to denote a convenient, reasonably efficient, bitstring representation of  $x$ .<sup>†</sup>) Basically, the representation details do not affect the outcome of our analysis, much.

- 3.9 REMARK There is a caveat. We have seen that conflating  $\{n \in \mathbb{N} \mid n \text{ is prime}\}$  with  $\{\sigma \in \mathbb{B}^* \mid \sigma \text{ represents a prime number}\}$  can cause confusion. The distinction between thinking of an algorithm as inputting a number and thinking of it as inputting the string representation of a number is the basis for describing the Big  $\mathcal{O}$  behavior of that algorithm as pseudopolynomial. This is because the binary representation of a number  $n$  takes  $\mathcal{O}(\lg n)$  bits and so inputting it takes  $\mathcal{O}(\lg n)$  ticks.

### V.3 Exercises

- ✓ 3.10 For each of these, list three examples and then—speaking informally, since some of them do not have formal definitions—describe the difference between them and an algorithm. (A) a heuristic (B) pseudocode (c) a Turing machine (D) a flowchart (E) source code (F) an executable (G) a process
- 3.11 Your friend asks, “So, if a problem is essentially a set of strings, what constitutes a solution?” Answer them.
- 3.12 What is the difference between a decision problem and a language decision problem?
- 3.13 As an illustration of the thesis that even surprising things can be represented reasonably efficiently and with reasonable fidelity in binary, we can do a simple calculation. (A) At 30 cm, the resolution of the human eye is about 0.01 cm. How many such pixels are there in a photograph that is 21 cm by 30 cm? (B) We can see about a million colors. How many bits per pixel is that? (C) How many bits for the photo, in total?
- 3.14 Name something important that cannot be represented in binary.
- ✓ 3.15 True or false: any two programs that implement the same algorithm must compute the same function. What about the converse?
- 3.16 Some tasks are hard to express as a language decision problem. Consider sorting the characters of a string into ascending order. Briefly describe why each of these language decision problems fails to capture the task’s essential difficulty. (A)  $\{\sigma \in \Sigma^* \mid \sigma \text{ is sorted}\}$  (B)  $\{\langle \sigma, p \rangle \mid p \text{ is a permutation that orders } \sigma\}$

---

<sup>†</sup>Many authors use diamond brackets to stand for a representation, as in ‘ $\langle \mathcal{G}, v_0, v_1 \rangle$ ’. Here, we reserve diamond brackets for sequences.

- ✓ 3.17 For each language decision problem, name three members of the set, if there are three, and then sketch an algorithm solving it.

- (A)  $\mathcal{L}_0 = \{ \langle n, m \rangle \in \mathbb{N}^2 \mid n + m \text{ is a square and one greater than a prime} \}$
- (B)  $\mathcal{L}_1 = \{ \sigma \in \{0, \dots, 9\}^* \mid \sigma \text{ represents in decimal a multiple of } 100 \}$
- (C)  $\mathcal{L}_2 = \{ \sigma \in \mathbb{B}^* \mid \sigma \text{ has more } 1\text{'s than } 0\text{'s} \}$
- (D)  $\mathcal{L}_3 = \{ \sigma \in \mathbb{B}^* \mid \sigma^R = \sigma \}$

3.18 Solve the language decision problem for (A) the empty language, (B) the language  $\mathbb{B}$ , and (C) the language  $\mathbb{B}^*$ .

3.19 For each language, sketch an algorithm that solves the language decision problem.

- (A)  $\{ \sigma \in \mathbb{B}^* \mid \sigma \text{ matches the regular expression } a^*ba^* \}$
- (B) The language defined by this grammar

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA \mid \epsilon \\ B &\rightarrow bB \mid \epsilon \end{aligned}$$

3.20 Solve each decision problem about Finite State machines,  $\mathcal{M}$ , by producing an algorithm. (A) Given  $\mathcal{M}$ , decide if the language accepted by  $\mathcal{M}$  is empty. (B) Decide if the language accepted by  $\mathcal{M}$  is infinite. (C) Decide if  $\mathcal{L}(\mathcal{M})$  is the set of all strings,  $\Sigma^*$ .

3.21 For each language decision problem, give an algorithm that runs in  $\mathcal{O}(1)$ .

- (A) The language of minimal-length binary representations of numbers that are nonzero.
- (B) The binary representations of numbers that exceed 1000.

3.22 In a graph, a **bridge edge** is one whose removal disconnects the graph. That is, there are two vertices that before the bridge is removed are connected by a path, but are not connected after it is removed. (More precisely, a **connected component** of a graph is a set of vertices that can be reached from each other by a path. A bridge edge is one whose removal increases the number of connected components.) The problem is: given a graph, find a bridge. Is this a function problem, a decision problem, a language decision problem, a search problem, or an optimization problem?

- ✓ 3.23 For each, give the categorization that best applies: a function problem, a decision problem, a language decision problem, a search problem, or an optimization problem. (A) The **Graph Connectedness** problem, which inputs a graph and decides whether for any two vertices there is a path between them. (B) The problem that inputs two natural numbers and returns their least common multiple. (C) The **Graph Isomorphism** problem that inputs two graphs and determines whether they are isomorphic. (D) The problem that takes in a propositional logic statement and returns an assignment of truth values to its inputs that makes the statement true, if there is such an assignment. (E) The **Nearest Neighbor** problem that inputs a weighted graph and a vertex,

and returns a vertex nearest the given one that does not equal the given one. (F) The **Discrete Logarithm** problem: given a prime number  $p$  and two numbers  $a, b \in \mathbb{N}$ , determine if there is a power  $k \in \mathbb{N}$  so that  $a^k \equiv b \pmod{p}$ . (G) The problem that inputs a bitstring and decides if the number that it represents in binary will, when converted to decimal, contain only odd digits.

- ✓ 3.24 For each, give the characterization that best applies: a function problem, a decision problem, a language decision problem, a search problem, or an optimization problem. (A) The **3-Satisfiability** problem, Problem 2.10 (B) The **Divisor** problem, Problem 2.38 (C) The **Prime Factorization** problem, Problem 2.39 (D) The  **$\mathcal{F}$ -SAT** problem, where the input is a propositional logic expression and the output is either an assignment of  $T$  and  $F$  to the expression's variables that makes it evaluate to  $T$ , or the string **None**. (E) The **Primality** problem, Problem 2.40

3.25 Express each task as a language decision problem. Include in the description explicit mention of the string representation. (A) Decide whether a number is a perfect square. (B) Decide whether a triple  $\langle x, y, z \rangle \in \mathbb{N}^3$  is a Pythagorean triple, that is, whether  $x^2 + y^2 = z^2$ . (C) Decide whether a graph has an even number of edges. (D) Decide whether a path in a graph has any repeated vertices.

- ✓ 3.26 Recast each as a language decision problem. Include explicit mention of the string representation. (A) Given a natural number, do its factors add to more than twice the number? (B) Given a Turing machine and input, does the machine halt on the input in less than ten steps? (C) Given a propositional logic statement, are there three different assignments that evaluate to  $T$ ? That is, are there more than three lines in the truth table that end in  $T$ ? (D) Given a weighted graph and a bound  $B \in \mathbb{R}$ , for any two vertices is there a path from one to the other with total cost less than the bound?

3.27 Recast each in language decision terms. Include explicit mention of the string representation. (A) **Graph Colorability**, Problem 2.7, (B) **Euler Circuit**, Problem 2.4, (C) **Shortest Path**, Problem 2.5.

3.28 Restate the **Halting** problem as a language decision problem.

- ✓ 3.29 As stated, the **Shortest Path** problem, Problem 2.5, is an optimization problem. Convert it into a parametrized family of decision problems. *Hint:* use the technique outlined following the **Traveling Salesman** problem, Problem 2.3.
- ✓ 3.30 Express each optimization problem as a parametrized family of language decision problems. (A) Given a 15 **Game** board, find the least number of slides that will solve it. (B) Given a Rubik's cube configuration, find the least number of moves to solve it. (C) Given a list of jobs that must be accomplished to assemble a car, along with how long each job takes and which jobs must be done before other jobs, find the shortest time to finish the entire car.

3.31 As stated, the **Hamiltonian Circuit** problem is a decision problem. Give a function version of this problem. Also give an optimization version.

3.32 The different problem types are related. Each of these inputs a square matrix  $M$  with more than 3 rows, and relates to a  $3 \times 3$  submatrix (form the submatrix by picking three rows and three columns, which need not be adjacent). Characterize each as a function problem, a decision problem, a search problem, or an optimization problem. (A) Find a submatrix that is invertible. (B) Decide if there is an invertible submatrix. (c) Return a submatrix that is invertible, or the string ‘None’. (D) Return a submatrix whose determinant has the largest absolute value. Also give a language for an associated language decision problem.

3.33 Recast each function problem as a decision problem.

- (A) The problem that inputs two natural numbers and returns their product.
- (B) The **Nearest Neighbor** problem, that inputs a weighted graph and a vertex and returns the vertex nearest the given one, but not equal to it.

3.34 The **Linear Programming** problem is described on page 285. Give a version that is a (A) language decision problem, (B) search problem, (C) function problem, and (D) optimization problem. (For some parts there is more than one sensible answer.)

3.35 An **independent set** in a graph is a collection of vertices such that no two are connected by an edge. Give a version of the problem of finding an independent set that is a (A) a decision problem, (B) language decision problem, (C) search problem, (D) function problem, and (E) optimization problem. (For some parts there is more than one reasonable answer.)

3.36 Give an example of a problem where the decision variant is solvable quickly but the search variant is not.

3.37 Let  $\mathcal{L}_F = \{ \langle n, B \rangle \in \mathbb{N}^2 \mid \text{there is an } m \in \{1, \dots, B\} \text{ that divides } n \}$  and consider its language decision problem. (A) Show that  $\langle d, B \rangle \in \mathcal{L}_F$  if and only if  $B$  is greater than or equal to the least prime factor of  $d$ . (B) Conclude that you can use a solution to the language recognition problem to solve the search problem that is given a number and returns a prime factor of that number.

- ✓ 3.38 Show how to use an algorithm that solves the **Shortest Path** problem to solve the **Vertex-to-Vertex Path** problem. How to use it on graphs that are not weighted?
- ✓ 3.39 Show that with an algorithm that quickly solves the **Subset Sum** problem, Problem 2.25, we can quickly solve the associated function problem of finding the subset.
- 3.40 Show how to use an algorithm that solves **Vertex-to-Vertex Path** problem to solve the **Graph Connectedness** problem, which inputs a graph and decides whether that graph is connected, so that for any two vertices there is a path between them.

## SECTION

## V.4 P

Recall that we usually blur the distinction between the problem of deciding membership in a language  $\mathcal{L}$  and the language itself. So to express that we are studying problems of a certain type we may say we are studying languages.

- 4.1 **DEFINITION** A **complexity class** is a collection of languages.

The term ‘complexity’ is there because these collections are often associated with some resource specification, so that a class consists of the languages that are accepted by a mechanical computer whose use of some resource fits the specification.<sup>†</sup>

- 4.2 **EXAMPLE** One complexity class is the collection of languages for which there is an deciding Turing machine that runs in time  $\mathcal{O}(n^2)$ . Thus  $\mathcal{C} = \{\mathcal{L}_0, \mathcal{L}_1, \dots\}$ , where each  $\mathcal{L}_j$  is decided by some machine  $\mathcal{P}_{i_j}$ , for which the function  $f$  relating the size of the machine’s input  $\sigma$  to the number of steps that the machine takes to finish is quadratic, that is,  $f$  is  $\mathcal{O}(n^2)$ .
- 4.3 **EXAMPLE** Another is the collection of languages accepted by some Turing machine that uses only in logarithmic space. That is, for such a machine with input string  $\sigma$ , the function  $f$  relating  $|\sigma|$  to the maximum number of tape squares that the machine visits in accepting a string of that length is logarithmic,  $f \in \mathcal{O}(\lg)$ .

Two points bear explication. As to the computing machine, researchers study not just Turing machines but other types of machines as well, including nondeterministic Turing machines and Turing machines with access to an oracle for random numbers. And as for the resource specification, it often involve bounds on the time or space behavior. But a class could instead be, for instance, the complement of  $\mathcal{O}(n^2)$ , so the specification isn’t always a bound.<sup>‡</sup>

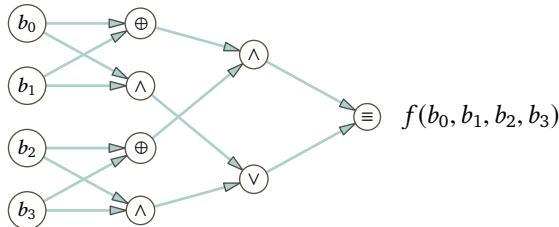
**Definition** The complexity class that we introduce now is the most important one. It is the collection of problems that under Cobham’s Thesis we take to be tractable.

- 4.4 **DEFINITION** A language decision problem is a member of the class **P** if there is an algorithm for it that on a deterministic Turing machine runs in polynomial time.
- 4.5 **EXAMPLE** The problem  $\mathcal{L} = \{\mathcal{G} \mid \text{there is a path between any two vertices}\}$  of deciding whether a given graph is connected is a member of **P**. To verify this, we must produce an algorithm that decides membership in this language, and that runs in polynomial time. One is to do a breadth first search of the graph. The runtime is cubic in the number of nodes.

---

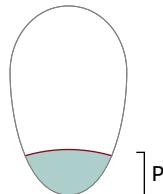
<sup>†</sup> There are other definitions of complexity class. Some authors require that in a class the characteristic function of each language can be computed under some resource specification. This has implications—if all of the members of a class must be computable by Turing machines then each class is countable. Here, we only say that it is a collection and so our definition is maximally general. <sup>‡</sup> At this writing there are 545 studied classes but the number changes frequently; see the Complexity Zoo, Section 4.

- 4.6 EXAMPLE Another member of  $\mathsf{P}$  is the problem of deciding whether two numbers are relatively prime,  $\{ \langle n_0, n_1 \rangle \in \mathbb{N}^2 \mid \text{their greatest common divisor is } 1 \}$ . As before, to verify that this language is a member of  $\mathsf{P}$  we produce an algorithm that determines membership and that runs in polytime. Euclid's algorithm fits the bill; it solves this problem and has runtime  $\mathcal{O}(\lg(\max(n_0, n_1)))$ .
- 4.7 EXAMPLE Still another member of  $\mathsf{P}$  is the String Search problem of deciding substring-ness,  $\{ \langle \sigma, \tau \rangle \in \Sigma^* \mid \sigma \text{ is a substring of } \tau \}$ . (Often in practice  $\tau$  is very long and is called the **haystack** while  $\sigma$  is short and is the **needle**.) The algorithm that first tests  $\sigma$  at the initial character of  $\tau$ , then at the next character, etc., has a runtime of  $\mathcal{O}(|\sigma| \cdot |\tau|)$ , which is  $\mathcal{O}(\max(|\sigma|, |\tau|)^2)$ .
- 4.8 EXAMPLE A **circuit** is a directed acyclic graph. Below, each vertex, called a **gate**, acts as a two input/one output Boolean function. The only exception is that some vertices are **input gates** that provide source bits (below on the left they are  $b_0, b_1, b_2, b_3 \in \mathbb{B}$ ). Edges are called **wires**. Below,  $\wedge$  is the boolean function ‘and’,  $\vee$  is ‘or’,  $\oplus$  is ‘exclusive or’, and  $\equiv$  is the negation of ‘exclusive or’, which returns 1 if and only if the two inputs bits are the same.



This circuit returns 1 if the sum of the input bits is a multiple of 3. The **Circuit Evaluation** problem inputs a circuit like this one and computes the output,  $f(b_0, b_1, b_2, b_3)$ . This problem is a member of  $\mathsf{P}$ .

- 4.9 EXAMPLE Although polytime is a restriction, nonetheless  $\mathsf{P}$  is a very large collection. More example members: (1) matrix multiplication, taken as a language decision problem for  $\{ \langle \sigma_0, \sigma_1, \sigma_2 \rangle \mid \text{they represent matrices with } M_0 \cdot M_1 = M_2 \}$  (2) minimal spanning tree,  $\{ \langle \mathcal{G}, T \rangle \mid T \text{ is a minimal spanning tree in } \mathcal{G} \}$  (3) edit distance, the number of single-character removals, insertions, or substitutions needed to transform between strings,  $\{ \langle \sigma_0, \sigma_1, n \rangle \mid \sigma_0 \text{ transforms to } \sigma_1 \text{ in at most } n \text{ edits} \}$ .



4.10 FIGURE: This blob contains all language decision problems, all  $\mathcal{L} \subseteq \mathbb{B}^*$ . Shaded is  $\mathsf{P}$ .

Two final observations. First, if a problem has an algorithm that is  $\mathcal{O}(\lg n)$  then that problem is in  $\text{P}$ . Second, the members of  $\text{P}$  are problems (more precisely, languages that represent problems), so it is wrong to say that an algorithm is in  $\text{P}$ .

**Effect of the model of computation** A problem is in  $\text{P}$  if it has an algorithm that is polytime. But algorithms are based on an underlying computing model. Is membership in  $\text{P}$  dependent on the model that we use?

In particular, our experience with Turing machines gives the sense that they involve a lot of tape moving. 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 often slower than related algorithms for other natural models, it is only by a factor of between  $n^2$  and  $n^4$ .<sup>†</sup> That is, if we have a problem for which there is a  $\mathcal{O}(n)$  algorithm on another model then we may find that on a Turing machine model it is  $\mathcal{O}(n^3)$ , or  $\mathcal{O}(n^4)$ , or  $\mathcal{O}(n^5)$ . So it is still in  $\text{P}$ .

A variation of Church's thesis, the [Extended Church's Thesis](#), posits that not only are all reasonable models of mechanical computation of equal power, but in addition that they are of equivalent speed in that we can simulate any reasonable model of computation<sup>‡</sup> in polytime on a probabilistic Turing machine.<sup>§</sup> Under the extended thesis, a problem that falls in the class  $\text{P}$  using Turing machines also falls in that class using any other natural models. (Note, however, that this thesis does not enjoy anything like the support of the original Church's Thesis. Also, we know of several problems, including the familiar Prime Factorization problem, that under the Quantum Computing model have algorithms with polytime solutions, but for which we do not know of any polytime solution in a non-quantum model. So the Quantum Computing model would provide a counterexample to the extended thesis, if we can produce physical devices matching that model.)

- 4.11 **REMARK** Recently a number of researchers claimed to have built devices that achieved [Quantum Supremacy](#), to have solved a problem using an algorithm running on a physical quantum computer that is not known to be solvable on a Turing machine or RAM machine in less than centuries.

There are reservations. For one, the claim is the subject of some scholarly dispute. For another, on its face, this is not general purpose computing; the problem solved is exotic. There are sound reasons to wonder whether quantum computers will ever be practical physical devices used for everyday problems, although scientists and engineers are making great progress. For our purposes we put this aside, but it is worth watching developments with keen interest.

**Naturalness** We give the class  $\text{P}$  our close attention because there are reasons to suppose that it is the best candidate for the collection of problems that have a feasible solution. We close with a discussion.

---

<sup>†</sup>We take a model to be 'natural' if it was not invented in order to be a counterexample to this. <sup>‡</sup>One definition of 'reasonable' is "in principle physically realizable" (Bernstein and Vazirani 1997). <sup>§</sup>A Turing machine with access to an oracle of random bits.

The first reason echoes 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, by Church's Thesis. Further, while their speeds may differ, all of these models run within polytime of each other.<sup>†</sup> That makes  $\mathbf{P}$  invariant under the choice of computing model: if a problem is in  $\mathbf{P}$  for any of these models then it is in  $\mathbf{P}$  for all. The fact that Turing machines are our standard is in some ways a historical accident but differences between the runtime behavior of any of these models is lost in the general polynomial sloppiness.

Another reason  $\mathbf{P}$  is a natural class is that we'd like that if two things,  $f$  and  $g$ , are easy to compute then a simple combination of the two is also easy. More precisely, fix total functions  $f, g: \mathbb{N} \rightarrow \mathbb{N}$  and consider these.

$$\mathcal{L}_f = \{ \text{str}(\langle n, f(n) \rangle) \in \mathbb{B}^* \times \mathbb{B}^* \mid n \in \mathbb{N} \} \quad \mathcal{L}_g = \{ \text{str}(\langle n, g(n) \rangle) \in \mathbb{B}^* \times \mathbb{B}^* \mid n \in \mathbb{N} \}$$

(Recall that  $\text{str}(\dots)$  means that we represent the argument reasonably efficiently as a bitstring.) With that recasting of functions as languages,  $\mathbf{P}$  is closed under function addition, scalar multiplication by an integer, subtraction, multiplication, and composition. It is also closed under language concatenation and the Kleene star operator. It is the smallest nontrivial class with these appealing properties.

But the main reason that  $\mathbf{P}$  is our candidate is Cobham's Thesis, the contention that the formalization of 'tractable problem' should be that it has a solution algorithm that runs in polynomial time. We discussed this on page 271; a person may object that polytime is too broad a class to capture this idea because a problem whose solution algorithms cannot be improved below a runtime of  $\mathcal{O}(n^{1000000})$  is really not feasible or tractable. Further, using diagonalization we can produce such problems. However the problems produced in that way are artificial and empirical experience over close to a century of computing is that problems with solution algorithms of very large degree polynomial time complexity do not seem to arise 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  $\mathcal{O}(n^{1000000})$ .

Moreover, often in the past when a researcher has produced an algorithm for a problem with a runtime that is even a moderately large degree polynomial then, with this foot in the door, over the next few years the community brings to bear an array of mathematical and algorithmic techniques that bring the runtime degree down to reasonable size.

Even if the objection to Cobham's Thesis is right and  $\mathbf{P}$  is too broad, the class would still be useful because if we could show that a problem is not in  $\mathbf{P}$  then we would have shown that it has no solution algorithm that is practical.<sup>‡</sup> (This is as in the first and second chapter, where we considered Turing machine computations that are unbounded. Showing that something is not solvable even for an unbounded computation also shows that it is not solvable within bounds.)

---

<sup>†</sup>All, that is, of the non-quantum natural models. <sup>‡</sup>This argument has lost some of force in recent years with the rise of *SAT* solvers. These attack problems believed to not be in  $\mathbf{P}$  and can solve instances of the problems of moderately large size, using only moderately large computing resources. See Extra C.

So Cobham's Thesis, to this point, has held up. Insofar as theory should be a guide for practice, this is a compelling reason to use  $\text{P}$  as a touchstone for other complexity classes.

#### V.4 Exercises

- ✓ 4.12 True or False: if the language is finite then the language decision problem is in  $\text{P}$ .
- ✓ 4.13 Your coworker says something mistaken, “I've got a problem whose algorithm is in  $\text{P}$ .” They are being a little sloppy with terms; how?
- ✓ 4.14 What is the difference between an order of growth and a complexity class?
- ✓ 4.15 Your friend says to you, “I think that the Circuit Evaluation problem takes exponential time. There is a final vertex. It takes two inputs, which come from two vertices, and each of those take two inputs, etc., so that a five-deep circuit can have thirty two vertices.” Help them see where they are wrong.
- 4.16 In class, someone says to the professor, “Why aren't all languages in  $\text{P}$  according to this definition? I'll design a Turing machine  $\mathcal{P}$  so that no matter what the input is, it outputs 1. It only needs one step, so it is polytime for sure.” Explain how this is mistaken.
- 4.17 True or false: if a problem has a logarithmic solution then it is in  $\text{P}$ .
- 4.18 True or false: if a language is decided by a machine then its complement is also accepted by some machine.
- ✓ 4.19 Show that the decision problem for  $\{\sigma \in \mathbb{B}^* \mid \sigma = \tau^3 \text{ for some } \tau \in \mathbb{B}^*\}$  is in  $\text{P}$ .
- ✓ 4.20 Show that the language of palindromes,  $\{\sigma \in \mathbb{B}^* \mid \sigma = \sigma^R\}$ , is in  $\text{P}$ .
- 4.21 Sketch a proof that each problem is in  $\text{P}$ .
  - (A) The  $\tau^3$  problem: given a bitstring  $\sigma$ , decide if it has the form  $\sigma = \tau^3$ .
  - (B) The problem of deciding which Turing machines halt within ten steps.
- ✓ 4.22 Consider the problem of Triangle: given an undirected graph, decide if it has a 3-clique, three vertices that are mutually connected.
  - (A) Why is this not the Clique problem, from page 282?
  - (B) Sketch a proof that this problem is in  $\text{P}$ .
- ✓ 4.23 Prove that each problem is in  $\text{P}$  by citing the runtime of an algorithm that suits.
  - (A) Deciding the language  $\{\sigma \in \{a, \dots, z\}^* \mid \sigma \text{ is in alphabetical order}\}$ .
  - (B) Deciding the language of correct sums,  $\{\langle a, b, c \rangle \in \mathbb{N}^3 \mid a + b = c\}$ .
  - (C) Analogous to the prior item, deciding this language of triples of matrices that give correct products,  $\{\langle A, B, C \rangle \mid \text{the matrices are such that } AB = C\}$ .
  - (D) Deciding the language of primes,  $\{1^k \mid k \text{ is prime}\}$ .
  - (E) Vertex-to-Vertex Path:  $\{\langle \mathcal{G}, v_0, v_1 \rangle \mid \text{the graph } \mathcal{G} \text{ has a path from } v_0 \text{ to } v_1\}$ .

4.24 Find which of these are currently known to be in  $\mathbf{P}$  and which are not.  
*Hint:* you may need to look up the fastest known algorithm. (A) Shortest Path  
 (B) Knapsack (c) Euler Path (d) Hamiltonian Circuit

4.25 Is the empty language  $\{\} \subset \mathbb{B}^*$  a member of  $\mathbf{P}$ ?

4.26 The problem of Graph Connectedness is: given a finite graph, decide if there is a path from any vertex to any other. Sketch an argument that this problem is in  $\mathbf{P}$ .

4.27 Following the definition of complexity class, Definition 4.1, is a discussion of the additional condition of being computed by some machine under a resource specification, such as a Big  $\mathcal{O}$  constraint on time or space.

- (A) Show that the set of regular languages forms a complexity class, and that it meets this additional constraint.
- (B) The definition of  $\mathbf{P}$  uses Turing machines. We can view a Finite State machine as a kind of Turing machine, one that consumes its input one character at a time, never writes to the tape, and, depending on the state that the machine is in when the input is finished, prints 0 or 1. With that, argue that any regular language is an element of  $\mathbf{P}$ .

4.28 We have already studied the collection  $\mathbf{RE}$  of languages that are computably enumerable.

- (A) Recast  $\mathbf{RE}$  as a class of language decision problems.
- (B) Following Definition 4.1 is a discussion of the additional condition of being computed by a machine under a resource specification. Show that  $\mathbf{RE}$  also satisfies this condition.

4.29 Is  $\mathbf{P}$  countable or uncountable?

4.30 If  $\mathcal{L}_0, \mathcal{L}_1 \in \mathbf{P}$  and  $\mathcal{L}_0 \subseteq \mathcal{L} \subseteq \mathcal{L}_1$ , must  $\mathcal{L}$  be in  $\mathbf{P}$ ?

✓ 4.31 Is the Halting problem in  $\mathbf{P}$ ?

4.32 A common modification of the definition of Turing machine designates one state as an accepting state. Then the machine **decides the language**  $\mathcal{L}$  if it halts on all input strings, and  $\mathcal{L}$  is the set of strings that such that the machine ends in the accepting state. A language is **decidable** if it is decided by some machine. Prove that every language in  $\mathbf{P}$  is decidable.

✓ 4.33 Draw a circuit that inputs three bits,  $b_0, b_1, b_2 \in \mathbb{B}$ , and outputs the value of  $b_0 + b_1 + b_2 \pmod{2}$ .

4.34 Prove that the union of two complexity classes is also a complexity class. What about the intersection? Complement?

✓ 4.35 Prove that  $\mathbf{P}$  is closed under the union of two languages. That is, prove that if two languages are both in  $\mathbf{P}$  then so is their union. Prove the same for the union of finitely many languages.

4.36 Prove that  $\mathbf{P}$  is closed under complement. That is, prove that if a language is in  $\mathbf{P}$  then so is its set complement.

4.37 Prove that the class of languages  $\text{P}$  is closed under reversal. That is, prove that if a language is an element of  $\text{P}$  then so is the reversal of that language (which is the language of string reversals).

4.38 Show that  $\text{P}$  is closed under the concatenation of two languages.

4.39 Show that  $\text{P}$  is closed under Kleene star, meaning that if  $\mathcal{L} \in \text{P}$  then  $\mathcal{L}^* \in \text{P}$ .  
*(Hint:  $\sigma \in \mathcal{L}^*$  if  $\sigma = \epsilon$ , or  $\sigma \in \mathcal{L}$ , or  $\sigma = \alpha \hat{\cup} \beta$  for some  $\alpha, \beta \in \mathcal{L}^*$ )*

4.40 Show that this problem is unsolvable: give a Turing machine  $\mathcal{P}$ , decide whether it runs in polytime on the empty input. *Hint:* if you could solve this problem then you could solve the Halting problem.

4.41 There are studied complexity classes besides those associated with language decision problems. The class  $\text{FP}$  consists of the binary relations  $R \subseteq \mathbb{N}^2$  where there is a Turing machine that, given input  $x \in \mathbb{N}$ , can in polytime find a  $y \in \mathbb{N}$  where  $\langle x, y \rangle \in R$ .

- (A) Prove that this class closed under function addition, multiplication by a scalar  $r \in \mathbb{N}$ , subtraction, multiplication, and function composition.
- (B) Where  $f : \mathbb{N} \rightarrow \mathbb{N}$  is computable, consider this decision problem associated with the function,  $\mathcal{L}_f = \{ \text{str}(\langle n, f(n) \rangle) \in \mathbb{B}^* \mid n \in \mathbb{N} \}$  (where the numbers are represented in binary). Assume that we have two functions  $f_0, f_1 : \mathbb{N} \rightarrow \mathbb{N}$  such that  $\mathcal{L}_{f_0}, \mathcal{L}_{f_1} \in \text{P}$ . Show that the natural algorithm to check for closure under function addition is pseudopolynomial.

4.42 Where  $\mathcal{L}_0, \mathcal{L}_1 \subseteq \mathbb{B}^*$  are languages, we say that  $\mathcal{L}_1 \leq_p \mathcal{L}_0$  if there is a function  $f : \mathbb{B}^* \rightarrow \mathbb{B}^*$  that is computable, total, that runs in polytime, and so that  $\sigma \in \mathcal{L}_1$  if and only if  $f(\sigma) \in \mathcal{L}_0$ . Prove that if  $\mathcal{L}_0 \in \text{P}$  and  $\mathcal{L}_1 \leq_p \mathcal{L}_0$  then  $\mathcal{L}_1 \in \text{P}$ .

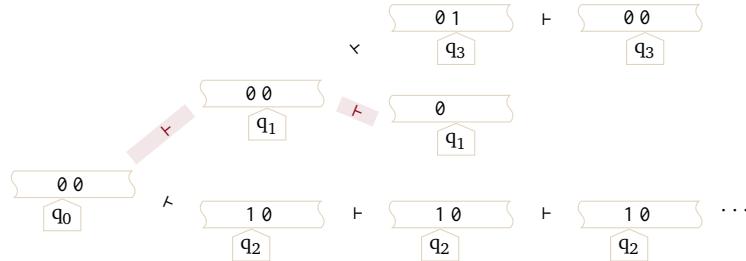
## SECTION

### V.5 NP

Recall that a Finite State machine is nondeterministic if from a present configuration and input it may pass to more than one next configuration (or one, or zero). We can make a nondeterministic Turing machine by allowing the same. Here is one; for instance, there are two instructions starting with  $q_0$  and  $0$  so if the machine is in  $q_0$  and it reads a  $0$  on the tape then it is legal both to go to state  $q_2$  and to state  $q_1$ .

$$\mathcal{P} = \{ q_001q_2, q_00Rq_1, q_10Bq_1, q_101q_3, q_211q_2, q_310q_3 \}$$

For such a machine the computational history can be more than a line, it can be a tree. Below is part of the tree for machine  $\mathcal{P}$  and input  $00$ , with the middle branch highlighted.



**Nondeterministic Turing machines** This modifies the definition of a Turing machine by changing the transition function so that it outputs sets.

- 5.1 **DEFINITION** A **nondeterministic Turing machine**  $\mathcal{P}$  is a finite set of instructions  $q_p T_p T_n q_n \in Q \times \Sigma \times (\Sigma \cup \{L, R\}) \times Q$ , where  $Q$  is a finite set of states and  $\Sigma$  is a finite set of tape alphabet characters, which contains at least two members, including blank, and does not contain the characters L or R. Some of the states,  $F \subseteq Q$ , are **accepting states**. The association of the present state and tape character with what happens next is given by the transition function,  $\Delta: Q \times \Sigma \rightarrow \mathcal{P}((\Sigma \cup \{L, R\}) \times Q)$ .

After the definition of the Turing machine, and of the Finite State machine and the nondeterministic Finite State machine, we described how these machines act as a sequence of ‘ $\vdash$ ’ steps from the initial configuration. Exercise 5.38 asks for a similar description for these machines.

Adding nondeterminism to Turing machines adds some wrinkles. The computation tree might have some branches that halt and some that don’t, or maybe some output 0 while some output 1. The simplest approach is not to define a function computed by a nondeterministic machine but instead to follow our strategy for Finite State machines and describe when the input string is accepted, so that they these machines are language deciders.

Of course, we have two mental models for how nondeterministic machines act. One is that the machine is unboundedly parallel and simultaneously computes all of the branches. The other is that the machine guesses which branch to follow—or is told by some demon—and then can deterministically check that branch. For both, the machine accepts an input string if at least one branch ends in an accepting state and does not accept the input if no branch ends in an accepting state.

That final sentence raises two points. First, once some branch accepts the input we might as well stop the computation, so we can take that accepting computations always halt. Second, “no branch ends” seemingly could mean that in a non-accepting computation some branches do not accept because their computation fails to halt. But we are using these machines as language deciders and we shall want to time them, including how long they take to not accept. So we will only define language-deciding when all branches halt.

- 5.2 **DEFINITION** Let  $\mathcal{P}$  be a nondeterministic Turing machine such that every branch in the computation tree, every sequence of valid transitions from the starting configuration, is finite. Then  $\mathcal{P}$  **accepts** an input string if at least one branch ends in an accepting state and otherwise **rejects** it. The machine **decides a language**  $\mathcal{L}$  when for every input string  $\sigma$ , if  $\sigma \in \mathcal{L}$  then  $\mathcal{P}$  accepts it while if  $\sigma \notin \mathcal{L}$  then  $\mathcal{P}$  rejects it.

Does adding nondeterminism to Turing machines add any new capabilities? For Finite State machines, nondeterminism does not make any difference in that a language is decided by some nondeterministic Finite State machine if and only if it is decided by a deterministic Finite State machine. But Pushdown machines are different: there are languages that a nondeterministic Pushdown machine can decide but that cannot be decided by any deterministic one.<sup>†</sup>

- 5.3 **LEMMA** For Turing machines, deterministic and nondeterministic machines decide the same languages.

*Proof* One direction is easy. A deterministic Turing machine is a special case of a nondeterministic one. So if a deterministic machine decides a language then a nondeterministic one does also.

Conversely, let the nondeterministic Turing machine  $\mathcal{P}$  decide the language  $\mathcal{L}$ . Consider a deterministic machine  $\mathcal{Q}$  that does a breadth-first search of  $\mathcal{P}$ 's computation tree. We will show that it decides the same language. Fix an input  $\sigma$ . If  $\mathcal{P}$  accepts  $\sigma$  then the search done by  $\mathcal{Q}$  will eventually find that node in the computation tree and then  $\mathcal{Q}$  accepts  $\sigma$ . If  $\mathcal{P}$  does not accept  $\sigma$  then every branch in its computation tree halts in a state that is not accepting. There is a longest such branch by König's lemma. So the breadth-first search will eventually exhaust all branches, and then  $\mathcal{Q}$  rejects  $\sigma$ .  $\square$

That proof is, basically, time-slicing. In everyday computing we simulate an unboundedly-parallel machine by having its CPU cycle among processes. For example, if on a machine's screen we have a window open to edit a program and another window showing a video, the computer may do the two tasks by updating the editor for a time, then updating the video, then back to the editor, etc. When we use a such a system we perceive that many things are happening at once although actually there is only one, or at least a limited number of simultaneous physical computations.<sup>‡</sup> This is a kind of dovetailing, a way of doing a breadth-first traversal of a number of computation branches.

So adding nondeterminism doesn't expand what Turing machines can compute. Nevertheless, these machines have several interesting aspects. We saw that nondeterministic Finite State machines are a good impedance match for many problems and similarly, nondeterministic Turing machines will help in thinking about some problems that on a serial device are hard to conceptualize. But that is not the most interesting thing. The most interesting thing is speed.

---

<sup>†</sup>One is the language of palindromes. <sup>‡</sup>Depending on how many cores are in the CPU.

**Speed** The real excitement is that a nondeterministic Turing machine might be much faster than a deterministic one.

- 5.4 EXAMPLE Consider the Satisfiability problem. Is this propositional logic formula satisfiable?

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

The natural approach is to compute a truth table and see whether the final column has any T's. Here, the formula is satisfiable because the TTF row ends in a T.

P	Q	R	$P \vee Q$	$P \vee \neg Q$	$\neg P \vee Q$	$\neg P \vee \neg Q \vee \neg R$	$Q \vee R$	(*)
F	F	F	F	T	T	T	F	F
F	F	T	F	T	T	T	T	F
F	T	F	T	F	T	T	T	F
F	T	T	T	F	T	T	T	F
T	F	F	T	T	F	T	F	F
T	F	T	T	T	F	T	T	F
T	T	F	T	T	T	T	T	T
T	T	T	T	T	T	F	T	F

For runtime, the number of table rows grows exponentially; it is 2 raised to the number of input variables. Going through the rows serially will be very slow.

Each line of the truth table is easy; the issue is that there are lots of lines. This situation is perfectly suited to unbounded parallelism. We could fork a child process for each line. These children are done quickly, certainly in polytime. If in the end any child is holding a T then the expression as a whole is satisfiable. That is, a nondeterministic machine does this job in polytime while a serial machine appears to require exponential time.

So while adding nondeterminism to Turing machines doesn't allow them to compute anything new, we might reasonably conjecture that it does allow them to compute those things faster.

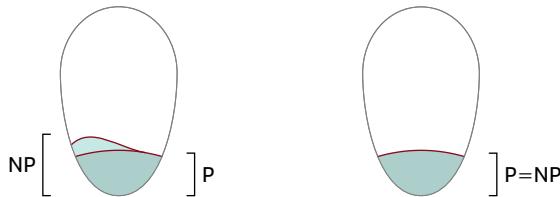
**Definition** Next we name a class of language decision problems associated with nondeterministic Turing machines.

- 5.5 DEFINITION The complexity class **NP** is the set of languages for which there is a nondeterministic Turing machine decider  $\mathcal{P}$  that runs in polytime, meaning that there is a polynomial  $p$  such that on input  $\sigma$ , all branches of  $\mathcal{P}$ 's computation halt in time  $p(|\sigma|)$ .

The following result is immediate because a deterministic Turing machine is a special case of a nondeterministic one.

- 5.6 LEMMA  $\mathbf{P} \subseteq \mathbf{NP}$

Very important: no one knows whether  $\mathbf{P}$  is a strict subset. That is, no one knows whether  $\mathbf{P} \neq \mathbf{NP}$  or  $\mathbf{P} = \mathbf{NP}$ . This is the biggest open problem in the Theory of Computing and we will say much more in the course of this chapter.

5.7 FIGURE: Which is it:  $P \subset NP$  or  $P = NP$ ?

A pattern in mathematical presentations is to have a definition that is conceptually clear, followed by a result that is what we use in practice to determine whether the definition applies. We now give that result. This is where we use the mental model of the machine guessing, or of being told an answer. Consider **Satisfiability**. Imagine that in Example 5.4 above the demon whispers, “Psst! Try  $\omega = TTF$ .” With that hint we can then verify using a deterministic machine.

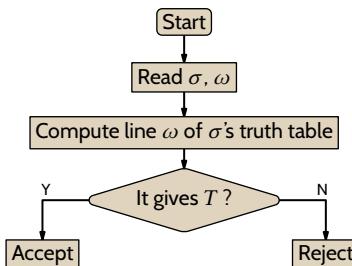
- 5.8 **DEFINITION** A **verifier** for a language  $\mathcal{L}$  is a deterministic Turing machine  $\mathcal{V}$  that inputs  $\langle \sigma, \omega \rangle \in \mathbb{B}^2$  and is such that  $\sigma \in \mathcal{L}$  if and only if there exists an  $\omega$  so that  $\mathcal{V}$  accepts  $\langle \sigma, \omega \rangle$ . The string  $\omega$  is called the **witness** or **certificate**.

That says  $\mathcal{V}$  ‘accepts’ its input but while we have given a definition of a nondeterministic Turing machine accepting its input, we have not given one for deterministic ones. We could give a modified definition with final states but for simplicity take it to mean that  $\mathcal{V}$  halts and outputs 1.

- 5.9 **LEMMA** A language is in  $NP$  if and only if it has a verifier that runs in time polynomial in  $|\sigma|$ . That is,  $\mathcal{L} \in NP$  if and only if there is a polynomial  $p$  and a deterministic Turing machine  $\mathcal{V}$  that halts on all inputs  $\langle \sigma, \omega \rangle$  in  $p(|\sigma|)$  time, and is such that  $\sigma \in \mathcal{L}$  exactly when there is a witness  $\omega$  where  $\mathcal{V}$  accepts  $\langle \sigma, \omega \rangle$ .

Before the lemma’s proof we will first discuss some aspects of both the definition and the lemma.

- 5.10 **EXAMPLE** Our touchstone is the **Satisfiability** problem. Using the lemma to show that this problem is in  $NP$  requires that we produce a deterministic Turing machine verifier. Consider  $\mathcal{V}$  below. Its first input  $\sigma$  is a Boolean expression while the second, the  $\omega$ , is a string that  $\mathcal{V}$  interprets as describing a line of  $\sigma$ ’s truth table.



If a candidate expression  $\sigma$  is satisfiable then there is a suitable witness, a line from

the truth table, so that  $\mathcal{V}$  can check that the named line gives a result of  $T$ . As an example, for the expression  $(*)$  above, take  $\omega = \text{TTF}$ . Clearly the verifier can do the checking in polytime. On the other hand, if a candidate  $\sigma$  is not satisfiable, as with the expression  $\sigma = P \wedge \neg P$ , then no  $\omega$  will cause  $\mathcal{V}$  to accept.

Before the next example, a few comments.

The most striking thing about the definition is that while it says that ‘there exists’ a witness  $\omega$ , it does not say where the witness comes from. A person with a computational mindset may well ask, “but how will we calculate the  $\omega$ ’s?” The point is not how to find them. The point is whether there is a deterministic Turing machine  $\mathcal{V}$  that can leverage a given hint  $\omega$  to verify in polytime that  $\sigma \in \mathcal{L}$ . That is, we don’t find the  $\omega$ ’s, we just use them.

Second, if  $\sigma \notin \mathcal{L}$  then the definition does not require a witness to that. Instead, what’s required is that from among all possible strings  $\omega$  there is none such that the verifier accepts  $\langle \sigma, \omega \rangle$ .<sup>†</sup>

The third comment relates to this asymmetry. Imagine that a demon hands you some papers and claims that they contain an unbeatable strategy for chess. Verifying requires stepping through the responses to each move, and responses to the responses, etc., and so there is lots of branching. But this problem is different than the kind we have been studying. To prove that the strategy is unbeatable we appear to have to check all of the branches, not just find one good one. It seems that a deterministic verifier must take exponential time. That would make the demon’s papers, in a sense, useless. So chess strategy is a kind of complement to the problems that we have been considering.

Also reflecting this asymmetry, it is not clear that  $\mathcal{L} \in \text{NP}$  implies that its complement  $\mathcal{L}^c$  is a member of NP. Consider Satisfiability. If a propositional logic expression  $\sigma$  is satisfiable then a witness to that is a pointer to a line of the truth table. But for non-satisfiability there is no natural witness; instead, the natural thing is to check all lines. As far as we know today, verifying that a Boolean formula is not satisfiable takes more than polytime. Consequently, where the complexity class co-NP contains the complements of languages from NP, we suspect that  $\text{NP} \neq \text{co-NP}$ .

Thus, the lemma explains something about the class NP: while P contains problems where we can find the answer in polytime, NP contains the problems whose answers are useful in that we can at least verify them in polytime.

Finally, the lemma requires that the runtime of the verifier is polynomial in  $|\sigma|$ , not polynomial in the length of its input  $\langle \sigma, \omega \rangle$ . If it said that latter then we could check the chess strategy just by using a witness that is exponentially long, and that consequently makes  $\langle \sigma, \omega \rangle$  exponentially long. Also observe that because  $\mathcal{V}$  runs in time polynomial in  $|\sigma|$ , for it to accept there must exist a witness whose length is polynomial in  $|\sigma|$ , because with  $\omega$ ’s that are too long the verifier cannot even input them before its runtime bound expires.

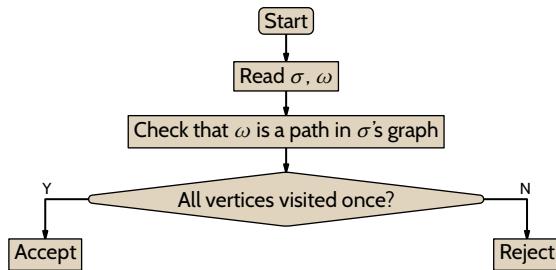
---

<sup>†</sup>With this in mind, perhaps a better term for  $\omega$  is “potential witness” or “proposed witness.” But those are not standard terms.

- 5.11 EXAMPLE The **Hamiltonian Path** problem is like the Hamiltonian Circuit problem except that instead of requiring that the starting vertex equals the ending one, it inputs two vertices. It is the problem of determining membership in this set.

$$\mathcal{L} = \{ \langle \mathcal{G}, v, \hat{v} \rangle \mid \text{path in } \mathcal{G} \text{ between } v \text{ and } \hat{v} \text{ visits every vertex exactly once} \}$$

We will show that this problem is in the class NP. We must produce a deterministic Turing machine verifier  $\mathcal{V}$ . It is sketched below. It takes as input  $\langle \sigma, \omega \rangle$ , where the candidate for membership in  $\mathcal{L}$  is  $\sigma = \langle \mathcal{G}, v, \hat{v} \rangle$ . The verifier interprets the witness to be a path,  $\omega = \langle v, v_1, \dots, \hat{v} \rangle$ .



If there is a Hamiltonian path then there exists a witness  $\omega$ , 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  $\sigma$  has no Hamiltonian path then for no  $\omega$  will  $\mathcal{V}$  be able to verify that  $\omega$  is such a path, and thus it will not accept any input pair starting with  $\sigma$ .

- 5.12 EXAMPLE The **Primality** problem asks whether a given number has a nontrivial factor.

$$\mathcal{L} = \{ n \in \mathbb{N}^+ \mid n \text{ has a divisor } a \text{ with } 1 < a < n \}$$

To show that  $\mathcal{L} \in \text{NP}$  we construct a verifier. It inputs  $\langle \sigma, \omega \rangle$  where  $\sigma$  represents a number  $n \in \mathbb{N}^+$ . For the witness  $\omega$  we can use a string that  $\mathcal{V}$  interprets as a number  $a$ , and then it tests whether  $1 < a < n$  and whether  $a$  divides  $n$ . If  $\sigma \in \mathcal{L}$  then there is a suitable such witness, while if  $\sigma \notin \mathcal{L}$  then no  $\omega$  will make  $\mathcal{V}$  accept the input. We can write this  $\mathcal{V}$  to run in polytime. Therefore  $\mathcal{L} \in \text{NP}$ .

We are ready now for the promised proof of Lemma 5.9.

*Proof* Suppose first that the language  $\mathcal{L}$  is accepted by a nondeterministic Turing machine  $\mathcal{P}$  in polytime. We will construct a deterministic verifier  $\mathcal{V}$  that runs in polytime. Let  $p: \mathbb{N} \rightarrow \mathbb{N}$  be the polynomial such that for any input  $\sigma \in \mathcal{L}$ , the machine  $\mathcal{P}$  has an accepting branch of length at most  $p(|\sigma|)$ . Use that branch to make a witness to  $\sigma$ 's acceptance, for instance as in the sequence  $\omega = \langle 3, 2, \dots \rangle$  meaning, “At the first node take the third child, then take the second child of that, etc.” Restated,  $\mathcal{P}$  has a finite number of states  $k$  so we can represent the accepting branch of its computation tree with a sequence  $\omega$  of at most  $p(|\sigma|)$  many numbers, each less than  $k$ . In total,  $\omega$ 's length is polynomial in  $|\sigma|$ . With this  $\omega$ , a deterministic machine  $\mathcal{V}$  can verify  $\mathcal{P}$ 's acceptance of  $\sigma$ , 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  $\tau$  if and only if  $\tau \in \mathcal{L}$ .

The key is that  $\hat{\mathcal{P}}$  is nondeterministic. Given a candidate  $\tau$  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  $\tau$ , while if  $\mathcal{V}$  does not accept then  $\hat{\mathcal{P}}$  rejects  $\tau$ .

We must check that  $\hat{\mathcal{P}}$  accepts  $\tau$  if and only if  $\tau \in \mathcal{L}$ . A nondeterministic machine accepts a string if there exists a branch that accepts the string, and rejects the string if every branch rejects it. Suppose first that  $\tau \in \mathcal{L}$ . Because  $\hat{\mathcal{V}}$  is a verifier, in this case there exists a witness  $\hat{\omega}$  (of length less than  $q(|\tau|)$ ) that will result in  $\hat{\mathcal{V}}$  accepting  $\langle \tau, \hat{\omega} \rangle$ , so there is a way for the prior paragraph to result in acceptance of  $\tau$ , and so  $\hat{\mathcal{P}}$  accepts  $\tau$ . 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  $\tau$ .  $\square$

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 everyday experience versus abstraction—is common and reasonable, so we will address it.

The first response is purely formal. Definition 5.8, as written, states that the candidate  $\tau$  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 in hardware.) When Turing formulated his definition there were no physical computers but they appeared soon after; will we someday have nondeterministic devices? Putting aside as too exotic proposals that involve things like time travel through wormholes, we don’t know of any candidates.<sup>†</sup> But that doesn’t mean that thinking about them is a purely academic exercise.<sup>‡</sup> The model of nondeterministic Turing machines has proven to be very fruitful.

---

<sup>†</sup>In order here is a caution about the machine types that seem likely to be coming, quantum computers. Well-established physical theory says that subatomic particles can be in a superposition of many states at once. Naively, it may seem that because of this essentially unbounded multi-way branching, if we could manipulate these particles then we would have nondeterministic computation. But, that we know of, this is false. That we know of, to get information out of a quantum system we must use interference. (Some popularizations wrongly suggest that quantum computers can try all potential solutions in parallel. That is, they miss the point about interference.) <sup>‡</sup>Not that there is anything wrong with academic exercises.

As evidence of that, the problems that are associated with these machines, the members of **NP**, are eminently practical, as witnessed by the fact that computer scientists have been trying to find fast solutions to many members of this class since computers have existed. For another, Lemma 5.9 rephrases questions about nondeterministic machines as questions about deterministic ones, the verifiers.

We close with a reflection. In this section we have defined the class of problems **NP** for which there is a good way to verify the solution, in contrast with the problems in **P** for which there is a good way to find the solution. Just as computably enumerable sets seem to be the limit of what can be in theory be known, polytime verification seems to be the limit of what can feasibly be done.

In the next section we will consider whether the two classes **P** and **NP** differ.

## V.5 Exercises

- ✓ 5.13 Your study partner asks, “In Lemma 5.9, since the witness  $\omega$  is not required to be effectively computable, why can’t I just take it to be the bit 1 if  $\sigma \in \mathcal{L}$ , and 0 if not? Then writing the verifier is easy: just ignore  $\sigma$  and follow the bit.” They are confused. Straighten them out.
- 5.14 Which is the negation of ‘at least one branch accepts’?
  - (A) Every branch accepts.
  - (B) At least one branch rejects.
  - (C) Every branch rejects.
  - (D) At least one branch fails to reject.
  - (E) None of these.
- ✓ 5.15 Decide if it is satisfiable. (A)  $(P \wedge Q) \vee (\neg Q \wedge R)$  (B)  $(P \rightarrow Q) \wedge \neg((P \wedge Q) \vee \neg P)$
- 5.16 True or false? If a language is in **P** then it is in **NP**.
- 5.17 Uh-oh. You find yourself with a nondeterministic Turing machine where on input  $\sigma$ , one branch of the computation tree accepts and one rejects. Some branches don’t halt at all. What is the upshot?
- ✓ 5.18 You get an exercise, *Write a nondeterministic algorithm that inputs a maze and outputs 1 if there is a path from the start to the end.*
  - (A) You hand in an algorithm that does backtracking to find any possible solution. Your professor sends it back, and says to try again. What was wrong?
  - (B) You hand in an algorithm that, each time it comes to a fork in the maze, chooses at random which way to go. Again you get it back with a note to work out another try. What is wrong with this one?
  - (C) Give a right answer.
- 5.19 Sketch a nondeterministic algorithm to search an unordered array of numbers for the number  $k$ . Describe it both in terms of unbounded parallelism and in terms of guessing.
- 5.20 A **simple substitution cipher** encrypts text by substituting one letter for another. Start by fixing a permutation of the letters, for example  $\langle W, P, \dots \rangle$

Then the cipher is that any A is replaced by a W, any B is replaced by a P, etc. Sketch three algorithms for decoding a substitution cipher (assume that you can recognize a correctly decoded string): (A) one that is deterministic, (B) one expressed in terms of unbounded parallelism, and (c) one expressed in terms of guessing.

- ✓ 5.21 Outline a nondeterministic algorithm that inputs a finite planar graph and outputs Yes if and only if the graph has a four-coloring. Describe it both in terms of unbounded parallelism and in terms of a demon providing a witness.
- 5.22 The **Linear Programming** problem is described on page 285. The related problem **Integer Linear Programming** also seeks to maximize a linear objective function  $F(x_0, \dots, x_n) = d_0x_0 + \dots + d_nx_n$  subject to linear constraints  $a_{i,0}x_0 + \dots + a_{i,n}x_n \leq b_i$  (or  $\geq b_i$ ), but with the restriction that all of  $x_j$ ,  $d_j$ ,  $b_j$ , and  $a_{i,j}$  are integers. Recast it as a family of language decision problems. Sketch a nondeterministic algorithm, giving both an unbounded parallelism formulation and a guessing formulation.
- ✓ 5.23 The **Semiprime** problem inputs a number  $n \in \mathbb{N}$  and decides if its prime factorization has exactly two primes,  $n = p_0^{e_0}p_1^{e_1}$  where  $e_i > 0$ . State it as a language decision problem. Sketch a nondeterministic algorithm that runs in polytime. Give both an unbounded parallelism formulation and a guessing formulation.
- 5.24 For each, give a language so that it is a language decision problem. Then give a polytime nondeterministic algorithm. State it in terms of guessing.
  - (A) **Three Dimensional Matching:** where  $X, Y, Z$  are sets of integers having  $n$  elements, given as input a set of triples  $M \subseteq X \times Y \times Z$ , decide if there is an  $n$ -element subset  $\hat{M} \subseteq M$  so that no two triples agree on their first coordinates, or second, or third.
  - (B) **Partition:** given a finite multiset  $A$  of natural numbers, decide if  $A$  splits into multisets  $\hat{A}, A - \hat{A}$  so the elements total to the same number,  $\sum_{a \in \hat{A}} a = \sum_{a \notin \hat{A}} a$ .
- 5.25 Sketch a nondeterministic algorithm that inputs a planar graph and a bound  $B \in \mathbb{N}$  and decides whether the graph is  $B$ -colorable, described both in terms of unbounded parallelism and also in terms of guessing.
- ✓ 5.26 For each problem, cast it as a language decision problem and then prove that it is in **NP** by filling in the blanks in this argument.

*Lemma 5.9 requires that we produce a deterministic Turing machine verifier,  $\mathcal{V}$ . It must input pairs of the form  $\langle \sigma, \omega \rangle$ , where  $\sigma$  is (1). It must have the property that if  $\sigma \in \mathcal{L}$  then there is an  $\omega$  such that  $\mathcal{V}$  accepts the input, while if  $\sigma \notin \mathcal{L}$  then there is no such witness  $\omega$ . And it must run in time polynomial in  $|\sigma|$ .*

*The verifier interprets the bitstring witness  $\omega$  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  $\omega$  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  $\omega$  will cause  $\mathcal{V}$  to accept the input pair.*

- (A) The **Double-SAT** problem inputs a propositional logic statement and decides whether it has at least two different substitutions of Boolean values that make it true.
- (B) The **Subset Sum** problem inputs a set of numbers  $S \subset \mathbb{N}$  and a target sum  $T \in \mathbb{N}$ , and decides whether least one subset of  $S$  adds to  $T$ .
- ✓ 5.27 In the game show *Countdown*, players get a target integer  $T \in [100 .. 999]$  and six numbers from  $S = \{1, 2, \dots, 10, 25, 50, 75, 100\}$  (these can be repeated). They make an arithmetic expression that evaluates to the target, using each given number at most once. The expression can use addition, subtraction, multiplication, and division without remainder. Show that the decision problem for  $\text{CD} = \{\langle s_0, \dots, s_5, T \rangle \in S^6 \times I \mid \text{a combination of the } s_i \text{ gives } T\}$  is in **NP**.
- ✓ 5.28 Recall that we recast **Traveling Salesman** optimization problem as a language decision problem for a family of languages. Show that each such language is in **NP** by applying Lemma 5.9, sketching a verifier that works with a suitable witness.
- 5.29 The problem of **Independent Sets** starts with a graph and a natural number  $n$  and decides whether in the graph there are  $n$ -many independent vertices, that is, vertices that are not connected. State it as a language decision problem, and use Lemma 5.9 to show that this problem is in **NP**.
- ✓ 5.30 Use Lemma 5.9 to show that the **Knapsack** problem is in **NP**.
- 5.31 True or false? For the language  $\{\langle a, b, c \rangle \in \mathbb{N}^3 \mid a + b = c\}$ , the problem of deciding membership is in **NP**.
- ✓ 5.32 The **Longest Path** problem is to input a graph and a bound,  $\langle G, B \rangle$ , and determine whether the graph contains a simple path of length at least  $B \in \mathbb{N}$ . (A path is simple if no two of its vertices are equal). Show that this is in **NP**.
- 5.33 Recast each as a language decision problem and then show it is in **NP**.
- (A) The **Linear Divisibility** problem inputs a pair of natural numbers  $\sigma = \langle a, b \rangle$  and asks if there is an  $x \in \mathbb{N}$  with  $ax + 1 = b$ .
- (B) Given  $n$  points scattered on a line, how far they are from each other defines a multiset. (Recall that a multiset is like a set but element repeats don't collapse.) The reverse of this problem, starting with a multiset  $M$  of numbers and deciding whether there exist a set of points on a line whose pairwise distances defines  $M$ , is the **Turnpike** problem.
- 5.34 Is **NP** countable or uncountable?
- ✓ 5.35 Show that this problem is in **NP**. A company has two delivery trucks. They work with a weighted graph called the ‘road map’. (Some vertex is distinguished as the start/finish.) Each morning the company gets a set of vertices,  $V$ . They must decide if there are two cycles such that every vertex in  $V$  is on at least one of the two cycles, and each cycle has length at most  $B \in \mathbb{N}$ .
- ✓ 5.36 Two graphs  $G_0, G_1$  are isomorphic if there is a one-to-one and onto function  $f: N_0 \rightarrow N_1$  such that  $\{v, \hat{v}\}$  is an edge of  $G_0$  if and only if  $\{f(v), f(\hat{v})\}$  is

an edge of  $\mathcal{G}_1$ . Consider the problem of computing whether two graphs are isomorphic.

- (A) Define the appropriate language.
- (B) Show that the problem of determining membership in that language is in NP.

5.37 The definition of when a nondeterministic machine decides a language, Definition 5.2, requires that every branch in the computation tree is finite. For language recognition we drop that. We say that a nondeterministic Turing machine  $P$  **recognizes** a language  $\mathcal{L}$  when if  $\sigma \in \mathcal{L}$  then there is at least one branch in the computation tree that accepts  $\sigma$ , while if  $\sigma \notin \mathcal{L}$  then no branch in the computation tree accepts (some branches may fail to accept because they are infinite). Show that if there is a nondeterministic machine that recognizes a language then there is a deterministic machine that also recognizes it.

5.38 Following the definition of Turing machine, on page 8, we gave a formal description of how these machines act. We did the same for Finite State machines on page 186, and for nondeterministic Finite State machines on page 194. Give a formal description of the action of a nondeterministic Turing machine.

- 5.39 (A) Show that the Halting problem is not in NP.
- (B) What is wrong with this reasoning? The Halting problem is in NP because given  $\langle \mathcal{P}, x \rangle$ , we can take as the witness  $\omega$  a number of steps for  $\mathcal{P}$  to halt on input  $x$ . If it halts in that number of steps then the verifier accepts, and if not then the verifier rejects.

## SECTION

### V.6 Reductions between problems

When we studied incomputability we considered a sense in which some problems are harder than others. Recall the Halts on Three problem, to decide membership in the set  $S = \{e \in \mathbb{N} \mid \phi_e(3) \downarrow\}$ . We showed that if we could solve this problem then we could solve the Halting problem and we denoted this with  $K \leq_T S$ . In general, a set  $B$  Turing reduces to  $A$ , written  $B \leq_T A$ , if there is a Turing machine that computes  $B$  from an  $A$  oracle, so that  $\phi_e^A = \mathbb{1}_B$ . Said another way, we can answer questions about membership in  $B$  by being given access to a routine that answers questions about membership in  $A$ .

In general in Mathematics we say that a problem  $B$  ‘reduces to’  $A$  when in order to solve  $B$ , it suffices to solve  $A$ . An example is that the problem of finding the maximum of a list of numbers reduces to the problem of sorting that list. The intuition is that  $A$  is at least as hard as  $B$ .<sup>†</sup>

---

<sup>†</sup>The phrase ‘reduces to’ can be confusing and often people get it the wrong way around. Perhaps they are misled by ‘ $B \leq_T A$ ’ into thinking that with  $B$  on the symbol’s low end, it must be reduced-to. But where for example  $A$  is the *Entscheidungsproblem* of answering all questions in Mathematics and  $B$  is Goldbach’s conjecture, the right terminology is that  $B$  reduces to  $A$  because a solution for  $A$  gives one for  $B$  as a side effect.

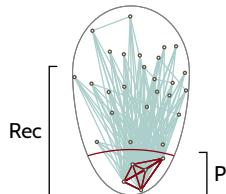
Accordingly, there are reductions between problems other than Turing reductions. Consider these two, where the second is a translation or alternate presentation of the other:  $S = \{e \mid \phi_e(3) \downarrow\}$  and  $T = \{e^2 + 1 \mid \phi_e(3) \downarrow\}$ . We can compute answers to questions about  $T$  from answers about  $S$  — it is as simple as  $x \in T$  iff the square root of  $x - 1$  is a member of  $S$ . We say that  $B$  is **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  $f(x) \in A$  (and thus for the above two,  $T \leq_m S$ ).

The difference between the two reducibilities is that with  $B \leq_T A$ , we answer questions about membership in  $B$  by asking a number of questions of an  $A$  oracle and then possibly doing more processing with that information. But with  $\leq_m$  we make only one oracle call and the reduction returns that call's result. So many-one reductions are a special case of Turing reductions, meaning that if  $B \leq_m A$  then  $B \leq_T A$ , but the converse does not hold; see Exercise 6.32.

This chapter focuses on efficient use of resources so it is natural to adapt Turing reduction and define a **Cook reduction** or **polytime Turing reduction**, denoted  $B \leq_T^P A$ , if there is an oracle Turing machine giving  $\phi_e^A = \mathbb{1}_B$  that runs in polytime. However, the more common reduction between problems results when we adapt many-one reduction by restricting the translation function to run in polytime.

**6.1 DEFINITION** Let  $\mathcal{L}_0, \mathcal{L}_1$  be languages, subsets of  $\mathbb{B}^*$ . Then  $\mathcal{L}_1$  is **polynomial time reducible** to  $\mathcal{L}_0$ , or **Karp reducible**, or **polynomial time mapping reducible**, or **polynomial time many-one reducible**, written  $\mathcal{L}_1 \leq_p \mathcal{L}_0$  or  $\mathcal{L}_1 \leq_m^P \mathcal{L}_0$ , if there is a **reduction function** or **transformation function**  $f: \mathbb{B}^* \rightarrow \mathbb{B}^*$  that is polynomial time computable and such that  $\sigma \in \mathcal{L}_1$  if and only if  $f(\sigma) \in \mathcal{L}_0$ .<sup>†</sup>

If there is a polytime reduction  $\mathcal{L}_1 \leq_p \mathcal{L}_0$  then a fast way to determine membership in  $\mathcal{L}_0$  gives a fast way to determine membership in  $\mathcal{L}_1$ .



**6.2 FIGURE:** This is the collection of all problems,  $\mathcal{L} \in \mathcal{P}(\mathbb{B}^*)$ , with a few shown as dots.

Ones with fast algorithms are at the bottom. Problems are connected if there is a polytime reduction from one to the other. Highlighted are connections within  $P$ .

<sup>†</sup>If between two problems there is a Karp reduction then there is a Cook reduction also but the converse does not hold. In particular, under Cook reduction a problem and its complement are equivalent but that's not true under Karp reduction. An example is that the complexity classes NP and co-NP (the problems whose complement is in NP) are equal under Cook reduction but appear to be separate under Karp reduction. Since any Karp reduction result is automatically a Cook reduction result, Karp reduction gives a more fine-grained partition of the problems and so while it may be a little less natural, it is the right one for us to study. Besides, Karp reduction is the standard in computational complexity.

- 6.3 EXAMPLE The Shortest Path problem inputs a weighted graph, two vertices, and a bound, and decides if there is path between the vertices of length less than the bound.

$$\mathcal{L}_0 = \{ \langle \mathcal{G}, v_0, v_1, B \rangle \mid \text{there is path from } v_0 \text{ to } v_1 \text{ of length less than } B \}$$

The Vertex-to-Vertex Path problem inputs an unweighted graph and two vertices, and decides if there is a path between the two.

$$\mathcal{L}_1 = \{ \langle \mathcal{H}, w_0, w_1 \rangle \mid \text{there is path between } w_0 \text{ and } w_1 \}$$

We will prove that  $\mathcal{L}_1 \leq_p \mathcal{L}_0$ . To decide whether some  $\langle \mathcal{H}, w_0, w_1 \rangle$  is a member of  $\mathcal{L}_1$ , we will translate that question into one of membership in  $\mathcal{L}_0$ . From  $\mathcal{H}$ , make a weighted graph  $\mathcal{G}$  by giving all its edges weight 1. Then  $\langle \mathcal{H}, w_0, w_1 \rangle \in \mathcal{L}_1$  if and only if  $f(\mathcal{H}) = \langle \mathcal{G}, w_0, w_1, |\mathcal{G}| \rangle \in \mathcal{L}_0$ . Clearly  $f$  can be done in polytime.

We gave the intuition that there is a polynomial reduction when one problem is a translation of the other, or at least a translation of a special case. The next example expands on that in a case where the problems initially don't seem related—one is a graph problem and the other is not—to give some insight into how to see such a relationship.

- 6.4 EXAMPLE The Clique problem is the decision problem for the language  $\mathcal{L} = \{ \langle \mathcal{G}, B \rangle \mid \mathcal{G} \text{ has a clique with } B \text{ vertices} \}$ . We will sketch that Satisfiability  $\leq_p$  Clique, so intuitively, Clique is at least as hard as Satisfiability.

Consider how to satisfy this Boolean expression  $E$ .

$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2)$$

The  $\wedge$ 's mean that the statement as a whole is  $T$  if and only if all of its clauses are  $T$ . The  $\vee$ 's mean that each clause is  $T$  if and only if any of its literals is  $T$ . So to satisfy the expression, select a literal from each clause and assign it the value  $T$ . For example, we can make  $E$  be  $T$  by selecting  $x_0$  from the first clause,  $x_2$  from the second, and  $\neg x_1$  from the third, and making them  $T$ . Similarly, if  $\neg x_1$  from the first and third clauses and  $x_3$  from the second are  $T$  then  $E$  is  $T$ . What we cannot do is pick  $x_2$  from the first and second and then  $\neg x_2$  from the third, because we cannot set both of these literals to be  $T$ .

That is, we can think of Satisfiability as a combinatorial problem. The clauses are like buckets and we select one thing from each bucket, subject to the constraint that the things we select must be pairwise compatible.

This view of Satisfiability gives a binary relation ‘can be compatibly picked’ between the literals in  $E$ 's clauses. So let  $\mathcal{G}_E$  be a graph whose vertices are pairs  $\langle c, \ell \rangle$  where  $c$  is the number of a clause and  $\ell$  is a literal in that clause. Two vertices  $v_0 = \langle c_0, \ell_0 \rangle$  and  $v_1 = \langle c_1, \ell_1 \rangle$  are connected by an edge if they come from different clauses,  $c_0 \neq c_1$ , and the literals are not negations of each other,  $\ell_0 \neq \neg \ell_1$ .

6.5 ANIMATION: Compatibility graph associated with  $E$ .

A choice of three mutually compatible vertices makes  $E$  be  $T$ . That is, the 3 clause expression  $E$  is satisfiable if and only if  $\mathcal{G}_E$  has a 3-clique.

More formally, the reduction function  $f$  inputs a propositional logic expression  $E$  and outputs a pair  $f(E) = \langle \mathcal{G}_E, B \rangle$  where  $\mathcal{G}_E$  is the compatibility graph associated with  $E$  defined in the prior paragraph and where  $B$  is the number of clauses in  $E$ . Then  $E \in SAT$  if and only if  $f(E) \in \mathcal{L}$ . Clearly this function can be computed in polytime.

- 6.6 EXAMPLE 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 a graph.

We will illustrate that the Graph  $k$ -Colorability problem reduces to the Satisfiability problem, Graph  $k$ -Colorability  $\leq_p$  Satisfiability, by focusing on the  $k = 3$  construction. (Larger  $k$ 's work much the same way but the  $k = 2$  case is somewhat different.)

Denote the set of 3-colorable graphs as  $\mathcal{L}_1$  and denote the set of satisfiable propositional logic statements as  $\mathcal{L}_0$ . To show that  $\mathcal{L}_1 \leq_p \mathcal{L}_0$  we must produce a translation function  $f$ . It inputs a graph  $\mathcal{G}$  and outputs a propositional logic expression  $E = f(\mathcal{G})$  such that the graph is 3-colorable if and only if the expression is satisfiable.

The function builds  $E$  by including clauses that state, in the language of propositional logic, the constraints to be met for the graph to be 3-colorable. Let  $\mathcal{G}$  have vertices  $\{v_0, \dots, v_{n-1}\}$ . The expression  $E$  will have  $3n$ -many Boolean variables:  $a_0, \dots, a_{n-1}$ , and  $b_0, \dots, b_{n-1}$ , and  $c_0, \dots, c_{n-1}$ . The idea is that if the  $i$ -th vertex  $v_i$  gets the first color then  $E$  will be satisfied when the associated variables have

$a_i = T, b_i = F, c_i = F$ , while if  $v_i$  gets the second color then  $a_i = F, b_i = T, c_i = F$ , and if  $v_i$  gets the third color then  $a_i = F, b_i = F, c_i = T$ .

For vertex  $v_i$ , the function gives the expression a clause saying that the vertex gets at least one color.

$$(a_i \vee b_i \vee c_i)$$

For each edge  $\{v_i, v_j\}$ , the function adds three clauses which together ensure that the edge does not connect two same-color vertices.

$$(\neg a_i \vee \neg a_j) \quad (\neg b_i \vee \neg b_j) \quad (\neg c_i \vee \neg c_j)$$

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 its remaining lines have the other kind of clauses.

$$\begin{array}{c} (a_0 \vee b_0 \vee c_0) \wedge (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge (a_3 \vee b_3 \vee c_3) \\ \wedge (\neg a_0 \vee \neg a_1) \wedge (\neg b_0 \vee \neg b_1) \wedge (\neg c_0 \vee \neg c_1) \\ \wedge (\neg a_0 \vee \neg a_3) \wedge (\neg b_0 \vee \neg b_3) \wedge (\neg c_0 \vee \neg c_3) \\ \wedge (\neg a_1 \vee \neg a_2) \wedge (\neg b_1 \vee \neg b_2) \wedge (\neg c_1 \vee \neg c_2) \\ \wedge (\neg a_1 \vee \neg a_3) \wedge (\neg b_1 \vee \neg b_3) \wedge (\neg c_1 \vee \neg c_3) \end{array}$$

By construction, there is an assignments of truth values for the variables to satisfy the expression if and only if the graph has a 3-coloring.

Completing the argument requires checking that the translation function, which inputs a bitstring representation of the graph and outputs a bitstring representation of the expression, is polynomial. That's clear so we omit the details.

A translation function is a kind of compiler. A programming language compiler inputs descriptions from one domain, such as a Racket program, and outputs a corresponding statement from another domain, such an executable in the machine's native format. Similarly, the function  $f$  above translates problem instances in the domain of graphs to those in the domain of propositional logic.

- 6.8 EXAMPLE We will show that **Subset Sum**  $\leq_p$  **Knapsack**. The **Knapsack** problem starts with a multiset of objects  $U = \{u_0, \dots, u_{n-1}\}$  (a multiset is a set in which members can appear more than once; basically, a unordered list), a bound  $W \in \mathbb{N}$  and a target  $V \in \mathbb{N}$ , along with two functions,  $w, v: U \rightarrow \mathbb{N}^+$  that give each  $u_i$  a weight  $w(u_i)$  and a value  $v(u_i)$ . It asks for a subset  $A \subseteq U$  such that the weight total does not exceed the bound  $W$  while the value total is at least as big as the target  $V$ .

$$\mathcal{L}_0 = \{ \langle U, w, v, W, V \rangle \mid \text{some } A \subseteq U \text{ has } \sum_{a \in A} w(a) \leq W \text{ and } \sum_{a \in A} v(a) \geq V \}$$

The **Subset Sum** problem starts with a target  $T \in \mathbb{N}^+$  and a multiset  $S =$

$\{s_0, \dots s_{k-1}\} \subset \mathbb{N}$ . It asks for a subset whose elements add to the target.

$$\mathcal{L}_1 = \{\langle S, T \rangle \mid \text{some } B \subseteq S \text{ has } \sum_{b \in B} b = T\}$$

Besides being computable in polytime, 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  $f(\langle S, T \rangle) \in \mathcal{L}_0$ .

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 fix the weight and value targets as  $W = V = T = 126$ . Then  $\langle S, T \rangle \in \mathcal{L}_1$  iff  $\langle S, w, v, W, V \rangle \in \mathcal{L}_0$ . In this way, we can think of the **Subset Sum** problem as a special case of Knapsack.

More generally, let  $f$  take the input  $\langle S, T \rangle$  to the output  $\langle S, w, v, T, T \rangle$ , where the functions  $w$  and  $v$  are given by  $w(s_i) = v(s_i) = s_i$ . Clearly  $\langle S, T \rangle \in \mathcal{L}_0$  if and only if  $f(\langle S, T \rangle) \in \mathcal{L}_1$ . Also clearly  $f$  can be done in polytime.

We close with some basic facts about polytime reduction.

6.9 **LEMMA** Polytime reduction is reflexive:  $\mathcal{L} \leq_p \mathcal{L}$  for all languages. It is also transitive:  $\mathcal{L}_2 \leq_p \mathcal{L}_1$  and  $\mathcal{L}_1 \leq_p \mathcal{L}_0$  imply that  $\mathcal{L}_2 \leq_p \mathcal{L}_0$ . Every nontrivial computable language is **P hard**: for  $\mathcal{L}_1 \in \text{P}$ , every language  $\mathcal{L}_0$  with  $\mathcal{L}_0 \neq \emptyset$  and  $\mathcal{L}_0 \neq \mathbb{N}$  satisfies that  $\mathcal{L}_1 \leq_p \mathcal{L}_0$ . The class **P** is closed downward: if  $\mathcal{L}_0 \in \text{P}$  and  $\mathcal{L}_1 \leq_p \mathcal{L}_0$  then  $\mathcal{L}_1 \in \text{P}$ . So also is the class **NP**.

*Proof* The first two sentences and the final sentence are Exercise 6.33.

For the third sentence fix a  $\mathcal{L}_0$  that is nontrivial, so there is a member  $\sigma \in \mathcal{L}_0$  and a nonmember  $\tau \notin \mathcal{L}_0$ . Let  $\mathcal{L}_1$  be an element of **P**. We will specify a transformation function  $f$  giving  $\mathcal{L}_1 \leq_p \mathcal{L}_0$ . For any  $\alpha \in \mathbb{B}^*$ , computing whether  $\alpha \in \mathcal{L}_1$  can be done in polytime. If it is a member then define  $f(\alpha) = \sigma$  while if not then define  $f(\alpha) = \tau$ .

For the downward closure of **P**, suppose that  $\mathcal{L}_1 \leq_p \mathcal{L}_0$  via the polytime function  $g$  and also suppose that there is a polytime algorithm for determining membership in  $\mathcal{L}_0$ . Determine membership in  $\mathcal{L}_1$  by starting with an input  $\sigma$ , finding  $g(\sigma)$ , and applying the  $\mathcal{L}_0$  algorithm to settle whether  $g(\sigma) \in \mathcal{L}_0$ . Where the  $\mathcal{L}_0$  algorithm runs in time that is  $\mathcal{O}(n^i)$  and where  $g$  runs in time that is  $\mathcal{O}(n^j)$ , determining  $\mathcal{L}_1$  membership in this way runs in time that is  $\mathcal{O}(n^{\max(i,j)})$ , which is polynomial.  $\square$

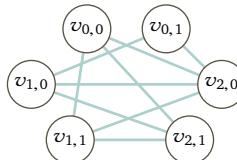
By the lemma, because Example 6.6 shows Graph Colorability  $\leq_p$  Satisfiability and Example 6.4 gives Satisfiability  $\leq_p$  Clique, we know that Graph Colorability  $\leq_p$  Clique.

To finish we note that the examples above give some sense of why the Satisfiability problem is convenient for reducibility. Often it is natural to describe the conditions in a problem with logical statements. The next section gives

a theorem saying that Satisfiability is a benchmark in that it is at least as hard as every problem in NP.

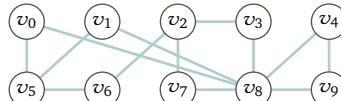
## V.6 Exercises

- 6.10 Suppose that  $\mathcal{L}_1 \leq_p \mathcal{L}_0$ . Which is true?
- A fast algorithm for  $\mathcal{L}_0$  would give a fast algorithm for  $\mathcal{L}_1$ .
  - A fast algorithm for  $\mathcal{L}_1$  would give a fast one for  $\mathcal{L}_0$ .
- 6.11 Suppose that  $\mathcal{L}_1 \leq_p \mathcal{L}_0$ . Which is the right way to use ‘reduces to’:  $\mathcal{L}_1$  reduces to  $\mathcal{L}_0$ , or  $\mathcal{L}_0$  reduces to  $\mathcal{L}_1$ ?
- ✓ 6.12 Show that if  $\mathcal{L}_0 \notin \text{P}$  and  $\mathcal{L}_0 \leq_p \mathcal{L}_1$  then  $\mathcal{L}_1 \notin \text{P}$  also. What about NP?
- 6.13 Prove that  $\mathcal{L} \leq_p \mathcal{L}^c$  if and only if  $\mathcal{L}^c \leq_p \mathcal{L}$ .
- 6.14 Your friend is bright but, temporarily, confused. “Lemma 6.9 says that every language in P is  $\leq_p$  every other nontrivial language. But there are uncountably many languages and only countably many f’s because they each come from some Turing machine. So I’m not seeing how there are enough reduction functions for a given language to be related to all others.” Un-confuse them.
- 6.15 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 Produce the compatibility graph for  $(x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_1) \wedge (x_0 \vee \neg x_1)$ .
- 6.17 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?



- ✓ 6.18 Suppose that the language  $A$  is polynomial time reducible to the language  $B$ ,  $A \leq_p B$ . Which of these are true?
- A tractable way to decide  $A$  can be used to tractably decide  $B$ .
  - If  $A$  is tractably decidable then  $B$  is tractably decidable also.
  - If  $A$  is not tractably decidable then  $B$  is not tractably decidable too.
- ✓ 6.19 The **Substring problem** inputs two strings and decides if the second is a substring of the first. The **Cyclic Shift problem** inputs two strings and decides if the second is a cyclic shift of the first. (Where  $\alpha = abcde$ , one cyclic shift is  $\beta = deabc$ . More precisely, if  $\alpha = a_0a_1 \dots a_{n-1}$  and  $\beta = b_0b_1 \dots b_{n-1}$  are length  $n$  strings, then  $\beta$  is a cyclic shift of  $\alpha$  when there is an index  $k \in \{0, \dots, n-1\}$  such that  $a_i = b_{(k+i) \bmod n}$  for all  $i < n$ .)
- Name three cyclic shifts of  $\alpha = 0110010$ .
  - Decide whether  $\beta = 101001101$  is a cyclic shift of  $\alpha = 001101101$ .
  - State the Substring problem as a language decision problem.

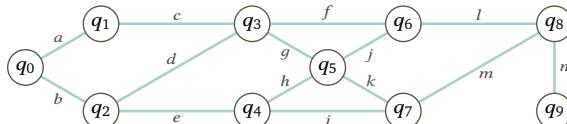
- (D) Also state the Cyclic Shift problem as a language decision problem.
- (E) Show that Cyclic Shift  $\leq_p$  Substring. Hint: for same length strings,  $\beta$  is a cyclic shift of  $\alpha$  if and only if  $\beta$  is a substring of  $\alpha \hat{\cdot} \alpha$ .
- ✓ 6.20 The Independent Set problem inputs a graph and a bound, and decides if there is a set of vertices, of size at least equal to the bound, that are not connected by any edge. The Vertex Cover problem inputs a graph and a bound and decides if there is a vertex set, of size less than or equal to the bound, such that every edge contains at least one vertex in the set.
- (A) State each as a language decision problem.
- (B) Consider this graph. Find a vertex cover with four elements.



- (C) In that graph find an independent set with six elements.
- (D) Show that in a graph,  $S$  is an independent set if and only if  $\mathcal{N} - S$  is a vertex cover, where  $\mathcal{N}$  is the set of vertices.
- (E) Conclude that Vertex Cover  $\leq_p$  Independent Set.
- (F) Also conclude that Independent Set  $\leq_p$  Vertex Cover.

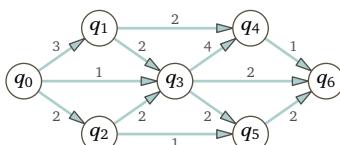
- 6.21 The Vertex Cover problem inputs a graph and a bound and decides if there is a vertex set, of size less than or equal to the bound, such that every edge contains at least one vertex in the set. The Set Cover problem inputs a set  $S$ , a collection of subsets  $S_0 \subseteq S, \dots, S_n \subseteq S$ , and a bound, and decides if there is a subcollection of the  $S_j$ , with a number of sets at most equal to the bound, whose union is  $S$ .

- (A) State each as a language decision problem.
- (B) Find a vertex cover for this graph.



- (C) Make a set  $S$  consisting of all of that graph's edges, and for each  $v$  make a subset  $S_v$  of the edges incident on that vertex. Find a set cover.
- (D) Show that Vertex Cover  $\leq_p$  Set Cover.

- ✓ 6.22 Show that Hamiltonian Circuit  $\leq_p$  Traveling Salesman. (A) State each as a language decision problem. (B) Produce the reduction function.
- ✓ 6.23 In this network, each edge is labeled with a capacity. (Imagine railroad lines going from  $q_0$  to  $q_6$ .)



The **Max-Flow** problem is to find the maximum total amount that can flow across the network, usually by using many paths at once. That is, we will find a **flow**  $F_{q_i, q_j}$  for each edge, subject to the constraints that the flow through an edge must not exceed its capacity and that the flow into a vertex must equal the flow out (except for the source  $q_0$  and the sink  $q_6$ ). The **Linear Programming** problem is described on page 285.

- (A) Express each as a language decision problem, remembering the technique of converting optimization problems using bounds.
- (B) By eye, find the maximum flow for the above network.
- (C) For each edge  $v_i v_j$ , define a variable  $x_{i,j}$ . Describe the constraints on that variable imposed by the edge's capacity. Also describe the constraints on the set of variables imposed by the limitation that for many vertices the flow in must equal the flow out. Finally, use the variables to give an expression to optimize in order to get maximum flow.
- (D) Show that **Max-Flow**  $\leq_p$  **Linear Programming**.

6.24 The **Max-Flow problem** inputs a directed graph where each edge is labeled with a capacity, and the task is to find the maximum amount that can flow from the source node to the sink node (for more, see Exercise 6.23). The **Drummer** problem starts with two same-sized sets, the rock bands,  $B$ , and potential drummers,  $D$ . Each band  $b \in B$  has a set  $S_b \subseteq D$  of drummers that they would agree to take on. The goal is to make the most number of matches.

- (A) Consider four bands  $B = \{b_0, b_1, b_2, b_3\}$  and drummers  $D = \{d_0, d_1, d_2, d_3\}$ . Band  $b_0$  likes drummers  $d_0$  and  $d_2$ . Band  $b_1$  likes only drummer  $d_1$ , and  $b_2$  also likes only  $d_1$ . Band  $b_3$  likes the sound of both  $d_2$  and  $d_3$ . What is the largest number of matches?
- (B) Express each as a language decision problem.
- (C) Draw a graph with the bands on the left and the drummers on the right. Make an arrow from a band to a drummer if there is a connection. Now add a source and a sink node to make a flow diagram.
- (D) Show that **Drummer**  $\leq_p$  **Max-Flow**.

6.25 The **3-Satisfiability** problem is to decide the satisfiability of CNF propositional logic expressions where every clause has at most three literals (see page 281). The **Strict 3-Satisfiability problem** requires that each clause has exactly three unequal literals. We will show that the two are inter-reducible.

- (A) Show the easy half, that **Strict 3-Satisfiability**  $\leq_p$  **3-Satisfiability**.
- (B) Also show that we can go from clauses with two literals to clauses with three by introducing an irrelevant variable:  $P \vee Q$  is equivalent to  $(P \vee Q \vee R) \wedge (P \vee Q \vee \neg R)$ . Along the same lines, show that  $P$  is equivalent to  $(P \vee Q \vee R) \wedge (P \vee \neg Q \vee R) \wedge (P \vee Q \vee \neg R) \wedge (P \vee \neg Q \vee \neg R)$ .
- (C) Show **3-Satisfiability**  $\leq_p$  **Strict 3-Satisfiability**.

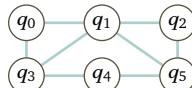
6.26 We will show that the **3-Satisfiability** problem, **3-SAT**, is inter-reducible with **SAT**. (The problem definitions are on page 281. Without loss of generality we

will assume that instances of  $SAT$  are in Conjunctive Normal form.)

- (A) Show the easy half, that  $3-SAT \leq_p SAT$ .
- (B) As a preliminary for the other reduction, show that the propositional logic implication  $P \rightarrow Q$  is equivalent to  $\neg P \vee Q$ .
- (C) To go from clauses with four literals to those with three, start with  $P \vee Q \vee R \vee S$ . Introduce a variable  $A$  such that  $A \leftrightarrow (P \vee Q)$ , that is,  $(A \rightarrow (P \vee Q)) \wedge (A \leftarrow (P \vee Q))$ . Show that  $(A \rightarrow (P \vee Q))$  is equivalent to  $(P \vee Q \vee \neg A)$ . Also verify that  $(P \vee Q) \rightarrow A$  is equivalent to  $(P \vee \neg Q \vee A) \wedge (\neg P \vee Q \vee A) \wedge (\neg P \vee \neg Q \vee A)$ . Conclude that  $P \vee Q \vee R \vee S$  is equivalent to  $(A \vee R \vee S) \wedge (P \vee Q \vee \neg A) \wedge (P \vee \neg Q \vee A) \wedge (\neg P \vee Q \vee A) \wedge (\neg P \vee \neg Q \vee A)$ .
- (D) For a five literal clause  $P \vee Q \vee R \vee S \vee X$ , find an equivalent propositional logic expression made of clauses having only three literals each.
- (E) Show that  $SAT \leq_p 3-SAT$ .

6.27 See page 281 for a definition of the 3-Satisfiability problem,  $3-SAT$ . The Independent Set problem inputs a graph and a bound, and decides if there is a set of vertices, of size at least equal to the bound, that are not connected by any edge.

- (A) In this graph, find an independent set with at least  $B = 3$  members.



- (B) State Independent Set as a language decision problem.
- (C) Decide if  $E = (P_0 \vee \neg P_1 \vee \neg P_2) \wedge (P_1 \vee P_2 \vee \neg P_3)$  is satisfiable.
- (D) State 3-Satisfiability as a language decision problem.
- (E) With the expression  $E$ , make a triangle for each of the two clauses, where the vertices of the first are labeled  $v_0$ ,  $\bar{v}_1$ , and  $\bar{v}_2$ , while the vertices of the second are  $w_1$ ,  $w_2$ , and  $\bar{w}_3$ . In addition to the edges forming the triangles, also put one connecting  $\bar{v}_1$  with  $w_1$ , and one connecting  $\bar{v}_2$  with  $w_2$ .
- (F) Sketch an argument that  $3\text{-Satisfiability} \leq_p \text{Independent Set}$ .

✓ 6.28 See page 281 for a definition of the 3-Satisfiability problem,  $3-SAT$ . The **Integer Linear Programming decision problem** starts with a list of linear inequalities with variables  $x_0, \dots, x_n$ , such as  $a_{i,0}x_0 + \dots + a_{i,n}x_n \leq b_i$  (or  $\geq b$ ), and it looks for a sequence  $\langle s_0, \dots, s_n \rangle$  that is feasible in that it satisfies all of the constraints, but with the restriction that the numbers must be integers,  $a_{i,j}, b_i, s_i \in \mathbb{Z}$ .

- (A) Consider the propositional logic clause  $P_0 \vee \neg P_1 \vee \neg P_2$ . Create variables  $z_0$ ,  $z_1$ , and  $z_2$  and list linear constraints such that each must be either 0 or 1. Also give a linear inequality that holds if and only if the clause is true.
- (B) Show that  $3\text{-Satisfiability} \leq_p \text{Integer Linear Programming}$ .

6.29 Consider the problem D of deciding whether a multivariable polynomial has any integer roots. That is, it is the language decision problem for this set.

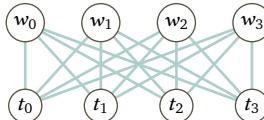
$$D = \{p \mid p \text{ is a polynomial and there is } \vec{c} \in \mathbb{Z}^n \text{ so that } p(\vec{c}) = 0\}$$

We will show that  $3\text{-SAT} \leq_p \mathsf{D}$  (page 281 has the definition of  $3\text{-SAT}$ ).

- (A) Argue that a one-clause disjunction has a value of  $T$  if and only if any of its literals has a value of  $T$ . For instance,  $E_0 = P_0 \vee \neg P_1$  is true if and only if  $P_0 = T$  or  $\neg P_1 = T$ .
  - (B) Associate  $E_0$  with the set  $S_{E_0} = \{x_0(1 - x_0), x_1(1 - x_1), x_0(1 - x_1)\}$  of three polynomials. Argue that all three have a value of 0 if and only if both variables have a value of either 0 or 1, and either  $x_0 = 0$  or  $x_1 = 1$ .
  - (C) For the expression  $E_1 = (P_0 \vee \neg P_1 \vee \neg P_2) \wedge (P_1 \vee P_2 \vee \neg P_3)$ , produce a set of polynomials  $S_{E_1}$  with the analogous properties.
  - (D) Combine the polynomials from  $S_{E_1}$  in the prior item into a single polynomial in such a way that it has an overall value of 0 if and only if all the members of  $S$  have a value of 0.
  - (E) Show that  $3\text{-SAT} \leq_p \mathsf{D}$ .
- ✓ 6.30 We can do reductions between problems of types other than language decision problems, including optimizations. The **Assignment problem** inputs two same-sized sets, of workers  $W = \{w_0, \dots, w_{n-1}\}$  and tasks  $T = \{t_0, \dots, t_{n-1}\}$ . For each worker-task pair there is a cost  $C(w_i, t_j)$ . The goal is to assign each of the tasks, one per worker, at minimal total cost. The **Traveling Salesman problem**, of course, asks for circuit of minimal total cost in a weighted graph.
- (A) By eye, solve this Assignment problem instance.

Cost $C(t_i, w_j)$		$w_0$	$w_1$	$w_2$	$w_3$
$t_0$	13	4	7	6	
$t_1$	1	11	5	4	
$t_2$	6	7	2	8	
$t_3$	1	3	5	9	

- (B) Consider this bipartite graph.

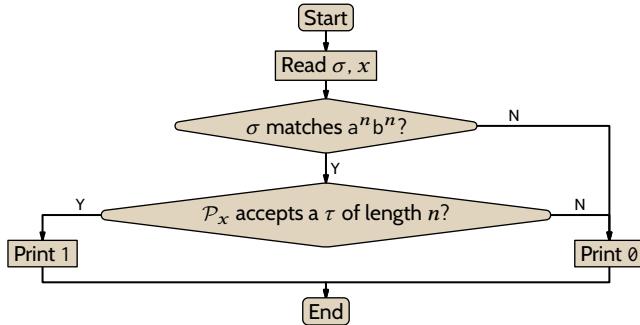


Each  $t_i$  is shown connected to each  $w_j$ . As edge weights, add the costs from the table. In addition, connect each pair of  $w$ 's with an edge of weight 0, and similarly connect each pair of  $t$ 's. Restate the Assignment problem as that of finding a circuit in this graph. Use this to show that Assignment  $\leq_p$  Traveling Salesman.

- 6.31 We will show that  $\text{Fin} \leq_p \text{Reg}$ , where they are the decision problems for the language  $R = \{x \in \mathbb{N} \mid \text{the language decided by } \mathcal{P}_x \text{ is regular}\}$  and also for the language  $F = \{i \in \mathbb{N} \mid \text{the language decided by } \mathcal{P}_i \text{ is finite}\}$  (this means that  $\mathcal{P}_i$  halts on all inputs and acts as the characteristic function of a set that is finite).

- (A) Adapt Example 5.2 from Chapter Four to show that any infinite subset of  $\{a^n b^n \mid n \in \mathbb{N}\}$  is not regular.

- (b) Argue that there is a Turing machine with the behavior below. Then apply the  $s\text{-}m\text{-}n$  lemma to parameterize  $x$ .



- (c) Using the prior item, produce the reduction function.

6.32 As discussed at the start of the section,  $B \leq_T A$  if there is an oracle  $A$  Turing machine that computes  $B$ , so that  $\phi_e^A = \mathbb{1}_B$ . And,  $B \leq_m A$  if there is a total computable function so that  $x \in B$  if and only if  $f(x) \in A$ . We will show that the two reductions  $\leq_T$  and  $\leq_m$  differ.

- (A) Let  $B$  be the nonempty computable set  $\{2n \mid n \in \mathbb{N}\}$  of even numbers and let  $A$  be the empty set. Show that  $B \leq_T A$  but it is not the case that  $B \leq_m A$ .
  - (B) Observe that  $A \leq_T A^c$  for all sets, and so  $K \leq_T K^c$ . Show that if  $B \leq_m A$  and  $B$  is computably enumerable then so is  $A$ , and conclude that  $K$  is not many-one reducible to its complement.
- ✓ 6.33 Lemma 6.9 leaves a couple of points undone. (A) Show that  $\leq_p$  is reflexive and transitive. (B) It says that nontrivial languages are P hard. What about trivial ones? Which languages reduce to the empty set? To  $\mathbb{B}^*$ ? (c) Show that NP is downward closed, that if  $\mathcal{L}_1 \leq_p \mathcal{L}_0$  and  $\mathcal{L}_0 \in \text{NP}$  then  $\mathcal{L}_1 \in \text{NP}$  also.
- 6.34 When  $\mathcal{L}_i \leq_p \mathcal{L}_j$ , does that mean that the best algorithm to decide  $\mathcal{L}_i$  takes time that is less than or equal to the amount taken by the best algorithm for  $\mathcal{L}_j$ ? Fix a language decision problem  $\mathcal{L}_0$  whose fastest algorithm is  $\mathcal{O}(n^3)$ , an  $\mathcal{L}_1$  whose best algorithm is  $\mathcal{O}(n^2)$ , a  $\mathcal{L}_2$  whose best is  $\mathcal{O}(2^n)$ , and a  $\mathcal{L}_3$  whose best is  $\mathcal{O}(\lg n)$ . In the array entry  $i, j$  below, put 'N' if  $\mathcal{L}_i \leq_p \mathcal{L}_j$  is not possible.

	$\mathcal{L}_0$	$\mathcal{L}_1$	$\mathcal{L}_2$	$\mathcal{L}_3$
$\mathcal{L}_0$	(0,0)	(0,1)	(0,2)	(0,3)
$\mathcal{L}_1$	(1,0)	(1,1)	(1,2)	(1,3)
$\mathcal{L}_2$	(2,0)	(2,1)	(2,2)	(2,3)
$\mathcal{L}_3$	(3,0)	(3,1)	(3,2)	(3,3)

6.35 Is there a connection between subset and polytime reducibility? Find languages  $\mathcal{L}_0, \mathcal{L}_1 \in \mathcal{P}(\mathbb{B}^*)$  for each: (A)  $\mathcal{L}_0 \subset \mathcal{L}_1$  and  $\mathcal{L}_0 \leq_p \mathcal{L}_1$ , (B)  $\mathcal{L}_0 \not\subset \mathcal{L}_1$  and  $\mathcal{L}_0 \leq_p \mathcal{L}_1$ , (C)  $\mathcal{L}_0 \subset \mathcal{L}_1$  and  $\mathcal{L}_0 \not\leq_p \mathcal{L}_1$ , (D)  $\mathcal{L}_0 \not\subset \mathcal{L}_1$  and  $\mathcal{L}_0 \not\leq_p \mathcal{L}_1$ .

## SECTION

## V.7 NP completeness

Because  $P \subseteq NP$ , the class **NP** contains lots of easy problems, ones with a fast algorithm. Nonetheless, the interest in the class is that it also contains lots of problems that seem to be hard. Can we prove that these problems are indeed hard?

This question was raised by S Cook in 1971. He noted that the idea of polynomial time reducibility gives us a way to make precise that an efficient solution for one problem implies an efficient solution for the other. He then showed that among the problems in **NP**, there are ones that are maximally hard. (This was also shown by L Levin but he was behind the Iron Curtain and knowledge of his work did not spread to the rest of the world for some time.)

Here, ‘maximally hard’ means that these are **NP** problems and they are at least as hard as any **NP** problem, in that if we could solve one of these then we could solve any **NP** problem at all.

**7.1 THEOREM (COOK-LEVIN THEOREM)** The Satisfiability problem is in **NP** and has the property any problem in **NP** reduces to it:  $\mathcal{L} \leq_p SAT$  for any  $\mathcal{L} \in NP$ .

First, we have already observed that  $SAT \in NP$  because, given a Boolean expression, we can use as a witness  $\omega$  an assignment of truth values that satisfies the expression.



Leonid Levin  
b 1948

Here is an outline of the proof’s other half. Given  $\mathcal{L} \in NP$ , we must show that  $\mathcal{L} \leq_p SAT$ . We produce a function  $f_{\mathcal{L}}$  that translates membership questions for  $\mathcal{L}$  into Boolean expressions, such that the membership answer is ‘yes’ if and only if the expression is satisfiable. What we know about  $\mathcal{L}$  is that its member  $\sigma$ ’s are accepted by a nondeterministic machine  $\mathcal{P}$  in time given by a polynomial  $q$ . With that, from  $\langle \mathcal{P}, \sigma, q \rangle$  the proof constructs a Boolean expression that yields true if and only if  $\mathcal{P}$  accepts  $\sigma$ . The Boolean expression encodes the constraints under which a Turing machine operates, such as that the only tape symbol that can be changed in the current step is the symbol under the machine’s head.

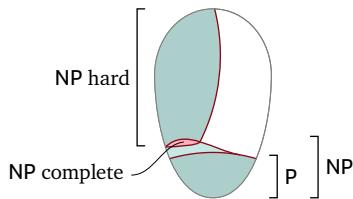
**7.2 DEFINITION** A problem is **NP hard** if every problem in **NP** reduces to it, that is,  $\mathcal{L}$  is **NP hard** if  $\hat{\mathcal{L}} \in NP$  implies that  $\hat{\mathcal{L}} \leq_p \mathcal{L}$ . A problem is **NP complete** if, in addition to being **NP hard**, it is also a member of **NP**.<sup>†</sup>

So a problem is **NP complete** if it is, in a sense, at least as hard as any member of **NP**. The sketch below illustrates.



Stephen Cook  
b 1939

<sup>†</sup>In general, for a complexity class  $C$ , a problem  $\mathcal{L}$  is **C hard** when all problems in that class reduce to it: if  $\hat{\mathcal{L}} \in C$  then  $\hat{\mathcal{L}} \leq_p \mathcal{L}$ . A problem is **C complete** if it is hard for that class and also is a member of that class.



7.3 FIGURE: The blob contains all problems. In the bottom is **NP**, drawn with **P** as a proper subset. The top has the **NP-hard** problems. The highlighted intersection is the set of **NP complete** problems.

The Cook-Levin Theorem says that there is at least one **NP** complete problem, namely  $\text{SAT}$ . In fact, we shall see that there are many such problems.

The **NP** complete problems are to the class **NP** as the problems Turing-equivalent to  $K$  are to the computably enumerable sets (recall that  $K$  is the solution to the Halting problem). They are at the top of their class—if we could solve the one problem then we could solve every other problem in that class.

7.4 LEMMA If  $\mathcal{L}_0$  is **NP** complete, and  $\mathcal{L}_0 \leq_p \mathcal{L}_1$ , and  $\mathcal{L}_1 \in \mathbf{NP}$  then  $\mathcal{L}_1$  is **NP** complete.

*Proof* Exercise 7.29. □

Soon after Cook raised the question of **NP** completeness, R Karp brought it to widespread attention. Karp noted that there are clusters of problems: there is a collection of problems solvable in time  $\mathcal{O}(\lg(n))$ , problems of time  $\mathcal{O}(n)$ , those of time  $\mathcal{O}(n \lg n)$ , etc. There is also a cluster of problems that seem much tougher. He gave a list of twenty one of these, drawn from Computer Science, Mathematics, and the natural sciences, where smart people had for years been unable find efficient algorithms. Karp showed that they are all **NP** complete, so if we could efficiently solve any then we could efficiently solve them all. Not every difficult problem is **NP** complete but many thousands of problems have been shown to be so and thus whatever it is that makes these problems hard, all of them share it.



Richard M Karp  
b 1935

Typically we prove that a problem is **NP** complete in two halves. First we show that it is in **NP** by exhibiting a witness  $\omega$  that a deterministic verifier can check in polytime. Second, we show that the problem is **NP** hard by showing that an **NP** complete problem reduces to it. These are the ones most often used. For instance, to show that some  $\mathcal{L}$  is **NP** hard, we might show that  $3\text{-SAT} \leq_p \mathcal{L}$ .

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  $B$ -vertex cover, a size  $B$  set of vertices  $C$  such that for any edge  $v_i v_j$ , at least one of its ends is a member of  $C$ .

**Clique** Given a graph and a bound  $B \in \mathbb{N}$ , decide if the graph has a  $B$ -clique, a set of  $B$ -many vertices where any two are connected.

**Hamiltonian Circuit** Given a graph, decide if it contains a Hamiltonian circuit, a cyclic path that includes each vertex.

**Partition** Given a finite multiset  $S$  of natural numbers, decide if there is a division of the set into the two parts  $\hat{S}$  and  $S - \hat{S}$  so the total of their elements is the same,  $\sum_{s \in \hat{S}} s = \sum_{s \notin \hat{S}} s$ .

- 7.6 **EXAMPLE** We will show that the **Traveling Salesman** problem is **NP** complete. Recall that we have recast it as the decision problem for the language of pairs  $\langle \mathcal{G}, B \rangle$ , where  $B$  is a parameter bound, and that this problem is a member of **NP**. We will show that it is **NP** hard by proving that the **Hamiltonian Circuit** problem reduces to it,  $\text{Hamiltonian Circuit} \leq_p \text{Traveling Salesman}$ .

We need a reduction function  $f$ . It must input an instance of **Hamiltonian Circuit**, a graph  $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$  whose edges are unweighted. Define  $f$  to return the instance of **Traveling Salesman** that uses  $\mathcal{N}$  as cities, that takes the distances between cities to be  $d(v_i, v_j) = 1$  if  $v_i v_j \in \mathcal{E}$  and  $d(v_i, v_j) = 2$  if  $v_i v_j \notin \mathcal{E}$ , and such that the bound is the number of vertices,  $B = |\mathcal{N}|$ .

This bound means that there will be a **Traveling Salesman** solution if and only if there is a **Hamiltonian Circuit** solution; namely, the salesman uses the edges that appear in the Hamiltonian circuit. All that remains is to argue that the reduction function runs in polytime. The number of edges in a graph is no more than twice the number of vertices so polytime in the input graph size is the same as polytime in the number of vertices. The reduction function's algorithm examines all pairs of vertices, which takes time that is quadratic in the number of vertices.

A common way to show that a given problem  $\mathcal{L}$  is **NP** hard is to show that a special case of  $\mathcal{L}$  is **NP** hard.

- 7.7 **EXAMPLE** The **Knapsack** problem starts with a multiset of objects  $S = \{s_0, \dots, s_{k-1}\}$ , each with a weight  $w(s_i) \in \mathbb{N}^+$  and a value  $v(s_i) \in \mathbb{N}^+$ , along with a weight bound  $B \in \mathbb{N}^+$  and value target  $T \in \mathbb{N}^+$ . We then look for a knapsack  $C \subseteq S$  whose elements have total weight less than or equal to the bound and total value greater than or equal to the target.

First, observe that this problem is in **NP**. As the witness we can use the  $k$ -bit string  $\omega$  such that  $\omega[i] = 1$  if  $s_i$  is in the knapsack  $C$ , 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  $C$ .

To finish we must show that Knapsack is NP hard. It is sufficient to show that a special case is NP hard. Consider the Knapsack instance where  $w(s_i) = v(s_i)$  for all  $s_i \in S$ , and where the two criteria each equal half of the weight total,  $B = T = 0.5 \cdot \sum_{0 \leq i < k} w(i)$ . This shows that any instance of the Partition problem, which is in the above basic list, can be expressed as a Knapsack instance, so the latter is NP hard.

Another common strategy in complexity proofs is to build the reduction function so that the behavior of the input instance's fundamental units is simulated by the subparts of the output instance. Such a construct is a **gadget**. The next example illustrate; it shows that  $3\text{-SAT} \leq_p 3\text{-Coloring}$  by having the clauses in the input boolean expression reflected by subgraphs in the output.

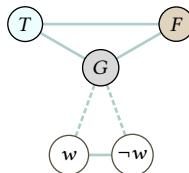
- 7.8 EXAMPLE Recall that a graph can be three-colored if we can partition its vertices into three categories, the colors, in such a way that no two same-colored vertices are connected by an edge. We will show that the **3-Coloring** problem,  $\mathcal{L} = \{\mathcal{G} \mid \text{the graph } \mathcal{G} \text{ has a three-coloring}\}$ , is NP complete.

The easy half is  $\mathcal{L} \in \text{NP}$ . As a witness we use a 3-coloring,  $\omega = \{C_0, C_1, C_2\}$ , where each  $C_i$  is a subset of  $\mathcal{G}$ 's vertices. Clearly we can produce a verifier that inputs the graph  $\sigma = \mathcal{G}$  along with  $\omega$  and certifies in polytime that  $\omega$  partitions the vertices, and also that never do two same-color vertices lie on the same edge.

The other half is to show that  $\mathcal{L}$  is NP hard. We will sketch the proof that  $3\text{-SAT}$  reduces to it, that  $3\text{-SAT} \leq_p 3\text{-Coloring}$ . The reduction function inputs a propositional logic expression, which it will expect to be in Conjunctive Normal Form. Appendix C contains the definitions but as a review, an example is  $(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee z)$ . Because clauses are joined by  $\wedge$ 's, the expression as a whole is  $T$  if and only if each clause is  $T$ . Inside each clause, because the literals such as  $x$  or  $\neg y$  are joined by  $\vee$ 's, the clause is  $T$  if and only if any of its literals is  $T$ . (We are working with  $3\text{-SAT}$ , so each clause contains three or fewer literals. Since we can change a clause with fewer literals such as  $x \vee y$ , into a clause with three, such as  $x \vee x \vee y$ , we can assume that clauses have exactly three literals.)

The reduction function must output instances of  $\mathcal{L}$ , that is, graphs. The output graph must be 3-colorable if and only if the input expression is satisfiable.

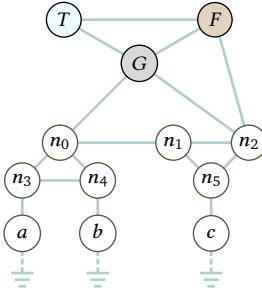
The graph below illustrates how we get some of what we want. It is built around a triangle of nodes labeled  $T$ ,  $F$ , and  $G$ . As a triangle in a 3-colored graph, no two nodes can have the same color.



The nodes labeled with the literals  $w$  and  $\neg w$  are connected to  $G$  and so each must have either the color of  $T$  or the color of  $F$ . They are also connected to each other

so one of them is the color of  $T$  while the other is the color of  $F$ . (The connections with  $G$  are just regular edges. We've drawn them with dotted lines to fit with the two graphs later in this example.)

Here is the gadget.



The nodes at the bottom will be labeled with literals. For instance, in the diagram (\*) below, the gadget on the right has  $a = \neg x$ ,  $b = \neg y$ , and  $c = z$ . In the gadget above those three nodes are connected to  $G$  but we've used a graphical shorthand, the symbol with three lines (the electrical ground symbol), because otherwise in diagram (\*) there would be too many edges to see clearly. We've drawn those edges as dashed to mark that they force  $T$ -ness or  $F$ -ness on the nodes, as described above. Finally, note also that the connections on the top right between  $n_2$  and  $F$  forces  $n_2$  to match  $T$ 's color.

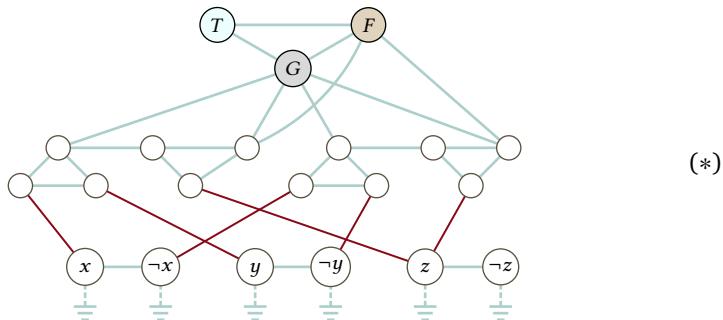
We will verify that this gadget is 3-colorable if and only if nodes  $a$ ,  $b$ , and  $c$  are not all the color of  $F$ . For “only if,” assume that  $a$ ,  $b$ , and  $c$  are the color of  $F$ . Then one of  $n_3$  and  $n_4$  is the color of  $T$  and the other is the color of  $G$ , and hence  $n_0$  is the color of  $F$ . As  $n_2$  is the color of  $T$  this implies that  $n_1$  is the color of  $G$  and this in turn gives that  $n_5$  is the color of  $F$ . That's impossible because  $c$  is the color of  $F$ .

For “if,” we need only exhibit that a 3-coloring exists for each remaining case.

$a$	$b$	$c$	$n_0$	$n_1$	$n_2$	$n_3$	$n_4$	$n_5$
$F$	$F$	$T$	$F$	$G$	$T$	$T$	$G$	$F$
$F$	$T$	$F$	$T$	$F$	$T$	$G$	$F$	$G$
$F$	$T$	$T$	$F$	$G$	$T$	$T$	$G$	$F$
$T$	$F$	$F$	$T$	$F$	$T$	$F$	$G$	$G$
$T$	$F$	$T$	$F$	$G$	$T$	$G$	$T$	$F$
$T$	$T$	$F$	$T$	$F$	$T$	$F$	$G$	$G$
$T$	$T$	$T$	$T$	$F$	$T$	$F$	$G$	$G$

Here is how to combine multiple gadgets for multiple clauses. This is the graph that the reduction function outputs for the input expression  $(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee z)$ . On the left is the gadget for the first clause and on the right is the gadget for the

second.



The highlighted edges match the two clauses.

One of Karp's points was the practical importance of NP completeness. Many problems from applications fall into this class. The next example illustrates.

- 7.9 EXAMPLE Scheduling is a rich source of difficult combinatorial problems. One is the problem of scheduling college classes into time slots such as Tuesday and Thursday from 8 until 9:20. Usually, colleges start the process by assigning classes to slots. Then students try to find classes that they need and that are offered in different time slots. 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.

Specifically, consider a college with  $t = 12$  available time slots. It must work  $n = 420$  classes into  $r = 60$  classrooms. (We will ignore many issues, such as that science lab classes must be in laboratory rooms.) Since  $12 \cdot 60 = 720$  and we need only accommodate 420 classes, this might seem easy. But when the 1724 students submit course requests, they create limits: on each student's list, each pair of classes puts the restriction on the college-wide schedule that those two classes cannot meet at the same time. Can the college find a schedule despite all these constraints?

The **Class Scheduling** problem inputs the number of time slots and rooms, along with the class requests  $s_i = \{c_{i_0}, c_{i_1}, \dots, c_{i_n}\}$  from each student, and decides if there is a way to allocate classes so that there is no conflict.

$$\mathcal{L} = \{\langle t, r, \{s_0, s_1, \dots\} \rangle \mid \text{a nonconflicting schedule exists}\}$$

We will show that this problem is NP-complete.

It is a member of NP because we can take as the witness  $\omega$  a schedule, a nonconflicting assignment of classes to rooms and times. Writing a verifier that inputs  $\sigma = \langle t, r, \{s_0, s_1, \dots\} \rangle$  and  $\omega$  and checks in polytime that  $\omega$  meets the restrictions, such as that each student's list has no conflict, is straightforward.

What remains is to show that this problem is NP hard. This problem is a good fit with the **Graph Colorability** problem: the colors are the  $t$ -many time slots, the nodes of the graph are the  $n$ -many classes, and two nodes are connected when

some student has requested them both. (There is a further restriction about the number of available rooms that we will handle along the way.) The prior example proves that **Graph Colorability** is NP hard for  $k = 3$  colors and an extension of that argument proves the same for any larger  $k$ . So we will show that **Class Scheduling** is NP hard by showing that **Graph Colorability**  $\leq_p$  **Class Scheduling**.

Assume that we are given an instance of **Graph Colorability**, a graph  $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$  and a natural number  $k$ . Here is the associated instance of the **Class Scheduling** problem: the number of classes is  $n = |\mathcal{V}|$ , as is the number of rooms, and the number of time slots is  $k$ . Each student takes two classes and an edge is in  $\mathcal{E}$  if and only if it connects some student's two. Clearly from the input instance we can in polytime produce the output instance. Also clearly, the **Graph Colorability** instance has a solution if and only if the associated **Class Scheduling** instance has a nonconflicting schedule.

Before we leave this discussion, we address a natural question: lots of problems are NP-complete but what problems are not?

Trivially, the definition gives that the empty language and the language of all strings are not complete. Also trivially, if a problem is not in NP then it is not NP-complete. For instance, as with the problem of finding a chess strategy, a problem can be so hard that we cannot even check its solution in polytime.

The more significant part of the answer to which problems are not NP complete is tied to whether or not P is unequal to NP. This is as-yet unknown, is the subject of a great deal of interest and research activity, and we will address in the next subsection. But, so that we have not just brushed past the issue, assume that  $P \neq NP$  (if  $P = NP$  then any nontrivial problem is NP hard). With that, most familiar problems are not NP complete—any text on algorithms is full of descriptions of problems that we can solve in polytime and  $P \neq NP$  implies that these problems are not NP-complete.

Still assuming that  $P \neq NP$ , finally consider  $NP - P$ , the problems in NP that cannot be solved with a polytime algorithm. Imagine that we are studying a problem that is in NP and no polytime algorithm occurs to us, even after considerable effort, and in addition we are unable to show that it is NP complete. It is worth noting that proving that such a problem is not NP complete is astonishingly difficult—at this moment the field does not know how to do it. Here are a few examples, problems that many experts believe are in  $NP - P$  but not NP complete, although no one can currently prove that: (i) the **Prime Factorization** problem is to produce the prime factorization of a given natural number,<sup>†</sup> (ii) the **Graph Isomorphism** problem is to decide if two graphs are isomorphic, and (iii) the **Vertex-to-Vertex Path** problem is to find if there is a path between given vertices. As always though, the standard caution applies that without proof these judgements could be mistaken.

---

<sup>†</sup> 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 the engineering that we need to build quantum computers. However, on classical computers most experts believe that this problem is in  $NP - P$  but not complete.

**P = NP?** Every deterministic Turing machine is trivially a nondeterministic machine and so  $P \subseteq NP$ . Thus every polytime problem is in  $NP$ . But what about the other direction—could it be that  $P = NP$  and every  $NP$  problem is, in this sense, easy?

We have seen that one way to think of nondeterministic machines is that they are unboundedly parallel. So the  $P$  versus  $NP$  question asks: does adding parallelism add speed?

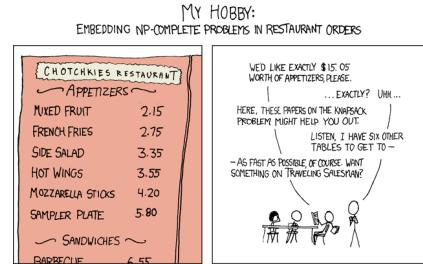
The short answer is that no one knows. There are a number of ways to settle the question. For example, by Lemma 7.4 if there is even one  $NP$  complete problem that we can prove is a member of  $P$ , then  $P = NP$ . Conversely, if someone shows that there is an  $NP$  problem that is not a member of  $P$  then  $P \neq NP$ . However, despite nearly a half century of effort by many brilliant people, no one has accomplished either one.

To explain the interest in the question, we first argue for its importance. As formulated in Karp's original paper, the question of whether  $P$  equals  $NP$  might seem of only technical interest.

A large class of computational problems involve the determination of properties of graphs, digraphs, integers, arrays of integers, finite families of finite sets, boolean formulas and elements of other countable domains. Through simple encodings ... these problems can be converted into language recognition problems, and we can inquire into their computational complexity. It is reasonable to consider such a problem satisfactorily solved when an algorithm for its solution is found which terminates within a number of steps bounded by a polynomial in the length of the input. We show that a large number of classic unsolved problems of covering, matching, packing, routing, assignment and sequencing are equivalent, in the sense that either each of them possesses a polynomial-bounded algorithm or none of them does.

But Karp showed that many problems that people had been struggling with in practice fall into this category. Researchers who had been looking for an efficient solution to *Vertex Cover* and those who have been working on *Clique* are working on the same problem, in that they are inter-translatable. By now the list of  $NP$  complete problems includes determining the best layout of transistors on a chip, developing accurate financial-forecasting models, analyzing protein-folding behavior in a cell, or finding the most energy-efficient airplane wing. So the question of whether  $P$  equals  $NP$  is extremely practical, and extremely important.<sup>†</sup>

Researchers often take proving that a problem is  $NP$  complete to be an ending

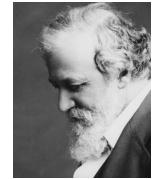


Courtesy xkcd.com

<sup>†</sup>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.”

point; they may feel that continuing to look for an algorithm is a waste since many of the world's best minds have failed to find one. They may turn to finding approximations (see Extra B), or to probabilistic methods.

We next argue that among many similar questions, each of which is important, P versus NP suggests itself as especially significant. First a philosophical take. At the start of this book we studied problems that are unsolvable. That is black and white — either a problem is mechanically solvable or it is not. In this chapter we find that many problems are solvable in principle but computing a solution seems to be infeasible. The poet R Browning wrote, “Ah, but a man's reach should exceed his grasp, Or what's a heaven for?” The set P consists of the problems that we can grasp, that we can feasibly solve. But if  $P \neq NP$  then the problems in  $NP - P$ , including the NP complete ones, are those that we can only reach. We can verify a correct answer but we cannot reliably find it. These problems seem to form a transition between the possible and the impossible.



Robert  
Browning,  
1812–1889

The sense that the P versus NP question fits into a larger intellectual program returns us to the book's opening. Recall the *Entscheidungsproblem* that was a motivation behind the definition of a Turing machine. It asks for an algorithm that inputs a mathematical statement and decides whether it is true. It is perhaps a caricature, but imagine that the job of mathematicians is to prove theorems. Then the *Entscheidungsproblem* asks if it is possible to replace mathematicians with mechanisms.

In the intervening century we have come to understand, through the work of Gödel and others, that there is a difference between a statement's being true and its being provable. Church and Turing expanded on this insight to show that the *Entscheidungsproblem* is unsolvable. Consequently, we change to asking for an algorithm that inputs statements and decides whether they are provable.

In principle there is a simple approach. A proof is a sequence of statements,  $\sigma_0, \sigma_1, \dots, \sigma_k$ , where the final statement is the conclusion and where each statement either is an axiom or else follows from the statements before it by an application of a rule of deduction (a typical rule allows the simultaneous replacement of all  $x$ 's with  $y + 4$ 's). A computer could brute-force the question of whether a given statement is provable by doing a dovetail, a breadth-first search of all derivations. If a proof exists then it will appear, eventually.<sup>†</sup>

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  $\sigma$  and bound, we ask if there is a sequence  $\omega$  of statements witnessing a proof that ends in  $\sigma$  and that is shorter than the bound. A computer can quickly check whether a given proof is valid — this problem is in NP. With the current status of the P versus NP problem, the answer to the question in the prior paragraph is that no one knows of a fast algorithm but no one

<sup>†</sup>That is, in a particular subject such as elementary number theory, the set of theorems is computably enumerable. (However, in number theory it is not computable.)

can show that there isn't one either.

As far back as 1956, Gödel raised these issues in a letter to von Neumann (this letter did not become public until years later).<sup>†</sup>

One can obviously easily construct a Turing machine, which for every formula  $F$  in first order predicate logic and every natural number  $n$ , allows one to decide if there is a proof of  $F$  of length  $n$  (length = number of symbols). Let  $\Psi(F, n)$  be the number of steps the machine requires for this and let  $\phi(n) = \max_F \Psi(F, n)$ . The question is how fast  $\phi(n)$  grows for an optimal machine. One can show that  $\phi(n) \geq k \cdot n$ . If there really were a machine with  $\phi(n) \sim k \cdot n$  (or even  $\sim k \cdot n^2$ ), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the *Entscheidungsproblem*, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number  $n$  so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that  $\phi(n)$  grows that 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.

In summary, we can compare P versus NP with the Halting problem. The Halting problem and related results tell us, in the light of Church's Thesis, what is knowable in principle. The P versus NP question, in contrast, speaks to what we can know in practice.

**Discussion** Certainly the P versus NP question is the sexiest one in the Theory of Computing today. It has attracted a great deal of gossip. In 2018, a poll of experts found that out of 152 respondents, 88% thought that  $P \neq NP$  while only 12% thought that  $P = NP$ . This subsection discusses some of the intuition involved in the question.

First, we address the intuition around the conjecture that  $P \neq NP$ . One way to think about the question is that a problem is in P if finding a solution is fast, while a problem is in NP if verifying the correctness of a given witness is fast. Then the claim that  $P \subseteq NP$  becomes the observation that if a problem is fast to solve then it must be fast to verify. But the other inclusion seems to most experts to be extremely unlikely. For example, speaking informally, S Aaronson has said, "I'd give it a 2 to 3 percent chance that P equals NP. Those are the betting odds that I'd take." Similarly, R Williams puts the chance that  $P \neq NP$  at 80%.

---

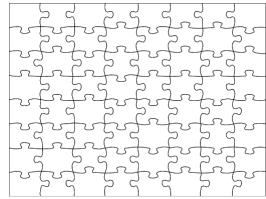
<sup>†</sup>At the meeting where Gödel, as an unknown fresh PhD, announced his Incompleteness Theorem, the only person who approached him with interest was von Neumann, who was already well established. Later, when Gödel was trying to escape the Nazis, von Neumann wrote to the director of the Institute for Advanced Study, "Gödel is absolutely irreplaceable. He is the only mathematician . . . about whom I would dare to make this statement." So they were professionally quite close. At the time of the letter, von Neumann had cancer, probably from his work on the Manhattan Project. Gödel was misinformed and wrote, "Since you now, as I hear, are feeling stronger, I would like to allow myself to write you about a mathematical problem, of which your opinion would very much interest me." Within a year von Neumann had died. We don't know if he replied or even read the letter.

As early as Karp's original paper there was a sense that  $P \neq NP$  was the natural guess. Here is the first paragraph of that paper.

All the general methods presently known for computing the chromatic number of a graph, deciding whether a graph has a Hamiltonian circuit, or solving a system of linear inequalities in which the variables are constrained to be 0 or 1, require a combinatorial search for which the worst case time requirement grows exponentially with the length of the input. In this paper we give theorems which strongly suggest ... that these problems, as well as many others, will remain intractable perpetually.

This intuition comes from a number of sources but an important one is the everyday experience that there is a genuine difference between the difficulty of finding a solution and that of verifying an existing solution is correct.

Imagine a jigsaw puzzle. We perceive that if a demon gave us an assembled puzzle  $\omega$ , then checking that it is correct is very much easier than it would have been to work out the solution from scratch. Checking for correctness is mechanical, tedious. But the finding of a solution, we perceive, is creative—we feel that solving a jigsaw puzzle by brute-force trying every possible piece against every other is too much computation to be practical.



Similarly, mathematicians find that verifying the correctness of a formally-described proof is routine. But finding that proof may be the work of a lifetime.



A Selman's plate,  
courtesy S Selman

There are schemes that leverage the separation that we believe exists between  $P$  and  $NP$ , between tractable and not, to engineering advantage. For example, secret codes are engineered so that, given an encrypted message, decrypting it with the key is fast and easy. But trying to decrypt it by wading through all possible keys is, we perceive, just not tractable.

Some commentators have extended this way of thinking beyond the narrow bounds of Theoretical Computer Science. Cook is one, “... Similar remarks apply to diverse creative human endeavors, such as designing airplane wings, creating physical theories, or even composing music. The question in each case is to what extent an efficient algorithm for recognizing a good result can be found.” Perhaps it is hyperbole to say that if  $P = NP$  then writing great symphonies would be a job for computers, a job for mechanisms, but it is correct to say that if  $P = NP$  and if we can write fast algorithms to recognize excellent music—and our everyday experience with Artificial Intelligence makes this seem more and more a possibility—then we could have fast mechanical writers of excellent music.

We finish with a taste of the intuition behind the contrarian view, the conjecture of some experts that  $P = NP$ .

Many observers have noted that there are cases where everyone “knew” that some algorithm was the fastest but in the end it proved not to be so. The section on Big- $O$  begins with one, the grade school algorithm for multiplication. Another is the problem of solving systems of linear equations. The Gauss's Method algorithm,

which runs in time  $\mathcal{O}(n^3)$ , is perfectly natural and had been known for centuries without anyone making improvements. However, while trying to prove that Gauss's Method is optimal, V Strassen found a  $\mathcal{O}(n^{\lg 7})$  method ( $\lg 7 \approx 2.81$ ).<sup>†</sup>

A more dramatic speedup happens with the Matching problem. It starts with a graph whose vertices represent people and such that pairs of vertices are connected if the people are compatible. We want a set of edges that is maximal, and such that no two edges share a vertex. The naive algorithm tries all possible match sets, which takes  $2^m$  checks where  $m$  is the number of edges. Even with only a hundred people there are more things to try than atoms in the universe. But since the 1960's we have an algorithm that runs in polytime.

Every day on the Theory of Computing blog feed there are examples of researchers producing algorithms faster than the ones previously known. A person can certainly have the sense that we are only just starting to explore what is possible with algorithms. R J Lipton captured this feeling.

Since we are constantly discovering new ways to program our “machines,” why not a discovery that shows how to factor? or how to solve *SAT*? Why are we all so sure that there are no great new programming methods still to be discovered? . . . I am puzzled that so many are convinced that these problems could not fall to new programming tricks, yet that is what is done each and every day in their own research.

Knuth has a related but somewhat different take.

Some of my reasoning is admittedly naive: It's hard to believe that  $P \neq NP$  and that so many brilliant people have failed to discover why. On the other hand if you imagine a number  $M$  that's finite but incredibly large . . . then there's a humongous number of possible algorithms that do  $n^M$  bitwise or addition or shift operations on  $n$  given bits, and it's really hard to believe that all of those algorithms fail.

My main point, however, is that I don't believe that the equality  $P = NP$  will turn out to be helpful even if it is proved, because such a proof will almost surely be nonconstructive. Although I think  $M$  probably exists, I also think human beings will never know such a value. I even suspect that nobody will even know an upper bound on  $M$ .

Mathematics is full of examples where something is proved to exist, yet the proof tells us nothing about how to find it. Knowledge of the mere existence of an algorithm is completely different from the knowledge of an actual algorithm.

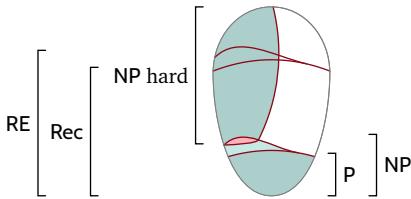
Of course, all this is speculation. Speculating is fun, and in order to make progress in their work, people must have some intuition. But in the end, we look to settle the question with proof.

## V.7 Exercises

7.10 This diagram is an extension of one we saw earlier. (It assumes that  $P \neq NP$ .)

---

<sup>†</sup>Here is an analogy: consider the problem of evaluating  $2p^3 + 3p^2 + 4p + 5$ . Someone might claim that writing it as  $2 \cdot p \cdot p \cdot p + 3 \cdot p \cdot p + 4 \cdot p + 5$  makes obvious that it requires six multiplications. But rewriting it as  $p \cdot (p \cdot (2 \cdot p + 3) + 4) + 5$  shows that it can be done with just three. That is, naturalness and obviousness do not guarantee that something is correct. Without a proof, we must worry that someone will produce a clever way to do the job with less.



On that, locate these languages.

- (A)  $K = \{\sigma \mid \sigma \text{ represents } x \in \mathbb{N} \text{ where } \phi_x(x) \downarrow\}$
- (B)  $\emptyset$
- (C)  $\mathcal{L}_B = \{\langle \mathcal{G}, v_0, v_1 \rangle \mid \text{there is a path from } v_0 \text{ to } v_1 \text{ of length at most } B\}$
- (D)  $SAT$
- ✓ 7.11 You hear someone say, “The **Satisfiability** problem is **NP** because it is not computable in polynomial time, so far as we know.” It’s a short sentence but find three mistakes.
- ✓ 7.12 Someone in your class says, “I will show that the **Hamiltonian Circuit** problem is not in **P**, which will demonstrate that **P**  $\neq$  **NP**. The algorithm to solve a given instance  $\mathcal{G}$  of the **Hamiltonian Circuit** problem is: generate all permutations of  $\mathcal{G}$ ’s vertices, test each to find if it is a circuit, and if any circuits appear then accept the input, else reject the input. For sure that algorithm is not polynomial, since the first step is exponential.” Where is their mistake?
- ✓ 7.13 Your friend says, “The problem of recognizing when one string is a substring of another has a polytime algorithm, so it is not in **NP**.” They have misspoken; help them out.
- 7.14 Someone in your study group wants to ask your professor, “Is the brute force algorithm for solving the **Satisfiability** problem **NP complete**?” Explain to them that it isn’t a sensible question, that they are making a type error.
- 7.15 True or false?
  - (A) The collection **NP** is a subset of the **NP complete** sets, which is a subset of **NP hard**.
  - (B) The collection **NP** is a specialization of **P** to nondeterministic machines, so it is a subset of **P**.
- ✓ 7.16 Assume that **P**  $\neq$  **NP**. Which of these statements can we infer from the fact that the **Prime Factorization** problem is in **NP**, but is not known to be **NP-complete**?
  - (A) There exists an algorithm for arbitrary instances of the **Prime Factorization** problem.
  - (B) There exists an algorithm that efficiently solves arbitrary instances of this problem.
  - (C) If we found an efficient algorithm for the **Prime Factorization** problem then we could immediately use it to solve **Traveling Salesman**.
- ✓ 7.17 Suppose that  $\mathcal{L}_1 \leq_p \mathcal{L}_0$ . For each, decide if you can conclude it. (A) If  $\mathcal{L}_0$  is **NP complete** then so is  $\mathcal{L}_1$ . (B) If  $\mathcal{L}_1$  is **NP complete** then so is  $\mathcal{L}_0$ .

- (c) If  $\mathcal{L}_0$  is NP complete and  $\mathcal{L}_1$  is in NP then  $\mathcal{L}_1$  is NP complete. (d) If  $\mathcal{L}_1$  is NP complete and  $\mathcal{L}_0$  is in NP then  $\mathcal{L}_0$  is NP complete. (e) It cannot be the case that both  $\mathcal{L}_0$  and  $\mathcal{L}_1$  are NP complete (f) If  $\mathcal{L}_1$  is in P then so is  $\mathcal{L}_0$ . (g) If  $\mathcal{L}_0$  is in P then so is  $\mathcal{L}_1$ .

7.18 Show that these are in NP but are not NP complete, assuming that  $P \neq NP$ .

- (A) The language of even numbers.
- (B) The language  $\{\mathcal{G} \mid \mathcal{G} \text{ has a vertex cover of size at most four}\}$ .

7.19 If  $P = NP$  then what happens with NP complete sets? Show that if  $P = NP$  then every nontrivial language in P is NP complete.

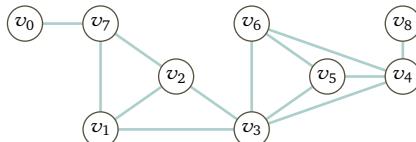
- ✓ 7.20 Traveling Salesman is NP complete. From  $P \neq NP$  which of the following statements could we infer?
  - (A) No algorithm solves all instances of Traveling Salesman.
  - (B) No algorithm quickly solves all instances of Traveling Salesman.
  - (C) Traveling Salesman is in P.
  - (D) All algorithms for Traveling Salesman run in polynomial time.

- ✓ 7.21 Prove that the **4-Satisfiability** problem is NP hard.

- ✓ 7.22 The Hamiltonian Path problem inputs a graph and decides if there are two vertices in that graph such that there is a path between those two that contains all the vertices.

(A) Show that Hamiltonian Path is in NP.

(B) This graph has a Hamiltonian path from  $v_0$  to  $v_8$ . Find it.

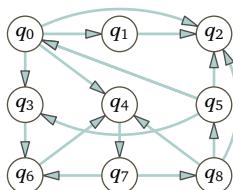


Why must those two be the endpoints?

- (c) Show that Hamiltonian Circuit  $\leq_p$  Hamiltonian Path.
- (d) Conclude that the Hamiltonian Path problem is NP complete.

- ✓ 7.23 The **Longest Path** problem is to input a graph and find the longest simple path in that graph.

(A) Find the longest path in this graph.



- (B) Remembering the technique for converting an optimization problem to a language decision problem by using bounds, state this as a language decision problem. Show that Longest Path  $\in NP$ .

- (c) Show that the **Hamiltonian Path** problem reduces to **Longest Path**. *Hint:* leverage the bound from the prior item.
- (d) Use the prior exercise to conclude that the **Longest Path** problem is NP complete.
- ✓ 7.24 The **Subset Sum** problem inputs a multiset  $T$  and a target  $B \in \mathbb{N}$ , and decides if there is a subset  $\hat{T} \subseteq T$  whose elements add to the target. The **Partition** problem inputs a multiset  $S$  and decides whether or not it has a subset  $\hat{S} \subset S$  so that the sum of elements of  $\hat{S}$  equals the sum of elements not in that subset.
- (A) Find a subset of  $T = \{3, 4, 6, 7, 12, 13, 19\}$  that adds to  $B = 30$ .
  - (B) Find a partition of  $S = \{3, 4, 6, 7, 12, 13, 19\}$ .
  - (C) Show that if the sum of the elements in a set is odd then the set has no partition.
  - (D) Express each problem as a language decision problem.
  - (E) Prove that **Partition**  $\leq_p$  **Subset Sum**. (*Hint:* handle separately the case where the sum of elements in  $S$  is odd.)
  - (F) Conclude that **Subset Sum** is NP complete.
- 7.25 The **3-Satisfiability** problem is to decide the satisfiability of propositional logic expression where every clause consists of three literals (consisting of different literals, the things between the  $\vee$ 's). The **Independent Set** problem inputs a graph and a bound, and decides if there is a set of vertices, of size at least equal to the bound, that are not connected to each other by an edge.
- (A) Find an independent set in this graph.
- 
- (B) State **Independent Set** as a language decision problem.
- (C) Decide if  $E = (P_0 \vee \neg P_1 \vee \neg P_2) \wedge (P_1 \vee P_2 \vee \neg P_3)$  is satisfiable.
- (D) State **3-Satisfiability** as a language decision problem.
- (E) With the expression  $E$ , make a triangle for each of the two clauses, where the vertices of the first are labeled  $v_0$ ,  $\bar{v}_1$ , and  $\bar{v}_2$ , while the vertices of the second are labeled  $w_1$ ,  $w_2$ , and  $\bar{w}_3$ . In addition to the edges forming the triangles, also put one connecting  $\bar{v}_1$  with  $w_1$ , and one connecting  $\bar{v}_2$  with  $w_2$ .
- (F) Sketch an argument that **3-Satisfiability**  $\leq_p$  **Independent Set**.
- ✓ 7.26 The difficulty in settling  $P = NP$  is to prove lower bounds. That is, the trouble lies in showing, for a given problem, that any algorithm at all must use such-and-such many steps. One common mistake is to reason that any algorithm for the problem must take at least as many steps as the length of the input, thinking that to compute the output the algorithm must at least read all of the input. We will exhibit a familiar problem for which this isn't true.

Consider the successor function. Show that it can be computed on a Turing machine without reading all of the input. More, show how to compute it in constant time, that it has an algorithm whose running time when the input is

large is the same as the running time when the input is small. Assume that the algorithm is given the input  $n$  in unary with the head under the leftmost 1, and that it ends with  $n + 1$ -many 1's and with the head under the leftmost 1.

7.27 Do we know of any problems in **NP** and not in **P**, and that are not **NP** complete?

7.28 Find three languages so that  $\mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$ , and  $\mathcal{L}_2, \mathcal{L}_0$  are **NP** complete, while  $\mathcal{L}_1 \in \mathbf{P}$ .

7.29 Prove Lemma 7.4.

7.30 The class **P** has some nice closure properties, and so does **NP**. (A) Prove that **NP** is closed under union, so that if  $\mathcal{L}, \hat{\mathcal{L}} \in \mathbf{NP}$  then  $\mathcal{L} \cup \hat{\mathcal{L}} \in \mathbf{NP}$ . (B) Prove that **NP** is closed under concatenation. (C) Argue that no one can prove that **NP** is not closed under set complement.

7.31 Is the set of **NP** complete sets countable or uncountable?

7.32 We will sketch a proof that the Halting problem is **NP** hard but not **NP**. Consider the language  $\mathcal{HP} = \{\langle \mathcal{P}_e, x \rangle \mid \phi_e(x) \downarrow\}$ . (A) Show that  $\mathcal{HP} \notin \mathbf{NP}$ . (B) Sketch an argument that for any problem  $\mathcal{L} \in \mathbf{NP}$ , there is a polynomial time computable verifier,  $f: \mathbb{B}^* \rightarrow \mathbb{B}^*$ , such that  $\sigma \in \mathcal{L}$  if and only if  $f(\sigma) \in \mathcal{HP}$ .

## SECTION

### V.8 Other classes

There are many other defined complexity classes. The next class is quite natural.

The Satisfiability problem is a touchstone result among problems in **NP**. We have discussed computing it using a nondeterministic Turing machine that is unboundedly parallel, or alternatively using a witness and verifier. But, naively, in the familiar computational setting of a deterministic machine, it appears that to solve it, we must go through the truth table line by line. That is, **SAT** appears to take exponential time.

**EXP** In this chapter's first section we included  $\mathcal{O}(2^n)$  and  $\mathcal{O}(3^n)$ , and by extension other exponentials, in the list of common orders of growth.

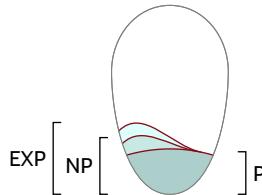
8.1 **DEFINITION** A language decision problem is an element of the complexity class **EXP** if there is an algorithm for solving it that runs in time  $\mathcal{O}(b^{p(n)})$  for some constant base  $b$  and polynomial  $p$ .

A first, informal, take is that **EXP** contains nearly every problem with which we concern ourselves in practice—it contains most problems that we seriously hope ever to attack. In contrast with polytime, where a rough summary is that its problems all have an algorithm that can conceivably be used, for the hardest problems in **EXP**, even the best algorithms are just too slow.

## 8.2 LEMMA $P \subseteq NP \subseteq EXP$

*Proof* Fix  $\mathcal{L} \in NP$ . We can verify  $\mathcal{L}$  on a deterministic Turing machine  $\mathcal{P}$  in polynomial time using a witness whose length is bounded by the same polynomial. Let this problem's bound be  $n^c$ .

We will decide  $\mathcal{L}$  in exponential time by brute-forcing it: we will use  $\mathcal{P}$  to run every possible verification. Trying any single witness requires polynomial time,  $n^c$ . Witnesses are in binary so for length  $\ell$  there are  $\sum_{0 \leq i \leq \ell} 2^i = 2^{\ell+1} - 1$  many possible ones; In total then, brute force requires  $\mathcal{O}(n^c 2^{n^c})$  operations. Finish by observing that  $n^c 2^{n^c}$  is in  $\mathcal{O}(2^{n^c})$ .  $\square$



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.

We know by a result called the Time Hierarchy Theorem that the three classes are not all equal. But, just as we don't today have a proof that  $P$  is a proper subset of  $NP$ , we also don't know whether or not there are  $NP$ -complete problems that absolutely require exponential time. The class  $NP$  could conceivably be contained in a smaller deterministic time complexity class—for instance, maybe **Satisfiability** can be solved in less than exponential time. But we just don't know.

**Time Complexity** Researchers have generalized to many more classes, trying to capture various aspects of computation. For instance, the impediment that a programmer runs across first is time.

8.4 **DEFINITION** Let  $f: \mathbb{N} \rightarrow \mathbb{N}$ . A decision problem for a language is an element of  $\text{DTIME}(f)$  if it is decided by a deterministic Turing machine that runs in time  $\mathcal{O}(f)$ . A problem is an element of  $\text{NTIME}(f)$  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  $\text{DTIME}(n^c)$  for some power  $c \in \mathbb{N}$ .

$$P = \bigcup_{c \in \mathbb{N}} \text{DTIME}(n^c) = \text{DTIME}(n) \cup \text{DTIME}(n^2) \cup \text{DTIME}(n^3) \cup \dots$$

The matching statements hold for **NP** and **EXP**.

$$\begin{aligned} \text{NP} &= \bigcup_{c \in \mathbb{N}} \text{NTIME}(n^c) = \text{NTIME}(n) \cup \text{NTIME}(n^2) \cup \text{NTIME}(n^3) \cup \dots \\ \text{EXP} &= \bigcup_{c \in \mathbb{N}} \text{DTIME}(2^{n^c}) = \text{DTIME}(2^n) \cup \text{DTIME}(2^{n^2}) \cup \text{DTIME}(2^{n^3}) \cup \dots \end{aligned}$$

*Proof* The only equality that is not immediate is the last one. Recall that a problem is in **EXP** if an algorithm for it that runs in time  $\mathcal{O}(b^{p(n)})$  for some constant base  $b$  and polynomial  $p$ . The equality above only uses the base 2. To cover the discrepancy, we will show that  $3^n \in \mathcal{O}(2^{(n^2)})$ . Consider  $\lim_{x \rightarrow \infty} 2^{(x^2)} / 3^x$ . Rewrite the fraction as  $(2^x/3)^x$ , which when  $x > 2$  is larger than  $(4/3)^x$ , which goes to infinity. This argument works for any base, not just  $b = 3$ .  $\square$

- 8.6 **REMARK** While the above description of **NP** reiterates its naturalness, as we saw earlier, the characterization that proves to be most useful in practice is that a problem  $\mathcal{L}$  is in **NP** if there is a deterministic Turing machine such that for each input  $\sigma$  there is a polynomial length witness  $\omega$  and the verification on that machine for  $\sigma$  using  $\omega$  takes polytime.

**Space Complexity** We can consider how much space is used in solving a problem.

- 8.7 **DEFINITION** A deterministic Turing machine **runs in space**  $s: \mathbb{B}^* \rightarrow \mathbb{R}^+$  if for all but finitely many inputs  $\sigma$ , the computation on that input uses less than or equal to  $s(|\sigma|)$ -many cells on the tape. A nondeterministic Turing machine **runs in space**  $s$  if for all but finitely many inputs  $\sigma$ , every computation path on that input takes less than or equal to  $t(|\sigma|)$ -many cells.

The machine must use less than or equal to  $s(|\sigma|)$ -many cells even on non-accepting computations.

- 8.8 **DEFINITION** Let  $s: \mathbb{N} \rightarrow \mathbb{N}$ . A language decision problem is an element of **DSPACE(s)**, or **SPACE(s)**, if that language is decided by a deterministic Turing machine that runs in space  $\mathcal{O}(s)$ . A problem is an element of **NSPACE(s)** if the language is decided by a nondeterministic Turing machine that runs in space  $\mathcal{O}(s)$ .

The definitions arise from a sense we have of a symmetry between time and space, that they are both examples of computational resources. (There are other resources; for instance we may want to minimize disk reading or writing, which may be quite different than space usage.) But space is not just like time. For one thing, while a program can take a long time but use only a little space, the opposite is not possible.

- 8.9 **LEMMA** Let  $f: \mathbb{N} \rightarrow \mathbb{N}$ . Then  $\text{DTIME}(f) \subseteq \text{DSPACE}(f)$ . As well, this holds for nondeterministic machines,  $\text{NTIME}(f) \subseteq \text{NSPACE}(f)$ .

*Proof* A machine can use at most one cell per step.  $\square$

## 8.10 DEFINITION

$$\text{PSPACE} = \bigcup_{c \in \mathbb{N}} \text{DSPACE}(n^c) = \text{DSPACE}(n) \cup \text{DSPACE}(n^2) \cup \text{DSPACE}(n^3) \cup \dots$$

$$\text{NPSPACE} = \bigcup_{c \in \mathbb{N}} \text{NSPACE}(n^c) = \text{NSPACE}(n) \cup \text{NSPACE}(n^2) \cup \text{NSPACE}(n^3) \cup \dots$$

So **PSPACE** is the class of problems that can be solved by a deterministic Turing machine using only a polynomially-bounded amount of space, regardless of how long the computation takes.

As even those preliminary results suggest, restricting by space instead of time allows for a lot more power.

8.11 LEMMA  $\text{P} \subseteq \text{NP} \subseteq \text{PSPACE}$ 

*Proof* For any problem in NP, check all possible witness strings  $\omega$ . These take at most polynomial space. If any proof string works then the answer to the problem is ‘yes’. Otherwise, the answer is ‘no’.  $\square$

Note that the method in the proof may take exponential time but it takes only polynomial space.

Here is a result whose proof is beyond our scope, but that serves as a caution that time and space are very different. We don’t know whether deterministic polynomial time equals nondeterministic polynomial time, but we do know the answer for space.

8.12 THEOREM (SAVITCH’S THEOREM)  $\text{PSPACE} = \text{NPSPACE}$ 

We finish with a list of the most natural complexity classes.

8.13 DEFINITION These are the **canonical complexity classes**

1.  $\text{L} = \text{DSPACE}(\lg n)$ , **deterministic log space** and  $\text{NL} = \text{NSPACE}(\lg n)$ , **nondeterministic log space**
2.  $\text{P}$ , **deterministic polynomial time** and  $\text{NP}$ , **nondeterministic polynomial time**
3.  $\text{E} = \bigcup_{k=1,2,\dots} \text{DTIME}(k^n)$  and  $\text{NE} = \bigcup_{k=1,2,\dots} \text{NTIME}(k^n)$
4.  $\text{EXP} = \bigcup_{k=1,2,\dots} \text{DTIME}(2^{n^k})$ , **deterministic exponential time** and  $\text{NEXP} = \bigcup_{k=1,2,\dots} \text{NTIME}(2^{n^k})$ , **nondeterministic exponential time**
5. **PSPACE**, **deterministic polynomial space**
6.  $\text{EXPSPACE} = \bigcup_{k=1,2,\dots} \text{DSPACE}(2^{n^k})$ , **deterministic exponential space**

**The Zoo** Researchers have studied a great many complexity classes. There are so many that they have been gathered into an online Complexity Zoo, at [complexityzoo.uwaterloo.ca/](http://complexityzoo.uwaterloo.ca/).

One way to understand these classes is that defining a class asks a type of Theory of Computing question. For instance, we have already seen that asking whether NP equals P is a way of asking whether unbounded parallelism makes any essential difference—can a problem change from intractable to tractable if we switch from a deterministic to a nondeterministic machine? Similarly, we know that  $P \subseteq PSPACE$ . In thinking about whether the two are equal, researchers are considering the space-time tradeoff: if you can solve a problem without much memory does that mean you can solve it without using much time?

Here is one extra class, to give some flavor of the possibilities. For more, see the Zoo.

The class **BPP**, **Bounded-Error Probabilistic Polynomial Time**, contains the problems solvable by an nondeterministic polytime machine such that if the answer is ‘yes’ then at least two-thirds of the computation paths accept and if the answer is ‘no’ then at most one-third of the computation paths accept. (Here all computation paths have the same length.) This is often identified as the class of feasible problems for a computer with access to a genuine random-number source. Investigating whether BPP equals P is asking whether every efficient randomized algorithm can be made deterministic: are there some problems for which there are fast randomized algorithms but no fast deterministic ones?

On reading in the Zoo, a person is struck by two things. There are many, many results listed—we know a lot. But there also are many questions to be answered—breakthroughs are there waiting for a discoverer.

## V.8 Exercises

- ✓ 8.14 Give a naive algorithm for each problem that is exponential. (A) **Subset Sum** problem (B)  **$k$  Coloring** problem
- 8.15 Show that  $n!$  is  $2^{\mathcal{O}(n^2)}$ . Show that **Traveling Salesman**  $\in$  EXP.
- ✓ 8.16 This illustrates how large a problem can be and still be in EXP. Consider a game that has two possible moves at each step. The game tree is binary.
  - (A) How many elementary particles are there in the universe?
  - (B) At what level of the game tree will there be more possible branches than there are elementary particles?
  - (C) Is that longer than a chess game can reasonably run?
- 8.17 We will show that a polynomial time algorithm that calls a polynomial time subroutine can run, altogether, in exponential time.
  - (A) Verify that the grade school algorithm for multiplication gives that squaring an  $n$ -bit integer takes time  $\mathcal{O}(n)$ .
  - (B) Verify that repeated squaring of an  $n$ -bit integer gives a result that has length  $2^i n$ , where  $i$  is the number of squarings.
  - (C) Verify that if your polynomial time algorithm calls a squaring subroutine  $n$  times then the complexity is  $\mathcal{O}(4^n n^2)$ , which is exponential.

## EXTRA

## V.A RSA Encryption

In this chapter we have built up the sense that there are functions that are intractable to compute. Here we see that we can leverage this to engineering advantage. For instance, schemes for holding elections are notoriously prone to manipulation and there are theorems saying that they must be. But we can hope to use system that, while manipulatable in principle, is constructed so that it is in practice infeasible to compute how to do the manipulation. This section describes another example, the celebrated RSA encryption system that is used to protect Internet commerce.

One of the great things about the interwebs, besides that you can get free *Theory of Computing* books, is that you can buy stuff. You send a credit card number and a couple of days later the stuff appears. For this to be practical, your credit card number must be kept secret. It must be encrypted.

When you visit a web site using a `https` address, that site sends you information, called a key, that your browser uses to encrypt your card number. The web site then uses a different key to decrypt. This is an important point: the decrypter must differ from than the encrypter since anyone on the net can see the encrypter information that the site sent you. But the site keeps the decrypter information private. These two, encrypter and decrypter, form a matched pair. We will describe the mathematical technologies that make this work.

**The arithmetic** We can view that everything on a modern computer is numbers. Consider the message ‘send money’. Its ASCII encoding is 115 101 110 100 32 109 111 110 101 121. Converting to a bitstring gives 01110011 01100101 01101110 01100100 00100000 01101101 01101111 01101110 01100101 01111001. In decimal that’s 544 943 221 199 950 100 456 825. So there is no loss in generality in viewing everything we do, including encryption systems, as numerical operations.

To make such systems, mathematicians and computer scientists have leveraged that there are things we can do easily, but that we do not know how to easily undo — that are numerical operations we can use for encryption that are fast, but such that the operations needed to decrypt (without the decrypter) are believed to be so slow that they are completely impractical. So this is the engineering of Big- $\mathcal{O}$ .

We will describe an algorithm based on the Factoring problem. We have algorithms for multiplying numbers that are fast. The algorithms that we have for starting with a number and decomposing it into factors are, by comparison, quite slow. To illustrate this, you might contrast the time it takes you to multiply two four-digit numbers by hand with the time it takes you to factor an eight-digit number chosen at random. Set aside an afternoon for that second job, it’ll take a while.

The algorithm that we shall describe exploits the difference. It was invented in 1976 by three graduate students, R Rivest, A Shamir, and L Adleman. Rivest read a paper proposing the idea of key pairs and decided to develop an implementation.

Over the course of a year, he and Shamir came up with a number of ideas and for each Adleman would then produce a way to break it. Finally they thought to use Fermat's Little Theorem. Adleman was unable to break it since, he said, it seemed that only solving Factoring would break it and no one knew how to do that. Their algorithm, called RSA, was first announced in Martin Gardner's *Mathematical Games* column in the August 1977 issue of *Scientific American*. It generated a tremendous amount of interest and excitement.

The basis of RSA is to find three numbers, a **modulus**  $n$ , an **encrypter**  $e$ , and a **decrypter**  $d$ , related by this equation (here  $m$  is the message, as a number).

$$(m^e)^d \equiv m \pmod{n}$$

The encrypted message is  $m^e \pmod{n}$ . To decrypt it, to recover  $m$ , calculate  $(m^e)^d \pmod{n}$ . These three are chosen so that knowing  $e$  and  $n$ , or even  $m$ , still leaves a potential secret-cracker who is looking for  $d$  with an extremely difficult job.

To choose them, first choose distinct prime numbers  $p$  and  $q$ . Pick these at random so they are of about equal bit-lengths. Compute  $n = pq$  and  $\varphi(n) = (p - 1) \cdot (q - 1)$ . Next, choose  $e$  with  $1 < e < \varphi(n)$  that is relatively prime to  $n$ . 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.)

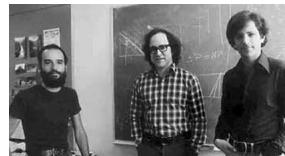
The pair  $\langle n, e \rangle$  is the **public key** and the pair  $\langle n, d \rangle$  is the **private key**. The length of  $d$  in bits is the **key length**. Most experts consider a key length of 2 048 bits to be secure for the mid-term future, until 2030 or so, when computers will have grown in power enough that they may be able to use an exhaustive brute-force search to find  $d$ .

- A.1 EXAMPLE Alice chooses the primes  $p = 101$  and  $q = 113$  (these are too small to use in practice but are good for an illustration) and then calculates  $n = pq = 11\,413$  and  $\varphi(n) = (p - 1)(q - 1) = 11\,200$ . To get the encrypter she randomly picks numbers  $1 < e < 11\,200$  until she gets one that is relatively prime to 11 200, choosing  $e = 3533$ . She publishes her public key  $\langle n, e \rangle = \langle 11\,413, 3\,533 \rangle$  on her home page. She computes the decrypter  $d = e^{-1} \pmod{11\,200} = 6\,597$ , and finds a safe place to store her private key  $\langle n, d \rangle = \langle 11\,413, 6\,597 \rangle$ .

Bob wants to say 'Hi'. In ASCII that's 01001000 01101001. If he converted that string into a single decimal number it would be bigger than  $n$  so he breaks it into two substrings, getting the decimals 72 and 105. Using her public key he computes

$$72^{3533} \pmod{11\,413} = 10496 \quad 105^{3533} \pmod{11\,413} = 4861$$

and sends Alice the sequence  $\langle 10496, 4861 \rangle$ . Alice recovers his message by using



Adi Shamir (b 1952), Ron Rivest (b 1947), Leonard Adleman (b 1945)

her private key.

$$10496^{6597} \bmod 11413 = 72 \quad 4861^{6597} \bmod 11413 = 105$$

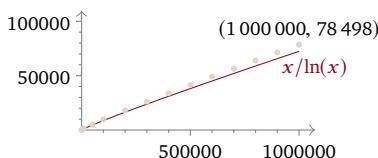
**The arithmetic, fast** We've just illustrated that RSA uses invertible operations. There are lots of ways to get invertible operations so our understanding of RSA is incomplete unless we know why it uses these particular operations. As discussed above, the important point is that they can be done quickly, but undoing them, finding the decrypter, is believed to take a very long time.

We start with a classic, beautiful, result from Number Theory.

- A.2 **THEOREM (PRIME NUMBER THEOREM)** The number of primes less than  $n \in \mathbb{N}$  is approximately  $n/\ln(n)$ ; that is, this limit is 1.

$$\lim_{x \rightarrow \infty} \frac{\text{number of primes less than } x}{(x/\ln x)}$$

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. (We will give examples only, rather than the full analysis of the operations.)

- A.3 **EXAMPLE** Multiplying 3 915 421 by 52 567 004 modulo 3 looks hard. The naive approach is to first take their product and then divide by 3 to find the remainder. But there is a more efficient way. Rather than multiply first and then reduce modulo  $m$ , reduce first and then multiply. That is, we know that if  $a \equiv b \pmod{m}$  and  $c \equiv d \pmod{m}$  then  $ac \equiv bd \pmod{m}$  and so since  $3 915 421 \equiv 1 \pmod{3}$  and  $52 567 004 \equiv 2 \pmod{3}$  we have this.

$$3 915 421 \cdot 52 567 004 \equiv 1 \cdot 2 \pmod{3}$$

Similarly, exponentiation modulo  $m$  is also efficient, both in time and in space.

- A.4 EXAMPLE Consider raising 4 to the 13-th power, modulo  $m = 497$ . The naive approach would be to raise 4 to the 13-th power, which is a very large number, and reduce modulo 497. But there is a better way.

Start by expressing the power 13 in base 2 as  $13 = 8 + 4 + 1 = 1101_2$ . So,  $4^{13} = 4^8 \cdot 4^4 \cdot 4^1$  and we need the 8-th power, the 4-th power, and the first power of 4. If we can efficiently get those powers then we can multiply them modulo  $m$  efficiently, so we will be set.

Get the powers by repeated squaring (modulo  $m$ ). Start with  $p = 1$ . Squaring gives  $4^2$ , then squaring again gives  $4^4$ , and squaring again gives  $4^8$ . Getting these powers (modulo  $m$ ) just requires a multiplication, which we can do efficiently.

The last thing we need for efficiently testing primality is to efficiently find the multiplicative inverse modulo  $m$ . Recall that two numbers are **relatively prime** or **coprime** if their greatest common divisor is 1. For example,  $15 = 3 \cdot 5$  and  $22 = 2 \cdot 11$  are relatively prime.

- A.5 LEMMA If  $a$  and  $m$  are relatively prime then there is an inverse for  $a$  modulo  $m$ , a number  $k$  such that  $a \cdot k \equiv 1 \pmod{m}$

*Proof* Because the greatest common divisor of  $a$  and  $m$  is 1, Euclid's algorithm gives a linear combination of the two, a  $sa + tm$  for some  $s, t \in \mathbb{Z}$ , that adds to 1. Doing the operations modulo  $m$  gives  $sa + tm \equiv 1 \pmod{m}$ . Since  $tm$  is a multiple of  $m$ , we have  $tm \equiv 0 \pmod{m}$ , leaving  $sa \equiv 1 \pmod{m}$ , and  $s$  is the inverse of  $a$  modulo  $m$ .  $\square$

Euclid's algorithm is efficient, both in time and space, so finding an inverse modulo  $m$  is efficient.

Now we can test for primes. The simplest way to test whether a number  $n$  is prime is to try dividing  $n$  by all possible factors. But that is very slow. There is a faster way, based on the next result.

- A.6 THEOREM (FERMAT) For a prime  $p$ , if  $a \in \mathbb{Z}$  is not divisible by  $p$  then  $a^{p-1} \equiv 1 \pmod{p}$ .

*Proof* Let  $a$  be an integer not divisible by the prime  $p$ . Multiply  $a$  by each number  $i \in \{1, \dots, p-1\}$  and reduce modulo  $p$  to get the numbers  $r_i = ia \pmod{p}$ .

We will show that the set  $R = \{r_1, \dots, r_{p-1}\}$  equals the set  $P = \{1, \dots, p-1\}$ . First,  $R \subseteq P$ . Because  $p$  is prime and does not divide  $i$  or  $a$ , it does not divide their product  $ia$ . Thus  $r_i = ia \not\equiv 0 \pmod{p}$  and so all the  $r_i$  are members of the set  $\{1, \dots, p-1\}$ .

To get inclusion the other way,  $P \subseteq R$ , note that if  $i_0 \neq i_1$  then  $r_{i_0} \neq r_{i_1}$ . For, with  $r_0 - r_1 = i_0a - i_1a = (i_0 - i_1)a$ , because  $p$  is prime and does not divide  $i_0 - i_1$  or  $a$  (as each is smaller in absolute value than  $p$ ), it does not divide their product. That means that the two sets have the same number of elements, so  $P \subseteq R$ .

Now multiply together all of the elements of that set.

$$\begin{aligned} a \cdot 2a \cdots (p-1)a &\equiv 1 \cdot 2 \cdots (p-1) \pmod{p} \\ (p-1)! \cdot a^{p-1} &\equiv (p-1)! \pmod{p} \end{aligned}$$

Cancelling the  $(p-1)!$ 's gives the result.  $\square$

- A.7 EXAMPLE Let the prime be  $p = 7$ . Any number  $a$  with  $0 < a < p$  is not divisible by  $p$ . Here is the list.

$a$	1	2	3	4	5	6
$a^{7-1}$	1	64	729	4096	15625	46656
$(a^6 - 1)/7$	0	9	104	585	2232	6665

For instance,  $15625 = 7 \cdot 2232 + 1$ .

Given  $n$ , if we find a base  $a$  with  $0 < a < n$  so that  $a^{n-1} \pmod{n}$  is not 1 then  $n$  is not prime.

- A.8 EXAMPLE Consider  $n = 415692$ . If  $a = 2$  then  $2^{415692} \equiv 58346 \pmod{415693}$  so  $n$  is not prime.

There are  $n$ 's 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 for base  $a = 2$  is  $n = 341 = 11 \cdot 31$ . However, computer searches suggest that these are very rare.

The rarity of exceptions suggests that we use a **probabilistic primality test**: given  $n \in \mathbb{N}$  to test for primality, pick at random a base  $a$  with  $0 < a < n$  and calculate whether  $a^n - 1 \equiv 1 \pmod{n}$ . If that is not true then  $n$  is not prime.<sup>†</sup> If it is true then we have evidence that  $n$  is prime.

Researchers have shown that if  $n$  is not prime then each choice of base  $a$  has a greater 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  $a_1^n - 1 \equiv 1 \pmod{n}$ . So we figure that there is at least a  $1 - (1/2)^2$  chance that  $n$  is prime. After  $k$ -many iterations of choosing a base, doing the calculation, and never finding that that  $n$  is not prime, then we have a greater than  $1 - (1/2)^k$  chance that  $n$  is prime.

In summary, if  $n$  passes  $k$ -many tests for any reasonable-sized  $k$  then we are quite confident that it is prime. Our interest in this test is that it is extremely fast; it runs in time  $\mathcal{O}(k \cdot (\log n)^2 \cdot \log \log n \cdot \log \log \log n)$ . So we can run it lots of times, becoming very confident, in not very much time.

- A.9 EXAMPLE We could test whether  $n = 7$  is prime by computing, say, that  $3^6 \equiv 1 \pmod{7}$ , and  $5^6 \equiv 1 \pmod{7}$ , and  $6^6 \equiv 1 \pmod{7}$ . The fact that  $n = 7$  does not fail makes us confident it is prime.

The RSA algorithm also uses this offshoot of Fermat's Little Theorem.

---

<sup>†</sup>In this case  $a$  is a witness to the fact that  $n$  is not prime.

**A.10 COROLLARY** Let  $p$  and  $q$  be unequal primes and suppose that  $a$  is not divisible by either one. 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.

$$a^{(p-1)(q-1)} \equiv 1 \pmod{p} \quad a^{(p-1)(q-1)} \equiv 1 \pmod{q}$$

Since  $a^{(p-1)(q-1)} - 1$  is divisible by both  $p$  and  $q$ , it is divisible by their product  $pq = n$ .  $\square$

Experts think that the most likely attack on RSA encryption is by factoring the modulus  $n$ . Anyone who factors  $n$  can use the same method as the RSA key setup process to turn the encrypter  $e$  into the decrypter  $d$ . That's why  $n$  is taken to be the product of two large primes; it makes factoring as hard as possible.

There is a factoring algorithm that takes only  $\mathcal{O}(b^3)$  time (and  $\mathcal{O}(b)$  space), called Shor's algorithm. But it runs only on quantum computers. At this moment there are no such computers built, although there has been progress on that. For the moment, RSA seems safe. (There are schemes that could replace it, if needed.)

### V.A Exercises

- ✓ A.11 There are twenty five primes less than or equal to 100. Find them.
- ✓ A.12 We can walk through an RSA calculation.
  - (A) For the primes, take  $p = 11$ ,  $q = 13$ . Find  $n = pq$  and  $\varphi(n) = (p-1) \cdot (q-1)$ .
  - (B) For the encoder  $e$  use the smallest prime  $1 < e < \varphi(n)$  that is relatively prime with  $\varphi(n)$ .
  - (C) Find the decoder  $d$ , the multiplicative inverse of  $e$  modulo  $n$ . (You can use Euclid's algorithm, or just test the candidates.)
  - (D) Take the message to be represented as the number  $m = 9$ . Encrypt it and decrypt it.
- A.13 To test whether a number  $n$  is prime, we could just try dividing it by all numbers less than it.
  - (A) Show that we needn't try all numbers less than  $n$ , instead we can just try all  $k$  with  $2 \leq k \leq \sqrt{n}$ .
  - (B) Show that we cannot lower than any further than  $\sqrt{n}$ .
  - (C) For input  $n = 10^{12}$  how many numbers would you need to test?
  - (D) Show that this is a terrible algorithm since it is exponential in the size of the input.
- A.14 Show that the probability that a random  $b$ -bit number is prime is about  $1/b$ .

## EXTRA

## V.B Good-enoughness

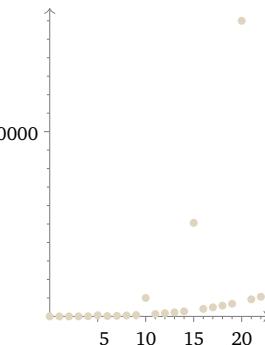
A theory shapes the way that you look at the world, at how you see and address what comes before you in practice. For instance, Newton's  $F = ma$  points to an approach to analyzing physical situations: if you see an acceleration then look around for a force. That approach has been fantastically successful, enabling us to build bridges, send people to the moon, etc. Likewise, Darwin's theory tells us that if there is a change in a species then look for a reproductive advantage.

Here we will point out a way in which a naive understanding of our theory can lead to a misunderstanding of what can be done in practice. Of course, the theorems are right — the proofs check out, the results stand up to formalization, etc. But in learning we build mental models of what those formal statements mean and there is a common misperception about solving problems that our theory labels “hard.”

Our theory grades problems by their best algorithms. In particular, Cobham's Thesis notes the sharp divide between  $\text{P}$  and super- $\text{P}$  and identifies  $\text{P}$  with the problems having tractable algorithms. We have already noted that just because a problem is in  $\text{P}$  does not mean that it has a practical algorithm; an example is one whose fastest algorithm is  $\mathcal{O}(n^{1000})$ . Another issue is when an algorithm is polytime but it has a huge coefficient, such as  $2^{1000} \cdot n^2$ .

The flip side of this is that just because a problem is  $\text{NP}$  hard does not mean that it is hard to solve in practice. For a first take on how this can be, consider this function.

$$f(n) = \begin{cases} n^2 & - \text{if } n \text{ is not a multiple of five} \\ n^{\lg n} & - \text{otherwise} \end{cases}$$



Most of the time it is quadratic, but occasionally it is super-polynomial. The exceptions could be rarer than shown, maybe every 10, or every  $10^{10}$ , or  $10^x$ . The definition of Big  $\mathcal{O}$  is such that as long as there are infinitely many superpoly exceptions then the function's growth is not poly.

Suppose that a problem's best algorithm is  $\mathcal{O}(f)$ . We classify it as hard. But if the exception comes once in  $10^{10}$  times then for any single instance the chance of a quadratic runtime is awfully good.

In short, a naive view of our theory suggests that instances of  $\text{NP}$  hard problems

are going to be too slow to solve except in extremely small cases, in toy assignments. This view is wrong. One example of an NP complete problem for which there are available very capable algorithms is the Traveling Salesman problem.



London pubs, via Google Earth

There are algorithms for Traveling Salesman that can in a reasonable time find solutions for problem instances involving 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. An example is that recently a group of applied mathematicians solved the minimal pub crawl, the shortest route to visit all 24 727 UK pubs. The optimal tour is 45 495 239 meters. The algorithm took 305.2 CPU days, running in parallel on up to 48 cores on Linux servers. That is a lot of computing but it is also a lot of pubs — this is not a toy.

Another group solved the Traveling Salesman instance of visiting all 24 978 cities in Sweden, giving a tour of about 72 500 kilometers. The approach was to find a nearly-best solution and then use that to find the best one. The final stages, that improved the lower bound by 0.000 023 percent, required eight years of computation time running in parallel on a network of Linux workstations.

There are a number of systems for solving the Traveling Salesman problem that are widely available. An example is that the Free mathematics system *Sage* includes one. Here is a brief example. It uses a graph that is sure to have a solution, and then we display the solution as an adjacency matrix. For more, see the documentation.

```
sage: g = graphs.HeawoodGraph()
sage: tsp = g.traveling_salesman_problem()
sage: tsp.adjacency_matrix()
[0 0 0 0 1 0 0 0 0 0 0 1]
[0 0 1 0 0 0 0 0 0 1 0 0]
[0 1 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 1 0]
[0 0 0 1 0 0 0 0 0 1 0 0 0]
[1 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 1 0]
[0 0 1 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 1]
[0 0 0 0 1 0 0 0 0 0 1 0 0]
[0 1 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 1 0]
```



Sweden tour

In summary, even though the theory says that a problem is hard does not mean that it cannot be attacked for non-toy instances. See also the next extra section.

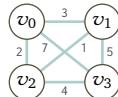
## V.B Exercises

B.1 Critique this from social media: “The Traveling Salesman problem is NP hard. That means that algorithms exist that solve the problem but these algorithms are

very slow. So for an input of size 100 you may already have to wait hundreds of years.”

B.2 A naive implementation is to try every possible circuit. If there are  $n$ -many cities then how many circuits are there?

B.3 Use the exhaustive search of all possible circuits to find the shortest Travelling Salesman Problem solution for a circuit involving this graph.



## EXTRA

### V.C *SAT* solvers

Our characterization of NP complete problems as quite hard may give the impression that for such a problem any algorithm is terribly slow. But the situation is more nuanced than that. The prior extra section discusses how a problem may have a best algorithm that is super polynomial but in practice the problem is tractable. Problems may have occasional hard instances — black holes where the computer goes in and does not come out — but on most instances we can find the answer in a reasonable time. In this section we will demonstrate a program to solve *SAT*, a *SAT solver*, and show how to use that as a general oracle.

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  $\mathcal{L}_0$ . We will take  $\mathcal{L}_1$  to be the Sudoku problem and  $\mathcal{L}_0$  to be *SAT*, so that we consider *Sudoku*  $\leq_p$  *SAT*.

As a review: an instance of a *Sudoku* problem is a 9-by-9 array with some of the cells already filled in. An example is below. We’ve solved it when we fill in the all of the blanks while satisfying some restrictions. Each row must contain each of the numbers 1–9. Likewise each column must contain 1–9. The same criteria applies to the nine subsquares.<sup>†</sup>

---

<sup>†</sup>To characterize the computational complexity of a problem such as *Sudoku* we must be able to describe how an algorithm’s use of some resource grows with its input size. For that we must frame the problem in a way that allows the input size to grow. Instead of eighty one variables  $x_{1,1}, \dots x_{9,9}$  we could generalize to have an arbitrary number,  $x_1, \dots x_n$ . Instead of those variables taking on the values 1–9 we could give them a value in  $1-k$ ; we could also call these ‘colors’ instead of ‘values’. And, in place of rows, columns, and subsquares that are driven by the geometry, an instance of the problem could have arbitrary size- $k$  sets,  $S_i \subseteq \{x_1, \dots x_k\}$ . With that, we can show that *Sudoku* is NP-complete. But having noted this, in this section we will ignore it and limit our attention to the traditional  $9 \times 9$  board.

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

For the reduction Soduku  $\leq_p \text{SAT}$  we must produce a function that inputs game boards and outputs Propositional Logic expressions. Recall that expressions such as  $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$  or  $(x_1 \vee x_2 \vee \neg x_3) \wedge x_4 \wedge (\neg x_2 \vee x_3)$  is in Conjunctive Normal form, CNF. These are the conjunction of clauses, where each clause is the disjunction of literals (a literal is either a single atom such as  $X$ , or the negation of an atom such as  $\neg X$ ). The SAT solver can insist input must be in this form without fear of missing anything because for any Boolean function there is a CNF expression that gives that behavior. More is in Appendix C.

The SAT solver that we use needs its input in a file that is formatted to a standard called DIMACS. It starts with a few header lines. First come comment lines, beginning with the character c. Then comes the problem line, starting with a p followed by a space, followed by the problem type, which for us is cnf, followed by the number of variables, and then followed by the number of clauses. The rest of the file consists of the clause descriptions. To describe a clause we list its indices. Thus we describe  $x_2 \vee x_5 \vee x_6$  with 2 5 6. Negations are described with negatives, so  $\neg x_5 \vee x_7 \vee \neg x_9$  matches -5 7 -9. Then these clause descriptions are separated by 0's.<sup>†</sup>

We will have many variables. For instance, for the row 1, column 1 entry, we will have a Boolean variable  $x_{1,1,1}$ , a variable  $x_{1,1,2}$ , etc., up to  $x_{1,1,9}$ . Only one of these nine will be  $T$  and the rest will be  $F$ . The variable  $x_{1,1,v}$  is  $T$  if in our solution the number in row 1 and column 1 is  $v$ . Otherwise this variable is  $F$ .

Thus there is a variable  $x_{r,c,v}$  for each row, column, and value triple  $r, c, v \in \{1, \dots, 9\}$ . That's a lot of variables,  $3^3 = 729$  of them.

We will work in  $x_{r,c,v}$ 's but we need a way to convert each to some  $x_k$ , and vice versa. 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 in row-number, column-number. If negative, then
; ; the predicate is to be negated.
; ; If variable-value < 0 then this routine uses the absolute value to compute the basic varnum,
; ; but returns the negative of the polynomial (indicating that the predicate is negated).
```

<sup>†</sup>Since the format uses 0 as the clause separator, we don't follow our usual practice of starting with the first variable named  $x_0$ . Rather, we start with  $x_1$ .

```
(define (triple->varnum row-number column-number variable-value)
  (let ([a-value (+ (* 81 (- (abs variable-value) 1))
                    (* 9 (- row-number 1)))
         (- column-number 1)
         1)]) ; add 1 because DIMACS uses 0 to terminate clauses
    (if (negative? variable-value)
        (* -1 a-value)
        a-value)))

;; varnum->triple From the variable number, return the associated row, column, and value
(define (varnum->triple v)
  (let* ([offset (- (abs v) 1)]
     [variable-value (quotient offset 81)]
     [vv-removed (- offset (* 81 variable-value))])
    [row-number (quotient vv-removed 9)]
    [column-number (remainder vv-removed 9)])
  (if (negative? v)
      (list (+ 1 row-number) (+ 1 column-number) (* -1 (+ 1 variable-value)))
      (list (+ 1 row-number) (+ 1 column-number) (+ 1 variable-value))))))
```

Here are a couple of example conversions using these routines, to give a feel for what's happening.

```
> (triple->varnum 3 4 5)
346
> (varnum->triple 346)
'(3 4 5)
```

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. It is as though we bought a puzzle book and opened first to the introduction describing the rules, and later opened to the page containing the specific partial board. To describe the general rules we need lot of clauses describing relationships among the variables, such as that exactly one of  $x_{1,1,1}, x_{1,1,2}, \dots, x_{1,1,9}$  is  $T$ . There are too many of these to write by hand so we will get the computer to write them.

At the top of the file are some constants that are described with words to 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))
```

Next we describe the relationships between the variables. For each row there are nine restrictions. For instance, to express that the first row contains an entry with the value 2, we need this clause.

$$x_{1,1,2} \vee x_{1,2,2} \vee x_{1,3,2} \vee x_{1,4,2} \vee x_{1,5,2} \vee x_{1,6,2} \vee x_{1,7,2} \vee x_{1,8,2} \vee x_{1,9,2}$$

The Racket code produces this corresponding list: ( (1 1 2) (1 2 2) (1 3 2) (1 4 2) (1 5 2) (1 6 2) (1 7 2) (1 8 2) (1 9 2) ). To express all of the row restrictions we need one such list for each row and value.

```
;; row-restrictions Return list of list of triples, each list of triples meaning
;; that each row has to have each value 1-9.
(define (row-restrictions)
  (define (one-row-one-value row-number variable-value)
    (for/list ([column-number ONETONINE])
      (list row-number column-number variable-value)))

  (for*/fold ([accumulator '()])
    #:result (reverse accumulator))
    ([variable-value ONETONINE]
     [row-number ONETONINE])
    (cons (one-row-one-value row-number variable-value) accumulator)))
```

Running that routine produces a list of lists. This is a typical member, requiring that at least one entry in row 3 must be an 8.

```
((3 1 8) (3 2 8) (3 3 8) (3 4 8) (3 5 8) (3 6 8) (3 7 8) (3 8 8) (3 9 8))
```

The column restrictions match the row ones.

```
;; column-restrictions Return list of list of triples, each list of triples meaning
;; that each row has to have each value 1-9.
(define (column-restrictions)
  (define (one-column-one-value column-number variable-value)
    (for/list ([row-number ONETONINE])
      (list row-number column-number variable-value)))

  (for*/fold ([accumulator '()])
    #:result (reverse accumulator))
    ([variable-value ONETONINE]
     [column-number ONETONINE])
    (cons (one-column-one-value column-number variable-value) accumulator)))
```

Here is a typical line, requiring that at least one entry in column 7 must be an 8.

```
((1 7 8) (2 7 8) (3 7 8) (4 7 8) (5 7 8) (6 7 8) (7 7 8) (8 7 8) (9 7 8))
```

For the subsquares, the form of the code is a bit different but the restrictions are the same in principle.

```
;; 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 a box restriction taken at random, 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 those restrictions 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.

We need one more set of restrictions, that no entry can contain two values. We will add clauses like  $\neg x_{3,4,1} \vee \neg x_{3,4,2}$ , meaning that the row 3 and column 4 entry cannot be both a 1 and a 2 (it could of course be neither).

```
;; entry-restrictions Return list of list of triples, each list of triples meaning
;; that each entry cannot be two separate values 1-9.
(define (entry-restrictions)
  (for*/list ([row-number ONETONINE]
             [column-number ONETONINE]
             [variable-value1 ONETONINE]
             [variable-value2 ONETONINE]
             #:unless (>= variable-value1 variable-value2))
    (list (list row-number column-number (* -1 variable-value1))
          (list row-number column-number (* -1 variable-value2)))))
```

This is one of the resulting lines, enforcing that the entry in row 8 and column 8 cannot be both 6 and 9 (again, the negative means logical negation).

```
((8 8 -6) (8 8 -9))
```

To finishes we must specify the initial board. For example, to tell the *SAT* solver that there is a 9 in row 1 and column 3 we include the one-literal clause  $x_{1,3,9}$ . Here are the first few lines of that routine.

```
;; INITIAL-CLAUSES The given layout of the board. Each row is a list with a
;; triple: row number, column number, integer.
(define INITIAL-CLAUSES
  (list (list '(1 3 9)) ; there is a 9 in position (1,3)
        (list '(1 8 1))
        (list '(1 9 5)))
```

Now we write all of these to a file.

```
;; CLAUSES The list of all clauses, including those auto generated.
(define CLAUSES
  (append INITIAL-CLAUSES (entry-restrictions)
          (row-restrictions) (column-restrictions) (box-restrictions)))

(define FILE-PREAMBLE
  (list (format "c ~a\n" FILENAME)
        "c DIMACS format file for SAT solver\n"
        (format "c ~a Jim Hefferon, hefferon.net. Public Domain.\n"
                (date->string (current-date)))
        (format "p cnf ~a ~a\n" (* 9 9 9) (length CLAUSES)))

(define FILE-LINES
  (append
    FILE-PREAMBLE
    (produce-clauses CLAUSES)))
```

```
;; dump-to-file Drop the clauses to the file
(define (dump-to-file)
  (define (dump-lines outfile)
    (for ([ln FILE-LINES])
      (display ln outfile)))

  (call-with-output-file* FILENAME dump-lines #:mode 'text #:exists 'replace))
```

The first few lines of the result `sudoku.cnf` look like this. This is the DIMACS format file.

```
c sudoku.cnf
c DIMACS format file for SAT solver
c 2022-01-16 Jim Hefferon, hefferon.net. Public Domain.
p cnf 729 3197
651 0
8 0
333 0
334 0
256 0
663 0
502 0
262 0
```

We can now run the *SAT* solver, MiniSat (Eén and Sörensson 2005).

```
ftpmaint@millstone:~/Documents/computing/src/scheme/complexity$ minisat sudoku.cnf sudoku.out
WARNING: for repeatability, setting FPU to use double precision
===== [ Problem Statistics ] =====
| |
| Number of variables: 729 |
| Number of clauses: 2044 |
| Parse time: 0.00 s |
| Eliminated clauses: 0.00 Mb |
| Simplification time: 0.00 s |
| |
===== [ Search Statistics ] =====
| Conflicts | ORIGINAL | 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
```

Here is the output (there are 729 numbers in the line but this page fits only a few).

```
ftpmaint@millstone:~/Documents/computing/src/scheme/complexity$ cat sudoku.out
SAT
-1 -2 -3 -4 -5 -6 -7 8 -9 -10 -11 12 -13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 24 -25 -26
```

To see the solution, here is a sample.

```
> (varnum->triple 8)
'(1 8 1)
```

Thus, the row 1 and column 8 entry holds the value 1.

Some simple routines built on what is above show the entire board.

```
(require "sudoku-dimacs.rkt")

(define (solution-triples solution)
  (map varnum->triple (hash-keys solution)))

(define (show-board triples)
  (map (lambda (x)
          (set-board-entry! BOARD x))
       triples)
  BOARD)

(define (show-solution-triples list-of-triples)
  (define (compare-triples x y)
    (cond
      [(< (first x) (first y)) #t]
      [(> (first x) (first y)) #f]
      [(< (second x) (second y)) #t]
      [(> (second x) (second y)) #f]
      [(< (third x) (third y)) #t]
      [else #f]))

  (sort
   list-of-triples
   compare-triples))

;; read the output from MiniSat
(define (read-minisat-output)
  (define (get-two-lines input-port)
    (let ([head-line (read-line input-port)]
         [data-line (read-line input-port)])
      (list head-line data-line)))

  (call-with-input-file "sudoku.out" get-two-lines))

(define (process-minisat-output pr)
  (let* ([data-line (second pr)]
         [number-list (map string->number (string-split data-line))]
         [triple-list (map varnum->triple number-list)]
         [filtered-triple-list (filter (lambda (x) (positive? (third x))) triple-list)])
    filtered-triple-list))

(define (show-solved-board)
  (show-board (cdr (show-solution-triples (process-minisat-output (read-minisat-output))))))
```

Running that final routine gives this.

```
> (show-solved-board)
'#( #(2 6 9 3 7 8 4 1 5)
  #(5 8 1 4 2 9 7 6 3)
  #(4 7 3 5 6 1 9 2 8)
  #(8 1 2 7 4 5 3 9 6)
  #(3 5 7 1 9 6 2 8 4)
  #(6 9 4 8 3 2 1 5 7)
  #(1 3 5 9 8 4 6 7 2)
```

```
#(9 4 6 2 5 7 8 3 1)  
#(7 2 8 6 1 3 5 4 9))
```

In summary, we can in reasonable time solve instances of  $SAT$  that are not toy exercises. They are large enough that to write the CNF form we had to resort to code.

### V.C Exercises

C.1 This board is billed online as a hard Sudoku. Use the routines from this section to solve it.

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



Part Four  
Appendices

## APPENDIX A    STRINGS

An **alphabet** is a nonempty and finite set of **symbols** (sometimes called **tokens**). We write symbols in a distinct typeface, as in `1` or `a`, because the alternative of quoting them would be clunky.<sup>†</sup> 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  $\varepsilon$ .

One potentially surprising aspect of a symbol is that it may contain more than one letter. For instance, a programming language may have `if` as a symbol, meaning that it is indecomposable into separate letters. Another example is that the Racket alphabet contains the symbols `or` and `car`, as well as allowing variable names such as `x`, or `lastname`. An example of a string is `(or a ready)`, which is a sequence of five alphabet elements,  $\langle(), \text{or}, \text{a}, \text{ready}, ()\rangle$ .

Traditionally, we denote an alphabet with the Greek letter  $\Sigma$ . In this book we will name strings with lower case Greek letters and denote the items in the string with the associated lower case roman letter, as in  $\sigma = \langle s_0, \dots, s_{n-1} \rangle$  and  $\tau = \langle t_0, \dots, t_{m-1} \rangle$ . The **length** of the string  $\sigma$ ,  $|\sigma|$ , 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 write  $\sigma[i]$ . A convenience of this form is that we use  $\sigma[-1]$  for the final character,  $\sigma[-2]$  for the one before it, etc. We also write  $\sigma[i:j]$  for the substring between terms  $i$  and  $j$ , including the  $i$ -th term but not the  $j$ -th, and we write  $\sigma[i:]$  for the tail substring that starts with term  $i$ .

The notations such as diamond brackets and commas are ungainly. For small-scale examples and exercises, we use the shortcut of working with alphabets of single-character symbols and then writing strings by omitting the brackets and commas. That is, we write  $\sigma = abc$  instead of  $\langle a, b, c \rangle$ .<sup>‡</sup> This convenience comes with the disadvantage that without the diamond brackets the empty string is just nothing, which is why we use the separate symbol  $\varepsilon$ .<sup>§</sup>

The alphabet consisting of the bit characters is  $\mathbb{B} = \{0, 1\}$ . Strings over this alphabet are **bitstrings** or **bit strings**.<sup>||</sup>

Where  $\Sigma$  is an alphabet, for  $k \in \mathbb{N}$  the set of length  $k$  strings over that alphabet is  $\Sigma^k$ . The set of strings over  $\Sigma$  of any finite length is  $\Sigma^* = \bigcup_{k \in \mathbb{N}} \Sigma^k$ . The asterisk symbol is the **Kleene star**, read aloud as “star.”

Strings are simple so there are only a few operations. Let  $\sigma = \langle s_0 \dots s_{n-1} \rangle$  and  $\tau = \langle t_0, \dots, t_{m-1} \rangle$  be strings over an alphabet  $\Sigma$ . The **concatenation**  $\sigma \cdot \tau$  or  $\sigma \tau$  appends the second sequence to the first:  $\sigma \cdot \tau = \langle s_0 \dots s_{n-1}, t_0, \dots, t_{m-1} \rangle$ . Where

<sup>†</sup>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. <sup>‡</sup>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  $\langle a, a, a \rangle$ . <sup>§</sup>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. <sup>||</sup>Some authors consider infinite bitstrings but ours will always be finite.

$\sigma = \tau_0 \hat{\cdot} \cdots \hat{\cdot} \tau_{m-1}$ , we say that  $\sigma$  **decomposes** into the  $\tau$ 's, and that each  $\tau_i$  is a **substring** of  $\sigma$ . The first substring,  $\tau_0$ , is a **prefix** of  $\sigma$ . The last,  $\tau_{m-1}$ , is a **suffix**.

A **power** or **replication** of a string is an iterated concatenation with itself, so that  $\sigma^2 = \sigma \hat{\cdot} \sigma$  and  $\sigma^3 = \sigma \hat{\cdot} \sigma \hat{\cdot} \sigma$ , etc. We write  $\sigma^1 = \sigma$  and  $\sigma^0 = \varepsilon$ . The **reversal**  $\sigma^R$  of a string takes the symbols in reverse order:  $\sigma^R = \langle s_{n-1}, \dots, s_0 \rangle$ . The empty string's reversal is  $\varepsilon^R = \varepsilon$ .

For example, let  $\Sigma = \{a, b, c\}$  and let  $\sigma = abc$  and  $\tau = bbaac$ . Then the concatenation  $\sigma\tau$  is  $abcbbaac$ . The third power  $\sigma^3$  is  $abcabcabc$ , and the reversal  $\tau^R$  is  $caabb$ . A **palindrome** is a string that equals its own reversal. Examples are  $\alpha = abba$ ,  $\beta = cdc$ , and  $\varepsilon$ .

## Exercises

- A.1 Let  $\sigma = 10110$  and  $\tau = 110111$  be bit strings. Find each. (A)  $\sigma \hat{\cdot} \tau$  (B)  $\sigma \hat{\cdot} \tau \hat{\cdot} \sigma$  (C)  $\sigma^R$  (D)  $\sigma^3$  (E)  $\theta^3 \hat{\cdot} \sigma$
- A.2 Let the alphabet be  $\Sigma = \{a, b, c\}$ . Suppose that  $\sigma = ab$  and  $\tau = bca$ . Find each. (A)  $\sigma \hat{\cdot} \tau$  (B)  $\sigma^2 \hat{\cdot} \tau^2$  (C)  $\sigma^R \hat{\cdot} \tau^R$  (D)  $\sigma^3$
- A.3 Let  $\mathcal{L} = \{\sigma \in \mathbb{B}^* \mid |\sigma| = 4 \text{ and } \sigma \text{ starts with } 0\}$ . How many elements are in that language?
- A.4 Suppose that  $\Sigma = \{a, b, c\}$  and that  $\sigma = abcccbba$ . (A) Is  $abcb$  a prefix of  $\sigma$ ? (B) Is  $ba$  a suffix? (C) Is  $bab$  a substring? (D) Is  $\varepsilon$  a suffix?
- A.5 What is the relation between  $|\sigma|$ ,  $|\tau|$ , and  $|\sigma \hat{\cdot} \tau|$ ? You must justify your answer.
- A.6 The operation of string concatenation follows a simple algebra. For each of these, decide if it is true. If so, prove it. If not, give a counterexample. (A)  $\alpha \hat{\cdot} \varepsilon = \alpha$  and  $\varepsilon \hat{\cdot} \alpha = \alpha$  (B)  $\alpha \hat{\cdot} \beta = \beta \hat{\cdot} \alpha$  (C)  $\alpha \hat{\cdot} \beta^R = \beta^R \hat{\cdot} \alpha^R$  (D)  $\alpha^{RR} = \alpha$  (E)  $\alpha^{iR} = \alpha^i$
- A.7 Show that string concatenation is not commutative, that there are strings  $\sigma$  and  $\tau$  so that  $\sigma \hat{\cdot} \tau \neq \tau \hat{\cdot} \sigma$ .
- A.8 In defining decomposition above we have ' $\sigma = \tau_0 \hat{\cdot} \cdots \hat{\cdot} \tau_{n-1}$ ', without parentheses on the right side. This takes for granted that the concatenation operation is associative, that no matter how we parenthesize it we get the same string. Prove this. Hint: use induction on the number of substrings,  $n$ .
- A.9 Prove that this constructive definition of string power is equivalent to the one above.

$$\sigma^n = \begin{cases} \varepsilon & - \text{if } n = 0 \\ \sigma^{n-1} \hat{\cdot} \sigma & - \text{if } n > 0 \end{cases}$$

## APPENDIX B FUNCTIONS

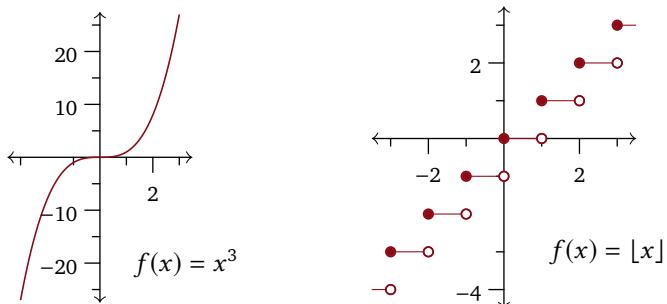
A function is an input-output relationship: each input is associated with a unique output. An example is the association of each input natural number with an output number that is twice as big. Another is the association of each string of characters with the length of that string. A third is the association of each polynomial  $a_nx^n + \dots + a_1x + a_0$  with a Boolean value  $T$  or  $F$ , depending on whether 1 is a root of that polynomial.

For the precise definition, fix two sets, a **domain**  $D$  and a **codomain**  $C$ . A **function**, or **map**,  $f: D \rightarrow C$  is a set of pairs  $(x, y) \in D \times C$ , subject to the restriction of being **well-defined**, that every  $x \in D$  appears in one and only one pair (more on this is below). We write  $f(x) = y$  or  $x \mapsto y$  and say ‘ $x$  maps to  $y$ ’. Note the difference between the arrow symbols in  $f: D \rightarrow C$  and  $x \mapsto y$ . We say that  $x$  is an **input** or **argument** to the function, and that  $y$  is an **output** or **value**.

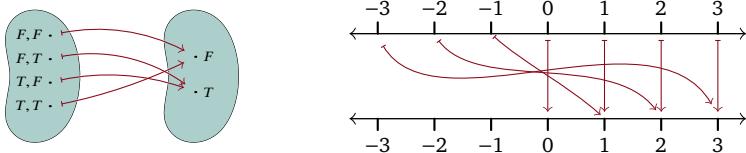
An important point is that a function isn’t a formula or rule. The function that gives the US presidents,  $f(0) = \text{George Washington}$ , etc., has no sensible formula and isn’t determined by any rule less complex than an exhaustive listing of cases. The same holds for a function that returns winners of the US World Series, including next year’s winner. 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 the inputs to the outputs with a calculation then that’s great, but that is not required.

Some functions take more than one input, such as  $\text{dist}(x, y) = \sqrt{x^2 + y^2}$ . We say that  $\text{dist}$  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, as with  $x = (3, 5)$ , then we often drop the parentheses, so that instead of  $f(x) = f((3, 5))$  we write  $f(3, 5)$ .

**Pictures** We often illustrate functions using the familiar  $xy$  axes; here are graphs of  $f(x) = x^3$  and  $f(x) = |x|$ .



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.



On the right is a variant of the bean diagram, using the number line to show the absolute value function mapping integers to integers.

**Codomain and range** Where  $S \subseteq D$  is a subset of the domain, its **image** is the set  $f(S) = \{f(s) \mid s \in S\}$ . Thus, under the squaring function the image of  $S = \{0, 1, 2\}$  is  $f(S) = \{0, 1, 4\}$ . Under the floor function  $g: \mathbb{R} \rightarrow \mathbb{R}$  given by  $g(x) = \lfloor x \rfloor$ , the image of the positive reals is the set of natural numbers.

The image of the entire domain  $D$  is the function's **range**,  $\text{ran}(f) = f(D) = \{f(d) \mid d \in D\}$ . For instance, the range of the floor function shown above is the set of integers.

Note the difference between the range and the codomain; the codomain is a convenient superset. An example is that for the function with real inputs  $f(x) = \sqrt{2x^4 + 2x^2 + 15}$ , we would usually be content to note that the polynomial is always nonnegative and so the output is real, writing  $f: \mathbb{R} \rightarrow \mathbb{R}$ , where the second  $\mathbb{R}$  is the codomain, rather than troubling to find its exact range.

**Domain** Sometimes a function's domain requires attention. Examples are that  $f(x) = 1/x$  is undefined at  $x = 0$  and that the function defined by the infinite series  $g(r) = 1 + r + r^2 + \dots$  diverges when  $r$  is outside the interval  $(-1..1)$ . Formally, when we define the function we must specify the domain to eliminate such problems, for instance by defining the domain of  $f$  as  $\mathbb{R} - \{0\}$ . However, we are usually casual about this, expecting that a reader will understand which inputs are an issue.

We sometimes have a function  $f: D \rightarrow C$  and want to cut the domain back to some subset  $S \subseteq D$ . The **restriction**  $f \upharpoonright_S$  is the function with domain  $S$  and codomain  $C$  defined by  $f \upharpoonright_S(x) = f(x)$ .

**In our subject** The Theory of Computation sometimes uses these terms in a way that is related to what's above but also is in conflict with it. Often when we study a function  $f$  we will fix a convenient set of inputs  $D$ , such as the set of all strings  $\Sigma^*$  or the natural numbers  $\mathbb{N}$ , and refer to  $D$  as the function's domain although it may be that  $f$  is 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, in this terminology the 'partial' is redundant since any function is partial, including a total function. But often 'partial' connotes that  $f$  is not defined for one or more  $d \in D$ .

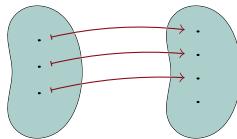
**Well-defined** The definition of a function contains the condition that each domain element maps to one and only one codomain element,  $y = f(x)$ . We refer to this condition by saying that functions are **well-defined**.

When we are considering a relationship between  $x$ 's and  $y$ 's and asking if it is a function, well-definedness is typically what is at issue.<sup>†</sup> For instance, consider the set of ordered pairs  $(x, y)$  where  $y^2 = x$ . If  $x = 9$  then both  $y = 3$  and  $y = -3$  are related to  $x$  so this is not a functional relationship—it is not well-defined—because  $x = 9$  does not have one and only one associated  $y$ . Another example is that when setting up a company's email we may decide to use each person's first initial and last name, but there could easily be more than one, say, mdouglas, making the relation (email, person) be not well-defined.

For a function  $f : \mathbb{R} \rightarrow \mathbb{R}$  that is suitable for graphing on  $xy$  axes, visual proof of well-definedness is that for any  $x$  in the domain, the vertical line through  $x$  intercepts  $f$ 's graph in exactly one point.

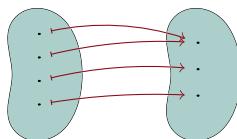
**One-to-one and onto** The definition of function has an asymmetry: among the ordered pairs  $(x, y)$ , it requires that each domain element  $x$  be in one pair and only one pair, but it does not require the same of the codomain elements.

A function is **one-to-one** (or **1-1** or an **injection**) if each codomain element  $y$  is in at most one pair. The function below is one-to-one because for every element  $y$  in the codomain, the bean on the right, there is at most one arrow ending at  $y$ .



The most common way to prove that a function  $f$  is one-to-one is to assume that  $f(x_0) = f(x_1)$  and then argue that therefore  $x_0 = x_1$ . If a function is suitable for graphing on  $xy$  axes then visual proof is that for any  $y$  in the codomain, the horizontal line at  $y$  intercepts the graph in at most one point.

A function is **onto** (or a **surjection**) if each codomain element  $y$  is in at least one pair. Thus, a function is onto if its codomain equals its range. The function below is onto because every element in the codomain bean has at least one arrow ending at it.



The most common way to verify that a function is onto is to start with a generic (that is, arbitrary) codomain element  $y$  and then exhibit a domain element  $x$  that maps to it. If a function is suitable for graphing on  $xy$  axes then visual proof is

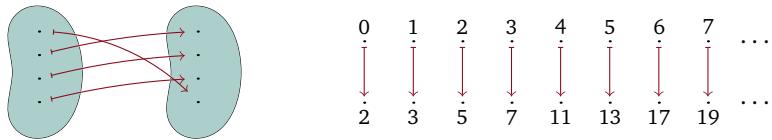
---

<sup>†</sup>Sometimes people say that they are, “checking that the function is well-defined.” In a strict sense this is confused, because if it is a function then it is by definition well-defined. However, while all tigers have stripes, we do sometimes say “striped tiger.” Natural language is funny that way.

that for any  $y$  in the codomain, the horizontal line at  $y$  intercepts the graph in at least one point.

As the above pictures suggest, where the domain and codomain are finite, when there is a function  $f: D \rightarrow C$  then we can conclude that the number of elements in the domain is less than or equal to the number in the codomain. Further, if the function is onto then the number of elements in the domain equals the number in the codomain if and only if the function is one-to-one.

**Correspondence** A function is a **correspondence** (or **bijection**) if it is both one-to-one and onto. The picture on the left shows a correspondence between two finite sets, both with four elements, and the picture on the right shows a correspondence between the natural numbers and the primes.



The most common way to verify that a function is a correspondence is to separately verify that it is one-to-one and that it is onto. Where the function is  $f: \mathbb{R} \rightarrow \mathbb{R}$ , so it can be graphed on  $xy$  axes, visual proof that it is a correspondence is that for any  $y$  in the codomain, the horizontal line at  $y$  intersects the graph in exactly one point.

As the picture above on the left suggests, where the domain and codomain are finite, if a function is a correspondence then its domain has the same number of elements as its codomain.

**Composition and inverse** If  $f: D \rightarrow C$  and  $g: C \rightarrow B$  then their **composition**  $g \circ f: D \rightarrow B$  is defined by  $g \circ f(d) = g(f(d))$ . For instance, the real functions  $f(x) = x^2$  and  $g(x) = \sin(x)$  combine to give  $g \circ f = \sin(x^2)$ .

Composition does not commute. Using the functions from the prior paragraph,  $f \circ g = \sin(x^2)$  and  $f \circ g = (\sin x)^2$  are different; for instance they are unequal when  $x = \pi$ . Composition can fail to commute more dramatically: if  $f: \mathbb{R}^2 \rightarrow \mathbb{R}$  is given by  $f(x_0, x_1) = x_0$ , and  $g: \mathbb{R} \rightarrow \mathbb{R}$  is  $g(x) = x$ , then  $g \circ f(x_0, x_1) = x_0$  is perfectly sensible but composition in the other order is not even defined.

The composition of one-to-one functions is one-to-one, and the composition of onto functions is onto. It follows then that the composition of correspondences is a correspondence.

An **identity function**  $\text{id}: D \rightarrow D$  is given by  $\text{id}(d) = d$  for all  $d \in D$ . It acts as the identity element in function composition, so that if  $f: D \rightarrow C$  then  $f \circ \text{id} = f$  and if  $g: C \rightarrow D$  then  $\text{id} \circ g = g$ . As well, if  $h: D \rightarrow D$  then  $h \circ \text{id} = \text{id} \circ h = h$ .

Given  $f: D \rightarrow C$ , if  $g \circ f$  is the identity function then  $g$  is a **left inverse** function of  $f$ , or what is the same thing,  $f$  is a **right inverse** of  $g$ . If  $g$  is both a left and right inverse of  $f$  then we simply say that it is an **inverse** (or **two-sided inverse**) of  $f$

and denoted it as  $f^{-1}$ . If a function has an inverse then that inverse is unique. A function has a two-sided inverse if and only if it is a correspondence.

### Exercises

B.1 Let  $f, g: \mathbb{R} \rightarrow \mathbb{R}$  be  $f(x) = 3x + 1$  and  $g(x) = x^2 + 1$ . (A) Show that  $f$  is one-to-one and onto. (B) Show that  $g$  is not one-to-one and not onto.

B.2 Show each of these.

(A) Let  $g: \mathbb{R}^3 \rightarrow \mathbb{R}^2$  be the projection map  $(x, y, z) \mapsto (x, y)$  and let  $f: \mathbb{R}^2 \rightarrow \mathbb{R}^3$  be  $(x, y) \mapsto (x, y, 0)$ . Then  $g$  is a left inverse of  $f$  but not a right inverse.

(B) The function  $f: \mathbb{Z} \rightarrow \mathbb{Z}$  given by  $f(n) = n^2$  has no left inverse.

(C) Where  $D = \{0, 1, 2, 3\}$  and  $C = \{10, 11\}$ , the function  $f: D \rightarrow C$  given by  $0 \mapsto 10, 1 \mapsto 11, 2 \mapsto 10, 3 \mapsto 11$  has more than one right inverse.

B.3 (A) Where  $f: \mathbb{Z} \rightarrow \mathbb{Z}$  is  $f(a) = a + 3$  and  $g: \mathbb{Z} \rightarrow \mathbb{Z}$  is  $g(a) = a - 3$ , show that  $g$  is inverse to  $f$ .

(B) Where  $h: \mathbb{Z} \rightarrow \mathbb{Z}$  is the function that returns  $n + 1$  if  $n$  is even and returns  $n - 1$  if  $n$  is odd, find a function inverse to  $h$ .

(C) If  $s: \mathbb{R}^+ \rightarrow \mathbb{R}^+$  is  $s(x) = x^2$ , find its inverse.

B.4 Fix  $D = \{0, 1, 2\}$  and  $C = \{10, 11, 12\}$ . Let  $f, g: D \rightarrow C$  be  $f(0) = 10, f(1) = 11, f(2) = 12$ , and  $g(0) = 10, g(1) = 10, g(2) = 12$ . Then: (A) verify that  $f$  is a correspondence (B) construct an inverse for  $f$  (C) verify that  $g$  is not a correspondence (D) show that  $g$  has no inverse.

B.5 (A) Prove that a composition of one-to-one functions is one-to-one. (B) Prove that a composition of onto functions is onto. With the prior item, this gives that a composition of correspondences is a correspondence. (C) Prove that if  $g \circ f$  is one-to-one then  $f$  is one-to-one. (D) Prove that if  $g \circ f$  is onto then  $g$  is onto. (E) If  $g \circ f$  is onto, must  $f$  be onto? If it is one-to-one, must  $g$  be one-to-one?

B.6 Prove.

(A) A function  $f$  has an inverse if and only if  $f$  is a correspondence.

(B) If a function has an inverse then that inverse is unique.

(C) The inverse of a correspondence is a correspondence.

(D) If  $f$  and  $g$  are each invertible then so is  $g \circ f$ , and  $(g \circ f)^{-1} = f^{-1} \circ g^{-1}$ .

B.7 Prove these for a function  $f$  with a finite domain  $D$ . They imply that corresponding finite sets have the same size. Hint: for each, you can do induction on either  $|D|$  or  $|\text{ran}(f)|$ .

(A)  $|\text{ran}(f)| \leq |D|$

(B) If  $f$  is one-to-one then  $|\text{ran}(f)| = |D|$ .

## APPENDIX C PROPOSITIONAL LOGIC

A **proposition** is a statement that has a Boolean value, that is, it is either true or false, which we write  $T$  or  $F$ . For instance, ‘7 is odd’ and ‘ $8^2 - 1 = 127$ ’ are

propositions, with values  $T$  and  $F$ . In contrast, ‘ $x$  is a perfect square’ is not a proposition because for some  $x$  it is  $T$  while for others it is not.

We can operate on propositions, including negating as with ‘it is not the case that 8 is prime’, or taking the conjunction of two propositions as with ‘5 is prime and 7 is prime’. The **truth tables** below define the behavior of **not** (also called **negation**), **and** (also called **conjunction**), and **or** (also called **disjunction**).

		not $P$		P and Q		P or Q	
		$P$	$\neg P$	$P$	$Q$	$P \wedge Q$	$P \vee Q$
$F$	$T$			$F$	$F$	$F$	$F$
$T$	$F$			$F$	$T$	$F$	$T$
				$T$	$F$	$F$	$T$
				$T$	$T$	$T$	$T$

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’s third column, third row:  $F$ .

In some fields the practice is to write 0 where we write  $F$  and 1 in place of  $T$ .

The advantage of using symbols over writing the sentences out is that we can express more things. For instance, if  $P$  stands for ‘7 is odd’,  $Q$  stands for ‘9 is a perfect square’, and  $R$  means ‘11 is prime’ then  $(P \vee Q) \wedge \neg(P \vee (R \wedge Q))$  is too complex to comfortably state in everyday language. We call that a propositional logic **expression** and denote it with a capital Roman letter such as  $E$ .

Truth tables help in working out the behavior of the complex statements by building them up from their components. The table below shows the input/output behavior of  $(P \vee Q) \wedge \neg(P \vee (R \wedge Q))$ .

$P$	$Q$	$R$	$P \vee Q$	$R \wedge Q$	$P \vee (R \wedge Q)$	$\neg(P \vee (R \wedge Q))$	expression
$F$	$F$	$F$	$F$	$F$	$F$	$T$	$F$
$F$	$F$	$T$	$F$	$F$	$F$	$T$	$F$
$F$	$T$	$F$	$F$	$F$	$F$	$T$	$F$
$F$	$T$	$T$	$F$	$F$	$F$	$T$	$F$
$T$	$F$	$F$	$F$	$F$	$F$	$T$	$F$
$T$	$F$	$T$	$F$	$F$	$F$	$T$	$F$
$T$	$T$	$F$	$F$	$F$	$F$	$T$	$F$
$T$	$T$	$T$	$F$	$F$	$F$	$T$	$F$

The three ‘ $\neg$ ’, ‘ $\wedge$ ’, and ‘ $\vee$ ’ are **operators** (or **connectives**). There are other operators; here are two common ones.

$P$	$Q$	$P$ implies $Q$		$P$ if and only if $Q$	
		$P \rightarrow Q$	$P \rightarrow Q$	$P \leftrightarrow Q$	$P \leftrightarrow Q$
$F$	$F$	$T$	$T$	$T$	$T$
$F$	$T$	$T$	$F$	$F$	$F$
$T$	$F$	$F$	$F$	$F$	$F$
$T$	$T$	$T$	$T$	$T$	$T$

Two statements are **equivalent** (or **logically equivalent**) if they have equal

output values whenever we give the same values to the variables. For instance,  $P \rightarrow Q$  is equivalent to  $\neg P \vee Q$ , because if we assign  $P = F, Q = F$  then they both give the value  $T$ , if we assign  $P = F, Q = T$  then they also give the same value, etc. That is, the statements are equivalent when their truth tables have the same final column. We denote equivalence using  $\equiv$ , as with  $P \rightarrow Q \equiv \neg P \vee Q$

The set of formulas describing when statements are equivalent is **Boolean algebra**. For instance, these are the distributive laws

$$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R) \quad P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$$

and these are **DeMorgan's laws**.

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q \quad \neg(P \vee Q) \equiv \neg P \wedge \neg Q$$

The three operators ‘ $\neg$ ’, ‘ $\wedge$ ’, and ‘ $\vee$ ’ form a complete set in that we can reverse the activity above: for any truth table we can use the three to produce an expression whose input/output behavior is that table. In short, we can produce expressions with any desired behavior. Here are two examples.

$P$	$Q$	$E_0$	$P$	$Q$	$R$	$E_1$
$F$	$F$	$T$	$F$	$F$	$F$	$F$
$F$	$T$	$F$	$F$	$F$	$T$	$T$
$T$	$F$	$F$	$F$	$T$	$F$	$T$
$T$	$T$	$F$	$F$	$T$	$T$	$F$
			$T$	$F$	$F$	$T$
			$T$	$F$	$T$	$F$
			$T$	$T$	$F$	$F$
			$T$	$T$	$T$	$F$

To produce  $E_0$ , on the left, focus on the  $T$  row. There,  $P = F$  and  $Q = F$  and so for  $E_0$  we can use the expression  $\neg P \wedge \neg Q$ . For  $E_1$ , on the right, focus on all of the  $T$  rows. Target the second row with  $\neg P \wedge \neg Q \wedge R$ . Target the third row with  $\neg P \wedge Q \wedge \neg R$ . For the fifth use  $P \wedge \neg Q \wedge \neg R$ . Then  $E_1$  joins these clauses with  $\vee$ 's.

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

In a propositional logic expression, a single variable such as  $P$  or  $Q$  is an **atom**. An atom or its negation, such as  $P$  or  $\neg P$ , is a **literal**. A **clause** is a number of literals joined by a connective (we stick to either  $\wedge$  or  $\vee$ ), so that  $\neg P \wedge \neg Q \wedge R$  is a clause.

The form of  $(*)$  above is important. A propositional logic expression is in **Disjunctive Normal form** or **DNF** if it is a disjunction of clauses, where each clause is a conjunction of literals.

Intuition requires that there be a matching approach that starts with the  $F$  rows. We illustrate with  $E_0$ . The second, third, and fourth rows are  $F$ . So  $E_0 \equiv \neg((\neg P \wedge Q) \vee (P \wedge \neg Q) \vee (P \wedge Q))$ . DeMorgan's second law gives  $\neg(\neg P \wedge \neg Q) \wedge \neg(P \wedge \neg Q) \wedge \neg(P \wedge Q)$ . Next DeMorgan's first law gives  $E_0 \equiv (P \vee Q) \wedge (\neg P \vee \neg Q) \wedge (\neg P \vee Q)$ .

**Conjunctive Normal form** or **CNF** is a conjunction of clauses, where each clause is a disjunction of literals. We use CNF more than DNF. With this form an expression evaluates to  $T$  if and only if all of its clauses evaluate to  $T$ . And each clause evaluates to  $T$  if and only if at least one of its literals evaluates to  $T$ .

A **Boolean function** has Boolean inputs, that is, they are either  $T$  or  $F$ , and Boolean outputs. By the prior paragraphs about DNF and CNF, each single-output Boolean function is determined by some Boolean expression.

### Exercises

C.1 Make a truth table for each of these propositions. (A)  $(P \wedge Q) \wedge R$  (B)  $P \wedge (Q \wedge R)$   
 (C)  $P \wedge (Q \vee R)$  (D)  $(P \wedge Q) \vee (P \wedge R)$

C.2 Make a truth table for these. (A)  $\neg(P \vee Q)$  (B)  $\neg P \wedge \neg Q$  (C)  $\neg(P \wedge Q)$   
 (D)  $\neg P \vee \neg Q$

C.3 For the tables below, construct a DNF propositional logic expression: (A) the table on the left, (B) the one on the right.

$P$	$Q$	$R$		$P$	$Q$	$R$	
$F$	$T$						
$F$	$F$	$T$	$T$	$F$	$F$	$T$	$F$
$F$	$T$	$F$	$T$	$F$	$T$	$F$	$T$
$F$	$T$	$T$	$F$	$F$	$T$	$T$	$F$
$T$	$F$	$F$	$F$	$T$	$F$	$F$	$F$
$T$	$F$	$T$	$T$	$T$	$F$	$T$	$F$
$T$	$T$	$F$	$F$	$T$	$T$	$F$	$T$
$T$	$T$	$T$	$F$	$T$	$T$	$T$	$T$

C.4 For the tables in the prior exercise, construct a CNF propositional logic expression: (A) the table on the left, (B) the one on the right.

C.5 There are sixteen binary logical operators. Give all sixteen truth tables, and give the operator's name, such as ' $P \rightarrow Q$ ' or ' $Q \rightarrow P$ '.



## Part Five

### Notes

These are citations 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

*in addition to technical detail, also attends to a breadth of knowledge* S Pinker emphasizes that a liberal approach involves 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

*Entscheidungsproblem* Pronounced *en-SHY-dungs-problem*.

*D Hilbert and W Ackermann* Hilbert was a very prominent mathematician, perhaps the world’s most prominent mathematician, and Ackermann was his student. So they made an impression when they wrote, “[This] must be considered the main problem of mathematical logic” (Hilbert and Ackermann 1950), p 73.

*mathematical statement* Specifically, the statement as discussed by Hilbert and Ackermann comes from a first-order logic (versions of the *Entscheidungsproblem* for other systems had been proposed by other mathematicians). First-order logic differs from propositional logic, the logic of truth tables, in that it allows variables. Thus for instance if you are studying the natural numbers then you can have a Boolean function  $\text{Prime}(x)$ . (In this context a Boolean function is traditionally called ‘predicate’.) To make a statement that is either true or false we must then quantify statements, as in the (false) statement “for all  $x \in \mathbb{N}$ ,  $\text{Prime}(x)$  implies  $\text{PerfectSquare}(x)$ .” The modifier “first-order” means that the variables used by the Boolean functions are members of the domain of discourse (for  $\text{Prime}$  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 2016) 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](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 human computers at work.



Katherine Johnson, b 1918

Another example is that, as told in the film *Hidden Figures*, the trajectory for US astronaut John Glenn’s pioneering orbit of Earth was found by the human computer Katherine Johnson and her colleagues, African

American women whose accomplishments are all the more impressive because they occurred despite appalling discrimination.

*don't involve random methods* We can build things that return completely random results; one example is a device that registers consecutive clicks on a Geiger counter and if the second gap between clicks is longer than the first it returns 1, else it returns 0. See also <https://blog.cloudflare.com/randomness-101-lavarand-in-production/>.

*analog devices* See (A/V Geeks 2013) about slide rules, (Wikipedia contributors 2016c) about nomograms, (navyreviewer 2010) about a naval firing computer, and (Unknown 1948) about a more general-purpose machine. See also <https://www.youtube.com/watch?v=qqlJ50zDgeA> about the Antikythera mechanism.

*reading results off of a slide rule or an instrument dial* Suppose that an intermediate result of a calculation is 1.23. If we read it off the slide rule with the convention that the resolution accuracy is only one decimal place then we write down 1.2. Doubling that gives 2.4. But doubling the original number  $2 \cdot 1.23 = 2.46$  and then rounding to one place gives 2.5.

*no upper bound* This explication is derived from (Rogers 1987), p 1–5.

*more is provided* Perhaps the clerk has a helper, or the mechanism has a tender.

*A reader may object that this violates the goal of the definition, to model physically-realizable computations* We often describe computations that do not have a natural resource bound. The algorithm for long division 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)

*LEGO's* See for instance <https://www.youtube.com/watch?v=RLPVCJjTNgk&t=114s>.

*Finally, it trims off a 1* The instruction  $q_4 \rightarrow q_1$  won't ever be reached, but it does no harm. It is there for the definition of a Turing machine, to make  $\Delta$  defined on all  $q_p T_p$ . See also the note to that definition.

*transition function* The definition describes  $\Delta$  as a function  $\Delta: Q \times \Sigma \rightarrow (\Sigma \cup \{L, R\}) \times Q$ . That is a bit of 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  $\Delta: Q \times \Sigma \rightarrow (\Sigma \cup \{L, R\}) \times (Q \cup \hat{Q})$ .) We have left this point out of the main presentation since it doesn't seem to cause confusion and the discussion can be a distraction.)

*a complete description of a machine's action* It is reasonable to ask why our standard model, the Turing machine, is one that is so basic that programming it can be annoying. Why not choose a real world machine? The reason is that, as here, we can completely describe the actions of the Turing machine model, or of any of the other simple model that are sometimes used, in only a few paragraphs. A real machine would take a full book, and a full semester. We do Turing machines because they are simple to describe (they are also historically important, and the work in Chapter Five needs them.)

*q is a state, a member of Q* We are vague about what 'states' are but we assume that whatever they are, the set of

states  $Q$  is disjoint from the set  $\Sigma \cup \{L, R\}$ .

a snapshot, an instant in a computation So the configuration, along with the Turing machine, encapsulates the future history of the computation.

rather than, “this shows  $\phi$  taking a string representing 3 in unary to a string representing 5.” That is, we do this for the same reason that we would say, “This is me when I was ten.” instead of, “This is a picture of me when I was ten.”

a physical system evolves through a sequence of discrete steps that are local, meaning that all the action takes place within one cell of the head Adapted from (Widgerson 2017).

constructed the first machine See (Leupold 1725).

A number of mathematicians See also (Wikipedia contributors 2014).

Church suggested to Gödel (Soare 1999)

established beyond any doubt (Gödel 1995)

Church's Thesis 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 intuitively computable by a human computer; see for instance (B. J. Copeland and Proudfoot 1999). But the thesis as stated here, that what can be done by a Turing machine is what can be done by any physical mechanism that is discrete and deterministic, is certainly the thesis as it is taken in the field today. And besides, Church and Turing did not in fact distinguish between the two cases; (Hodges 2016) points to Church's review of Turing's paper in the *Journal of Symbolic Logic*: “The author [i.e. Turing] proposes as a criterion that an infinite sequence of digits 0 and 1 be ‘computable’ that it shall be possible to devise a computing machine, occupying a finite space and with working parts of finite size, which will write down the sequence to any desired number of terms if allowed to run for a sufficiently long time. As a matter of convenience, certain further restrictions are imposed on the character of the machine, but these are of such a nature as obviously to cause no loss of generality—in particular, a human calculator, provided with pencil and paper and explicit instructions, can be regarded as a kind of Turing machine.” This has Church referring to the human calculator not as the prototype but instead as a special case of the class of defined machines.

we cannot give a mathematical proof 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.” (SE user Ryan Williams 2010)

formalizes the notion of ‘effective’ or ‘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 a machine 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.)

*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 (SE 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, everything changed. A Becquerel discovered radiation. 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](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 (B. Jack Copeland 1996), (B. Jack Copeland 1999), and (B. Jack Copeland 2002) argue that computations can be done that are beyond the capabilities of Turing machines, while (Davis 2004), (Davis 2006), and (Gandy 1980) give arguments that most Theory of Computing researchers consider persuasive.

*Often when we want to show that something is computable by a Turing machine* The same point stated another way, from (SE author Andrej Bauer 2018): In books on computability theory it is common for the text to skip details on how a particular machine is to be constructed. The author of the computability book will mumble something about the Turing-Church thesis somewhere in the beginning. This is to be read as “you will have to do the missing parts yourself, or equip yourself with the same sense of inner feeling about computation as I did”. Often the author will give you hints on how to construct a machine, and call them “pseudo-code”, “effective procedure”, “idea”, or some such. The Church-Turing thesis is the social convention that such descriptions of machines suffice. (Of course, the social convention is not arbitrary but rather based on many years of experience on what is and is not computable.) . . . I am not saying that this is a bad idea, I am just telling you honestly what is going on. . . . So what are we supposed to do? We certainly do not want to write out detailed constructions of machines, because then students will end up thinking that’s what computability theory is about. It isn’t. Computability theory is about contemplating what machines we could construct if we wanted to, but we don’t. As usual, the best path to wisdom is to pass through a phase of confusion.

*Suppose that you have infinitely many dollars.* (Joel David Hamkins 2010)

*H Grassmann produced a more elegant definition* In 1888 Dedekind used this definition to give the first rigorous proof of the laws of elementary school arithmetic.

*it specifies the meaning, the semantics, of the operation* A Perl’s epigram, “Recursion is the root of computation since it trades description for time” expresses this idea. The recursive definition implicitly includes steps, 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 of there being something perplexing about recursion is often expressed with a story. The philosopher W James gave a public lecture on cosmology, and afterward 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, “If your theory is correct then what does this turtle stand on?” “You’re very clever, Mr James,” she replied, “but I have an answer. The first turtle stands on the back of a second, far larger, turtle.” James persisted, “And this second turtle, Madam?” Immediately she crowed, “It’s no use Mr James—it’s turtles all the way down!” See <https://xkcd.com/1416>. (Wikipedia contributors 2016e)

Another widely known reference is that with the invention of better and better microscopes, scientists studying fleas came to see that on them were even smaller 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 (William S. Renwick 1949).

*Towers of Hanoi* The puzzle was invented by E Lucas in 1883 but the next year H De Parville made of it quite a great problem with the delightful problem statement.

*hyperoperation* (Goodstein 1947)

$\mathcal{H}_3(4, 4)$  is much greater than the number of elementary particles in the universe The radius of the universe is about  $45 \times 10^9$  light years. That's about  $10^{62}$  Plank units. A system of much more than  $r^{1.5}$  particles packed in  $r$  Plank units will collapse rapidly. So the number of particles is less than  $10^{92}$ , which is about  $2^{305}$ , which is much less than  $\mathcal{H}_3(4, 4)$ . (Levin 2016)

*a programming language having only bounded loops computes all of the primitive recursive functions* (Meyer and Ritchie 1966)

*output only primes* In fact, there is no 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 this.

$$p(\hat{n} + k \cdot \hat{m}) \equiv p(\hat{n}) \pmod{\hat{m}}$$

Because  $p(\hat{n}) = \hat{m}$ , this gives that  $p(\hat{n} + k \cdot \hat{m}) = \hat{m}$  since that is the only prime number that is a multiple of  $\hat{m}$ , and  $p$  outputs only primes. But with that,  $p(n) = \hat{m}$  has infinitely many roots, and is therefore the constant polynomial.  $\square$

*looking for something that is not there* Goldbach's conjecture is that every even number can be written as the sum of at most two primes. Here are the first few instances:  $2 = 2$ ,  $4 = 2 + 2$ ,  $6 = 3 + 3$ ,  $8 = 5 + 3$ ,  $10 = 7 + 3$ . A natural attack is to do an unbounded computer search. As of this writing the conjecture has been tested up to  $10^{18}$ .

*Collatz conjecture* See (Wikipedia contributors 2019a).

*One of its design goals* Secondary goals are to output a picture of the configuration after each step, and to be easy to understand for a reader new to Racket.

$\sin(x)$  may be calculated via its Taylor polynomial The Taylor series is  $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$  We might do a practical calculation by deciding that a sufficiently good approximation is to terminate that series at the  $x^5$  term, giving a Taylor polynomial.

*C Shannon* See this profile of him: <http://www.newyorker.com/tech/elements/claudeshannon-thefather-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](https://en.wikipedia.org/wiki/A_Symbolic_Analysis_of_Relay_and_Switching_Circuits).

*kind of nor gate* This shows an N-type Metal Oxide Semiconductor Transistor. There are many other types.

*problem of humans on Mars* To get there the idea was to use a rocket ship impelled by dropping a sequence of atom bombs out the bottom; the energy would let the ship move rapidly around the solar system. This sounds like a crank plan but it is perfectly feasible (Brower 1983). Having been a key person in the development of the atomic bomb, von Neumann was keenly aware of their capabilities.

*J Conway* Conway was a magnetic person, and extraordinarily creative. See an excerpt from an excellent biography at <https://www.ias.edu/ideas/2015/roberts-john-horton-conway>.

*computer craze* (Bellos 2014)

*To start* A good way to experiment is the Free program Golly; see <http://golly.sourceforge.net/>.

*zero-player game* See <https://www.youtube.com/watch?v=R9Plq-D1gEk>.

*a rabbit* Discovered by A Trevorror in 1986.

*We will show here* This presentation is based on that of (Hennie 1977), (Smoryński 1991), and (Robinson 1948).

*giving a programming language that computes primitive recursive functions* See the history at (Brock 2020).

*LOOP program* (Meyer and Ritchie 1966)

## Background

*Deep Field movie* <https://www.youtube.com/watch?v=yDiD8F9ItXo>

*two paradoxes* These are 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 rate higher than others. The first reference speaks to the cardinal nature of numbers and the second is 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  $\mathbb{N}$*  Its existence is guaranteed by the Axiom of Infinity, one of the standard axioms of Mathematics, the Zermelo-Frankel axioms.

*due to Zeno* Zeno gave a number of related paradoxes of motion. See (Wikipedia contributors 2016f) (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 version of exploiting open-endedness is the 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).

*numbers 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 triangular number

$\text{cantor}(x, y) = x + [(x + y)(x + y + 1)/2]$  The Fueter-Pólya Theorem says that this is essentially the only quadratic function that serves as a pairing; see (Smoryński 1991). No one knows whether there are pairing functions that are any other kind of polynomial.

*memoization* The term was invented by Donald Michie (Wikipedia contributors 2016b), who among other accomplishments was a coworker of Turing's in the World War II effort to break the German secret codes.

*assume that we have a family of correspondences  $f_j : N \rightarrow S_j$*  To pass from the original collection of infinitely many onto functions  $f_i : \mathbb{N} \rightarrow S_i$  to a single, uniform, family of onto functions  $f_j(i) = f(j, y)$  we need some version of the Axiom of Choice, perhaps Countable Choice. We omit discussion of that because it would take us far afield.

*doesn't matter much* For more on "much" see (Rogers 1958).

*but that we won't make precise* One problem with this scheme is that it depends on the underlying computer. Imagine that your computer uses eight bit words. If we want the map from a natural number to a source code and the input number is 9 then in binary that's 1001, which is not eight bits and to disassemble it you need to pad the it out to the machine's word length, as 00001001. Another issue is the ambiguity caused by leading 0's, e.g. the bit string 00000000 00001001 also represent 9 but disassembles to a two-operation source. We could

address this by imagining that the operation with instruction code 00000000 is NOP and then disallow source code that starts with such an instruction (reasoning that starting a serial program with fewer NOP's won't change its input-output behavior), except for the source consisting of a single NOP. But we are getting into the weeds of computer architecture here, which is not where we want to be, so we take this numbering scheme only informally.

*adding the instruction  $q_{j+k} \text{BB} q_{j+k}$*

This is essentially what a compiler calls ‘unreachable code’ in that it is not a state that the machine will ever be in.

*central to the entire Theory of Computation* The classic text (Rogers 1987) says, “It is not inaccurate to say that our theory is, in large part, a ‘theory of diagonalization’.”

*This technique is diagonalization* The argument just sketched is often called Cantor's diagonal proof, although it was not Cantor's original argument for the result, and although the argument style is not due to Cantor but instead to Paul du Bois-Reymond. The fact that scientific results are often attributed to people who are not their inventor is *Stigler's law of eponymy*, because it wasn't invented by Stigler (who attributes it to Merton). In mathematics this is called *Boyer's Law*, who didn't invent it either. (Wikipedia contributors 2015).

*Musical Chairs* It starts with more children than chairs. Some music plays and the children walk around the chairs. But the music stops suddenly and each child tries to sit, leaving someone without a chair. That child has to leave the game, a chair is removed, and the game proceeds.

*so many real numbers* This is a Pigeonhole Principle argument.

*Your study partner is confused about the diagonal argument* From (SE author Kaktus and various others 2019).

*ENIAC, reconfigure by rewiring.* Jean Jennings (left), Marlyn Wescoff (center), and Ruth Licherman program the ENIAC, circa 1946. U. S. Army Photo

*A pattern in technology is for jobs done in hardware to migrate to software* One story that illustrates the naturalness of this involves the English mathematician C Babbage, and his protogee A Lovelace. In 1812 Babbage was developing tables of logarithms. These were calculated by computers—the word then current for the people who computed them by hand. To check the accuracy he had two people do the same table and compared. He was annoyed at the number of discrepancies and had the idea to build a machine to do the computing. He got a government grant to design and construct a machine called the difference engine, which he started in 1822. This was a single-purpose device, what we today would call a calculator. One person who became interested in the computations was an acquaintance of his, Lovelace (who at the time was named Byron, as she was the daughter of the poet Lord Byron).



Charles Babbage, 1791–1871



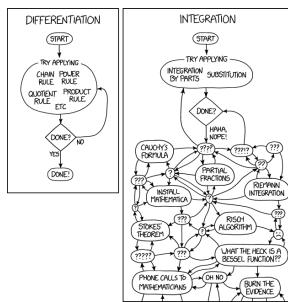
Ada Lovelace (nee Byron), 1815–1852

However, this machine was never finished because Babbage had the thought to make a device that would be programmable, and that was too much of a temptation. Lovelace contributed an extensive set of notes on a proposed new machine, the analytical engine, and has become known as the first programmer.

*controlled by cards* It weaves with hooks whose positions, raised or lowered, are determined by holes punched in the cards

*have the same output behavior* A technical point: Turing machines have a tape alphabet. So a universal machine's input or output can only involve symbols that it is defined as able to use. If another machine has a different tape alphabet then how can the universal machine simulate it? As usual, we define things so that the universal machine manipulates representations of the other machine's alphabet. This is similar to the way that an everyday computer represents decimals using binary.

*flow chart* 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](http://archive.computerhistory.org/resources/text/Remington_Rand/Univac.Flowmatic.1957.102646140.pdf).

*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](https://en.wikipedia.org/wiki/Feynman_point). Most experts guess that for any  $n$  the decimal expansion contains a sequence of  $n$  consecutive nines but no one has proved or disproved that.

*there is a difference between showing that this function is computable* This is a little like Schrödinger's cat paradox (see [https://en.wikipedia.org/wiki/Schr%C3%B6dinger%27s\\_cat](https://en.wikipedia.org/wiki/Schr%C3%B6dinger%27s_cat)) in that it seems that one of the two is

right but we just don't know which.

*"something is computable if you can write a program for it"* is naive From (SE 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  $\pi$*  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  $\pi$  is not rational, so  $\pi$  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.")

*it must be effective* In fact, careful analysis shows that it is primitive recursive.

*In  $f$ 's top case the output value 42 doesn't matter* Sometimes we use 42 when we need an arbitrary output value, because of its connection with *The Hitchhiker's Guide to the Galaxy*, (Adams 1979). See also (Wikipedia contributors 2020a).

*undecidable* The word 'undecidable' is used in mathematics in two different ways. The definition here of course applies to the Theory of Computation. In relation to Gödel's theorems, it means that a statement is cannot be proved true or proved false within a given formal system.

*halt on some inputs but not on others* A Turing machine could fail to halt because it has an infinite loop. The Turing machine  $P_0 = \{ q_0Bq_0, q_011q_0 \}$  never halts, cycling forever in state  $q_0$ . We could patch this problem; we could write a program `inf_loop_decider` that at each step checks whether a machine has ever before in this computation had the same configuration as it has now. This program will detect infinite loops like the prior one.

However, note that there are machines that fail to halt but do not have loops, in that they never repeat a configuration. One is  $P_1 = \{ q_0B1q_1, q_11Rq_0 \}$  which when started on a blank tape will endlessly move to the right, writing 1's.

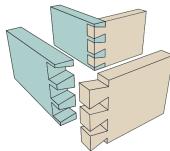
*496 and 8128* The divisors of 496 are 1, 2, 4, 8, 16, 31, 62, 124, 248, and 496. The divisors of 8128 are 1, 2, 4, 8, 16, 32, 64, 127, 254, 508, 1016, 2032, 4064, and 8128.

*understand the form of all even perfect numbers* A number is an even perfect number if and only if it has the form  $(2^p - 1) \cdot 2^{p-1}$  where  $2^p - 1$  is prime.

*involving an unbounded search* 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 at the moment we are studying what can be done in principle.

*functions that solve it* (Wikipedia contributers 2017h)

*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.



*won't be a physically-realizable discrete and deterministic mechanism* 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." (Turing 1938)

*magic smoke* See (Wikipedia contributers 2017f).

*we will instead describe it conceptually* For a full treatment see (Rogers 1987).

*the notion of partial computable function seems to have an in-built defense against diagonalization* (Odifreddi 1992), p 152.

*this machine's name is its behavior* Nominative determinism is the theory that a person's name has some influence over what they do with their life. Examples are: the sprinter Usain Bolt, the US weatherman Storm Fields, the baseball player Prince Fielder, and the Lord Chief Justice of England and Wales named Igor Judge, I Judge. See [https://en.wikipedia.org/wiki/Nominative\\_determinism](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 is that the assertion that they are equal refers to the number that they represent, not to the representation itself. That is, in mention '1/2' and '3/6' are different strings but in use, they point to the same number.

*mathematical fable* This mathematical fable came from David Hilbert in 1924. It was popularized by George Gamow in *One, Two, Three ... Infinity*. (Kragh 2014).

*Napoleon's downfall in the early 1800's* See (Wikipedia contributers 2017d).

*period of prosperity and peace* See (Wikipedia contributers 2017i).

*A A Michelson, who wrote in 1899, "The more important fundamental laws and facts of physical science have all been discovered, and these are now so firmly established that the possibility of their ever being supplanted in consequence of new discoveries is exceedingly remote."* Michelson was a major figure, whose opinions carried weight. From 1901 to 1903 he was president of the American Physical Society. In 1910–1911 he was president of the American

Association for the Advancement of Science and from 1923–1927 he was president of the National Academy of Sciences. In 1907 he received the Copley Medal from the Royal Society in London, and the Nobel Prize. He remains well known today for the Michelson–Morley experiment that tried to detect the presence of aether, the hypothesized medium through which light waves travel.

*working out the rules of a game by watching it being played* 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 2017)

*the discovery of radiation* This happened in 1896, before Michelson’s statement. Often the significance of things takes time to be apparent

*Einstein became an overnight celebrity* “Einstein Theory Triumphs” was the headline in *The New York Times*. JJ Thomson, president of the Royal Society, referred to the experiment’s success as “one of the momentous, if not the most momentous, pronouncements in the history of human thought.” And, when Einstein arrived in New York by boat in 1921, reporters were delighted to find not a stuffy academic but instead someone who was very endearing, quotable, and photogenic. The hair, the scruffy clothes, and the violin, all made him seem the definition of a genius, which of course continues today.

*“everything is relative.”* Of course, the history around Einstein’s work is vastly more complex and subtle. But we are speaking of the broad understanding, not of the truth.

*loss of certainty* This phrase is the title of a famous popular book on mathematics, by M Klein. The book is fun and a thought-provoking read. Also thought-provoking are some criticisms of the book. (Wikipedia contributors 2019b) is good introduction to both.

*the development of a fetus is that it basically just expands* The issue was whether the fetus began preformed or as a homogeneous mass, see (Maienschein 2017). Today we have similar questions about the Big Bang—we are puzzled to explain how a mathematical point, which is without internal structure and entirely homogeneous, could develop into the very non-homogeneous universe that we see today.

*infinite regress* This line of thinking often depends on the suggestion that all organisms were created at the same time, that they have existed since the beginning of the posited creation.

*discovery by Darwin and Wallace of descent with modification through natural selection* Darwin wrote in his autobiography, “The old argument of design in nature, as given by Paley, which formerly seemed to me so conclusive, fails, now that the law of natural selection has been discovered. We can no longer argue that, for instance, the beautiful hinge of a bivalve shell must have been made by an intelligent being, like the hinge of a door by man. There seems to be no more design in the variability of organic beings and in the action of natural selection, than in the course which the wind blows. Everything in nature is the result of fixed laws.”

*the car is in some way less complex than the robot* This is an information theoretic analog of the Second Law of Thermodynamics. E Musk has expressed something of the same sentiment, “The extreme difficulty of scaling production of new technology is not well understood. It’s 1000% to 10,000% harder than making a few prototypes. The machine that makes the machine is vastly harder than the machine itself.” See <https://twitter.com/elonmusk/status/1308284091142266881>.

*self-reference* ‘Self-reference’ describes something that refers to itself. The classic example is the Liar paradox, the statement attributed to the Cretian Epimenides, “All Cretans are liars.” Because he is Cretian we take the statement to be an utterance about utterances by him, that is, to be about itself. If we suppose that the statement is true then it asserts that anything he says is false, so the statement is false. But if we suppose that it is false then we take that he is saying the truth, that all his statements are false. Its a paradox, meaning that the reasoning seems locally sound but it leads to a global impossibility.

This is related to Russell’s paradox, which lies at the heart of the diagonalization technique, that if we define the collection of sets  $R = \{S \mid S \notin S\}$  then  $R \in R$  holds if and only if  $R \notin R$  holds.

Self-reference is obviously related to recurrence. You see it sometimes pictured as an infinite recurrence, as here on the front of a chocolate product.



Because of this product, having a picture contain itself is sometimes known as the Droste effect.

Besides the Liar paradox there are many others. One is Quine’s paradox, a sentence that asserts its own falsehood.

“Yields falsehood when preceded by its quotation”  
yields falsehood when preceded by its quotation.

If this sentence were false then it would be saying something that is true. If this sentence were true then what it says would hold and it would be not true.

A wonderful popular book exploring these topics and many others is (Hofstadter 1979).

*quine* Named for the philosopher Willard Van Orman Quine.

*for routines to have access to their code* Introspection is the ability to inspect code in the system, such as to inspect the type of objects. Reflection is the ability to make modifications at runtime.

*We will show how a routine can know its source* This is derived from the wonderful presentation in (Sipser 2013).

*The verb ‘to quine’* Invented by D Hofstadter .

*which n-state Turing Machine leaves the most 1’s after halting* R H Bruck famously wrote (R H Bruck n.d.), “I might compare the high-speed computing machine to a remarkably large and awkward pencil which takes a long time to sharpen and cannot be held in the fingers in the usual manner so that it gives the illusion of responding to my

thoughts, but is fitted with a rather delicate engine and will write like a mad thing provided I am willing to let it dictate pretty much the subjects on which it writes.” The Busy Beaver machine is the maddest writer possible.

*Radó noted in his 1962 paper* This paper (Radó 1962) is exceptionally clear and interesting.

$\Sigma(n)$  is unknowable See (Aaronson 2012a). See also <https://www.quantamagazine.org/the-busy-beaver-game-illuminates-the-fundamental-limits-of-math-20201210/>.

a 7918-state Turing machine The number of states needed has since been reduced. As of this writing it is 1919.

*the standard axioms for Mathematics* This is ZFC, the Zermelo–Fraenkel axioms with the Axiom of Choice. (In addition, they also took the hypothesis of the Stationary Ramsey Property.)

*take the floor* Let the  $n$ -th triangle number be  $t(n) = 0 + 1 + \dots + n = n(n + 1)/2$ . The function  $t$  is monotonically increasing and there are infinitely many triangle numbers. Thus for every natural number  $c$  there is a unique triangle number  $t(n)$  that is maximal so that  $c = t(n) + k$  for some  $k \in \mathbb{N}$ . Because  $t(n + 1) = t(n) + n + 1$ , we see that  $k < n + 1$ , that is,  $k \leq n$ . Thus, to compute the diagonal number  $d$  from the Cantor number  $c$  of a pair, we have  $(1/2)d(d + 1) \leq c < (1/2)(d + 1)(d + 2)$ . Applying the quadratic formula to the left half and right halves gives  $(1/2)(-3 + \sqrt{1 + 8c}) < d \leq (1/2)(-1 + \sqrt{1 + 8c})$ . Taking  $(1/2)(-1 + \sqrt{1 + 8c})$  to be  $\alpha$  gives that  $c \in (\alpha - 1 .. \alpha]$  so that  $d = \lfloor \alpha \rfloor$ . (Scott 2020)

*let's extend to tuples of any size* See [https://en.wikipedia.org/wiki/You\\_aren%27t\\_gonna\\_need\\_it](https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it).

## Languages

*having elephants move to the left side of a road or to the right* Less fancifully, we could be making a Turing machine out of LEGO’s and want to keep track by sliding a block from one side of a column to the other. Or, we could use an abacus.

*we could translate any such procedure* While a person may quite sensibly worry that elephants could be not just on the left side or the right, but in any of the continuum of points in between, we will make this assertion without more philosophical analysis than by just referring to the discrete nature of our mechanisms (as Turing basically did). That is, we take it as an axiom.

*finite set { 1000001, 1100001 }* Although it looks like two strings plucked from the air, the language is not without sense. The bitstring 1000001 represents capital A in the ASCII encoding, while 1100001 is lower case a. The American Standard Code for Information Interchange, ASCII, is a widely used, albeit quite old, way of encoding character information in computers. The most common modern character encoding is UTF-8, which extends ASCII. For the history see <https://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt>.

*palindrome* Sometimes people call Psychology the study of college freshmen because so many studies start, roughly, “we put a bunch of college freshmen in a room, lied to them about what we were doing, and . . . ” In the same way, Theory of Computing sometimes seems like the study of palindromes.

*words from English that are palindromes* Some people like to move beyond single word palindromes to make sentence-length palindromes that make some sense. Some of the more famous are: (1) supposedly the first sentence ever uttered, “Madam, I’m Adam” (2) Napoleon’s lament, “Able was I ere I saw Elba” and (3) “A man, a plan, a canal: Panama”, about Theodore Roosevelt.

*In practice a language is usually given by rules* Linguists started formalizing the description of language, including phrase structure, at the start of the 1900's. Meanwhile, string rewriting rules as formal, abstract systems were introduced and studied by mathematicians including Axel Thue in 1914, Emil Post from the 1920's through the 1940's and Turing in 1936. Noam Chomsky, while teaching linguistics to students of information theory at MIT, combined linguistics and mathematics by taking Thue's formalism as the basis for the description of the syntax of natural language. (Wikipedia contributors 2017e)

*"the red big barn" sounds wrong.* Experts vary on the exact rules but one source gives (article) + number + judgment/attitude + size, length, height + age + color + origin + material + purpose + (noun), so that "big red barn" is size + color + noun, as is "little green men." This is called the Royal Order of Adjectives; see <http://english.stackexchange.com/a/1159>. A person may object by citing "big bad wolf" but it turns out there is another, stronger, rule that if there are three words then they have to go I-A-O and if there are two words then the order has to be I followed by either A or O. Thus we have tick tock but not tock tick. Similarly for tic-tac-toe, mishmash, King Kong, or dilly dally.

*very strict rules* Everyone who has programmed has had a compiler chide them about a syntax violation.

*grammars are the language of languages.* From Matt Swift, <http://matt.might.net/articles/grammars-bnf-ebnf/>.

*this grammar* Taken from [https://en.wikipedia.org/wiki/Formal\\_grammar](https://en.wikipedia.org/wiki/Formal_grammar).

*dangling else* See [https://en.wikipedia.org/wiki/Dangling\\_else](https://en.wikipedia.org/wiki/Dangling_else).

*postal addresses.* Adapted from [https://en.wikipedia.org/wiki/BackusNaur\\_Form](https://en.wikipedia.org/wiki/BackusNaur_Form).

*Recall Turing's prototype computer* In this book we stick to grammars where each rule head is a single nonterminal. That greatly restricts the languages that we can compute. More general grammars can compute more, including every set that can be decided by a Turing machine.

*often state their problems* For instance, see the blogfeed for Theoretical Computer Science <http://cstheory-feed.org/> (Various authors 2017)

*represent a graph in a computer* 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.

*a standard way to express grammars* One factor influencing its adoption was a letter that D Knuth wrote to the *Communications of the ACM* (D. E. Knuth 1964). He listed some advantages over the grammar-specification methods that were then widely used. Most importantly, he contrasted BNF's '<addition operator>' with 'A', saying that the difference is a great addition to "the explanatory power of a syntax." He also proposed the name Backus Naur Form.

*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 ten years from now. 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, (2) years are four digits to prevent the Y2K problem (Encyclopædia Britannica 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

*an idealized one* Turing machines are ‘idealized’ in that, for instance, the tape length is not bounded by the number of elementary particles in the universe.

*what can be done by a machine with a bounded number of possible configurations* From Rabin, Scott, Finite Automata and Their Decision Problems, 1959: *Turing machines are widely considered to be the abstract prototype of digital computers; workers in the field, however, have felt more and more that the notion of a Turing machine is too general to serve as an accurate model of actual computers. It is well known that even for simple calculations it is impossible to give an a priori upper bound on the amount of tape a Turing machine will need for any given computation. It is precisely this feature that renders Turing's concept unrealistic. In the last few years the idea of a finite automaton has appeared in the literature. These are machines having only a finite number of internal states that can be used for memory and computation. The restriction on finiteness appears to give a better approximation to the idea of a physical machine. Of course, such machines cannot do as much as Turing machines, but the advantage of being able to compute an arbitrary general recursive function is questionable, since very few of these functions come up in practical applications.*

*transition function*  $\Delta: Q \times \Sigma \rightarrow Q$  Some authors allow the transition function to be partial. That is, some authors allow that for some state-symbol pairs there is no next state. This choice by an author is a matter of convenience, as for any such machine you can create an error state  $q_{\text{error}}$  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 2017k).

*how phone numbers were originally handled 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. This was abandoned by the early 1950s. (Wikipedia contributors 2017g).

*switching with physical devices* The devices to do the switching were invented in 1889 by an undertaker whose competitor's wife was the local telephone operator and routed calls to her husband's business. (Wikipedia contributors 2017b)

*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.

*Traveling Salesman problem* See <https://nbviewer.jupyter.org/url/norvig.com/ipython/TSP.ipynb>.

*US lower forty eight* See <https://wiki.openstreetmap.org/wiki/TIGER>.

*simultaneously in many different states* :-)

*no-state* A person can wonder about no-state. Where is it, exactly? We can think that it is like what happens if you write a program with a sequence of if-then statements and forget to include an else. Obviously a computer goes somewhere, the instruction pointer points to some next address, but what happens is not sensible in terms of the model you've written.

Alternatively, the wonderful book (Hofstadter 1979) describes a place named Tumbolia, which is where holes go when they are filled (also where your lap goes when you stand). Perhaps the machines go there.

*amb*( $\sigma$ ) The name *amb* abbreviates 'ambiguous function'. Here is a small example. Essentially *Amb*( $x, y, z$ ) splits the computation into three possible futures: a future in which the value  $x$  is yielded, a future in which the value  $y$  is yielded, and a future in which the value  $z$  is yielded. The future which leads to a successful subsequent computation is chosen. The other "parallel universes" somehow go away. (*Amb* called with no arguments fails.) The output is 2 4 because *Amb*(1, 2, 3) correctly chooses the future in which  $x$  has value 2, *Amb*(7, 6, 4, 5) chooses 4, and consequently *Amb*( $x * y = 8$ ) produces a success.

These were described by John McCarthy in (McCarthy 1963). "Ambiguous functions are not really functions. For each prescription of values to the arguments the ambiguous function has a collection of possible values. An example of an ambiguous function is *less*( $n$ ) defined for all positive integer values of  $n$ . Every non-negative integer less than  $n$  is a possible value of *less*( $n$ ). First we define a basic ambiguity operator *amb*( $x, y$ ) whose possible values are  $x$  and  $y$  when both are defined: otherwise, whichever is defined. Now we can define *less*( $n$ ) by *less*( $n$ ) = *amb*( $n - 1$ , *less*( $n - 1$ ))."

*demonic* The term 'demon' is associated with 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.” (Computing 2017)

*mathematical model of neurons* (Wikipedia contributors 2017c)

*have a vowel in the middle* Most speakers of American English cite the vowels as ‘a’, ‘e’, ‘i’, ‘o’, and ‘u’. See (Bigham 2014).

*before and after pictures* This diagram is derived from (Hopcroft, Motwani, and Ullman 2001).

*The fact that we can describe these languages in so many different ways* (SE author David Richerby 2018).

*just list all the cases* In practice the suggestion in the first paragraph to list all 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 regular language. But constructing a Finite State machine for it is silly. In addition, a finite regular 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  $4 = 2 + 2$ ,  $6 = 3 + 3$ ,  $8 = 3 + 5$ , ... (see [https://en.wikipedia.org/wiki/Goldbach%27s\\_conjecture](https://en.wikipedia.org/wiki/Goldbach%27s_conjecture)). Computer testing shows that this pattern continues to hold up to very large numbers but no one knows if it is true for all evens. Now consider the set consisting of the string  $\sigma \in \{0, \dots, 9\}^*$  representing 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.

*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 ≤ k < 2n* This gives an algorithm that inputs a Finite State machine and determines, in a finite time, if it recognizes an infinite language.

*be aware that another algorithm* See (Knuutila 2001).

*For the third* This is derived from the presentation in (Hopcroft, Motwani, and Ullman 2001).

*Most modern programming languages are context free* This is a good approximation but 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 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. (SE author rici 2021)

\d We shall ignore cases of non-ASCII digits, that is, cases outside 0–9.

*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 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 significant parts of an HTML document. See (bobnicae 2009).

*regex golf* See <https://alf.nu/RegexGolf>, and <https://nbviewer.jupyter.org/url/norvig.com/ipython/xkcd1313.ipynb>.

## Complexity

*mirrors the subject’s history* This is like the slogan “ontogeny recapitulates phylogeny” for the now-discredited Biological theory that the development of an embryo, which is called ontogeny, goes through same stages as the adult stages in the evolution of the animal’s ancestors, which is phylogeny.

*A natural next step is to look to do jobs efficiently* S Aaronson states it more provocatively as, “[A]s computers became widely available starting in the 1960s, computer scientists increasingly came to see computability theory as not asking quite the right questions. For, almost all the problems we actually want to solve turn out to be computable in Turing’s sense; the real question is which problems are *efficiently* or *feasibly* computable.” (Aaronson 2011)

*A Karatsuba* See [https://en.wikipedia.org/wiki/Anatoly\\_Karatsuba](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,  $28 \cdot 476 = 21 \cdot 2^{10} + 216 \cdot 2^5 + 60$ ). 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.

The ' $f = \mathcal{O}(g)$ ' notation is very common, but awkward. See also [https://whystartat.xyz/wiki/Big\\_0\\_notation](https://whystartat.xyz/wiki/Big_0_notation). appear most often 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.

table below shows why. This table is adapted from (Garey and Johnson 1979).

there are  $3.16 \times 10^7$  seconds in a year. The easy way to remember this is the bumper sticker slogan by Tom Duff from Bell Labs: “ $\pi$  seconds is a nanocentury.”

very, very much larger than polynomial growth. According to an old tale from India, the Grand Vizier Sissa Ben Dahir invented chess and so was granted a wish by the delighted Indian King, Shirham. Sissa said, “Majesty, give me a grain of wheat to place on the first square of the board, and two grains of wheat to place on the second square, and four grains of wheat to place on the third, and eight grains of wheat to place on the fourth, and so on. Oh, King, let me cover each of the 64 squares of the board.”

“And is that all you wish, Sissa, you fool?” exclaimed the astonished King.

“Oh, Sire,” Sissa replied, “I have asked for more wheat than you have in your entire kingdom. Nay, for more wheat than there is in the whole world, truly, for enough to cover the whole surface of the earth to the depth of the twentieth part of a cubit.”

Sissa has the right idea but his arithmetic is slightly off. A cubit is the length of a forearm, from the tip of the middle finger to the bottom of the elbow, so perhaps twenty inches. The geometric series formula gives  $1 + 2 + 4 + \dots + 2^{63} = 2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615 \approx 1.84 \times 10^{19}$  grains of rice. The surface area of the earth, including oceans, is 510 072 000 square kilometers. There are  $10^{10}$  square centimeters in each square kilometer so the surface of the earth is  $5.10 \times 10^{18}$  square centimeters. That’s between three and four grains of rice on every square centimeter of the earth. Not rice an inch thick, but still a lot.

Another way to get a sense of the amount of rice is: there are about 7.5 billion people on earth so it is on the order of  $10^8$  grains of rice for each person in the world. There are about  $1\,000\,000 = 10^7$  grains of rice in a bushel. In sum, ten bushels for each person.

Cobham’s thesis. Credit for this goes to both A Cobham and J Edmonds, separately; see (Cobham 1965) and (Edmonds 1965).



Jack Edmonds, b 1934



Alan Cobham, 1927–2011

Cobham’s paper starts by asking whether “is it harder to multiply than to add?” a question that we still cannot answer. Clearly we can add two  $n$ -bit numbers in  $\mathcal{O}(n)$  time, but we don’t know whether we can multiply in linear time.

Cobham then goes on to point out the distinction between the complexity of a problem and the running time of a particular algorithm to solve that problem, and notes that many familiar functions, such as addition, multiplication, division, and square roots, can all be computed in time “bounded by a polynomial in the lengths of the numbers involved.” He suggests we consider the class of all functions having this property.

As for Edmunds, in a “Digression” he writes: “An explanation is due on the use of the words ‘efficient algorithm.’ According to the dictionary, ‘efficient’ means ‘adequate in operation or performance.’ This is roughly the meaning I want—in the sense that it is conceivable for [this problem] to have no efficient algorithm. . . . There is an obvious finite algorithm, but that algorithm increases in difficulty exponentially with the size of the graph. It is by no means obvious whether or not there exists an algorithm whose difficulty increases only algebraically with the size of the graph . . . If only to motivate the search for good, practical algorithms, it is important to realize that it is mathematically sensible even to question their existence.”

*tractable* Another word that you can see in this context is ‘feasible’. Some authors use them to mean the same thing, roughly that we can solve reasonably-sized problem instances using reasonable resources. But some authors use ‘feasible’ to have a different connotation, for instance explicitly disallowing inputs are too large, such as having too many bits to fit in the physical universe. The word ‘tractable’ is more standard and works better with the definition that includes the limit as the input size goes to infinity, so here we stick with it.

*slower than the right by four assignments* Assuming that the compiler does not optimize it out of the loop.

*The definition of Big O ignores constant factors* This discussion originated as (SE author babou and various others 2015).

*the order of magnitude of these constants* For a rough idea of that these may be, here are some numbers that every programmer should know.

Operation	Cost in nanoseconds
Cache reference	0.5–7
Branch mispredict	5
Main memory reference	100
Send 1K bytes over 1 Gbps network	10 000
Read 1 MB sequentially from disk	20 000 000
Send packet CA to Netherlands to CA	150 000 000

A nanosecond is  $10^{-9}$  seconds. For more, see <https://www.youtube.com/watch?v=JEpsKnWZrJ8&app=desktop>.

*update that standard* Even Knuth had to update standards, from his machine model MIX to MMIX.

*an important part of the field’s culture* That is, these are storied problems.

*inventor of the quaternion number system* See [https://en.wikipedia.org/wiki/History\\_of\\_quaternions](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 appeared in the same journal and issue as Gödel's Incompleteness Theorem. The two were close friends.

*Königsberg* This happens to be the hometown of David Hilbert.

*no circuit is possible* For each 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.

*the countries must be contiguous* A notable example of a non-contiguous country in the world today is that Russia is separated from Kaliningrad, the city that used to be known as Königsberg.

*we can draw it in the plane* This is because the graph comes from a planar map.

*start with a planar graph* The graph is undirected and without loops.

*Counties of England and the derived planar graph* This is today's map. At the time, some counties were not contiguous.

*it was controversial*

See [https://www.maa.org/sites/default/files/pdf/upload\\_library/22/Ford/Swart697-707.pdf](https://www.maa.org/sites/default/files/pdf/upload_library/22/Ford/Swart697-707.pdf).

*An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*  
A brief biography is at <https://www.3quarksdaily.com/3quarksdaily/2018/02/george-boole-and-the-calculus-of-thought-5.html>.

*Conjunctive Normal form* Any Boolean function can be expressed in that form.

*Below are the numbers for the 2020 election* Here are the state abbreviations.

<i>State</i>	<i>Abbr.</i>	<i>State</i>	<i>Abbr.</i>	<i>State</i>	<i>Abbr.</i>
Alabama	AL	Kentucky	KY	North Dakota	ND
Alaska	AK	Louisiana	LA	Ohio	OH
Arizona	AZ	Maine	ME	Oklahoma	OK
Arkansas	AR	Maryland	MD	Oregon	OR
California	CA	Massachusetts	MA	Pennsylvania	PA
Colorado	CO	Michigan	MI	Rhode Island	RI
Connecticut	CT	Minnesota	MN	South Carolina	SC
Delaware	DE	Mississippi	MS	South Dakota	SD
District of Columbia	DC	Missouri	MO	Oklahoma	OK
Florida	FL	Montana	MT	Tennessee	TN
Georgia	GA	Nebraska	NE	Texas	TX
Hawaii	HI	Nevada	NV	Utah	UT
Idaho	ID	New Hampshire	NH	Vermont	VT
Illinois	IL	New Jersey	NJ	Virginia	VA
Indiana	IN	New Mexico	NM	Washington	WA
Iowa	IA	New York	NY	Wisconsin	WI
Kansas	KS	North Carolina	NC	Wyoming	WY

*ignore some fine points* For example, 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.”

*popularized as a toy* It was invented by Noyes Palmer Chapman, a postmaster in Canastota, New York. As early as 1874 he showed friends a precursor puzzle. By December 1879 copies of the improved puzzle were circulating in the northeast and students in the American School for the Deaf and other started manufacturing it. They became popular as the “Gem Puzzle.” Noyes Chapman had applied for a patent in February, 1880. By that time the game had become a craze in the US, somewhat like Rubik’s Cube a century later. It was also popular in Canada and Europe. See (Wikipedia contributors 2017a)

*We know of no efficient algorithm to find divisors* An effort in 2009 to factor a 768-bit number (232-digits) used

hundreds of machines and took two years. The researchers estimated that a 1024-bit number would take about a thousand times as long.

*Factoring seems to be hard* Finding factors has for many years been thought hard. For instance, a number is called a Mersenne prime if it is a prime number of the form  $2^n - 1$ . They are named after M Mersenne, a French friar and important figure in the early sharing of scientific results, who studied them in the early 1600's. He observed that if  $n$  is prime then  $2^n - 1$  may be prime, for instance with  $n = 3, n = 7, n = 31$ , and  $n = 127$ . He suspected that others of that form were also prime, in particular  $n = 67$ .

On 1903-Oct-31 F N Cole, then Secretary of the American Mathematical Society, made a presentation at a math meeting. When introduced, he went to the chalkboard and in complete silence computed  $2^{67} - 1 = 147\,573\,952\,589\,676\,412\,927$ . He then moved to the other side of the board, wrote 193 707 721 times 761 838 257 287, and worked through the calculation, finally finding equality. When he was done Cole returned to his seat, having not uttered a word in the hour-long presentation. His audience gave him a standing ovation.

Cole later said that finding the factors had been a significant effort, taking "three years of Sundays."

*Platonic solids* See (Wikipedia contributors 2017j).

*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).

*default interpretation of 'problem'* Not every computational problem is naturally expressable as a language decision problem Consider the task of sorting the characters of strings into ascending order. We could try to express it as the language of sorted strings  $\{\sigma \in \Sigma^* \mid \sigma \text{ is sorted}\}$ . But this does not require that we find a good way to sort an unsorted input. Another thought is to consider the language of pairs  $\langle\sigma, p\rangle$  where  $p$  is a permutation of the numbers  $0, \dots, |\sigma| - 1$  that brings the string into ascending order. But here also the formulation seems to not capture the sorting problem, in that recognizing a correct permutation feels different than generating one from scratch.

*Both of these show the collection of languages* One misleading aspect of this picture is that there are uncountably many languages but only countably many Turing machines, and hence only countably many computable or computably enumerable languages. So, shown to scale, the computably enumerable area of the blob would be an infinitesimally small speck at the very bottom. But such a picture would not show the features we want to illustrate, so these drawings take a graphical license.

*the shaded collection Rec consists of the Turing computable languages* The name Rec is because these used to be known as the 'recursive' languages.

*input two numbers and output their midpoint* See <https://hal.archives-ouvertes.fr/file/index/docid/576641/filename/computing-midpoint.pdf>.

*final two bits are 00* Decimal representation is not much harder since a decimal number is divisible by four if and only if the final two digits are in the set { 00, 04, ... 96 }.

*everything of interest can be represented with reasonable efficiency by bitstrings* See <https://rjlipton.wordpress.com/2010/11/07/what-is-a-complexity-class/>. Of course, a wag may say that if it cannot be represented by bitstrings then it isn't of interest. But we mean something less tautological: we mean that if we could want to compute with it then it can be put in bitstrings. For example, we find that we can process speech, adjust colors on an image, or regulate pressure in a rocket fuel tank, all in bitstrings, despite what may at first encounter seem to be the inherently analog nature of these things.

*Beethoven's 9th Symphony* The official story is that CD's are 72 minutes long so that they can hold this piece. *researchers often do not mention representations* This is like a programmer saying, “My program inputs a number” rather than, “My program inputs the binary representation of a number.” It is also like a person saying, “That’s me on the card” rather than “On that card is a picture of me.”

*leaving implementation details to a programmer* (Grossman 2010)

*complexity class* There are various definitions, which are related but not equivalent. Some authors fold in the requirement that that a class be associated with some resource specification. This has some implications because if an author, for instance, requires that each class be problems that are somehow solvable by Turing machines then each class is countable. Our definition is more general and does not imply that a class is countable.

*the time or space behavior* We will concentrate our attention resource bounds in the range from logarithmic and exponential, because these are the most useful for understanding problems that arise in practice.

*less than centuries* See the video from Google at <https://www.youtube.com/watch?v=-ZNEzzDcllU> and S Aaronson’s Quantum Supremacy FAQ at <https://www.scottaaronson.com/blog/?p=4317>.

*the claim is the subject of some scholarly dispute* 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 close attention* This discussion gained much from the material in (Allender, Loui, and Regan 1997). This includes several direct quotations.

*RE* Recall that ‘recursively enumerable’ is an older term for ‘computably enumerable’.

*adds some wrinkles* But it avoids a wrinkle that we needed for Finite State machines,  $\varepsilon$  transitions, since Turing machines are not required to consume their input one character at a time.

*function computed by a nondeterministic machine* One thing that we can do is to define that the nondeterministic machine computes  $f: \mathbb{B}^* \rightarrow \mathbb{B}^*$  is that if on an input  $\sigma$ , all branches halt and they all leave the same value on the tape, which we call  $f(\sigma)$ . Otherwise, the value is undefined,  $f(\sigma) \uparrow$ .

*might be much faster* R Hamming gives this example to demonstrate that an order of magnitude change in speed can change the world, can change what can be done: we walk at 4 mph, a car goes at 40 mph, and an airplane

goes at 400 mph. This relates to the bug picture that opens this chapter.

*we don't find the  $\omega$ 's, we just use them* This is like the Mechanical Turk [https://en.wikipedia.org/wiki/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 2017l)—there is a strategy for one of the two players that forces a win or a draw, no matter how the opponent plays

*a deterministic verifier must take exponential time* In fact, in the terminology of a later section, chess is known to be EXP complete. See (Fraenkel and Lichtenstein 1981).

*in a sense, useless* Being given an answer with no accompanying justification is a problem. This is like the Feynman algorithm for doing Physics: “The student asks . . . what are Feynman’s methods? [M] Gell-Mann leans coyly against the blackboard and says: Dick’s method is this. You write down the problem. You think very hard. (He shuts his eyes and presses his knuckles parodically to his forehead.) Then you write down the answer.” (Gleick 1992) It is also like the mathematician S Ramanujan, who relayed that the advanced formulas that he produced came in dreams from the god Narasimha. Some of these formulas were startling and amazing, but some of them were wrong. (India Today 2017) And of course the most famous example of a failure to provide backing is Fermat writing in a book he was reading that that there are no nontrivial instances of  $x^n + y^n = z^n$  for  $n > 2$  and then saying, “I have discovered a truly marvelous proof of this, which this margin is too narrow to contain.”

*the verifier cannot even input them before its runtime bound expires* Some authors instead define that the verifier runs in time polynomial in its input,  $\langle \sigma, \omega \rangle$ , and add the explicit restriction that  $|\omega|$  must be polynomial in  $|\sigma|$ .

*How is that legal?* This is reminiscent of Quantum Bogosort, a facetious sorting algorithm. Given an unordered list of length  $n$ , it uses a quantum source of randomness to generate a permutation of  $n$ . It reorders the input according to that permutation. If the list is now sorted then good. If not then the algorithm destroys the entire universe. Assuming the Many World’s Hypothesis, the result is that in any surviving universe the list has been sorted in linear time.

*Countdown* For a brief description see [https://en.wikipedia.org/wiki/Countdown\\_\(game\\_show\)](https://en.wikipedia.org/wiki/Countdown_(game_show)). A fun version, worth searching for videos, is [https://en.wikipedia.org/wiki/8\\_Out\\_of\\_10\\_Cats\\_Does\\_Countdown](https://en.wikipedia.org/wiki/8_Out_of_10_Cats_Does_Countdown).

*many-one reducible* The name comes from the fact that still another reducibility is one-one reducibility, where the function must be one-to-one.

*a graph* This is the Petersen graph, a rich source of counterexamples for conjectures in Graph Theory

*Drummer problem* This is often called the *Marriage* problem, where the men pick suitable women. But perhaps it is time for a new paradigm.

*NP complete* The name came from a contest run by Knuth; see <http://blog.computationalcomplexity.org/2010/11/by-any-other-name-would-be-just-as-hard.html>.

*there are many such problems* The “at least as hard” is true in the sense that such problems can answer questions about any other problem in that class. However, note that it might be that one NP complete problem runs in

nondeterministic time that is  $\mathcal{O}(n)$  while another runs in  $\mathcal{O}(n^{1000000})$  time. So this sense is at odds with our earlier characterization of problems that are harder to solve.

*These are the ones most often used* These are from the classic standard reference (Garey and Johnson 1979).

*a gadget* See <https://cs.stackexchange.com/a/1249/50343> from the Computer Science Stack Exchange user JeffE

*tied to whether or not P is unequal to NP* Ladner's theorem is that if  $P \neq NP$  then there is a problem in  $NP - P$  that is not  $NP$  complete.

*A large class* See (Karp 1972).

*an ending point* That is, as Pudlák observes, we treat  $P \neq NP$  as an informal axiom. (Pudlák 2013)

*caricature* Paul Erdős joked that a mathematician is a machine for turning coffee into theorems.

*completely within the realm of possibility that  $\phi(n)$  grows that slowly* Hartmanis observes (Hartmanis 2017) that it is interesting that Gödel, the person who destroyed Hilbert's program of automating mathematics, seemed to think that these problems quite possibly are solvable in linear or quadratic time.

*In 2018, a poll* The poll was conducted by W Gasarch, a prominent researcher and blogger in Computational Complexity. There were 124 respondents. For the description see <https://www.cs.umd.edu/users/gasarch/BLOGPAPERS/pollpaper3.pdf>. Note the suggestions that both respondents and even the surveyor took the enterprise in a light-hearted way.

*88% thought that  $P \neq NP$*  Gasarch divided respondents into experts, the people who are known to have seriously thought about the problem, and the masses. The experts were 99% for  $P \neq NP$ .

*S Aaronson has said* See (Roberts 2021) for both the Aaronson and Williams estimates.

*Cook is one* See (S. Cook 2000).

*Many observers* For example, (Viola 2018)

*$\mathcal{O}(n^{\lg 7})$  method* ( $\lg 7 \approx 2.81$ ) Strassen's algorithm is used in practice. The current record is  $\mathcal{O}(n^{2.37})$  but it is not practical. It is a galactic algorithm because while runs faster than any other known algorithm when the problem is sufficiently large, but the first such problem is so big that we never use the algorithm. For other examples see (Wikipedia contributors 2020b).

*Matching problem* The Drummer problem described earlier is a special case of this for bipartite graphs.

*more things to try than atoms in the universe* There are about  $10^{80}$  atoms in the universe. A graph with 100 vertices has the potential for  $\binom{100}{2}$  edges, which is about  $100^2$ . Trying every edge would be  $2^{10000} \approx 10^{10000/3.32}$  cases, which is much greater than  $10^{80}$ .

*since the 1960's we have an algorithm* Due to J Edmonds.

*Theory of Computing blog feed* (Various authors 2017)

*R J Lipton captured this feeling* (Lipton 2009)

*Knuth has a related but somewhat different take* (D. Knuth 2014)

*exploits the difference* Recent versions of the algorithm used in practice incorporate refinements that we shall not discuss. The core idea is unchanged.

*Their algorithm, called RSA* Originally the authors were listed in the standard alphabetic order: Adleman, Rivest, and Shamir. Adleman objected that he had not done enough work to be listed first and insisted on being listed last. He said later, “I remember thinking that this is probably the least interesting paper I will ever write.”

*tremendous amount of interest and excitement* In his 1977 column, Martin Gardner posed a \$100 challenge, to crack this message: 9686 9613 7546 2206 1477 1409 2225 4355 8829 0575 9991 1245 7431 9874 6951 2093 0816 2982 2514 5708 3569 3147 6622 8839 8962 8013 3919 9055 1829 9451 5781 5254 The ciphertext was generated by the MIT team from a plaintext (English) message using  $e = 9007$  and this number  $n$  (which is too long to fit on one line).

$$\begin{aligned} & 114, 381, 625, 757, 888, 867, 669, 235, 779, 976, 146, 612, 010, 218, 296, 721, 242, \\ & 362, 562, 561, 842, 935, 706, 935, 245, 733, 897, 830, 597, 123, 563, 958, 705, \\ & 058, 989, 075, 147, 599, 290, 026, 879, 543, 541 \end{aligned}$$

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*.

*computer searches suggest that these are very rare* For instance, among the numbers less than  $2.5 \times 10^{10}$  there are only 21 853  $\approx 2.2 \times 10^4$  pseudoprimes base 2; that’s six orders of magnitude.

*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  $2.7 \times 10^{14}$ , but only 246 683  $\approx 2.4 \times 10^5$ -many Carmichael numbers.

*the minimal pub crawl* See (W. Cook et al. 2017).

*An example is that the Free mathematics system Sage includes one* See also <https://www.youtube.com/watch?v=q8nQTNvCrjE> about the Concorde TSP solver.

## Appendices

*empty string, denoted  $\varepsilon$*  Possibly  $\varepsilon$  came as an abbreviation for ‘empty’. Some authors use  $\lambda$ , possibly from the German word for ‘empty’, *leer*. (Sirén 2016)

*reversal  $\sigma^R$  of a string* The most practical current notion of a string, the Unicode standard, does not have string reversal. All of the naive ways to reverse a string run into problems for arbitrary Unicode strings which may contain non-ASCII characters, combining characters, ligatures, bidirectional text in multiple languages, and so on. For example, merely reversing the chars (the Unicode scalar values) in a string can cause combining marks to become attached to the wrong characters. Another example is: how to reverse ab<backspace>ab? The Unicode Consortium has not gone through the effort to define the reverse of a string because there is no real-world need for it. (From <https://qntm.org/trick>.)

# Credits

## Prologue

I.1.11 SE user Shuzheng, <https://cs.stackexchange.com/q/45589/50343>

I.1.12 Question by SE user Arsalan MGR, <https://cs.stackexchange.com/q/135343/50343>

I.2.9 SE user Yuval Filmus, <https://cs.stackexchange.com/a/135170/50343>

I.2.13 <http://www.ivanociardelli.altervista.org/wp-content/uploads/2016/09/Solutions-to-exercises.pdf>

## Background

II.2 Image credit: Robert Williams and the Hubble Deep Field Team (STScI) and NASA.

II.1 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.2.39 The answer derives from one by Edward James, along with one by Keith Ramsay.

II.3.17 User scherk at pbworks.com.

II.3.27 Michael J Neely

II.3.29 Answer from Stack Exchange member Alex Becker.

II.4.1 ENIAC Programmers, 1946 U. S. Army Photo from Army Research Labs Technical Library

II.4.6 Started on Stack Exchange

II.4.9 From a Stack Exchange question.

II.5.12 CS SE user Kyle Strand <https://cs.stackexchange.com/q/11645/50343>.

II.5.13 SE user npostavs, <https://cs.stackexchange.com/a/44875/50343>

II.5.33 SE user Raphael <https://cs.stackexchange.com/a/44901/50343>

II.6.10 Question by SE user MathematicalOrchid, <https://cs.stackexchange.com/q/2811/67754>, and answer by SE user Andrej Bauer.

II.6.28 SE user Rajesh R

II.8.12 <http://people.cs.aau.dk/~srba/courses/tutorials-CC-10/t5-sol.pdf>

II.8.14 SE user Karolis Juodelė

II.8.17 SE user Noah Schweber

II.9.11 (Rogers 1987), p 214.

II.9.13 (Rogers 1987), p 214.

II.9.16 (Rogers 1987), p 214.

II.A.1 <https://www.ias.edu/ideas/2016/pires-hilbert-hotel>

II.C.2 <https://research.swtch.com/zip> and Kevin Matulef

II.C.3 [http://en.wikipedia.org/wiki/Quine\\_%28computing%29](http://en.wikipedia.org/wiki/Quine_%28computing%29)

II.C.4 J Avigad Computability and Incompleteness Lecture notes, [https://www.andrew.cmu.edu/user/avigad/Teaching/candi\\_notes.pdf](https://www.andrew.cmu.edu/user/avigad/Teaching/candi_notes.pdf)

## 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 [https://en.wikipedia.org/wiki/Buffalo\\_buffalo\\_Buffalo\\_buffalo\\_buffalo\\_buffalo\\_Buffalo\\_buffalo](https://en.wikipedia.org/wiki/Buffalo_buffalo_Buffalo_buffalo_buffalo_buffalo_Buffalo_buffalo),  
<https://cse.buffalo.edu/~rapaport/BuffaloBuffalo/buffalobuffalo.html>

III.2.36 <http://www.cs.utsa.edu/~wagner/CS3723/grammar/examples.html>

III.3.25 T Zaremba, <http://www.geom.uiuc.edu/~zaremba/graph3.html>.

III.A.10 <http://people.cs.ksu.edu/~schmidt/300s05/Lectures/GrammarNotes/bnf.html>

## Automata

IV.1.42 From *Introduction to Languages* by Martin, edition four, p 77.

IV.4.24 (Rich 2008), <https://math.stackexchange.com/a/1102627>

IV.4.27 (Rich 2008)

IV.5.19 SE user David Richerby, <https://cs.stackexchange.com/a/97885/67754>

IV.5.23 (Rich 2008)

IV.5.30 SE user Brian M Scott, <https://math.stackexchange.com/a/1508488>

IV.5.31 <https://cs.stackexchange.com/a/30726>

IV.6.15 <https://www.eecs.wsu.edu/~cook/tcs/l10.html>

## Complexity

V.4 Some of the discussion is from <https://softwareengineering.stackexchange.com/a/20833>.

V.4 Discussion of the third issue started as <https://cs.stackexchange.com/questions/9957/justification-for-neglecting-constants-in-big-o>.

V.4 The fourth point derives from <https://stackoverflow.com/a/19647659>.

V.4 This discussion originated as (SE author templatetypedef 2013).

V.1.50 Stack Exchange user templatetypedef <https://stackoverflow.com/a/19647659/7168267>

V.1.52 Stack Exchange user Daniel Fischer, <https://math.stackexchange.com/a/674039>, and Stack Exchange user anon, <https://math.stackexchange.com/a/61741>

V.1.58 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 T. McCulloch, <https://npcomplete.owu.edu/2014/06/03/3-dimensional-matching/>

V.2.64 Jan Verschelde, <http://homepages.math.uic.edu/~jan/mcs401/partitioning.pdf>

V.3.11 A.A. at <https://rjlipton.wordpress.com/2010/11/07/what-is-a-complexity-class/#comment-8872>

V.4.16 <https://cs.stackexchange.com/q/57518>

V.5.19 Paul Black, <https://xlinux.nist.gov/dads/HTML/nondetermAlgo.html>

V.6.26 SE user JesusIsLord at <https://cstheory.stackexchange.com/a/47031/4731>

V.6.28 SE user user326210, <https://math.stackexchange.com/a/2564255>

V.6.31 Neal E Young, University of California Riverside

V.2 By Psyon (Own work) CC BY-SA 3.0 [https://commons.wikimedia.org/wiki/File:Jigsaw\\_Puzzle.svg](https://commons.wikimedia.org/wiki/File:Jigsaw_Puzzle.svg)

V.7.12 William Gasarch, <https://www.cs.umd.edu/~gasarch/COURSES/452/F14/poly.pdf>

V.7.16 <http://www.cs.princeton.edu/courses/archive/fall02/cos126/exercises/np.html>

V.7.17 <http://www.cs.princeton.edu/courses/archive/fall02/cos126/exercises/np-sol.html>

V.7.20 <http://www.cs.princeton.edu/courses/archive/fall02/cos126/exercises/np.html>

V.7.23 Y Lyuu, <https://www.csie.ntu.edu.tw/~lyuu/complexity/2016/20161129s.pdf>

V.7.28 SE user Yuval Filmus <https://cs.stackexchange.com/a/132902/50343>

V.8.17 SE user Yuval Filmus <https://cs.stackexchange.com/a/54452/50343>

## **Appendices**

### **Notes**

# Bibliography

- A/V Geeks, YouTube 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, Scott (Aug. 14, 2011). *Why Philosophers Should Care About Computational Complexity*. URL: <https://arxiv.org/abs/1108.1791>.
- (May 3, 2012a). *The 8000th Busy Beaver number eludes ZF set theory: new paper by Adam Yedidia and me*. URL: <http://www.scottaaronson.com/blog/?p=2725>.
- (Aug. 30, 2012b). *The Toaster-Enhanced Turing Machine*. URL: <http://www.scottaaronson.com/blog/?p=1121> (visited on 05/28/2015).
- Adams, Douglas (1979). *The Hitchhiker's Guide to the Galaxy*. Harmony Books. ISBN: 9780345391803.
- Allender, Eric, Michael C. Loui, and Kenneth W. Regan (1997). “Complexity Classes”. In: ed. by Mikhail J. Atallah and Marina Blanton. Boca Raton, Florida: CRC Press. Chap. 27.
- Bellos, Alex (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, Ethan and Umesh Vazirani (1997). “Quantum Complexity Theory”. In: *SIAM Journal of Computing* 26.5, pp. 1411–1473.
- Bigham, D S (Aug. 19, 2014). *How Many Vowels Are There in English? (Hint: It's More Than 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](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, Robert (2000). “Proving Church’s Thesis”. In: *Philosophia Mathematica* 8, pp. 244–258.
- Blanda, Stephanie (2013). *The Six Degrees of Kevin Bacon*. [Online; accessed 2019-Apr-01]. URL: <https://blogs.ams.org/mathgradblog/2013/11/22/degrees-kevin-bacon/>.
- bobnica, stackoverflow user (2009). *Answer to: RegEx match open tags except XHTML self-contained tags*. URL: <https://stackoverflow.com/a/1732454/7168267> (visited on 01/27/2019).

- Bragg, Melvyn (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, David C. (2020). *Discovering Dennis Ritchie's Lost Dissertation*. [Online; accessed 2020-Jun-20]. URL: <https://computerhistory.org/blog/discovering-dennis-ritchies-lost-dissertation/>.
- Brower, Kenneth (1983). *The Starship and the Canoe*. Harper Perennial; Reprint edition. ISBN: 978-0060910303.
- Church, Alonzo (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.
- Computing, Free Online Dictionary of (2017). *Stephen Kleene*. [Online; accessed 21-June-2017]. URL: <http://foldoc.org/Stephen%20Kleene>.
- Cook, Stephen (2000). *The P vs NP Problem*. Official problem description. Clay Mathematics Institute. URL: <https://www.claymath.org/sites/default/files/pvsn.pdf> (visited on 01/11/2018).
- Cook, William 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. Jack (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, Russ (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, Martin (2004). "The Myth of Hypercomputation". In: *Alan Turing: Life and Legacy of a Great Thinker*. Ed. by Christof 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.
- Dershovitz, Nachum and Yuri 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, Jack (1965). "Paths, trees, and flowers". In: *Canadian Journal of Mathematics* 17, pp. 449–467.
- Eén, Niklas and Niklas Sörensson (2005). *MiniSat*. URL: <http://minisat.se/> (visited on 05/16/2022).
- Encyclopædia Britannica, The Editors of (2017). *Y2K bug*. URL: <https://www.britannica.com/technology/Y2K-bug> (visited on 05/10/2017).

- Euler, L (1766). "Solution d'une question curieuse que ne paroît soumise à 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, Niall (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/science/tech/article-1220286/Sir-Tim-Berners-Lee-admits-forward-slashes-web-address-mistake.html> (visited on 11/29/2018).
- Fortnow, Lance and Bill Gasarch (2002). *Computational Complexity Blog*. [Online; accessed 2017-Nov-13]. URL: <http://blog.computationalcomplexity.org/2002/11-foundations-of-complexitylesson-7.html>.
- Fraenkel, Aviezri S. and David 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.
- Gandy, Robin (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.
- Garey, Michael and David Johnson (1979). *Computers and Intractability, A Guide to the Theory of NP Completeness*. W. H. Freeman.
- Gleick, James (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 Paul Benacerraf and Hilary Putnam. Cambridge University Press, pp. 470–494.
- (1995). "Undecidable diophantine propositions". In: *Collected works Volume III: Unpublished essays and lectures*. Ed. by S. Feferman et al. Oxford University Press.
- Goodstein, R. L. (Dec. 1947). "Transfinite Ordinals in Recursive Number Theory". In: *Journal of Symbolic Logic* 12.4, pp. 123–129.
- Grossman, Lisa (2010). *Metric Math Mistake Muffed Mars Mereorology Misstion*. [Online; accessed 2017-May-25]. URL: <https://www.wired.com/2010/11/1110mars-climate-observer-report/>.
- Hartmanis, J (2017). *Gödel, von Neumann and the P =?NP Problem*. URL: <http://www.cs.cmu.edu/~15455/hartmanis-on-godel-von-neumann.pdf> (visited on 12/25/2017).
- Hennie, Fred (1977). *Introduction to Computability*. Addison-Wesley.
- Hilbert, David and Wilhelm Ackermann (1950). *Principles of theoretical logic*. Trans. by R E Luce. AMS Chelsea Publishing.
- Hodges, Andrew (2016). *Alan Turing* in the Stanford Encyclopedia of Philosophy. URL: <http://www.turing.org.uk/publications/stanford.html> (visited on 04/06/2016).
- (1983). *Alan Turing: the enigma*. Simon and Schuster. ISBN: 0-671-49207-1.
- Hofstadter, Douglas R. (1979). *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books.
- Hopcroft, John E, Rajeev Motwani, and Jeffrey D Ullman (2001). *Introduction to Automata Theory, Languages, and Computation*. 2nd ed. Pearson Education. ISBN: 0201441241.
- Huggett, Nick (2010). *Zeno's Paradoxes* — Stanford Encyclopedia of Philosophy. [Online; accessed 23-Dec-2016]. URL: <https://plato.stanford.edu/entries/paradox-zeno/#ParMot>.

- India Today (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).
- Joel David Hamkins, mathoverflow.net user (2010). *Answer to: Infinite CPU clock rate and hotel Hilbert*. URL: <https://mathoverflow.net/a/22038> (visited on 04/19/2017).
- Jones, Neil D. (1997). *Computability and Complexity From a Programming Perspective*. 1st ed. MIT Press. ISBN: 978-0262100649.
- Karp, Richard 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, Stephen (1952). *Introduction to Metamathematics*. North-Holland Amsterdam.
- 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, Donald (May 20, 2014). *Twenty Questions for Donald Knuth*. URL: <http://www.info.rmit.com/articles/article.aspx?p=2213858> (visited on 02/17/2018).
- Knuth, Donald E. (Dec. 1964). *Backus Normal Form vs. Backus Naur Form*. Letter to the Editor.
- Knuutila, Timo (2001). “Redescribing an algorithm by Hopcroft”. In: *Theoretical Computer Science* 250, pp. 333–363.
- Kragh, Helge (Mar. 27, 2014). *The True (?) Story of Hilbert’s Infinite Hotel*. URL: <http://arxiv.org/abs/1403.0059>.
- Leupold, Jacob, 1674–1727 (1725). “Details of the mechanisms of the Leibniz calculator, the most advanced of its time”. In: Illustration in: *Theatrum arithmeticо-geometricum, das ist . . . [bound with Theatrum machinarium, oder, Schau-Platz der Heb-Zeuge/Jacob Leupold. Leipzig, 1725]*. Leipzig: Zufinden bey dem Autore und Joh. Friedr. Gleditschens seel. Sohn: Gedruckt bey Christoph Zunkel, 1727. URL: <https://www.loc.gov/resource/cph.3c10471/> (visited on 11/14/2016).
- Levin, Leonid A. (Dec. 7, 2016). *Fundamentals of Computing*. URL: <https://www.cs.bu.edu/fac/lnd/toc/>.
- Lipton, Richard Jay (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, Jane (2017). “Epigenesis and Preformationism”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2017. Metaphysics Research Lab, Stanford University.
- McCarthy, John (1963). *A Basis for a Mathematical Theory of Computation*. URL: <http://www-formal.stanford.edu/jmc/basis1.pdf> (visited on 06/15/2017).
- Meyer, Albert R. and Dennis M. Ritchie (1966). *Research report: The complexity of loop programs*. Tech. rep. 1817. IBM.
- N. J. A. Sloane, editor (2019). *The On-Line Encyclopedia of Integer Sequences*, A000290. URL: <https://oeis.org/A000290> (visited on 03/02/2019).
- navyreviewer, YouTube user (2010). *Mechanical computer part 1*. URL: <https://www.youtube.com/watch?v=mpkTHyfr0pM> (visited on 08/09/2015).
- Odifreddi, Piergiorgio (1992). *Clasical Recursion Theory*. Elsevier Science. ISBN: 0-444-87295-7.

- Piccinini, Gualtiero (2017). "Computation in Physical Systems". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2017. Metaphysics Research Lab, Stanford University.
- Pinker, Steven (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, cs.stackexchange user (2014). *Answer to: What exactly is an algorithm?* URL: <https://cs.stackexchange.com/a/31953> (visited on 12/27/2018).
- Pudlák, Pavel (2013). *Logical Foundations of Mathematics and Computational Complexity*. Springer. ISBN: 978-3-319-34268-9.
- R H Bruck (n.d.). "Computational Aspects of Certain Combinatorial Problems". In: *AMS Symposium in Applied Mathematics* 6, p. 31.
- Radó, Tibor (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, Paul (2011). <http://rendell-attic.org/gol/tm.htm>. URL: <http://rendell-attic.org/gol/tm.htm> (visited on 07/21/2015).
- Rich, Elaine (2008). *Automata, Computability, and Complexity*. Pearson. ISBN: 978-0-13-228806-4.
- Roberts, Siobhan (Oct. 27, 2021). "The 50-year-old problem that eludes theoretical computer science". In: *MIT Technology Review*.
- Robinson, Raphael (1948). "Recursion and Double Recursion". In: *Bulletin of the American Mathematical Society* 10, pp. 987–993.
- Rogers Jr., Hartley (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.
- Scott, Brian M. (Feb. 14, 2020). *Inverting the Cantor pairing function*. Stack Exchange user <http://math.stackexchange.com/users/12042/brian-m-scott>. URL: <http://math.stackexchange.com/q/222835> (visited on 10/28/2012).
- SE 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>.
- SE 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>.
- SE 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>.
- SE 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>.

- SE 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>.
- SE author rici (2021). *Are modern programming languages context-free?* [Online; accessed 2021-May-11]. Stack Overflow discussion board. URL: <https://cs.stackexchange.com/a/140082/50343>.
- SE 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>.
- SE user 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).
- Sipser, Michael (2013). *Introduction to the Theory of Computation*. 3rd ed. Cengage. ISBN: 978-1-133-18779-0.
- Sirén, Jouni (CS StackExchange user) (2016). *Answer to: What is the origin of  $\lambda$  for empty string?* Accessed 2016-October-20. URL: <http://cs.stackexchange.com/a/64850/50343>.
- Smoryński, Craig (1991). *Logical Number Theory I*. Springer-Verlag.
- Soare, Robert I. (1999). “Computability and Incomputability”. In: *Handbook of Computability Theory*. Ed. by E. R. Griffor. North-Holland, Amsterdam, pp. 3–36.
- Thompson, Ken (Aug. 1984). “Reflections on trusting trust”. In: *Communications of the ACM* 27 (8), pp. 761–763.
- Thomson, James 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.
- (1938). “Systems of Logic Based on Ordinals”. PhD. Princeton University.
- U.S. Naval Observatory, Time Service Dept. (2017). *Leap Seconds*. [Online; accessed 10-May-2017]. URL: <http://tycho.usno.navy.mil/leapsec.html>.
- Unknown (1948). *UCLA’s 1948 Mechanical Computer*. Accessed 2019-September-18. URL: <https://vimeo.com/70589461>.
- Various authors (2017). *Theory of Computing Blog Aggregator*. [Online; accessed 17-May-2017]. URL: <http://cstheory-feed.org/>.
- Viola, Emanuele (Feb. 16, 2018). *I believe P=NP*. URL: <https://emanueleviola.wordpress.com/2018/02/16/i-believe-pnp/> (visited on 02/16/2018).
- Widgerson, Avi (2017). *Mathematics and Computation*. [Draft of a to-be-published book; accessed 2017-Oct-27]. URL: <https://www.math.ias.edu/avi/book>.
- Wikipedia (2016). *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](https://en.wikipedia.org/w/index.php?title=The_Imitation_Game&oldid=723336480).
- Wikipedia contributors (2014). *History of the Church–Turing thesis — Wikipedia, The Free Encyclopedia*. [Online; accessed 2-October-2016]. URL: [https://en.wikipedia.org/w/index.php?title=History\\_of\\_the\\_Church%20%93Turing\\_thesis&oldid=618643863](https://en.wikipedia.org/w/index.php?title=History_of_the_Church%20%93Turing_thesis&oldid=618643863).
- (2015). *Stigler’s law of eponomy — Wikipedia, The Free Encyclopedia*. URL: [https://en.wikipedia.org/w/index.php?title=Stigler%27s\\_law\\_of\\_eponomy&oldid=691378684](https://en.wikipedia.org/w/index.php?title=Stigler%27s_law_of_eponomy&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](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](https://en.wikipedia.org/w/index.php?title=Donald_Michie&oldid=708156000) (visited on 03/24/2016).
- (2016c). *Nomogram* — Wikipedia, The Free Encyclopedia. [Online; accessed 6-October-2016]. URL: <https://en.wikipedia.org/w/index.php?title=Nomogram&oldid=742964268>.
- (2016d). *Ross–Littlewood paradox* — Wikipedia, The Free Encyclopedia. [Online; accessed 9-February-2017]. URL: [https://en.wikipedia.org/w/index.php?title=Ross%20%80%93Littlewood\\_paradox&oldid=739534216](https://en.wikipedia.org/w/index.php?title=Ross%20%80%93Littlewood_paradox&oldid=739534216).
- (2016e). *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](https://en.wikipedia.org/w/index.php?title=Turtles_all_the_way_down&oldid=736001775).
- (2016f). *Zeno's paradoxes* — Wikipedia, The Free Encyclopedia. [Online; accessed 23-December-2016]. URL: [https://en.wikipedia.org/w/index.php?title=Zeno%27s\\_paradoxes&oldid=752685211%7D](https://en.wikipedia.org/w/index.php?title=Zeno%27s_paradoxes&oldid=752685211%7D).
- (2017a). *15 puzzle* — Wikipedia, The Free Encyclopedia. [Online; accessed 16-September-2017]. URL: [https://en.wikipedia.org/w/index.php?title=15\\_puzzle&oldid=789930961](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](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](https://en.wikipedia.org/w/index.php?title=Artificial_neuron&oldid=780239713).
- (2017d). *Aubrey–Maturin series* — Wikipedia, The Free Encyclopedia. [Online; accessed 28-March-2017]. URL: [https://en.wikipedia.org/w/index.php?title=Aubrey%20%80%93Maturin\\_series&oldid=771937634](https://en.wikipedia.org/w/index.php?title=Aubrey%20%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%20%80%93Naur\\_form&oldid=778354081](https://en.wikipedia.org/w/index.php?title=Backus%20%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](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](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](https://en.wikipedia.org/w/index.php?title=Pax_Britannica&oldid=775067301).
- (2017j). *Platonic solid* — Wikipedia, The Free Encyclopedia. [Online; accessed 2017-October-22]. URL: [https://en.wikipedia.org/w/index.php?title=Platonic\\_solid&oldid=801264236](https://en.wikipedia.org/w/index.php?title=Platonic_solid&oldid=801264236).

- Wikipedia contributors (2017k). *Unicode* — Wikipedia, The Free Encyclopedia. URL: <https://en.wikipedia.org/w/index.php?title=Unicode&oldid=784443067>.
- (2017l). *Zermelo's theorem (game theory)* — Wikipedia, The Free Encyclopedia. [Online; accessed 2017-Nov-26]. URL: [https://en.wikipedia.org/w/index.php?title=Zermelo%27s\\_theorem\\_\(game\\_theory\)&oldid=806070716](https://en.wikipedia.org/w/index.php?title=Zermelo%27s_theorem_(game_theory)&oldid=806070716).
- (2018). *Paradox* — Wikipedia, The Free Encyclopedia. [Online; accessed 14-December-2018]. URL: <https://en.wikipedia.org/w/index.php?title=Paradox&oldid=871193884>.
- Wikipedia contributors (2017). *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](https://en.wikipedia.org/w/index.php?title=Philipp_von_Jolly&oldid=764485788).
- (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](https://en.wikipedia.org/w/index.php?title=Mathematics:_The_Loss_of_Certainty&oldid=879406248).
- (2019c). *Maxwell's demon* — Wikipedia, The Free Encyclopedia. [Online; accessed 1-January-2020]. URL: [https://en.wikipedia.org/w/index.php?title=Maxwell%27s\\_demon&oldid=930445803%7D](https://en.wikipedia.org/w/index.php?title=Maxwell%27s_demon&oldid=930445803%7D).
- (2019d). *Partial application* — Wikipedia, The Free Encyclopedia. [Online; accessed 26-December-2019].
- (2020a). *Foobar* — Wikipedia, The Free Encyclopedia. [Online; accessed 2020-Feb-14]. URL: <https://en.wikipedia.org/w/index.php?title=Foobar&oldid=934819128>.
- (2020b). *Galactic algorithm* — Wikipedia, The Free Encyclopedia. [Online; accessed 2020-Jun-17]. URL: [https://en.wikipedia.org/w/index.php?title=Galactic\\_algorithm&oldid=957279293](https://en.wikipedia.org/w/index.php?title=Galactic_algorithm&oldid=957279293).
- William S. Renwick (May 6, 1949). *The start of the EDSAC log*. [Online; accessed 2019-Mar-02]. URL: <https://www.cl.cam.ac.uk/relics/elog.html>.

# Index

- + operation on a language, 218
- 15 Game problem, 286, 299
- 3 Dimensional Matching problem, 331
- 3-Coloring problem, 333
- 3-SAT, *see* 3-Satisfiability problem, *see also* Satisfiability problem
- 3-Satisfiability
  - Strict variant, 326
- 3-Satisfiability problem, 281, 299, 326–328, 331, 344
- 4-Satisfiability problem, 343
- accept a language, 149
- accept an input, 186, 194, 197
- acceptable numbering, 73
- accepted language, *see also* recognized, 293
- accepting state, 13, 181, 182, 308
  - nondeterministic Finite State machine, 193
  - Pushdown machine, 237
- accepts, 182, 186, 194, 197
- Ackermann function, 31–33, 35, 49–53
- Ackermann, W, 3
  - picture, 33
- action set, 8
- action symbol, 8
- addition, 6
- adjacency matrix, 166
- adjacent, 164
- Adleman, L
  - picture, 350
- Agrawal, M
  - picture, 287
- AKS primality test, 287
- algorithm, 292
  - definition, 292
  - reliance on model, 292
- alphabet, 147, 368
  - input, 182
  - Pushdown machine, 237
  - tape, 8
- amb function, 398
- ambiguous grammar, 157
- argument, to a function, 370
- Aristotle’s Paradox, 61, 63
- Assignment problem, 328
- asymptotically equivalent, 267, 276
- atom, 287
- Backus, J
  - picture, 173
- Backus-Naur form, BNF, 173
- Berra, Y
  - picture, 191
- Big  $\mathcal{O}$ , 265
- Big  $\Theta$ , 267
- bijection, 373
- bit string, *see also* bitstring
- bitstring, 368
- blank, 8, 308
- blank, B, 5
- BNF, 173–177
- body of a production, 152
- Bogosort, 407
- Boole, G
  - picture, 280
- boolean, 280
  - expression, 280
  - function, 280
  - variable, 280
- Boolean algebra, 376
- Boolean function, 377
- bottom,  $\perp$ , 237
- BPP, Bounded-Error Probabilistic Polynomial Time problem, 349
- bridge, 298
- Broadcast problem, 283
- Brocard’s problem, 102
- Busy Beaver, 136–139
- button
  - start, 5
- c.e. set, *see also* computably enumerable
- caching, 71
- Cantor’s correspondence, 68–76
- Cantor’s Theorem, 78
- Cantor, G
  - and diagonalization, 389
  - picture, 63
- cardinality, 61–83

less than or equal to, 78  
cellular automaton, 46  
certificate, 311  
**Chromatic Number** problem, 280, 295  
Church's Thesis, 14–21  
    and uncountability, 79  
    argument by, 19  
    clarity, 17  
    consistency, 16  
    convergence, 16  
    coverage, 15  
    Extended, 303  
Church, A  
    picture, 14  
    Thesis, 15  
circuit, 165, 302  
    Euler, 165  
    gate, 302  
    Hamiltonian, 165  
    wire, 302  
**Circuit Evaluation** problem, 302  
class, 148, 301  
    complexity, 301  
**Class Scheduling** problem, 335  
**Clique** problem, 282, 305, 320, 332  
clique, in a graph, 282  
closed walk, 164  
closure under an operation, 215  
CNF, 160, 281, 327, 333, 359, 377  
CNF, Conjunctive Normal Form, 280  
co-NP, 312  
Cobham's Thesis, 271  
Cobham, A  
    picture, 401  
    Thesis, 271  
codomain, 370  
codomain versus range, 371  
Collatz conjecture, 100  
coloring of a graph, 166  
colors, 279  
complete, 117  
    for a class, 330  
    NP, 330  
complete graph, 168  
complexity class, 301  
    canonical, 348  
EXP, 345  
NP, 310  
P, 301  
    polytime, 301  
complexity function, 265  
Complexity Zoo, 301, 348  
composition, 373  
computable  
    relative to a set, 114  
    set, 109  
computable function, 11  
computable relation, 11  
computable set, 11  
computably enumerable, 108–113  
    in an oracle, 121  
    K is complete, 117  
computably enumerable set, 109  
    collection of, RE, 306  
computation  
    distributed, 292  
    Finite State machine, 186  
    nondeterministic Finite State machine,  
        194, 197  
    relative to an oracle, 114  
    step, 8  
    Turing machine, 9  
concatenation of languages, 148  
concatenation of strings, 368  
configuration, 8, 186, 194, 197  
    halting, 186, 194, 197  
    initial, 8  
conjunction, 375  
Conjunctive Normal Form, 333, 359, 377  
Conjunctive Normal form, 160  
conjunctive normal form, 280  
connected component, 298  
connected graph, 165  
connectives  
    logical, 375  
context free  
    grammar, 153  
    language, 244  
control, of a machine, 5  
converge, 10  
Conway, J  
    picture, 46  
Cook reducibility, 319  
Cook, S

picture, 330  
Cook-Levin theorem, 330  
correspondence, 62, 373  
    Cantor's, 70  
countable, 64  
countably infinite, 64  
Course Scheduling problem, 289  
CPU of Turing machine, 5  
Crossword problem, 285  
current symbol, 8  
CW, 169  
cycle, 164  
**Cyclic Shift** problem, 324

daemon, *see* demon  
DAG, directed acyclic graph, 164  
dangling else, 157  
De Morgan, A  
    picture, 279  
decidable, 293  
    language, 103  
    set, 109  
decidable language, 306  
decide a language, 149  
decided, 13  
decided language, 186, 293  
    of a nondeterministic Turing machine,  
        309  
decider, 11  
decides  
    language, 306  
    set, 11  
decides language, 306  
decision problem, 3, 11, 293  
decrypter, 351  
degree of a vertex, 167  
degree sequence, 167  
demon, or daemon, 193  
DeMorgan's laws  
    for logic expressions, 376  
derivation, 153, 155  
derivation tree, 153  
description number, 73  
determinism, 8, 16  
diagonal enumeration, 70  
diagonalization, 76–83, 121, 122  
    effectivized, 92

digraph, 164  
directed acyclic graph, 164  
directed graph, 164  
Discrete Logarithm problem, 299  
disjunction, 375  
Disjunctive Normal form, 376  
distinguishable states, 227  
distributed computation, 292  
distributive laws  
    for logic expressions, 376  
diverge, 10  
**Divisor** problem, 286, 299  
DNF, 376  
domain, 370, 371  
    in the Theory of Computation, 371  
**Double-SAT** problem, 317  
doubler function, 3, 12  
dovetailing, 109  
Droste effect, 394  
Drummer problem, 326  
DSPACE, 347  
**DTIME**, 346

edge, 163  
edge weight, 164  
Edmunds, J  
    picture, 401  
effective, 3  
effective function, 9–11  
Electoral College, 285  
empty language  
    decision problem, 298  
empty string,  $\epsilon$ , 8, 368  
encrypter, 351  
*Entscheidungsproblem*, 3, 11, 14, 293, 338  
    unsolvability, 99  
enumerate, 64  
 $\epsilon$  closure, 200  
 $\epsilon$  moves, 196  
 $\epsilon$  transitions, 196–199  
equinumerous sets, 63  
equivalent growth rates, 267  
equivalent propositional logic statements,  
    375  
**Euler Circuit** problem, 279, 299  
Euler circuit, 165  
Euler, L

picture, 278  
 eval, 85  
 exclusive or, 44  
**EXP**, 345–346  
 expansion of a production, 152  
**Ext**, extensible functions, 120  
 Extended Church’s Thesis, 303  
 extended regular expression, 245  
 extended transition function, 186  
     for nondeterministic Finite State machines, 197  
     nondeterministic Finite State machine, 194  
 extensible, 120  
 $\mathcal{F}$ - $SAT$  problem, 299  
**Factoring** problem, 350, 351  
**Prime Factorization** problem, 299  
 Fermat number, 36  
 Fibonacci numbers, 29  
**Fin** problem, 328  
 final state, 181, 182  
     nondeterministic Finite State machine, 193  
 finite set, 64  
 Finite State automata, *see* Finite State machine  
 Finite State machine, 180–191  
     accept string, 186, 197  
     accepting state, 182  
     alphabet, 182  
     computation, 186  
     configuration, 186  
     final state, 182  
     halting configuration, 186  
     initial configuration, 186  
     input string, 186  
     language of, 186  
     minimization, 226–236  
     next-state function, 182  
     nondeterminism, 307  
     nondeterministic, 193  
     powerset construction, 199  
     product, 215  
     reject string, 186  
     state, 182  
     step, 186  
     transition function, 182  
 Fixed point theorem, 121–127  
     discussion, 125–126  
 Flauros, Duke of Hell  
     picture, 193  
 flow, 326  
 flow chart, 85  
**Four Color** problem, 279  
 function, 370  
     91 (McCarthy), 30  
 Ackermann, 50  
 argument, 370  
 Big  $\mathcal{O}$ , 265  
 Big  $\Theta$ , 267  
 Boolean, 377  
 boolean, 280  
 codomain, 370  
 composition, 373  
 computable, 11  
 computed by a Turing machine, 9  
 converge, 10  
 correspondence, 62, 373  
 definition, 370  
 diverge, 10  
 domain, 370  
 doubler, 3, 12  
 effective, 3  
 enumeration, 64  
 exponential growth, 269  
 extended transition, 186  
 extensible, 120  
 general recursive, 35  
 identity, 373  
 image under, 371  
 index, 370  
 injection, 372  
 inverse, 373  
 left inverse, 373  
 logarithmic growth, 269  
 $\mu$  recursive function (mu recursive), 35  
 next-state, 8, 182  
 one-to-one, 62  
 oneo-to-one, 372  
 onto, 62, 372  
 order of growth, 265  
 output, 370  
 pairing, 69, 70, 139

- partial, 10, 371
- partial recursive, 35
- polynomial growth, 269
- predecessor, 6
- projection, 24, 35
- range, 371
- recursive, 11, 35
- reduction, 319
- restriction, 371
- right inverse, 373
- successor, 12, 21, 24, 35
- surjection, 372
- total, 10, 120, 371
- transition, 8, 182, 308
- translation, 319
- unpairing, 69, 70, 139
- value, 370
- well-defined, 370, 371
- zero, 24, 35
- function problem, 293
- functions, 370–374
  - same behavior, 103
  - same order of growth, 267
- gadget
  - example of, 334
  - in complexity arguments, 333
- Galilei, G (Galileo)
  - picture, 61
- Galileo, 61
- Galileo’s Paradox, 61, 63, 65
- Game of Life, 46–49
- gate, 43, 302
- general recursion, 31–37
- general recursive function, 35
- general unsolvability, 96–99
- Gödel number, 73
- Gödel, K, 14
  - letter to von Neumann, 339
  - picture, 15
  - picture with Einstein, 130
- Gödel’s theorem, 14
- Goldbach’s conjecture, 102
- grammar, 152–163
  - ambiguous, 157
  - Backus-Naur form, BNF, 173
  - BNF, Backus-Naur form, 173
- body of a production, 152
- context free, 153
- derivation, 153
- expansion of a production, 152
- head, 152
- linear, 204
- nonterminal, 152
- production, 152, 153
- rewrite rule, 152, 153
- right linear, 204
- start symbol, 153
- syntactic category, 153
- terminal, 152
- graph, 163–173
  - adjacent edges, 164
  - bridge edge, 298
  - circuit, 165
  - clique, 282
  - closed walk, 164
  - coloring, 166–167
  - colors, 279
  - complete, 168
  - connected, 165
  - connected component, 298
  - cycle, 164
  - degree sequence, 167
  - digraph, 164
  - directed, 164
  - directed acyclic, 164
  - edge, 163
  - edge weight, 164
  - Euler circuit, 165
  - Hamiltonian circuit, 165
  - induced subgraph, 165
  - isomorphism, 167–168
  - loop, 164
  - matrix representation, 166
  - multigraph, 164
  - node, 163
  - open walk, 164
  - path, 164
  - planar, 169, 279
  - representation, 165–166
  - simple, 163
  - spanning subgraph, 282
  - subgraph, 165
  - trail, 164

transition, 7  
 traversal, 164–165  
 tree, 164, 282  
 vertex, 163  
 vertex cover, 282  
 vertex degree, 167  
 walk, 164  
 walk length, 164  
 weighted, 164  
**Graph Colorability** problem, 280, 299, 321, 335  
**Graph Connectedness** problem, 298, 300  
**Graph Isomorphism** problem, 298, 336  
 Grassmann, H, 22  
     picture, 21  
 guessing  
     by a machine, 193  
 guessing by a machine, 196  
  
 Hailstone function, 100  
 Halt light, 5  
 halting configuration, 186, 194, 197  
 Halting problem, 92–94, 103  
     as a decision problem, 299  
     discussion, 94–95  
     in intellectual culture, 128–131  
     reduction to another problem, 99  
     relativized, 117  
     significance, 95–96  
     unsolvability, 93  
 halting state, 12  
**Halts On Three** problem, 96, 115, 318  
 Hamilton, W R  
     picture, 277  
 Hamiltonian circuit, 165  
**Hamiltonian Circuit** problem, 277, 299, 313, 325, 332  
**Hamiltonian Path** problem, 313, 343  
 hard  
     for a class, 330  
     NP, 330  
 haystack, 302  
 head  
     read/write, 4  
 head of a production, 152  
 Hilbert’s Hotel, 127–128  
 Hilbert, D, 3

picture, 129  
 Hofstadter, D, 394  
 hyperoperation, 32  
  
 I/O head, *see* read/write head  
 identity function, 373  
*Ignorabimus*, 129  
 image under a function, 371  
 implies, 45  
 Incompleteness Theorem, 14  
**Independent Set** problem, 290, 300, 325, 327, 344  
 index number, 73  
 index set, 103  
 induced subgraph, 165  
 infinite set, 64  
 infinity, 61–68, 83  
 initial configuration, 8, 186, 194, 197  
 injection, 372  
 input alphabet, 182  
 input string, 186, 194, 197  
 input symbol, 8  
 input, to a function, 370  
 instruction, 5, 8, 308  
**Integer Linear Programming** problem, 316  
     decision problem, 327  
 inverse of a function, 373  
     left, 373  
     right, 373  
     two-sided, 373  
 isomorphic, 167  
 isomorphism, 167  
  
*k* Coloring problem, 166, 280  
 K, the Halting problem set, 93, 110  
     complete among c.e. sets, 117  
 K<sub>0</sub>, set of halting pairs, 102, 112, 116  
 Karatsuba, A, 262  
 Karp reducible, 319  
 Karp, R  
     picture, 331  
 Kayal, N  
     picture, 287  
 Kleene star, 64, 147, 149, 368  
     regular expression, 206  
 Kleene’s fixed point theorem, 122  
 Kleene’s theorem, 207–211

Kleene, S, 35  
picture, 205  
 $K_n$ , 168  
Knapsack problem, 284, 293, 317, 322, 332  
Knight's Tour problem, 277  
Knuth, D  
picture, 272  
Kolmogorov, A  
picture, 261  
König's lemma, 165  
Königsberg, 278  
L'Hôpital's Rule, 268  
lambda calculus,  $\lambda$  calculus, 15  
language, 147–151  
+ operation, 218  
accept, 149  
accepted by a Finite State machine, *see*  
language, recognized by a Finite  
State machine  
accepted by Turing machine, 107, 293  
class, 148  
concatenation, 148  
context free, 244  
decidable, 103, 306  
decide, 149  
decided, 293  
decided by a Finite State machine, 186  
decided by a Turing machine, 13, 306  
decision problem, 293  
derived from a grammar, 155  
grammar, 153  
Kleene star, 149  
non-regular, 220–226  
of a Finite State machine, 186  
of a nondeterministic Finite State ma-  
chine, 194  
operations on, 148  
power, 149  
recognize, 149  
recognized, 293  
recognized by a Finite State machine,  
186  
recognized by Turing machine, 293  
regular, 214–220  
reversal, 149  
verifier, 311  
language decision problem, 293  
last in, first out (LIFO) stack, 237  
left inverse, 373  
leftmost derivation, 153  
LEGO, 5  
length, 164  
length of a string, 368  
Life, Game of, 46–49  
LIFO stack, 237  
light  
Halt, 5  
Linear Divisibility problem, 317  
Linear Programming language decision prob-  
lem, 285, 300, 316, 326  
Lipton's Thesis, 296  
logical connectives, 375  
logical operator  
and, 375  
not, 375  
or, 375  
logical operators, 375  
Longest Path problem, 317, 343  
loop, 164  
LOOP program, 53–58  
machine  
state, 9  
many-one reducible, 319  
map, *see* function  
Marriage problem, *see* Drummer problem or  
Matching problem  
Matching problem, 341  
matching, three dimensional, 283  
Max-Flow problem, 326  
McCarthy's 91 function, 30  
memoization, 71  
memory, 4  
metacharacter, 153, 173, 205  
Minimal Spanning Tree problem, 293  
minimization, 34  
minimization of a Finite State machine, 226–  
236  
minimization, unbounded, 35  
Minimum Spanning Tree problem, 282  
modulus, 351  
Morse code, 169  
 $\mu$ -recursion (mu recursion), 34

$\mu$  recursive function, 35  
 multigraph, 164  
 multiset, 284  
 Musical Chairs, 78  
  
 $n$ -distinguishable states, 227  
 $n$ -indistinguishable states, 227  
 Naur, P  
     picture, 173  
 Nearest Neighbor problem, 298, 300  
 negation, 375  
 next state, 5, 8  
 next tape symbol, 5  
 next-state function, 8, 182  
     nondeterministic Finite State machine,  
         193  
 NFSM, *see* nondeterministic Finite State machine  
 node, 163  
 nondeterminism, 191–204  
     for Finite State machines, 193, 307  
     for Turing machines, 307  
 nondeterministic Finite State machine, 193  
     accept string, 194, 197  
     computation, 194, 197  
     configuration, 194, 197  
     convert to a deterministic machine, 199  
      $\epsilon$  moves, 196  
      $\epsilon$  transitions, 196  
     halting configuration, 194, 197  
     initial configuration, 194, 197  
     input string, 194, 197  
     language of, 194  
     language recognized, 194  
     reject string, 194, 197  
 nondeterministic machine  
     recognizes a language, 318  
 nondeterministic Pushdown machine, 241–  
     243  
 nondeterministic Turing machine  
     accepting state, 308  
     decided language, 309  
     definition, 308  
     instruction, 308  
     transition function, 308  
 nonterminal, 152, 153  
 NP, 307–318  
  
 NP complete, 330–336  
     basic problems, 331  
 NP hard, 330  
 NSPACE, 347  
 NTIME, 346  
 numbering, 73  
     acceptable, 73  
  
 $\Omega$ , Big Omega, 267  
 $\sigma$ , omicron, 267  
 one-to-one function, 62, 372  
 onto function, 62, 372  
 open walk, 164  
 operators  
     logical, 375  
 optimization problem, 293  
 oracle, 113–121  
     computably enumerable in, 121  
 oracle Turing machine, 114  
 order of growth, 261–276  
     function, 265  
     Hardy hierarchy, 270  
 ouroboros, 84  
 output, from a function, 370  
  
 P, 301–307  
 P hard, 323  
 P versus NP, 310, 336–341  
 pairing function, 69, 70, 139  
 Paley, W  
     picture, 131  
 palindrome, 14, 148, 241, 369  
 paradox  
     Aristotle's, 61  
     Galileo's, 61  
     Zeno's, 65  
 parameter, 87  
 Parameter theorem, 88  
 parametrization, 87–90  
 parametrizing, 87  
 parse tree, 153  
 partial function, 10, 371  
 partial recursive function, 35  
 Partition problem, 284, 316, 332, 333, 344  
 path, 164  
 perfect number, 95  
 Péter, R

picture, 49  
 Petersen graph, 168, 170  
 pipe symbol, | , 152  
 planar graph, 169, 279  
 pointer, in C, 125  
 polynomial time, 301  
 polynomial time reducibility, 319  
 polytime, 301  
 power of a language, 149  
 power of a string, 369  
 powerset construction, 199  
 predecessor function, 6  
 prefix of a string, 369  
 present state, 5, 8  
 present tape symbol, 5  
 primality, 287  
**Primality** problem, 286, 293, 299, 313  
**Prime Factorization** problem, 286, 292, 293,  
     336, 342  
 primitive recursion, 22–30, 35  
     arity, 23  
 primitive recursive functions, 24  
 private key, 351  
 problem, 292  
     decision, 293  
     function, 293  
     Halting, 93, 94  
     language decision, 293  
     optimization, 293  
     search, 293  
     unsolvable, 94  
 problem miscellany, 277–291  
 problems  
     tractable, 271  
     unsolvable, 108  
 product construction, 215  
 production, 153  
 production in a grammar, 152  
 program, 292  
 projection function, 24, 35  
 property  
     semantic, 103  
 proplogic, 374–377  
 Propositional logic  
     Conjunctive Normal form, 160, 281, 327,  
         359, 377  
     Disjunctive Normal form, 376  
     propositional logic  
         atom, 287  
         Boolean algebra, 376  
         Boolean function, 377  
         DeMorgan’s laws, 376  
         distributive laws, 376  
         exclusive or, 44  
         implication, 45  
         operators, 375  
     pseudopolynomial, 274  
 public key, 351  
 Pumping lemma, 221  
 pumping length, 221  
 Pushdown automata, *see* pushdown machine  
 Pushdown machine, 237–245  
     halting, 238  
     input alphabet, 237  
     nondeterministic, 241–243  
     stack alphabet, 237  
     transition function, 237, 238  
 pushdown stack, 237  
 Quantum Bogosort, 407  
 Quantum Computing, 303  
     Quantum Supremacy, 303  
 Quantum Supremacy, 303  
 quine, 132  
 Quine’s paradox, 394  
 r.e. set, *see* computably enumerable set  
 Radó, T  
     picture, 137  
 RAM, *see* Random Access machine  
 Random Access machine, 271  
 range of a function, 371  
 RE, computably enumerable sets, 306  
 RE, computably enumerable sets, 294  
 reachable vertex, 165, 281  
 read/write head, 4  
 REC, computable sets, 294, 405  
 recognize a language, 149  
 recognized  
     by a nondeterministic Finite State ma-  
         chine, 194  
     recognized language, 293  
         of a Finite State machine, 186  
 recognizes a language

nondeterministic machine, 318  
recursion, 21–37  
Recursion theorem, 122  
recursive function, 11, 35  
recursive set, 11  
recursively enumerable set, *see* computably enumerable set  
reduces to, 114  
reducibility  
    Cook, 319  
    Karp, 319  
    polynomial time, 319  
    polytime, 319  
    polytime many-one, 319  
    polytime mapping, 319  
    polytime Turing, 319  
reducible  
    many-one, 319  
reduction  
    from the Halting problem, 99  
reduction function, 319  
Reductions between problems, 318–329  
*Reflections on Trusting Trust*, 135  
Reg problem, 328  
regex, 245  
regular expression, 205–214  
    extended, 245  
    in practice, 245–253  
    operator precedence, 206  
    regex, 245  
    semantics, 206  
    syntax, 206  
regular language, 214–220  
reject an input, 186, 194, 197  
rejecting state, 13  
rejects, 182  
relation, computable, 11  
relativized Halting problem, 117  
replication of a string, 369  
representation, of a problem, 296  
restriction, 371  
reversal of a language, 149  
reversal of a string, 369  
rewrite rule, 152, 153  
Rice’s theorem, 103–108  
right inverse, 373  
right linear, 204  
Ritchie, D, 53  
    picture, 53  
Rivest, R  
    picture, 350  
RSA Encryption, 350–355  
*s-m-n* theorem, 88  
same behavior, functions with, 103  
same order of growth, 267  
*SAT*, *see* Satisfiability, 294, 295, 326  
*SAT* solver, 358  
Satisfiability problem, 281, 290, 311, 320, 321, 330  
    as a language recognition problem, 294  
    on a nondeterministic Turing machine, 310  
satisfiable, 280  
Satisfying assignment problem, 295  
Saxena, N  
    picture, 287  
schema of primitive recursion, 23  
Science United, 292  
search problem, 293  
self reproducing program, 132  
self reproduction, 131–136  
self-delimiting code  
    of a string, 9  
semantic property, 103  
semicomputable set, 109  
semidecidable set, 109  
semidecide a language, 149  
semiprime, 286  
Semiprime problem, 316  
set  
    c.e., 109  
    cardinality, 63  
    computable, 11, 109  
    computably enumerable, 108–113  
    countable, 64  
    countably infinite, 64  
    decidable, 109  
    decider, 11  
    equinumerous, 63  
    finite, 64  
    index, 103  
    infinite, 64  
    oracle, 113–121

r.e., *see* computably enumerable set  
 recursive, 11  
 recursively enumerable, *see* computably enumerable set  
 reduces to, 114  
 semicomputable, 109  
 semidecidable, 109  
 $T$  equivalent, 116  
 Turing equivalent, 116  
 uncountable, 77  
 undecidable, 94  
**Set Cover** problem, 325  
 Shamir, A  
     picture, 350  
 Shannon, C  
     picture, 42  
**Shortest Path** problem, 279, 293, 299, 300, 320  
     ~, asymptotically equivalent, 276  
 simple graph, 163  
**SPACE**, 347  
 span a graph, 282  
 spanning subgraph, 282  
 **$st$ -Connectivity** problem, *see* Vertex-to-Vertex Path problem  
 **$st$ -Path** problem, *see* Vertex-to-Vertex Path problem  
 stack, 237  
     alphabet, 237  
     bottom,  $\perp$ , 237  
     LIFO, Last-In, First-Out, 237  
     pop, 237  
     push, 237  
 Start button, 5, 182  
 start state, 5, 182  
     Pushdown machine, 237  
 start symbol, 153  
 state, 182  
     accept, 13  
     accepting, 181, 182, 308  
     final, 181, 182  
     halting, 12  
     next, 5  
     present, 5  
     reject, 13  
     start, 5  
     unreachable, 107  
                 working, 12  
 state machine, 9  
 states, 4  
     distinguishable, 227  
     n-distinguishable, 227  
     n-indistinguishable, 227  
     set of, 8  
 Stator Square, 404  
**STCON** problem, *see* Vertex-to-Vertex Path problem  
 step, 8, 186  
 store, of a machine, 4  
 str(...), 297  
**Strict 3-Satisfiability**, 326  
 string, 147, 368–369  
     concatenation, 368  
     decomposition, 369  
     empty, 8, 368  
     length, 368  
     power, 369  
     prefix, 369  
     replication, 369  
     reversal, 369  
     self-delimiting code, 9  
     substring, 369  
     suffix, 369  
 string accepted  
     by deterministic Finite State machine, 182, 186  
     by nondeterministic Finite State machine, 194, 197  
 string rejected, 182  
**String Search** problem, 302  
 subgraph, 165  
     induced, 165  
**Subset Sum** problem, 284, 293, 300, 317, 322, 344  
 substring, 369  
**Substring** problem, 324  
 successor function, 12, 21, 24, 35  
 suffix of a string, 369  
 surjection, 372  
 symbol, 8, 147, 368  
     action, 8  
     current, 8  
     input, 8  
     next, 8

syntactic category, 153  
syntactic property, 103

*T* equivalent, 116  
*T* reducible, 114  
table, transition, 7  
tape, 4  
tape alphabet, 8  
tape symbol, 8  
    blank, 5  
terminal, 152, 153  
tetration, 32  
Thompson, K  
    picture, 135  
Three Dimensional Matching problem, 316  
Three-dimensional Matching problem, 283  
time taken by a machine, 271  
token, 147, 368  
Tot, total functions, 120  
total function, 10, 120, 371  
Towers of Hanoi, 26  
tractable, 269–271  
trail, 164  
transformation function, *see* reduction function  
transition function, 8, 182, 308  
    extended, 186, 197  
    graph of, 7  
    Pushdown machine, 237  
    table of, 7  
transition graph, 7  
transition table, 7  
translation function, 319  
Traveling Salesman problem, 191, 278, 295,  
    325, 328, 332, 343, 357  
tree, 164, 282  
Triangle problem, 305  
triangular numbers, 25  
truth table, 281, 375  
Turing equivalent, 116  
Turing machine, 3–14  
    accept a language, 107  
    accepting a language, 306  
    accepting state, 13  
    action set, 8  
    action symbol, 8  
    computation, 9

configuration, 8  
control, 5  
CPU, 5  
current symbol, 8  
decidable, 293  
decides a set, 11  
deciding a language, 306  
definition, 8  
description number, 73  
deterministic, 8  
for addition, 6  
function computed, 9  
Gödel number, 73  
index number, 73  
input symbol, 8  
instruction, 5, 8  
language accepted, 293  
language decided, 13, 293  
language recognized, 293  
multitape, 21  
next state, 5, 8  
next symbol, 5, 8  
next-state function, 8  
nondeterminism, 307  
numbering, 73  
palindrome, 14  
present state, 5, 8  
present symbol, 5  
rejecting state, 13  
simulator, 37–41  
tape alphabet, 8  
transition function, 8  
universal, 84–86  
    with oracle, 114

Turing reducibility, 319  
Turing reducible, 114, 318  
Turing, A  
    picture, 3  
Turnpike problem, 317  
two-sided inverse, 373

unbounded minimization, 34  
unbounded search, 34  
uncountable, 77  
undecidable, 94  
Unicode, 183, 398  
uniformity, 86–87

Universal Turing machine, 84–86  
universality, 83–92  
unpairing function, 69, 70, 139  
unreachable state, 107  
unsolvability, 108  
unsolvable problem, 94, 108  
use-mention distinction, 125

value, of a function, 370  
verifier, 311  
    polysize, 311  
vertex, 163  
    reachable, 165, 281  
vertex cover, 282  
Vertex Cover problem, 282, 290, 325  
Vertex cover problem, 332  
Vertex-to-Vertex Path problem, 281, 300, 305,  
    320, 336  
von Neumann, J  
    picture, 46

walk, 164  
walk length, 164  
weight, 164  
weighted graph, 164  
well-defined, 370, 371  
wire, 302  
witness, 311  
word, *see* string  
working state, 12

XOR, 44

⊤, yields  
    Finite State machine, 186  
    for Turing machines, 9  
    nondeterministic Finite State machine,  
        194

Zeno’s Paradox, 65  
zero function, 24, 35  
Zoo, Complexity, 348